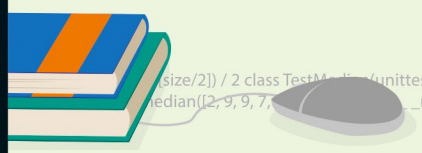
Thomas Theis

Mit Kapitel zum Raspberry Pi



Einstieg in Python

Ideal für Programmiereinsteiger

6. Auflage

- Schritt für Schritt eigene Programme entwickeln
- Mit vielen Beispielen und Übungsaufgaben
- GUI, OOP, Datenbank- und Internetanwendungen u. v. m.

Alle Codebeispiele zum Download

Rheinwerk Computing

Kapitel 3
Programmierkurs

Der folgende Programmierkurs mit ausführlichen Erläuterungen führt Sie schrittweise in die Programmierung mit Python ein. Begleitet wird der Kurs von einem Programmierprojekt, das die vielen Teilaspekte zu einem Ganzen verknüpft.

3.1 Ein Spiel programmieren

Damit Sie die Programmiersprache Python auf unterhaltsame Weise kennenlernen, werden Sie im Folgenden ein Spiel programmieren. Es wird im Verlauf des Buchs kontinuierlich erweitert und verbessert. Zunächst wird der Ablauf des Spiels beschrieben.

Kopfrechnen

Nach Aufruf des Programms wird dem Benutzer eine Kopfrechenaufgabe gestellt. Er gibt das von ihm ermittelte Ergebnis ein, und das Programm bewertet seine Eingabe.

```
Die Aufgabe: 9 + 26
Bitte eine Zahl eingeben:
34
34 ist falsch
Bitte eine Zahl eingeben:
35
35 ist richtig
Ergebnis:  35
Anzahl Versuche: 2
```

Das Spiel wächst

Das Spiel wird in mehreren Einzelschritten erstellt. Zunächst entsteht eine ganz einfache Version. Mit zunehmenden Programmierkenntnissen entwickeln Sie immer komplexere Versionen. Die im jeweiligen Abschnitt erlernten Programmierfähigkeiten setzen Sie unmittelbar zur Verbesserung des Spielablaufs ein.

Versionen

In späteren Abschnitten des Buchs entstehen weitere Versionen des Spiels. Es begleitet Sie auf diese Weise durch das gesamte Buch. Unter anderem wird es um die folgenden Möglichkeiten erweitert:

- Es werden mehrere Aufgaben gestellt.
- Die benötigte Zeit wird gemessen.
- Der Name des Spielers und die benötigte Zeit werden als Highscore-Liste dauerhaft in einer Datei oder einer Datenbank gespeichert.
- Die Highscore-Liste wird mit neuen Ergebnissen aktualisiert und auf dem Bildschirm dargestellt.
- Es gibt eine Version auf einer grafischen Benutzeroberfläche.
- Es gibt eine Version, die das Spielen im Internet ermöglicht.

3.2 Variablen und Operatoren

Zur Speicherung von Werten werden Variablen benötigt. Operatoren dienen zur Ausführung von Berechnungen.

3.2.1 Berechnung und Zuweisung

Im folgenden Programm wird eine einfache Berechnung mithilfe eines Operators durchgeführt. Das Ergebnis der Berechnung wird mit dem Gleichheitszeichen einer Variablen zugewiesen. Es erfolgt eine Ausgabe. Diese Schritte kennen Sie bereits aus Abschnitt 2.1.5, »Variablen und Zuweisung«.

```
# Werte
a = 5
b = 3

# Berechnung
c = a + b

# Ausgabe
print("Die Aufgabe:", a, "+", b)
print("Das Ergebnis:", c)
```

Listing 3.1 Datei zuweisung.py

Die Ausgabe des Programms lautet:

```
Die Aufgabe: 5 + 3
Das Ergebnis: 8
```


In den beiden Variablen `a` und `b` wird jeweils ein Wert gespeichert. Die beiden Werte werden addiert, das Ergebnis wird der Variablen `c` zugewiesen. Die Aufgabenstellung wird ausgegeben, anschließend das Ergebnis. Der Benutzer des Programms hat noch keine Möglichkeit, in den Ablauf einzugreifen.

Verarbeitung, Ausgabe

3.2.2 Eingabe einer Zeichenkette

In diesem Abschnitt wird die Funktion `input()` zur Eingabe einer Zeichenkette durch den Benutzer eingeführt. Ein kleines Beispiel:

input()

```
# Eingabe einer Zeichenkette
print("Bitte einen Text eingeben")
x = input()
print("Ihre Eingabe:", x)
```

Listing 3.2 Datei eingabe_text.py

Die Ausgabe könnte wie folgt aussehen:

```
Bitte einen Text eingeben
Python ist toll
Ihre Eingabe: Python ist toll
```

Der Benutzer gibt einen kleinen Satz ein. Dieser Satz wird in der Variablen `x` gespeichert und anschließend ausgegeben.

3.2.3 Eingabe einer Zahl

Im weiteren Verlauf des Spiels ist es notwendig, die Eingabe des Benutzers als Zahl weiterzuverwenden. Dazu muss die Zeichenkette, die die Funktion `input()` liefert, in eine Zahl umgewandelt werden.

Umwandlung

Zur Umwandlung gibt es u. a. die folgenden Funktionen:

- Die eingebaute Funktion `int()` wandelt eine Zeichenkette in eine ganze Zahl um; eventuell vorhandene Nachkommastellen werden abgeschnitten.

int()

- Die eingebaute Funktion `float()` wandelt eine Zeichenkette in eine Zahl mit Nachkommastellen um.

float()

- Falls die Zeichenkette für `float()` keine gültige Zahl enthält, kommt es zu einem Programmabbruch. Bei `int()` muss es sich sogar um eine gültige ganze Zahl handeln. In Abschnitt 3.6, »Fehler und Ausnahmen«, lernen Sie, wie Sie Programmabbrüche abfangen.

Abbruch

Ein Beispiel:

```
# Eingabe einer Zahl
print("Bitte eine ganze Zahl eingeben")
x = input()
print("Ihre Eingabe:", x)

# Umwandlung in ganze Zahl
xganz = int(x)
print("Als ganze Zahl:", xganz)

# Mit Berechnung
xdoppel = xganz * 2
print("Das Doppelte:", xdoppel)
```

Listing 3.3 Datei eingabe_zahl.py

Die Ausgabe könnte wie folgt aussehen:

Bitte eine ganze Zahl eingeben
6
Ihre Eingabe: 6
Als ganze Zahl: 6
Das Doppelte: 12

Der Benutzer gibt eine Zeichenkette ein. Diese Zeichenkette wird mithilfe der eingebauten Funktion `int()` in eine ganze Zahl umgewandelt. Die Zahl und das Doppelte der Zahl werden ausgegeben.

3.2.4 Spiel, Version mit Eingabe

Das Spiel, in dem der Benutzer eine oder mehrere Kopfrechenaufgaben lösen soll, erhält eine Eingabe. Es wird wie folgt geändert:

```
# Werte und Berechnung
a = 5
b = 3
c = a + b
print("Die Aufgabe:", a, "+", b)

# Eingabe
print("Bitte eine Zahl eingeben:")
z = input()
```



```
# Eingabe in Zahl umwandeln
zahl = int(z)

# Ausgabe
print("Ihre Eingabe:", z)
print("Das Ergebnis:", c)
```

Listing 3.4 Datei spiel_eingabe.py

Eine mögliche Ausgabe des Programms:

Die Aufgabe: 5 + 3
Bitte eine Zahl eingeben:
9
Ihre Eingabe: 9
Das Ergebnis: 8

Das Programm gibt die Aufforderung `Bitte eine Zahl eingeben:` aus und hält an. Die Eingabe des Benutzers wird in der Variablen `z` gespeichert. Die Zeichenkette `z` wird mithilfe der Funktion `int()` in eine ganze Zahl verwandelt.

