Coding-Skills für Nicht-Informatiker

Johannes Schildgen

Python für KI- und Daten-Projekte

+ KI einsetzen, Daten analysieren, Routineaufgaben automatisieren
+ Mit wichtigen Werkzeugen wie Jupyter Notebooks, pandas und Matplotlib
+ Daten normalisieren, auswerten und professionell visualisieren

Mit Datenmaterial und allen Beispielprojekten zum Download

Rheinwerk Computing

Kapitel 4

Mit Dateien arbeiten

Dateien, in denen Texte, Bilder, Excel-Tabellen oder andere Daten gespeichert sind, lassen sich mit Python auf eine einfache Art einlesen und schreiben. Wir schauen uns in diesem Kapitel an, wie dies für die einzelnen Dateitypen funktioniert und wie du Automationen auf ganzen Ordnern erstellst.

In unseren bisherigen Programmen standen alle zu verarbeitenden Daten entweder direkt im Programmcode, z. B. `vorname = "Anna"`, oder wurden während der Laufzeit des Programms vom Benutzer eingegeben: `vorname = input("Vorname: ")`.

In der Praxis hat man Daten jedoch häufig in auf dem Computer gespeicherten Dateien vorliegen. Diese können aus dem Internet stammen, jemand hat sie einem zugesendet, eine Maschine hat die Datei erzeugt, oder eine andere Anwendung hat uns die Datei als Ausgabe geliefert. Wir schauen uns nun an, wie man Dateien, die teilweise auch sehr groß sein können, mit Python einlesen kann und wie sich mit Python Daten in Dateien schreiben lassen.

Eine Datei hat stets einen Dateinamen, einen Dateityp und einen Pfad, der besagt, wo genau – also in welchem Ordner – die Datei gespeichert ist:

C:\Users\Kai\Documents\Verkaufszahlen.xlsx

Der Dateiname ist hier *Verkaufszahlen.xlsx*, der Dateityp ist eine Excel-Datei, was du an der Dateiendung `.xlsx` erkennst, und der Pfad ist die komplette Angabe vom Laufwerksbuchstaben *C:* über die Ordner und Unterordner bis zum Dateinamen selbst. Je nachdem, ob du Windows, macOS oder Linux nutzt, werden Dateipfade etwas anders dargestellt. Bei Windows hat man Laufwerke, also *C:*, *D:* usw. Bei macOS sieht ein Pfad so aus: */Users/kai/Documents/Verkaufszahlen.xlsx*, und unter Linux wie folgt: */home/kai/Documents/Verkaufszahlen.xlsx*. Einen wichtigen Unterschied erkennst du bei den Schrägstrichen (engl. »Slash«): Während Windows Backslashes \ verwendet, werden die einzelnen Pfadabschnitte in macOS und Linux mit / getrennt.

Wenn du dir über Pfade lieber keine Gedanken machen willst, kannst du in deinen Programmen auch einfach relative Pfadangaben verwenden, z. B. *Verkaufszahlen.xlsx*. Das bedeutet, dass die Datei in genau dem gleichen Verzeichnis liegt, »wo du gerade bist«, also dort, wo auch dein *.py*-Python-Skript gespeichert ist. Hier eine kleine Übersicht über Dateipfade:

- *D:\daten\datei.txt* – absolute Pfadangabe unter Windows (beginnt immer mit einem Laufwerksbuchstaben)
- */tmp/datei.txt* – absolute Pfadangabe unter macOS und Linux (beginnt immer mit einem /, dem sogenannten Wurzelverzeichnis)
- *datei.txt* oder *.\datei.txt* bzw. *./datei.txt* – relative Pfadangabe: Datei liegt im gleichen Verzeichnis.
- *daten\datei.txt* bzw. *daten/datei.txt* – relative Pfadangabe: Datei liegt im Unterordner *daten*.
- *..\datei.txt* bzw. *../datei.txt* – relative Pfadangabe: Datei liegt ein Ordner obendrüber.

Hier ist noch eine kurze Übersicht über wichtige Dateiformate. Danach legen wir los und schauen uns an, wie man mit diesen Dateien in Python arbeitet.

Endung	Dateityp	Beschreibung
.txt	Textdatei	reiner Text aus Zeichen
.csv	Comma-Separated Values	Tabellarische Daten. In jeder Zeile sind die Werte z. B. mit Kommas getrennt.
.xlsx, .xls	Tabellenkalkulationsdatei	aus Microsoft Excel
.png, .jpg, .gif	Bild	Grafiken, Fotos ...
.pdf	Portable Document Format	Dokumente
.json	JavaScript Object Notation	Strukturierte Daten im Textformat. Unterstützung für Listen, verschachtelte Strukturen und mehr
.xml	Extensible Markup Language	textbasiertes Austauschformat mit hierarchischer Struktur

Tabelle 4.1: Dateiformate

4.1 Textdateien lesen und schreiben

Textdateien haben üblicherweise die Dateiendung *.txt*, sie bestehen aus beliebig vielen Zeilen Text, und du kannst sie in einem Texteditor öffnen, z. B. unter Windows mit dem Editor (englisch: Notepad) oder Notepad++, unter macOS mit TextEdit und unter Linux mit gedit, kate, nano oder vim. Mit Visual Studio Code geht das unter allen Betriebssystemen allerdings auch.

Was muss ich in meinem Jupyter Notebook in die Codezelle schreiben, um den Inhalt aus der Datei gedichte.txt zu lesen?

Kein Problem! In Python (und damit auch in einem Jupyter Notebook) kannst du den Inhalt einer Datei sehr einfach lesen. Angenommen, deine Datei heißt *gedichte.txt* und liegt im gleichen Verzeichnis wie dein Notebook, kannst du Folgendes in eine Codezelle schreiben:

```
# Datei öffnen und Inhalt lesen
with open("gedichte.txt", "r", encoding="utf-8") as datei:
    inhalt = datei.read()

# Inhalt anzeigen
print(inhalt)
```

Hier wird der komplette Inhalt der Datei *gedichte.txt* als String in die Variable `inhalt` geschrieben und danach ausgegeben.

Zwei Dinge sind an diesem Code vielleicht etwas schwierig zu verstehen: Zum einen das `with`, das behandele ich gleich, und zum anderen der Aufruf `open("gedichte.txt", "r", encoding="utf-8")`. Die Funktion `open` öffnet die Datei, sodass du sie mit Python lesen oder schreiben kannst. Was du genau machen möchtest, gibst du mit dem zweiten Parameter an. `"r"` steht für den Lese-Modus (*read*); wir wollen die Datei ja lesen. Die Angabe des Encodings ist oftmals sinnvoll, weil ansonsten Umlaute eventuell nicht korrekt gelesen werden. Je nachdem, mit welchem Programm die Datei nämlich erstellt wurde, können Umlaute und Sonderzeichen intern auf unterschiedliche Arten gespeichert werden. Ärgerlich ist, dass unter Windows das Standard-Encoding meist `"cp1252"` ist und unter Linux und macOS UTF-8. Wenn du auf Nummer sicher gehen willst, solltest du am besten beim Schreiben und Lesen von Dateien immer das passende Encoding angeben, z. B. `encoding="utf-8"`.

