Python 3 für Einsteiger

Eine kompakte Zusammenfassung der wichtigsten Grundlagen.

Basierend auf dem *Openbook Python 3* von Johannes Ernesti und Peter Kaiser, erschienen im Rheinwerk Verlag.

Version 0.1
Oktober 2020

Typesetting by LaTeX

Inhaltsverzeichnis

1 Über Python

1.1 Entstehung

Die Programmiersprache Python wurde Anfang der 1990er-Jahre von dem Niederländer Guido van Rossum am Centrum voor Wiskunde en Informatica (CWI) in Amsterdam entwickelt. Mittlerweile hat sich Python zu einer der beliebtesten Programmiersprachen entwickelt, die mächtig und zugleich leicht zu erlernen ist.

1.2 Grundlegende Konzepte

Obwohl Python viele Sprachelemente gängiger Skriptsprachen implementiert, handelt es sich um eine **interpretierte** und **plattformunabhängige** Programmiersprache. Der Unterschied zwischen einer Programmier- und einer Skriptsprache liegt im **Compiler**. Ähnlich wie `Java` oder `C#` verfügt Python über einen Compiler, der aus dem Quelltext ein Kompilat erzeugt, den sogenannten **Byte-Code**. Dieser Byte-Code wird dann in einer virtuellen Maschine, dem Python-Interpreter, ausgeführt.
Bei der Installation von Python erhält man neben dem Interpreter und dem Compiler auch eine umfangreiche Standardbibliothek und mit der `python-API` eine Schnittstelle um C-Programme einzubinden (für Hardware-Nahe oder performante Code Teile).

1.3 Hilfe und Dokumentation

Man findet eine Vielzahl von Python Tutorials und Foren-Einträgen im WWW. Leider kann reines *google'n* oft recht Zeitaufwendig werden, weil man meist in einem Forum landet und sich erst einlesen muss. Aus diesem Grund sei an dieser Stelle auf die sehr gute offizielle Doku von Python verwiesen:
`https://docs.python.org/3/`
im Speziellen auf die **Library Reference** in der die Funktionen der Standard Library dokumentiert sind:
`https://docs.python.org/3/library/index.html`

2 Python Interpreter - interaktiver Modus

Startet man den Python-Interpreter ohne Argumente, gelangt man in den sogenannten interaktiven Modus. Dieser Modus bietet dem Programmierer die Möglichkeit, Kommandos direkt an den Interpreter zu senden, ohne zuvor ein Programm erstellen zu müssen. Der interaktive Modus wird häufig genutzt, um schnell etwas auszuprobieren oder zu testen.
Zum Starten öffnet man einen Linux Terminal oder die Windows Commandline und gibt `python3` ein:
Im interaktiven Modus kann man dann wie in einem Programm z.B. Variablen deklarieren und Werte zuweisen oder einfache Berechnungen ausführen.

```
markus@pc:~$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 10
>>> y = 20
>>> z = x + y
>>> print(z)
30
>>> z
30
>>> (12*4)/3
16.0
>>> quit()
markus@pc:~$
```

3 Ein Programm schreiben

Im Gegensatz zum interaktiven Modus, der eine wechselseitige Interaktion zwischen Programmierer und Interpreter ermöglicht, wird der Quellcode eines Programms in eine Datei geschrieben. Diese wird als Ganzes vom Interpreter eingelesen und ausgeführt.
Wir verwenden zum Schreiben der Programme einen einfachen Texteditor mit Syntax-Hightlighting wie z.B. Notepad++
(Aufwändige IDEs (**I**ntegrated **D**evelopment **E**nvironment) bitten zwar viele Vorteile wie z.B. Auto-Completion, Versionskontrollen, Debugging-Features, usw. sind aber für Einsteiger eher verwirrend und überladen)
Für die Programmfiles verwenden wir die File-Endung `.py`
Nenne wir unser erstes Programm z.B: 01.py

Programm 01.py:

```
'''
Das erste Python-Programm - traditionel
ein Hello-World :)
Markus
Okt.2020
'''

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

Liefert folgenden Output:

```
>python3 01.py
Hello-World!
```

3.1 Syntax und best practice

Das Wort Syntax kommt aus dem Griechischen und bedeutet »Satzbau«. Unter der Syntax einer Programmiersprache ist die vollständige Beschreibung erlaubter und verbotener Konstruktionen zu verstehen. Die Syntax wird durch eine Grammatik festgelegt, an die sich der Programmierer zu halten hat. Tut er es nicht, so verursacht er den allseits bekannten **Syntax Error**.

3.1.1 Zeilenlänge

Prinzipiell können Quellcodezeilen beliebig lang werden. Viele Programmierer beschränken die Länge ihrer Quellcodezeilen jedoch, damit beispielsweise mehrere Quellcodedateien nebeneinander auf den Bildschirm passen oder der Code auch auf Geräten mit einer festen Zeilenbreite angenehm zu lesen ist. Eine geläufige **maximale Zeilenlänge sind 80 Zeichen**. Innerhalb von Klammern darf man Quellcode zwar beliebig umbrechen, doch an vielen anderen Stellen ist man an die strengen syntaktischen Regeln von Python gebunden, aber mit Hilfe der **Backslash-Notation** ist es möglich, Quellcode an nahezu beliebigen Stellen in eine neue Zeile umzubrechen z.B.:

Programm mit Backslash-Notation:

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

x \
= \
12

print(x)
```

Liefert folgenden Output:

```
Hello World!
12
```

3.1.2 Anweisungen

Grundsätzlich besteht ein Python-Programm aus einzelnen *Anweisungen*, die im einfachsten Fall genau eine Zeile im Quelltext einnehmen z.B.:

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

Es ist auch möglich mehrere Anweisungen in eine Zeile zu schreiben und sie mit `;` zu trennen, nur verschlechtert dies meist die Lesbarkeit des Codes und sollte nicht zu häufig verwendet werden.

```
x = 7; y = 12; z = 42.42;
```

3.1.3 Anweisungsblöcke

Einige Anweisungen lassen sich in einen *Anweisungskopf* und einen *Anweisungskörper* unterteilen, wobei der Körper weitere Anweisungen enthalten kann z.B.:

```
if (y < 10):
    print("x ist")
    print("groesser als")
    print("10")
```

Die Zugehörigkeit des Körpers zum Kopf wird in Python durch einen **Doppelpunkt** am Ende des Anweisungskopfs und durch eine tiefere Einrückung des Anweisungskörpers festgelegt. Die **Einrückungstiefe** ist üblicherweise jeweils vier Leerzeichen.
ACHTUNG: bei der Verwendung von TAB! Tabulatorsprünge könne je nach verwendetem Editor unterschiedlich eingestellt sein - bitte im Editor konfigurieren!

3.1.4 Kommentare

Ein Kommentar ist ein kleiner Text, der eine bestimmte Stelle des Quellcodes erläutert und auf Probleme, offene Aufgaben oder Ähnliches hinweist. Ein Kommentar wird vom Interpreter einfach ignoriert, ändert also am Ablauf des Programms nichts. Die einfachste Möglichkeit, einen Kommentar zu verfassen, ist der **Zeilenkommentar**. Diese Art des Kommentars wird mit dem #-Zeichen begonnen und endet mit dem Ende der Zeile.
Die zweite Möglichkeit Kommentare zu verfassen sind **Blockkommentare**. Sie sind durch drei einfache oder doppelte Anführungszeichen begrenzt und werden häufig als Programm-Header (Info zum Programm) verwendet:

```
'''
Ein Blockkommentar am Anfang - meist mit Info zum Inhalt des
Programms, sowie Datum und Autor.
17.10.20
Markus
'''
#Zeilen Kommentar z.B. nun erstelle ich eine Variable
x = 12  #und weise ihr den Wert 12 zu
```

3.1.5 Fehlermeldungen

Fehler passieren jedem, nur wenn er sich in hunderten Code-Zeilen versteckt, kann es langwierig werden ihn zu finden. Python versucht bei der Fehlersuche zu helfen, in dem es die Zeile und die Zeilennummer anzeigt in welcher der Fehler auftrat. Weiters meldet es den Fehlertypen.
Typisch für eine Interpretersprache meldet es den ersten Fehler den es findet, d.h. es können nach dem dieser behoben wurde noch weitere Fehler auftauchen.
Im folgenden ein Syntaxfehler weil ein Zuweisung zu einer Variable nicht gemacht wurde:

Syntaxfehler im Code:

```
'''
Ein fehlerhaftes Programm
mit einem Syntaxfehler
Markus
Okt.2020
'''

x = 8
y =
z = x + y
print("z=",z)
```

Fehlermeldung:

```
File "02.py", line 9
    y =
      ^
SyntaxError: invalid syntax
```

Oder sehr häufig bei Python-Anfängern -> ein Einrückungsfehler (IndentationError) :)

Einrückungsfehler im Code:

```
'''
Ein fehlerhaftes Programm
mit einem Einrückungsfehler
Markus
Okt.2020
'''

x = 8
if (x < 10):
print("haette einruecken sollen ... ")
```

Fehlermeldung:

```
 File "02.py", line 10
    print("haette einruecken sollen ... ")
    ^
IndentationError: expected an indented block
```

3.1.6 Variablen Namen

Es ist *best practice* Variablen mit kleinen Buchstaben zu beginne und bei verknüpften Namen einen _ zu verwenden:

```
x = 12
die_Antwort_auf_alles = 42
my_list = [12,34,56,99]
```

Weiters sollten Kommentare und Namen möglichst in Englisch geschrieben werden und wenn in Deutsch dann ohne Sonderzeichen wie Umlaute oder ß.

3.1.7 Einfache Ein- Ausgabe mit print() und input()

Die `print()` und `input()` Funktionen werden in den folgenden kleinen Beispielprogrammen so häufig verwendet, dass sie vorab kurz erklärt werden sollten.

3.1.7.1 print()

Die `print()` Funktion erlaubt es Daten auf den Bildschirm (Standard Output - stdout) auszugeben.

Die print() Funktion:

```
'''
Ein paar Beispiele der print() Funktion
Markus
Okt.2020
'''
#Einen String ausgeben
print("Hallo World!")

#Eine Variable an einen String anhaengen
x = 8.123
print("x=",x)

#Einen Wert ausgeben
print(42/2)
```

Bildschirmausgabe:

```
Hallo World!
x= 8.123
21.0
```

Auffallend ist vielleicht, das die `print()` Funktion einen automatischen Zeilenumbruch erzeugt. Möchte man auf diesen Zeilenumbruch verzichten, kann man den Parameter `end=""` übergeben z.B.:

Die print() Funktion:

```
'''
Ein paar Beispiele der print() Funktion
Markus
Okt.2020
'''
#Einen String ausgeben ohne Zeilenumbruch
print("Hallo ",end="")

#noch einen String ohne Zeilenumbruch
print("World ",end="")

#nun mit Zeilenumbruch
print("!")
```

Bildschirmausgabe:

```
Hallo World !
```

Für mehr Informationen zum Thema formatierte-Ausgabe siehe Paragraph 4.3.3.3 auf Seite 13.