Übung u_eingabe_inch

Inch in Zentimeter

Schreiben Sie ein Programm zur Eingabe und Umrechnung von beliebigen Inch-Werten in Zentimeter. Speichern Sie das Programm in der Datei *u_eingabe_inch.py*. Rufen Sie das Programm auf, und testen Sie es. Die Ausgabe kann z. B. wie folgt aussehen:

Bitte geben Sie den Inch-Wert ein:
3.5
3.5 Inch sind 8.89 cm

Übung u_eingabe_gehalt

Steuer berechnen

Schreiben Sie ein Programm zur vereinfachten Berechnung der Steuer. Der Anwender wird dazu aufgefordert, sein monatliches Gehalt einzugeben. Anschließend werden 18 % dieses Betrags berechnet und ausgegeben. Nutzen Sie die Datei *u_eingabe_gehalt.py*. Die Ausgabe kann z. B. wie folgt aussehen:

Geben Sie Ihr Gehalt in Euro ein:
2500
Es ergibt sich eine Steuer von 450.0 Euro

3.2.5 Zufallszahlen

Natürlich wird nicht immer die gleiche Kopfrechenaufgabe gestellt. In Python steht ein Zufallszahlengenerator zur Verfügung. Er erzeugt zufällige Zahlen, die wir zur Bildung der Aufgabenstellung nutzen.

Modul importieren

Die Funktionen des Zufallszahlengenerators befinden sich in einem zusätzlichen Modul, das zunächst importiert werden muss. Das Programm wird wie folgt verändert:

```
# Modul random importieren
import random

# Zufallsgenerator initialisieren
random.seed()

# Zufallswerte und Berechnung
a = random.randint(1,10)
b = random.randint(1,10)
c = a + b
print("Die Aufgabe:", a, "+", b)

# Eingabe
print("Bitte eine Zahl eingeben:")
z = input()
zahl = int(z)

# Ausgabe
print("Ihre Eingabe:", z)
print("Das Ergebnis:", c)
```

Listing 3.5 Datei spiel_zufallszahl.py

Eine mögliche Ausgabe des Programms, abhängig von den gelieferten zufälligen Werten, sieht wie folgt aus:

Die Aufgabe: 8 + 3
Bitte eine Zahl eingeben:
7
Ihre Eingabe: 7
Das Ergebnis: 11

import

Zusätzliche Module können Sie mithilfe der Anweisung `import` in das Programm einbinden. Die Funktionen dieser Module können Sie anschließend in der Schreibweise `Modulname.Funktionsname` aufrufen.

seed()

Der Aufruf der Funktion `seed()` des Moduls `random` führt dazu, dass der Zufallszahlengenerator mit der aktuellen Systemzeit initialisiert wird. Anderenfalls könnte es passieren, dass anstelle einer zufälligen Auswahl immer wieder die gleichen Zahlen geliefert würden.

randint()

Die Funktion `randint()` des Moduls `random` liefert eine ganze Zufallszahl im angegebenen Bereich. Im vorliegenden Fall ist dies also eine zufällige Zahl von 1 bis 10.

secrets

Die Funktionen des Moduls `random` können für die zufälligen Werte eines Spiels genutzt werden. Falls Sie zufällige Werte zum Zwecke der Verschlüsselung oder aus Gründen der Sicherheit benötigen, sollten Sie die Funktionen des Moduls `secrets` nutzen, das es seit Python 3.6 gibt.

3.2.6 Typhinweise

Seit Python 3.6 können Sie angeben, welchen Datentyp eine Variable haben soll. Allerdings haben diese Typhinweise (englisch: *type hints*), auch *Annotationen* genannt, bisher nur provisorischen Charakter. Ihre Regeln können sich noch ändern und sind nicht bindend.

Type Hints sollen u. a. die Lesbarkeit des Codes für Entwickler verbessern und bessere Hilfestellungen der Editoren innerhalb einer Entwicklungsumgebung ermöglichen. Allerdings gilt nach wie vor eine Grundregel der Entwickler von Python: »Python wird eine dynamisch typisierte Sprache bleiben«.

3.3 Verzweigungen

Programmablauf

In den bisherigen Programmen werden alle Anweisungen der Reihe nach ausgeführt. Zur Steuerung des Programmablaufs werden allerdings häufig Verzweigungen benötigt. Innerhalb des Programms wird anhand einer Bedingung entschieden, welcher Zweig des Programms ausgeführt wird.

3.3.1 Vergleichsoperatoren

Kleiner, größer, gleich

Bedingungen werden mithilfe von Vergleichsoperatoren formuliert. Tabelle 3.1 listet die Vergleichsoperatoren mit ihrer Bedeutung auf.

Operator	Bedeutung
>	größer als
<	kleiner als
>=	größer als oder gleich
<=	kleiner als oder gleich
==	gleich
!=	ungleich

Tabelle 3.1 Vergleichsoperatoren

3.3.2 Einfache Verzweigung

if-else

Im folgenden Beispiel wird untersucht, ob eine Zahl positiv ist. Ist dies der Fall, wird `Diese Zahl ist positiv` ausgegeben, anderenfalls lautet die Ausgabe `Diese Zahl ist 0 oder negativ`. Es wird also nur eine der beiden Anweisungen ausgeführt.

```
x = 12
print("x:", x)

if x > 0:
    print("Diese Zahl ist positiv")
else:
    print("Diese Zahl ist 0 oder negativ")
```

Listing 3.6 Datei verzweigung_einfach.py

Die Ausgabe des Programms lautet:

```
x: 12
Diese Zahl ist positiv
```

In der ersten Zeile erhält die Variable `x` den Wert `12`.

if, Doppelpunkt

In der vierten Zeile wird mithilfe von `if` eine Verzweigung eingeleitet. Danach wird eine Bedingung formuliert (hier `x > 0`), die entweder *wahr* oder *falsch* ergibt. Anschließend kommt ein Doppelpunkt.

Einrücken

Es folgen eine oder mehrere Anweisungen, die nur ausgeführt werden, falls die Bedingung *wahr* ergibt. Die Anweisungen müssen innerhalb des sogenannten `if`-Zweigs mithilfe der ⇥-Taste eingerückt werden, damit

Python die Zugehörigkeit zur Verzweigung erkennen kann. Gleichzeitig macht die Einrückung das Programm übersichtlicher.

else, Doppelpunkt

In der sechsten Zeile wird mithilfe der Anweisung `else` der alternative Teil der Verzweigung eingeleitet. Es folgt wiederum ein Doppelpunkt.

Dann folgen eine oder mehrere Anweisungen, die nur ausgeführt werden, falls die Bedingung *falsch* ergibt. Auch diese Anweisungen müssen eingerückt werden.