Den gleichen Programmcode wie eben kannst du auch ohne das `with … as …:` wie folgt schreiben:

```
datei = open("gedichte.txt", "r", encoding="utf-8")
inhalt = datei.read()
datei.close()
print(inhalt)
```

Das sieht vielleicht auf den ersten Blick simpler und besser verständlich aus, aber dennoch solltest du lieber die `with`-Schreibweise verwenden. Es gibt nämlich das Prinzip: Was man

öffnet, sollte man auch wieder schließen. Wenn wir eine Datei öffnen, belegt das Ressourcen, und möglicherweise wird die Datei dadurch auch gesperrt, sodass eine andere Anwendung sie währenddessen nicht mehr öffnen kann. Wenn wir die Datei am Ende mittels `datei.close()` schließen, ist das wunderbar. Aber dieses Schließen vergisst man leider oft, und dann können Probleme auftreten. Außerdem würde unsere Datei nicht geschlossen, wenn beim Lesen ein Fehler auftritt, da wir dann gar nicht erst bis zur `close`-Zeile kämen. Mit der `with`-Schreibweise sind wir immer auf der sicheren Seite. Dabei kümmert sich Python darum, die Datei automatisch zu schließen, sobald der Block abgearbeitet wurde; auch dann, wenn ein Fehler auftrat.

```
with open("gedichte.txt", "r", encoding="utf-8") as datei:
    inhalt = datei.read()
print(inhalt)
```

Die Variable `datei` existiert nur in dem eingerückten Codeblock, der nach der `with`-Zeile kommt. In diesem Codeblock arbeiten wir mit der Datei. In unserem Fall wird mit `datei.read()` ihr kompletter Inhalt ausgelesen und als String in die Variable `inhalt` geschrieben. Damit können wir auch nach dem Codeblock weiterarbeiten.

Nun hast du gesehen, wie du etwas aus einer Datei liest. Als Nächstes wollen wir etwas in eine Textdatei schreiben:

> *Ein Python-Programm, das die Zahlen von 1 bis 100 in eine Datei zahlen.txt schreibt.*

Klar! Hier ist ein einfaches Python-Programm, das die Zahlen von 1 bis 100 in eine Datei *zahlen.txt* schreibt, jeweils eine Zahl pro Zeile:

```
# Datei öffnen (w = write) oder erstellen
with open("zahlen.txt", "w") as datei:
    # Zahlen von 1 bis 100 durchlaufen
    for zahl in range(1, 101):
        datei.write(f"{zahl}\n")  # Zahl in die Datei schreiben
                                  # und Zeilenumbruch hinzufügen
```

Dieses Mal wird der Modus `"w"` (*write*) verwendet. Wir wollen ja in die Datei *zahlen.txt* schreiben. Wenn sie noch nicht existiert, wird sie angelegt. Existiert sie bereits, wird ihr kompletter Inhalt gelöscht und durch den neuen Inhalt überschrieben. Als wir vorhin aus einer Textdatei gelesen haben, stand innerhalb des `with`-Blocks `datei.read()`. Der Aufruf liefert uns den Inhalt als String. Nun beim Schreiben verwenden wir `datei.write(…)` und übergeben der `write`-Methode einen String – in unserem Fall die Zahl, aber formatiert als String,

und einen Zeilenumbruch, damit jede Zahl in einer separaten Zeile steht. Wir dürfen `datei.write` so oft aufrufen, wie wir wollen. Alles zusammen landet dann letztendlich in der Datei.

In der folgenden Liste siehst du eine Übersicht über die Dateimodi, die du beim Öffnen einer Datei verwenden kannst:

- `"r"` (*read*): Die Datei wird zum Lesen geöffnet; sie muss existieren.
- `"w"` (*write*): Die Datei wird zum Schreiben geöffnet; sie wird überschrieben oder neu angelegt.
- `"a"` (*append*): Die Datei wird geöffnet, um neue Inhalte an vorhandene Inhalte hinten anzuhängen.
- `"r+"`: Die Datei wird zum Lesen und Schreiben geöffnet; sie muss existieren.

4.2 CSV-Dateien

Der folgende Inhalt ist in einer CSV-Datei *schuelernoten.csv* gespeichert:

```
Name,Klasse,Fach,Note
Anna Müller,10a,Mathematik,1
Ben Schmidt,10a,Mathematik,2
Clara Meier,8b,Mathematik,3
David Klein,8b,Mathematik,2
Eva Wagner,9a,Englisch,2
Felix Braun,9b,Englisch,1
Ivan Becker,9a,Biologie,3
```

Die erste Zeile ist in vielen CSV-Dateien der sogenannte *Header*. Darin stehen noch keine wirklichen Daten, sondern die Namen der Spalten. Erst ab der zweiten Zeile geht es mit den einzelnen Datensätzen los. Innerhalb einer Zeile sind die Spaltenwerte mit Kommas getrennt. Bei unserer Datei sind es immer vier Werte pro Zeile, da wir vier Spalten haben. CSV steht zwar für *Comma-Separated Values*, oft wird aber auch ein Tabulator, ein senkrechter Strich, ein Semikolon oder etwas anderes als Spaltentrenner verwendet.

Es gibt in Python eine Vielzahl von Modulen für die Arbeit mit CSV-Dateien, zwei davon sind `csv` und `pandas`. Um `pandas` wird es im nächsten Kapitel gehen. Wenn du mit großen Dateien zu tun hast, umfangreiche Analysen und Berechnungen durchführen möchtest oder die Daten graphisch visualisieren willst, schaue dir unbedingt `pandas` an. Für simple Dinge, wie wir sie in diesem Kapitel vorhaben, reicht aber auch das Modul `csv`. Anders als `pandas` musst du das `csv`-Modul auch nicht separat installieren, weil es in Python eingebaut ist.

schuelernoten.csv sieht so aus:

```
Name,Klasse,Fach,Note
Anna Müller,10a,Mathematik,1
...
```

Ich brauche ein Python-Programm, das einfach nur die Namen der Schüler und Schülerinnen ausgibt.

Du kannst die CSV-Datei in Python sehr einfach einlesen und nur die Namen ausgeben. Hier ist ein kurzes Beispiel:

```
import csv

# Pfad zu deiner CSV-Datei
datei = 'schuelernoten.csv'

with open(datei, newline='', encoding='utf-8') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print(row['Name'])
```

Zuerst wird das Modul `csv` mittels `import csv` importiert. Danach rufen wir `csv.DictReader(csvfile)` auf, um die Inhalte aus der CSV-Datei zu lesen. Mit diesem `reader` können wir dann so arbeiten, wie wir es von Listen in Python kennen: `for row in reader:` liefert eine Zeile aus der CSV-Datei nach der anderen. Innerhalb dieser `for`-Schleife lässt sich mittels `row['Spaltenname']` auf die Werte in den einzelnen Spalten zugreifen. In unserem Fall liefert in jedem Schleifendurchlauf `row['Name']` jeweils einen Namen einer Schülerin oder eines Schülers.

Welche Schulklassen gibt es? Jede nur einmal auflisten, bitte.

