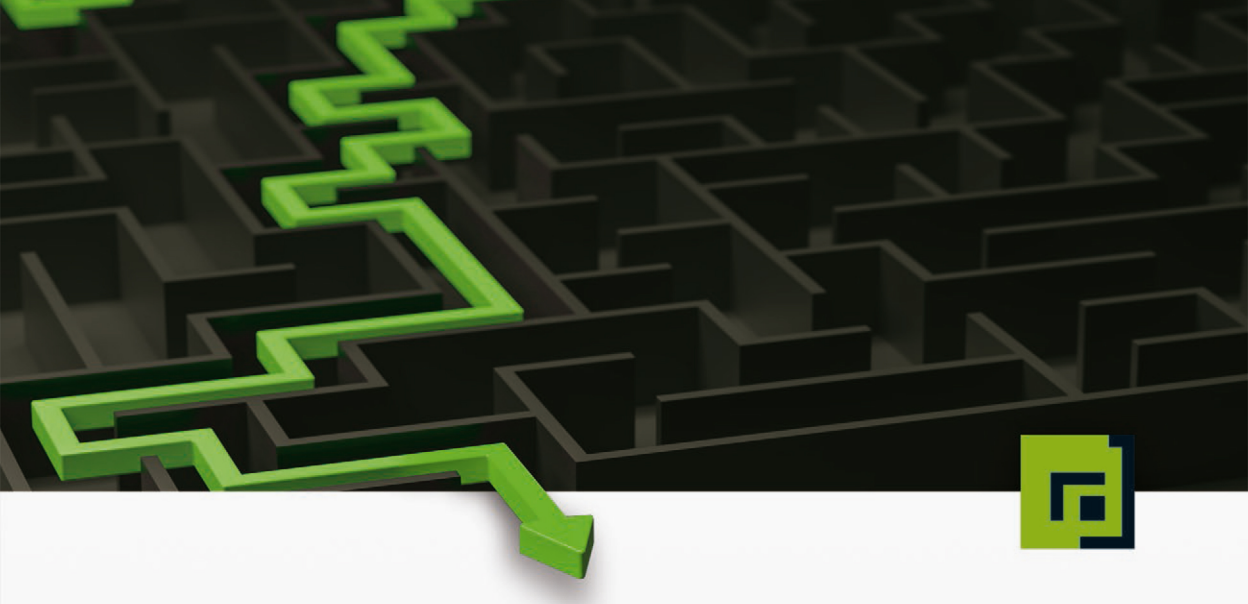


Python Challenge

Michael Inden

Fit für Prüfung, Job-Interview und Praxis
– mit 100 Aufgaben und Musterlösungen

dpunkt.verlag

Dipl.-Inform. Michael Inden ist Oracle-zertifizierter Java-Entwickler. Nach seinem Studium in Oldenburg hat er bei diversen internationalen Firmen in verschiedenen Rollen etwa als Softwareentwickler, -architekt, Consultant, Teamleiter sowie Trainer gearbeitet. Zurzeit ist er als CTO und Leiter Academy in Zürich tätig.

Michael Inden hat über zwanzig Jahre Berufserfahrung beim Entwurf komplexer Softwaresysteme gesammelt, an diversen Fortbildungen und mehreren Java-One-Konferenzen teilgenommen. Sein besonderes Interesse gilt dem Design qualitativ hochwertiger Applikationen mit ergonomischen GUIs sowie dem Coaching. Sein Wissen gibt er gerne als Trainer in internen und externen Schulungen und auf Konferenzen weiter, etwa bei der Java User Group Switzerland, bei der JAX/W-JAX, ch.open und den IT-Tagen.

Papier plus+ PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.plus

Michael Inden

Python Challenge

Fit für Prüfung, Job-Interview und Praxis – mit 100 Aufgaben und Musterlösungen

dpunkt.verlag

Michael Inden
michael_inden@hotmail.com

Lektorat: Michael Barabas
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Michael Inden
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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.

ISBN:
Print 978-3-86490-809-5
PDF 978-3-96910-140-7
ePub 978-3-96910-141-4
mobi 978-3-96910-142-1

1. Auflage 2021
Copyright © 2021 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

PEFC zertifiziert
Das Papier für dieses Buch stammt aus nachhaltig bewirtschafteten Wäldern und kontrollierten Quellen.
PEFC/04-31-0810 www.pefc.de

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Für unsere bezaubernde Prinzessin Sophie Jelena

Inhaltsverzeichnis

II Fortgeschrittenere und kniffligere Themen 301

Vorwort

Zunächst einmal bedanke ich mich bei Ihnen, dass Sie sich für dieses Buch entschieden haben. Hierin finden Sie eine Vielzahl an Übungsaufgaben zu den unterschiedlichsten Themengebieten, die kurzweilige Unterhaltung durch Lösen und Implementieren der Aufgaben bieten und Sie so entweder auf Bewerbungsgespräche einstimmen oder aber einfach Ihre Problemlösungsfähigkeiten verbessern.

Übung macht den Meister

Wir alle kennen das Sprichwort: »Übung macht den Meister.« Im Handwerk und in diversen Bereichen des realen Lebens wird viel geübt und der Ernstfall ist eher selten, etwa im Sport, bei Musikern und in anderen Bereichen. Merkwürdigerweise ist dies bei uns Softwareentwicklern oftmals deutlich anders. Wir entwickeln eigentlich fast die gesamte Zeit und widmen uns dem Üben und Lernen bzw. Einstudieren eher selten, teilweise gar nicht. Wie kommt das?

Vermutlich liegt das neben dem in der Regel vorherrschenden Zeitdruck auch daran, dass nicht so viel geeignetes Übungsmaterial zur Verfügung steht – es gibt zwar Lehrbücher zu Algorithmen sowie Bücher zu Coding, aber meistens sind diese entweder zu theoretisch oder zu Sourcecode-lastig und beinhalten zu wenig Erklärungen der Lösungswege. Das will dieses Buch ändern.

Wieso dieses Buch?

Wie kam ich dazu, dieses Buchprojekt in Angriff zu nehmen? Das hat mehrere Gründe. Zum einen wurde ich immer wieder per Mail oder persönlich von Teilnehmern meiner Workshops gefragt, ob es nicht ein Übungsbuch als Ergänzung zu meinem Buch »Der Weg zum Java-Profi« [4] geben würde. Dadurch kam die erste Idee auf.

Doch wirklich ausgelöst hat das Ganze dann, dass ein Google-Recruiter mit einer Jobanfrage recht überraschend auf mich zukam. Als Vorbereitung für die dann bevorstehenden Jobinterviews und zur Auffrischung meiner Kenntnisse machte ich mich auf die Suche nach geeigneter Lektüre und entwickelte selbst schon ein paar Übungsaufgaben. Dabei entdeckte ich das großartige, aber teilweise auch recht anspruchsvolle Buch »Cracking the coding interview« von Gayle Laakmann McDowell [6], das mich weiter inspirierte. Als Folge davon machte ich mich zunächst an ein auf Java ausgerichtetes Buchprojekt namens »Java Challenge«. Im Laufe der Zeit kam die Idee auf, etwas

Entsprechendes auch für Python umzusetzen. Somit basiert diese Python-Ausgabe auf der Java-Version, allerdings wurde das gesamte Buch überarbeitet und pythonifiziert. Dazu habe ich an einigen Stellen Dinge ergänzt, leicht abgewandelt oder teilweise entfernt. Insbesondere zeige ich, da wo es sinnvoll ist, wie man mit Python-Besonderheiten wie List Comprehensions u. Ä. Lösungen prägnanter gestalten kann.

An wen richtet sich dieses Buch?

Dieses Buch ist kein Buch für Programmierneulinge, sondern richtet sich an Leser, die bereits etwas Python-Know-how besitzen und dieses mithilfe von Übungen vertiefen wollen. Anhand kleiner Programmieraufgaben erweitern Sie auf unterhaltsame Weise Ihr Wissen rund um Python, Algorithmen und gutes Design.

Dieses Buch richtet sich im Speziellen an folgende Zielgruppen:

1. Schüler und Studierende – Zunächst sind dies Schüler mit Interesse an Informatik sowie Informatikstudierende, die Python als Sprache schon ganz passabel beherrschen und nun ihr Wissen anhand von Übungen vertiefen wollen.
2. Lehrer und Dozierende – Selbstverständlich können auch Lehrer und Dozierende von diesem Buch und seiner Vielzahl unterschiedlich schwieriger Aufgaben profitieren, entweder als Anregung für den eigenen Unterricht oder als Vorlage für Übungen oder Prüfungen.
3. Hobbyprogrammierer und Berufseinsteiger – Außerdem richtet sich das Buch an engagierte Hobbyprogrammierer, aber auch Berufseinsteiger, die gerne mit Python programmieren und sich weiterentwickeln wollen. Das Lösen der Aufgaben hilft darüber hinaus, für potenzielle Fragen in Jobinterviews gut vorbereitet zu sein.
4. Erfahrene Softwareentwickler und -architekten – Schließlich ist das Buch für erfahrene Softwareentwickler und -architekten bestimmt, die ihr Wissen ergänzen oder auffrischen wollen, um ihre Junior-Kollegen besser unterstützen zu können, und dafür ein paar Anregungen und frische Ideen suchen. Zudem lassen sich diverse Aufgaben auch in Jobinterviews verwenden, mit dem Komfort, die Musterlösungen direkt zum Vergleich parat zu haben. Aber auch für die alten Hasen sollte es zur Lösungsfindung und zu Algorithmen und Datenstrukturen das eine oder andere Aha-Erlebnis geben.

Generell verwende ich die maskuline Form, um den Text leichter lesbar zu halten. Natürlich beziehe ich damit alle Leserinnen mit ein und freue mich über diese ganz besonders.

Was vermittelt dieses Buch?

Dieses Buch enthält einen bunten Mix an Übungsaufgaben zu verschiedenen Themengebieten. Mitunter gibt es auch einige Knobelaufgaben, die zwar nicht direkt für die Praxis wichtig sind, aber indirekt doch, weil Sie Ihre Fähigkeiten zur Kreativität und zur Lösungsfindung verbessern.

Neben Übungsaufgaben und dokumentierten Lösungen war es mir wichtig, dass jeder im Buch behandelte Themenbereich mit einer kurzen Einführung startet, damit auch diejenigen Leser abgeholt werden, die in einigen Gebieten vielleicht noch nicht so viel Know-how aufgebaut haben. Damit können Sie sich dann an die Aufgaben bis etwa zum mittleren Schwierigkeitsgrad wagen. In jedem Themengebiet finden sich immer auch einige leichtere Aufgaben zum Einstieg. Mit etwas Übung sollten Sie sich dann an etwas schwierigere Probleme wagen. Mitunter gibt es herausfordernde Knacknüsse, an denen sich besser Experten versuchen oder solche, die es werden wollen.