3.3.3 Spiel, Version mit Bewertung der Eingabe

Eingabe prüfen

Das Spiel, in dem der Benutzer eine oder mehrere Kopfrechenaufgaben lösen soll, wird um eine Bewertung der Eingabe erweitert. Mithilfe einer einfachen Verzweigung wird untersucht, ob sie richtig oder falsch war. Das Programm verändert sich wie folgt:

```
# Zufallsgenerator
import random
random.seed()

# Werte und Berechnung
a = random.randint(1,10)
b = random.randint(1,10)
c = a + b
print("Die Aufgabe:", a, "+", b)

# Eingabe
print("Bitte eine Zahl eingeben:")
z = input()
zahl = int(z)

# Verzweigung
if zahl == c:
    print(zahl, "ist richtig")
else:
    print(zahl, "ist falsch")
    print("Ergebnis: ", c)
```

Listing 3.7 Datei spiel_verzweigung.py

Eine mögliche Ausgabe des Programms wäre:

Die Aufgabe: 2 + 8
Bitte eine Zahl eingeben:
11
11 ist falsch
Ergebnis: 10

Die Eingabe wird in eine Zahl umgewandelt. Entspricht diese dem Ergebnis der Rechnung, wird `ist richtig` ausgegeben. Entspricht sie nicht dem Ergebnis, so wird `ist falsch` ausgegeben und das korrekte Ergebnis genannt.

Übung u_verzweigung_einfach

Das vereinfachte Programm zur Berechnung der Steuer wird verändert. Der Anwender soll dazu aufgefordert werden, sein monatliches Gehalt einzugeben. Liegt es über 2.500 Euro, sind 22 % Steuern zu zahlen, ansonsten 18 %. Nutzen Sie die Datei *u_verzweigung_einfach.py*.

Es ist nur *eine* Eingabe erforderlich. Innerhalb des Programms soll anhand des Gehalts entschieden werden, welcher Steuersatz zur Anwendung kommt. Die Ausgabe kann z. B. wie folgt aussehen:

Geben Sie Ihr Gehalt in Euro ein:
3000
Es ergibt sich eine Steuer von 660.0 Euro

oder sie kann so aussehen:

Geben Sie Ihr Gehalt in Euro ein:
2000
Es ergibt sich eine Steuer von 360.0 Euro

3.3.4 Mehrfache Verzweigung

Mehrere Möglichkeiten

In vielen Anwendungsfällen gibt es mehr als zwei Möglichkeiten, zwischen denen zu entscheiden ist. Dazu wird eine mehrfache Verzweigung benötigt. Im folgenden Beispiel wird untersucht, ob eine Zahl positiv, negativ oder gleich 0 ist. Es wird eine entsprechende Meldung ausgegeben:

if-elif-else

```
x = -5
print("x:", x)
if x > 0:
    print("x ist positiv")
elif x < 0:
    print("x ist negativ")
else:
    print("x ist gleich 0")
```

Listing 3.8 Datei verzweigung_mehrfach.py

Die Ausgabe des Programms:

x: –5
x ist negativ

Die Verzweigung wird mithilfe von `if` eingeleitet. Falls `x` positiv ist, werden die nachfolgenden, eingerückten Anweisungen ausgeführt. **if**

Nach `elif` wird eine weitere Bedingung formuliert. Sie wird nur untersucht, falls die erste Bedingung (nach dem `if`) nicht zutrifft. Falls `x` negativ ist, werden die nachfolgenden, eingerückten Anweisungen ausgeführt. **elif**

Die Anweisungen nach dem `else` werden nur durchgeführt, falls keine der beiden vorherigen Bedingungen zutraf (nach dem `if` und nach dem `elif`). In diesem Fall bedeutet das, dass `x` gleich 0 ist, da es nicht positiv und nicht negativ ist. **else**

Hinweis

Innerhalb einer Verzweigung können mehrere `elif`-Anweisungen vorkommen. Sie werden der Reihe nach untersucht, bis das Programm zu einer Bedingung kommt, die zutrifft. Die weiteren `elif`-Anweisungen oder eine `else`-Anweisung werden in diesem Fall nicht mehr beachtet.

Übung u_verzweigung_mehrfach

Das Programm zur Berechnung der Steuer soll weiter verändert werden (Datei *u_verzweigung_mehrfach.py*). Der Anwender soll sein monatliches Gehalt eingeben. Anschließend wird seine Steuer nach der folgenden Tabelle berechnet:

Gehalt	Steuersatz
mehr als 4.000 Euro	26 %
2.500 bis 4.000 Euro	22 %
weniger als 2.500 Euro	18 %

3.3.5 Logische Operatoren

Logische Verknüpfung

Mithilfe der logischen Operatoren `and`, `or` und `not` können mehrere Bedingungen miteinander verknüpft werden.

and

- Eine Bedingung, die aus einer oder mehreren Einzelbedingungen besteht, die jeweils mit dem Operator `and` (= und) verknüpft sind, ergibt *wahr*, wenn *jede* der Einzelbedingungen *wahr* ergibt.

or

- Eine Bedingung, die aus einer oder mehreren Einzelbedingungen besteht, die mit dem Operator `or` (= oder) verknüpft sind, ergibt *wahr*, wenn *mindestens eine* der Einzelbedingungen *wahr* ergibt.

not

- Der Operator `not` (= nicht) kehrt den Wahrheitswert einer Bedingung um, d. h., eine falsche Bedingung wird wahr, eine wahre Bedingung wird falsch.

Der Zusammenhang wird auch in Tabelle 3.2 bis Tabelle 3.4 gezeigt.

Bedingung 1	Bedingung 2	Ergebnis
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

Tabelle 3.2 Auswirkung des logischen Operators »and«

Bedingung 1	Bedingung 2	Ergebnis
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Tabelle 3.3 Auswirkung des logischen Operators »or«

Bedingung	Ergebnis
wahr	falsch
falsch	wahr

Tabelle 3.4 Auswirkung des logischen Operators »not«

Ein Beispiel:

```
x = 12
y = 15
z = 20
print("x:", x)
print("y:", y)
print("z:", z)

# Bedingung 1
if x<y and x<z:
    print("x ist die kleinste Zahl")

# Bedingung 2
if y>x or y>z:
    print("y ist nicht die kleinste Zahl")

# Bedingung 3
if not y<x:
    print("y ist nicht kleiner als x")
```

Listing 3.9 Datei operator_logisch.py

Die Ausgabe des Programms:

```
x: 12
y: 15
z: 20
x ist die kleinste Zahl
y ist nicht die kleinste Zahl
y ist nicht kleiner als x
```

Mehrere Entscheidungen

Bedingung 1 ergibt *wahr*, wenn `x` kleiner als `y` *und* kleiner als `z` ist. Dies trifft bei den gegebenen Anfangswerten zu. Bedingung 2 ergibt *wahr*, wenn `y` größer als `x` *oder* `y` größer als `z` ist. Die erste Bedingung trifft zu, also ist die gesamte Bedingung wahr: `y` ist nicht die kleinste der drei Zahlen. Bedingung 3 ist wahr, wenn `y` *nicht* kleiner als `x` ist. Dies trifft hier zu. Zu allen Verzweigungen kann es natürlich auch einen `else`-Zweig geben.