Alles klar! Du willst also die einzigartigen Klassen aus deiner CSV-Datei herausfiltern. Hier ist ein kurzes Python-Beispiel:

```
import csv
datei = 'schuelernoten.csv'
klassen = set()  # ein Set speichert nur einzigartige Werte
with open(datei, newline='', encoding='utf-8') as csvfile:
    reader = csv.DictReader(csvfile)
```

```
    for row in reader:
        klassen.add(row['Klasse'])

# Einmalig auflisten
for klasse in sorted(klassen):
    print(klasse)
```

Auch hier wird wieder über die Zeilen der CSV-Datei iteriert und in jedem Schleifendurchlauf auf den Wert der Spalte `'Klasse'` zugegriffen. Da aber jede Schulklasse (10a, 8b …) nur einmal ausgegeben werden soll, erfolgt in diesem Programm die Ausgabe der Klasse nicht sofort. Stattdessen werden die Schulklassen in eine Menge (Englisch: *set*) eingefügt. Da Mengen keine Duplikate speichern, ist am Ende der Schleife jede Schulklasse nur einmal enthalten. Am Ende iterieren wir aber nicht mittels `for klasse in klassen:` über die Einträge in der Menge – das würde sie in einer willkürlichen Reihenfolge ausgeben –, sondern über `sorted(klassen)`. Die Funktion `sorted` wandelt unsere Menge in eine sortierte Liste um, über die wir dann iterieren können.

Wir haben Mengen im vorherigen Kapitel immer in geschweifte Klammern geschrieben, z. B. `{5, 3}`. In unserem Programm liegen uns aber zu Beginn noch gar keine Einträge vor, daher benötigen wir die leere Menge. Diese schreibt man in Python aber nicht mit `{}` (das wäre ein leeres Dictionary), sondern mit `set()`. Und in diese Menge werden dann nach und nach die einzelnen Strings aus der CSV-Datei eingefügt, in der die Schulklassenbezeichnungen stehen.

Frage mit `input(...)` *nach Name, Klasse, Fach und Note und füge dies zur CSV-Datei hinzu.*

Hier ist ein kleines Python-Skript, das einen neuen Eintrag per `input()` abfragt und ihn direkt an die CSV-Datei anhängt:

```
import csv
datei = 'schuelernoten.csv'

# Eingaben vom Benutzer
name = input("Name des Schülers: ")
klasse = input("Klasse: ")
fach = input("Fach: ")
note = input("Note: ")
```

```
# Öffne die Datei im Anhängemodus 'a' (append)
with open(datei, mode='a', newline='', encoding='utf-8') as csvfile:
    writer = csv.writer(csvfile)
    # Schreibe die neue Zeile
    writer.writerow([name, klasse, fach, note])

print(f"Eintrag für {name} wurde hinzugefügt.")
```

Um in eine CSV-Datei zu schreiben, holen wir uns mittels `csv.writer(csvfile)` einen sogenannten *Writer*. Damit lässt sich eine Zeile in die Datei schreiben. `writer.writerow` (lies: *write row*) nimmt eine Liste mit den Spaltenwerten entgegen und schreibt sie in die CSV-Datei. Da der Dateimodus `"a"` und nicht `"w"` verwendet wurde, bleiben die Daten, die aktuell in der Datei gespeichert sind, unberührt. Die neue Zeile wird einfach unten angehängt.

Wir haben uns nun angeschaut, wie man Daten aus einer CSV-Datei liest, wie man sie in eine Datei schreibt und wie man sie unten an eine Datei anhängt. Wenn du wissen willst, wie du etwas innerhalb einer Datei änderst, frage einfach ChatGPT, beispielsweise so:

> *Ein Python-Programm, das die Note eines bestimmten Schülers ändert.*

Den resultierenden Programmcode schauen wir uns jetzt aber nicht an. Das ist nicht so spektakulär, aber leider auch nicht besonders schön. Du musst nämlich tatsächlich die Datei öffnen und zunächst alle Daten in einer Liste speichern. Als Nächstes änderst du innerhalb dieser Liste den entsprechenden Eintrag, und wenn du damit fertig bist, iterierst du in einer Schleife über alle Einträge in der Liste und schreibst sie Zeile für Zeile zurück in die Datei. Das ist der Standardweg in Python. Man ändert also nicht *in place* die eine Stelle, die geändert werden soll, sondern du musst tatsächlich die komplette Datei mit neuen Inhalten neu schreiben. Der Grund dafür ist folgender: Ändert sich eine Note von `2` in `1.7`, was ja zwei Zeichen länger ist, müsste alles ab dort in der CSV-Datei um zwei Zeichen (zwei Byte) nach rechts verschoben werden. Das Ganze ist aber nur so komplex, wenn du etwas mitten in einer Datei ändern willst. Willst du einfach bei einer existierenden CSV-Datei eine Zeile am Ende hinzufügen, geht das einfach, indem du die Datei, wie im obigen Beispiel gezeigt, im Append-Modus öffnest und mit `writer.writerow` die neue Zeile unten dranhängst.

4.3 Dateien verwalten

4.3.1 Datei anlegen

Wenn du den Modus `"w"` eine Datei schreibst, die es nicht gibt, wird sie neu angelegt:

```
with open('leere_datei.txt', 'w', encoding='utf-8'):
    pass
```

Der Python-Befehl `pass` steht dafür, dass in diesem Codeblock nichts gemacht wird. Mit dem `open`-Befehl öffnen wir die Datei zum Schreiben. Da sie nicht existiert, wird sie angelegt. Und da ansonsten nichts weiter gemacht wird, schließt Python die Datei direkt wieder. So legst du also eine neue Datei mit leerem Inhalt an. Im Modul `pathlib` gibt es aber auch noch etwas, womit du eine leere Datei auch so anlegen kannst:

```
from pathlib import Path

pfad = Path('leere_datei.txt')
pfad.touch()
```

Hier wird nicht das komplette Modul `pathlib` importiert, sondern nur die darin enthaltene Klasse `Path`. Das macht auch die Verwendung einfacher. Du darfst nun einfach `Path` schreiben statt `pathlib.Path`. Eine Klasse ist so etwas wie ein spezieller Datentyp. Genau wie du mit `set()`eine Menge erzeugst, legst du mit `Path('leere_datei.txt')` ein Objekt an, das die Datei repräsentieren soll. Und mit `touch()` wird die Datei angelegt, falls sie noch nicht existiert.

Sollte die Datei nicht im aktuellen Ordner liegen, kannst du immer auch einen relativen oder absoluten Pfad angeben. Obwohl Windows bei Pfaden eigentlich den Backslash (\) verwendet, empfiehlt es sich trotzdem, Ordner immer mit / zu trennen, z. B. `"ein_ordner/leere_datei.txt"`. Python unterstützt diese Schreibweise unter allen Betriebssystemen, dann ist es überall einheitlich, und wir haben nicht die Probleme, die bei Strings auftreten, in denen `\` vorkommen. Willst du unter Windows unbedingt den Backslash vermeiden, schreibe Pfade bitte in einen sogenannten Raw-String: `r'ein_ordner\leere_datei.txt'`. Dann bleibt der String so, wie er ist, und Python denkt nicht, `\l` sei ein spezielles Steuerzeichen, so wie `\n`.

4.3.2 Dateien auf Existenz prüfen und löschen

Im Modul `os` (*operating system*) sind viele Funktionen, die mit Betriebssystemfunktionalitäten zu tun haben. Dazu zählt auch die Arbeit mit Dateien.

```
import os
datei = 'meine_datei.txt'

if os.path.exists(datei):
    os.remove(datei)
    print(f"Datei '{datei}' wurde gelöscht.")
```

```
else:
    print(f"Datei '{datei}' existiert nicht.")
```

Die Funktion `os.path.exists` nimmt einen Dateipfad entgegen und liefert `True` oder `False`, je nachdem, ob die Datei existiert oder nicht. `os.remove` sorgt dafür, dass die Datei gelöscht wird.