3.1.7.2 input()

Die Funktion `input()` liest eine Eingabe vom Benutzer ein und gibt sie in Form eines *Strings* zurück. Als optionaler kann ein String übergeben werden, der vor der Eingabeaufforderung ausgegeben werden soll.

Die input() Funktion:

```
'''
Die input() Funktion
Markus
Okt.2020
'''
#Eine Eingabe ohne String
print("Bitte einen String eingeben:")
s = input()

#Eingabe mit String
s2 = input("Bitte eine Zahl eingeben ->")

print("s =",s," und s2 =",s2)
```

Bildschirmausgabe:

```
Bitte einen String eingeben:
Ich habe Hunger ...
Bitte eine Zahl eingeben ->42
s = Ich habe Hunger ...  und s2 = 42
```

Bitte beachten, dass die `input()` Funktion immer einen String liefert, auch wenn eine Zahl eingegeben wurde! Mit den eingebauten Funktionen `int()` und `float()` kann man den Input-String in eine Zahl umwandeln.

Die input() Funktion mit Umwandlung in Zahlen:

```
'''
Die input() Funktion
mit Umwandlung in Zahlen
Markus
Okt.2020
'''
z1 = input("Bitte 1. Zahl eingeben ->")
z2 = input("Bitte 2. Zahl eingeben ->")
print("Ohne Umwandlung:")
print("z1 + z2 =",z1+z2)

print("mit int-Umwandlung:")
print("z1 + z2 =",int(z1)+int(z2))

print("mit float-Umwandlung:")
print("z1 + z2 =",float(z1)+float(z2))
```

Bildschirmausgabe:

```
Bitte 1. Zahl eingeben ->12
Bitte 2. Zahl eingeben ->30
Ohne Umwandlung:
z1 + z2 = 1230
mit int-Umwandlung:
z1 + z2 = 42
mit float-Umwandlung:
z1 + z2 = 42.0
```

4 Variablen, Datentypen und dynamische Typisierung

4.1 Dynamische Typisierung

Wenn man in Python eine Variable deklariert, muss man im Vergleich zu vielen anderen Programmiersprachen **NICHT** angeben welchem Datentyp die Variable repräsentiert. Außerdem kann eine Variable ihren Datentyp dynamisch Wechseln. Mit der build-in Funktion `type()` kann man abfragen zu welcher Klasse eine Variable gehört z.B.:

Beispiel Code:

```
a = 3
print("a =",a,"and of type=",type(a))
a = a/2
print("a =",a,"and of type=",type(a))
a = "hello"
print(" a=",a,"and of type=",type(a))
```

Zugehörige Ausgabe:

```
a = 3 and of type= <class 'int'>
a = 1.5 and of type= <class 'float'>
a = hello and of type= <class 'str'>
```

4.2 Numerische Datentypen

Es gibt vier numerische Datentypen:

Datentyp	Beschreibung	Veränderlichkeit
`int`	integer - ganze Zahlen	unveränderlich
`float`	floating point number - Gleitkommazahl	unveränderlich
`bool`	boolean - boolsche Werte	unveränderlich
`complex`	complex numbers - komplexe Zahlen	unveränderlich

Tabelle 1: Numerische Datentypen

Alle numerischen Datentypen sind **unveränderlich** oder **immutable**. Das bedeutet nicht, dass es keine Operatoren gibt, um Zahlen zu verändern, sondern vielmehr, dass nach jeder Veränderung eine neue Instanz des jeweiligen Datentyps erzeugt werden muss. Das scheint jetzt nicht besonders wichtig zu sein, wird später aber noch einmal wichtig wenn wir Daten an Funktionen übergeben!

Die numerischen Datentypen bilden eine Gruppe, weil sie thematisch zusammengehören. Diese Zusammengehörigkeit zeigt sich auch darin, dass die numerischen Datentypen viele gemeinsame Operatoren haben:

4.2.1 Arithmetische Operatoren

Arithmetische Operatoren:

Operator	Ergebnis
x + y	Summe von x und y
x - y	Differenz von x und y
x * y	Produkt von x und y
x / y	Quotient von x und y
x % y	Modulus - Rest beim Teilen von x durch y
+x	positives Vorzeichen
-x	negatives Vorzeichen
x ** y	x hoch y
x // y	abgerundeter Quotient von x und y

Tabelle 2: Arithmetische Operatoren

Erweiterte Zuweisungen:

Operator	Entsprechung
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x **= y	x = x ** y
x //= y	x = x // y

Tabelle 3: Erweiterte Zuweisungen

4.2.2 Vergleichende Operatoren

Ein vergleichender Operator ist ein Operator, der aus zwei Instanzen einen Wahrheitswert berechnet.
Jeder dieser vergleichenden Operatoren liefert als Ergebnis einen Wahrheitswert. Ein solcher Wert wird zum Beispiel als Bedingung einer `if`-Anweisung erwartet:

```
if x < 4:
    print("x ist kleiner als 4")
```

Operator	Ergebnis
==	wahr, wenn x und y gleich sind
!=	wahr, wenn x und y verschieden sind
<	wahr, wenn x kleiner ist als y
<=	wahr, wenn x kleiner oder gleich y ist
>	wahr, wenn x größer ist als y
>=	wahr, wenn x größer oder gleich y ist

Tabelle 4: Vergleichende Operatoren

Man kann beliebig viele der vergleichenden Operatoren zu einer Reihe verketten:

```
if 2 < x < 4:
    print("x liegt zwischen 2 und 4, ist also 3 ;-)")
```

4.2.3 Integer - int

Für den Raum der ganzen Zahlen gibt es in Python den Datentypen `int`. Im Gegensatz zu vielen anderen Programmiersprachen unterliegt dieser Datentyp in seinem Wertebereich keinen prinzipiellen Grenzen, was den Umgang mit großen ganzen Zahlen in Python sehr komfortabel macht.
Deklaration und Verwendung sind sehr einfach:

```
a_int_variable = 123
```

wobei man bei _ verwenden kann um die Lesbarkeit von großen Zahlen zu erhöhen:

```
a_lage_number = 1_000_000
```

oder auch die Potenzschreibweise verwenden:

```
# x soll 12*10^9 sein (also 12 Milliarden)
x = 12e9
```

4.2.3.1 Zahlensysteme

Ganze Zahlen können in Python in mehreren Zahlensystemen dargestellt werden. Zum bis jetzt verwendeten Dezimalsystem wird in der Technik oft noch das **Binär-** und das **Hexadezimalsystem** verwendet.

Beispiel Code:

```
#Dezimalsystem
a = 42
print("a =",a)

#Binärsystem
b = 0b101010
print("b =",b)

#Hexadezimalsystem
c = 0x2a
print("c =",c)
```

Zugehörige Ausgabe:

```
a = 42
b = 42
c = 42
```

Bitte beachten das es sich bei den Zahlensystemen nur um eine alternative Schreibweise des gleichen Wertes handelt. Der Datentyp `int` springt nicht in eine Art Hexadezimalmodus, sobald er einen solchen Wert enthält, sondern das Zahlensystem ist nur bei Zuweisungen oder Ausgaben von Bedeutung. Wie man an der Ausgabe erkennt werden standardmäßig alle Zahlen im Dezimalsystem ausgegeben.

4.2.3.2 Bit-Operationen

Da die Binärdarstellung in der Informatik relativ wichtig ist, sind für den Datentyp `int` daher einige zusätzliche Operatoren definiert, die auf Bit-Ebene arbeiten:

Operator	Erw. Zuweisung	Ergebnis
`x & y`	`x &= y`	bitweises UND von x und y (AND)
`x | y`	`x |= y`	bitweises nicht ausschließendes ODER von x und y (OR)
`x ^ y`	`x ^= y`	bitweises ausschließendes ODER von x und y (XOR)
`~x`		bitweises Komplement von x
`x << n`	`x <<= n`	Bit-Verschiebung um n Stellen nach links
`x >> n`	`x >>= n`	Bit-Verschiebung um n Stellen nach rechts

Tabelle 5: Bit-Operationen

4.2.4 Fließkommazahlen - float

Zum Speichern einer Gleitkommazahl mit begrenzter Genauigkeit wird der Datentyp `float` verwendet. Es gelten die gleichen Hilfen wie für `int` also sind folgende Schreibweisen erlaubt:

```
x = 3.1415
y = 1.23e-3
z = 42.000_000_001
```

Aufgrund der Begrenztheit von `float` können reelle Zahlen nicht unendlich präzise gespeichert werden. Stattdessen werden sie mit einer bestimmten Genauigkeit angenähert.
ACHTUNG: Wenn man genaue Rechnungen vornehmen muss kann man auf das Modul `decimal` aus der Standardbibliothek zurückgreifen.
Außerdem kann der Datentyp `float` nicht beliebig groß werden. Wenn das Limit überschritten ist, wird die Zahl als `inf` (infinity) gespeichert. Es kommt also zu keinem Fehler, und es ist immer noch möglich, eine übergroße Zahl mit anderen zu vergleichen oder manche (+ und *) Rechnungen durchzuführen. Subtraktion oder Division zweier unendlich großer Zahlen sind nicht möglich und es wird das Ergebnis `nan` (not a number) zurückgegeben.

4.2.5 Boolesche Werte - bool

Der Datentyp `bool` kann nur zwei Werte annehmen: `True` oder `False` diese Werden aber intern als 1 und 0 dargestellt - daher die Zuordnung zu Numerischen Datentypen. Es gibt drei logische Operatoren: Welche beliebig kombiniert

Operator	Ergebnis
`not x`	logische Negierung von x
`x and y`	logisches UND zwischen x und y
`x or y`	logisches (nicht ausschließendes) ODER zwischen x und y

Tabelle 6: Boolesche Operatoren

werden können. Natürlich auch mit den vergleichenden numerischen Operatoren!

```
x = True; y = False;
if (x or y):
        print("Entweder x oder y hat den Wert 'True'")


x = 4; y = 3; z = 7;
if (x > y or (y < z and x != 0)):
        print("Keine Ahnung was das bedeutet!")
```

4.2.5.1 Wahrheitswerte nicht-boolescher Datentypen

Mithilfe der Funktion `bool` lassen sich Instanzen eines jeden Basisdatentyps in einen booleschen Wert überführen. Bei sequenziellen Datentypen (werden später besprochen) entspricht der Status *leer* dem Wert `False` und bei den numerischen Datentypen der Wert null. Ein paar Beispiele aus dem Interpreter:

```
>>> bool(7)
True
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool("hello")
True
>>> bool("")
False
```

4.2.6 Komplexe Zahlen - compolex

Der Datentyp `complex` hat zwei Attribute: einen Realteil und einen Imaginärteil. Außerdem wird noch die Methode `.conjugate()` zur Verfügung gestellt, welche die konjugiert komplexe Zahl liefert:

```
>>> a = 7 + 3j
>>> a.imag
3.0
>>> a.real
7.0
>>> a.conjugate()
(7-3j)
```