Übung u_operator

Das Programm zur Berechnung der Steuer soll wiederum verändert werden (Datei *u_operator.py*). Die Tabelle sieht nun wie folgt aus:

Gehalt	Familienstand	Steuersatz
> 4.000 Euro	ledig	26 %
> 4.000 Euro	verheiratet	22 %
<= 4.000 Euro	ledig	22 %
<= 4.000 Euro	verheiratet	18 %

Übung u_datum

Entwickeln Sie ein Programm zur Prüfung einer Datumsangabe (Datei *u_datum.py*). Der Benutzer soll die drei Bestandteile eines Datums einzeln eingeben. Anschließend wird ermittelt, ob es sich um ein falsches oder ein richtiges Datum handelt. Gehen Sie bei der Entwicklung wie nachfolgend beschrieben vor. Testen Sie Ihr Programm nach jedem Schritt:

- Untersuchen Sie den eingegebenen Wert für den Tag. Falls er kleiner als 1 oder größer als 31 ist, handelt es sich um ein falsches Datum.
- Untersuchen Sie den eingegebenen Wert für den Monat. Falls er kleiner als 1 oder größer als 12 ist, handelt es sich um ein falsches Datum.
- Geben Sie den Wert aus, den der letzte Tag des betreffenden Monats hat. Denken Sie daran, dass es nur drei mögliche Fälle gibt: 28, 30 oder 31 Tage. Die Regeln für Schaltjahre werden noch nicht beachtet.
- Untersuchen Sie den eingegebenen Wert für den Tag. Geben Sie aus, ob er kleiner als 1 oder größer als der letzte Tag des betreffenden Monats ist.
- Untersuchen Sie den eingegebenen Wert für das Jahr. Geben Sie aus, ob es sich um ein Schaltjahr handelt. Die vereinfachte Regel für ein Schaltjahr lautet: Falls sich der Wert ohne Rest durch 4 teilen lässt, handelt es sich um ein Schaltjahr.
- Kombinieren Sie die bisherigen Schritte miteinander. Falls der Wert für den Tag kleiner als 1 oder größer als der letzte Tag des betreffenden Monats ist (mit Berücksichtigung der Regel für ein Schaltjahr), handelt es sich um ein falsches Datum, ansonsten um ein richtiges Datum.
- Erweitern Sie das Programm. Die vollständige Regel für ein Schaltjahr lautet: Falls sich der Wert ohne Rest durch 4 teilen lässt, aber nicht ohne Rest durch 100 teilen lässt, handelt es sich um ein Schaltjahr. Es handelt sich aber auch um ein Schaltjahr, falls sich der Wert ohne Rest durch 400 teilen lässt.

Ein Aufruf des fertigen Programms könnte wie folgt aussehen:

```
Tag des Datums eingeben:
29
Monat des Datums eingeben:
2
Jahr des Datums eingeben:
2000
Letzter Tag: 29
Richtiges Datum
```

3.3.6 Mehrere Vergleichsoperatoren

Kurzform

Bedingungen können auch mehrere Vergleichsoperatoren enthalten. Manche Verzweigungen sind auf diese Weise verständlicher. Ein Beispiel:

```
x = 12
y = 15
z = 20
print("x:", x)
print("y:", y)
print("z:", z)

# Bedingung 1
if x < y < z:
    print("y liegt zwischen x und z")
```

Listing 3.10 Datei operator_vergleich.py

Die Ausgabe des Programms:

```
x: 12
y: 15
z: 20
y liegt zwischen x und z
```

Die Bedingung `x < y < z` entspricht dem Ausdruck `x < y and y < z`. In der Kurzform ist sie allerdings besser lesbar.

3.3.7 Spiel, Version mit genauer Bewertung der Eingabe

Nun kann die Eingabe des Benutzers in dem Kopfrechenspiel auf verschiedene Art und Weise genauer bewertet werden:

- mithilfe einer mehrfachen Verzweigung
- anhand logischer Operatoren
- anhand von Bedingungen mit mehreren Vergleichsoperatoren

Das Programm wird wie folgt verändert:

```
# Zufallsgenerator
import random
random.seed()

# Werte und Berechnung
a = random.randint(1,10)
b = random.randint(1,10)
c = a + b
print("Die Aufgabe:", a, "+", b)

# Eingabe
print("Bitte eine Zahl eingeben:")
z = input()
zahl = int(z)

# Mehrfache Verzweigung, logische Operatoren
# Bedingungen mit mehreren Vergleichsoperatoren
if zahl == c:
    print(zahl, "ist richtig")
elif zahl < 0 or zahl > 100:
    print(zahl, "ist ganz falsch")
elif c-1 <= zahl <= c+1:
    print(zahl, "ist ganz nahe dran")
else:
    print(zahl, "ist falsch")

# Ende
print("Ergebnis: ", c)
```

Listing 3.11 Datei spiel_operator.py

Eine mögliche Ausgabe des Programms wäre:

```
Die Aufgabe: 2 + 1
Bitte eine Zahl eingeben:
4
4 ist ganz nahe dran
Ergebnis:  3
```

or

Insgesamt werden vier Möglichkeiten angeboten: über `if`, zweimal `elif` und `else`. Ist die Eingabe kleiner als 0 *oder* größer als 100, so ist sie ganz falsch. Dies wird mithilfe des logischen Operators `or` gelöst.

Mehrere Vergleichsoperatoren

Unterscheidet sich die eingegebene Zahl vom richtigen Ergebnis nur um den Wert 1, ist die Eingabe ganz nahe dran. Dies wird mit einer Bedingung ermittelt, die mehrere Vergleichsoperatoren enthält.

3.3.8 Rangfolge der Operatoren

In vielen Ausdrücken treten mehrere Operatoren auf. Bisher sind dies Rechenoperatoren, Vergleichsoperatoren und logische Operatoren. Die einzelnen Teilschritte folgen der sogenannten Rangfolge der Operatoren. Die Teilschritte, bei denen höherrangige Operatoren beteiligt sind, werden zuerst ausgeführt.

Höchster Rang oben

Tabelle 3.5 gibt die Rangfolge der bisher verwendeten Operatoren in Python an, beginnend mit den Operatoren, die den höchsten Rang haben. Gleichrangige Operatoren stehen jeweils in einer Zeile. Teilschritte, in denen mehrere Operatoren gleichen Ranges stehen, werden von links nach rechts ausgeführt.

Operator	Bedeutung
`+ -`	positives Vorzeichen einer Zahl, negatives Vorzeichen einer Zahl
`* / % //`	Multiplikation, Division, Modulo, Ganzzahldivision
`+ -`	Addition, Subtraktion
`< <= > >= == !=`	kleiner, kleiner oder gleich, größer, größer oder gleich, gleich, ungleich
`not`	logische Verneinung

Tabelle 3.5 Rangfolge der bisher genutzten Operatoren

Operator	Bedeutung
`and`	logisches Und
`or`	logisches Oder

Tabelle 3.5 Rangfolge der bisher genutzten Operatoren (Forts.)

3.4 Schleifen

Neben der Verzweigung gibt es eine weitere wichtige Struktur zur Steuerung von Programmen: die Schleife. Mithilfe einer Schleife ermöglichen Sie die wiederholte Ausführung eines Programmschritts.

Wiederholung

Es muss zwischen zwei Typen von Schleifen unterschieden werden: der `for`-Schleife und der `while`-Schleife. Der jeweilige Anwendungsbereich der beiden Typen wird durch folgende Merkmale definiert:

for

- Eine `for`-Schleife wird verwendet, wenn ein Programmschritt für eine regelmäßige, zum Zeitpunkt der Anwendung bekannte Folge von Werten wiederholt ausgeführt werden soll.

while

- Eine `while`-Schleife wird verwendet, wenn sich erst durch Eingaben des Anwenders ergibt, ob ein Programmschritt ausgeführt werden soll und wie oft er wiederholt wird.