4.3.3 Datei verschieben, umbenennen und kopieren

Mit den Funktionen `move` und `copy` aus dem Modul `shutil` (Shell Utilities) lassen sich Dateien an einen anderen Ort verschieben, umbenennen und kopieren:

```
import shutil

shutil.move('datei.txt', 'ein_ordner/datei.txt')  # verschieben
shutil.move('datei.txt', 'neuer_name.txt')        # umbenennen
shutil.copy('datei.txt', 'backup_datei.txt')      # nur Inhalt
shutil.copy2('datei.txt', 'backup_datei.txt')     # inkl. Metadaten
```

Mit Metadaten sind Erstelldatum, Zeitpunkt der letzten Änderung, Zugriffsrechte etc. gemeint. Die Funktion `copy2` übernimmt auch diese Metadaten beim Kopieren; bei `copy` wird eine Datei angelegt mit dem jetzigen Zeitpunkt als Erstell- und Änderungsdatum.

4.3.4 Dateien in einem Verzeichnis auflisten

Wenn du einen Ordner hast, in dem mehrere Dateien sind, hast du in Python verschiedene Möglichkeiten, mit einer Schleife über alle Dateien zu iterieren, um dir zum Beispiel die Dateinamen ausgeben zu lassen oder etwas für jede einzelne Datei zu tun:

```
import os
for dateiname in os.listdir("dateien"):
    if dateiname.endswith(".txt"):
        print(dateiname) # der Dateiname als String, z. B. "datei1.txt"

import glob
for dateiname in glob.glob("dateien/*.txt"):
    print(dateiname)  # der Pfad als String, z. B. "dateien/datei1.txt"

from pathlib import Path
ordner = Path("dateien")
for datei in ordner.glob("*.txt"):
    print(datei)   # Path-Objekt
```

Die drei Beispiele zeigen, dass die Variablen `dateiname` bzw. `datei` jedes Mal etwas leicht anderes enthalten. In den ersten beiden Varianten ist es der Dateiname bzw. der Pfad zur Datei als String. Die dritte Variante liefert uns ein `Path`-Objekt, mit dem wir direkt weiterarbeiten können. `datei.read_text()` würde uns zum Beispiel direkt den Textinhalt aus der Datei als String liefern, `datei.stat().st_size` gibt uns die Größe der Datei in Bytes.

4.4 Beispiel: Textanalyse

Ich habe einen Ordner dateien, in dem mehrere Textdateien sind. Erstelle ein Python-Programm, das mir folgende Statistiken ausgibt:

- *Anzahl Zeichen und Wörter je Datei*
- *Anzahl Zeichen und Wörter insgesamt*

Außerdem soll eine Datei woerter.csv erstellt werden, die die 100 häufigsten Wörter (in allen Dateien zusammen) zusammen mit deren Anzahl speichert, z. B. `der,421`.

Klar! Ich erstelle ein Python-Skript, das genau das macht. Ich gehe davon aus, dass sich alle Textdateien in einem Ordner namens *dateien* befinden. Das Skript sammelt die Statistiken pro Datei und insgesamt und erstellt die *woerter.csv* mit den 100 häufigsten Wörtern.

Hier ist ein komplettes Beispiel:

```
import os
import csv

folder = "dateien"
all_words = {}
total_chars = 0
total_words = 0

for filename in os.listdir(folder):
    if filename.endswith(".txt"):
        path = os.path.join(folder, filename)
        with open(path, "r", encoding="utf-8") as f:
            text = f.read()
            chars = len(text)
            words = text.split()

            print(f"{filename}: {chars} Zeichen, {len(words)} Wörter"
```

```
)

            total_chars += chars
            total_words += len(words)

            for w in words:
                w = w.lower()
                all_words[w] = all_words.get(w, 0) + 1

print(f"Gesamt: {total_chars} Zeichen, {total_words} Wörter")

# 100 häufigste Wörter
top_words = sorted(all_words.items(), key=lambda x: x[1],
                   reverse=True)[:100]

with open("woerter.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    for word, count in top_words:
        writer.writerow([word, count])
```

Die Variablen `total_chars` und `total_words` werden zu Beginn des Programms auf `0` gesetzt. Die Zahlen in diesen Variablen werden später für jede Datei um deren Zeichenanzahl und Wortanzahl erhöht, sodass sie am Ende die Gesamtanzahlen enthalten.

In der Schleife `for filename in os.listdir(folder):` wird jede einzelne Datei im Ordner angeschaut und zunächst überprüft, ob sie die Endung *.txt* hat. Falls ja, wird die Datei zum Lesen geöffnet. Dazu setzt `os.path.join` den Dateipfad aus dem Ordner- und dem Dateinamen zusammen.

Nun kann der Text aus der Datei gelesen und die Anzahl der Zeichen und Wörter gezählt werden. Außerdem wird jedes Wort im Text als Key zu einem Dictionary namens `all_words` hinzugefügt. Der entsprechende Value ist die Anzahl der Vorkommen dieses Wortes. Um am Ende die 100 häufigsten Wörter zu finden, wird dieser leider etwas schwierig zu verstehende Ausdruck verwendet:

```
top_words = sorted(all_words.items(), key=lambda x: x[1],
                   reverse=True)[:100]
```

`all_words.items()` erzeugt aus unserem Dictionary eine Liste von Tupeln mit jeweils zwei Einträgen: einem Wort und der Anzahl seines Vorkommens, z. B. `("der", 421)`. In dem Ausdruck repräsentiert die Variable `x` ein solches Tupel. `x[0]` wäre in unserem Fall das Wort `"der"` und `x[1]` dessen Anzahl `421`. Die Funktion `sorted` erzeugt daraus nun eine neue

Liste von Tupeln, jedoch so, dass sie nach `x[1]`, also der Anzahl, sortiert ist. Mit dem Operator `[:100]` wird schließlich die Liste so abgeschnitten, dass nur noch die ersten 100 Elemente in der Variablen `top_words` stehen.

Der Rest des Programms ist wieder besser verständlich: Wir öffnen die CSV-Datei im Write-Modus und schreiben für jeden Eintrag in der `top_words`-Liste eine Zeile in die CSV-Datei.

Wenn ChatGPT Python-Code erzeugt, sind darin häufig komplexe und teilweise schwierig zu verstehende pythonische Codezeilen zu finden. Es ist absolut nicht notwendig, dass du selbst eine Zeile wie die oben stehende formulieren kannst. Aber du solltest grob verstehen, was dort passiert. Im Zweifel kannst du ChatGPT ja auch Rückfragen stellen:

> *Was macht* `[:100]`*?*

> Das `[:100]` ist Python-Slicing und bedeutet: »Nimm nur die ersten 100 Elemente der Liste«.

4.5 Excel-Dateien

CSV-Dateien und Excel-Dateien sind auf den ersten Blick relativ ähnlich. Sie speichern beide tabellarische Daten, also Daten, die in Zeilen und Spalten angeordnet sind. Die Arbeit mit CSV-Dateien ist aber deutlich einfacher, weil sie im Grunde genommen einfach nur Textdateien sind. Du kannst sie im Texteditor betrachten, aber auch mit Excel, LibreOffice und Co. öffnen. Daher solltest du, wenn du nicht unbedingt auf Excel-Dateien angewiesen bist, im Zweifel lieber mit CSV-Dateien arbeiten.