4.3 Sequenzielle Datentypen

Unter sequenziellen Datentypen wird eine Klasse von Datentypen zusammengefasst, die Folgen von gleichartigen oder verschiedenen *Elementen* verwalten. Die in sequenziellen Datentypen gespeicherten Elemente haben eine definierte Reihenfolge, und man kann über eindeutige *Indizes* auf sie zugreifen. Die fünf sequenziellen Datentypen sind in Tabelle 7 aufgelistet :

Datentyp	**speichert**	**Veränderlichkeit**
`list`	Listen beliebiger Instanzen	veränderlich
`tuple`	Listen beliebiger Instanzen	unveränderlich
`str`	Text als Sequenz von Buchstaben	unveränderlich
`bytes`	Binärdaten als Sequenz von Bytes	unveränderlich
`bytearray`	Binärdaten als Sequenz von Bytes	veränderlich

Tabelle 7: Sequenzielle Datentypen

Der Datentyp `str` ist für die Speicherung und Verarbeitung von Texten vorgesehen. Daher besteht eine `str`-Instanz aus einer Folge von Buchstaben, Leer- und Interpunktionszeichen sowie Zeilenvorschüben – also genau den Bausteinen, aus denen Texte in menschlicher Sprache bestehen. Dies funktioniert auch mit regionalen Sonderzeichen wie beispielsweise den deutschen Umlauten.
Im Gegensatz dazu kann eine Instanz des Datentyps `bytes` einen binären Datenstrom, also eine Folge von Bytes, speichern. Der Datentyp `bytearray` ist ebenfalls in der Lage, Binärdaten zu speichern. Allerdings sind `bytearray`-Instanzen anders als `bytes`-Instanzen veränderlich.

4.3.0.1 Operationen auf Instanzen sequenzieller Datentypen

Für alle sequenziellen Datentypen sind folgende Operationen definiert (s und t sind dabei Instanzen desselben sequenziellen Datentyps; i, j, k und n sind ganze Zahlen; x ist eine Referenz auf eine beliebige Instanz). Die Auflistung der Operationen ist in Tabelle 8 dargestellt :

Notation	Beschreibung
`x in s`	Prüft, ob x in s enthalten ist. Das Ergebnis ist ein Wahrheitswert.
`x not in s`	Prüft, ob x nicht in s enthalten ist. Das Ergebnis ist True/False.
`s + t`	Das Ergebnis ist eine neue Sequenz, die die Verkettung von s und t enthält.
`s += t`	Erzeugt die Verkettung von s und t und weist sie s zu.
`s * n oder n * s`	Liefert eine neue Sequenz, die die Verkettung von n Kopien von s enthält.
`s *= n`	Erzeugt das Produkt s * n und weist es s zu.
`s[i]`	Liefert das i-te Element von s.
`s[i:j]`	Liefert den Ausschnitt aus s von i bis j.
`s[i:j:k]`	Liefert den Ausschnitt aus s von i bis j, wobei nur jedes k-te Element beachtet wird.
`len(s)`	Gibt die Anzahl der Elemente von s zurück.
`max(s)`	Liefert das größte Element von s, sofern es eine Ordnungsrelation gibt.
`min(s)`	Liefert das kleinste Element von s, sofern es eine Ordnungsrelation gibt.
`s.index(x[, i[, j]])`	Gibt den Index k des ersten Vorkommens von x in der Sequenz s im Bereich i <= k < j zurück.
`s.count(x)`	Zählt, wie oft x in der Sequenz s vorkommt.

Tabelle 8: Operationen für sequenzielle Datentypen

4.3.1 Listen - list

Listen können beliebige Instanzen unterschiedlicher Datentypen aufnehmen. Eine Liste kann also durchaus Zahlen, Strings oder auch weitere Listen als Elemente enthalten. Eine neue Liste wird erzeugt, indem man eine Aufzählung (durch Beistrich getrennt) ihrer Elemente in eckige Klammern `[]` schreibt:

```
#leere Liste anlegen
my_list = []

#gefuellte Liste anlegen
my_second_list = [3, 4.09, "hello", True, 7]
```

Eine Liste kann auch mit einer *List Comprehension* erzeugt werden. Dabei werden nicht alle Elemente der Liste explizit aufgelistet, sondern über eine Bildungsvorschrift ähnlich einer *for-Schleife* erzeugt. Die folgende *List Comprehension* erzeugt beispielsweise eine Liste mit den Quadraten der Zahlen von 0 bis 9.

```
>>> [i*i for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Im Gegensatz zu den bisher besprochenen sequenziellen Datentypen kann sich der Inhalt einer Liste auch nach ihrer Erzeugung ändern, weshalb eine Reihe weiterer Operatoren und Methoden - siehe Tabelle 9 (zusätzlich zu Tabelle 8 auf Seite 10) für sie verfügbar ist:

Operator	Wirkung
s[i] = x	Das Element von s mit dem Index i wird durch x ersetzt.
s[i:j] = t	Der Teil s[i:j] wird durch t ersetzt. Dabei muss t iterierbar sein.
s[i:j:k] = t	Die Elemente von s[i:j:k] werden durch die von t ersetzt.
del s[i]	Das i-te Element von s wird entfernt.
del s[i:j]	Der Teil s[i:j] wird aus s entfernt. Das ist äquivalent zu s[i:j] = [].
del s[i:j:k]	Die Elemente der Teilfolge s[i:j:k] werden aus s entfernt.

Tabelle 9: Operatoren für Listen

4.3.1.1 Methoden für Listen

Zur Erinnerung: Der Unterschied zwischen einer *Funktion* und einer *Methode* ist, dass eine *Methode* immer nur auf eine Instanz mittel Punkt-Notation aufgerufen wird.

Methode	**Wirkung**
s.append(x)	Hängt x ans Ende der Liste s an.
s.extend(t)	Hängt alle Elemente der Liste t ans Ende der Liste s an.
s.insert(i, x)	Fügt x an der Stelle i in die Liste s ein. Die folgenden Elemente rücken nach hinten.
s.pop([i])	Gibt das i-te Element der Liste s zurück und entfernt es aus s.
s.remove(x)	Entfernt das erste Vorkommen von x aus der Liste s.
s.reverse()	Kehrt die Reihenfolge der Elemente in s um.
s.sort([key, reverse])	Sortiert die Liste s mittels key-words

Tabelle 10: Methoden von List-Instanzen

Noch zwei kurze Beispiele zur besseren Verständlichkeit der `.sort()` Methode:

```
>>> l = ["Katharina", "Peter", "Jan", "Florian", "Paula", "Ben"]
>>> l.sort(key=len)
>>> l
['Jan', 'Ben', 'Peter', 'Paula', 'Florian', 'Katharina']

>>> l = [4, 2, 7, 3, 6, 1, 9, 5, 8]
>>> l.sort(reverse=True)
>>> l
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

4.3.1.2 Besonderheiten von mutablen Datentypen

Im Zusammenhang mit Pythons `list`-Datentyp ergeben sich ein paar Besonderheiten, die nicht unmittelbar ersichtlich sind, und die im Prinzip für alle mutablen Datentypen gelten. Am besten vergleichen wir `str` (immutable) mit `list` (mutable):

String-(immutable)-Verhalten:

```
>>> a = "Hello"
>>> b = a
>>> b += " Besi"
>>> b
'Hello Besi'
>>> a
'Hello'
>>> a is b
False
```

List-(mutable)-Verhalten:

```
>>> a = [1,2,3]
>>> b = a
>>> b += [4,5,6]
>>> b
[1, 2, 3, 4, 5, 6]
>>> a
[1, 2, 3, 4, 5, 6]
>>> a is b
True
```

Beim immutable String bewirkt die Zeile `b = a`, dass eine neue Instanz vom Typ String erzeugt wird und ihr die gleiche Zeichenabfolge zugewiesen wird wie a.
Bei der mutable List bewirkt die Zeile `b = a`, dass eine Referenz-Variable b auf die gleiche Listen-Instanz verweist! Das bedeutet, egal ob ich die Liste über die Referenz `a` oder `b` manipuliere, es wird immer der gleiche Datensatz bearbeitet! Wenn man eine wirkliche Kopie einer Listen-Instanz anlegen möchte, muss man den Inhalt der Liste kopieren: `b = a[:]`

4.3.2 Tuple - unveränderliche Listen

Der Datentyp `list` ist sehr flexibel und wird häufig verwendet. Seine Mächtigkeit und Flexibilität haben aber auch den Nachteil, dass die Verwaltung einer Liste intern relativ ressourcenaufwendig ist. Oft wird gar nicht die Flexibilität einer Liste benötigt, sondern nur ihre Fähigkeit, Referenzen auf beliebige Instanzen zu speichern. Deshalb existiert in Python neben `list` der Datentyp `tuple`, der im Gegensatz zu `list` immutabel ist.
Zum Erzeugen neuer tuple-Instanzen dienen die runden Klammern, die (wie bei den Listen) durch Kommata getrennt, die Elemente des Tupels enthalten:

```
>>> a = (1, 2, 3, 4, 5)
>>> a[3]
4
```

Es gelten die gleichen Operatoren wie in Tabelle 8 auf Seite 10.

4.3.3 Strings - str, bytes, bytearray

Die Zeichen, die eine Instanz des Datentyps `str` speichern kann, sind Buchstaben, Satz- und Leerzeichen oder auch Umlaute. Im Gegensatz dazu sind die Datentypen `bytes` und `bytearray` für die Speicherung von Binärdaten vorgesehen. Daher bestehen Instanzen der Datentypen `bytes` und `bytearray` aus einer Folge einzelner Bytes, also ganzen Zahlen von 0 bis 255.
Wir werden uns bis auf Weiteres nur mit `str`-Instanzen beschäftigen, da sich der Umgang mit `str` nicht wesentlich von dem mit `bytes` unterscheidet.
Um neue `str`-Instanzen zu erzeugen, gibt es folgende Möglichkeiten:

```
>>> string1 = "Ich wurde mit doppelten Hochkommata definiert"
>>> string2 = 'Ich wurde mit einfachen Hochkommata definiert'
>> string3 = """Erste Zeile!
... Ui, noch eine Zeile"""
>>> a = ("Stellen Sie sich einen schrecklich "
...      "komplizierten String vor, den man "
...      "auf keinen Fall in eine Zeile schreiben "
...      "kann.")
>>> a
'Stellen Sie sich einen schrecklich komplizierten String vor, den man auf
keinen Fall in eine Zeile schreiben kann.'
```

4.3.3.1 Steuerzeichen

Es gibt besondere Textelemente, die den Textfluss steuern und sich auf dem Bildschirm nicht als einzelne Zeichen darstellen lassen. Zu diesen sogenannten Steuerzeichen zählen unter anderem der Zeilenvorschub, der Tabulator oder der Rückschritt (von engl. backspace). Die Darstellung solcher Zeichen innerhalb von String-Literalen erfolgt mittels spezieller Zeichenfolgen, den *Escape-Sequenzen*. *Escape-Sequenzen* werden von einem Backslash \ eingeleitet, dem die Kennung des gewünschten Sonderzeichens folgt. Die Escape-Sequenz `\n` steht beispielsweise für einen Zeilenumbruch:

Escape-Sequenz	Bedeutung
`\a`	Bell (BEL) erzeugt einen Signalton.
`\b`	Backspace (BS) setzt die Ausgabeposition um ein Zeichen zurück.
`\f`	Formfeed (FF) erzeugt einen Seitenvorschub.
`\n`	Linefeed (LF) setzt die Ausgabeposition in die nächste Zeile.
`\r`	Carriage Return (CR) setzt die Ausgabeposition an den Anfang der nächsten Zeile.
`\t`	Horizontal Tab (TAB) hat die gleiche Bedeutung wie die Tabulatortaste.
`\v`	Vertikaler Tabulator (VT); dient zur vertikalen Einrückung.
`\"`	doppeltes Hochkomma
`\'`	einfaches Hochkomma
`\\`	Backslash, der wirklich als solcher in dem String erscheinen soll

Tabelle 11: Escape-Sequenzen

Steuerzeichen stammen aus der Zeit, als die Ausgaben hauptsächlich über Drucker erfolgten. Deshalb haben einige dieser Zeichen heute nur noch eine geringe praktische Bedeutung.

