Wirtschafts-
wissenschaftliche
Fakultät

Informationsverarbeitungs-
Versorgungseinheit (IVV)
Wirtschaftswissenschaften

Walter Schmitting

Schlüsselqualifikationsveranstaltung im Bachelorstudium der Wirtschaftswissenschaften

„Programmierung mit Python“

Booklet zum Modul 04:
Grundlagen Python

Version 1.25 / 06.03.2024

Inhalt

4. Grundlagen Python

Vorbemerkungen

Mit diesem Booklet wird versucht, ein Bedürfnis der Teilnehmer/innen der Schlüsselqualifikationsveranstaltung „Programmierung mit Python“ zu decken: Gewünscht wurde eine Möglichkeit, die Darstellung im Rahmen der Einführung in die Grundlagen noch einmal „in Ruhe“ nachvollziehen zu können. Hintergrund ist der Umstand, dass man in der Veranstaltung die jeweiligen Operationen zwar auf dem Dozentenrechner über den Beamer „sieht“, am eigenen Rechner „nachbaut“, aber wenig Zeit verbleibt, umfassende Notizen dazu anzufertigen. An dieser Stelle werden die Beispiele noch einmal im Detail erklärt und der Text der „Tonspur“ der Veranstaltung wiederholt. Last not least bleibt am rechten Rand Platz für Notizen.

Im Folgenden wird die Nutzung von Python in der Version 3.12 unterstellt – obgleich die meisten hier besprochenen Operationen so grundlegend sind, dass sie mit nahezu jeder Python-Version 3 realisierbar sein sollten. Als IDE („Integrated Development Environment“, integrierte Entwicklungsumgebung) wird die Software PyCharm der Firma JetBrains in der entgeltfrei verwendbaren Community Edition (aktueller Stand: Version 2023.3.3) verwendet.

Natürlich gibt es eine Vielzahl an „Einführungen in Python“ am Markt (oder für die Studierenden: in der Universitäts- und Landesbibliothek). Daher kann und soll hier nicht das Rad neu erfunden werden: Die Inhalte finden sich – vermutlich in großer Ähnlichkeit – auch in anderen Darstellungen zum Thema wieder. Da hier frei formuliert wird, unterbleibt – außer bei expliziten Übernahmen, so z.B. von Abbildungen – jedoch ein Quellennachweis. Dies ist keine wissenschaftliche Arbeit.

An einigen Stellen werden Vereinfachungen vorgenommen oder Aspekte vernachlässigt, die die Ausführungen bei einer generellen Betrachtung als nicht ganz korrekt erscheinen lassen. Diese Verkürzungen werden aus didaktischen Gründen in Kauf genommen, um die Übersicht auf die für Anfänger/innen wesentlichen Punkte zu konzentrieren.

Darstellung

Im Folgenden wird Python-Programmcode stets in der Schriftart „Consolas“ in roter Farbe dargestellt:

```
print("Hello World!")
```

Ausgaben auf der Konsole von PyCharm werden hier gleichfalls in dieser Schriftart – allerdings in blauer Farbe – ausgewiesen.

Eine sehr kurze Einführung in PyCharm …

In der Veranstaltung findet die Einführung in die IDE („Integrated Development Environment – Integrierte Entwicklungsumgebung) im Modul 03 statt. Für diejenigen unter den Teilnehmer/innen, die schon *vor* der Veranstaltung mit diesem Booklet arbeiten wollen, folgt im Weiteren eine sich auf das Essentielle beschränkende Einführung in PyCharm.

Beginnen wir mit der Frage, was denn eine solche „IDE“ nun überhaupt ist? Kurz gefasst ist es eine Sammlung und Integration der wichtigsten Werkzeuge für die Softwareentwicklung unter einer Oberfläche bzw. in einem Programm. Man könnte ein Python-Programm auch sehr spartanisch unter einem Editor (z.B. Notepad) oder sogar unter Word schreiben – da gäbe es dann aber keine Quelltextformatierung, keine Syntaxhilfe, keine Verwaltung virtueller Umgebungen, keine Paketverwaltung, keinen Debugger … all die Hilfsprogramme, die zur Entwicklung notwendig sind, würden dann fehlen. Kurz gesagt dient eine IDE dazu, uns die Konzentration auf das Wesentliche zu ermöglichen: Das Coden selbst – und zwar „mit allem Komfort“.

Für Python gibt es etliche IDEs – hier wurde PyCharm ausgewählt, da es sehr anfängerfreundlich ist. PyCharm ist ein kommerzielles Produkt der Firma JetBrains. Es gibt die hier verwendete „Community Edition“, die entgeltfrei nutzbar ist, und eine „Professional Edition“, die bezahlt werden muss. PyCharm steht unter Linux, MacOS und auch Windows zur Verfügung.

Nach der Installation von Python (nicht zu vergessen (!), siehe dazu diverse Anleitungen im Web …) und PyCharm auf Ihrem Rechner (in den Pools ist die Software bereits installiert) wird man beim ersten Start stets aufgefordert, zunächst ein Projekt anzulegen. Für die Arbeit mit diesem Booklet sollten Sie i.d.S. das Projekt „Grundlagen“ anlegen. Es erscheint beim Start zunächst das in Abb. 1 wiedergegebene „Begrüßungsfenster“. Hier ist der Button „New Project“ (oben, rechte Seite) zu betätigen.

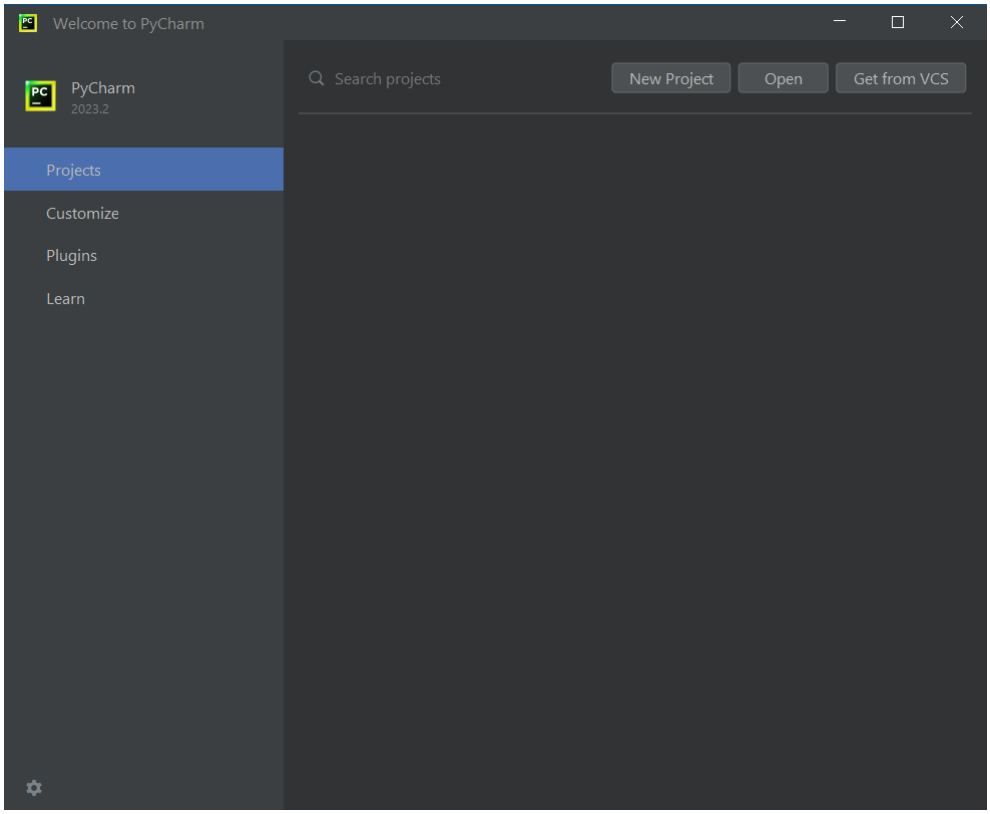

Abb. 1: PyCharm-Begrüßungsfenster

Es öffnet sich daraufhin das Fenster „New Project“, welches sich hier in Abb. 2 wiederfindet. In diesem wird ganz oben der Projektname angegeben (hier: „Grundlagen“) Darunter wird als „Location“ der Ordner für das Projekt im Dateisystem festgelegt. Dieser lautet hier „D:\Data\PyPrg\Projekte\Python_Kurs\“. Der Projektordner wird als „Grundlagen“ unter diesem Pfad angelegt. Alle Projektdateien sind folglich immer in einem Ordner platziert.

Sofern ein kleines „Rumpfprogramm“, hier als „welcome script“ bezeichnet, generiert werden soll, ist ganz unten noch ein Haken zu setzen. Darauf kann hier allerdings (genau wie auf eine Git-Anbindung) verzichtet werden.

Darunter wird die IDE unter dem Reiter „Custom environment“ angewiesen, für dieses Projekt auch eine virtuelle Umgebung („Virtual Environment“) anzulegen, Der Sinn und Zweck virtueller Umgebungen wird im Modul 03 erläutert. Sehr kurz gefasst: Module, Packages oder Pakete, welche zusätzliche Funktionalitäten über das Basis-Python hinaus für eigene Programme bereitstellen, möchte man i.d.R. projektweise verwalten und nicht pauschal der Basisinstallation von Python

hinzufügen. Dies gelingt, indem für Projekte jeweils eigene virtuelle Umgebungen angelegt werden, in welchen die Module dann verwaltet werden. Diese virtuelle Umgebung soll hier mit im oben gewählten Projektordner platziert werden. Alternativ könnte die virtuelle Umgebung unter Windows auch im Benutzerordner des Anwenders abgelegt werden.

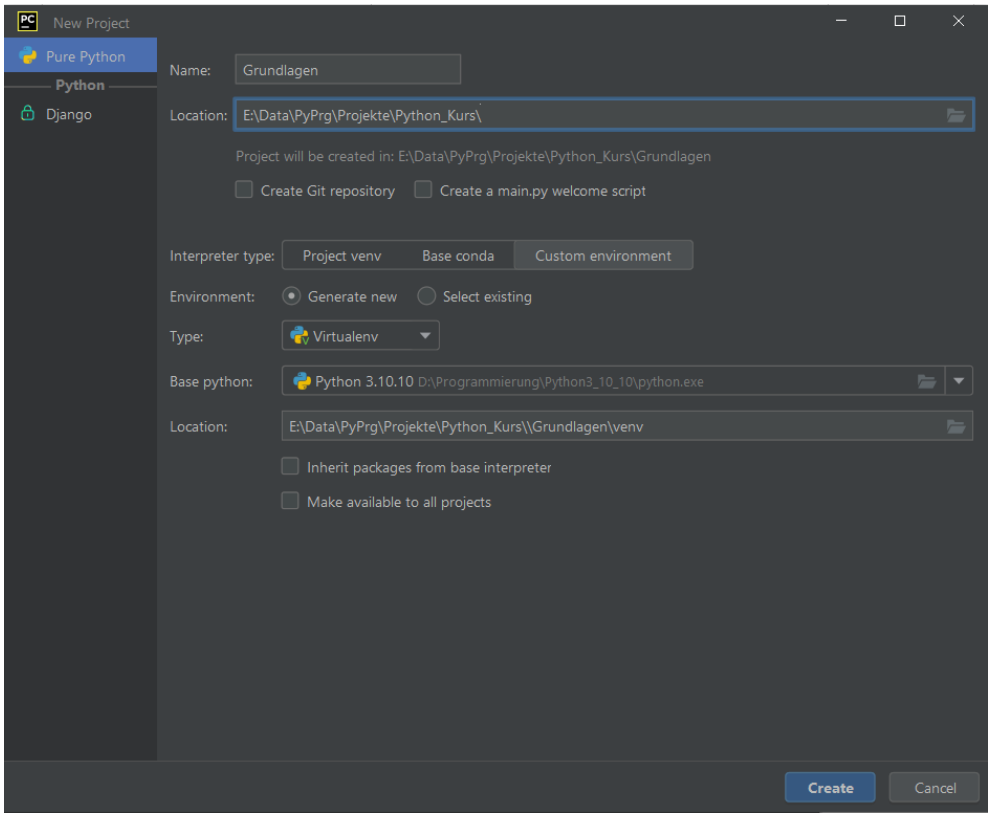

Abb. 2: PyCharm – Fenster „New Project“

Nach dem Klick auf den Button „Create“ unten rechts öffnet sich das Hauptfenster von Python. In Abb. 3 ist dieses dargestellt – allerdings wurde hier bereits eine Zeile Programmtext eingegeben und es existieren schon eine Python-Datei namens „Grundlagen.py“ im Projektordner. Sehen wir uns zunächst den Aufbau und die Bereiche des Hauptfensters an.

Oben findet sich die Menüleiste mit den Drop-Down-Menüs. Auf einzelne Menüpunkte wird folgend eingegangen, sobald diese benötigt werden. Darunter ist auf der linken Seite der Projektordner samt der in ihm befindlichen Dateien und Unterordner sichtbar. Unter dem Ordner „venv“ finden sich z.B. die Dateien der virtuellen Umgebung. Rechts

davon ist der Bereich platziert, in welchem der Code eingegeben und bearbeitet wird. Dort wurde bereits eine Programmzeile eingegeben:

```
print("Hello World!")
```

Der eingegebene Code stellt bereits ein lauffähiges Programm dar. Klickt man auf das das kleine grüne Dreieck im oberen rechten Teil des Fensters, so wird er ausgeführt. Es öffnet sich unten (wie in Abb. 3 zu sehen) ein Ausgabefenster, in welchem die Ausgabe …

```
"Hello World"
```

… erscheint. Darüber wird im Ausgabefenster immer die ausgeführte Codedatei angegeben, darunter an dieser Stelle eine fehlerfreie Ausführung bestätigt („ … exit code 0“).

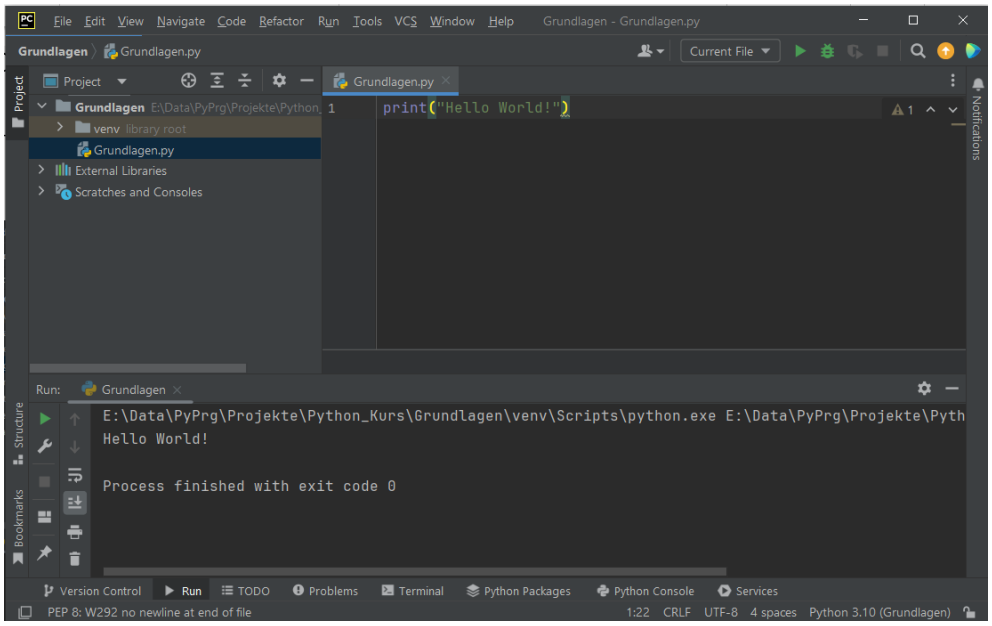

Abb. 3: PyCharm – Hauptfenster

In Abb. 3 existiert die Python-Datei „Grundlagen.py“ bereits. Ihre Anlage wird im Modul 03 erläutert – in der Kurzfassung: Im Menü wird „File“ – „New“ – „Python File“ gewählt, anschließend der Namen eingetragen und schließlich die Anlage bestätigt. In dieser Datei wird im Weiteren gearbeitet. Nach Änderungen speichert sich die Datei selbständig – mit der Tastenkombination „Strg+S“ können jedoch auch jederzeit alle gerade offenen Dateien gespeichert werden. Geöffnet wird eine Python-Datei durch einen Doppelklick auf den Dateinamen im Projektordnerfenster links.