Excel-Dateien lassen sich nicht einfach im Texteditor öffnen. `.xlsx` und `.xls` sind Dateiendungen für sogenannte Binärformate. Man braucht Microsoft Excel oder ein anderes Programm, das mit diesen Dateitypen umgehen kann. Das liegt daran, dass Excel-Tabellen mehr sind als einfach nur Zeilen und Spalten, in denen irgendwelche Werte stehen. Excel-Tabellen unterstützen Formeln und formatierte Zellen (farbig, fett, kursiv …), und es ist möglich, in einer sogenannten Excel-Arbeitsmappe mehrere Tabellenblätter anzulegen.

Wenn du mit Python eine Excel-Datei lesen oder schreiben willst, ist der eleganteste Weg, das Modul `pandas` zu verwenden. Für CSV-Dateien gilt das aber ebenfalls. Dieses Modul werde ich im nächsten Kapitel vorstellen. Dennoch gibt es aber auch andere Module, mit denen Du auf Excel-Dateien zugreifen kannst. Das folgende Beispiel verwendet das Modul `openpyxl`. Beachte, dass du dieses Modul vor der ersten Verwendung erst installieren musst, indem du `pip install openpyxl` im Terminal eingibst (siehe Abschnitt 3.15, »Module importieren und Pakete installieren mit ›pip‹«).

Ich habe eine Excel-Tabelle ausgaben.xlsx mit den Spalten `Datum`, `Bezeichnung`, `Betrag`. *Erstelle ein Python-Programm und verwende* `openpyxl`, *um alle Ausgaben und den Gesamtbetrag auszugeben.*

```
from openpyxl import load_workbook

# Excel-Datei laden
wb = load_workbook("ausgaben.xlsx")
sheet = wb.active  # aktives Arbeitsblatt auswählen

gesamtbetrag = 0

print("Alle Ausgaben:")
print("-" * 40)

# Zeilen durchlaufen, erste Zeile ist die Überschrift
for row in sheet.iter_rows(min_row=2, values_only=True):
    datum, bezeichnung, betrag = row
    print(f"Datum: {datum}, Bezeichnung: {bezeichnung},
          Betrag: {betrag}")
    gesamtbetrag += betrag

print("-" * 40)
print(f"Gesamtbetrag: {gesamtbetrag}")
```

Die Variable `wb` repräsentiert die komplette Excel-Arbeitsmappe. Mit `wb.worksheets[0]` könnten wir auf das erste Arbeitsblatt zugreifen, mit `wb["Januar 2026"]` geht das über den Blattnamen. In unserem Programm greifen wir mit `wb.active` auf dasjenige Blatt zu, das beim Speichern der Datei aktiv war.

Wundere dich nicht über den Ausdruck `"-" * 40` – er erzeugt schlicht einen String mit 40 Bindestrichen. Es soll also eine solche Trennlinie ausgegeben werden: `----------`.

Die Hauptaufgabe in unserem Programm erledigt die Schleife `for row in sheet.iter_rows(min_row=2, values_only=True):` Das liefert uns Zeile für Zeile die Daten aus der Excel-Tabelle. Mit `min_row=2` wird die erste Zeile übersprungen, denn anders als in Python beginnen Zeilennummern in Excel bei 1. In dieser ersten Zeile stehen üblicherweise die Spaltenüberschriften. Und `values_only=True` sorgt dafür, dass `row` ein Tupel ist, in dem für jede Spalte ihr jeweiliger Wert steht, z. B. `("2026-01-01", "Stromrechnung", 85.70)` In

`row[2]` steht auch selbst dann der Wert `85.70`, wenn er gar nicht so direkt in der Excel-Tabelle steht, sondern erst durch einen Formelausdruck berechnet wird.

Schauen wir uns an, wie wir etwas in eine Excel-Tabelle schreiben:

> *Frage den Benutzer nach Datum, Bezeichnung und Betrag und schreibe das ans Ende der Excel-Tabelle.*

```
from openpyxl import load_workbook

# Excel-Datei laden
wb = load_workbook("ausgaben.xlsx")
sheet = wb.active  # aktives Arbeitsblatt auswählen

# Benutzereingaben
datum = input("Datum (z. B. 2025-08-29): ")
bezeichnung = input("Bezeichnung: ")
while True:
    try:
        betrag = float(input("Betrag: "))
        break
    except ValueError:
        print("Bitte eine gültige Zahl eingeben!")

# Neue Zeile ans Ende anhängen
sheet.append([datum, bezeichnung, betrag])

# Excel-Datei speichern
wb.save("ausgaben.xlsx")
print("Eintrag wurde hinzugefügt!")
```

Nachdem das Excel-Arbeitsblatt geöffnet wurde, wird mit `sheet.append` eine Zeile ans Ende der Tabelle angefügt und anschließend die Datei mit `wb.save` gespeichert.

Exception-Handling

In dem gerade gezeigten Programm gibt es einen `try-except`-Block. Ein solcher wird in Python verwendet, damit bei bestimmten Fehlern das Programm nicht abstürzt. Gäbe nämlich ein Benutzer einen Beitrag ein, der gar keine Zahl ist (z. B. `"Hallo"`), würde das Programm einen `ValueError` zeigen und sich sofort beenden. Da die Um-

wandlung der Benutzereingabe in eine Zahl mit der `float`-Funktion aber innerhalb eines `try`-Blocks erfolgt, sagt man Python: Versuch es doch einfach mal! Und wenn es nicht klappt, wird statt eines Fehlers das gemacht, was im `except`-Block steht. Der Benutzer wird dann aufgefordert, erneut eine Zahl einzugeben. Ist diese dann gültig, wird die `while True`-Schleife mit dem `break` beendet, und das Programm geht weiter.

4.6 Bilddateien

Genau wie Excel-Dateien sind Bilder sogenannte *Binärdateien*. Man kann sie nicht in einem Texteditor betrachten, sondern braucht dafür ein spezielles Programm. Für Python gibt es einige Module, die Bilder öffnen und deren Daten und Metadaten auslesen und auch Bilder modifizieren. Mit den Daten eines Bildes sind die einzelnen Bildpunkte gemeint, also wo im Bild welche Farbe ist. In den Metadaten sind die Abmessungen (Breite und Höhe in Pixeln) gespeichert sowie der Aufnahmezeitpunkt, GPS-Koordinaten, Kameramodell und mehr. In den folgenden Beispielen wird das Modul `PIL` (Python Imaging Library), verwendet. Die neuste Version davon heißt Pillow und kann mittels `pip install pillow` installiert werden.

Ein Python-Programm, das bild.jpg auf 1024 Pixel Breite verkleinert.

Klar! Hier ist ein einfaches Python-Programm, das ein Bild *bild.jpg* auf eine Breite von 1024 Pixeln verkleinert und die Proportionen beibehält. Wir verwenden dafür die Pillow-Bibliothek:

```
from PIL import Image

# Bild öffnen
bild = Image.open("bild.jpg")

# Neue Breite
neue_breite = 1024

# Berechne das Verhältnis, um die Höhe proportional anzupassen
breite, hoehe = bild.size
neue_hoehe = int((neue_breite / breite) * hoehe)

# Bild verkleinern
bild_verkleinert = bild.resize((neue_breite, neue_hoehe),
                               Image.LANCZOS)
```

```
# Speichern
bild_verkleinert.save("bild_verkleinert.jpg")

print(f"Bild verkleinert auf {neue_breite}x{neue_hoehe} Pixel.")
```

Die Methode `bild.resize` erzeugt ein neues Bild, das die übergebene Breite und Höhe hat. Während die Breite in unserem Beispiel immer 1.024 Pixel sein soll, müssen wir die Höhe des neuen Bildes vorher erst berechnen, damit die Proportionen erhalten bleiben: `int((neue_breite / breite) * hoehe)`. Die Höhe des Originalbildes wird also mit dem Skalierungsfaktor `neue_breite / breite` multipliziert. Die Funktion `int` rundet das Ergebnis auf eine ganze Zahl. Außerdem benötigt `bild.resize` die Info, welches Verfahren für das Ändern der Größe gewählt werden soll. `Image.LANCZOS` ist ein Algorithmus, der eine gute Bildqualität beim Verkleinern von Fotos liefert.