4.3.3.2 String-Methoden

Es gibt eine Vielzahl von String-Methoden deshalb wird in Tabelle 12 auf Seite 13 nur eine kleine Auswahl vorgestellt. Für die vollständige Auflistung wird auf die Python-Docs verwiesen:
`https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str`

Methode	Beschreibung
`s.split([sep, maxsplit])`	Teilt s bei Vorkommen von sep. Die Suche beginnt am String-Anfang.
`s.find(sub, [start, end])`	Sucht den String sub im String s. Die Suche beginnt am String-Anfang.
`s.count(sub, [start, end])`	Zählt die Vorkommen von sub in s.
`s.replace(old, new, [count])`	Ersetzt die Vorkommen von old im String s durch new.
`s.lower()`	Ersetzt alle Großbuchstaben in s durch entsprechende Kleinbuchstaben.
`s.upper()`	Ersetzt alle Kleinbuchstaben in s durch entsprechende Großbuchstaben.
`s.strip([chars])`	Entfernt bestimmte Zeichen am Anfang und am Ende des Strings s.

Tabelle 12: Eine kleine Auswahl von String-Methoden

4.3.3.3 Formatierung von Strings

Oft möchte man seine Bildschirmausgaben auf bestimmte Weise anpassen. Um beispielsweise eine dreispaltige Tabelle von Zahlen anzuzeigen, müssen – abhängig von der Länge der Zahlen – Leerzeichen eingefügt werden, damit die einzelnen Spalten untereinander angezeigt werden. Eine Anpassung der Ausgabe ist auch nötig, wenn man einen Geldbetrag ausgeben möchten, der in einer float-Instanz gespeichert ist, die mehr als zwei Nachkommastellen besitzt. Mithilfe von `format` kann man in einem String Platzhalter durch bestimmte Werte ersetzen lassen. Diese Platzhalter sind durch geschweifte Klammern eingefasst und können sowohl Zahlen als auch Zeichenketten sein.
Im folgenden Beispiel lassen wir die Platzhalter 0 und 1 durch zwei Zahlen ersetzen:

```
>>> "Es ist {0}:{1} Uhr".format(13, 37)
'Es ist 13:37 Uhr'
```

Es könne natürlich auch Variablen übergeben werden und auch die Platzhalter müssen nicht nummeriert werden. Wenn sie ohne Nummerierung verwendet werden, fühlt Python einfach der Reihenfolge der format-Parameter folgend ein z.B.:

```
>>> h = 13
>>> m = 37
>>> "Es ist {}:{} Uhr".format(h,m)
'Es ist 13:37 Uhr'
```

Bisher haben wir mithilfe von `format` nur Platzhalter durch bestimmte Werte ersetzt, ohne dabei festzulegen, nach welchen Regeln die Ersetzung vorgenommen wird. Um dies zu erreichen, können *Formatangaben* (engl. format specifier) durch einen *Doppelpunkt* getrennt vom Namen des Platzhalters angegeben werden. Um beispielsweise eine Gleitkommazahl auf zwei Nachkommastellen gerundet auszugeben, benutzt man die Formatangabe `.2f`:

```
>>> "Betrag: {:.2f} Euro".format(13.37690)
'Betrag: 13.38 Euro'
```

Wird als Formatangabe eine einfache Ganzzahl verwendet, legt sie die minimale Breite fest, die der ersetzte Wert einnehmen soll. Möchte man beispielsweise eine Tabelle mit Namen ausgeben und sicherstellen, dass alles bündig untereinander steht, erreicht man dies folgendermaßen:

```
>>> f = "{:15}|{:15}"
>>> print(f.format("Vorname", "Nachname")); print(f.format("Markus", "Signitzer"))
Vorname        |Nachname
Markus         |Signitzer
```

Wenn man die minimale Breite eines Feldes angibt, kann die Ausrichtung des Wertes bestimmt werden, falls er – wie es im ersten der beiden oben genannten Beispiele der Fall war – nicht die gesamte Breite ausfüllt. Insgesamt gibt es vier Ausrichtungsarten: `,<,>,=,^`

```
>>> "Endpreis: {:^15} Euro".format(321)
'Endpreis:       321       Euro'
>>> "Endpreis: {:<15} Euro".format(-321)
'Endpreis: -321            Euro'
>>> "Endpreis: {:>15} Euro".format(321)
'Endpreis:             321 Euro'
>>> "Endpreis: {:=15} Euro".format(-321)
'Endpreis: -           321 Euro'
```

Um bei Zahlenwerten die Ausgabe weiter anpassen zu können, gibt es verschiedene Ausgabetypen, die ganz am Ende der Formatangabe eingefügt werden. Beispielsweise werden mit der Typangabe `b` Ganzzahlen in Binärschreibweise ausgegeben und mit `x` in Hexadezimaldarstellung ausgegeben:

```
>>> "Die Zahl {} in Binär: {:b}".format(42,42)
'Die Zahl 42 in Binär: 101010'
>>> "Die Zahl {} in Hex: {:x}".format(42,42)
'Die Zahl 42 in Hex: 2a'
```

4.4 Zuordnungen - Mappings

Die Kategorie Mappings (dt. »Zuordnungen«) enthält Datentypen, die eine Zuordnung zwischen verschiedenen Objekten herstellen. Der einzige Datentyp, der in diese Kategorie fällt, ist das *Dictionary*.

4.4.1 Dictionary - dict

er Name des Datentyps `dict` gibt bereits einen guten Hinweis darauf, was sich dahinter verbirgt: Ein Dictionary enthält beliebig viele Schlüssel-Wert-Paare (engl. key/value pairs), wobei der Schlüssel nicht unbedingt wie bei einer Liste eine ganze Zahl sein muss. Der Datentyp `dict` ist mutabel, also veränderlich.

```
>>> my_dict = {
... "Firma" : "Besi",
... "Antwort auf alles" : 42,
... 7 : "eine Primzahl"
... }
>>> my_dict[7]
'eine Primzahl'
>>> my_dict["Antwort auf alles"]
42
```

4.4.1.1 Methoden für Dictionarys

Methode	Beschreibung
`d.clear()`	Leert das Dictionary d.
`d.copy()`	Erzeugt eine Kopie von d.
`d.get(k, [x])`	Liefert d[k], wenn der Schlüssel k vorhanden ist, ansonsten x.
`d.items()`	Gibt ein iterierbares Objekt zurück, das alle Schlüssel-Wert-Paare von d durchläuft.
`d.keys()`	Gibt ein iterierbares Objekt zurück, das alle Schlüssel von d durchläuft.
`d.pop(k)`	Gibt den zum Schlüssel k gehörigen Wert zurück und löscht das Schlüssel-Wert-Paar aus d.
`d.popitem()`	Gibt ein willkürliches Schlüssel-Wert-Paar von d zurück und entfernt es aus dem Dictionary.
`d.setdefault(k, [x])`	Das Gegenteil von get. Setzt d[k] = x, wenn der Schlüssel k nicht vorhanden ist.
`d.update(d2)`	Fügt ein Dictionary d2 zu d hinzu und überschreibt die Werte bereits vorhandener Schlüssel.
`d.values()`	Gibt ein iterierbares Objekt zurück, das alle Werte von d durchläuft.

Tabelle 13: Dictionary-Methoden

5 Kontrollstrukturen

Unter einer Kontrollstruktur versteht man ein Konstrukt zur Steuerung des Programmablaufs. Dabei unterscheidet man in Python zwei Arten von Kontrollstrukturen: *Schleifen* und Fallunterscheidungen (oder *Verzweigungen* genannt). Schleifen dienen dazu, einen Code-Block mehrmals auszuführen. Fallunterscheidungen hingegen knüpfen einen Code-Block an eine Bedingung. Python kennt jeweils zwei Unterarten von Schleifen und Fallunterscheidungen, die wir im Folgenden behandeln werden.
Achtung auf die Syntax: Kontrollstrukturen enden mit einem : und der folgende Code-Block **muss eingerückt sein!**

5.1 Verzweigungen: if - elif - else

Die einfachste Möglichkeit der Fallunterscheidung ist die `if`-Anweisung. Eine `if`-Anweisung besteht aus einem Anweisungskopf, der eine Bedingung enthält, und aus einem Code-Block als Anweisungskörper.
Der Code-Block wird nur ausgeführt, wenn sich die Bedingung als wahr herausstellt. Die Bedingung einer `if`-Anweisung muss dabei ein Ausdruck sein, der als Wahrheitswert (`True` oder `False`) interpretiert werden kann.

```
if x == 1:
    print("x hat den Wert 1")

if x < 1 or x > 5:
    print("x ist kleiner als 1 ...")
    print("... oder groesser als 5")
```

In vielen Fällen ist es mit einer einzelnen `if`-Anweisung nicht getan, und man benötigt eine ganze Kette von Fallunterscheidungen. So möchten wir im nächsten Beispiel zwei unterschiedliche Strings ausgeben, je nachdem, ob x < 0 oder x = 0 gilt. Dazu können zwei aufeinanderfolgende `if`-Anweisungen verwendet werden:

```
if x < 0:
    print("x ist kleiner Null")
if x == 0:
    print("x ist gleich Null")
```

Dies ist aus Sicht des Interpreters aber **ineffizient** denn die zweite Bedingung wird auch noch überprüft, selbst wenn die erste schon zutrifft! Die zweite Fallunterscheidung bräuchte jedoch nicht mehr in Betracht gezogen zu werden, wenn die Bedingung der ersten bereits `True` ergeben hat.
Um solche Fälle aus Sicht des Interpreters effizienter und aus Sicht des Programmierers übersichtlicher zu machen, kann eine `if`-Anweisung um einen oder mehrere sogenannte `elif`-Zweige (`elif` ist ein Kürzel für `else if`.) erweitert werden. Die Bedingung eines solchen Zweiges wird nur evaluiert, wenn alle vorangegangenen `if`- bzw. `elif`-Bedingungen `False` ergeben haben.
Das obere Beispiel kann also effizienter geschrieben werden:

```
if x < 0:
    print("x ist kleiner Null")
elif x == 0:
    print("x ist gleich Null")
```