Eine `for`-Schleife wird auch als *Zählschleife* bezeichnet, eine `while`-Schleife als *bedingungsgesteuerte Schleife*.

3.4.1 for-Schleife

Die Anwendung der `for`-Schleife verdeutlicht das folgende Beispiel:

```
for i in 2, 7.5, -22:
    print("Zahl:", i, ", Quadrat:", i*i)
```

Listing 3.12 Datei schleife_for.py

Folgende Ausgabe wird erzeugt:

```
Zahl: 2 , Quadrat: 4
Zahl: 7.5 , Quadrat: 56.25
Zahl: –22 , Quadrat: 484
```


Abfolge von Zahlen

Die erste Zeile ist wie folgt zu lesen: Führe die nachfolgenden eingerückten Anweisungen für jede Zahl in der Abfolge `2`, `7.5` und `-22` aus. Nenne diese Zahl in diesen Anweisungen `i`. Nach der Zahlenfolge muss, ebenso wie bei `if-else`, ein Doppelpunkt notiert werden. Anstelle von `i` können Sie auch eine andere Variable als Schleifenvariable nutzen. Die Zahlen werden zusammen mit einem kurzen Informationstext ausgegeben bzw. quadriert und ausgegeben.

> **Hinweis**
>
> Eine solche Abfolge von Objekten, die z. B. in einer `for`-Schleife durchlaufen werden kann, nennt man auch *iterierbares Objekt* oder kurz *Iterable*. Diese Iterables werden uns in Python noch häufig begegnen. Den Vorgang des Durchlaufens nennt man auch *Iterieren*.

3.4.2 Schleifenabbruch mit »break«

break

Die Anweisung `break` bietet eine weitere Möglichkeit zur Steuerung von Schleifen. Sie führt zu einem unmittelbaren Abbruch einer Schleife. Sie wird sinnvollerweise mit einer Bedingung verbunden und häufig bei Sonderfällen eingesetzt. Ein Beispiel:

```
for i in 12, -4, 20, 7:
    if i*i > 200:
        break
    print("Zahl:", i, ", Quadrat:", i*i)

print("Ende")
```

Listing 3.13 Datei schleife_break.py

Diese Ausgabe wird erzeugt:

```
Zahl: 12 , Quadrat: 144
Zahl: –4 , Quadrat: 16
Ende
```

Schleife verlassen

Die Schleife wird unmittelbar verlassen, wenn das Quadrat der aktuellen Zahl größer als 200 ist. Die Ausgabe innerhalb der Schleife erfolgt auch nicht mehr. Das Programm wird nach der Schleife fortgesetzt.

3.4.3 Geschachtelte Kontrollstrukturen

Wie das vorherige Programm verdeutlicht, können Kontrollstrukturen (also Verzweigungen und Schleifen) auch geschachtelt werden. Dies bedeutet, dass eine Kontrollstruktur eine weitere Kontrollstruktur enthält. Diese kann ihrerseits wiederum eine Kontrollstruktur enthalten usw. Ein weiteres Beispiel:

```
for x in -2, -1, 0, 1, 2:
    if x > 0:
        print(x, "positiv")
    else:
        if x < 0:
            print(x, "negativ")
        else:
            print(x, "gleich 0")
```

Listing 3.14 Datei schachtelung.py

Es wird diese Ausgabe erzeugt:

```
–2 negativ
–1 negativ
0 gleich 0
1 positiv
2 positiv
```

Zur Erläuterung:

Äußeres for
- Die äußerste Kontrollstruktur ist eine `for`-Schleife. Alle einfach eingerückten Anweisungen werden – gemäß der Schleifensteuerung – mehrmals ausgeführt.

Äußeres if
- Mit der äußeren `if`-Anweisung wird die erste Verzweigung eingeleitet. Ist `x` größer als 0, wird die folgende zweifach eingerückte Anweisung ausgeführt. Ist `x` nicht größer als 0, wird die innere `if`-Anweisung hinter der äußeren `else`-Anweisung untersucht.

Inneres if
- Trifft die Bedingung der inneren `if`-Anweisung zu, werden die folgenden dreifach eingerückten Anweisungen ausgeführt. Trifft sie nicht zu, werden die dreifach eingerückten Anweisungen ausgeführt, die der inneren `else`-Anweisung folgen.

Mehrfach einrücken

Zu beachten sind besonders die mehrfachen Einrückungen, damit die Tiefe der Kontrollstruktur von Python richtig erkannt werden kann.

Nach einem Doppelpunkt hinter dem Kopf einer Kontrollstruktur wird in der Entwicklungsumgebung IDLE automatisch mit einem Tabulatorsprung eingerückt. Dieser Sprung erzeugt standardmäßig vier Leerzeichen, dadurch werden die Kontrollstrukturen für den Entwickler klar erkennbar.

Mindestens ein Leerzeichen

Falls Sie mit einem anderen Editor arbeiten, müssen Sie darauf achten, dass um mindestens ein Leerzeichen eingerückt wird, damit Python die Kontrollstruktur erkennt. Sinnvoller ist eine Einrückung um zwei oder mehr Leerzeichen, damit die Struktur auch für den Entwickler gut erkennbar ist.

3.4.4 Spiel, Version mit for-Schleife und Abbruch

Vier Versuche

Die `for`-Schleife wird nun dazu genutzt, die Eingabe und die Bewertung des Kopfrechenspiels insgesamt viermal zu durchlaufen. Der Benutzer hat somit vier Versuche, das richtige Ergebnis zu ermitteln. Die Anweisung `break` dient zum Abbruch der Schleife, sobald der Benutzer das richtige Ergebnis eingegeben hat.

```
# Zufallsgenerator
import random
random.seed()

# Werte und Berechnung
a = random.randint(1,10)
b = random.randint(1,10)
c = a + b
print("Die Aufgabe:", a, "+", b)

# Schleife mit for
for i in 1, 2, 3, 4:
    # Eingabe
    print("Bitte eine Zahl eingeben:")
    z = input()
    zahl = int(z)

    # Verzweigung
    if zahl == c:
        print(zahl, "ist richtig")
        # Abbruch der Schleife
        break
    else:
```

```
        print(zahl, "ist falsch")

# Ende
print("Ergebnis: ", c)
```

Listing 3.15 Datei spiel_for.py

Folgende Ausgabe wird erzeugt:

```
Die Aufgabe: 7 + 9
Bitte eine Zahl eingeben:
12
12 ist falsch
Bitte eine Zahl eingeben:
16
16 ist richtig
Ergebnis:  16
```

Maximal viermal

Die Aufgabe wird einmal ermittelt und gestellt. Der Benutzer wird maximal viermal dazu aufgefordert, ein Ergebnis einzugeben. Jede seiner Eingaben wird bewertet. Ist bereits einer der ersten drei Versuche richtig, wird die Schleife vorzeitig abgebrochen.