Tipps und Hinweise aus der Praxis

Dieses Buch ist mit diversen Praxistipps gespickt. In diesen werden interessante Hintergrundinformationen präsentiert oder es wird auf Fallstricke hingewiesen.

Tipp:

In derart formatierten Kästen finden sich im späteren Verlauf des Buchs immer wieder einige wissenswerte Tipps und ergänzende Hinweise zum eigentlichen Text.

Schwierigkeitsgrad im Überblick

Für ein ausgewogenes, ansprechendes Übungsbuch bedarf es selbstverständlich einer Vielzahl an Aufgaben verschiedener Schwierigkeitsstufen, die Ihnen als Leser die Möglichkeit bieten, sich schrittweise zu steigern und Ihre Kenntnisse auszubauen. Dabei setze ich zwar ein gutes Python-Grundwissen voraus, allerdings erfordern die Lösungen niemals ganz tiefes Wissen über ein Themengebiet oder ganz besondere Sprachfeatures.

Damit der Schwierigkeitsgrad einfach und direkt ersichtlich ist, habe ich die von anderen Bereichen bekannte Sternekategorisierung genutzt, deren Bedeutung in diesem Kontext in nachfolgender Tabelle etwas genauer erläutert wird.

Sterne (Bedeutung)	Einschätzung	Zeitaufwand
★☆☆☆☆ (sehr leicht)	Die Aufgaben sollten ohne große Vorkenntnisse mit einfachem Python-Wissen in wenigen Minuten lösbar sein.	< 15 min
★★☆☆☆ (leicht)	Die Aufgaben erfordern ein wenig Nachdenken, sind aber dann direkt zu lösen.	< 30 min
★★★☆☆ (mittel)	Die Aufgaben sind mit etwas Nachdenken, ein wenig Strategie und manchmal durch die Betrachtung verschiedener Rahmenbedingungen gut schaffbar.	~ 30 – 45 min
★★★★☆ (schwierig)	Erprobte Problemlösungsstrategien, gutes Wissen zu Datenstrukturen und fundierte Python-Kenntnisse sind zur Lösung nötig.	~ 45 – 90 min
★★★★★ (sehr schwierig)	Die Aufgaben sind wirklich knifflig und schwierig zu lösen. Das sind erst dann Kandidaten, nachdem die anderen Aufgaben Ihnen keine größeren Schwierigkeiten mehr bereiten.	> 60 min

Dies sind jeweils nur Einschätzungen von meiner Seite und eher grobe Einordnungen. Bedenken Sie bitte, dass die von jedem Einzelnen wahrgenommene Schwierigkeit auch sehr von seinem Background und Wissensstand abhängt. Ich habe schon erlebt, dass sich Kollegen mit Aufgaben schwergetan haben, die ich als recht einfach empfand. Aber auch das Gegenteil kenne ich: Während andere eine Aufgabe anscheinend spielend lösen, ist man selbst am Verzweifeln, weil der Groschen einfach nicht fällt. Manchmal hilft eine Kaffeepause oder ein kleiner Spaziergang. Lassen Sie sich auf keinen Fall demotivieren – jeder hat irgendwann mit irgendeiner Art von Aufgabe zu kämpfen.

Hinweis: Mögliche Alternativen zu den Musterlösungen

Beachten Sie bitte, dass es für Problemstellungen nahezu immer einige Varianten gibt, die für Sie vielleicht sogar eingängiger sind. Deswegen werde ich ab und an als Denkanstoß interessante Alternativen zur (Muster-)Lösung präsentieren.

Aufbau dieses Buchs

Nachdem Sie eine grobe Vorstellung über den Inhalt dieses Buchs haben, möchte ich die Themen der einzelnen Kapitel kurz vorstellen.

Wie bereits angedeutet, sind die Übungsaufgaben thematisch gruppiert. Dabei bilden die fünf Kapitel nach der Einleitung die Grundlage und die darauffolgenden drei Kapitel behandeln fortgeschrittenere Themengebiete.

Kapitel 1 – Einleitung Dieses Kapitel beschreibt den Aufbau der folgenden Kapitel mit den Abschnitten Einführung, Aufgaben und Lösungen. Zudem wird ein Grundgerüst für die oftmals zur Prüfung der Lösungen genutzten Unit Tests vorgestellt. Abschließend gebe ich Hinweise zum Ausprobieren der Beispiele und Lösungen.

Kapitel 2 – Mathematische Aufgaben Das zweite Kapitel widmet sich mathematischen Operationen sowie Aufgaben zu Primzahlen und dem römischen Zahlensystem. Darüber hinaus präsentiere ich ein paar Ideen zu Zahlenspielereien.

Kapitel 3 – Rekursion Rekursion ist ein wichtiger Basisbaustein bei der Formulierung von Algorithmen. Dieses Kapitel gibt einen kurzen Einstieg und die diversen Übungsaufgaben sollten dabei helfen, Rekursion zu verstehen.

Kapitel 4 – Strings Strings sind bekanntermaßen Zeichenketten, die eine Vielzahl an Funktionen bieten. Ein solides Verständnis ist elementar wichtig, da nahezu kein Programm ohne Strings auskommt. Deswegen werden wir in diesem Kapitel die Verarbeitung von Zeichenketten anhand verschiedener Übungsaufgaben kennenlernen.

Kapitel 5 – Basisdatenstrukturen: Listen, Sets und Dictionaries Python bietet von Haus aus Listen, Mengen (Sets) und Schlüssel-Wert-Abbildungen (Dictionaries). Für den Programmieralltag ist ein sicherer Einsatz aller drei Container von großem Vorteil, was durch die Übungsaufgaben trainiert wird.

Kapitel 6 – Arrays Arrays bilden in vielen Programmiersprachen Grundbausteine. In Python sind Listen gebräuchlicher. Bezüglich Performance und Speicherverbrauch besitzen Arrays aber deutliche Vorteile. Grund genug, sich das Ganze in diesem Kapitel genauer anzuschauen.

Kapitel 7 – Rekursion Advanced Kapitel 3 hat das Thema Rekursion einleitend behandelt. Dieses Kapitel thematisiert fortgeschrittenere Aspekte. Wir starten mit der Optimierungstechnik namens Memoization. Im Anschluss schauen wir uns Backtracking als eine Problemlösungsstrategie an, die auf Versuch und Irrtum beruht und mögliche Lösungswege durchprobiert. Damit lassen sich diverse Algorithmen ziemlich verständlich und elegant formulieren.

Kapitel 8 – Bäume Baumstrukturen spielen in der Informatik sowohl in der Theorie als auch in der Praxis eine wichtige Rolle. In vielen Anwendungskontexten lassen sich Bäume gewinnbringend einsetzen, etwa für die Verwaltung eines Dateisystems, die Darstellung eines Projekts mit Teilprojekten und Aufgabenpaketen oder eines Buchs mit Kapiteln, Unterkapiteln und Abschnitten.

Kapitel 9 – Suchen und Sortieren Suchen und Sortieren sind zwei elementare Themen der Informatik. Die Python-Standardbibliothek setzt beide um und nimmt einem dadurch Arbeit ab. Jedoch lohnt sich auch ein Blick hinter die Kulissen, etwa auf verschiedene Sortierverfahren und deren spezifische Stärken und Schwächen.

Kapitel 10 – Schlusswort und ergänzende Literatur In diesem Kapitel fasse ich das Buch zusammen und gebe vor allem einen Ausblick auf ergänzende Literatur. Um Ihr Können zu erweitern, ist neben dem Programmiertraining auch das Studium von weiteren Büchern empfehlenswert. Eine Auswahl an hilfreichen Titeln bildet den Abschluss des Hauptteils dieses Buchs.

Anhang A – Kurzeinführung Pytest Zum Prüfen kleinerer Programmbausteine haben sich Unit Tests bewährt. Mit Pytest ist das Ganze insbesondere beim Formulieren von Testfällen für mehrere Eingabekombinationen ziemlich komfortabel. Weil viele der hier im Buch erstellten Lösungen mit Unit Tests geprüft werden, gibt dieser Anhang einen Einstieg in die Thematik.

Anhang B – Kurzeinführung Dekoratoren In diesem Anhang werden Dekoratoren beschrieben. Diese ermöglichen es, elegante Realisierungen von Querschnittsfunktionalitäten transparent vorzunehmen, also ohne Erweiterungen in der Implementierung einer Funktion selbst. Beispielsweise kann man sie für Parameterprüfungen verwenden, aber auch für Memoization, ein fortgeschrittenes Rekursionsthema.

Anhang C – Schnelleinstieg O-Notation In diesem Buch verwende ich manchmal zur Abschätzung des Laufzeitverhaltens und zur Einordnung der Komplexität von Algorithmen die sogenannte O-Notation. Dieser Anhang stellt Wesentliches dazu vor.

Konventionen und ausführbare Programme

Verwendete Zeichensätze

Im gesamten Text gelten folgende Konventionen bezüglich der Schriftart: Der normale Text erscheint in der vorliegenden Schriftart. Dabei werden wichtige Textpassagen *kursiv* oder ***kursiv und fett*** markiert. Englische Fachbegriffe werden eingedeutscht großgeschrieben. Zusammensetzungen aus englischen und deutschen (oder eingedeutschten) Begriffen werden mit Bindestrich verbunden, z. B. Plugin-Manager. Sourcecode-

Listings sind in der Schrift `courier` gesetzt, um zu verdeutlichen, dass dieser Text einen Ausschnitt aus einem Python-Programm wiedergibt. Auch im normalen Text werden Klassen, Methoden, Funktionen, Konstanten und Übergabeparameter in dieser Schriftart dargestellt.

Verwendete Abkürzungen

Im Buch verwende ich die in der nachfolgenden Tabelle aufgelisteten Abkürzungen. Weitere Abkürzungen werden im laufenden Text in Klammern nach ihrer ersten Definition aufgeführt und anschließend bei Bedarf genutzt.

Abkürzung	Bedeutung
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
(G)UI	(Graphical) User Interface
IDE	Integrated Development Environment

Verwendete Python-Version

Dieses Buch nutzt das aktuelle Python 3.8. Viele Lösungen müssten mit minimalen Anpassungen auch in Python 2.7 laufen. Das habe ich jedoch nur stichprobenartig geprüft. Generell ist es für neue Projekte sinnvoll, auf das modernere Python 3 zu setzen.