Sowohl `Image.open` als auch die Methode `save` erkennen anhand der angegebenen Dateiendung, um welche Art von Bild es sich handelt. In unserem Fall haben wir es mit einem JPEG-Bild zu tun. Das Format wird oft für Fotos verwendet und speichert die Bilddaten in einer komprimierten Form. Würden wir der `save`-Methode `"bild_verkleinert.png"` übergeben, würde das Bild in eine PNG-Datei umgewandelt. Diese Datei wäre vermutlich größer, da PNG ein verlustfreies Format ist.

Ich habe einen Ordner bilder. *Ein Python-Programm soll darin Unterordner für jedes Jahr erstellen (2020, 2021 usw.) und die Bilder anhand ihres Aufnahmedatums in diese Unterordner verschieben.*

Alles klar! Wir können das in Python mit den Modulen `os`, `shutil` und Pillow (`PIL`) machen. Pillow wird gebraucht, um das Aufnahmedatum aus den EXIF-Daten der Bilder zu lesen. Hier ist ein komplettes Beispielprogramm:

```
import os
import shutil
from PIL import Image
from PIL.ExifTags import TAGS

# Ordner mit den Bildern
source_folder = "bilder"

# Funktion, um das Aufnahmedatum aus EXIF auszulesen
def get_image_year(image_path):
```

```
    try:
        image = Image.open(image_path)
        exif_data = image._getexif()
        if exif_data:
            for tag_id, value in exif_data.items():
                tag = TAGS.get(tag_id, tag_id)
                if tag == "DateTimeOriginal":
                    year = value.split(":")[0]
                    return year
    except Exception as e:
        print(f"Fehler bei {image_path}: {e}")
    return None

# Alle Dateien im Ordner durchgehen
for filename in os.listdir(source_folder):
    file_path = os.path.join(source_folder, filename)
    if os.path.isfile(file_path):
        year = get_image_year(file_path)
        if year:
            # Zielordner erstellen, falls nicht vorhanden
            target_folder = os.path.join(source_folder, year)
            os.makedirs(target_folder, exist_ok=True)
            # Bild verschieben
            shutil.move(file_path,
                        os.path.join(target_folder, filename))
        else:
            print(f"Kein Aufnahmedatum gefunden für {filename}")
```

Da es etwas umständlich ist, an das Jahr heranzukommen, in dem das Bild aufgenommen wurde, wurde in unserem Programm für genau diesen Zweck eine Funktion `get_image_year` erstellt. Sie nimmt einen Pfad zu einer Bilddatei entgegen und liefert das Aufnahmejahr zurück. Dazu werden die einzelnen im Bild gespeicherten Metadaten betrachtet, bis das Datum gefunden wurde. Dieses liegt in einem solchen Format vor: `2025:12:24 19:32:07`. Das Jahr ist also vor dem ersten Doppelpunkt.

Die Hauptarbeit des Programms wird in der Schleife `for filename in os.listdir(source_folder):` gemacht. Zunächst wird Datei für Datei durchgegangen und mit `if os.path.isfile(file_path):` überprüft, ob es wirklich eine Datei und kein Ordner ist. Falls ja, extrahieren wir das Jahr, legen, falls noch nicht vorhanden, einen Ordner für dieses Jahr an und verschieben die Datei in diesen Ordner.

4.7 JSON-Dateien

JSON steht für *JavaScript Object Notation*. In eine JSON-Datei schreibt man also Daten so hinein, wie man ein Objekt in der Programmiersprache JavaScript schreiben würde. Und das ist ziemlich ähnlich wie die Schreibweise von Dictionarys und Listen in Python:

```
{
 "titel": "Ein Buch",
 "isbn": "978-123456789X",
 "autoren": ["Anita A.", "Bob B."],
 "verlag": { "name": "Rheinwerk Verlag", "ort": "Bonn" }
}
```

Oft wird JSON als Rückgabe von APIs verwendet. Wenn du also Daten über deine Schnittstelle aus dem Internet abrufst (das machen wir in Kapitel »APIs verwenden«), kommt oft ein String im JSON-Format zurück. Da es in diesem Kapitel aber um die Arbeit mit Dateien geht, schauen wir uns an, wie man Daten aus einer JSON-Datei liest.

Die Datei buecher.json sieht wie folgt aus:

```
[
 {"titel": "Ein Buch", "isbn": "978-123456789X", "autoren": ["Anita
A.", "Bob B."], "verlag": { "name": "Rheinwerk Verlag", "ort": "Bonn"
}},
 ...
]
```

Schreibe ein Python-Programm, das die Titel aller Bücher auflistet.

Du kannst die Datei ganz einfach mit Python einlesen und dann alle Titel ausgeben.

Hier ein Beispielprogramm:

```
import json

# Datei einlesen
with open("buecher.json", "r", encoding="utf-8") as f:
    buecher = json.load(f)

# Titel aller Bücher ausgeben
for buch in buecher:
    print(buch["titel"])
```

Zunächst wird die *buecher.json* genauso geöffnet, wie wir es schon von normalen Text- und CSV-Dateien kennen. Anschließend wandelt die Funktion `json.load` den kompletten Inhalt der Datei in eine Python-Liste um. Alles, was in den JSON-Daten enthalten ist, ist danach von einem dazu passenden Python-Datentyp:

- `buecher` – eine Liste mit allen Büchern
- `len(buecher)` – Anzahl der Bücher
- `buecher[0]` – das erste Buch als Dictionary
- `buecher[0]["titel"]` – Titel des ersten Buchs als String
- `buecher[0]["autoren"]` – Autoren des ersten Buchs als Liste von Strings
- `buecher[0]["autoren"][0]` – erster Autor des ersten Buchs als String
- `buecher[0]["verlag"]` – Verlagsinformationen des ersten Buchs als Dictionary
- `buecher[0]["verlag"]["ort"]` – Verlagsort des ersten Buchs als String

Wie kann ich mit Python ein neues Buch in die Datei einfügen?

Du kannst die JSON-Datei erst einlesen, dann ein neues Buch (Dictionary) zur Liste hinzufügen und anschließend wieder abspeichern.