3.4.5 for-Schleife mit »range()«

range()

Meist werden Schleifen für regelmäßige Abfolgen von Zahlen genutzt. Dabei erweist sich der Einsatz der Funktion `range()` als sehr nützlich. Ein Beispiel:

```
for i in range(3,11,2):
    print("Zahl:", i, "Quadrat:", i*i)
```

Listing 3.16 Datei range_drei.py

Es wird diese Ausgabe erzeugt:

```
Zahl: 3 Quadrat: 9
Zahl: 5 Quadrat: 25
Zahl: 7 Quadrat: 49
Zahl: 9 Quadrat: 81
```

Ganze Zahlen

Der englische Begriff *range* bedeutet »Bereich«. Innerhalb der Klammern hinter `range` können bis zu drei ganze Zahlen, durch Kommata getrennt, eingetragen werden:

- Die erste ganze Zahl (hier 3) gibt den Beginn des Bereichs an, für den die folgenden Anweisungen ausgeführt werden. **Beginn**
- Die zweite ganze Zahl (hier 11) kennzeichnet das Ende des Bereichs. Es ist die erste Zahl, für die die Anweisungen *nicht* mehr ausgeführt werden. **Ende**
- Die dritte ganze Zahl (hier 2) gibt die Schrittweite für die Schleife an. Die Zahlen, für die die Anweisungen ausgeführt werden, stehen zueinander also jeweils im Abstand von +2. **Schrittweite**

Der Aufruf der Funktion `range()` mit den Zahlen 3, 11 und 2 ergibt somit die Abfolge: 3, 5, 7, 9.

Wird die Funktion `range()` nur mit zwei Zahlen aufgerufen, so wird eine Schrittweite von 1 angenommen. Ein Beispiel: **Schrittweite 1**

```
for i in range(5,9):
    print("Zahl:", i)
```

Listing 3.17 Datei range_zwei.py

Es wird folgende Ausgabe erzeugt:

```
Zahl: 5
Zahl: 6
Zahl: 7
Zahl: 8
```

Wird die Funktion `range()` sogar nur mit einer Zahl aufgerufen, so wird diese Zahl einfach als die obere Grenze angesehen. Als untere Grenze gilt 0. Außerdem wird wiederum eine Schrittweite von 1 angenommen. Ein Beispiel: **Beginn bei 0**

```
for i in range(3):
    print("Zahl:", i)
```

Listing 3.18 Datei range_eins.py

Die Ausgabe:

```
Zahl: 0
Zahl: 1
Zahl: 2
```

Sinnvolle Kombinationen

Hinweis

Bei der Funktion `range()` können eine oder mehrere der drei Zahlen auch negativ sein. Achten Sie auf sinnvolle Zahlenkombinationen. Die Angabe `range(3,-11,2)` ist nicht sinnvoll, da man von der Zahl +3 in Schritten von +2 nicht zur Zahl –11 gelangt. Python fängt solche Schleifen ab und lässt sie nicht ausführen. Dasselbe gilt für die Zahlenkombination `range(3,11,-2)`.

Keine ganze Zahl

Für den regelmäßigen Ablauf von Zahlen mit Nachkommastellen muss die Schleifenvariable passend umgerechnet werden. Ein Beispiel:

```
# 1. Version
for x in range(18,22):
    print(x/10)
print()

# 2. Version
x = 1.8
for i in range(4):
    print(x)
    x = x + 0.1
```

Listing 3.19 Datei range_nachkomma.py

Die Ausgabe lautet:

```
1.8
1.9
2.0
2.1

1.8
1.9000000000000001
2.0
2.1
```

Es werden jeweils die Zahlen von 1,8 bis 2,1 in Schritten von 0,1 erzeugt.

- In der ersten Version werden die ganzen Zahlen von 18 bis 21 erzeugt und anschließend durch 10 geteilt.
- In der zweiten Version wird mit dem ersten Wert begonnen und innerhalb der Schleife jeweils um 0,1 erhöht. Dabei müssen Sie vorher errechnen, wie häufig die Schleife durchlaufen werden muss.

Übung u_range

Schleife erkennen

Ermitteln Sie durch Überlegen (nicht durch einen einfachen Aufruf) die Ausgabe des folgenden Programms (Datei *u_range.py*).

```
print("Schleife 1")
for i in 2, 3, 6.5, -7:
    print(i)

print("Schleife 2")
for i in range(2,11,3):
    print(i)

print("Schleife 3")
for i in range(-3,14,4):
    print(i)

print("Schleife 4")
for i in range(3,-11,-3):
    print(i)
```

Übung u_range_inch

Schreiben Sie ein Programm, das die folgende Ausgabe erzeugt (Datei *u_range_inch.py*).

15 Inch = 38.1 cm
20 Inch = 50.8 cm
25 Inch = 63.5 cm
30 Inch = 76.2 cm
35 Inch = 88.9 cm
40 Inch = 101.6 cm

Es handelt sich um eine regelmäßige Liste von Inch-Werten, für die der jeweilige Zentimeter-Wert durch Umrechnung mit dem Faktor 2,54 ermittelt wird. Es ist keine Eingabe durch den Anwender notwendig.

3.4.6 Spiel, Version mit »range()«

Mit Zähler

Im Kopfrechenspiel wird die Schleife zur Wiederholung der Eingabe nun mithilfe von `range()` gebildet. Gleichzeitig haben Sie damit einen Zähler, der die laufende Nummer des Versuchs enthält. Sie können ihn verwenden, um dem Benutzer die Anzahl der Versuche mitzuteilen.

```
# Zufallsgenerator
import random
random.seed()

# Werte und Berechnung
a = random.randint(1,10)
b = random.randint(1,10)
c = a + b
print("Die Aufgabe:", a, "+", b)

# Schleife mit range
for versuch in range(1,10):
    # Eingabe
    print("Bitte eine Zahl eingeben:")
    z = input()
    zahl = int(z)

    # Verzweigung
    if zahl == c:
        print(zahl, "ist richtig")
        # Abbruch der Schleife
        break
    else:
        print(zahl, "ist falsch")

# Anzahl ausgeben
print("Ergebnis: ", c)
print("Anzahl Versuche:", versuch)
```

Listing 3.20 Datei spiel_range.py

Die Ausgabe lautet:

```
Die Aufgabe: 10 + 5
Bitte eine Zahl eingeben:
13
13 ist falsch
Bitte eine Zahl eingeben:
15
15 ist richtig
```



```
Ergebnis: 15
Anzahl Versuche: 2
```

Der Benutzer hat maximal neun Versuche: `range(1,10)`. Die Variable `versuch` dient als Zähler für die Versuche. Nach Eingabe der richtigen Lösung (oder nach vollständigem Durchlauf der Schleife) wird dem Benutzer die Anzahl der Versuche mitgeteilt.

Zähler

3.4.7 while-Schleife

Bedingte Schleife

Die `while`-Schleife dient zur Steuerung einer Wiederholung mithilfe einer Bedingung. Im folgenden Programm werden zufällige Zahlen addiert und ausgegeben. Solange die Summe der Zahlen kleiner als 30 ist, wird der Vorgang wiederholt. Ist die Summe gleich oder größer als 30, wird das Programm beendet.

```
# Zufallsgenerator
import random
random.seed()

# Initialisierung
summe = 0

# while-Schleife
while summe < 30:
    zzahl = random.randint(1,8)
    summe = summe + zzahl
    print("Zahl:", zzahl, "Zwischensumme:", summe)

print("Ende")
```

Listing 3.21 Datei schleife_while.py

Eine mögliche Ausgabe des Programms sähe wie folgt aus:

```
Zahl: 3 Zwischensumme: 3
Zahl: 8 Zwischensumme: 11
Zahl: 5 Zwischensumme: 16
Zahl: 8 Zwischensumme: 24
Zahl: 7 Zwischensumme: 31
Ende
```

Zunächst wird die Variable für die Summe der Zahlen auf 0 gesetzt.