Als letzte Erweiterung der `if`-Anweisung ist es möglich, alle bisher unbehandelten Fälle auf einmal abzufangen. So möchten wir beispielsweise nicht nur einen entsprechenden String ausgeben, wenn x < 0 bzw. x == 0 gilt, sondern zusätzlich in allen anderen Fällen - also wenn x positiv. Dazu kann eine `if`-Anweisung um einen `else`-Zweig erweitert werden. Ist dieser vorhanden, muss er an das Ende der `if`-Anweisung geschrieben werden!

```
if x < 0:
    print("x ist kleiner Null")
elif x == 0:
    print("x ist gleich Null")
else:
    print("x ist groesser Null")
```

5.1.1 Bedingte Ausdrücke - conditional expression

Angenommen wir möchten einer Variable var einen Wert zuweisen, dieser ist aber abhängig vom Wert einer anderen Variable x:

```
if x == 1:
    var = 10
else:
    var = 20
```

Vier Zeilen Code scheint etwas lang dafür - mit bedingten Ausdrücken geht es kürzer, ABER auch zu Lasten der Lesbarkeit des Codes.
Ein bedingter Ausdruck hat den folgenden Aufbau:
`A if Bedingung else B`
also könnten wir die oberen vier Zeilen auf eine verkürzen mit:

```
var = (10 if x == 1 else 20)
```

Man kann natürlich alle möglichen Bedingungen einbauen, nur leidet die Lesbarkeit des Codes schon sehr:

```
xyz = (a * 2 if (a > 10 and b < 5) else b * 2)
```

5.2 Schleifen

Eine Schleife ermöglicht es, einen Code-Block, den Schleifenkörper, mehrmals hintereinander auszuführen. Python unterscheidet zwei Typen von Schleifen: die `while`-Schleife als einfaches Schleifenkonstrukt und die `for`-Schleife zum Durchlaufen komplexerer Datentypen.

5.2.1 Die while-Schleife

Grundsätzlich besteht eine `while`-Schleife aus einem Schleifenkopf, in dem die Bedingung steht, sowie einem **eingerückten** Schleifenkörper, der dem auszuführenden Code-Block entspricht. Die Schleife läuft solange die Bedingung erfüllt ist.

```
x = 10
while (x >= 0):
        #Schleifenkoerper wird solange ausgefuehrt,
    #solange die Bedingung True ist
        print(x)
        x -= 1

print("Fertig!")
```

5.2.1.1 Vorzeitiger Abbruch

Mit dem Schlüsselwort `break` kann die Schleife vorzeitig abgebrochen werden:

```
x = 10
while (x >= 0):
        #Schleifenkoerper wird solange ausgefuehrt,
        #solange die Bedingung True ist
        if (x = 5):
                break
        print(x)
        x = x - 1

print("Fertig!")
```

Code der unter dem `break` Statement folgt wird nicht mehr ausgeführt! Es wird sofort zum Ende der Schleife gesprungen und der Code nach der Schleife wird ausgeführt.
Manchmal möchte man aber zwischen einem frühen Abbruch und einem normalen Schleifendurchlauf unterscheiden können z.B. macht es in unserem Beispiel vielleicht wenig Sinn nach dem frühen Abbruch das *Fertig!* auszugeben.

5.2.1.2 Unterscheidung zwischen Abbruch und Durchlaufen

Man kann einer `while`-Schleife einen `else`-Block anhängen, welcher nur dann ausgeführt wird, wenn die Schleife durchgelaufen ist, also ihre Bedingung das erste Mal `False` ergibt.

```
x = 10
while (x >= 0):
        #Schleifenkoerper wird solange ausgefuehrt,
        #solange die Bedingung True ist
        if (x = 5):
                break
        print(x)
        x = x - 1

else:
        print("Fertig!")
```

5.2.1.3 Abbruch nur eines Schleifendurchlaufs

Die `continue`-Anweisung bricht (im Gegensatz zu `break`) nicht die gesamte Schleife, sondern nur den aktuellen Schleifendurchlauf ab.
Im folgenden Beispiel verwenden wir `continue` zur Absicherung einer Division und `break` zum Beenden einer Endlosschleife.

```
x = 100
print("Bitte eine Zahl eingeben durch die sie x dividieren")
print("möchten oder den String 'end' um das Programm zu beenden.")

while(True):
        user_input = input("Bitte Zahl eingeben ->")
        if (user_input == 'end'):
                print("bye!")
                break
        i = float(user_input)
        if (i == 0):
                print("Division by zero!")
                print("Bitte eine andere Zahl eingeben!")
                continue
        print("Erg = {:.2f}".format(x/i))
```

5.2.2 Die for-Schleife

Eine `for`-Schleife wird verwendet, um ein iterierbares Objekt (z.B. ein String oder eine Liste) zu durchlaufen.
Dazu wird das Schlüsselwort `for` geschrieben, gefolgt von einem *Bezeichner*, dem Schlüsselwort `in` und dem iterierbaren Objekt. Darauf folgt, eine Ebene weiter eingerückt, der Schleifenkörper.
Über den gewählten *Bezeichner* kann im Schleifenkörper auf das jeweils aktuelle Element des iterierbaren Objekts zugegriffen werden.

For-Loops

```
my_list = [1,2,3]
print(my_list)
for x in my_list:
        print(x)
#oder
name = "Besi"
print(name)
for letter in name:
        print(letter)
```

Zugegehöriger Output:

```
[1, 2, 3]
1
2
3
Besi
B
e
s
i
```

Die im Zusammenhang mit der `while`-Schleife besprochenen Schlüsselworte `break` und `continue` zum Abbrechen

einer Schleife bzw. eines Schleifendurchlaufs können auch mit der `for`-Schleife verwendet werden und haben dieselbe Bedeutung. Außerdem kann eine `for`-Schleife analog zur `while`-Schleife über einen `else`-Zweig verfügen, der genau dann ausgeführt wird, wenn die Schleife vollständig durchgelaufen ist und nicht mittels `break` vorzeitig abgebrochen wurde.

5.2.2.1 Die for-Schleife als Zählschleife

Im Zusammenhang mit der `for`-Schleife ist die eingebaute Funktion `range()` besonders interessant. Sie erzeugt ein iterierbares Objekt, das alle ganzen Zahlen eines bestimmten Bereichs durchläuft. Man kann der `range()` Funktion bis zu drei Parameter übergeben:

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Der Platzhalter *start* steht dabei für die Zahl, mit der begonnen wird. Die Schleife wird beendet, sobald *stop* erreicht wurde. **ACHTUNG!** Der Schleifenzähler selbst erreicht niemals den Wert *stop*, er bleibt stets um eins kleiner!
In jedem Schleifendurchlauf wird der Schleifenzähler um *step* erhöht. Sowohl *start* als auch *stop* und *step* müssen ganze Zahlen sein, könne aber auch negativ sein.
Wenn alle Werte angegeben sind, sieht die for-Schleife folgendermaßen aus:

```
#Vorwaertszaehlen von 1 bis inkl. 9 mit Schrittweite 2
for i in range(1,10,2):
        print(i)

#Rueckwaertszaehlen von 10 bis inkl. 2 mit Schrittweite -4
for i in range(10,1,-4):
        print(i)
```

5.3 Die pass-Anweisung

Während der Entwicklung eines Programms kommt es vor, dass eine Kontrollstruktur vorerst nur teilweise implementiert wird. Der Programmierer erstellt einen Anweisungskopf, fügt aber keinen Anweisungskörper an, da er sich vielleicht zuerst um andere, wichtigere Dinge kümmern möchte. Ein in der Luft hängender Anweisungskopf ohne entsprechenden Körper ist aber ein Syntaxfehler.
Zu diesem Zweck existiert die `pass`-Anweisung – sie teilt dem Interpreter mit an dieser Stelle einfach gar nichts zu machen. Sie kann folgendermaßen angewendet werden:

```
if x == 1:
    pass
elif x == 2:
    print("x hat den Wert 2")
```

In diesem Fall ist im Körper der `if`-Anweisung nur `pass` zu finden. Sollte x also den Wert 1 haben, passiert schlicht und einfach nichts.
Die `pass`-Anweisung hat den Zweck, Syntaxfehler in vorläufigen Programmversionen zu vermeiden. Fertige Programme enthalten in der Regel keine `pass`-Anweisungen.

6 Funktionen

Funktionen werden in der Programmierung dazu eingesetzt, um Redundanzen im Quellcode zu vermeiden. Das bedeutet, dass Code-Stücke, die in der gleichen oder einer ähnlichen Form öfter im Programm benötigt werden, nicht jedes Mal neu geschrieben, sondern in einer Funktion gekapselt werden. Diese Funktion kann dann an den Stellen, an denen sie benötigt wird, aufgerufen werden. Darüber hinaus bilden Funktionen ein elegantes Hilfsmittel, um einen langen Quellcode sinnvoll in Unterprogramme aufzuteilen. Das erhöht die Les- und Wartbarkeit des Codes.
Eine Funktion besteht aus drei Dingen:

- Eine Funktion muss einen **Namen** haben, über den sie in anderen Teilen des Programms aufgerufen werden kann. Die Zusammensetzung des Funktionsnamens erfolgt nach denselben Regeln wie die Namensgebung einer Referenz.
- Eine Funktion muss eine **Schnittstelle** haben, über die Informationen vom aufrufenden Programmteil in den Kontext der Funktion übertragen werden. Eine Schnittstelle kann aus beliebig vielen (unter Umständen auch keinen) **Parametern** bestehen. Funktionsintern wird jedem dieser Parameter ein Name gegeben. Sie lassen sich dann wie Referenzen im Funktionskörper verwenden.
- Eine Funktion muss einen Wert **zurückgeben** (mit dem key-word `return`). Jede Funktion gibt automatisch `None` zurück, wenn der Rückgabewert nicht ausdrücklich angegeben wurde.