Download, Sourcecode und ausführbare Programme

Der Sourcecode der Beispiele steht auf der Webseite

`www.dpunkt.de/python-challenge`

zum Download bereit und ist in ein PyCharm-Projekt[1] integriert. Weil dies ein Buch zum Mitmachen ist, sind viele der Programme ausführbar – natürlich ist eine Ausführung in der IDE bzw. als Unit Test möglich.

Viele Codeschnipsel lassen sich aber auch hervorragend im Python-Kommandozeileninterpreter ausprobieren. Um das zu gewährleisten, sind mitunter bereits entwickelte Funktionen an geeigneter Stelle nochmals abgebildet.

[1] PyCharm ist eine kostenfrei unter `https://www.jetbrains.com/de-de/pycharm/` erhältliche, sehr empfehlenswerte IDE – genau genommen ist es eine auf Python ausgerichtete Variante von Intellij IDEA.

Danksagung

Ein Fachbuch zu schreiben ist eine schöne, aber arbeitsreiche und langwierige Aufgabe. Alleine kann man dies kaum bewältigen. Daher möchte ich mich an dieser Stelle bei allen bedanken, die direkt oder indirekt zum Gelingen des Buchs beigetragen haben. Insbesondere konnte ich bei der Erstellung des Manuskripts auf ein starkes Team an Korrekturlesern zurückgreifen. Es ist hilfreich, von den unterschiedlichen Sichtweisen und Erfahrungen profitieren zu dürfen.

Ein herzlicher Dank geht an Martin Stypinski für diverse nützliche Hinweise und Anregungen. Ebenfalls möchte ich Jean-Claude Brantschen für seine hilfreichen Vorschläge danken. Ihr habt mich noch stärker pythonifiziert :-) Auch verschiedene Kommentare von Rainer Grimm und Tobias Overkamp rund um Python und elegante Lösungen haben das Buch weiter verbessert. Schließlich hat Michael Kulla wie bei vielen meiner Bücher auch diese Python-Variante kritisch begutachtet. Vielen Dank an alle!

Da dieses Buch auf Basis der Java-Version entstanden ist, wird nachfolgend die Danksagung der Java Challenge wiederholt: Zunächst einmal möchte ich mich bei Michael Kulla, der als Trainer für Java SE und Java EE bekannt ist, für sein mehrmaliges, gründliches Review vieler Kapitel, die fundierten Anmerkungen und den tollen Einsatz ganz herzlich bedanken. Ebenfalls bin ich Prof. Dr. Dominik Gruntz für eine Vielzahl an Verbesserungsvorschlägen sehr dankbar. Zudem erhielt ich die eine oder andere hilfreiche Anregung von Jean-Claude Brantschen, Prof. Dr. Carsten Kern und Christian Heitzmann. Wieder einmal hat auch Ralph Willenborg ganz genau gelesen und so diverse Tippfehler gefunden. Vielen Dank dafür!

Ebenso geht ein Dankeschön an das Team des dpunkt.verlags (Dr. Michael Barabas, Martin Wohlrab, Anja Weimer und Birgit Bäuerlein) für die tolle Zusammenarbeit. Außerdem möchte ich mich bei Tobias Overkamp für die fundierte fachliche Durchsicht sowie bei Ursula Zimpfer für ihre Adleraugen beim Copy-Editing bedanken.

Abschließend geht ein lieber Dank an meine Frau Lilija für ihr Verständnis und ihre Unterstützung, vor allem auch für etliche Stupser, um auf das Fahrrad zu steigen und eine Runde zu drehen, anstatt nur am Buch zu arbeiten.

Anregungen und Kritik

Trotz großer Sorgfalt und mehrfachen Korrekturlesens lassen sich missverständliche Formulierungen oder sogar Fehler leider nicht vollständig ausschließen. Falls Ihnen etwas Derartiges auffallen sollte, so zögern Sie bitte nicht, mir dies mitzuteilen. Gerne nehme ich auch Anregungen oder Verbesserungsvorschläge entgegen. Kontaktieren Sie mich bitte per Mail unter:

`michael_inden@hotmail.com`

Zürich, im Dezember 2020
Michael Inden

1 Einleitung

Herzlich willkommen zu diesem Übungsbuch! Bevor Sie loslegen, möchte ich kurz darstellen, was Sie bei der Lektüre erwartet.

Dieses Buch behandelt verschiedene praxisrelevante Themengebiete und deckt diese durch Übungsaufgaben unterschiedlicher Schwierigkeitsstufen ab. Die Übungsaufgaben sind (größtenteils) voneinander unabhängig und können je nach Lust und Laune oder Interesse in beliebiger Reihenfolge gelöst werden. Neben den Aufgaben finden sich die jeweiligen Lösungen inklusive einer kurzen Beschreibung des zur Lösung verwendeten Algorithmus sowie dem eigentlichen, an wesentlichen Stellen kommentierten Sourcecode.

1.1 Aufbau der Kapitel

Jedes Kapitel ist strukturell gleich aufgebaut, sodass Sie sich schnell zurechtfinden werden.

Einführung

Ein Kapitel beginnt jeweils mit einer Einführung in die jeweilige Thematik, um auch diejenigen Leser abzuholen, die mit dem Themengebiet vielleicht noch nicht so vertraut sind, oder aber, um Sie auf die nachfolgenden Aufgaben entsprechend einzustimmen.

Aufgaben

Danach schließt sich ein Block mit Übungsaufgaben und folgender Struktur an:

Aufgabenstellung Jede einzelne Übungsaufgabe besitzt zunächst eine Aufgabenstellung. Dort werden in wenigen Sätzen die zu realisierenden Funktionalitäten beschrieben. Oftmals wird auch schon eine mögliche Signatur als Anhaltspunkt zur Lösung angegeben.

Beispiele Ergänzend finden sich fast immer Beispiele zur Verdeutlichung mit Eingaben und erwarteten Ergebnissen. Nur für einige recht einfache Aufgaben, die vor allem zum Kennenlernen eines APIs dienen, wird mitunter auf Beispiele verzichtet.

Oftmals werden in einer Tabelle verschiedene Wertebelegungen von Eingabeparameter(n) sowie das erwartete Ergebnis dargestellt, etwa wie folgt:

Eingabe A	Eingabe B	Ergebnis
[1, 2, 4, 7, 8]	[2, 3, 7, 9]	[2, 7]

Für die Angaben gelten folgende Notationsformen:

- "AB" – steht für textuelle Angaben
- True / False – repräsentieren boolesche Werte
- 123 – Zahlenangaben
- [value1, value2, ...] – steht für Sets oder Listen
- { key1 : value1, key2 : value2, ... } – beschreibt Dictionaries

Lösungen

Auch der Teil der Lösungen besitzt die nachfolgend beschriebene Struktur.

Aufgabenstellung und Beispiele Zunächst finden wir nochmals die Aufgabenstellung, sodass wir nicht ständig zwischen Aufgaben und Lösungen hin- und herblättern müssen, sondern das Ganze in sich abgeschlossen ist.

Algorithmus Danach folgt eine Beschreibung des gewählten Algorithmus zur Lösung. Aus Gründen der Didaktik zeige ich bewusst auch einmal einen Irrweg oder eine nicht so optimale Lösung, um daran dann Fallstricke aufzudecken und iterativ zu einer Verbesserung zu kommen. Tatsächlich ist die eine oder andere Brute-Force-Lösung manchmal sogar schon brauchbar, bietet aber Optimierungspotenziale. Exemplarisch werde ich immer wieder entsprechende, manchmal verblüffend einfache, aber oft auch sehr wirksame Verbesserungen vorstellen.

Python-Shortcut Mitunter werden in der Aufgabenstellung gewisse Python-Standardfunktionalitäten explizit zur Realisierung der Lösung ausgeschlossen, um ein Problem algorithmisch zu durchdringen. In der Praxis sollten Sie aber die Standards nutzen. In diesem eigenen kurzen Abschnitt »Python-Shortcut« zeige ich, wie man damit die Lösung oftmals kürzer und prägnanter gestalten kann.

Prüfung Teilweise sind die Aufgaben recht leicht oder dienen nur dem Kennenlernen von Syntax oder API-Funktionalität. Dafür scheint es mir oftmals ausreichend, ein paar Aufrufe direkt mit dem Python-Kommandozeileninterpreter auszuführen. Deshalb verzichte ich hierfür auf Unit Tests. Gleiches gilt auch, wenn wir bevorzugt eine grafische Aufbereitung einer Lösung, etwa die Darstellung eines Sudoku-Spielfelds, zur Kontrolle nutzen und der korrespondierende Unit Test vermutlich schwieriger verständlich wäre.

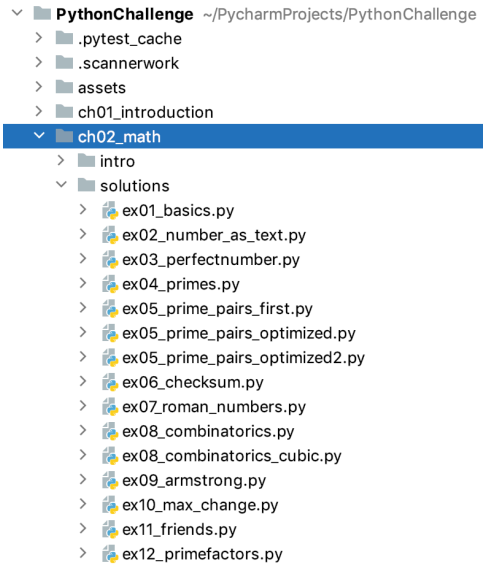
Je komplizierter allerdings die Algorithmen werden, desto mehr lauern auch Fehlerquellen, wie falsche Indexwerte, eine versehentliche oder unterbliebene Negation oder ein übersehener Randfall. Deswegen bietet es sich an, Funktionalitäten mithilfe von Unit Tests zu überprüfen – in diesem Buch kann das aus Platzgründen natürlich nur exemplarisch für wichtige Eingaben geschehen. Insgesamt existieren jedoch rund 80 Unit-Test-Module mit über 600 Testfällen. Ein ziemlich guter Anfang. Trotzdem sollte in der Praxis das Netz an Unit Tests und Testfällen wenn möglich noch umfangreicher sein.

1.2 Grundgerüst des PyCharm-Projekts

Auch das mitgelieferte PyCharm-Projekt orientiert sich in seinem Aufbau an demjenigen des Buchs und bietet für die Kapitel mit Übungsaufgaben jeweils ein eigenes Verzeichnis pro Kapitel, z. B. `ch02_math` oder `ch07_recursion_advanced`.

Einige der Sourcecode-Schnipsel aus den jeweiligen Einführungen finden sich in einem Unterverzeichnis `intro`. Die bereitgestellten (Muster-)Lösungen werden in jeweils eigenen Unterverzeichnissen namens `solutions` gesammelt und die Module sind gemäß Aufgabenstellung wie folgt benannt: `ex<Nr>_<Aufgabenstellung>.py`.