Einrücken

Die `while`-Anweisung leitet die Schleife ein. Die wörtliche Übersetzung der Zeile lautet: *solange die Summe kleiner als 30 ist*. Dies bezieht sich auf die nachfolgenden eingerückten Anweisungen.

Doppelpunkt

Nach dem Wort `while` folgt (wie bei einer `if`-Anweisung) eine Bedingung, die mithilfe von Vergleichsoperatoren erstellt wird. Auch hier dürfen Sie den Doppelpunkt am Ende der Zeile nicht vergessen, ähnlich wie bei `if-else` und `for`.

Summierung

Es wird eine zufällige Zahl ermittelt und zur bisherigen Summe addiert. Die neue Summe errechnet sich also aus der alten Summe plus der eingegebenen Zahl. Die neue Summe wird ausgegeben.

Ende der Schleife

Die Anweisung zur Ausgabe des Texts »Ende« wird erst erreicht, wenn die Summe den Wert `30` erreicht oder überschritten hat.

3.4.8 Spiel, Version mit while-Schleife und Zähler

Solange Eingabe falsch

Die `while`-Schleife wird nun auch im Kopfrechenspiel zur Wiederholung der Eingabe genutzt. Der Benutzer hat beliebig viele Versuche, die Aufgabe zu lösen. Die Variable `versuch`, die als Zähler für die Versuche dient, muss separat gesteuert werden. Sie ist nicht mehr automatisch eine Schleifenvariable.

```
# Zufallsgenerator
import random
random.seed()

# Werte und Berechnung
a = random.randint(1,10)
b = random.randint(1,10)
c = a + b
print("Die Aufgabe:", a, "+", b)

# Schleife initialisieren
zahl = c + 1

# Anzahl initialisieren
versuch = 0

# Schleife mit while
while zahl != c:
    # Anzahl Versuche
    versuch = versuch + 1

    # Eingabe mit Umwandlung
    print("Bitte eine Zahl eingeben:")
    z = input()
    zahl = int(z)

    # Verzweigung
    if zahl == c:
        print(zahl, "ist richtig")
    else:
        print(zahl, "ist falsch")

# Anzahl ausgeben
print("Ergebnis: ", c)
print("Anzahl Versuche:", versuch)
```

Listing 3.22 Datei spiel_while.py

Die Ausgabe hat sich nicht geändert.

Anzahl der Versuche

Der Benutzer hat beliebig viele Versuche. Die `while`-Schleife läuft, solange die richtige Lösung nicht ermittelt wird. Die Variable `zahl` wird mit einem Wert vorbesetzt, der dafür sorgt, dass die `while`-Schleife mindestens einmal läuft. Die Variable `versuch` wird mit 0 vorbesetzt und dient als laufende Nummer. Nach Eingabe der richtigen Lösung wird dem Benutzer die Anzahl der Versuche mitgeteilt.

Übung u_while

Ende mit 0

Schreiben Sie ein Programm (Datei *u_while.py*), das den Anwender wiederholt dazu auffordert, einen Wert in Inch einzugeben. Der eingegebene Wert soll anschließend in Zentimeter umgerechnet und ausgegeben werden. Das Programm soll nach der Eingabe des Werts 0 beendet werden.

Bei einer `while`-Schleife wird immer angegeben, gemäß welcher Bedingung wiederholt werden soll, und nicht, gemäß welcher Bedingung beendet werden soll. Daher müssen Sie in diesem Programm formulieren: *Solange die Eingabe ungleich 0 ist*.

3.4.9 Kombinierte Zuweisungsausdrücke

walrus-operator

Mit Python 3.8 wird der Operator `:=` für kombinierte Zuweisungen eingeführt. Er führt auch den Spitznamen *walrus-operator*, aufgrund der Ähnlichkeit mit den Augen und den Zähnen eines Walrosses. Er ermöglicht kürzere Ausdrücke, in denen allerdings eine höhere Komplexität steckt. Nachfolgend sehen Sie, wie Sie mithilfe des Operators das Beispiel aus Abschnitt 3.4.7 verkürzen können:

```
import random
random.seed()
summe = 0
while (summe := summe + random.randint(1,8)) < 30:
    print("Zwischensumme:", summe)
print("Ende")
```

Listing 3.23 Datei schleife_zuweisung.py

Mehrere Abläufe

Die Zeile, die mit dem Schlüsselwort `while` beginnt, beinhaltet mehrere Abläufe. Zunächst wird die Summe um einen zufällig ermittelten Wert erhöht. Anschließend wird die neue Summe mit der Zahl 30 verglichen. Dieser Vergleich steuert die Schleife. Aufgrund des niedrigen Vorrangs des Operators `:=` muss die Zuweisung in runden Klammern stehen.

Die Ausgabe sieht, in Abhängigkeit von den zufällig ermittelten Zahlen, wie vorher aus:

```
Zwischensumme: 1
Zwischensumme: 9
Zwischensumme: 11
Zwischensumme: 17
Zwischensumme: 23
Zwischensumme: 29
Ende
```

3.5 Entwicklung eines Programms

Teile eines Programms

Bei der Entwicklung Ihrer eigenen Programme sollten Sie Schritt für Schritt vorgehen. Stellen Sie zuerst einige Überlegungen dazu an, wie das gesamte Programm aufgebaut sein sollte, und zwar auf Papier. Aus welchen Teilen sollte es nacheinander bestehen? Versuchen Sie anschließend nicht, das gesamte Programm mit all seinen komplexen Bestandteilen auf einmal zu

schreiben! Dies ist der größte Fehler, den Einsteiger (und manchmal auch Fortgeschrittene) machen können.

Einfache Version

Schreiben Sie zunächst eine einfache Version des ersten Programmteils. Anschließend testen Sie sie. Erst nach einem erfolgreichen Test fügen Sie den folgenden Programmteil hinzu. Nach jeder Änderung testen Sie wiederum. Sollte sich ein Fehler zeigen, wissen Sie, dass er aufgrund der letzten Änderung aufgetreten ist. Nach dem letzten Hinzufügen haben Sie eine einfache Version Ihres gesamten Programms erstellt.

Komplexe Version

Nun ändern Sie einen Teil Ihres Programms in eine komplexere Version ab. Auf diese Weise machen Sie Ihr Programm Schritt für Schritt komplexer, bis Sie schließlich das gesamte Programm so erstellt haben, wie es Ihren anfänglichen Überlegungen auf Papier entspricht.

Änderungen

Manchmal ergibt sich während der praktischen Programmierung noch die eine oder andere Änderung gegenüber Ihrem Entwurf. Das ist kein Problem, solange sich nicht der gesamte Aufbau ändert. Sollte dies allerdings der Fall sein, kehren Sie noch einmal kurz zum Papier zurück und überdenken den Aufbau. Das bedeutet nicht, dass Sie die bisherigen Programmzeilen löschen müssen, sondern möglicherweise nur ein wenig ändern und anders anordnen.

Einzelne Schritte

Schreiben Sie Ihre Programme übersichtlich. Falls Sie gerade überlegen, wie Sie drei, vier bestimmte Schritte Ihres Programms auf einmal machen können: Machen Sie daraus einfach einzelne Anweisungen, die der Reihe nach ausgeführt werden. Dies vereinfacht eine eventuelle Fehlersuche. Wenn Sie (oder eine andere Person) Ihr Programm später einmal ändern oder erweitern möchten, gelingt der Einstieg in den Aufbau des Programms wesentlich schneller.