Zur Definition einer Funktion wird in Python das Schlüsselwort `def` verwendet.
Beispiel: Fakultät einer Zahl berechnen:

```
def factorial(zahl):
    if zahl < 0:
        return None
    result = 1
    for i in range(2, zahl+1):
        result *= i
    return result

while True:
    eingabe = input("Geben Sie eine Zahl ein: ")
    if eingabe == "end":
        print("bye!")
        break
    else:
        eingabe = int(eingabe)
    ergebnis = factorial(eingabe)
    if ergebnis is None:
        print("Fehler bei der Berechnung")
    else:
        print("Die Fakultät der Zahl {} ist {}".format(eingabe,ergebnis))
```

Output:

```
Geben Sie eine Zahl ein: 4
Die Fakultät der Zahl 4 ist 24
Geben Sie eine Zahl ein: -5
Fehler bei der Berechnung
Geben Sie eine Zahl ein: 10
Die Fakultät der Zahl 10 ist 3628800
Geben Sie eine Zahl ein: end
bye!
```

Selbstverständlich man im Quelltext mehrere eigene Funktionen definieren und aufrufen. Das folgende Beispiel soll bei Eingabe einer negativen Zahl keine Fehlermeldung, sondern die Fakultät des Betrags dieser Zahl ausgeben:

```
def betrag(zahl):
    if zahl < 0:
        return -zahl
    else:
        return zahl

def factorial(zahl):
    result = 1
    for i in range(2, zahl+1):
        result *= i
    return result

while True:
    eingabe = input("Geben Sie eine positive Zahl ein: ")
    if eingabe == "end":
        print("bye!")
        break
    else:
        eingabe = int(eingabe)
        if eingabe < 0:
            eingabe = betrag(eingabe)
    ergebnis = factorial(eingabe)
    print("Die Fakultät der Zahl {} ist {}".format(eingabe,ergebnis))
```

6.1 Funktionsparameter

6.1.1 Optionale Parameter

Manchmal möchte man einer Funktion noch zusätzliche Funktionalität ermöglichen. Dies kann mit Hilfe von optionalen Parametern geschehen. Optionale Parameter müssen am Ende der Parameter-Liste stehen und eine default Wert zugewiesen bekommen.
Im folgenden Beispiel definieren wir eine Funktion `div_and_round()` welche zwei Zahlen dividieren soll und optional das Ergebnis auf eine Anzahl Nachkommastellen runden soll.
Wir brauchen also mindestens zwei Parameter (die zwei Zahlen die wir dividieren) und eine optionalen Parameter für die Rundungsgenauigkeit.

Beispiel-Code:

```
def div_and_round(a,b,c=None):
    erg = a/b
    return round(erg,c)

#Aufruf mit nur zwei Parametern
print(div_and_round(12,7))

#Aufrufe mit dem optionalen dritten Parameter
print(div_and_round(12,7,2))
print(div_and_round(12,7,5))
```

Zugegehöriger Output:

```
2
1.71
1.71429
```

6.1.2 Beliebige Anzahl von Parametern

Wenn man bei der Funktionsdefinition einen Parameternamen einen * voranstellt, wird intern unter diesem Parameternamen ein Tupel erzeugt, dass alle übergebenen (beliebig viele) Parameter enthält.
Im folgenden Beispiel soll eine Funktion `summe()` definiert werden, welche mindestens zwei Zahlen als Parameter benötigt, aber auch mit beliebig vielen mehr funktioniert.

Beispiel-Code:

```
def summe(a,b,*weitere):
    erg = a + b
    for x in weitere:
        erg = erg + x
    return erg

#Aufruf mit nur zwei Parametern
print(summe(12,7))
#Aufrufe mit beliebig vielen Parametern
print(summe(12,7,6,7,12))
print(summe(12,7,45,3,9,2,13))
```

Zugegehöriger Output:

```
19
44
91
```

6.1.3 Entpacken einer Parameterliste

Wenn man eine Funktion definiert hat, welcher man beliebig viele Parameter übergeben kann, wäre es gut wenn man Collections (Listen, Tuples oder Dictionarys) übergeben könnte und nicht Einzelparameter.
Dies funktioniert durch ein einen vorangestellte * für Tuples und Listen und zwei ** bei Dictionarys (Achtung! Dann muss bei der Definition der Funktion auch zwei ** verwendet werden).
Diese Technik wird *unpacking* genannt.

Beispiel-Code:

```
def summe(*beliebig_viele):
    erg = 0
    for x in beliebig_viele:
        erg = erg + x
    return erg

my_list = [1,2,3,4,5,6,7,8,9]
#Aufruf mit einer Liste
print(summe(*my_list))
#Aufrufe mit einem Tuple von range()
print(summe(*range(101)))
```

Zugegehöriger Output:

```
45
5050
```

6.2 Namensräume - name space

Zunächst einmal müssen zwei Begriffe unterschieden werden. Wenn wir uns im Kontext einer Funktion, also im Funktionskörper, befinden, dann können wir dort selbstverständlich Referenzen und Instanzen erzeugen und verwenden. Diese haben jedoch nur unmittelbar in der Funktion selbst Gültigkeit. Sie existieren im **lokalen** Namensraum. Im Gegensatz dazu existieren Referenzen des Hauptprogramms im **globalen** Namensraum. Begrifflich wird auch zwischen globalen Referenzen und lokalen Referenzen unterschieden. Dazu folgendes Beispiel:
vspace.1cm
Ist a gleich a ???:

```
def f(a):
    print(a)

a = 12
f(42)
```

Zugegehöriger Output:

```
42
```

und nicht 12 !!!

6.2.1 Zugriff auf den globalen Namensraum

Im lokalen Namensraum eines Funktionskörpers kann jederzeit lesend auf eine globale Referenz zugegriffen werden, solange keine lokale Referenz gleichen Namens existiert: Beispiel:

Lesezugriff auf globale Referenz:

```
def f():
    print(b)

b = 12
f()
```

Zugegehöriger Output:

```
12
```

Sobald versucht wird, schreibend auf eine globale Referenz zuzugreifen, wird stattdessen eine entsprechende lokale Referenz erzeugt: Beispiel:

Schreibzugriff auf globale Referenz geht so nicht:

```
def f():
    b = 7
    print("b in der f() = {}".format(b))

b = 12
f()
print("b im globalen Namensraum = {}".format(b))
```

Zugegehöriger Output:

```
b in der f() = 7
b im globalen Namensraum = 12
```

... geht so also nicht!

Eine Funktion kann dennoch mithilfe der `global`-Anweisung schreibend auf eine globale Referenz zugreifen. Dazu muss im Funktionskörper das Schlüsselwort `global` geschrieben werden, gefolgt von einer oder mehreren globalen Referenzen: Beispiel:

Schreibzugriff auf globale Referenz geht so nicht:

```
def f():
    global b
    b = 7
    print("b in der f() = {}".format(b))

b = 12
f()
print("b im globalen Namensraum = {}".format(b))
```

Zugegehöriger Output:

```
b in der f() = 7
b im globalen Namensraum = 7
```

... so geht so also doch!

6.3 Funktionen - fortgeschrittene Themen und Standard-Funktionen

Es gäbe noch einige Dinge zum Thema Funktionen zu besprechen, wie z.B. anonyme Funktionen mittels Lambda oder Rekursion uvm. aber dies würde den Umfang dieses Skripts sprengen. Zum Vertiefen in diese Themen bitte die Suchmaschine ihrer Wahl verwenden!
Zur Beschreibung der Standard-Funktionen von Python verweise ich wieder auf die offizielle Python-Doc:
`https://docs.python.org/3/library/functions.html`

7 Module und die Standardbibliothek

Unter Modularisierung versteht man die Aufteilung des Quelltextes in sogenannte Module. Ein Modul stellt üblicherweise Datentypen und Funktionen bereit, die einem bestimmten Zweck dienen, beispielsweise der Arbeit mit Dateien eines bestimmten Dateiformats. Module können in einem Programm eingebunden werden und stellen dem Programmierer dann die enthaltene Funktionalität zur Verfügung. Grundsätzlich gibt es zwei Arten von Modulen:

- Zum einen kann jedes Python-Programm globale Module, auch Bibliotheken genannt, einbinden. Globale Module werden systemweit installiert und stehen allen Python-Programmen gleichermaßen zur Verfügung. Es ist möglich, eigene globale Module zu schreiben oder ein globales Modul eines Drittanbieters zu installieren.
- Die zweite Möglichkeit zur Modularisierung sind lokale Module. Darunter versteht man die Kapselung einzelner Programmteile in eigene Programmdateien. Diese Dateien können wie Bibliotheken eingebunden werden, sind aber in keinem anderen Python-Programm verfügbar. Diese Form der Modularisierung hilft bei der Programmierung, da sie dem Programmierer die Möglichkeit gibt, langen Programm-Code überschaubar auf verschiedene Programmdateien aufzuteilen.

In Python besteht der einzige Unterschied zwischen lokalen und globalen Modulen darin, wo sie gespeichert sind. Während sich lokale Module in der Regel im Programmverzeichnis bzw. in einem seiner Unterverzeichnisse befinden, sind globale Module in einigen festgelegten Verzeichnissen der Python-Installation gespeichert.

7.1 Einbinden globaler Module

Ein **globales Modul**, sei es ein Teil der Standardbibliothek oder ein selbst geschriebenes, kann mithilfe der `import`-Anweisung eingebunden werden. Zum Beispiel das Modul `math` der Standardbibliothek. Das ist ein Modul, das mathematische Funktionen wie `sin` oder `cos` sowie mathematische Konstanten wie `pi` bereitstellt. Um sich diese Funktionalität in einem Programm zunutze machen zu können, wird die `import`-Anweisung in der folgenden Form verwendet:

```
#ein einzelnes Modul importieren
import math

#mehrere Module gleichzeitig importieren
import math, random

#oder besser ein Modul pro Zeile
import math
import random
```

Nachdem ein Modul eingebunden wurde, wird ein neuer *Namensraum (name-space)* mit seinem Namen erstellt. Über diesen sind alle Funktionen, Datentypen und Werte des Moduls im Programm nutzbar. Mit einem Namensraum kann wie mit einer Instanz umgegangen werden, und die Funktionen des Moduls können wie Methoden des Namensraums verwendet werden. Ein kurzes Beispiel aus dem interaktiven Modus:

```
>>> import math
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> math.pi
3.141592653589793
>>> math.sin(0)
0.0
>>> math.cos(0)
1.0
```

Man sieht der Zugriff auf die Referenz `pi` kann nur über den name-space `math.pi` zugegriffen werden.
Es ist aber auch möglich, den Namen des name-spce durch eine import/as-Anweisung festzulegen:

```
>>> import math as m
>>> m.pi
3.141592653589793
```

Darüber hinaus kann die `import`-Anweisung so verwendet werden, dass kein eigener Namensraum für das eingebundene Modul erzeugt wird, sondern alle oder nur ausgewählte Elemente des Moduls in den globalen Namensraum des Programms eingebunden werden. Dies geschieht mit dem key-word `from`:

```
#alle Elemente des Moduls importieren
>>> from math import *
>>> pi
3.141592653589793

#ausgewählte Elemente des Moduls importieren
>>> from math import (sin,pi)
>>> pi
3.141592653589793
>>> sin(0)
0.0
>>> cos(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
```

7.2 Lokale Module

Wenn man z.B. ein nützliches kleines Programm geschrieben hat und die Funktionalität dieses Programms in anderen, eigenen Programmen weiterverwenden möchte, kann man es als Modul importieren. Nehmen wir an, dass wir vor einiger Zeit ein kleines Mathematik Programm geschrieben haben, nämlich *mathehelfer.py*.

```
def fak(n):
    ergebnis = 1
    for i in range(2, n+1):
        ergebnis *= i
    return ergebnis

def kehr(n):
    return 1 / n
```