Hier ein Beispiel:

```
import json

# Datei einlesen
with open("buecher.json", "r", encoding="utf-8") as f:
    buecher = json.load(f)

# Neues Buch definieren
neues_buch = {
    "titel": "Einführung in Python",
    "isbn": "978-1111111111",
    "autoren": ["Eva Beispiel"],
    "verlag": {
        "name": "Rheinwerk Verlag",
        "ort": "Bonn"
    }
}

# Neues Buch hinzufügen
buecher.append(neues_buch)
```

```
# Datei wieder abspeichern
with open("buecher.json", "w", encoding="utf-8") as f:
    json.dump(buecher, f, ensure_ascii=False, indent=4)

print("Neues Buch hinzugefügt!")
```

Die Variable `buecher` enthält nach dem Lesen der JSON-Datei die Liste aller Bücher. Mit `buecher.append` fügen wir unser neues Buch, das in der Variablen `neues_buch` steht, in diese Liste ein. Dabei ist `neues_buch` ein ganz normales Python-Dictionary, auch wenn es im Programmcode wie JSON aussieht. Erst beim Schreiben in die Datei werden dieses Dictionary und die darin befindlichen Daten mit der Funktion `json.dump` in einen JSON-String umgewandelt und in die Datei geschrieben. Die Option `ensure_ascii=False` sorgt dafür, dass Sonderzeichen unverändert in die Datei geschrieben werden. Ohne diese Angabe würde in unserem Beispiel der Titel des Buchs als `"Einf\u00fchrung in Python"` gespeichert. Die andere Option, `indent=4`, formatiert das JSON innerhalb der Datei, sodass es besser lesbar ist. In jeder Ebene werden die Daten mit vier Leerzeichen eingerückt. Ohne die `indent`-Option würden alle Daten in eine einzige lange Zeile geschrieben.

Wenn JSON-Strings in Python-Datenstrukturen umgewandelt werden, nennt man dies *Parsing* oder *Deserialisieren*. Die oben stehende Funktion `json.load` parst den Text aus der Datei und speichert die darin enthalten Daten in die Python-Liste `buecher`. Der umgekehrte Weg, Python-Daten als JSON zu formatieren, wird *Serialisieren* genannt. Dies übernimmt in unserem Programm die Funktion `json.dump`. Das Parsen funktioniert bei uns aber nur, wenn die ganze Datei in gültiger JSON-Syntax ist. Dazu muss – da wir ja mehrere Bücher in der Datei speichern – ganz am Anfang eine öffnende eckige Klammer stehen, dann kommen die einzelnen Bücher { … } mit Kommas getrennt, und am Ende steht eine schließende Klammer. Häufig hat man jedoch auch JSON-Dateien vorliegen, die etwas anders formatiert sind, wie wir im folgenden Beispiel sehen:

Bei meiner JSON-Datei steht kein `[` *am Anfang und* `]` *am Ende. In jeder Zeile ist ein Buch.*

Dann hast du keine klassische JSON-Liste ([…]), sondern eine JSONL-Datei (JSON Lines). Jeder Datensatz steht als eigenes JSON-Objekt in einer neuen Zeile.

```
# Titel aller Bücher ausgeben
import json
with open("buecher.json", "r", encoding="utf-8") as f:
    for line in f:
        buch = json.loads(line)
        print(buch["titel"])
```

```
# Neues Buch hinzufügen
neues_buch = {
    "titel": "Python Tricks",
    "isbn": "978-1111111111",
    "autoren": ["Eva Beispiel"],
    "verlag": {
        "name": "Rheinwerk Verlag",
        "ort": "Bonn"
    }
}

with open("buecher.json", "a", encoding="utf-8") as f:
    f.write(json.dumps(neues_buch, ensure_ascii=False) + "\n")
```

Nach dem Öffnen der Datei iterieren wir, wie du es vom Lesen von Textdateien kennst, mittels `for line in f:` über die einzelnen Zeilen. `line` ist ein String, in dem die JSON-Daten *eines einzigen* Buches stehen. Die Funktion `json.loads` – das steht für *load String* – parst die Zeile. Heraus kommt in unserem Fall ein Dictionary, weil die Zeile ja mit `{` beginnt und mit `}` endet.

Der Programmcode zum Hinzufügen eines neuen Buchs ist etwas simpler geworden als vorhin, da hier nicht zuerst der komplette Inhalt der Datei eingelesen werden muss. Wir können die Datei einfach im Append-Modus öffnen und eine Zeile unten anfügen. `json.dumps` – auch hier steht das `s` für String – serialisiert unser Dictionary, in dem das neue Buch steht, in einen JSON-String. Danach lässt sich dieser mit `f.write` in die Datei schreiben.

4.8 XML-Dateien

XML (Extensible Markup Language) ist genau wie JSON ein textbasiertes Format, das häufig zur Speicherung und zum Austausch von Daten verwendet wird. Beide Formate eignen sich für die Speicherung semistrukturierter Daten. Darunter versteht man Daten, bei denen das Schema in den Daten selbst steckt, was es sehr flexibel macht. Sie sind nicht unstrukturiert, anders als z. B. dieser Text. Und sie sind auch nicht strukturiert wie z. B. Tabellen in relationalen Datenbanken, bei denen zuvor ein festes Tabellenschema definiert werden muss. Ein JSON-Dokument und auch ein XML-Dokument kann also prinzipiell beliebig strukturierte Inhalte haben. Dennoch folgen bei XML die Dokumente in der Praxis oft doch einem Schema, damit man sich bei der Programmierung besser darauf einstellen kann. Dazu gibt es *XSD* (XML Schema Definition), ein Format, das beschreibt, wie ein XML-Dokument aufgebaut werden darf. Zusätzlich existieren für XML Anfrage- und Transformationssprachen,

wie XPath, XQuery und XSLT. Damit können simple und komplexe Anfragen auf XML-Daten gestellt werden, um Daten, die im Dokument stecken, abzufragen oder um ein XML-Dokument in eine andere Struktur umzuwandeln.

Ein bekanntes Anwendungsbeispiel für XML sind RSS-Feeds (Rich Site Summary). Diese werden von News-Webseiten und Podcasts zur Verfügung gestellt und enthalten beispielsweise Informationen über die neusten Artikel oder Folgen, damit andere Dienste, wie zum Beispiel News-Suchmaschinen, Benachrichtigungsservices oder Podcast-Apps, darauf zugreifen können. Ein Auszug aus dem RSS-Feed der Tagesschau:

```
<rss xmlns:content="…" xmlns:dc="…" version="2.0">
 <channel>
  <title>tagesschau.de - …</title>
  …
  <item>
   <title>Studien zur Energiewende: Erneuerbare billiger als Gas</title>
   <link>https://www.tagesschau.de/...</link>
   <description>Wirtschaftsministerin Reiche setzt in der …</description>
   <pubDate>Mon, 15 Sep 2025 12:29:40 +0200</pubDate>
   …
  </item>
  …
 </channel>
</rss>
```

Die wichtigsten Bestandteile aus XML-Dokumenten sind:

- *Elemente* starten mit einem öffnenden Tag und enden mit einem schließenden Tag. Sie können Text oder andere Elemente (Kindelemente) enthalten, z. B. `<title>Studien zur Energiewende</title>`.
- *Attribute* beschreiben Elemente genauer, im Dokument oben z. B. das `content`-Attribut `<rss xmlns:content="...">`.
- *Namespace*: Namensräume, die aus unterschiedlichen Standards stammen, um Element- und Attributnamen zu standardisieren, z. B. `xmlns:` für Allgemeines in XML-Dokumenten oder `dc:` für Dublin Core, einem Standard zur Beschreibung von Metadaten für digitale Inhalte.