4.1 Variablen und Datentypen

Variablen

Bevor nun jedoch die ersten komplexeren Codezeilen eingegeben werden können, ist die Idee der „Variablen“ zu klären. Eine Variable kann in ihrer Natur als Behältnis für eine Information umschrieben werden. Sofern z.B. eine im Rahmen des zu schreibenden Programms relevante Information der Vorname „Peter“ (eine Kette von Buchstaben, also Zeichen) sei, sollte dieser auch im Programmablauf gespeichert und verwendet werden. Dafür wird eine Variable als Behältnis verwendet. Damit man diese Variable später wieder ansprechen und verwenden kann, benötigt sie einen Namen. Dieser könnte z.B. „vorname“ lauten. Unter Python wird die Zuweisung des Variableninhalts „Peter“ auf den Variablennamen „vorname“ mit einem Gleichheitszeichen vorgenommen:

```
vorname = "Peter"
```

Um zu prüfen, ob unter dem Variablennamen tatsächlich der gewünschte Inhalt gespeichert wurde, kann man diesen mit der Print-Funktion wieder ausgeben lassen:

```
print(vorname)
```

Lässt man dieses kleine Programm nun „laufen“ (indem man, wie in Modul 03 erläutert, den Button mit dem kleinen grünen Dreieck oben rechts im Hauptfenster von PyCharm betätigt oder Umschalt + F10 auf der Tastatur drückt – dabei sollte links vom Button „Current File“ ausgewählt sein), so erscheint zunächst – sofern noch nicht eingeblendet – das Fenster der Ausgabekonsole von PyCharm. In diesem wird dann angedruckt:

```
C:\Users\...\python.exe D:\Data\...\grundlagen.py
Peter

Process finished with exit code 0
```

Oben wird in der Konsole zunächst immer angegeben, welcher Code (Dateipfad und Dateiname) gerade ausgeführt wird. Die Angabe wurde hier durch Punkte ein wenig verkürzt, um noch in die Zeile zu passen. Sofern das Programm fehlerfrei ausgeführt bzw. beendet wird, erscheint die letzte Zeile „Process finished with exit code 0“. Bei der zukünftigen Angabe von Konsolenausgaben in diesem Text werden das

erste (Dateipfad/Datei) und das letzte (Meldung fehlerfreier Ausführung) Element zukünftig weggelassen.

Ähnlich kann auch mit Zahlen verfahren werden. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = 10
print(a)
```

In der Konsole wird nun die „10“ ausgegeben, wenn das Programm ausgeführt wird. Der Unterschied zum vorherigen Beispiel besteht darin, dass die Zahl 10 ohne führende und schließende Anführungszeichen dem Variablennamen „a“ gleichgesetzt wird. Daraus – wie auch aus der Verwendung von Zahlenzeichen – schließt Python, dass hier eine Zahl übergeben wurde. Mit dieser Zahl – und anderen in Variablen gespeicherten Zahlen – kann nun auch gerechnet werden. Die nachstehenden Codezeilen werden ergänzt:

```
a = a * 2
b = 4
c = a + b
print(a)
print(c)
```

Die Ausgabe in der Konsole lautet nun bei Ausführung des Programms:

```
10
20
24
```

Mit Zahlen – in der Verkörperung durch Variablen – kann man also auch rechnen.

Datentypen

Dies ist nun der Punkt, an welchem wir uns mit den Datentypen der Variablen auseinandersetzen müssen. Sofern wir eine Variable als ein Behältnis für eine Information ansehen, besteht offenbar ein Unterschied zwischen Zeichenketten wie „Peter“ und Zahlen wie der 10 (bewusst ohne Anführungsstriche!). Mit Zahlen will man i.d.R. auch rechnen. Mit Zeichenketten will man gleichfalls spezifische Operationen durchführen – z.B. alle Buchstaben der Zeichenkette in der Variablen zu Groß- oder Kleinbuchstaben wandeln oder verschiedene Zeichenketten aneinanderfügen (also wieder verketten).

Für diese Operationen müssen die Variablen als Behältnisse jedoch die notwendige Eignung aufweisen. Man muss also nicht nur einen Variablennamen zuweisen, sondern auch den Typ der Variablen festlegen. Daher hat jede Variable „ihren“ Datentyp. Es wird z.B. grundsätzlich zwischen Text/Zeichenketten und Zahlen unterschieden. Vom Datentyp der Variablen hängt ab, welche Operationen anschließend mit den Variablen durchgeführt werden können.

Die für die Grundlagen wesentlichen Datentypen unter Python (es gibt noch mehr … die hier erst einmal vernachlässigt werden …) sind:

- Strings (str): Text/Zeichenketten
- Boolesche Werte (bool), also Wahrheitswerte wie Ja/Nein
- Zahlen:
  - Ganzzahlen (int), Zahlen ohne Nachkommastellen
  - Gleitkommazahlen (float), Zahlen mit Nachkommastellen

Zu den Datentypen für Python-Variablen, speziell zu Ganz- und Gleitkommazahlen wäre noch vieles mehr wissenswert – z.B., wie groß (oder klein) die Zahlen werden dürfen, wie viele Nachkommastellen darstellbar sind ... und etliches andere. Soweit notwendig kommt die Darstellung später darauf zurück. Die Booleschen Werte (Wahrheitswerte) werden im Abschnitt 4.3 behandelt.

Der Datentyp einer Variablen muss bei etlichen anderen Programmiersprachen vor der ersten Verwendung der Variablen explizit im Code mitgeteilt werden. Das nennt sich „Typisierung“. Man muss die Variable mit dem Datentyp anmelden.

Bei Python ist das nicht so. Python entscheidet automatisch über den Datentyp – wenn der Variablen eine Zahl ohne Nachkommastellen zugewiesen wird, wird der Datentyp Ganzzahl (int) zugeordnet. In einem erweiterten Kontext wird diese Vorgehensweise als „Duck-Typing“ beschrieben – wenn ein Vogel schnattert, läuft und fliegt wie eine Ente, dann nennt man ihn eben „Ente“.

Dazu ein Beispiel … Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = 10
b = "10"
```

In der ersten Zeile wird die Zahl 10 auf die Variable „a“ zugewiesen. Python wird den Datentyp „int“ wählen. In der zweiten Zeile wird die

„10“ in Anführungsstrichen – also als Text – auf die Variable „b“ zugwiesen. Python wählt den Datentyp „str“. Mit der Variablen „b“ können wir nicht mehr direkt rechnen bzw. Rechenoperationen zeigen evtl. unerwartete Ergebnisse:

```
b = b * 2
print(b)
```

Nach der Ausführung des Programms erscheint in der Konsole die Ausgabe:

```
1010
```

Statt die Zahl 10 mit 2 zu multiplizieren (Ergebnis 20), hat Python den String „10“ zweimal hintereinander gehängt.

Die automatische Zuweisung des Datentyps zu Variablen ohne explizite Typisierung unter Python ist einerseits arbeitssparend und beschleunigt die Entwicklung – das ist die Lichtseite. Die Schattenseite wird bei der Fehlersuche (dem „Debugging“) in Programmen deutlich: Ohne explizite Typisierung kann es zur Laufzeit zu unerwarteten automatischen Typisierungen kommen, die nicht verarbeitet werden können und zu Fehlern führen. Diese sind zuweilen schwer zu identifizieren.

Den Typ einer Variablen kann man unter Python jederzeit mit der Funktion „type()“ abfragen. In Fortsetzung des obigen Codes wird ergänzt:

```
print(type(a))
print(type(b))
```

Bei Ausführung des Programms erscheint nun auf der Konsole:

```
1010
<class 'int'>
<class 'str'>
```

Die Variable „a” wurde also erwartungsgemäß als Ganzzahl („int“) typisiert, die Variable „b“ als Zeichenkette („str“).

Unter Python ist allerdings – sofern gewünscht! – auch ein expliziter Hinweis auf die Typisierung („Type Hint“) möglich. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a :str = "1"
print(type(a))
```

In diesem Fall wird mittels des „:str“ hinter dem Variablennamen ein Hinweis auf den intendierten Typ gegeben. Eine statische Typisierung bewirkt dies jedoch nicht. Es besteht allerdings in Verbindung mit Zusatzmodulen wie z.B. „mypy“ die Möglichkeit, gezielt mögliche Fehler in der Typisierung zu identifizieren.

Der Datentyp einer Variablen ist nach einer (automatischen) Typisierung keinesfalls dauerhaft festgelegt. Dazu folgt ein Beispiel – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = 3
print(a)
print(type(a))
a = 4.4
print(a)
print(type(a))
```

In diesem Code wird zunächst eine Ganzkommazahl auf die Variable „a“ zugewiesen. Führt man das Programm aus, so wird auch der Variablentyp als „int“ ausgewiesen. Im weiteren Ablauf wird dann allerdings in der vierten Zeile eine Ganzkommazahl auf die gleiche Variable zugewiesen. Die Ausgabe in der Konsole zeigt, dass sich dann auch der Datentyp der Variablen entsprechend anpasst und zu „float“ wechselt (die Konsolenausgaben werden hier nicht noch einmal wiedergegeben).

Ein anderes Beispiel dazu – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = 3
print(type(a))
a = a/2
print(a)
print(type(a))
```

Auf der Konsole erzeugt dieses Programm bei der Ausführung die nachstehende Ausgabe:

```
<class 'int'>
1.5
<class 'float'>
```

Der Variablen „a” wird zunächst der Wert 3 zugewiesen. Damit wird sie automatisch als Ganzzahl („int“) typisiert. Dann wird der Variableninhalt durch 2 geteilt und wiederum auf die Variable „a“ neu zuge-

wiesen. Das Ergebnis der Division ist eine Gleitkommazahl – entsprechend passt Python den Datentyp zu „float“ an.

Bei der Wahl der Variablennamen sind unter Python einige Regeln zu beachten (ähnlich wie bei vielen anderen Programmiersprachen):

- Variablennamen müssen stets mit einem Buchstaben oder einem Unterstrich beginnen.
- Erlaubte Zeichen in Variablennamen sind nur Buchstaben, Zahlen und Unterstriche. Sonderzeichen sind unzulässig.

Wichtig ist des Weiteren, dass Groß- und Kleinschreibung bei Variablennamen unterschieden werden. Diese Konvention wurde aus der Linux-Welt (bzw. ursprünglich Unix-Welt) übernommen. Dies bedeutet, dass „a“ eine andere Variable ist als „A“ und dass auch „alpha“ und „Alpha“ zwei getrennte Variablen darstellen.

Nach dem Styleguide „Python Enhancement Proposals“ in der Version 8 (kurz „pep8“) wird empfohlen, Variablennamen immer durchgängig klein zu schreiben.

Funktionen (Exkurs)

Mit einem sehr knappen Exkurs sollen hier die „Funktionen“ eingeführt werden – hauptsächlich, da sie hier schon einige Zeit schamlos genutzt werden, ohne sie überhaupt vorzustellen … dies wird jetzt nachgeholt!

Jede Person mit Grundkenntnissen in Excel kennt die in eine Zelle zu schreibende Funktion „=Summe(A2:A4)“. Der Funktionsname ist „Summe“. Der Funktion werden in den Klammern Argumente übergeben. Bei der „Summe()“ ist es der zu summierende Zellbereich. Die Funktion hat eine Rückgabe (in die Zelle, in welche sie geschrieben wurde). Diese Rückgabe ist die Summe der Zahlen im angegebenen Zellbereich.

Typisch für Funktionen sind also äußerlich zunächst einmal die Klammern, in welchen eines oder mehrere Argumente übergeben werden können (aber nicht müssen). Die Funktion „Jetzt()“ unter Excel z.B. bleibt ohne Argumente, weist aber als Funktion die funktionstypischen Klammern auf. Typisch ist des Weiteren, dass eine Funktion bei ihrem Aufruf irgendetwas für uns erledigt – was, hängt von der konkreten Funktion ab. Die Funktion „Jetzt()“ unter Excel gibt z.B. das aktuelle

Datum und die aktuelle Uhrzeit in der Zelle zurück, aus der sie aufgerufen wurde.

Funktionen werden wir unter Python immer wieder begegnen. Im Rahmen der Veranstaltung werden die Teilnehmer/innen auch noch eigene Funktionen schreiben. Verwendet wurden bislang hier die Funktionen „print()“ (die den als Argument übergebenen Text auf die Konsole ausgibt) sowie die Funktion „type()“, die den Datentyp einer als Argument übergebenen Variablen zurückgibt.

Datentypen wandeln

Es wurde oben schon ein Beispiel gezeigt, in welchem Python den Datentyp einer Variablen selbständig nach einer Rechenoperation anpasst. Die Konvertierung zwischen Datentypen kann jedoch auch im Programmcode erzwungen werden.

Dazu ein Beispiel – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = 4
print(type(a))
a = float(a)
print(a)
print(type(a))
a = str(a)
print(a)
print(type(a))
```

Hier wird zunächst der Wert 4 auf die Variable „a“ zugewiesen. Die Konsolenausgabe (die hier nicht wiedergegeben wird) zeigt, dass Python dieser Variable erwartungsgemäß den Datentyp „int“ zuweist. Mittels der Funktion „float()“ wird sodann in Zeile 3 des Codes der Variablen dieser der Typ „float“ (Gleichkommazahl) zugewiesen. Die Ausgabe in der Konsole in Zeile 4 ergibt sich zu 4.0. Es wird also für die Gleichkommazahl auch eine Nachkommastelle ausgegeben. Abschließend wird die Variable „a“ mittels der Funktion „str()“ in eine Zeichenkette gewandelt. Die Ausgabe der Variablen erhält die ausgegebene Nachkommastelle und zeigt sich wiederum als 4.0.

Pop-Up-Exkurs: Kommentare im Code

Abschließend noch ein kleiner Exkurs zum Kommentieren von Programmen. Kommentare sind jegliche Erläuterungen, die im Programmcode stehen, aber nicht (mit) ausgeführt werden.

Technisch beginnen Kommentarzeilen unter Python immer mit einer Raute (synonym Hashtag). Darüber hinaus gibt es noch den Inline-Kommentar, der mit ausführbarem Code in einer Zeile steht.

Dazu ein Beispiel – es lohnt sich nicht, das unter Python einzutippen, ein aufmerksamer Blick genügt:

```
# Dies ist eine Kommentarzeile
a = 2
print(a)   # Dies ist ein Inline-Kommentar
```

Kommentare werden regelmäßig unterschätzt – „man weiß ja, was man tut, andere müssen es nicht verstehen.“. Drei Monate später weiß man es selbst oft allerdings nicht mehr, geschweige denn, warum man es getan hat und wie dieser geniale Code überhaupt funktionieren kann (je genialer, desto weniger!) … und falls dann doch eine andere Person dieses Machwerk überarbeiten muss, sind Hopfen und Malz verloren. Gängiger Kommentar: „Das musste ich vollständig neu schreiben, um es zu verstehen … und erst danach konnte ich die Änderungen vornehmen!“

Kurzum: Man sollte (für sich selbst und andere) großzügig kommentieren. Zumindest sollte man für Codeabschnitte in Kommentaren festhalten, welches ihre Aufgabe ist und wie ungefähr sie diese Aufgabe bewältigen. Weitergehend sollte man alles Wissen mit Kommentaren im Code dokumentieren, welches sich nicht unmittelbar aus dem Code ergibt.

4.2 Mathematische Operatoren

Die mathematischen Operatoren unter Python bieten auch für den Anfänger/innen in der Programmierung wenig Überraschungen. Tab. 1 gibt einen Überblick über die einfachen mathematischen Operatoren.

Operator	Bedeutung	Beispiel	Ergebnis
+	Addition	3 + 8	11
-	Subtraktion	14 - 7	7
*	Produkt	3 * 9	27
/	Division	12 / 5	2.4
//	Division abgerundet	12 // 5	2
%	Modulo (Rest)	12 % 7	5
**	Exponent	4 ** 4	256

Tab. 1: Einfache mathematische Operatoren unter Python (ähnlich bei https://digitaleprofis.de)

Etwas ungewöhnlich ist die Darstellung der Exponentiation bzw. Potenz: Aus vielen anderen Programmiersprachen oder auch Excel kennt man diese mit einem Zirkumflex, also z.B. als 4^4 = 256. Unter Python wird das Zirkumflex durch zwei aufeinander folgende Sternchen ersetzt.