Üblicherweise sollten selbstgeschriebene Module im Unterverzeichnis *site-packages* der Python-Installation speichern. Dort werden üblicherweise auch Bibliotheken von Drittanbietern installiert.
Alternativ kann auch das `sys` Modul geladen werden und der Pfad zu den lokalen Modulen zum `sys.path` mittels `.append()` angehängt werden. z.B.

```
import sys
# the mathehelfer-v01 dir contains mathehelfer.py
sys.path.append('/foo/bar/mathehelfer-v01')

import mathehelfer
```

Beziehungsweise ist gar keine Pfadangabe notwendig wenn das Modul im gleichen Unterverzeichnis liegt wie das Programm, das es importiert.
Achtung: beim import Statement ist dann nur der Modulname ohne File-Extension anzugeben. Also mathehelfer und NICHT mathehelfer.py

7.3 Die Standard Bibliothek

Die Standardbibliothek enthält Module und Pakete für mathematische Hilfsfunktionen, reguläre Ausdrücke, den Zugriff auf Informationen des Betriebssystems oder auf das Dateisystem, parallele Programmierung, Datenspeicherung sowie Netzwerkkommunikation und zum Debugging bzw. zur Qualitätssicherung.
`https://docs.python.org/3/library/index.html`

8 Dateien

Bevor wir Dateien behandeln müssen wir kurz das Konzept von Datenströmen (*data streams*) behandeln.
Unter einem *data stream* versteht man eine kontinuierliche Folge von Daten. Dabei werden zwei Typen unterschieden: Von eingehenden Datenströmen (*downstreams*) können Daten gelesen und in ausgehende Datenströme (*upstreams*) geschrieben werden. Bildschirmausgaben, Tastatureingaben sowie Dateien und Netzwerkverbindungen werden als Datenstrom betrachtet.
Es gibt zwei Standarddatenströme, die wir bereits verwendet haben, ohne es zu wissen: Sowohl die Ausgabe eines Strings auf dem Bildschirm als auch eine Benutzereingabe sind nichts anderes als Operationen auf den Standardeingabe- bzw. -ausgabeströmen *stdin* und *stdout*. Auf den Ausgabestrom *stdout* kann mit der eingebauten Funktion `print()` geschrieben und von dem Eingabestrom *stdin* mittels `input()` gelesen werden.
Einige Betriebssysteme erlauben es, Datenströme im *Text-* und *Binärmodus* zu öffnen. Der Unterschied besteht darin, dass im Textmodus bestimmte Steuerzeichen berücksichtigt werden. So wird ein im Textmodus geöffneter Strom beispielsweise nur bis zum ersten Auftreten des sogenannten `EOF`-Zeichens gelesen, das das Ende einer Datei (*end of file*) signalisiert. Im Binärmodus hingegen wird der vollständige Inhalt des Datenstroms eingelesen.
Als letzte Unterscheidung gibt es Datenströme, in denen man sich beliebig positionieren kann, und solche, in denen das nicht geht. Eine Datei stellt zum Beispiel einen Datenstrom dar, in dem die Schreib-/Leseposition beliebig festgelegt werden kann. Beispiele für einen Datenstrom, in dem das nicht funktioniert, sind der Standardeingabestrom (*stdin*) oder eine Netzwerkverbindung.

8.1 Daten aus einer Datei lesen

Zunächst einmal muss die Datei zum Lesen geöffnet werden. Dazu verwenden wir die Built-in Function `open()`. Diese gibt ein sogenanntes Dateiobjekt (*file object*) zurück:

```
fobj = open("cd-sammlung.csv","r")
```

Als ersten Parameter von `open()` übergeben wir einen String, der den Pfad zur gewünschten Datei enthält. Es sind sowohl relative als auch absolute Pfade erlaubt! Ein absoluter Pfad identifiziert eine Datei ausgehend von der Wurzel im Dateisystembaum. [1] Ein relativer Pfad bezieht sich auf das aktuelle Arbeitsverzeichnis des Programms. Hier kann die Verknüpfung »..« für das übergeordnete Verzeichnis verwendet werden. Im Beispiel ist ein relativer Pfad angegeben, die Datei *cd-sammlung.csv* muss sich also im gleichen Verzeichnis befinden wie das Programm.
Der zweite Parameter ist ebenfalls ein String und spezifiziert den Modus, in dem die Datei geöffnet werden soll, wobei *"r"* für »*read*« steht und bedeutet, dass die Datei zum Lesen geöffnet wird. Das von der Funktion zurückgegebene Dateiobjekt verknüpfen wir mit der Referenz `fobj`. Sollte die Datei nicht vorhanden sein, wird ein `FileNotFoundError` erzeugt, dazu aber noch später mehr im Kapitel 9 auf Seite 29.

Nachdem `open()` aufgerufen wurde, können mit dem Dateiobjekt Daten aus der Datei gelesen werden. Nachdem das Lesen der Datei beendet worden ist, muss sie explizit durch Aufrufen der Methode `.close()` geschlossen werden:

```
fobj.close()
```

Nach Aufruf dieser Methode können keine weiteren Daten mehr aus dem Dateiobjekt gelesen werden.

[1] Unter Windows könnte ein absoluter Pfad folgendermaßen aussehen:
C:\Programme\TestProgramm\cd-sammlung.csv
und unter Linux:
/home/TestProgram/cd-sammlung.csv

Im nächsten Schritt möchten wir die Datei zeilenweise auslesen. Dies ist relativ einfach, da das Dateiobjekt zeilenweise iterierbar ist. Wir können also die altbekannte `for`-Schleife verwenden:

Zeilenweises Lesen des Files:

```
fobj = open("cd-sammlung.csv", "r")
for line in fobj:
    print(line)
fobj.close()
```

Zugegehöriger Output:

```
CD-Code,Title

CL01,Vivaldi Die vier Jahreszeiten

CL02,Bach Toccata und Fuge

MT01,Judas Priest Screaming for Vengeance

MT02,Metallica Black Album

GR01,Nirvana Nevermind

GR02,Green Day Dookie


```

Hm, wo kommen die extra Leerzeilen her? Sind sind im File so nicht enthalten!
Wenn wir das file nochmals im binary mode öffnen sehen wir am Ende jeder Zeile das Steuerzeichen `\n` welches eigentlich den Zeilensprung anzeigt. Zur Erinnerung: Die Steuerzeichen wurden schon im Kapitel 4.3.3.1 auf Seite 12 besprochen!

Zeilenweises Lesen des Files im Byte Modus:

```
fobj = open("cd-sammlung.csv", "rb")
for line in fobj:
    print(line)
fobj.close()
```

Zugegehöriger Output:

```
b'CD-Code,Title\n'
b'CL01,Vivaldi Die vier Jahreszeiten\n'
b'CL02,Bach Toccata und Fuge\n'
b'MT01,Judas Priest Screaming for Vengeance\n'
b'MT02,Metallica Black Album\n'
b'GR01,Nirvana Nevermind\n'
b'GR02,Green Day Dookie\n'
```

Da wir aber die `print()`-Funktion zum Ausgeben verwenden, welche standardmäßig sowieso einen Zeilensprung anhängt wirkt es als ob eine Leerzeile vorhanden wäre.
Im Kapitel 4.3.3.2 auf Seite 12 haben wir eine kleine Auswahl an String-Methoden kennengelernt. Eine davon ist die `.strip()` Methode, mit der wir Teile eines Strings entfernen können. Verwenden wir sie um das Steuerzeichen `\n` aus dem String zu entfernen:

Zeilenweises Lesen des Files und Steuerzeichen entfernen:

```
fobj = open("cd-sammlung.csv", "r")
for line in fobj:
    print(line.strip("\n"))
fobj.close()
```

Zugegehöriger Output:

```
CD-Code,Title
CL01,Vivaldi Die vier Jahreszeiten
CL02,Bach Toccata und Fuge
MT01,Judas Priest Screaming for Vengeance
MT02,Metallica Black Album
GR01,Nirvana Nevermind
GR02,Green Day Dookie
```

Nun haben wir den gewünschten Output!
Mit der `.split()` Methode könnte wir nun den Zeilenstring noch weiter aufteilen, in dem wir als Trennzeichen das Komma ",ängeben, und so den File-Inhalt weiterverarbeiten. Zum Beispiel in diesem kleine CD-Code Abfrage Programm:

CD-Code Abfrage Programm:

```
fobj = open("cd-sammlung.csv", "r")
sammlung = {}
for line in fobj:
    line = line.strip("\n")
    mini_dict = line.split(",")
    sammlung[mini_dict[0]] = mini_dict[1]
fobj.close()

while(True):
    cd_code = input("Bitte CD-Code eingeben ->")
    if cd_code in sammlung:
        print("Der CD-Code verweist auf:", sammlung[cd_code])
    else:
        if (cd_code == "end"):
            print("bye!")
            break
        print("Unbekannter CD-Code")
```

Zugegehöriger Output:

```
Bitte CD-Code eingeben ->MT01
Der CD-Code verweist auf: Judas Priest Screaming for Vengeance
Bitte CD-Code eingeben ->GR01
Der CD-Code verweist auf: Nirvana Nevermind
Bitte CD-Code eingeben ->end
bye!
```

8.2 Daten in eine Datei schreiben

Um eine Datei zum Schreiben zu öffnen, verwenden wir ebenfalls die Built-in Function `open()`. Diese Funktion erwartet einen Modus als zweiten Parameter, der im letzten Abschnitt `"r"` für *»read«* sein musste. Analog dazu muss `"w"` (für *»write«*) angegeben werden, wenn die Datei zum Schreiben geöffnet werden soll. Sollte die gewünschte Datei bereits vorhanden sein, wird sie geleert. Eine nicht vorhandene Datei wird erstellt.
Es gibt außer `"r"` und `"w"` noch andere Möglichkeiten, siehe dazu Tabelle 14 auf Seite 28.
File zum Schreiben öffnen:

```
fobj = open("ausgabe.txt", "w")
```

und wie vorher gilt, dass die Datei nach dem Schreiben geschlossen werden muss:

```
fobj.close()
```

Als kleines Beispiel wollen wir die Zahlen von 1 bis 4 zeilenweise in ein File schreiben:

Zeilenweises Schreiben:

```
fobj = open("numbers.txt","w")
for i in range(1,5):
    fobj.write("{}\n".format(i))
fobj.close()
```

File-Inhalt:

```
1
2
3
4
```

Bei Schreiben müssen wir uns selbst um den Zeilenumbruch kümmern, deshalb ist in der `.write()` Methode explizit das Steuerzeichen `\n` angegeben.

8.3 Mehr zum File-object

8.3.1 Optionen bei der Erzeugung

Die Built-in Function `open()` öffnet eine Datei und gibt das erzeugte Dateiobjekt zurück.
Die ersten beiden Parameter haben wir in den vorangegangenen Abschnitten bereits besprochen. Dabei handelt es sich um den Dateinamen bzw. den Pfad zur zu öffnenden Datei und um den Modus, in dem die Datei zu öffnen ist.