Um XML-Dateien mit Python einzulesen, können wir das Modul `xml.etree.ElementTree` verwenden. Es ist Teil von Python, muss also nicht separat installiert werden. Das Modul erlaubt es, XML-Dokumente aus einer Datei oder aus anderen Quellen zu öffnen und den Inhalt zu parsen, damit wir danach auf die Texte, die in Elementen oder Attributen stehen, zugreifen können oder um über eine Liste von Elementen zu iterieren. Das folgende Programm öffnet die Datei *tagesschau.xml*, die wie oben dargestellt aufgebaut ist, und iteriert über alle

`<item>`-Elemente, die jeweils Kindelemente des `<channel>`-Elements sind. `<channel>` wiederum ist ein Kindelement des Wurzelelements (Root) `<rss>`. So bezeichnet man das oberste Element des ganzen Dokuments.

```
import xml.etree.ElementTree as ET

tree = ET.parse("tagesschau.xml")
root = tree.getroot()

print("== Neueste Meldungen ==")
for item in root.findall("./channel/item"):
    title = item.find("title").text
    print(title)
```

Die `findall`-Methode nimmt als Argument eine Anfrage in der Sprache *XPath* entgegen. XPath-Anfragen sehen so ähnlich aus wie Dateipfade, ermöglichen aber auch den Einsatz von Filterprädikaten und vielem mehr. Im gezeigten Beispiel werden alle `<item>`-Elemente durchlaufen, dann wird innerhalb der Schleife jeweils auf ihr Kindelement `<titel>` zugriffen und der darin enthaltene Text entnommen. Die Titel von News-Beiträgen, die wir auf diese Art aus der XML-Datei auslesen, werden daraufhin auf der Konsole ausgegeben.

In der Praxis bietet es sich an, das XML nicht erst lokal herunterzuladen und dann mittels `ET.parse` diese XML-Datei zu öffnen, sondern stattdessen den XML-Inhalt direkt aus dem Internet in eine String-Variable zu laden und ihn dann mittels `ET.fromstring` zu parsen. Das geht mit dem `requests`-Modul und wird in Kapitel 9, »APIs verwenden«, behandelt.

4.9 Konfigurationsdateien

Bisher war in unseren Python-Programmen immer ziemlich viel hart gecodet. Wir haben z. B. den Dateinamen der zu öffnenden Datei einfach direkt in den Programmcode geschrieben. Wenn später noch mehr Dinge dazukommen – wie zum Beispiel ein Passwort, um sich irgendwo zu verbinden, ein API-Token oder eine URL –, kann das schnell unübersichtlich werden. Das sind nämlich alles Dinge, die man eventuell in der Zukunft einmal anpassen muss. Ein erster sinnvoller Ansatz ist, solche Konfigurationen am besten ganz oben in das Python-Programm zu schreiben, damit man nicht lange danach suchen muss:

```
SPRACHE = "DE"
# ...
print("Hallo" if SPRACHE == "DE" else "Hello")
```

Häufig schreibt man solche Variablen komplett in Großbuchstaben (`SPRACHE`), um zu zeigen, dass es eigentlich nichts Variables ist, sondern eine Art Konstante, also etwas, was man einmal festlegt und danach nur noch ausliest.

Noch eleganter kann es aber vor allem bei komplexen Programmen sein, solche Konfigurationen nicht in den Programmcode selbst, sondern in eine separate Datei zu speichern, beispielsweise in eine *config.json*:

```
{ "sprache": "DE", "ordner": "C:/Users/Kai/Documents/" }
```

Im eigentlichen Python-Programm wird diese Konfigurationsdatei zunächst eingelesen, danach lässt sich auf die entsprechenden Werte zugreifen:

```
import json
with open("config.json", "r") as f:
    config = json.load(f)
print("Hallo" if config["sprache"] == "DE" else "Hello")
```

Ein anderer Grund, wieso Konfigurationsdateien sinnvoll sein können: Wir können auch in die Datei hineinschreiben und uns somit etwas für die nächste Programmausführung merken. Es wäre doch komfortabel, wenn unser Programm beim nächsten Mal direkt in dem gleichen Ordner nach Dateien sucht, in dem wir zuletzt gearbeitet haben. Für solche Fälle können wir einfach den aktuell ausgewählten Ordner in die Konfigurationsdatei zurückschreiben:

```
print(f"Aktueller Ordner: {config["ordner"]}")
ordner = input("Anderen Ordnerpfad eingeben oder Enter drücken: ")
if ordner == "":
    ordner == config["ordner"]
else:
    config["ordner"] == ordner
    with open("config.json", "w") as f:
        json.dump(config, f, indent=4)
```

Gibt man nichts ein und drückt einfach ↵, steht in der Variablen `ordner` der Wert, der aus der Konfigurationsdatei gelesen wird. Gibt man stattdessen aber etwas ein, wird dieser Wert in die Datei geschrieben, sodass man den eingegebenen Ordnerpfad beim nächsten Programmaufruf auslesen kann.

Die gängigsten Dateiformate für Konfigurationsdateien sind neben JSON auch INI und YAML. Für all diese Formate gibt es entsprechende Python-Module, z. B. das `configparser`-Modul für INI-Dateien. Die Verwendung dieser Module ist sehr ähnlich wie bei JSON.

In Abschnitt 9.1, »API-Abfragen mit ›requests‹«, werden wir das `configparser`-Modul verwenden. In Abschnitt 9.4, »ChatGPT-API«, folgt ein Beispiel, in dem wir Konfigurationen in einer Datei mit Umgebungsvariablen speichern.

Inhalt

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Python für KI- und Daten-Projekte

Pythonprogrammierung und Datenanalyse – so geht's! In diesem Buch lernst du die Programmiersprache anhand von Beispielprojekten kennen, die sich anschließend leicht auf deine eigenen Daten übertragen lassen. So kannst du ohne Vorkenntnisse einsteigen.

Datenanalyse für Einsteiger

Ob für Uni, Wissenschaft oder Büro: Python ist die perfekte Programmiersprache für die Datenanalyse. Zusätzlich helfen KI-Algorithmen bei der Text- und Bildanalyse.

Dein Python- und KI-Werkzeugkasten

Johannes Schildgen zeigt dir wichtige Tools und Methoden, die du im konkreten Anwendungsfall einsetzen kannst. So hast du immer das richtige Werkzeug an der Hand.

Alle Daten auf einen Blick

Von der Visualisierung als Diagramm oder Graph bis hin zur Automatisierung lästiger Routineaufgaben – hier lernst du, wie du deinen Umgang mit Daten einfach optimieren kannst.

Mit Datenmaterial und allen Beispielprojekten zum Download

Johannes Schildgen ist Professor für Datenbanken mit dem Schwerpunkt Big Data und vermittelt Studierenden u. a. die Programmiersprache Python. In seinem Arbeitsalltag gibt er unterhaltsam und verständlich sein Wissen über KI und andere IT-Themen weiter.

Loslegen mit Python

+ Visual Studio Code
+ Jupyter Notebooks
+ Variablen und Datentypen
+ Module importieren und nutzen
+ Python im Web einsetzen
+ Routineaufgaben automatisieren

Mit Daten arbeiten

+ CSV, Excel, Word, PDF und Bilddateien
+ Diagramme erstellen
+ Daten bereinigen und analysieren
+ APIs und Datenbanken nutzen

KI in Aktion

+ Programmieren mit der KI
+ Prompt Engineering
+ Text- und Bildanalyse
+ Die wichtigsten KI-Algorithmen
+ Regression, Klassifikation und Clustering

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 29,90 [D] € 30,80 [A]
Programmierung/KI
ISBN 978-3-367-11151-0