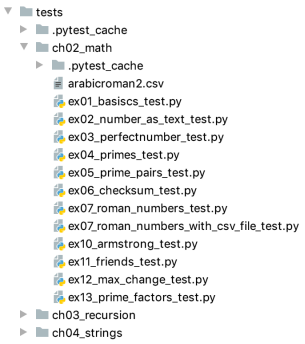
Sourcen Nachfolgend ist ein Ausschnitt für das Kapitel 2 gezeigt:



Utility-Klassen Alle in den jeweiligen Kapiteln entwickelten nützlichen Utility-Funktionen sind im bereitgestellten PyCharm-Projekt in Form von Utility-Modulen enthalten. Diese kombinieren wir dann in einem Modul `xyz_utils`, das in einem eigenen

Unterverzeichnis `util` liegt – für das Kapitel zu mathematische Aufgabenstellungen im Unterverzeichnis `ch02_math.util`. Gleiches gilt für die anderen Kapitel und Themengebiete.

Test-Klassen Exemplarisch sind nachfolgend einige Tests zu Kapitel 2 gezeigt:



1.3 Grundgerüst für die Unit Tests mit PyTest

Um den Rahmen des Buchs nicht zu sprengen, zeigen die abgebildeten Unit Tests jeweils nur die Testfunktionen, jedoch oftmals nicht die Imports. Damit Sie ein Grundgerüst haben, in das Sie die Testfunktionen einfügen können, sowie als Ausgangspunkt für eigene Experimente ist nachfolgend ein typisches Modul gezeigt:

```
import pytest

from ch02_math.solutions import ex01_basics
from ch02_math.solutions.ex01_basics import calc, \
    calc_sum_and_count_all_numbers_div_by_2_or_7_v2


@pytest.mark.parametrize("m, n, expected",
                         [(6, 7, 0), (3, 4, 6), (5, 5, 5)])
def test_calc(m, n, expected):
    assert calc(m, n) == expected


@pytest.mark.parametrize("n, expected",
                         [(3, {"sum": 2, "count": 1}),
                          (8, {"sum": 19, "count": 4}),
                          (15, {"sum": 63, "count": 8})])
def test_calc_sum_and_count_all_numbers_div_by_2_or_7(n, expected):
    assert calc_sum_and_count_all_numbers_div_by_2_or_7_v2(n) == expected
```

Neben den Imports sehen wir die ausgiebig genutzten parametrisierten Tests, die das Prüfen mehrerer Wertkombinationen auf einfache Weise erlauben. Für Details und eine Kurzeinführung in Pytest schauen Sie bitte in Anhang A.

1.4 Anmerkung zum Programmierstil

In diesem Abschnitt möchte ich noch vorab etwas zum Programmierstil sagen, weil in Diskussionen ab und an einmal die Frage aufkam, ob man gewisse Dinge nicht kompakter gestalten sollte.

Gedanken zur Sourcecode-Kompaktheit

In der Regel sind mir beim Programmieren und insbesondere für die Implementierungen in diesem Buch vor allem eine leichte Nachvollziehbarkeit sowie eine übersichtliche Strukturierung und damit später eine vereinfachte Wartbarkeit und Veränderbarkeit wichtig. Deshalb sind die gezeigten Implementierungen möglichst verständlich programmiert und dadurch ist vielleicht nicht jedes Konstrukt maximal kompakt. Dem Aspekt der guten Verständlichkeit möchte ich in diesem Buch den Vorrang geben. Auch in der Praxis kann man damit oftmals besser leben als mit einer schlechten Wartbarkeit, dafür aber einer kompakteren Programmierung.

Beispiel 1

Schauen wir uns zur Verdeutlichung ein kleines Beispiel an. Zunächst betrachten wir die lesbare, gut verständliche Variante zum Umdrehen des Inhalts eines Strings, die zudem sehr schön die beiden wichtigen Elemente des rekursiven Abbruchs und Abstiegs verdeutlicht:

```
def reverse_string(input):
    # rekursiver Abbruch
    if len(input) <= 1:
        return input

    first_char = input[0]
    remaining = input[1:]

    # rekursiver Abstieg
    return reverse_string(remaining) + first_char
```

Die folgende deutlich kompaktere Variante bietet diese Vorteile nicht:

```
def reverse_string_short(input):
    return input if len(input) <= 1 else \
        reverse_string_short(input[1:]) + input[0]
```

Überlegen Sie kurz, in welcher der beiden Funktionen Sie sich sicher fühlen, eine nachträgliche Änderung vorzunehmen. Und wie sieht es aus, wenn Sie noch Unit Tests ergänzen wollen: Wie finden Sie passende Wertebelegungen und Prüfungen?

Beispiel 2

Lassen Sie mich noch ein weiteres Beispiel anbringen, um meine Aussage zu verdeutlichen. Nehmen wir folgende der Standardfunktion `count()` nachempfundene Funktion `count_substrings()`, die die Anzahl der Vorkommen eines Strings in einem anderen zählt und für die beiden Eingaben "halloha" und "ha" das Ergebnis 2 liefert.

Zunächst implementieren wir das einigermaßen geradeheraus wie folgt:

```
def count_substrings(input, value_to_find):
    # rekursiver Abbruch
    if len(input) < len(value_to_find):
        return 0

    count = 0
    remaining = ""

    # startet der Text mit der Suchzeichenfolge?
    if input.startswith(value_to_find):
        # Treffer: Setze die Suche nach dem gefundenen
        # Begriff nach der Fundstelle fort
        remaining = input[len(value_to_find):]
        count = 1
    else:
        # entferne erstes Zeichen und suche erneut
        remaining = input[1:]

    # rekursiver Abstieg
    return count_substrings(remaining, value_to_find) + count
```

Schauen wir uns an, wie man dies kompakt zu realisieren versuchen könnte:

```
def count_substrings_short(input, value_to_find):
    return 0 if len(input) < len(value_to_find) else \
        (1 if input.startswith(value_to_find) else 0) + \
        count_substrings_short(input[1:], value_to_find)
```

Würden Sie lieber in dieser Funktion oder in der zuvor gezeigten ändern?

Übrigens: Die untere enthält noch eine subtile funktionale Abweichung! Bei den Eingaben von "XXXX" und "XX" »konsumiert« die erste Variante immer die Zeichen und findet zwei Vorkommen. Die untere bewegt sich aber jeweils nur um ein Zeichen weiter und findet somit drei Vorkommen.

Und weiter: Die Integration der oben realisierten Funktionalität des Weiterschiebens um den gesamten Suchstring in die zweite Variante wird zu immer undurchsichtigerem Sourcecode führen. Dagegen kann man oben das Weiterschieben um nur ein Zeichen einfach umsetzen und diese Funktionalität dann sogar aus dem `if` herausziehen.

Dekoratoren und Sanity Checks am Funktionsanfang

Um für stabile Programme zu sorgen, ist es oftmals eine gute Idee, die Parameter zentraler Funktionen auf Gültigkeit zu prüfen. Das kann man in Form einfacher `if`-Abfragen realisieren. In Python geht das aber eleganter mithilfe von Dekoratoren. Werfen Sie zum Einstieg doch bitte einen Blick in Anhang B.

Blockkommentare in Listings

Beachten Sie bitte, dass sich in den Listings diverse Blockkommentare finden, die der Orientierung und dem besseren Verständnis dienen. In der Praxis sollte man derartige Kommentierungen mit Bedacht einsetzen und lieber einzelne Sourcecode-Abschnitte in Funktionen auslagern. Für die Beispiele des Buchs dienen diese Kommentare aber als Anhaltspunkte, weil die eingeführten oder dargestellten Sachverhalte für Sie als Leser vermutlich noch neu und ungewohnt sind.

```
# startet der Text mit der Suchzeichenfolge?
if input.startswith(value_to_find):
    # Treffer: Setze die Suche nach dem gefundenen
    # Begriff nach der Fundstelle fort
    remaining = input[len(value_to_find):]
    count = 1
else:
    # entferne erstes Zeichen und suche erneut
    remaining = input[1:]
```

PEP 8 und Zen of Python

Neben meinen bereits präsentierten Gedanken zum Programmierstil möchte ich noch zwei Dinge explizit erwähnen:

- PEP 8 – Coding-Standard (PEP = Python Enhancement Proposal)
- Zen of Python – Gedanken zu Python

PEP 8 – Coding-Standard Der offizielle Coding-Standard ist als PEP 8 unter `https://www.python.org/dev/peps/pep-0008/` online verfügbar. Dieser soll dabei helfen, sauberen, einheitlichen und verständlichen Python-Code zu schreiben. Tendenziell legt man in der Python-Community mehr Wert auf schönen Sourcecode, als dies in anderen Sprachen der Fall ist. Generell ist aber »Hauptsache es funktioniert« keine nachhaltige Strategie, wie ich es auch bereits motiviert habe.

Allerdings gibt es ein paar wenige Dinge, über die man auch geteilter Meinung sein kann, etwa die Begrenzung der Zeilenlänge auf 79 Zeichen. Bei heutigen HiDPI-Monitoren und Auflösungen jenseits von Full-HD sind sicher auch längere Zeilen von rund 120 Zeichen möglich. Aber allzu lang sollte eine Zeile auch nicht werden – vor allem, wenn man einmal zwei Versionsstände einer Datei miteinander vergleichen möchte, kann dies sonst störend sein.

