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.

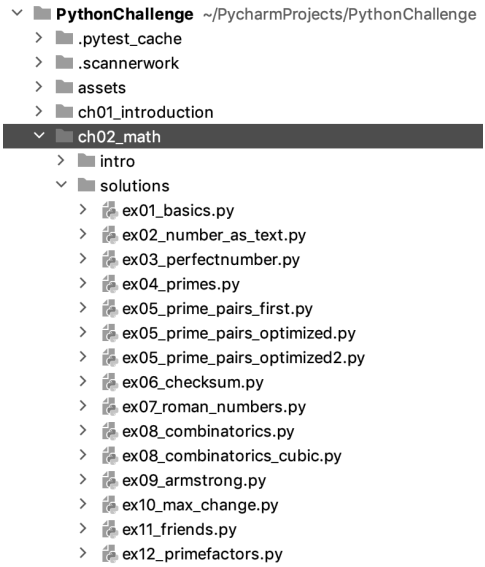
Je komplizierter allerdings die Algorithmen werden, desto mehr lauern auch Fehlerquellen, wie falsche Indexwerte, eine versehentliche oder unterbliebene Negation oder ein übersehener Randfall. Deswegen bietet es sich an, Funktionalitäten mithilfe von Unit Tests zu überprüfen – in diesem Buch kann das 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`.

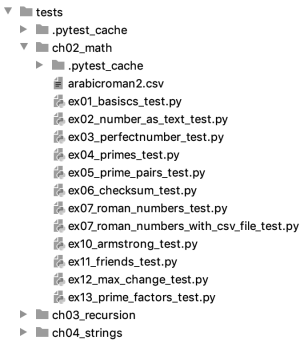
Sourcen Nachfolgend ist ein Ausschnitt für das Kapitel 2 gezeigt:



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 nachfolgende 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 Methodenanfang

Um für stabile Programme zu sorgen, ist es oftmals eine gute Idee, die Parameter einer Methode 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.