Die Reihenfolge, in welcher zu rechnen ist (präziser: Operatorrangfolge bzw. Präzedenz) folgt gleichfalls dem gängigen Schema: Die Potenzrechnung (mit Exponenten) geht vor Punktrechnung (Division und Multiplikation), die Punktrechnung geht vor Strichrechnung (Addition und Subtraktion). Soll die Berechnungsfolge in spezifischen Fällen anders gestaltet werden, so sind Klammern zu verwenden.

4.3 Boolesche Werte

Boolesche (logische) Werte bzw. Wahrheitswerte sind wahr und falsch. Unter Python entspricht dies True und False. Es ist zu beachten, dass diese im Programmcode mit einem Großbuchstaben beginnen müssen.

Boolesche (logische) Operatoren sind das „und“, „oder“ und „nicht“. Unter Python entspricht dies „and“, „or“ sowie „not“. Diese werden im Programmcode sämtlich durchgängig klein geschrieben.

Die konkrete Handhabung soll gleich anhand von Beispielen demonstriert werden. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
print(3==4)
```

Während Zuweisungen von Werten auf Variablen unter Python mit einem einfachen Gleichheitszeichen vorgenommen werden, ist das doppelte Gleichheitszeichen der Vergleichsoperator. Da die Zahl 3 ungleich der Zahl 4 ist, ist das in der Konsole ausgegebene Ergebnis bei Ausführung des Codes ein False.

Als nächstes wird der Vergleichsoperator für Ungleichheit eingeführt. Wir ändern den obigen Code zu:

```
print(3!=4)
```

Das Ausrufungszeichen gefolgt von einem Gleichheitszeichen ist der Vergleichsoperator für Ungleichheit. Das Ergebnis ist erwartungsgemäß bei Ausführung des Codes ein True, da die Zahl 3 ungleich der Zahl 4 ist.

Die Operatoren für kleiner, größer, kleiner gleich und größer gleich entsprechen der üblichen Notierung mit „<“, „>“, „<=“ und „>=“. Ein Beispiel dazu mag genügen – wir ändern den obigen Code zu:

```
print(3>4)
```

Da die getroffene Aussage falsch ist (die Zahl 3 ist nicht größer als die Zahl 4, sondern umgekehrt!), ist die Ausgabe in der Konsole erwartungsgemäß False.

Als erster Boolescher Operator sei das „nicht“ eingeführt. Auch dazu ein Beispiel – wir ändern den bislang eingetragenen Code zu:

```
print(not 3!=4)
```

Das „not“ repräsentiert die logische Verneinung. Der Ausdruck (3!=4) wäre eigentlich wahr (True). Der „not“-Operator dreht das Ergebnis zu False, welches dann auch in der Konsole bei Ausführung des Codes ausgegeben wird.

Das logische „or“ und „and“ ermöglicht dann viele „Logeleien“ als Spielereien mit der Logik, die besonders für Klausuraufgaben beliebt sind. Ein Beispiel – wir ändern den derzeit eingetragenen Code zu:

```
print(True or False)
```

Das bei Ausführung des Codes in der Konsole ausgegebene Ergebnis ist ein True, da die Aussage „True or False“ aufgrund des logischen Oder immer wahr ist. Hingegen würde der Ausdruck „True and False“ immer zu einem False führen, da ein Ausdruck niemals gleichzeitig wahr und falsch sein kann.

Die hier mehrfach durchgeführten Prüfungen von Aussagen auf ihren Wahrheitsgehalt (mit True oder False als Ergebnis) nennt man boolesche Ausdrücke (boolean expressions). Ein boolescher Ausdruck gibt immer ein True oder ein False zurück.

Auch die booleschen Operatoren sind unter Python mit einer Rangfolge belegt. Der stärkste Operator ist das „not“, gefolgt vom „and“ und dann vom „or“. Dazu mag ein Beispiel genügen … wir ändern den bislang eingetragenen Code zu:

```
print(True and False or True)
```

In diesem Falle bindet das „and“ zunächst stärker als das „or“, so dass zunächst der Teilausdruck „True and False“ ausgewertet wird. Das Ergebnis ist – wie schon oben erläutert – ein False. Damit verkürzt sich der Ausdruck auf „False or True“. Auch dazu wurde schon oben erläutert, dass das Ergebnis ein True ist. Dies wird dann auch bei Ausführung des Codes in der Konsole ausgegeben.

Will man andere Rangfolgen der booleschen Operatoren realisieren, so muss wiederum mit Klammern gearbeitet werden.

4.4 Strings

Strings – deutsch: „Zeichenfolgen“ – stehen bei der Zuweisung unter Python in doppelten (“ “) oder einfachen (‘ ’) Anführungsstrichen. Sie können durch ein „+“ (Plus) verkettet sowie – wie schon gezeigt – durch ein „*“ (Sternchen) wiederholt werden.

Dazu ein Beispiel – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = "alpha"
b = ‘beta’
c = a + b
print(c)
print(a*2)
```

Die Konsolenausgabe lautet:

```
alphabeta
alphaalpha
```

Die Anführungszeichen – einfach oder doppelt – dienen hier im Sinne von Steuerungszeichen dazu, Strings bei der Zuweisung zu kennzeichnen. Problematisch kann es werden, wenn Anführungszeichen auch mit als normaler Text ausgegeben werden sollen. Unproblematisch ist noch das folgende erste Beispiel – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
print('Sie nannten ihn "Hombre".')
```

Bei der Ausführung des Codes erscheint auf der Konsole:

```
Sie nannten ihn "Hombre".
```

So weit, so gut. Wie sieht es nun aber aus, wenn wir den obigen Code verändern zu:

```
print("Sie nannten ihn "Hombre".")
```

Daraus resultiert auf der Konsolenausgabe eine Fehlermeldung (i.d.R. bei PyCharm in roter Schrift):

```
  File "D:\...\grundlagen.py", line 1
    print("Sie nannten ihn "Hombre".")
          ^^^^^^^^^^^^^^^^^^^^^^^^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Der Pfad bei der Angabe der Code-Datei („File“) wurde hier mit Punkten ein wenig gekürzt, um die Ausgabe auf eine Zeile zu bringen.

Fehlermeldungen sind im Allgemeinen zunächst ärgerlich – ein fehlerfreier Code ist natürlich schöner als ein fehlerbehafteter. Fehlermeldungen geben jedoch auch meist klare Hinweise auf die Natur des Fehlers und helfen, ihn zu beseitigen. Hier wird der problematische Codeabschnitt in der Fehlermeldung wiedergegeben – damit wissen wir schon, wo das Problem (vermutlich!) liegt. PyCharm vermutet korrekt einen Syntaxfehler – einen solchen stellt die falsche Verwendung von Anführungsstrichen dar. Nun gut, die Vermutung, dass wir ein Komma vergessen haben, ist leider falsch …

Sofern Steuerzeichen (nicht nur Anführungsstriche!) bei einer Ausgabe als normale Buchstaben dargestellt werden sollen, kann ihnen die Steuerungswirkung durch einen vorangestellten Backslash genommen wer-

den. Sofern der obige Code wie nachstehend verändert wird, ist die Ausgabe auf der Konsole wieder unproblematisch:

```
print("Sie nannten ihn \"Hombre\".")
```

Eine nützliche Erweiterung bei der Ausgabe von Strings bietet die sogenannte „f“-Schreibweise: Mit dem vorangestellten „f“ und geschweiften Klammern können Mischungen aus Zeichenketten und nicht als Zeichenketten typisierten Variablen ausgegeben werden. Dazu ein Beispiel – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = 3
print(f"Robert hat {a} Kinder.")
```

Auf der Konsole erscheint bei Ausführung des Codes:

```
Robert hat 3 Kinder.
```

Diese Möglichkeit, Zeichenketten mit anders typisierten Variablen zu mischen, ist recht praktisch für die Ausgabe der Ergebnisse von Berechnungen und Analysen.

Strings sind unter Anderem – wie fast alles unter Python – „Objekte“. Obgleich die objektorientierte Programmierung nicht Gegenstand dieser Veranstaltung ist, lassen sich doch etliche Techniken der objektorientierten Programmierung sehr einfach (und ohne weitergehende Kenntnisse) nutzen. Auf Objekte können z.B. Methoden angewandt werden. Dies kann z.B. geschehen, indem der Name der jeweiligen Methode (wiederum als Funktion) mit einem Punkt an den Namen der Variablen, auf deren Inhalt sie angewandt werden soll, angehängt wird.

Als Beispiel wird hier die Wandlung zwischen Groß- und Kleinschreibung bei einem String herangezogen. Folgend wird der Inhalt eines Strings zunächst von einer vorhandenen Großschreibung in eine Kleinschreibung gewandelt und dann die Großschreibung wieder hergestellt. Dazu dienen die Methoden bzw. Funktionen „lower()“ und „upper()“. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
a = "GROSSES THEMA"
print(a)
a = a.lower()
print(a)
a = a.upper()
print(a)
```

Als Konsolenausgabe ergibt sich bei Ausführung des Codes:

```
GROSSES THEMA
grosses thema
GROSSES THEMA
```

Es ist in den Zeilen 3 und 5 des Codes ersichtlich, dass der mit der jeweiligen Methode veränderte Inhalt der Variablen wieder erneut auf die Variable zugewiesen wird. Unter Python ist eine Vielzahl an Methoden für die unterschiedlichen Datentypen von Variablen verfügbar.

Sofern man mit Strings arbeitet, muss man oft an ihnen „herumschnippeln“ – sie gezielt zerschneiden, links oder rechts etwas abschneiden usw.. Dies führt zum für Python (auch über die Verarbeitung von Strings hinaus) wichtigen Thema „Indizierung und Slicing“ im nächsten Abschnitt.

4.5 Indizierung und Slicing

Indizierung

Ein String bzw. eine Zeichenkette ist – wie es der Name schon sagt – aufzufassen als eine Folge (Sequenz, Kette) von einzelnen Buchstaben. Wir können den ersten, zweiten, dritten usw. Buchstaben dieser Kette klar identifizieren. Beim Indizieren und Slicing geht es darum, wie auf einzelne Elemente oder Teilmengen von Elementen in Sequenzen zugegriffen werden kann – z.B. auf eine Folge von vier Buchstaben in einem String.

Die Vorgehensweise bei solchen Zugriffen wird folgend am Beispiel von Strings demonstriert. Unter Python gibt es jedoch über Strings hinaus noch weitere Arten von Sequenzen – wie z.B. Listen, Tupel oder Dictionaries (als Datenstrukturen behandelt im Abschnitt 4.6). Auch auf diese lassen sich die vermittelten Vorgehensweisen dann anwenden. Daher ist Indizierung und Slicing ein wichtiges Thema für eine Einführung in die Programmiersprache Python.

Die Indizierung findet unter Python nativ „nullbasiert“ statt (vgl. folgend auch Abb. 4, Zeile „von links …“). Dies bedeutet, dass die Elemente einer Sequenz nicht von 1 an aufwärts nummeriert werden (1, 2, 3, …), sondern beginnend mit 0 (0, 1, 2, 3 …). Das zweite Element der Sequenz trägt dann z.B. die Nr. 1. Davon lassen sich viele Anfänger

/innen verwirren. Zur Verwirrung trägt darüber hinaus noch bei, dass wir später etliche Ausnahmen von dieser Regel begegnen werden.

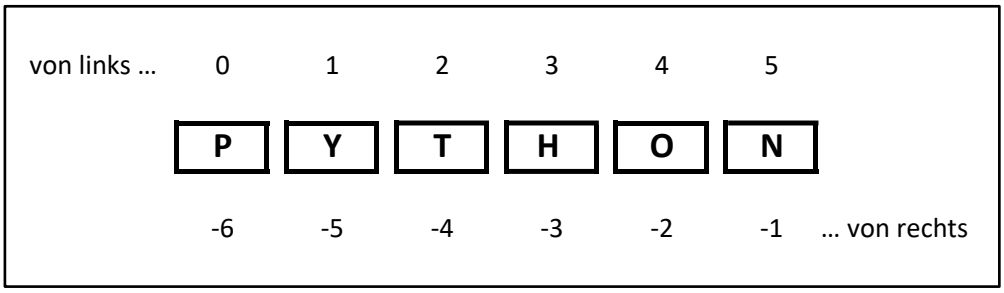

Abb. 4: Indizierung eines Strings unter Python
(Beispiel übernommen von Zumstein (2022))

Die erste Ausnahme folgt direkt: Unter Python kann eine Sequenz nicht nur „vorwärts“ (vom ersten zum letzten Element, in Abb. 4 von links nach rechts) indiziert werden, sondern auch „rückwärts“ (in Abb. 4 von rechts nach links, Zeile „von rechts …“). Die Indizierung beginnt dann beim letzten Element der Sequenz mit -1 und wird weiter in die negativen Zahlen fortgeschrieben (-1, -2, -3 …). Dabei wird nun nicht mit der 0 begonnen, sondern mit der -1. Diese Indizierung ist also schon nicht mehr nullbasiert.

Die Indizierung bzw. die Ansprache eines indizierten Elements wird im Programmcode immer in eckige Klammern gesetzt. Dazu ein erstes Beispiel unter Nutzung der Indizierung von links – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
s = "PYTHON"
print(s[1])
```

Wird dieser Code ausgeführt, so wird in der Konsole ausgegeben:

```
Y
```

Mit der Indizierung 1 wird also der zweite Buchstabe der Zeichenkette „PYTHON“ angesprochen bzw. aus dem String herausgeschnitten.

Nun wird der obige Code wie nachstehend verändert:

```
s = "PYTHON"
print(s[-2])
```

Bei Ausführung wird nun in der Konsole der Buchstabe „O“ ausgegeben – der zweite Buchstabe von hinten.

Slicing

Eigentlich waren die letzten Codebeispiele auch schon der Einstieg ins Slicing – in der einfachsten Form. Beim Slicing geht es darum, einer Sequenz vorbestimmte Elemente zu entnehmen, sie „herauszuschneiden“. Auch oben wurden einzelne Buchstaben bereits aus einem String herausgeschnitten – naja, korrekt beschrieben: herauskopiert.

Das Slicing bietet jedoch mehr Möglichkeiten als nur ein einzelnes Element (oben: einzelne Buchstaben) herauszuschneiden. Dabei gilt verallgemeinernd die Syntax „sequenz[start:stop:step]“, die im Weiteren zu erklären ist. Wie schon oben stehen die drei Elemente der Syntax in eckigen Klammern. Sie werden untereinander durch Doppelpunkte getrennt.

Eine erste Orientierung: „start“ gibt an, wo ein Schnitt beginnt. „stop“ gibt an, wo der Schnitt endet. „step“ ermöglicht es, beim Schnitt Elemente zu überspringen.

„start“ und „stop“ haben nun wieder eine Besonderheit, die oft zur Verwirrung führt: Python verwendet an dieser Stelle halboffene Intervalle. „start“ ist dabei in den Schnittbereich eingeschlossen, „stop“ hingegen ausgeschlossen. Sofern man also z.B. aus dem Wort „PYTHON“ mit der Anweisung [1:4:1] schneidet, wird vom zweiten bis zum vierten Buchstaben geschnitten – neben den halboffenen Intervallen für „start“ und „stop“ ist auch noch die oben eingeführte Nullbasierung des Index zu beachten. Die „1“ für „step“ ist übrigens der Default – dabei wird die Schrittweite 1 vorgegeben und jeder Buchstabe zwischen „start“ und „stop“ als halboffenem Intervall mitgenommen. Das Ergebnis dieses Schnitts wäre dann die Ausgabe von „YTH“.

„step“ steuert die Entnahme von Elementen zwischen „start“ und „stop“. Setzt man z.B. „step“ auf 3, so würde jedes dritte Element innerhalb der vorgegebenen Grenzen von links nach rechts zurückgegeben. „step“ als -2 würde jedes zweite Element von rechts nach links liefern. Lässt man „step“ einfach weg, so greift der Default von 1.

Dies soll mit einigen Beispielen demonstriert werden – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen neu ein:

```
s = "PYTHON"
print(s[:4])
```

In diesem Fall wird der String vom ersten Buchstaben (da „start“ fehlt, beginnt der Schnitt am Anfang der Sequenz) bis zum dritten Buchstaben angesetzt (Nullbasierung!). Da kein „step“ angegeben wird, gilt der Default. Das Ergebnis wäre „PYTH“ (auf die Konsolenausgaben wird bei diesen Beispielen verzichtet). Analog hätte man auch schreiben können:

```
print(s[0:4:1])
```

Dieser Ausdruck führt zum gleichen Ergebnis wie zuvor. Ersetzen wir die letzte Zeile des Codes nun durch:

```
print(s[2:4])
```

Jetzt werden alle Buchstaben vom dritten bis zum vierten Buchstaben entnommen bzw. kopiert. Das Ergebnis wäre „TH“.

Als nächstes wird die letzte Codezeile verändert zu:

```
print(s[-3:])
```

„start“ steht nun auf dem dritten Element von rechts (!). Für „stop“ ist kein Wert angegeben. Dies bedeutet, dass vom dritten Element von rechts aus bis zum Ende des Strings geschnitten bzw. kopiert wird. Das Ergebnis ist „HON“. Eine Variation dieser Anweisung könnte sein:

```
print(s[-3:-1])
```

Wieder wird beim dritten Element von rechts gestartet. Gestoppt wird beim letzten Element, das aber aufgrund des halboffenen Intervalls ausgeschlossen ist. Die Rückgabe erfolgt dann zu „HO“.

Eine weitere Veränderung der letzten Codezeile soll die Wirkung des dritten Parameters „step“ zeigen:

```
print(s[::2])
```

„start“ und „stop” sind nicht angegeben, daher wird der gesamte String mit in die Betrachtung einbezogen. „step“ steht auf zwei; mithin wird jedes zweite Element des Strings zurückgegeben. Dabei ist zu beachten, dass mit dem ersten Element des Strings begonnen wird. Die Ausgabe in der Konsole ergibt sich zu „PTO“.

Als letztes Beispiel variieren wir die letzte Codezeile nun noch einmal besonders verwirrend zu:

```
print(s[-1:-4:-1])
```

Der Start ist das letzte Element (also das erste von rechts). Das Ende des Schnitts findet sich wegen des halboffenen Intervalls beim dritten Element von rechts. Die Schrittweite für den „step“ ist negativ, also werden die Elemente von rechts aus ausgelesen. Es ergibt sich die Ausgabe „NOH“ in der Konsole.

Das war Ihnen bislang zu einfach, eher trivial? Die Komplexität lässt sich noch steigern – mit den verketteten Slicing-Operationen. Dabei wird eine zweite (dritte, vierte …) Slicing-Operation auf die jeweils vorher ausgeführte Slicing-Operation angewandt. Wir variieren wiederum die letzte Codezeile, um ein Beispiel einzuführen:

```
print(s[-2:][1])
```

Der erste Slice gibt für [-2:] zunächst „ON“ zurück. Auf diesen ersten Slice wird dann die zweite Slicing-Operation angewandt. Aus dem „ON“ wird folglich der zweite Buchstabe (Nullbasierung!) herausgeschnitten und zurückgegeben: Dies ist das „N“. Analog hätten wir auch schreiben können:

```
print(s[-1])
```

Indizierung und Slicing lassen sich nicht nur auf Strings, sondern auch auf andere Datenstrukturen anwenden. Deren Einführung ist Gegenstand des nächsten Abschnitts.

4.6 Datenstrukturen

Grundlegende Überlegungen

Mit der Einführung von Variablen wird es möglich, Informationen bestimmter Typen unter Angabe eines „Etiketts“, eines Namens in Programmen zu speichern. Unpraktisch wird dies, sobald in einem Zusammenhang stehende Einzelinformationen in größerer Zahl vorliegen. Sollen z.B. die Namen der Monate des Jahres („Januar“, „Februar“, … „Dezember“) gespeichert werden, so könnte man zwar die Variablen „monat01“, „monat02“ … bis „monat12“ einführen – aber spätestens dann, wenn die Monate der Reihe nach genannt bzw. durchlaufen werden sollen, wird es recht mühsam.

Die grundlegende Idee ist es nun, die Variablen in einem Vektor oder einer Matrix anzuordnen. Die Matrix oder der Vektor erhält dann einen

Namen. Auf die einzelnen Elemente innerhalb von Vektor oder Matrix kann mittels einer Indizierung zugegriffen werden.

Solche Vektoren oder Matrizen werden üblicherweise als „Datenstrukturen“ bezeichnet. Die wesentlichen nativen Datenstrukturen unter Python sind Listen und Wörterbücher (dictionaries). Darüber hinaus gibt es unter anderem auch noch Tupel und Mengen (sets). Diese werden aber eher seltener genutzt. Die vier genannten Datenstrukturen werden folgend vorgestellt. Dabei gilt die meiste Aufmerksamkeit aufgrund ihrer Bedeutung den Listen und Wörterbüchern.

Listen

Elemente dieser Datenstruktur dürfen auch unterschiedliche Datentypen haben – nicht alle Elemente müssen den gleichen Datentyp aufweisen (obwohl das in der Praxis der häufigste Fall ist). Listen werden mit eckigen Klammern umschlossen bzw. notiert.

Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und legen eine Liste an:

```
liste1 = ["eins", "zwei", "drei"]
print(liste1)
```

Auf der Konsole wird bei Ausführung des Codes ausgegeben:

```
['eins', 'zwei', 'drei']
```

Die einzelnen Elemente der Liste sind hier Strings. Die Elemente der Liste werden jeweils untereinander durch ein Komma getrennt. Die Liste trägt den Namen „liste1“. Es sei eine weitere Liste eingeführt, welche aus Ganzzahlen besteht:

```
liste2 = [1, 2, 3]
print(liste2)
```

Die Ausgabe auf der Konsole sieht bei Ausführung des Codes wie nachstehend aus:

```
['eins', 'zwei', 'drei']
[1, 2, 3]
```

Listen lassen sich mit einem Plus verketten. In diesem Falle sieht man folgend auch, dass Listen nicht „typrein“ sein müssen – mithin aus Elementen unterschiedlicher Datentypen bestehen können (in diesem Fall Strings und Ganzzahlen, also Integers). Die Verkettung der beiden Listen erhält den neuen Namen „liste3“.

```
liste3 = liste1 + liste2
print(liste3)
```

Bei Ausführung des Codes wird auf der Konsole ausgegeben:

```
['eins', 'zwei', 'drei']
[1, 2, 3]
['eins', 'zwei', 'drei', 1, 2, 3]
```

Listen können auch in beliebiger Tiefe ineinander verschachtelt werden. Die bisherigen Listen waren Vektoren. Soll eine Matrix erzeugt werden, so werden Listen in einer Tiefe von einer Stufe ineinander verschachtelt. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und legen eine verschachtelte Liste als zweidimensionale Matrix an:

```
liste4 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(liste4)
```

Die Ausgabe auf der Konsole bei Ausführung des Codes ergibt sich zu:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Auf die einzelnen Elemente wie auch Elementgruppen der verschachtelten Liste lässt sich nun über die Indizierung mittels des bekannten Slicing zugreifen. Die Ausgabe der zweiten Zeile der „liste4“ gelingt z.B. mittels:

```
print(liste4[1])
```

Sofern wir die Zeile „print(liste4)“ von oben vor der Ausführung des Codes löschen, lautet die Ausgabe auf der Konsole nun:

```
[4, 5, 6]
```

Die „liste4“ besteht aus drei Elementen, die wiederum Listen sind. Ausgegeben wurde wegen der Nullbasierung der Indizierung die zweite Subliste mit all ihren Elementen (den einzelnen Zahlen).

Die im vorherigen Abschnitt eingeführten Techniken der Slicing erlauben es nun, beliebig auf die Elemente der Matrix, welche „liste4“ bildet, zuzugreifen. Exemplarisch soll nun aus der zweiten Subliste nur das zweite und das dritte Element ausgegeben werden:

```
print(liste4[1][1:])
```

Sofern vor der Ausführung des Codes die vorherige Ausgabe „print (liste4[1])“ gelöscht wurde, erscheint auf der Konsole:

```
[5, 6]
```

Zusammenfassend kann festgehalten werden, dass mittels verschachtelter Listen *n*-dimensionale Matrizen abgebildet werden können. Mittels Indizierung und Slicing kann sowohl auf einzelne Elemente in diesen Datenstrukturen wie auch Gruppen von Elementen zugegriffen werden.

Python verfügt auch über Funktionalitäten, um Listen im Programmablauf zu ändern. So können Elemente an eine Liste angehängt werden. Dies geschieht mittels der Methode „.append()“. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und fügen an eine Namensliste einen neuen Namen an:

```
n = ["Petra", "Klaus"]
n.append("Hans")
print(n)
```

Bei Ausführung des Codes erscheint auf der Konsole erwartungsgemäß die Ausgabe:

```
['Petra', 'Klaus', 'Hans']
```

Soll eine Einfügung in eine Liste an einer bestimmten Indexposition ausgeführt werden, so geschieht dies mit der Methode „.insert()“. Die Einfügung erfolgt stets vor der angegebenen Indexposition. Soll der Name „Helge“ am Anfang der bestehenden Liste eingefügt werden, so muss die Indexposition 0 angegeben werden. Wir eliminieren den letzten „print()“-Befehl im Code und ergänzen:

```
n.insert(0, "Helge")
print(n)
```

Wird der Code ausgeführt, so erscheint in die Konsolenausgabe erwartungsgemäß die entsprechend ergänzte Liste:

```
['Helge', 'Petra', 'Klaus', 'Hans']
```

Zu beachten ist, wie hier auch schon deutlich wird, die Nullbasierung des Index. Eine Einfügung vor dem zweiten Namen der Liste (nach Eliminierung des letzten „print()“-Befehls im Code) sei:

```
n.insert(1, "Helga")
print(n)
```

In der Konsole erscheint nach Ausführung des Codes die Ausgabe:

```
['Helge', 'Helga', 'Petra', 'Klaus', 'Hans']
```

Soll das letzte Element einer Liste gelöscht werden, so geschieht dies mit der „.pop()“-Methode. Im Code wird der letzte „print()“-Befehl eliminiert und eingefügt:

```
n.pop()
print(n)
```

Auf die Wiedergabe der um den Namen „Hans“ gekürzten Liste aus der Konsolenausgabe bei Ausführung des Codes wird hier verzichtet.

Ein Eintrag in der Liste auf einer bestimmten Position kann z.B. mit der „del“-Anweisung gelöscht werden. Diese Anweisung ist keine Methode wie „.insert()“ oder „.pop()“ und wird daher nicht mit einem Punkt an den Listennahmen angehängt. Nach der Löschung des letzten „print()“-Befehls wird der Code ergänzt um:

```
del n[1]
print(n)
```

Der Eintrag auf der angegebenen Indexposition 1 – es gilt wieder die Nullbasierung! –, hier der Name „Helga“, wird aus der Liste gelöscht. Die aktuelle Liste sieht nun nach der Ausführung des Codes in der Konsolenausgabe wie nachstehend aus:

```
['Helge', 'Petra', 'Klaus']
```

Für die Entfernung von Elementen aus Listen kann aber auch die „.pop()“-Methode unter Angabe von Indexpositionen oder die „.remove()“-Methode verwendet werden. Googeln Sie die entsprechende Handhabung und testen Sie es!

Abschließend sollen noch einige weitere Operationen mit Listen erläutert werden. Eine aufkommende Frage mag sein, wie viele Elemente die Liste aktuell aufweist. Dies lässt sich mit der Funktion „len()“ (für „length“ wie Länge) feststellen. Der letzte „print()“-Befehl im Code wird variiert zu:

```
print(len(n))
```

Nach der Ausführung des Codes erscheint in der Konsolenausgabe die Zahl 3. Dies passt zur letzten Ausgabe der Liste oben. Mittels „len()“ kann auch die Anzahl Buchstaben eines Strings festgestellt werden. Der letzte „print()“-Befehl wird verändert zu:

```
print(len(n[2]))
```

Damit wird die Länge des dritten und letzten Namens in der Liste ermittelt (Nullbasierung!). Der Name „Klaus“ weist fünf Buchstaben auf. Genau diese Zahl erscheint dann auch bei Ausführung des Codes in der Konsolenausgabe.

Zuweilen möchte man wissen, ob ein bestimmtes Element in einer Liste enthalten ist. Dafür kann der „in“-Operator verwendet werden. Der letzte „print()“-Befehl des Codes wird umgeschrieben in:

```
print("Petra" in n)
```

Damit wird gefragt, ob der String „Petra“ in der Liste „n“ enthalten ist. Wird der Code ausgeführt, so erscheint ein „True“ in der Konsolenausgabe. Die letzte Codezeile ist also wieder ein boolescher Ausdruck (vgl. Abschnitt 4.3). Probieren Sie es noch einmal mit einem nicht in der Liste enthaltenen Namen bzw. String aus und testen Sie, ob wirklich ein False bei Ausführung des Codes in der Konsolenausgabe erscheint!

Eine weitere im Kontext von Listen häufig angewandte Operation ist die Sortierung. Die aktuelle Liste „n“ kann sortiert ausgegeben werden, indem die Funktion „sorted()“ angewandt wird. Der letzte „print()“-Befehl wird variiert und um einen zweiten „print()“-Befehl für eine weitere Ausgabe ergänzt:

```
print(sorted(n))
print(n)
```

Bei Ausführung des Codes erscheint in der Konsolenausgabe:

```
['Helge', 'Klaus', 'Petra']
['Helge', 'Petra', 'Klaus']
```

Der erste „print()“-Befehl erzeugt eine sortierte Ausgabe der aktuellen Liste. Die Liste selbst ist aber nicht sortiert worden – dies zeigt der zweite „print()“-Befehl. Die gespeicherte Liste ist noch immer unsortiert.

Soll die Liste selbst im Speicher sortiert werden, so kann die Methode „.sort()“ auf die Liste angewandt werden. Die letzten beiden „print()“-Befehle werden aus dem Code gestrichen und es wird ergänzt:

```
n.sort()
print(n)
```

Bei Ausführung des Codes ergibt sich die Konsolenausgabe erwartungsgemäß zu:

```
['Helge', 'Klaus', 'Petra']
```

Bislang wurde die Sortierrichtung nicht erwähnt – es wurde einfach immer aufsteigend sortiert. Was aber nun, wenn die Liste absteigend alphabetisch sortiert werden soll? – Dies wird von einem Parameter bewirkt, welcher der Methode „.sort()“ mitgegeben wird. Die vorletzte Zeile des Codes wird verändert zu:

```
n.sort(reverse = True)
```

Bei erneuter Ausführung des Codes erscheint die alphabetisch absteigend sortierte Liste in der Konsolenausgabe:

```
['Petra', 'Klaus', 'Helge']
```

Allerdings hat die Sortierung auch ihre Grenzen. Sortieren lassen sich nur datentypreine Listen. Versuchsweise kann man z.B. die aktuelle, noch aus drei Namen bestehende Liste durch eine Ganzzahl erweitern:

```
n.append(3)
```

Ein anschließender Versuch, die neue Liste dann wiederum zu sortieren, führt zu einer Fehlermeldung.

Abschließend soll noch der Fall betrachtet werden, dass ein Listenelement geändert werden soll – der „Klaus“ oben soll durch „Klaas“ ersetzt werden. Auch hier wird wiederum mit der Indizierung gearbeitet. Wir löschen die letzte Zeile des Codes (das „.append()“) und ergänzen:

```
n[1]="Klaas"
print(n)
```

Die Ausgabe in der Konsole ergibt sich nun bei Ausführung des Programms zu:

```
['Petra', 'Klaus', 'Helge']
['Petra', 'Klaas', 'Helge']
```

Damit endet die Erläuterung von Listen an dieser Stelle. Es gibt (natürlich!) noch etliche andere Möglichkeiten, mit Listen zu arbeiten … diese werden dann im Verlauf dieser Veranstaltung eingeführt, sobald sie benötigt werden.

Wörterbücher / Dictionaries

Auf die einzelnen Elemente einer Liste kann man i.d.R. nur über eine Indexposition zugreifen. Das kann mühsam sein. In Wörterbüchern hingegen ist ein Zugriff auf die Elemente auch über einen Schlüssel (z.B.

einen Namen) möglich. Wörterbücher ordnen Schlüssel auf Werte zu. Schlüssel und Werte können dabei jeden beliebigen Datentyp annehmen. Es gibt allerdings eine Einschränkung: Jeden Schlüssel kann es in einem Wörterbuch nur ein einziges Mal geben (ähnlich einem Primärschlüssel in einer Datenbank).

Wörterbücher sind durch die sie umschließenden geschweiften Klammern und die Schlüssel-Wert-Zuordnung durch Doppelpunkte charakterisiert. Hier wird ein Beispiel eingeführt, welches jedem chemischen Element seine Position/Nummer im Periodensystem zuordnet. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und legen ein Wörterbuch mit dem Namen „cel“ (für „chemische Elemente“) an:

```
cel = {"H": 1, "Pb": 82, "Au": 78}
print(cel)
```

Wird der Code ausgeführt, erscheint als Ausgabe auf der Konsole eine Wiedergabe des Wörterbuches:

```
{'H': 1, 'Pb': 82, 'Au': 78}
```

Soll nun im Wörterbuch etwas nachgeschlagen werden, wird dafür der Schlüssel verwendet. Dieser wird in eckigen Klammern – wie eine Indexposition – angegeben. Damit ist der Nachschlag mit Schlüssel im Wörterbuch nur eine Variante des Zugriffs über einen Index. Der letzte „print()“-Befehl im Code wird modifiziert zu:

```
print(cel["H"])
```

Auf der Konsole erscheint bei Ausführung des Codes als Ausgabe die nachgeschlagene Zahl 1 – also der dem Schlüssel zugeordnete Wert. Wir stellen fest, dass das Wörterbuch fehlerhaft ist – Gold („Au“) hat nicht etwa die Nummer 78, sondern 79 im Periodensystem. Der Wert kann durch eine Ergänzung des Codes neu zum Schlüssel zugewiesen werden (der letzte „print()“-Befehl wird zuvor gelöscht):

```
cel["Au"] = 79
print(cel)
```

Auf der Konsole wird bei Ausführung des Codes das korrigierte Wörterbuch ausgegeben:

```
{'H': 1, 'Pb': 82, 'Au': 79}
```

Auch neue Schlüssel-Wertpaare lassen sich hinzufügen. Nach Löschung des letzten „print()“-Befehls wird der Code ergänzt mit:

```
cel["O"] = 8
print(cel)
```

Der neue Schlüssel für Sauerstoff, welcher bislang nicht im Wörterbuch enthalten war, wird nun diesem hinzugefügt. Die Ausgabe auf der Konsole bei Ausführung des Codes enthält nun auch diesen:

```
{'H': 1, 'Pb': 82, 'Au': 79, 'O': 8}
```

Das neue Element wird dabei hinten an das Wörterbuch angefügt.

Auch zu Wörterbüchern gäbe es noch Etliches mehr zu berichten – so z.B. über das Zusammenführen von Wörterbüchern, die Setzung von Standardschlüsseln und vieles mehr. Auch diese weiteren Möglichkeiten können und werden wir erkunden, sobald wir sie benötigen. Werfen wir folgend noch einen kurzen Blick auf die Tupel und die Mengen.

Tupel

Tupel sind Listen sehr ähnlich. Ihr wesentliches Merkmal ist ihre Unveränderlichkeit. Sie werden als durch Komma getrennte Werte zugewiesen. Die Zuweisung kann aber auch in runde Klammern gesetzt werden. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und legen ein Tupel mit dem Namen „w“ (für „Währungen“) an:

```
w = "EUR", "GBP", "USD"    oder: w = ("EUR", "GBP", "USD")
print(w)
```

Bei Ausführung des Codes erscheint das Tupel als Ausgabe in der Konsole:

```
('EUR', 'GBP', 'USD')
```

Ein Zugriff auf das Element eines Tupels erfolgt (wie bei Listen) über den Index. Wir variieren den letzten „print()“-Befehl im Code zu:

```
print(w[1])
```

Bei Ausführung des Codes wird in der Konsolenausgabe der Text „GBP“ angezeigt. Der Index ist – wie gewohnt – nullbasiert. Da das Tupel „unveränderlich“ ist, kann es nur erweitert werden, indem es mit der Erweiterung neu auf den Namen des Tupels zugewiesen wird (da

fragt man sich, was unter „unveränderlich“ zu verstehen ist …). Wir löschen den letzten „print()“-Befehl und ergänzen als Code:

```
w = w + ("CHF",)
print(w)
```

Wichtig ist es dabei, das Komma vor der schließenden Klammer nicht zu vergessen. Fehlt dieses, so kommt es zu einer Fehlermeldung. In der Konsolenausgabe erscheint bei Ausführung des Codes:

```
('EUR', 'GBP', 'USD', 'CHF')
```

Zuletzt noch ein Blick auf die Mengen.

Mengen / Sets

Die mit ihrem englischen Namen auch als „Sets“ bezeichneten Mengen sind Auflistungen, die keine doppelten Elemente enthalten dürfen. Wie beim Primärschlüssel einer Datenbank darf es also nur eindeutige Elemente geben. Einer der sinnvollen Einsatzzwecke ist es damit z.B., die eindeutigen Elemente einer Liste oder eines Tupels zu bestimmen. Mengen werden in geschweiften Klammern aufgeschrieben. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und legen eine Menge mit dem Namen „m1“ mit Währungskennungen an:

```
m1 = {"EUR", "USD", "NOK"}
print(m1)
```

Die Konsolenausgabe ergibt sich bei Ausführung des Codes zu:

```
{'USD', 'NOK', 'EUR'}
```

Nach der Löschung des letzten „print()“-Befehls definieren wir nun eine *Liste* (!) „liw“, die aus Währungskennungen besteht. In dieser Liste gibt es allerdings Dubletten (doppelte Elemente):

```
liw = ["USD", "USD", "CHF", "EUR", "USD", "EUR"]
```

Um aus dieser Liste nun sämtliche eindeutigen Elemente herauszuziehen, können wir sie mit dem Konstruktor von Mengen, der Funktion „set()“, in eine Menge verwandeln. „set()“ verwandelt eine Liste oder ein Tupel in eine Menge. Dabei verschwinden sämtliche Dubletten und nur noch eindeutige Werte verbleiben:

```
m2 = set(liw)
print(m2)
```

Nachdem der Code ausgeführt wurde, erscheint als Ausgabe in der Konsole die um Dubletten bereinigte Menge:

```
{'USD', 'EUR', 'CHF'}
```

Dies ist allerdings nur ein Weg, um Dubletten aus Listen oder Tupeln zu entfernen – es gibt noch andere. Schließlich erlauben die Sets bzw. Mengen auch mengentheoretische Operationen mit den Daten. Wir löschen den letzten „print()“-Befehl aus dem Code und ergänzen:

```
m3 = m1.union(m2)
print(m1)
print(m2)
print(m3)
```

Die Methode „.union()“ als Funktion weist auf die neue Menge „m3“ die Vereinigungsmenge aus „m1“ und „m2“ zu. In der Konsolenausgabe erscheint nun:

```
{'USD', 'NOK', 'EUR'}
{'USD', 'EUR', 'CHF'}
{'USD', 'NOK', 'EUR', 'CHF'}
```

Auch bei dieser Operation werden durch das Konstrukt der Menge unter Python evtl. Dubletten unterdrückt. Genauso lassen sich auch Schnittmengen aus zwei gegebenen Mengen bilden, in denen dann nur die in beiden Mengen erscheinenden Elemente zu finden sind. Man ersetze in der viertvorletzten Zeile des Codes das „.union()“ durch ein „.intersection()“ wie nachstehend (die Ausgabe der drei Mengen mittels des „print()“-Befehls verbleibt unverändert):

```
m3 = m1.intersection(m2)
print(m1)
print(m2)
print(m3)
```

Die entsprechende Konsolenausgabe bei Ausführung des Codes ergibt sich nun zur Schnittmenge aus „m1“ und „m2“:

```
{'EUR', 'NOK', 'USD'}
{'EUR', 'CHF', 'USD'}
{'EUR', 'USD'}
```

Nur „EUR“ und „USD“ sind in den beiden ersten Mengen gemeinsam enthalten und bilden daher die Schnittmenge.

Die Funktion „set()“ wurde hier als Konstruktor eingeführt, um aus einer Liste eine Menge zu erzeugen. Analog zum „set()“ gibt es natürlich

auch Konstruktoren für Listen („list()“), für Wörterbücher („dict()“) und Tupel („tuple()“), um die verschiedenen Datenstrukturen untereinander umzuwandeln.

4.7 Bedingungen und Wiederholungen (Steuerungsfluss)

Steuerungsfluss

Mit diesem Abschnitt setzen wir nach etlichen Präliminarien nun endlich Kurs auf das Herz der Programmierung: Den Steuerungsfluss. Beim Steuerungsfluss geht es zum ersten darum, dass im Ablauf eines Programms bestimmte Teile desselben nur *bedingt* ausgeführt werden – d.h., nur dann, wenn eine oder mehrere Voraussetzungen erfüllt sind. Zum zweiten geht es darum, dass bestimmte Teile des Programms *mehrfach* ausgeführt werden sollen. Will man z.B. dreihundertmal „Hello World“ auf der Konsole ausgeben, so könnte man natürlich entsprechend oft „print(„Hello World“)“ untereinander in den Code schreiben. Eleganter und effizienter ist es hingegen, eine Technik einzusetzen, welche die Wiederholung abbildet und steuert (das wird im weiteren Verlauf der Darstellung eine „Schleife“ sein).

Konstrukt des Codeblocks

Das erste Konstrukt, welches für den Steuerungsfluss benötigt wird, ist der „Codeblock“. Als ein Codeblock soll hier ein Teil des Codes definiert werden, der einem bestimmten Zweck dienen soll. Dies könnten z.B. Anweisungen sein, die ausgeführt werden sollen, sofern eine bestimmte Bedingung zutrifft. Alternativ könnten es auch Anweisungen sein, welche mehrfach wiederholt werden sollen. Schließlich könnte es sich auch um eine eigenerstellte Funktion im Programmcode handeln (diese wird allerdings erst im Abschnitt 4.8 eingeführt).

Auch in den meisten anderen Programmiersprachen gibt es Codeblöcke. Diese werden dort allerdings z.B. durch eine Setzung in Klammern oder bestimmte Schlüsselworte zu Beginn und Ende des Codeblocks gegen den Rest des Codes abgegrenzt. Diese Abgrenzung geschieht in Python durch eine *Einrückung* des Codes im Editor. Während bei vielen anderen Programmiersprachen Einrückungen des Codes nur der Lesbarkeit dienen, haben sie bei Python einen ganz konkreten Zweck und steuern den Programmablauf. Daher spricht man bei Python im

Hinblick auf die Einrückung auch von „Significant White Space“, also „signifikanten Leerzeichen“. Tatsächlich gibt es auch für die Tiefe der Einrückung (also die Anzahl der Leerzeichen) einen Standard: Dies sind vier Leerzeichen. PyCharm verwendet diese Anzahl automatisch (und setzt keinen Tabulator!).

Bedingungen

Als Einstieg ein einfaches Beispiel für einen bedingten Codeblock – und gleichzeitig die Einführung des „if“ („wenn“). Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen ein:

```
if bedingung:
    pass          # Nichts tun!
```

„bedingung“ ist hier eine Variable. Diese muss den Wert True annehmen, damit der eingerückte Codeblock (hier nur aus einer Zeile bestehend) ausgeführt werden kann. Damit sich ein sicheres True ergibt, hätte die erste Zeile z.B. „if 1 == 1:“ lauten können. Die Zeile vor dem Codeblock wird immer mit einem Doppelpunkt abgeschlossen (vergisst man leicht …!). Das Ende des Codeblocks wird immer durch die erste darauffolgende Zeile definiert, welche *nicht mehr* eingerückt ist. Eine Besonderheit ist hier die Anweisung „pass“. Diese benötigt man, um einen Codeblock zu konstituieren, der (erstmal) nichts tut.

Es folgt ein umfangreicheres Beispiel, welches nicht nur die Anwendung des bedingten Codeblocks thematisiert, sondern auch noch andere Sprachelemente einführt. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und beginnen mit einer Eingabeaufforderung („input()“-Funktion) in der Konsole:

```
a = input("Geben Sie eine ganze Zahl zwischen 0 und 10 ein: ")
```

Die Funktion „input()“ erlaubt Eingaben des/der Benutzer/s/in auf der Konsole. Dabei gibt „input()“ immer einen String zurück. Dieser wird hier auf die Variable „a“ zugewiesen. Im nächsten Schritt muss daher die eingegebene Zahl für die weitere Verarbeitung in eine Ganzzahl gewandelt werden. Wir ergänzen den Code:

```
a = int(a)
```

Die Funktion „int()“ wandelt u.a. einen String in eine Ganzzahl. Diese wird hier wieder auf die Variable „a“ zugewiesen, wodurch der

Datentyp von „a“ auf die Ganzzahl wechseln sollte. Die wird allerdings nur dann fehlerfrei passieren, sofern der/die Benutzer/in auch eine Ganzzahl eingibt. Gibt er hingegen einen String ein, so resultiert eine Fehlermeldung – nachfolgend ein Beispiel:

```
Geben Sie eine ganze Zahl zwischen 0 und 10 ein: Herbert
Traceback (most recent call last):
  File "D:\...\grundlagen.py", line 2, in <module>
    a = int(a)
ValueError: invalid literal for int() with base 10: 'Herbert'

Process finished with exit code 1
```

Die nette Bitte des Programmierers/der Programmiererin um die Eingabe einer Ganzzahl kann also vom Benutzer/der Benutzerin ignoriert werden. Die Fehlermeldung, welche daraus resultiert, dass die Funktion „int()“ auf einen String angewandt wird, möchte man als Programmierer/in natürlich gerne „abfangen“ – also dafür sorgen, dass sie nicht auftritt und der/die Benutzer/in einen Hinweis auf seinen Fehler erhält. Notwendig ist eine Sicherheitsprüfung. Dafür gibt es unter Python das Try-Except-Konstrukt. Wir löschen die letzte Zeile des Codes und fügen ein:

```
try:
    a = int(a)
except ValueError:
    print("Das war keine ganze Zahl! Abbruch!")
else:
    pass
```

Der Codeblock mit der Kopfzeile „try:“ wird zunächst nach der Benutzereingabe ausgeführt bzw. durchlaufen. Sofern in diesem Codeblock der unten hinter „except“ spezifizierter Fehlertyp auftritt, wird der Codeblock mit der Kopfzeile „except …:“ ausgeführt. In diesem Fall wird der Fehler als ein „ValueError“ vorgegeben. Ein solcher tritt auf, wenn man versucht, mit der „int()“-Funktion einen String in eine Ganzzahl zu wandeln. Der „except …:“-Codeblock gibt hier eine Fehlermeldung für den/die Benutzer/in aus. Damit endet das Programm dann in diesem Falle. Falls der/die Benutzer/in jedoch korrekt eine Ganzzahl eingegeben hat, wird der „except …:“-Codeblock ignoriert und die Ausführung setzt sich im „else:“-Codeblock („ansonsten …“) fort. Dort steht als Platzhalter für andere Aktionen zunächst ein „pass“. Führen Sie den Code aus und testen Sie – bei Eingabe eines Strings sollte der

Fehlerhinweis erscheinen, nach Eingabe einer Ganzzahl sollte das Programm einfach enden.

Mit dem Try-Except-Konstrukt kann der Programmierer/die Programmiererin also Fehler bestimmter Typen (es gibt noch andere als den „ValueError“!) abfangen und den Abbruch des Programms mit einer Fehlermeldung verhindern. Zugleich ist das Try-Except-Konstrukt auch ein gutes Beispiel für die bedingte Ausführung von Programmcode.

Damit nun nach der korrekten Eingabe einer Ganzzahl noch etwas Sinnvolles passiert (und um weitere Fallunterscheidungen einzuführen …), wird das „pass“ im Code nun ersetzt (um die Einrückung deutlich zu machen, wird das letzte „else:“ dabei noch einmal wiederholt):

```
else:
    if a <= 10 and a >= 5:
        print("Zahl liegt zwischen 5 und 10!")
    elif a >= 0 and a < 5:
        print("Zahl liegt zwischen 0 und 4!")
    else:
        print("Zahl liegt nicht im geforderten Bereich!")
```

Das „if …:“ ist die hier schon bekannte Fallunterscheidung. Sofern die Bedingung erfüllt wird, erfolgt die Ausgabe von „Zahl liegt zwischen 5 und 10!“ auf der Konsole. Das „elif“ ist eine Verkürzung von „else if“ und führt eine zweite Fallunterscheidung mit einer Konsolenausgabe ein (davon könnten auch noch weitere folgen …). Das „else:“ schließlich fängt alle Restfälle, die durch die vorherigen Bedingungen noch nicht ausdrücklich erfasst wurden, ab und weist mit einer Konsolenausgabe darauf hin. Dies wäre hier z.B. der Fall, in welchem die Zahl 11 eingegeben wurde.

Probieren Sie die verschiedenen Eingabemöglichkeiten aus und testen Sie, ob auch die erwarteten Ausgaben resultieren!

Die Einrückung ist, wie schon oben erläutert, essentiell. Rücken Sie z.B. probehalber einmal das erste „else:“ im Code um vier Zeichen ein und testen Sie, was passiert … es wird auf einen Syntax-Fehler hingewiesen; der Code läuft nicht mehr glatt durch.

Mit einem zweiten Beispiel sei eine Sequenz als Datenstruktur – hier exemplarisch eine Liste – auf einen vorhandenen Inhalt hin geprüft. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und schreiben:

```
liste = []
if liste: # oder: if len(liste) > 0:
    print(f"Listeninhalt: {liste}")
else:
    print("Liste ist leer.")
```

Hier wird zu Beginn eine leere Liste mit dem Namen „liste“ eingeführt. Sodann wird untersucht, ob die Liste einen Inhalt hat. Sofern dies der Fall ist, nimmt „liste“ automatisch auch den Wert True an – bei einer leeren Liste hingegen den Wert False. Alternativ hätte man auf einen evtl. Inhalt der Liste auch – wie schon früher erläutert – mit „len(liste) > 0“ prüfen können (testen Sie dies als Alternative!). Natürlich sollten Sie auch den alternativen Fall überprüfen, dass die Liste einen Inhalt hat – indem Sie z.B. in der ersten Zeile „liste = [1,2]“ schreiben. Dann sollte der Listeninhalt mit der „print()“-Funktion in der „f“-Schreibweise auch in der Konsole ausgegeben werden.

Schleifen / Wiederholungen

Eine Schleife führt einen Codeblock wiederholt aus – entweder mit einer vorgegebenen Anzahl an Durchläufen oder mit einer Anzahl von Durchläufen, welche sich aus dem Programm selbst ergibt.

In einem ersten Beispiel werden die Elemente einer Liste durchlaufen und einzeln (zeilenweise) ausgegeben. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen als Code ein:

```
wr = ["USD", "CHF", "EUR"]
for w in wr:
    print(w)
```

Bei Ausführung des Codes erscheint auf der Konsole die Ausgabe:

```
USD
CHF
EUR
```

Die Variable „w” ist hier die Zählervariable. Sie nimmt jeden Wert in der Liste einmal an und gibt ihn dann mit dem „print()“ aus.

Sofern eine explizite Zählervariable als Zahl benötigt wird, bietet sich der Rückgriff auf die integrierte Funktion „range()“ an. „range()“ gibt eine Folge von Zahlen zurück. Mögliche Argumente der Funktion sind dabei „range(start, stop, step)“. Diese Argumente sind als Zahlen zu übergeben. Falls nur ein Argument übergeben wird, so wird dies als

„stop“ interpretiert. Es werden also Zahlen bis zur Zahl „stop“ erzeugt. Werden zwei Argumente übergeben, so erfolgt eine Interpretation als „start, stop“; mithin werden Zahlen von „start“ bis „stop“ erzeugt. In beiden Fällen sind wiederum eine Nullbasierung und ein halboffenes Intervall für die Erzeugung zu beachten. Erst bei der Übergabe von drei Argumenten wird das dritte dann als „step“ interpretiert und als Schrittweite für die Erzeugung verwendet (wie oben schon im Rahmen des Slicing eingeführt).

Ein Beispiel als Demonstration – wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen ein:

```
l = list(range(5))
print(l)
```

Unter dem Namen „l“ wird eine Liste angelegt, welche mit den Zahlen 0 bis 4 (Nullbasierung und halboffenes Intervall!) gefüllt wird. Entsprechend lautet die Konsolenausgabe der Liste „l“ bei Ausführung des Codes:

```
[0, 1, 2, 3, 4]
```

Ein weiteres Beispiel soll die zwei anderen Parameter der „range()“-Funktion beleuchten. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und schreiben:

```
l = list(range(2, 5, 2))
print(l)
```

Statt des direkten Blicks auf die Ausgabe bei Ausführung in der Konsole folgt hier eine Analyse der gewählten Parameter: Die Ausgabe beginnt hier mit der Zahl 2 („start“). Sie endet bei der Zahl 4 (halboffenes Intervall, „stop“). Der Parameter „step“ steht auf 2: Folglich wird die Ausgabezahl 3 übersprungen. Die Ausgabe wird „[2, 4]“ lauten.

Eine Schleife im Programmkontext wird üblicherweise unter Verwendung des Sprachelements „for“ gebildet. Auch dazu noch ein Beispiel (wir löschen die bislang geschriebenen Codezeilen oder speichern sie unter anderem Namen):

```
For i in range(3):
    print(i)
```

Die Verwendung des „i“ als Zählervariable hat eine lange Tradition in der Programmierung und wird daher häufiger zu finden sein (bei geschachtelten Schleifen wird dann für weitere Zähler traditionell das

Alphabet fortgeschrieben – „j“, „k“, „l“, …). Das „print()“ sorgt für eine zeilenweise Ausgabe in der Konsole bei Ausführung:

```
0
1
2
```

Die Nullbasierung und die halboffenen Intervalle führen auch hier – wie beim Slicing – bei Anfänger/innen oft zu Irritationen. Daher sollten entsprechende eigene Konstrukte im Zweifel getestet und über die temporäre Ausgabe von Zählerwerten verifiziert werden.

Sofern man für eine bestehende Liste eine explizite, z.B. mit ausgegebene Zählervariable benötigt, kann man auch die Funktion „enumerate()“ verwenden. Die Funktion gibt den Listenindex (nullbasiert) zurück, also 0, 1, 2, … Auch dazu ein Beispiel (wir löschen die bislang geschriebenen Codezeilen oder speichern sie unter anderem Namen), wir tragen den Code ein:

```
wr = ["USD", "CHF", "EUR"]
for i, w in enumerate(wr):
    print(i, w)
```

Hier wird neben dem Listenindex auch noch der Inhalt der Liste mit ausgegeben. Auf der Konsole erscheint bei Ausführung des Codes:

```
0 USD
1 CHF
2 EUR
```

Schleifen können nicht nur im Kontext von Listen, sondern auch für Tupel und Mengen verwendet werden. Beim Wörterbuch hingegen läuft der Zugriff über die jeweiligen Schlüssel – Schleifen sind möglich, können jedoch nicht über den Index gesteuert werden. Exemplarisch eine Schleife über den Inhalt eines Wörterbuches (wir löschen die bislang geschriebenen Codezeilen oder speichern sie unter anderem Namen):

```
cel = {"H": 1, "Pb": 82, "Au": 79}
for el in cel:
    print(el)
```

Die Ausgabe auf der Konsole bei Ausführung des Programms gibt die Elementenkürzel aus:

```
H
Pb
Au
```

Mit der Methode „.items()“ lassen sich gleichzeitig der Schlüssel und sein Wert als Datensatz bzw. Paar abrufen. Wir verändern die beiden letzten Codezeilen zu:

```
for el, nr in cel.items():
    print(el, nr)
```

Entsprechend erscheinen bei Ausführung des Programms in der Ausgabe auf der Konsole:

```
H 1
Pb 82
Au 79
```

Soll eine Schleife vorzeitig verlassen bzw. ihre Ausführung abgebrochen werden, so ist dies mit der Anweisung „break“ möglich. Auch dazu ein Beispiel (wir löschen die bislang geschriebenen Codezeilen oder speichern sie unter anderem Namen):

```
for i in range(15):
    if i == 2:
        break
    else:
        print(i)
```

Eigentlich soll die Schleife über die Zahlen 0, 1, 2, 3, … 14 laufen. Sobald die Zahl 2 erreicht wird, wird sie jedoch hier mit einem „break“ unterbrochen. Findet keine Unterbrechung statt, so wird der aktuelle Inhalt der Zählervariablen ausgegeben. In der Ausgabe erscheinen bei Ausführung des Programms (in eigenen Zeilen) die 0 und die 1. Die 2 wird schon nicht mehr ausgegeben.

Auch ein Überspringen der restlichen Elemente eines Codeblocks einer Schleife ist möglich. Dafür gibt es die Anweisung „continue“. Auch hier ein Beispiel (wir ersetzen das „break“ im aktuellen Code durch ein „continue“ und die Zahl 15 im „range()“ durch die Zahl 5):

```
for i in range(5):
    if i == 2:
        continue
    else:
        print(i)
```

In der Ausgabe auf der Konsole erscheinen bei Ausführung des Programms – zeilenweise untereinander! – die Werte 0, 1, 3 und 4. Die Zahl 2 fehlt – sie wurde durch das „continue“ übersprungen.

Zuletzt sei noch die While-Schleife vorgestellt. Unter Python führt sie im Gegensatz zur For-Schleife den Codeblock nur so lange immer wieder aus, wie eine bestimmte Bedingung erfüllt ist. Den schnellsten Einblick ermöglicht auch hier ein Beispiel. Wir löschen die bislang geschriebenen Codezeilen (oder speichern sie unter anderem Namen) und schreiben im Code-Editor:

```
n = 0
while n <= 2:
    print(n)
    n += 1
```

Zunächst ist es hilfreich zu wissen, dass der Ausdruck „n += 1“ dem Ausdruck „n = n + 1“ entspricht und lediglich eine verkürzte Schreibweise unter Python darstellt.

Die While-Schleife wird durchlaufen, solange die Variable „n“ kleiner oder gleich der Zahl 2 ist. In der Konsole ergeben sich bei Ausführung des Programms folglich die zeilenweisen Ausgaben 0, 1, 2.

Die While-Schleife zeigt eine größere Flexibilität als die For-Schleife, sofern die Fortsetzung der Schleife bzw. die Wiederholung des Codeblocks von Entwicklungen abhängig ist, die sich erst im Programmablauf innerhalb des Codeblocks ergeben. Allerdings ist die Gefahr, versehentlich eine unendlich laufende Schleife zu erzeugen, auch größer: Der/die Programmierer/in muss sich selbständig darum kümmern, dass die Schleife auch irgendwann einmal endet – im obigen Beispiel durch das Hochzählen von „n“.

4.8 Strukturierung des Codes: Funktionen und Module

Die Strukturierung des Programmcodes durch die Verwendung von Funktionen und Modulen dient hauptsächlich dazu, die Lesbarkeit sowie Verständlichkeit sicherzustellen. Damit sollen letztendlich die Codepflege und die Fehlersuche effektiver und effizienter gestaltet werden. Bei der Strukturierung durch Module wird der Programmcode auf verschiedene Dateien aufgeteilt. Hingegen finden sich Funktionen innerhalb von Modulen.

Funktionen

Funktionen als eines der wichtigsten Konstrukte von Programmiersprachen wurden bereits früher eingeführt (siehe S. 13). So wurde das hier häufig verwendete „print()“ z.B. auch schon als Funktion charakterisiert: Es hat einen Funktionsnamen („print“), dem runde Klammern folgen. In den runden Klammern werden ggf. Parameter – hier die zu druckenden Inhalte – übergeben. Die Funktion erledigt eine Aufgabe und liefert einen Rückgabewert, in diesem Fall die Ausgabe auf der Konsole.

Schon eingeführte Funktionen unter Python sind in der Sprache definiert. Im Unterschied dazu geht es folgend darum, eigene Funktionen zu schreiben. Dies zahlt sich insbesondere dann aus, wenn die gleiche Aufgabe mehrfach hintereinander bewältigt werden muss: Dann wird einfach die aufgabenerfüllende Funktion wiederholt aufgerufen. Funktionen sind insofern eine Umsetzung des DRY-Prinzips der Programmierung („Don’t repeat yourself“), da sie es ersparen, Codeteile mehrfach zu schreiben.

Eigene Funktionen unter Python beginnen immer mit dem Schlüsselwort „def“. Diesem folgt der Funktionsname, an den sich stets Klammern anschließen. In den Klammern werden Parameter oder Argumente an die Funktion übergeben. Dabei handelt es sich um Informationen, welche die Funktion benötigt, um ihre Aufgabe zu erfüllen. Bei den Argumenten gibt es zum einen erforderliche Argumente, die nicht ausgelassen werden können. Zum anderen gibt es auch optionale Argumente: Sofern diese nicht übergeben werden, werden sie durch einen vorgegebenen Standardwert ersetzt. Nach der schließenden Klammer folgt stets ein Doppelpunkt.

Der Körper einer Funktion ist immer ein Codeblock – und daher eingerückt. Innerhalb des Codeblocks kann es dabei durchaus weitere Codeblöcke (z.B. im Kontext von Schleifen oder Bedingungen) geben, die dann wiederum weiter eingerückt werden. Im Codeblock wird sich bei einer klassischen Funktion oft (einmalig oder mehrfach) die Anweisung „return“ finden. Hinter dieser werden die von der Funktion ggf. zurückzugebenden Werte aufgeführt.

Das war nun reichlich Theorie. Sehen wir uns ein Beispiel an – einen „Klassiker“, die Umrechnung von Temperaturen (von Grad Fahrenheit oder Grad Kelvin in Grad Celsius). Dieses Beispiel findet man oft in Einführungen in die Programmierung. Wir löschen die bislang ge-

schriebenen Codezeilen (oder speichern sie unter anderem Namen) und tragen im Code-Editor ein:

```
def conversion_celsius(degrees, source="fahrenheit"):
    if source.lower() == "fahrenheit":
        return (degrees-32)*(5/9)
    elif source.lower() == "kelvin":
        return degrees-273.15
    else:
        return(f"Kann {source} nicht umrechnen!")
```

Bevor wir die Funktion näher betrachten, benötigen wir noch einen Aufruf – eine Funktion ist nicht selbständig als Code lauffähig, sondern bedarf immer eines Aufrufs aus einem anderen Programmteil (das darf auch wieder eine Funktion sein!). Wir setzen daher mit zwei Leerzeilen Abstand darunter ein „Hauptprogramm“:

```
# ----- Main ------
print(conversion_celsius(20, "Fahrenheit"))
```

Betrachten wir nun zunächst die Funktion: Sie hat zwei Übergabeparameter bzw. Argumente. Das erste mit Namen „degrees“ ist ein Pflichtargument – die umzurechnende Gradzahl. Diese muss bei einem Aufruf stets mit übergeben werden. Das zweite Argument „source“ ist hingegen optional, könnte also auch weggelassen werden. Sofern „source“ nicht angegeben wird, wird der Variablen der Wert „fahrenheit“ zugewiesen. Es handelt sich mithin um die Temperaturskala, aus welcher in Grad Celsius konvertiert werden soll. In Fällen, in welchen bei einem optionalen Argument kein Standardwert festgelegt werden kann oder soll, kann statt der Wertzuweisung auch „=None“ geschrieben werden (das macht hier allerdings keinen Sinn).

Was passiert nun im Weiteren? – Es werden mittels „if“ und „elif“ Fallunterscheidungen nach den beiden hier berücksichtigten Eingabe-Temperaturskalen (Fahrenheit und Kelvin) vorgenommen. Hinter einem „return“ erfolgt die entsprechende Berechnung, deren Ergebnis von der Funktion zurückgegeben wird. Der übergebene Wert von „source“ wird bei der Fallunterscheidung mittels der Methode „.lower()“ in Kleinbuchstaben gewandelt. Damit soll eine abweichende Groß-/Kleinschreibung bei der Eingabe durch den Nutzer abgefangen und ggf. korrigiert werden – trotz dieser soll die Funktion ihre Aufgabe erfüllen können.

Das am Ende der Fallunterscheidungen stehende „else:“ fängt von „fahrenheit“ und „kelvin“ abweichende Werte der Variablen „source“ ab und liefert unter Einsatz der „f“-Schreibweise einen Hinweis auf den nicht verarbeitbaren Wert von „source“ auf der Konsole.

Beim Funktionsaufruf im „Hauptprogramm“ (unter „Main“) wird der Rückgabewert der Funktion nicht auf eine Variable zugewiesen, sondern die Funktion direkt in einen „print()“-Befehl eingebettet. Damit wird der Rückgabewert direkt auf der Konsole ausgegeben.

Für den Funktionsaufruf gibt es noch mögliche Variationen: Hier erfolgte er im Hauptprogramm als:

```
print(conversion_celsius(20, "Fahrenheit"))
```

Damit wurden die Argumente als sogenannte „Positionsargumente“ übergeben. Dabei werden die Werte entsprechend ihrer Reihenfolgeposition den von der Funktion erwarteten Argumenten (hier: erst „degrees“, dann „source“) zugewiesen. Die Alternative ist ein Aufruf mit „Schlüsselwortargumenten“. Dieser bindet die Argumente bzw. deren Variablenbezeichnungen explizit mit in den Funktionsaufruf ein. Ein solcher könnte z.B. hier lauten:

```
print(conversion_celsius(source = "kelvin", degrees = 0))
```

Hier werden beim Funktionsaufruf die Argumente mit ihren Namen bzw. Variablenbezeichnungen übergeben. Sie erscheinen jetzt in anderer Reihenfolge als oben. Das ist kein Problem, da sie ja aufgrund der verwendeten Namen in der Funktion klar zugeordnet werden können.

Gültigkeitsbereiche der Variablen

Bislang gab es keinen Grund, über die Gültigkeitsbereiche von Variablen nachzudenken. Eine Variable galt bislang im gesamten von uns geschriebenen Programm. Mit der Strukturierung des Codes ändert sich das nun.

Variablen sind in Python grundsätzlich lokaler Natur, sind lokale Variablen. Eine in einer Funktion A verwendete Variable „a“ z.B. gilt nur innerhalb dieser Funktion. Legen wir eine zweite Funktion B an und arbeiten hier wieder mit einer Variablen „a“, so haben die beiden „a“s aus Funktion A und B nichts miteinander zu tun (geschweige denn, dass sie den gleichen Wert hätten). Auch auf Modulebene (z.B. im oben verwendeten „Hauptprogramm“) gilt dies: Ein Modul X möge aus den

Funktionen A und B sowie einem kleinen Hauptprogramm zu deren Aufruf bestehen (analog zur obigen Konstruktion). Eine im Hauptprogramm verwendete Variable „a“ hätte nun wiederum nichts mit den gleichnamigen Variablen in den Funktionen A und B zu tun. Auch in einem anderen Modul Y gibt es diese Variable nicht. Variablen in Funktionen sind funktionslokal, Variablen in Modulen modullokal. Gleiches gilt für Datenstrukturen (wie z.B. Listen), die als Erweiterungen von Variablen anzusehen sind.

„Globale“ Variablen, die überall (in jeder Funktion und jedem Modul des Programms) gelten, sind möglich, aber als schlechter Programmierstil verpönt. Globale Variablen können viele Fehler verursachen, für die sich die Fehlersuche (das „Debugging“) schwierig gestaltet. Eine strenge Lokalität ist für Anfänger/innen vielleicht zuerst gewöhnungsbedürftig, aber langfristig von Vorteil.

Module

Im Rahmen dieser Einführung wurde bislang in der Datei „grundlagen.py“ gearbeitet, die am Anfang angelegt wurde. Für ein sehr großes Programmierprojekt ist das Arbeiten in nur einer einzigen Datei aber in der Regel unpraktisch. Es entstehen vielleicht viele tausend Zeilen Code, hunderte von Funktionen … das wird schnell unübersichtlich – obwohl man Funktionen unter PyCharm sehr schön „einklappen“ kann (versuchen Sie es mal im Editor mit der vorhin erstellten Funktion!). Trotzdem wäre die Arbeit in einer einzigen Datei bei einem großen Projekt unbequem, würde die Fehlerwahrscheinlichkeit erhöhen und wäre schlecht zu pflegen.

Daher teilt man große Programmierprojekte üblicherweise auf eine Anzahl an Dateien auf, die im gleichen Programmverzeichnis liegen. Jede Datei stellt dann ein (eigenerstelltes) Modul dar (zu den fremderstellten Modulen kommen wir exemplarisch im Abschnitt 4.9).

Viele Programmierer/innen arbeiten mit einer Hauptdatei, die z.B. „main.py“ heißen könnte und die zentralen Steuerungsmechanismen für das Programm enthält. Die erstellten Funktionen werden oft nach Einsatzbereichen gruppiert und dann in Module, also eigene *.py-Dateien, ausgelagert. Dabei gilt als Faustregel, dass ein Modul nur so umfangreich sein sollte, dass sich Verwendungszweck und Inhalte noch mit ein bis zwei Sätzen beschreiben lassen. Viele Programmierer/innen schreiben diese dann auch als Kommentar an den Anfang des Moduls.

Die Programmierumgebung bzw. Python muss nun natürlich irgendwie erfahren, dass alle Module zu einem einzigen Programm gehören. Zu diesem Zweck werden Module *importiert*. Auch hier bleibt alle Theorie grau – daher wollen wir die Nutzung eines Moduls nun exemplarisch umsetzen. Wie schließen zunächst die bislang verwendete Codedatei „grundlagen.py“.

Mit den Unterlagen zur Veranstaltung wird eine Datei „temperaturen.py“ bereitgestellt. Diese wird nun in das Programmverzeichnis kopiert. Durch einen Doppelklick auf den Dateinamen in der Projektübersicht links im PyCharm-Fenster wird die Datei im Editor geöffnet. Man sieht, dass es sich im Wesentlichen um die oben entwickelte Funktion „convert_celsius()“ handelt:

```
TEMPERATUR_SKALEN = ("fahrenheit", "kelvin", "celsius")

def convert_celsius(degrees, source="fahrenheit"):
    if source.lower() == "fahrenheit":
        return (degrees-32) * (5/9)
    elif source.lower() == "kelvin":
        return degrees - 273.15
    else:
        return f"Kann {source} nicht umrechnen."

print("Dies ist das Temperaturmodul.")
```

Wir lassen nun den Code in diesem Modul einmal ausführen. Dabei ist darauf zu achten, dass links neben dem kleinen grünen Dreieck unter PyCharm im DropDown „Current File“ ausgewählt ist. In der Konsole wird ausgegeben:

```
Dies ist das Temperaturmodul.
```

Ausgeführt wird also nur der Modulcode ganz unten (das „print()“), nicht aber die Funktion. Dies ist auch folgerichtig, da die Funktion im Modulcode nicht aufgerufen wird. Der Modulcode wurde hier nur zur Demonstration eingefügt – normalerweise wird in Modulen kein Modulcode eingefügt, sondern es finden sich dort nur Funktionen.

Diese Datei „temperaturen.py“ wollen wir im Weiteren als Modul verwenden. Als zentrale Datei unseres Programmierprojektes legen wir die Datei „main.py“ neu im Programmverzeichnis an (kurz: „File“ – „New“ – „Python File“ – Namen „main“ eintragen). Diese neue Datei „main.py“ öffnen wir dann durch einen Doppelklick im Editor und tragen ein:

```
import temperaturen
```

Mit dieser Zeile wird das Modul „temperaturen“ importiert. Führen wir nur den Code (auf „Current File“ achten!) aus, so erscheint in der Ausgabe auf der Konsole wieder der Text „Dies ist das Temperaturmodul.“. Dies ist die Folge der Existenz des Modulcodes im Modul. Im Modul „temperaturen.py“ findet sich – auch außerhalb jeglicher Funktionen – ganz oben noch die Definition einer Liste als Code mit Zuweisung auf eine Variable:

```
TEMPERATUR_SKALEN = ("fahrenheit", "kelvin", "celsius")
```

Auch diese Variablenzuweisung wird beim Import des Moduls mit übernommen. Wir ergänzen im Modul „main.py“ den Code um die Zeile:

```
print(temperaturen.TEMPERATUR_SKALEN)
```

Wird der Code im Modul „main.py“ erneut ausgeführt, so wird die Liste – welche hier mit dem vorangestellten Namen des Moduls als Variable angesprochen wird – auch mit auf der Konsole angedruckt:

```
Dies ist das Temperaturmodul.
('fahrenheit', 'kelvin', 'celsius')
```

Beim Import werden also auch die Variablen übernommen und sind dann (mit vorangestelltem Modulnamen und einem Punkt separiert) im Modul „main.py“ verfügbar. – In der Regel wird man das jedoch nicht wünschen. In der Regel finden sich in den Modulen nur Funktionen und kein eigenständiger Code. Wir modifizieren den „print()“-Befehl unter „main.py“ und nutzen die Funktion für eine Temperaturumrechnung:

```
print(temperaturen.convert_celsius(120, "fahrenheit"))
```

Bei Ausführung des Codes erscheint als Ausgabe im Konsolenfenster:

```
Dies ist das Temperaturmodul.
48.88888888888889
```

Wiederum wird durch den Import auch der Modulcode ausgeführt und dessen Ausgabe erscheint neben der eigentlich gewünschten Ausgabe der umgerechneten Temperatur. Beim Import wird eventuell existierender Modulcode immer von oben nach unten mit abgearbeitet. Funktionen werden jedoch ignoriert. Dem Funktionsaufruf selbst wird im Code der Name des Moduls – abgetrennt mit einem Punkt – vorangestellt.

Bei langen Modulnamen ist es nun etwas anstrengend, immer den langen Modulnamen mit vor die genutzten Funktionen zu setzen. Daher gibt es eine Vereinfachung – man kann einen kürzeren Alias für den Modulnamen definieren. Die Importanweisung am Anfang des Moduls „main.py“ wird modifiziert zu:

```
import temperaturen as tp
```

Im Weiteren ist es dann nicht mehr notwendig, jedes Mal „temperaturen.“ vor einen Funktionsaufruf aus dem Modul zu setzen. Dies kann zu „tp.“ verkürzt werden.

Wichtig ist es, darauf zu achten, dass sich Modulnamen wie auch vergebene Aliase niemals doppeln dürfen. Module müssen – auch als Aliase – eindeutig benannt werden. Dies wird insbesondere dann wichtig, wenn wir neben eigenerstellten Modulen auch noch fremderstellte Module (von denen es eine unüberschaubare Zahl gibt) verwenden (siehe mit einer solchen Verwendung Abschnitt 4.9).

Da Modulcode – wie nun mehrfach gezeigt – unsinnig ist, entfernen wir ihn aus dem Modul „temperaturen.py“. Öffnen Sie die Datei und entfernen Sie die erste und letzte Zeile, so dass nur noch die Funktion „convert_celsius()“ verbleibt!

Zuweilen will man nicht ein ganzes Modul importieren, sondern nur eine genau ausgewählte Funktion aus diesem nutzen. Dann bietet es sich an, nicht mehr das gesamte Modul zu importieren, sondern nur die gewünschte Funktion daraus. Exemplarisch wird die erste Zeile mit dem Import in „main.py“ erneut modifiziert zu:

```
from temperaturen import convert_celsius
```

Nun wird nicht mehr das gesamte Modul „temperaturen“ importiert, sondern nur noch die Funktion „convert_celsius()“ aus diesem. Allerdings muss nun auch die Temperaturumrechnung im „print()“-Befehl angepasst werden:

```
print(convert_celsius(120, "fahrenheit"))
```

Die Funktion „convert_celsius()“ kann nun direkt angesprochen werden. Das vorangestellte „temperaturen.“ oder „tp.“ ist nicht mehr erforderlich. Bei Ausführung des Codes im Modul „main.py“ erscheint auf der Konsole das Ergebnis der Umrechnung:

```
48.88888888888889
```

Anmerkung am Rande: Den Modulcode haben wir oben gerade erst aus „temperaturen.py“ entfernt. Sofern er noch da wäre, würde er aber – trotzdem nur eine Funktion aus dem Modul importiert wurde – wieder mit ausgeführt …

Auch hier ist auf die Namen zu achten: Sofern in „main.py“ auch eine Funktion „convert_celsius()“ existieren würde, käme es zu einem Namenskonflikt und damit einer Fehlermeldung. Solange hingegen vollständige Module importiert werden, sind gleichnamige Funktionen kein Problem, da über den vorangestellten Modulnamen oder dessen Alias immer noch eine Differenzierung möglich ist.

4.9 Exemplarische Nutzung des Fremdmoduls „datetime“

Im vorherigen Abschnitt wurde gezeigt, wie die eigene Programmierung in Modulen organisiert werden kann. Der Fokus lag damit auf selbsterstellten Modulen.

Eine wesentliche Stärke der Programmiersprache Python ist es allerdings, dass es eine nahezu unüberschaubare Anzahl an Fremdmodulen für alle möglichen Zwecke gibt. Frei nach dem Motto: Wenn ich mir eine bestimmte generelle Aufgabe auch nur ausdenken kann, dann gibt es vermutlich auch schon ein Modul dafür …

Diese Fremdmodule werden oft auch als „Packages“ oder „Pakete“ bezeichnet. Die wichtigsten Fremdmodule sind bereits Bestandteil der Programmiersprache Python (als sogenannte Standardbibliotheken). Sie können jederzeit über die „import“-Anweisung eingebunden und genutzt werden. Jene Pakete, die nicht direkter Bestandteil von Python sind, aber als freie Software zur Verfügung stehen („Open Source“), sind in großen Paketbibliotheken („Repositories“) organisiert. PyCharm als integrierte Entwicklungsumgebung stellt auch gleich einen Paketmanager bereit und bietet z.B. den direkten Zugriff auf das wichtige Repository „PyPI“ („Python Package Index“). Der Paketmanager kann dann das Paket herunterladen und es für die Nutzung bereitstellen (dazu kommt es noch im weiteren Verlauf der Veranstaltung). Im Kontext anderer Entwicklungsumgebungen wird oft das originäre Paketverwaltungsprogramm „pip“ genutzt – deshalb sollte man schon einmal davon gehört haben und diese Bezeichnung einordnen können.

In diesem Abschnitt soll exemplarisch mit einem Fremdmodul aus der Standardbibliothek gearbeitet werden. Es handelt sich um das Paket „datetime“, welches umfassenden Funktionalitäten bereitstellt, um mit oft relevanten Datums- und Zeitdaten zu operieren. Andere oft genutzte Pakete in diesem Kontext sind „time“ und „calendar“. Auch diese sind Bestandteil der Standardbibliothek. Auf „datetime“ wird auch vom Paket „pandas“ (ein Paket zur Datenhandhabung und -analyse) aus, so z.B. für die Zeitreihenanalyse, zurückgegriffen. Das Paket „pandas“ wird im weiteren Verlauf der Veranstaltung noch intensiver genutzt werden.

Wir öffnen unter PyCharm wieder die Codedatei „grundlagen.py“ (und schließen andere evtl. noch offene Dateien). Sofern sich in „grundlagen.py“ noch Code findet, löschen wir diesen (oder speichern ihn unter anderem Namen). Die Nutzung des Fremdmoduls „datetime“ beginnt mit dessen Import:

```
# Das datetime-Modul mit Alias "dt" importieren
import datetime as dt
```

Dabei wird auch gleich ein Alias namens „dt“ vergeben, damit sich die weiteren Codezeilen schneller tippen lassen. „datetime“-Objekte müssen „instanziiert“, also durch Aufruf erzeugt werden. Wir erzeugen zwei Objekte als Zeitangaben/Zeitmarken und weisen sie auf Variablen zu. Der Code wird nach zwei Leerzeilen ergänzt mit:

```
# Zwei datetime-Objekte instanziieren
zeitmarke1 = dt.datetime(2030,1,31,14,30)
zeitmarke2 = dt.datetime.today()
print(zeitmarke1)
print(zeitmarke2)
```

Etwas verwirrend mag jetzt sein, dass im Modul „datetime“ wiederum eine Klasse (Methodengruppe) „datetime“ existiert, welche hier für die Generierung der Zeitmarken genutzt wird. Der ersten Zeitmarke wurde ein Datum samt Uhrzeit (31.01.2030, 14:30 Uhr) zugewiesen. Die zweite Zeitmarke übernimmt das aktuelle Datum mit der aktuellen Uhrzeit. Bei Ausführung des Codes erscheint auf der Konsole die folgende Ausgabe:

```
2030-01-31 14:30:00
2023-09-06 11:50:08.734156                (bei Ihnen anders …!)
```

Es fällt auf, dass beim aktuellen Datum mit aktueller Uhrzeit auch Sekundenbruchteile mit ausgewiesen werden.

„datetime“ bietet auch verschiedene Möglichkeiten, um Datumsbestandteile als Attribute eines Datums zu extrahieren. Wir verändern die beiden letzten „print()“-Anweisungen wie nachstehend:

```
# Tag abrufen
print(zeitmarke1.day)
print(zeitmarke2.weekday())
```

Ausgegeben werden in zwei einzelnen Zeilen zunächst die Zahl 31 und dann die Zahl 2. „.day“ liefert offensichtlich den Kalendertag des Monats zurück, während der „.weekday()“ als Methode den Mittwoch mitteilt – die Wochentage werden ab dem Montag nullbasiert gezählt, womit Werte von 0 bis 6 möglich sind.

In Anwendungskontexten besteht oft ein Interesse an der Differenz zweier Zeitpunkte (wie lange hat z.B. die Erledigung einer bestimmten Aufgabe gedauert?). Eine Zeitdifferenz lässt sich (wie z.B. auch bei Excel) einfach durch die Subtraktion des späteren vom früheren Zeitpunkt bestimmen. Wir löschen die drei letzten Codezeilen und tragen neu ein:

```
# Zeitdifferenzen bestimmen
delta = zeitmarke1 - zeitmarke2
print(delta)
```

Bei Ausführung wird das Ergebnis in Tagen (sowie Stunden, Minuten, Sekunden) in der Konsole ausgegeben:

```
2338 days, 22:44:50.560591
```

Mit solchen Zeitdifferenzen kann auch gerechnet werden. Wir löschen die drei letzten Zeilen Code und schreiben unter Verwendung der Methode „.timedelta()“ neu:

```
# Rechnen mit Zeitdifferenzen
zeitmarke3 = zeitmarke2 + dt.timedelta(days=4, hours=1, minutes=1)
print(zeitmarke3)
```

In der Konsole wird bei Ausführung des Codes das aktuelle Datum mit aktueller Zeit zuzüglich vier Tagen, einer Stunde und einer Minute ausgegeben:

```
2023-09-10 16:51:16.648641
```

Schließlich lassen sich „datetime“-Objekte natürlich auch als Zeichenketten „hübscher“ bzw. in frei wählbarer Darstellung ausgeben. Wir löschen die drei letzten Zeilen Code und tragen ein:

```
# Formatierung als Zeichenketten
zeitstr = zeitmarke2.strftime("%d.%m.%Y %H:%M")
print(zeitstr)
print(f"{zeitmarke2:%d.%m.%Y %H:%M}")
```

Bei Ausführung des Codes ergibt sich auf der Konsole für beide „print()“-Befehle die gleiche Ausgabe:

```
06.09.2023 16:00
06.09.2023 16:00
```

Im obigen Code wird zunächst die „zeitmarke2“ mittels der Methode „.strftime()“ formatiert und auf eine Variable „zeitstr“ als String zugewiesen. Dabei wird in einer als Argument übergebenen Maske (in Anführungszeichen) die gewünschte Darstellung übergeben. „%d“ steht dabei z.B. für die Tage, „%M“ z.B. für die Minuten. Beim „print()“ in der letzten Zeile wird mittels der „f“-Schreibweise die Maske direkt innerhalb des „print()“ genutzt, um die gewünschte Formatierung herbeizuführen.

Letztlich kann auch eine Zeichenkette (die z.B. aus einer Datei eingelesen wurde) in ein „datetime“-Objekt gewandelt werden (diesen Vorgang bezeichnet man auch als „parsen“). Wir eliminieren die letzten vier Codezeilen im Editor und tragen neu ein:

```
# String in datetime-Objekt wandeln
zeitmarke4 = dt.datetime.strptime("12.01.2026", "%d.%m.%Y")
print(zeitmarke4)
```

Bei Ausführung des Codes ergibt sich als Ausgabe auf der Konsole:

```
2026-01-12 00:00:00
```

Hier wird die Methode „.strptime()“ innerhalb der Klasse „datetime“ verwendet, um den String in ein „datetime“-Objekt zu überführen. Als Argument wird dabei zunächst die zu konvertierende Zeichenkette übergeben. Mit dem zweiten Argument wird deren Aufbau beschrieben, anhand dessen dann die Konvertierung durchgeführt wird. Dabei werden die gleichen Schlüssel (z.B. „%d“ für den Tag des Monats) verwendet, die auch oben bereits bei der Ausgabe herangezogen wurden.

Das „datetime“-Modul bietet noch zahlreiche weitere Funktionalitäten zur Handhabung von Zeit- und Datumsinformationen. Diese sind auch oft für Aufgaben des Data Science, die mit Python bewältigt werden sollen, von Relevanz. An dieser Stelle jedoch sollen die bislang gewonnenen Einblicke hinreichen.

4.10 Schreiben in und Lesen aus Textdateien

Bislang wurden zu verarbeitende Informationen über die „input()"-Funktion in die Programme eingegeben oder – wenig realitätsgetreu – in den Code eingebettet. Größere Informationsmengen liegen allerdings oft in Form von Dateien vor, welche uns geliefert werden. Genauso wird man nicht immer alle Ausgaben nur von der Konsole ablesen wollen, sondern auch zuweilen den Wunsch haben, diese dauerhafter in Dateien zu speichern.

Auch wenn später aus anderen Dateitypen (wie z.B. Excel-Dateien) gelesen oder in solche geschrieben wird, bietet das Schreiben und Lesen von einfachen Textdateien schon wesentliche Einblicke in die entsprechenden Operationen und ist als Einführung gut geeignet.

Für den Anfang ist es dabei am einfachsten, die zu lesenden Dateien in das Projektverzeichnis (hier: „…\grundlagen\") zu legen bzw. Dateien dorthin zu schreiben. Das Projektverzeichnis ist jenes Verzeichnis, in welchem auch die bearbeiteten *.py-Dateien liegen. Dateien, die im Projektverzeichnis liegen, können direkt mit dem Dateinamen angesprochen werden. Mit der Angabe von Pfaden zu anderen Dateiorten machen wir uns an späterer Stelle in dieser Veranstaltung vertraut.

Einlesedatei

Sofern eine Textdatei mit Python eingelesen werden soll, wird zunächst eine Textdatei benötigt. Diese soll hier direkt unter PyCharm im Projektverzeichnis angelegt werden. Sofern links das Projektverzeichnis mit den Programmdateien (*.py) geöffnet ist, kann über das Menü mit „File" – „New" – „File" der Dialog zur Benennung einer neu anzulegenden Datei aufgerufen werden. Dort wird nun „einlesen.txt" als Dateiname eingetragen und mit der Eingabetaste bestätigt. Die neue Datei „einlesen.txt" erscheint links unter den Projektdateien. Durch einen Doppelklick kann sie im Editor von PyCharm zur Bearbeitung geöffnet werden. Wir geben drei Textzeilen ein:

```
Testeinlesedatei – erste Zeile
Testeinlesedatei – zweite Zeile
Testeinlesedatei – dritte Zeile
```

Die Dateiinhalte können mit der Tastenkombination „strg + s" gespeichert werden. Auch wenn die Datei einfach mit dem Schließknopf oben am Reiter geschlossen wird, werden die Inhalte gespeichert. Wir

schließen die Datei „einlesen.txt“ – die Argwöhnischen unter uns können ja noch einmal prüfen, ob der gewünschte Text tatsächlich in der Datei zu finden ist.

Datei einlesen mit „.readline()“

Im ersten Schritt soll nun zeilenweise aus der Datei „einlesen.txt“ gelesen werden. Wir öffnen unter PyCharm wieder die Codedatei „grundlagen.py“. Sofern sich in dieser noch Code findet, löschen wir diesen (oder speichern ihn unter anderem Namen). Dann tragen wir nachstehenden Code ein:

```
# ----- Lesen aus Datei mit "readline" -----
datei = input("Dateiname? ")
dateinr = open(datei)
inhalt = dateinr.readline()
print(inhalt)
dateinr.close
```

Nach einer Kommentarzeile wird zunächst beim Benutzer/der Benutzerin des Programms der Dateiname mittels eines „input()“ abgefragt und auf die Variable „datei“ zugewiesen. Im nächsten Schritt wird mittels der Anweisung „open()“ die Datei geöffnet. Dabei wird als Argument die Variable „datei“, also der abgefragte Dateiname, übergeben. Existiert eine Datei mit dem eingegebenen Namen nicht, so wird das Programm hier mit einer Fehlermeldung abbrechen. Haben Sie eine Idee, wie man einen solchen Abbruchfehler abfangen könnte? – Falls nicht, blättern Sie einmal bis zum Try-Except-Konstrukt zurück …

Die Anweisung „open()“ ist wieder eine Funktion. Diese hat bei erfolgreichem Öffnen der Datei auch einen Rückgabewert. Dieser Rückgabewert („file handle“, „file object“) identifiziert die geöffnete Datei und erlaubt weitere Operationen mit dieser. Er wird hier auf die Variable „dateinr“ zugewiesen (obgleich es streng genommen keine Nummer ist … Sie können den Inhalt ja einmal mit „print()“ auf die Konsole ausgeben lassen …).

Die Datei ist nun geöffnet – genauer gesagt: Im Lesemodus, für das Einlesen von Inhalten aus der Datei, geöffnet. Der vollständige „open()“-Befehl müsste an dieser Stelle lauten: „open(datei, 'r')“. Das Öffnen zum Einlesen ist allerdings der sogenannte Standardfall (der „default“) – sofern man also das 'r' weglässt, denkt es sich Python automatisch dazu, da dies ein optionales Argument der Funktion ist.

Nun kann eingelesen werden. Dafür wird in diesem ersten Beispiel die Methode „.readline()“ verwendet. Diese Methode wird auf die Datei, welche durch die Variable „dateinr“ repräsentiert wird, angewandt. Das Ergebnis des Auslesens wird auf die Variable „inhalt“ zugewiesen. „.readline()“ liest, wie es der Name schon andeutet, nur eine einzige Zeile aus der Datei – genau bis zum ersten Zeilenumbruch in derselben.

Diese erste Zeile wird folgend mit dem „print()“-Befehl auf der Konsole ausgegeben – damit wir sehen können, was passiert ist.

Grundsätzlich sollte man – auch im echten Leben – immer „hinter sich“ aufräumen. Beim Programmieren gehört dazu, dass man eine Datei, die man geöffnet hat, auch wieder schließt, wenn sie nicht mehr benötigt wird. Dies passiert hier in der letzten Zeile des Codes. Die Methode „.close“ (schließen) wird auf die geöffnete Datei angewandt, welche wiederum durch die Variable „dateinr“ repräsentiert wird.

Bei Ausführung des Codes (achten Sie dabei immer auf das „Current File“!) erscheint nachstehende Ausgabe auf der Konsole:

```
Dateiname? einlesen.txt
Testeinlesedatei - erste Zeile
```

In der ersten Zeile der Ausgabe wurde zunächst nach dem Dateinamen für die einzulesende Datei gefragt. Dieser wurde hier (korrekt) eingegeben. In der zweiten Zeile findet sich die Ausgabe der ersten Zeile aus der einzulesenden Datei.

Im weiteren Verlauf der Veranstaltung – so auch am kommenden zweiten Tag – wird es immer wieder darauf ankommen, dass Sie die einzelnen „Wissensbausteine“ aus dieser Einführung miteinander kombinieren können. In diesem Sinne könnte z.B. eine Aufgabe lauten: Lesen Sie die Zeilen der Datei unter Verwendung einer Schleife auf eine Liste ein! Zur Lösung dieses Problems müssen Sie einerseits in früheren Abschnitten vermitteltes Wissen (Was ist eine Liste? Wie funktioniert eine Schleife?) heranziehen, andererseits werden Sie vermutlich noch zusätzliche Informationen benötigen … und wen fragen Sie dann? Richtig, „Tante Google“ …

Datei einlesen mit „.read()“

Es gibt unter Python Alternativen zur Methode „readline()“, so z.B. das „.read()“. Wir modifizieren die Kommentarzeile unseres bisherigen Codes zu:

```
# ----- Lesen aus Datei mit "read" -----
```

Darüber hinaus gehen wir in die vierte Zeile des Codes und entfernen das „line“ aus dieser, so dass sie wie nachstehend aussieht:

```
inhalt = dateinr.read()
```

Das „.read()“ frisst die Datei mit „Haut und Haaren“ – wenn der Code nun ausgeführt wird, erscheint auf der Konsole die nachstehende Ausgabe:

```
Dateiname? einlesen.txt
Testeinlesedatei - erste Zeile
Testeinlesedatei - zweite Zeile
Testeinlesedatei - dritte Zeile
```

„.read()“ liest also die gesamte Datei – unabhängig von Zeilenumbrüchen – in ihrer Gesamtheit auf die Variable „inhalt“ ein – allerdings nur, falls sie nicht zu groß für den freien Speicher ist (ein Limit, an welches man eher selten stößt – dann gibt es einen „MemoryError“, der natürlich auch wieder mit dem Try-Except-Konstrukt abgefangen werden kann).

Ob es vorteilhafter ist, eine Textdatei zeilenweise zu lesen oder sie „am Stück“ in eine Variable zu überführen, hängt allein vom Anwendungszweck ab. Da wir hier noch recht zweckfrei unterwegs sind, gibt es also vorläufig noch keine Antwort dazu.

Datei schreiben mit „.write()“

Nun soll einmal in eine Datei geschrieben werden. Zu diesem Zweck wird der aktuelle Code in „grundlagen.py“ gelöscht (oder unter anderem Namen gespeichert). Als neuer Code wird eingetragen:

```
# ----- Schreiben in Datei mit "write" -----
dateinr = open("ausgabe.txt", 'w')
```

Wir verzichten nun auf die Abfrage des Namens der zu schreibenden Datei und setzen diesen direkt („hart“) in den Code. Als Öffnungsmodus der Datei wird nun „w“ gewählt – dies steht für „write“, also „schreiben“. Es gibt noch eine Reihe weiterer Modi – einer davon, das „a“ (für „append“), wird gleich noch präsentiert werden. Zur Identifikation der Datei wird die bekannte Variable „dateinr“ verwendet. Wir tragen weiteren Code ein:

```
dateinr.write("Tim"+"\n")
dateinr.write("Theodor"+"\n")
```

```
dateinr.close
```

Das „.write()“ ist hier die Entsprechung des „print()“ für die Konsole – es gibt die folgend als Argument in Klammern übergebenen Inhalte in eine Datei aus. Dabei sind „Tim“ und „Theodor“ die Texte, die wir ausgeben wollen – das jeweils nachgesetzte „\n“ ist ein Steuerungszeichen, welches für einen Zeilenumbruch sorgt. Probieren Sie es später einfach einmal ohne diese „\n“ aus und sehen Sie, was passiert!

Wenn wir den Code ausführen, passiert auf der Konsole … nichts! Das ist auch wenig verwunderlich, da es auch keine Ausgabe auf die Konsole gibt. Links unter PyCharm im Projektverzeichnis sollte jedoch eine neue Datei „ausgabe.txt“ erscheinen. Diese können wir mit einem Doppelklick auch öffnen und den Inhalt ansehen. In der Datei sollten in zwei Zeilen die obigen männlichen Vornamen erscheinen.

Anhängen an Datei mit „.write()“ im „append“-Modus

Eine schon bestehende Textdatei kann durch weitere Inhalte ergänzt werden. Dieses „Anhängen“ („append“) soll folgend demonstriert werden. Wir modifizieren zunächst die Kommentarzeile zu:

```
# ----- Anhängen an Datei mit "append" / "a" -----
```

Folgend wird das Öffnen der Datei wie nachstehend geändert:

```
dateinr = open("ausgabe.txt", "a")
```

Das kleine „w“ für den Modus „write“ wurde nun durch ein kleines „a“ für den Modus „append“ (anhängen) ersetzt. Würde man erneut eine Datei gleichen Namens mit einem „w“ schreiben, so würde die alte Datei überschrieben. Im weiteren soll nur noch eine (statt zwei) Zeilen herausgeschrieben werden – die „.write()“-Methode wird dabei genauso eingesetzt wie im ersten Schreibbeispiel:

```
dateinr.write("Thekla"+"\n")
dateinr.close
```

Abschließend wird die Datei geschlossen. Führt man das veränderte Programm nun aus, so ergibt sich erneut keine Ausgabe auf der Konsole. Ein Blick in die Datei offenbart jedoch, dass nun der Name „Thekla“ in der letzten Zeile zu finden ist, mithin an die Datei angehängt wurde:

```
Tim
Theodor
```

```
Thekla
```

Soweit zunächst die Ausführungen zum Schreiben in und Lesen aus Dateien. Natürlich gibt es auch hier noch etliches mehr an sinnvollem Wissen – das werden wir uns allerdings dann aneignen, wenn wir es benötigen.

Wie geht es weiter?

Hier endet nun der Grundlagenteil mit den ersten Schritten in Python. Im entsprechenden Modul 04 der Veranstaltung „Programmierung mit Python“ hören Sie unter dem Gliederungspunkt 4.11 noch einiges zum „Style Guide“ PEP in der Version 8 – das wird hier allerdings nicht mehr dargestellt.

Am zweiten Tag der Veranstaltung gilt es, das hier präsentierte Grundlagenwissen in konkrete Lösungen vorgegebener Probleme umzusetzen – ggf. unter Erwerb weiteren Wissens („Tante Google“ wird gerne helfen!). Dafür stehen Ihnen (bis zu) sechs Aufgabenstellungen zur Verfügung, welche Sie allein oder in Gruppen bearbeiten sollen …