Für den Parameter *mode* muss ein String übergeben werden, wobei alle gültigen Werte und ihre Bedeutung in Tabelle 14 aufgelistet sind:

Modus	Beschreibung
`"r"`	Die Datei wird ausschließlich zum Lesen geöffnet.
`"w"`	Wird zum Schreiben geöffnet. Bestehende Datei wird überschrieben.
`"a"`	Wird zum Schreiben geöffnet. Bestehende Datei gleichen Namens wird **nicht** überschrieben, sondern erweitert.
`"x"`	Wird zum Schreiben geöffnet. Wenn bereits vorhanden wird eine FileExistsError-Exception geworfen.
`"r+", "w+","a+", "x+"`	Die Datei wird zum Lesen und Schreiben geöffnet.
`"rb", "wb", "ab", "xb", "r+b", "w+b", "a+b", "x+b"`	Die Datei wird im Binärmodus geöffnet. Achtung: In diesem Fall werden bytes-Instanzen anstelle von Strings verwendet.

Tabelle 14: File modi

Es gibt noch weitere Parameter wie z.B. das verwendete File-Encoding, was aber den Umfang dieser Einführung sprengen würde. Für mehr Informationen bitte einen Blick auf die offizielle Python Docs werfen:
`https://docs.python.org/3/library/functions.html#open`

8.4 Methoden und Attribute eines File-objects

Die beim Öffnen angegebenen Parameter können über die Attribute `name`, `encoding`, `errors`, `mode` und `newlines` des resultierenden Dateiobjekts wieder gelesen werden.
Die folgende Tabelle 15 fasst die wichtigsten Methoden eines Dateiobjekts kurz zusammen:

Methode	Beschreibung
`close()`	Schließt ein bestehendes Dateiobjekt. Danach kann keine Lese- oder Schreiboperationen mehr durchgeführt werden.
`fileno()`	Gibt den Deskriptor der geöffneten Datei als ganze Zahl zurück.
`flush()`	Verfügt, dass anstehende Schreiboperationen sofort ausgeführt werden.
`isatty()`	True, wenn das Dateiobjekt auf einem Datenstrom geöffnet wurde, der nicht an beliebiger Stelle geschrieben oder gelesen werden kann.
`next()`	Liest die nächste Zeile der Datei ein und gibt sie als String zurück.
`read([size])`	Liest size Bytes der Datei ein oder weniger, wenn vorher das Ende der Datei erreicht wurde. Sollte size nicht angegeben sein, wird die Datei vollständig eingelesen. Die Daten werden abhängig vom Lesemodus als String oder bytes-String zurückgegeben.
`readline([size])`	Liest eine Zeile der Datei ein. Durch Angabe von size lässt sich die Anzahl der zu lesenden Bytes begrenzen.
`readlines([sizehint])`	Liest alle Zeilen und gibt sie in Form einer Liste von Strings zurück. Sollte sizehint angegeben sein, wird nur gelesen, bis ungefähr sizehint Bytes gelesen wurden.
`seek(offset, [whence])`	Setzt die aktuelle Schreib-/Leseposition in der Datei auf offset.
`tell()`	Liefert die aktuelle Schreib-/Leseposition in der Datei.
`truncate([size])`	Löscht in der Datei alle Daten, die hinter der aktuellen Schreib-/Leseposition stehen, bzw. – sofern angegeben – alles außer den ersten size Bytes.
`write(str)`	Schreibt den String str in die Datei.
`writelines(iterable)`	Schreibt mehrere Zeilen in die Datei. Das iterierbare Objekt iterable muss Strings durchlaufen, möglich ist zum Beispiel eine Liste von Strings.

Tabelle 15: Methoden eines File-objects

8.4.1 Schreib-/Leseposition veränder

Aufgrund der speziellen Natur von Dateien ist es möglich, die Schreib- bzw. Leseposition beliebig zu verändern. Dazu existieren die Methoden `.seek()` und `.tell()` des Dateiobjekts.
Die Methode `.seek()` eines Dateiobjekts setzt die Schreib-/Leseposition innerhalb der Datei. Sie ist das Gegenstück zur Methode `.tell()`, die die aktuelle Schreib-/Leseposition zurückgibt.
Dieses Thema sprengt aber auch den Rahmen dieser Python Einführung und deshalb sei an dieser Stelle wieder auf

die offizielle Python Doc verwiesen:
`https://docs.python.org/3/tutorial/inputoutput.html#tut-files`

9 Exceptions

Das *Exception-Handling* ist ein wichtiges Programmierkonzept. Es geht darum wie sich ein Programm verhalten soll wenn ein Fehler zur Laufzeit auftritt also z.B. wenn ein File auf das man zugreifen möchte nicht vorhanden ist.
Nehmen wir an eine Funktion, die wiederum von einer anderen Funktion aufgerufen wurde, läuft in einen Fehler. Im diesem Fall erzeugt die Unterfunktion dann eine sogenannte *Exception* und wirft sie, bildlich gesprochen, nach oben. Die Ausführung der Funktion ist damit beendet. Jede übergeordnete Funktion hat jetzt drei Möglichkeiten:

- Sie fängt die *Exception* ab, führt den Code aus, der für den Fehlerfall vorgesehen ist, und fährt dann normal fort. In einem solchen Fall bemerken weitere übergeordnete Funktionen die *Exception* nicht.
- Sie fängt die *Exception* ab, führt den Code aus, der für den Fehlerfall vorgesehen ist, und wirft die *Exception* weiter nach oben. In einem solchen Fall ist auch die Ausführung dieser Funktion sofort beendet, und die übergeordnete Funktion steht vor der Wahl, die *Exception* abzufangen oder nicht.
- Sie lässt die *Exception* passieren, ohne sie abzufangen. In diesem Fall ist die Ausführung der Funktion sofort beendet, und die übergeordnete Funktion steht vor der Wahl, die *Exception* abzufangen oder nicht.

Es gibt eine Reihe von eingebauten *Exception*, aber man kann als Programmiere auch eigene *Exception* erzeugen.

9.1 Abfangen einer Exception

Zum Abfangen einer *Exception* wird eine `try/except`-Anweisung verwendet. Eine solche Anweisung besteht zunächst aus zwei Teilen:

- Der `try`-Block wird durch das Schlüsselwort `try` eingeleitet, gefolgt von einem Doppelpunkt und einem beliebigen Code-Block, der um eine Ebene weiter eingerückt ist. In diesem Code-Block wird der, möglicherweise gefährliche, Code ausgeführt. Wenn in diesem Code-Block eine *Exception* auftritt, wird seine Ausführung sofort beendet und der `except`-Zweig der Anweisung ausgeführt.
- Der `except`-Zweig wird durch das Schlüsselwort `except` eingeleitet, gefolgt von einer optionalen Liste von *Exception*-Typen, für die dieser `except`-Zweig ausgeführt werden soll. Achtung: mehrere *Exception*-Typen müssen in Form eines Tupels angegeben werden. Hinter der Liste der *Exception*-Typen kann, ebenfalls optional, das Schlüsselwort `as` stehen, gefolgt von einem frei wählbaren Bezeichner. Hier kann festgelegt werden unter welchem Namen man auf die gefangene *Exception*-Instanz im `except`-Zweig zugreifen kann. Auf diesem Weg kann man beispielsweise auf die in dem `args`-Attribut der *Exception*-Instanz abgelegten Informationen zugreifen.
  Danach folgen ein Doppelpunkt und, um eine Ebene weiter eingerückt, ein beliebiger Code-Block. Dieser Code-Block wird nur dann ausgeführt, wenn innerhalb des `try`-Blocks eine der aufgelisteten *Exception* geworfen wurde.

Beispiel, wir schreiben eine kleine Funktion, der wir als Parameter einen Filenamen übergeben und die für uns das File öffnen soll und und das File-object zurückgeben soll:

```
def get_file_object(name):
    try:
        return open(name)
    except (FileNotFoundError, TypeError):
        print("Error: Either wrong name-type or the file is not there!")
        return None
```

Wir könnten das aber noch genauer machen indem wir mehrere `except`-Zweige verwenden:

```
def get(name):
    try:
        return open(name)
    except FileNotFoundError:
        print("Wrong File-name or the file does not exist!")
        return None
    except TypeError:
        print("name should be of type string!")
        return None
```

Umgekehrt könne wir auch alle Arten von *Exceptions* auf einmal abfangen. Dazu wird ein `except`-Zweig ohne Angabe eines *Exception*-Typs geschrieben:

```
def get(name):
    try:
        return open(name)
    except:
        print("Something went wrong!!!")
        return None
```

Zusätzlich kann eine `try/except`-Anweisung über einen `else`- und einen `finally`-Zweig verfügen, die jeweils nur einmal pro Anweisung vorkommen dürfen. Der dem `else`-Zweig zugehörige Code-Block wird ausgeführt, wenn keine *Exception* aufgetreten ist, und der dem `finally`-Zweig zugehörige Code-Block wird in jedem Fall nach Behandlung aller *Exceptions* und nach dem Ausführen des entsprechenden else-Zweiges ausgeführt, egal, ob oder welche *Exceptions* vorher aufgetreten sind. Dieser `finally`-Zweig eignet sich daher besonders für Dinge, die in jedem Fall erledigt werden müssen, wie beispielsweise das Schließen eines Dateiobjekts.

9.2 Werfen einer Exception

Es ist auch möglich, mithilfe der `raise`-Anweisung selbst eine *Exception* zu werfen. Dazu wird das Schlüsselwort `raise`, gefolgt von einer Instanz, geschrieben. Diese darf nur Instanz einer von BaseException abgeleiteten Klasse sein.
Beispiel:

```
def give_me_an_int(number):
    if (type(number) == int):
        print("Thanks!")
    else:
        raise TypeError("He, das ist kein Integer!")

give_me_an_int(12)
give_me_an_int("twelve")
```

Output:

```
Thanks!
Traceback (most recent call last):
  File "exception_01.py", line 10, in <module>
    give_me_an_int("twelve")
  File "exception_01.py", line 5, in give_me_an_int
    raise TypeError("He, das ist kein Integer!")
TypeError: He, das ist kein Integer!
```

An diesem Beispiel sieht man auch sehr gut wie Python versucht mittels `Traceback` den Fehlerhergang zu rekonstruieren! Der Funktionsaufruf in line 10 mit dem Parameter `"twelve"` hat dann in der Funktion `give_me_an_int` in 5 den Fehler verursacht.

10 Nützliches

Zum Abschluss möchte ich noch mals die sehr gute offizielle Python-Doc in Erinnerung rufen:
`https://docs.python.org/3/`
Und auch darauf hinweisen, dass das **sorgfältige Lesen von Fehlermeldungen** oft schneller zum Ziel führt als WWW-Suchen, die oft in Foren führen und meist geraume Zeit beanspruchen.

Viel Spaß mit Python!