Zen of Python Interessanterweise ist in den Kommandozeileninterpreter eine Ausgabe von Stilhinweisen, auch als Zen of Python bekannt, eingebaut. Diese erhält man durch einen Aufruf von:

```
>>> import this
```

Es kommt zu folgender Ausgabe:

```
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Tooling Wie schon erwähnt, bietet sich PyCharm als IDE an und gibt direkt im Editor verschiedene Hinweise zum Stil und zu Verbesserungsmöglichkeiten. Eine Konfiguration kann man unter `Preferences-> Editor-> Code Style -> Python` sowie `Preferences-> Editor-> Inspections-> Python` vornehmen. Insbesondere gibt es bei letzterer die Möglichkeit, `PEP8 coding style violation` zu aktivieren.

Alternativ oder ergänzend können Sie das Tool `flake8` wie folgt installieren:

```
pip3 install flake8
```

Dieses hilft dabei, verschiedene potenzielle Probleme und Verstöße gegen PEP 8 aufzudecken, wenn Sie es wie folgt aufrufen:

```
flake8 <mypythonmodule>.py mydirwithmodules ...
```

Es gibt noch weitere Tools. Empfehlenswert für größere Projekte, wenn auch etwas aufwendiger, ist es, eine statische Sourcecode-Analyse mittels Sonar auszuführen. Dazu ist allerdings Sonar und auch ein Sonar Runner zu installieren. Dafür erhält man dann aber eine schöne Übersicht sowie eine Historisierung, sodass man sowohl positive als auch negative Trends schnell erkennen und bei Bedarf gegensteuern kann.

Weitere Informationen Weitere Informationen, wie Sie sauberes Python schreiben, finden Sie in folgenden Büchern:

- »Python-Tricks – Praktische Tipps für Fortgeschrittene« von Dan Bader [2]
- »Mastering Python« von Rick van Hattern [14]

1.5 Anmerkung zu den Aufgaben

Bei der Lösung der Aufgaben ist es das Ziel, sich mit den dazu notwendigen Algorithmen und Datenstrukturen zu befassen. Python bietet eine recht umfangreiche Sammlung an Funktionalitäten, etwa zur Ermittlung von Summen und Minimum von Listen oder gar komplexeren Dingen wie der Berechnung von Permutationen.

Einige der Aufgaben lassen sich mit den vorgefertigten Standardfunktionalitäten in wenigen Zeilen lösen. Das ist jedoch nicht das Ziel, denn die Übungsaufgaben dienen dem algorithmischen Verständnis und der Erweiterung Ihrer Problemlösungsstrategien. Wenn man dies selbst ergründet und löst, lernt man viel dabei. Dinge selbst zu entwickeln, ist nur für das Training gedacht, nicht für den Praxiseinsatz: Bedenken Sie bitte, dass in realen Projekten der Standardfunktionalität von Python immer der Vorzug gegeben werden sollte und Sie nicht im Traum daran denken sollten, etwas selbst zu erfinden, wofür es schon eine vorgefertigte Lösung gibt. Deswegen weise ich oftmals in einem eigenen kurzen Abschnitt »Python-Shortcut« auf eine Lösung hin, die Python-Standardfunktionalität verwendet.

1.6 Ausprobieren der Beispiele und Lösungen

Grundsätzlich verwende ich möglichst nachvollziehbare Konstrukte und keine ganz besonders ausgefallenen Syntax- oder API-Features. Vielfach können Sie die abgebildeten Sourcecode-Schnipsel einfach in den Python-Kommandozeileninterpreter kopieren und ausführen. Alternativ finden Sie alle relevanten Sourcen in dem zum Buch mitgelieferten PyCharm-Projekt. Dort lassen sich die Programme durch eine `main()`-Funktion starten oder durch oftmals vorhandene korrespondierende Unit Tests überprüfen.

Los geht's: Entdeckungsreise Python Challenge

So, nun ist es genug der Vorrede und Sie sind bestimmt schon auf die ersten Herausforderungen durch die Übungsaufgaben gespannt. Deshalb wünsche ich Ihnen nun viel Freude mit diesem Buch sowie einige neue Erkenntnisse beim Lösen der Übungsaufgaben und beim Experimentieren mit den Algorithmen.

Wenn Sie zunächst eine Auffrischung Ihres Wissens zu Unit Tests, zum Python-Kommandozeileninterpreter oder der O-Notation benötigen, bietet sich ein Blick in die Anhänge an.

I Grundlagen

2 Mathematische Aufgaben

In diesem Kapitel lernen wir zunächst Grundlegendes zu einigen mathematischen Operationen und zu Primzahlen, aber auch zum römischen Zahlensystem kennen. Darüber hinaus präsentiere ich ein paar Ideen zu Zahlenspielereien. Mit diesem Wissen sollten Sie für die Vielzahl an Übungsaufgaben gut gewappnet sein.

2.1 Einführung

Kurzeinführung Division und Modulo

Neben Multiplikation und Division wird auch die Modulo-Operation (%) recht häufig verwendet. Sie dient dazu, den Rest einer Division zu ermitteln. Veranschaulichen wir uns dies wie folgt für Ganzzahlen, bei denen Divisionsreste mit / beachtet werden und mit // unter den Tisch fallen:

$$(\mathbf{5} * 7 + 3) \,//\, 7 = 38 \,//\, 7 = 5$$
$$(5 * 7 + \mathbf{3}) \,\%\, 7 = 38 \,\%\, 7 = 3$$

Selbst mit diesen wenigen Operationen lassen sich diverse Aufgabenstellungen lösen. Wir rufen uns folgende Dinge für Aktionen auf Zahlen in Erinnerung:

- $n \,\%\, 10$ – Ermittelt den Rest einer Division durch 10 und somit die letzte Ziffer.
- $n \,/\, 10$ – Teilt durch den Wert 10. Seit Python 3 liefert das eine Fließkommazahl als Ergebnis. Benötigt man eine Ganzzahl, so kann man eine Typumwandlung mit int(), etwa int(value / 10), nutzen.
- $n \,//\, 10$ – Teilt ebenfalls durch den Wert 10. Weil der Operator $//$ eine Ganzzahldivision ohne Rest ausführt, ist es damit möglich, die letzte Ziffer abzuschneiden.

Extraktion von Ziffern Zur Extraktion der Ziffern einer Zahl kombinieren wir Modulo und Division so lange, wie der verbleibende Wert größer als 0 ist:

```
def extract_digits(number):
    remaining_value = number
    while remaining_value > 0:
        digit = remaining_value % 10
        remaining_value = remaining_value // 10
        print(digit, end=' ')
    print()
```

In Python gibt es noch eine Besonderheit mit der Built-in-Funktion `divmod()`, die als Ergebnis sowohl den Teiler als auch den Rest liefert – als Abkürzung für die häufig in Kombination aufgerufenen Operatoren. Zudem nutzen wir das Tuple Unpacking nachfolgend aus, wodurch das Ergebnis der jeweiligen Variablen zugewiesen wird:

```
def extract_digits(number):
    remaining_value = number
    while remaining_value > 0:
        remaining_value, digit = divmod(remaining_value, 10)
        print(digit, end=' ')
    print()
```

Wir rufen diese Funktion auf, um deren Arbeitsweise zu verstehen – bitte beachten Sie, dass die Ziffern in umgekehrter Reihenfolge ausgegeben werden und Leerzeilen bei der Verarbeitung im Python-Kommandozeileninterpreter mitunter Probleme machen: In IDEs wie PyCharm ist dies dagegen problemlos möglich – ich werde Leerzeilen für die Beispiele nutzen, sofern dies klareren Sourcecode gibt.

```
>>> extract_digits(123)
3 2 1
```

Anzahl Ziffern ermitteln Statt einzelne Ziffern zu extrahieren, kann man mithilfe einer wiederholten Division auch die Anzahl der Ziffern einer Dezimalzahl ermitteln, indem man einfach so lange durch 10 teilt, bis kein Rest mehr übrigbleibt:

```
def count_digits(number):
    count = 0
    remaining_value = number
    while remaining_value > 0:
        remaining_value = remaining_value // 10
        count += 1
    return count
```

Kurzeinführung Teiler

Nachfolgend schauen wir uns an, wie man alle echten Teiler einer Zahl, also diejenigen ohne die Zahl selbst, ermitteln kann. Der Algorithmus ist recht einfach: Wir durchlaufen alle Zahlen bis zur Hälfte des Werts (alle höheren Werte können keine ganzzahligen Teiler sein, weil ja die 2 bereits ein Teiler ist) und prüfen, ob diese die gegebene Zahl ohne Rest teilen. Ist das der Fall, so ist diese Zahl ein Teiler und wird in eine Ergebnisliste aufgenommen. Das Ganze implementieren wir wie folgt:

```
def find_proper_divisors(value):
    divisors = []
    for i in range(1, value // 2 + 1):
        if value % i == 0:
            divisors.append(i)
    return divisors
```

Noch eine kleine Anmerkung zur Namensgebung: Für Schleifenvariablen sind kurze Namen wie `i` gebräuchlich, aber `current_number` wäre auch eine lesbare Alternative.

Mithilfe von List Comprehension[1] schreiben wir die Berechnung kürzer:

```
def find_proper_divisors(value):
    return [i for i in range(1, value // 2 + 1) if value % i == 0]
```

Rufen wir diese Funktion einmal auf, um deren Arbeitsweise zu verstehen und durch die erwartungskonforme Ausgabe zu bestätigen:

```
>>> find_proper_divisors(6)
[1, 2, 3]
>>> find_proper_divisors(24)
[1, 2, 3, 4, 6, 8, 12]
```

Kurzeinführung Primzahlen

Eine Primzahl ist eine natürliche Zahl, die größer als 1 und ausschließlich durch sich selbst und durch 1 teilbar ist. Es gibt zwei recht einfach zu verstehende Algorithmen, um zu prüfen, ob eine gegebene Zahl eine Primzahl ist, bzw. um Primzahlen bis zu einem vorgegebenen Maximalwert zu berechnen.

Brute-Force-Algorithmus für Primzahlen Ob eine Zahl eine Primzahl ist oder nicht, lässt sich wie folgt bestimmen: Man schaut für die zu prüfende Zahl ausgehend von der 2 bis maximal zur Hälfte der Zahl, ob die momentane Zahl ein Teiler der ursprünglichen Zahl ist.[2] In dem Fall ist es keine Primzahl, ansonsten muss weiter geprüft werden. In Python lässt sich das folgendermaßen formulieren:

```
def is_prime(potentially_prime):
    for i in range(2, potentially_prime // 2 + 1):
        if potentially_prime % i == 0:
            return False
    return True
```

Probieren wir die Berechnung einmal aus:

```
>>> primes = []
>>> for number in range(2, 25):
...     if is_prime(number):
...         primes.append(number)
... print(primes)
```

Mithilfe von List Comprehension schreiben wir dies kürzer:

```
>>> primes = [number for number in range(2, 25) if is_prime(number)]
... print(primes)
```

In beiden Fällen erhalten wir dann als korrektes Ergebnis die Primzahlen kleiner 25:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

[1] Als List Comprehension bezeichnet man einen Ausdruck, der basierend auf einer Sequenz von Werten sowie einer Berechnungsvorschrift eine neue Ergebnisliste erzeugt.

[2] Optimiert muss man tatsächlich nur bis zur Wurzel berechnen. Darauf gehe ich kurz im nachfolgenden Praxistipp »Mögliche Optimierungen« ein.

Optimierung: Sieb des Eratosthenes Der Algorithmus zur Bestimmung der Primzahlen bis zu einem vorgegebenen Maximalwert nennt sich das ***Sieb des Eratosthenes*** und geht auf den gleichnamigen griechischen Mathematiker zurück.

Das Ganze funktioniert wie folgt: Initial werden alle Zahlen startend bei zwei bis zu dem vorgegebenen Maximalwert aufgeschrieben, etwa:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$$

Alle Zahlen gelten zunächst als potenzielle Kandidaten für Primzahlen. Nun werden schrittweise diejenigen Zahlen gestrichen, die keine Primzahlen sein können. Man nimmt die kleinste unmarkierte Zahl, hier zunächst die 2. Diese entspricht der ersten Primzahl. Nun streicht man alle Vielfachen davon, also im Beispiel $4, 6, 8, 10, 12, 14$:

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15$$

Weiter geht es mit der Zahl 3. Das ist die zweite Primzahl. Nun werden wieder die Vielfachen gestrichen, nachfolgend $6, 9, 12, 15$:

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}$$

Die nächste unmarkierte Zahl und somit eine Primzahl ist die 5. Das Verfahren wiederholt man so lange, wie es noch nicht durchgestrichene Zahlen nach der aktuellen Primzahl gibt:

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}$$

Damit verbleibt dann als Ergebnis für alle Primzahlen kleiner 15:

$$2, 3, 5, 7, 11, 13$$

In Aufgabe 4 sollen Sie das Sieb des Eratosthenes selbst implementieren. Dann können Sie Ihren Algorithmus ergänzend zu den obigen mit folgenden Werten prüfen:

Grenzwert	Resultat
25	[2, 3, 5, 7, 11, 13, 17, 19, 23]
50	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Tipp: Mögliche Optimierungen

Wie man sieht, werden oftmals Zahlen mehrfach durchgestrichen. Wenn man mathematisch etwas bewanderter ist, lässt sich zeigen, dass mindestens ein Primfaktor einer zusammengesetzten Zahl immer kleiner gleich der Wurzel der Zahl selbst sein muss. Der Grund ist, dass wenn x ein Teiler größer als $\sqrt{n}$ ist, dann gilt, dass $p = n/x$ kleiner als $\sqrt{n}$ ist und somit dieser Wert bereits ausprobiert worden ist. Dadurch kann man das Streichen der Vielfachen optimieren: Zum einen kann man das Streichen mit dem Quadrat der Primzahl beginnen, weil alle kleineren Vielfachen bereits gestrichen sind. Zum anderen muss die Berechnung nur bis zur Wurzel der oberen Grenze erfolgen. Mehr Details liefert `https://de.wikipedia.org/wiki/Sieb_des_Eratosthenes`.

2.1.1 Römische Zahlen

Das römische Zahlensystem arbeitet mit speziellen Buchstaben und Kombinationen daraus, um Zahlen zu repräsentieren. Dabei gilt folgende Grundabbildung:[3]

Römische Ziffer	I	V	X	L	C	D	M
Wert	1	5	10	50	100	500	1000

Der jeweilige Wert ergibt sich normalerweise aus der Addition der Werte der einzelnen Ziffern von links nach rechts, wobei normalerweise (siehe die nachfolgenden Regeln) links die größte und rechts die kleinste Zahl steht, beispielsweise XVI für den Wert 16.

Hinweis: Größere Zahlen

Für die Darstellung größerer römischer Zahlen (im Bereich von zehntausend und mehr) gibt es spezielle Schreibweisen, weil keine vier oder mehr M aufeinander folgen dürfen. Dies ist für die Aufgaben dieses Buchs nicht von Relevanz und kann vom Leser bei Interesse im Internet oder anderen Quellen nachgeschaut werden.

Regeln

Die römischen Zahlen werden nach bestimmten Regeln zusammengesetzt:

1. **Additionsregel**: Gleiche Ziffern nebeneinander werden addiert, etwa XXX = 30 Ebenso gilt dies für kleinere Ziffern nach größeren, z. B. XII = 12.
2. **Wiederholungsregel**: Es dürfen maximal drei gleiche Ziffern aufeinanderfolgen. Nach Regel 1 könnte man die Zahl 4 als IIII schreiben, was diese Regel 2 verbietet. Hier kommt die Subtraktionsregel ins Spiel.
3. **Subtraktionsregel**: Steht ein kleineres Zahlzeichen vor einem größeren, so wird der entsprechende Wert subtrahiert. Schauen wir nochmals auf die 4: Diese kann man als Subtraktion $5 - 1$ realisieren. Das wird im römischen Zahlensystem als IV notiert. Für die Subtraktion gelten folgende Regeln:
   - I steht nur vor V und X
   - X steht nur vor L und C
   - C steht nur vor D und M

[3] Interessanterweise gibt es bei den römischen Zahlen den Wert 0 nicht.

Beispiele

Zum besseren Verständnis und zur Verdeutlichung der obigen Regeln schauen wir uns einige Notationen römischer Zahlen und die korrespondierenden Werte an:

$$\begin{aligned} VII &= 5+1+1 &&= 7 \\ MDCLXVI &= 1000+500+100+50+10+5+1 &&= 1666 \\ MMXVIII &= 1000+1000+10+5+1+1+1 &&= 2018 \\ MMXIX &= 1000+1000+10-1+10 &&= 2019 \end{aligned}$$

Bemerkenswertes

Die bei uns heute verbreiteten arabischen Zahlen nutzen das Zehnersystem, bei dem die Position der Ziffern über deren Wert entscheidet: Somit kann die 7 einmal die Zahl selbst sein, aber auch für 70 oder 700 stehen. Im römischen Zahlensystem steht die V aber immer für eine 5, unabhängig von der Position.

Aufgrund dieses besonderen Aufbaus der römischen Zahlen sind viele mathematische Operationen aufwendig, selbst eine einfache Addition kann schon eine stärkere oder manchmal gar eine komplette Veränderung der Zahl bewirken: Das sieht man sehr schön für die Zahlen 2018 und 2019 oder für die Addition III + II = V. Schlimmer noch: Deutlich komplexer ist eine Multiplikation oder Division – es gibt Vermutungen, dass dies einer der Faktoren war, warum das römische Weltreich untergegangen ist.

2.1.2 Zahlenspielereien

Nachfolgend schauen wir uns ein paar spezielle Zahlenkonstellationen an:

- Vollkommene oder perfekte Zahlen
- Armstrong-Zahlen
- Prüfsummen

Bei vielen der nachfolgend genutzten Algorithmen untergliedert man Zahlen in ihre Ziffern, um entsprechende Zahlenspielereien machen zu können.

Vollkommene oder perfekte Zahlen

Laut Definition wird eine Zahl als »vollkommene Zahl« oder auch »perfekte Zahl« bezeichnet, wenn ihr Wert gleich der Summe ihrer echten Teiler ist (also ohne sich selbst). Klingt etwas merkwürdig, ist aber ganz einfach. Betrachten wir als Beispiel die Zahl 6: Sie hat als echte Teiler die Zahlen 1, 2 und 3. Interessanterweise gilt nun:

$$1+2+3=6$$

Schauen wir uns noch ein Gegenspiel an: die Zahl 20. Diese besitzt die echten Teiler 1, 2, 4, 5 und 10, deren Summe ist jedoch 22 und nicht 20:

$$1+2+4+5+10=22$$

Armstrong-Zahlen

Im Folgenden betrachten wir sogenannte Armstrong-Zahlen: Das sind Zahlen, deren einzelne Ziffern zunächst mit der Anzahl der Ziffern in der Zahl potenziert und danach summiert werden. Wenn diese Summe dann mit dem Wert der ursprünglichen Zahl übereinstimmt, so spricht man von einer Armstrong-Zahl. Um das Ganze etwas einfacher zu halten, schauen wir uns den Spezialfall einer dreistelligen Zahl an. Als Armstrong-Zahl muss für diese Zahl folgende Gleichung erfüllt sein:

$$100 * x + 10 * y + z = x^3 + y^3 + z^3$$

Dabei sind die Ziffern der Zahl als x, y und z modelliert und jeweils alle dem Wertebereich von 0 bis 9 entnommen.

Die Formel $100 * x + 10 * y + z$ ergibt sich aus der Position der Ziffern und einer textuellen Darstellung von `"xyz"`, also

$$100 * 1 + 10 * 5 + 3 = \text{"153"}$$

$$100 * 3 + 10 * 7 + 1 = \text{"371"}$$

Nach dieser Vorüberlegung betrachten wir zwei Beispiele, für die die obige Formel erfüllt ist:

$$\begin{aligned} 153 = 1^3 + 5^3 + 3^3 &= 1 + 125 + 27 = 153 \\ 371 = 3^3 + 7^3 + 1^3 &= 27 + 343 + 1 = 371 \end{aligned}$$

Abwandlung Als Abwandlung ist es auch ganz interessant, für welche Ziffern bzw. Zahlen folgende Gleichungen erfüllt sind:

$$100 * x + 10 * y + z = x^1 + y^2 + z^3$$

oder

$$100 * x + 10 * y + z = x^3 + y^2 + z^1$$

Für die erste Gleichung gibt es folgende Lösungen:

$$[135, 175, 518, 598]$$

Für die zweite Gleichung existiert für x, y, z im Bereich bis 100 keine Lösung. Wenn Sie mögen, können Sie das beim Implementieren des Bonusteils von Aufgabe 9 selbst nachvollziehen – oder einfach in die Lösungen schauen.

Algorithmus für eine einfache Prüfsumme

In diverse Zahlen ist eine Prüfsumme hineincodiert, sodass die Validität ermittelt werden kann. Das gilt etwa für Kreditkartennummern und bei Datenübertragungen über spezielle Protokolle.

Nehmen wir an, es wäre eine Prüfsumme für eine Zahl mit vier Ziffern (nachfolgend als a bis d modelliert) zu berechnen. Dann könnte man positionsbasiert folgende Rechnung vornehmen:

$$abcd \Rightarrow (a * 1 + b * 2 + c * 3 + d * 4) \,\%\, 10$$

Auch hier möchte ich die Berechnung anhand von Beispielen verdeutlichen:

Eingabe	Positionsberechnung	Wert	Prüfsumme
1111	1 * 1 + 1 * 2 + 1 * 3 + 1 * 4	1 + 2 + 3 + 4 = 10	10 % 10 = 0
1234	1 * 1 + 2 * 2 + 3 * 3 + 4 * 4	1 + 4 + 9 + 16 = 30	30 % 10 = 0
4321	4 * 1 + 3 * 2 + 2 * 3 + 1 * 4	4 + 6 + 6 + 4 = 20	20 % 10 = 0
7271	7 * 1 + 2 * 2 + 7 * 3 + 1 * 4	7 + 4 + 21 + 4 = 36	36 % 10 = 6
0815	0 * 1 + 8 * 2 + 1 * 3 + 5 * 4	0 + 16 + 3 + 20 = 39	39 % 10 = 9
5180	5 * 1 + 1 * 2 + 8 * 3 + 0 * 4	5 + 2 + 24 + 0 = 31	31 % 10 = 1

2.2 Aufgaben

2.2.1 Aufgabe 1: Grundrechenarten (★☆☆☆☆)

Aufgabe 1a: Grundrechenarten (★☆☆☆☆)

Schreiben Sie eine Funktion `calc(m, n)`, die zwei Variablen vom Typ `int` multipliziert, das Produkt dann halbiert und den ganzzahligen Rest bezüglich der Division durch 7 ausgibt.

Beispiele

m	n	m * n	m * n // 2	Resultat ((n * m // 2) % 7)
6	7	42	21	0
5	5	25	12	5

Als kleiner Hinweis zur Erinnerung hier nochmals: Bei einer Ganzzahldivision wird der Rest abgeschnitten, deswegen ergibt $25/2$ als Ergebnis den Wert 12.

Aufgabe 1b: Statistik (★★☆☆☆)

Ermitteln Sie die Summe sowie die Anzahl der natürlichen Zahlen, die durch 2 oder 7 teilbar sind, bis zu einem gegebenen Maximalwert (exklusiv) und geben Sie diese auf der Konsole aus. Schreiben Sie eine Funktion `calc_sum_and_count_all_numbers_div_by_2_or_7(max_exclusive)`. Erweitern Sie das Ganze, sodass statt der Konsolenausgabe eine Rückgabe der beiden Werte erfolgt.

Beispiele

			Resultat	
Maximalwert	Teilbar durch 2	Teilbar durch 7	Anzahl	Summe
3	2	-/-	1	2
8	2, 4, 6	7	4	19
15	2, 4, 6, 8, 10, 12, 14	7, 14	8	63

Aufgabe 1c: Gerade oder ungerade Zahl (★☆☆☆☆)

Erstellen Sie die Funktionen `is_even(n)` und `is_odd(n)`, die prüfen, ob die übergebene Ganzzahl gerade bzw. ungerade ist.

2.2.2 Aufgabe 2: Zahl als Text (★★☆☆☆)

Schreiben Sie eine Funktion `number_as_text(n)`, die für eine gegebene positive Zahl vom Typ `int` die jeweiligen Ziffern in korrespondierenden Text umwandelt.

Starten Sie mit folgendem Fragment für die letzte Ziffer einer Zahl:

```
def number_as_text(n):
    remainder = n % 10
    value_to_text = ""

    if remainder == 0:
        value_to_text = "ZERO"
    if remainder == 1:
        value_to_text = "ONE"

    # ...

    return value_to_text
```

Beispiele

Eingabe	Resultat
7	"SEVEN"
42	"FOUR TWO"
24680	"TWO FOUR SIX EIGHT ZERO"
13579	"ONE THREE FIVE SEVEN NINE"

2.2.3 Aufgabe 3: Vollkommene Zahlen (★★☆☆☆)

Laut Definition wird eine Zahl als »vollkommene / perfekte Zahl« bezeichnet, wenn ihr Wert gleich der Summe ihrer echten Teiler ist. Das gilt etwa für die 6 oder die 28:

$$\begin{aligned} 1+2+3 &= 6 \\ 1+2+4+7+14 &= 28 \end{aligned}$$

Schreiben Sie eine Funktion `calc_perfect_numbers(max_exclusive)`, die die vollkommenen Zahlen bis zu einem Maximalwert, z. B. 10.000, errechnet.

Beispiele

Eingabe	Resultat
1000	[6, 28, 496]
10000	[6, 28, 496, 8128]

2.2.4 Aufgabe 4: Primzahlen (★★☆☆☆)

Schreiben Sie eine Funktion `calc_primes_up_to(max_value)` zur Berechnung aller Primzahlen bis zu einem vorgegebenen Maximalwert. Zur Erinnerung: Eine Primzahl ist eine natürliche Zahl, die größer als 1 und ausschließlich durch sich selbst und durch 1 teilbar ist. Zur Berechnung existiert das sogenannte Sieb des Eratosthenes, was zuvor schon beschrieben wurde.

Beispiele Prüfen Sie Ihren Algorithmus mit folgenden Werten:

Eingabe	Resultat
15	[2, 3, 5, 7, 11, 13]
25	[2, 3, 5, 7, 11, 13, 17, 19, 23]
50	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

2.2.5 Aufgabe 5: Primzahlpaare (★★☆☆☆)

Berechnen Sie alle Paare von Primzahlen mit einem Abstand von 2 (Zwilling), 4 (Cousin) und 6 (sexy) bis zu einer oberen Grenze für n. Für Zwillinge gilt dann:

$$isPrime(n) \;\&\&\; isPrime(n+2)$$

Beispiele Folgende Ergebnisse werden für den Grenzwert 50 erwartet:

Art	Resultat
Zwilling	{3: 5, 5: 7, 11: 13, 17: 19, 29: 31, 41: 43}
Cousin	{3: 7, 7: 11, 13: 17, 19: 23, 37: 41, 43: 47}
Sexy	{5: 11, 7: 13, 11: 17, 13: 19, 17: 23, 23: 29, 31: 37, 37: 43, 41: 47, 47: 53}

2.2.6 Aufgabe 6: Prüfsumme (★★☆☆☆)

Erstellen Sie eine Funktion `calc_checksum(input)`, die zu einer beliebig langen als String vorliegenden Zahl für deren Prüfsumme positionsbasiert folgende Rechnung vornimmt – dabei seien die n Ziffern als z_1 bis z_n modelliert:

$$z_1 z_2 z_3 \ldots z_n \Rightarrow (1 * z_1 + 2 * z_2 + 3 * z_3 + \ldots + n * z_n) \,\%\, 10$$

Beispiele

Eingabe	Summe	Resulat
"11111"	1 + 2 + 3 + 4 + 5 = 15	15 % 10 = 5
"87654321"	8 + 14 + 18 + 20 + 20 + 18 + 14 + 8 = 120	120 % 10 = 0

2.2.7 Aufgabe 7: Römische Zahlen (★★★★☆)

Aufgabe 7a: Römische Zahlen → Dezimalzahlen (★★★☆☆)

Schreiben Sie eine Funktion `from_roman_number(roman_number)`, die aus einer textuell vorliegenden, gültigen römischen Zahl die korrespondierende Dezimalzahl errechnet.[4]

Aufgabe 7b: Dezimalzahlen → Römische Zahlen (★★★★☆)

Schreiben Sie eine Funktion `to_roman_number(value)`, die eine Dezimalzahl in eine (gültige) römische Zahl in textueller Form umwandelt.

Beispiele

Arabisch	Römisch
17	"XVII"
444	"CDXLIV"
1971	"MCMLXXI"
2020	"MMXX"

2.2.8 Aufgabe 8: Kombinatorik (★★☆☆☆)

Aufgabe 8a: Berechnung von $a^2 + b^2 = c^2$

Berechnen Sie alle Kombinationen der Werte a, b und c (jeweils startend ab 1 und kleiner 100), für die folgende Formel gilt:

$$a^2 + b^2 = c^2$$

Bonus (★★★☆☆) Reduzieren Sie die Laufzeit von $O(n^3)$ auf $O(n^2)$. Konsultieren Sie bei Bedarf den Anhang C für eine Einführung in die O-Notation.

Aufgabe 8b: Berechnung von $a^2 + b^2 = c^2 + d^2$

Berechnen Sie alle Kombinationen der Werte a, b, c und d (jeweils startend ab 1 und kleiner 100), für die folgende Formel gilt:

$$a^2 + b^2 = c^2 + d^2$$

Bonus (★★★☆☆) Reduzieren Sie die Laufzeit von $O(n^4)$ auf $O(n^3)$.

[4] Für syntaktisch ungültige römische Zahlen, wie IXD, darf ein falsches Ergebnis, hier 489, errechnet werden – durch zweimal Subtraktionsregel hintereinander: $0 - 1 - 10 + 500$.

2.2.9 Aufgabe 9: Armstrong-Zahlen (★★☆☆☆)

Diese Übung behandelt dreistellige Armstrong-Zahlen. Per Definition versteht man darunter Zahlen, für deren Ziffern x, y und z von 1 bis 9 folgende Gleichung erfüllt ist:

$$100 * x + 10 * y + z = x^3 + y^3 + z^3$$

Schreiben Sie eine Funktion `calc_armstrong_numbers()` zur Berechnung aller Armstrong-Zahlen für x, y und z (jeweils < 10).

Beispiele

$$\begin{aligned} 153 = 1^3 + 5^3 + 3^3 &= 1 + 125 + 27 = 153 \\ 371 = 3^3 + 7^3 + 1^3 &= 27 + 343 + 1 = 371 \end{aligned}$$

Bonus Finden Sie eine generische Variante mit Funktionen oder Lambdas und probieren Sie dann folgende drei Formeln aus:

$$\begin{aligned} 100 * x + 10 * y + z &= x^3 + y^3 + z^3 \\ 100 * x + 10 * y + z &= x^1 + y^2 + z^3 \\ 100 * x + 10 * y + z &= x^3 + y^2 + z^1 \end{aligned}$$

2.2.10 Aufgabe 10: Max Change Calculator (★★★★☆)

Nehmen wir an, wir hätten eine Sammlung von Münzen bzw. Zahlen unterschiedlicher Werte. Schreiben Sie eine Funktion `calc_max_possible_change(values)`, die für positive Ganzzahlen ermittelt, welche Beträge damit ausgehend vom Wert 1 *nahtlos* erzeugt werden können. Als Ergebnis soll der Maximalwert zurückgeliefert werden.

Beispiele

Eingabe	Mögliche Werte	Maximum
1	1	1
1, 1	1, 2	2
1, 5	1	1
1, 2, 4	1, 2, 3, 4, 5, 6, 7	7
1, 2, 3, 7	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13	13
1, 1, 1, 1, 5, 10, 20, 50	1, 2, 3, 4, 5, 6, . . . , 30, . . . , 39	39

2.2.11 Aufgabe 11: Befreundete Zahlen (★★☆☆☆)

Zwei Zahlen n_1 und n_2 heißen befreundet, wenn die Summe ihrer Teiler der jeweils anderen Zahl entsprechen:

$$sum(divisors(n_1)) = n_2$$

$$sum(divisors(n_2)) = n_1$$

Schreiben Sie eine Funktion `calc_friends(max_exclusive)` zur Berechnung aller befreundeten Zahlen bis zu einem übergebenen Maximalwert.

Beispiele

Eingabe	Teiler
$\sum(divisors(220)) = 284$	div(220) = 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110
$\sum(divisors(284)) = 220$	div(284) = 1, 2, 4, 71, 142
$\sum(divisors(1184)) = 1210$	div(1184) = 1, 2, 4, 8, 16, 32, 37, 74, 148, 296, 592
$\sum(divisors(1210)) = 1184$	div(1210) = 1, 2, 5, 10, 11, 22, 55, 110, 121, 242, 605

2.2.12 Aufgabe 12: Primfaktorzerlegung (★★★☆☆)

Jede natürliche Zahl größer 1 lässt sich als eine Multiplikation von Primzahlen darstellen – denken Sie daran, die 2 ist auch eine Primzahl. Schreiben Sie eine Funktion `calc_prime_factors(value)`, die eine Liste mit Primzahlen liefert, deren Multiplikation die gewünschte Zahl ergeben.

Beispiele

Eingabe	Primfaktoren	Resultat
8	2 * 2 * 2	[2, 2, 2]
14	2 * 7	[2, 7]
42	2 * 3 * 7	[2, 3, 7]
1155	3 * 5 * 7 * 11	[3, 5, 7, 11]
2222	2 * 11 * 101	[2, 11, 101]

2.3 Lösungen

2.3.1 Lösung 1: Grundrechenarten (★☆☆☆☆)

Lösung 1a: Grundrechenarten (★☆☆☆☆)

Schreiben Sie eine Funktion `calc(m, n)`, die zwei Variablen vom Typ `int` multipliziert, das Produkt dann halbiert und den ganzzahligen Rest bezüglich der Division durch 7 ausgibt.

Beispiele

m	n	m * n	m * n // 2	Resultat ((n * m // 2) % 7)
6	7	42	21	0
5	5	25	12	5

Als kleiner Hinweis zur Erinnerung hier nochmals: Bei einer Ganzzahldivision wird der Rest abgeschnitten, deswegen ergibt $25/2$ als Ergebnis den Wert 12.

Algorithmus Die Implementierung folgt einfach der mathematischen Operationen:

```
def calc(m, n):
    return m * n // 2 % 7
```

Statt des speziellen Operators `//` kann man auch eine Umwandlung des Ergebnisses der einfachen Division in eine Ganzzahl durch einen Aufruf von `int()` vornehmen:

```
def calc_v2(m, n):
    return int(m * n / 2) % 7
```

Lösung 1b: Statistik (★★☆☆☆)

Ermitteln Sie die Summe sowie die Anzahl der natürlichen Zahlen, die durch 2 oder 7 teilbar sind, bis zu einem gegebenen Maximalwert (exklusiv) und geben Sie diese auf der Konsole aus. Schreiben Sie eine Funktion `calc_sum_and_count_all_numbers_div_by_2_or_7(max_exclusive)`. Erweitern Sie das Ganze, sodass statt der Konsolenausgabe eine Rückgabe der beiden Werte erfolgt.

Beispiele

			Resultat	
Maximalwert	Teilbar durch 2	Teilbar durch 7	Anzahl	Summe
3	2	-/-	1	2
8	2, 4, 6	7	4	19
15	2, 4, 6, 8, 10, 12, 14	7, 14	8	63

Algorithmus Die Implementierung ist ein klein wenig komplexer als zuvor und nutzt zwei Variablen für Anzahl und Summe sowie eine Schleife. Dabei wird mit Modulo geprüft, ob die Teilbarkeit gegeben ist:

```
def calc_sum_and_count_all_numbers_div_by_2_or_7(max_exclusive):
    count = 0
    sum = 0

    for i in range(1, max_exclusive):
        if i % 2 == 0 or i % 7 == 0:
            count += 1
            sum += i

    print("count:", count)
    print("sum:", sum)
```

Verbleibt noch der Wunsch nach der Rückgabe der beiden Werte. Mit Python ist das kein Problem, da man dazu Tupel nutzen kann, etwa mit `return (sum, count)` oder noch kürzer `return sum, count`.

Noch etwas übersichtlicher ist es, schlicht ein Dictionary zu nutzen – das macht den Unit Test nachher auch sehr gut lesbar:

```
def calc_sum_and_count_all_numbers_div_by_2_or_7_v2(max_exclusive):
    count = 0
    sum = 0

    for i in range(1, max_exclusive):
        if i % 2 == 0 or i % 7 == 0:
            count += 1
            sum += i

    return {"sum": sum, "count": count}
```

Lösung 1c: Gerade oder ungerade Zahl (★☆☆☆☆)

Erstellen Sie die Funktionen `is_even(n)` und `is_odd(n)`, die prüfen, ob die übergebene Ganzzahl gerade bzw. ungerade ist.

Algorithmus Die Implementierung nutzt jeweils den Modulo-Operator. Eine Zahl ist gerade, sofern eine Division durch 2 keinen Rest besitzt, andernfalls ist sie ungerade:

```
def is_even(n):
    return n % 2 == 0


def is_odd(n):
    return n % 2 != 0
```

Prüfung

Zum Test der Aufgabe 1a nutzen wir einen parametrisierten Test und für die Angabe der Eingabewerte für m und n sowie für das Ergebnis eine kommaseparierte Aufzählung – zum Auffrischen Ihres Wissens zu Pytest empfiehlt sich ein Blick in Anhang A.

```
@pytest.mark.parametrize("m, n, expected",
                         [(6, 7, 0), (3, 4, 6), (5, 5, 5)])
def test_calc(m, n, expected):
    assert calc(m, n) == expected
```

Zur Kontrolle des Aufgabenteils 1b nutzen wir die Python-Kommandozeile:

```
>>> calc_sum_and_count_all_numbers_div_by_2_or_7(8)
count: 4
sum: 19
```

Generell ist es aber zu bevorzugen, Unit Tests zu erstellen. In anderen Sprachen wäre bereits ein kombinierter Rückgabewert eine erste Hürde – mit Python und Tupeln in Kombination mit Dictionaries gestaltet sich das sehr leicht:

```
@pytest.mark.parametrize("n, expected",
                         [(3, {"sum": 2, "count": 1}),
                          (8, {"sum": 19, "count": 4}),
                          (15, {"sum": 63, "count": 8})])
def test_calc_sum_and_count_all_numbers_div_by_2_or_7_v2(n, expected):
    assert calc_sum_and_count_all_numbers_div_by_2_or_7_v2(n) == expected
```

Die Prüfung der Aufgabe 1c auf gerade oder ungerade ist so einfach, dass wir uns hier mit zwei exemplarischen Aufrufen in der Python-Kommandozeile begnügen:

```
>>> is_even(2)
True
>>> is_odd(7)
True
```

2.3.2 Lösung 2: Zahl als Text (★★☆☆☆)

Schreiben Sie eine Funktion `number_as_text(n)`, die für eine gegebene positive Zahl vom Typ `int`die jeweiligen Ziffern in korrespondierenden Text umwandelt.

Beispiele

Eingabe	Resultat
7	"SEVEN"
42	"FOUR TWO"
24680	"TWO FOUR SIX EIGHT ZERO"
13579	"ONE THREE FIVE SEVEN NINE"

Algorithmus Berechne immer den Rest (also die letzte Ziffer), gib diesen aus und teile dann durch zehn. Wiederhole dies, bis kein Rest mehr existiert. Beachte, dass die Repräsentation der Ziffer vorne an den Text angehängt werden muss, weil immer die letzte Ziffer extrahiert wird und ansonsten die Texte der Ziffern in der falschen Reihenfolge geliefert würden:

```
def number_as_text(n):
    value = ""
    remaining_value = n
    while remaining_value > 0:
        remainder_as_text = digit_as_text(remaining_value % 10)
        remaining_value = int(remaining_value / 10)
        value = remainder_as_text + " " + value

    return value.strip()
```

Die Abbildung von Ziffer auf Text realisieren wir mit einem Dictionary wie folgt:

```
value_to_text_mapping = {
    0: "ZERO", 1: "ONE", 2: "TWO", 3: "THREE", 4: "FOUR",
    5: "FIVE", 6: "SIX", 7: "SEVEN", 8: "EIGHT", 9: "NINE"}


def digit_as_text(n):
    return value_to_text_mapping[n % 10]
```

Python-Shortcut Wie einleitend erwähnt, kann man bei Division und Modulo die Python-Built-in-Funktion `divmod()` oftmals gewinnbringend einsetzen. Dadurch verändert sich der Ablauf nur minimal:

```
def number_as_text(n):
    value = ""
    remaining_value = n
    while remaining_value > 0:
        remaining_value, remainder = divmod(remaining_value, 10)
        value = digit_as_text(remainder) + " " + value

    return value.strip()
```

Es gibt noch eine Variante, die zeichenweise durch die Zahl geht, indem zunächst ein String daraus erstellt wird:

```
def number_as_text_shorter(n):
    value = ""
    for ch in str(n):
        value += digit_as_text(int(ch)) + " "

    return value.strip()
```

Prüfung

Zur Prüfung auf korrekte Funktionsweise nutzen wir einen parametrisierten Test, der mit Pytest elegant zu formulieren ist:

```
@pytest.mark.parametrize("n, expected",
                         [(7, "SEVEN"), (42, "FOUR TWO"),
                          (7271, "SEVEN TWO SEVEN ONE"),
                          (24680, "TWO FOUR SIX EIGHT ZERO"),
                          (13579, "ONE THREE FIVE SEVEN NINE")])
def test_number_as_text(n, expected):
    assert number_as_text(n) == expected
```

2.3.3 Lösung 3: Vollkommene Zahlen (★★☆☆☆)

Laut Definition wird eine natürliche Zahl als »vollkommene / perfekte Zahl« bezeichnet, wenn ihr Wert gleich der Summe ihrer echten Teiler ist. Das gilt etwa für die 6 oder die 28:

$$\begin{aligned} 1+2+3 &= 6 \\ 1+2+4+7+14 &= 28 \end{aligned}$$

Schreiben Sie eine Funktion `calc_perfect_numbers(max_exclusive)`, die die vollkommenen Zahlen bis zu einem Maximalwert, z. B. 10.000, errechnet.

Beispiele

Eingabe	Resultat
1000	[6, 28, 496]
10000	[6, 28, 496, 8128]

Algorithmus Die einfachste Variante besteht darin, alle Zahlen von 2 bis zur Hälfte des gewünschten Maximalwerts zu prüfen, ob diese einen Teiler der ursprünglichen Zahl darstellen. In dem Fall wird die Summe der Teiler um genau den Wert erhöht. Die Summe startet mit dem Wert 1, weil dies immer ein gültiger Teiler ist. Zum Abschluss muss man nur noch die ermittelte Summe mit der eigentlichen Zahl vergleichen:

```
def is_perfect_number_simple(number):
    # immer durch 1 teilbar
    sum_of_multipliers = 1

    for i in range(2, int(number / 2) + 1):
        if number % i == 0:
            sum_of_multipliers += i

    return sum_of_multipliers == number
```