Kontrolle

Sie können die Funktion `print()` zur Kontrolle von Werten und zur Suche von logischen Fehlern einsetzen. Zusätzlich können Sie einzelne Zeilen Ihres Programms als Kommentar kennzeichnen, um festzustellen, welcher Teil des Programms fehlerfrei läuft und welcher Teil demnach fehlerbehaftet ist.

3.6 Fehler und Ausnahmen

Fehler

Macht der Anwender nach einer Eingabeaufforderung eine falsche Eingabe (gibt er z. B. keine Zahl, sondern einen Text ein), wird das Programm mit einer Fehlermeldung beendet. Bisher gehe ich vereinfacht davon aus, dass

der Anwender korrekte Eingaben vornimmt. In diesem Abschnitt beschreibe ich, wie Sie Fehler vermeiden oder abfangen.

3.6.1 Basisprogramm

Abbruch möglich

Durch das Abfangen von falschen Eingaben wird die Benutzung eines Programms für den Anwender deutlich komfortabler. Im folgenden Programm soll eine ganze Zahl eingegeben werden. Da der Benutzer dieses Programm durch die Eingabe von Text oder Sonderzeichen zum Abbruch bringen kann, wird es immer weiter verbessert.

```
# Eingabe
print("Bitte geben Sie eine ganze Zahl ein")
z = input()

# Umwandlung
zahl = int(z)

# Ausgabe
print("Sie haben die ganze Zahl", zahl, "richtig eingegeben")
print("Ende des Programms")
```

Listing 3.24 Datei fehler_basis.py

Richtige Eingabe

Gibt der Anwender eine Zahl (z. B. 12) ein, läuft das Programm fehlerfrei bis zum Ende und erzeugt die folgende Ausgabe:

```
Bitte geben Sie eine ganze Zahl ein
12
Sie haben die ganze Zahl 12 richtig eingegeben
Ende des Programms
```

Falsche Eingabe

Macht der Anwender jedoch eine falsche Eingabe (z. B. 3a), bricht das Programm vorzeitig ab und erzeugt die folgende Ausgabe:

```
Bitte geben Sie eine ganze Zahl ein
3a
Traceback (most recent call last):
  File "C:\Python38\Beispiele\fehler_basis.py", line 6, in <module>
    zahl = int(z)
ValueError: invalid literal for int() with base 10: '3a'
```


Fehlerstelle

Diese Informationen weisen auf die Stelle im Programm hin, an der der Fehler bemerkt wird (Datei *fehler_basis.py*, Zeile 6). Außerdem wird die Art des Fehlers mitgeteilt (`ValueError`).

3.6.2 Fehler abfangen

Verbessern

Zunächst soll eine Fehleingabe abgefangen werden, um einen Abbruch des Programms zu vermeiden. Zu diesem Zweck müssen Sie die Stelle herausfinden, an der eine Fehleingabe auftreten kann. Hier müssen Sie das Programm verbessern.

Fehler abfangen

Das Programm soll zukünftig an dieser Stelle alle Arten von Fehlern abfangen, die Python automatisch erkennen kann. Dies erreichen Sie in einem ersten Schritt durch die folgende Änderung:

```
# Eingabe
print("Bitte geben Sie eine ganze Zahl ein")
z = input()

# Versuch der Umwandlung
try:
    zahl = int(z)
    print("Sie haben die ganze Zahl", zahl, "richtig eingegeben")

# Fehler bei Umwandlung
except:
    print("Sie haben die ganze Zahl nicht richtig eingegeben")

print("Ende des Programms")
```

Listing 3.25 Datei fehler_abfangen.py

Kein Abbruch mehr

Wenn der Anwender eine richtige ganze Zahl eingibt, läuft das Programm wie bisher. Bei einer falschen Eingabe erscheint nun eine entsprechende Meldung. Das Programm bricht jedoch nicht ab, sondern läuft bis zum Ende.

```
Bitte geben Sie eine ganze Zahl ein
3a
Sie haben die ganze Zahl nicht richtig eingegeben
Ende des Programms
```

try Die Anweisung `try` leitet eine Ausnahmebehandlung ein. Ähnlich wie bei der `if`-Anweisung gibt es verschiedene Zweige, die das Programm durchlaufen kann. Das Programm versucht (englisch: *try*), die Anweisungen durchzuführen, die eingerückt nach `try` stehen. Falls die Eingabe erfolgreich ist, wird der `except`-Zweig nicht ausgeführt, ähnlich wie beim `else`-Zweig der `if`-Anweisung.

except Ist die Eingabe dagegen nicht erfolgreich und handelt es sich um einen Fehler, wird der Fehler oder die Ausnahme (englisch: *exception*) mit der Anweisung `except` abgefangen. In diesem Fall werden alle eingerückten Anweisungen im `except`-Zweig durchgeführt. Das Programm läuft ohne Abbruch zu Ende, da der Fehler zwar auftritt, aber abgefangen wird.

Nach `try` und `except` muss jeweils ein Doppelpunkt gesetzt werden, wie bei `if-else`, `for` oder `while`.

Hinweise

Kritische Zeile Nur die *kritische Zeile* wird in die Ausnahmebehandlung eingebettet. Sie sollten sich also Gedanken darüber machen, welche Stellen Ihres Programms fehlerträchtig sind. Eine Eingabeaufforderung ist solch eine kritische Stelle. Andere Fehlermöglichkeiten sind z. B. die Bearbeitung einer Datei (die möglicherweise nicht existiert) oder die Ausgabe an einen Drucker (der vielleicht nicht eingeschaltet ist).

3.6.3 Eingabe wiederholen

Erneute Eingabe In einem zweiten Schritt wird dafür gesorgt, dass der Anwender nach einer falschen Eingabe eine erneute Eingabe machen kann. Der gesamte Eingabevorgang mit Ausnahmebehandlung wird so lange wiederholt, bis die Eingabe erfolgreich war. Betrachten Sie das folgende Programm:

```
# Initialisierung der while-Schleife
fehler = 1

# Schleife bei falscher Eingabe
while fehler == 1:
    # Eingabe
    print("Bitte geben Sie eine ganze Zahl ein")
    z = input()
    # Versuch der Umwandlung
    try:
        zahl = int(z)
        print("Sie haben die ganze Zahl", zahl,
              "richtig eingegeben")
        fehler = 0
    # Fehler bei Umwandlung
    except:
        print("Sie haben die ganze Zahl nicht richtig eingegeben")

print("Ende des Programms")
```

Listing 3.26 Datei fehler_eingabe_neu.py

Hinweis

Beachten Sie die doppelte Einrückung: einmal nach `while` und noch einmal nach `try-except`.

Nachfolgend wird eine mögliche Eingabe gezeigt – zunächst mit einem Fehler, anschließend fehlerfrei:

```
Bitte geben Sie eine ganze Zahl ein
3a
Sie haben die ganze Zahl nicht richtig eingegeben
Bitte geben Sie eine ganze Zahl ein
12
Sie haben die ganze Zahl 12 richtig eingegeben
Ende des Programms
```

Eingabe wiederholen Die Variable `fehler` ist notwendig, um die Eingabe gegebenenfalls wiederholen zu können. Sie wird zunächst auf den Wert `1` gesetzt. Es wird eine `while`-Schleife formuliert, in der der Eingabevorgang mit der Ausnahmebehandlung eingebettet ist. Die Schleife wird wiederholt, solange die Variable `fehler` den Wert `1` hat.

Eingabe erfolgreich? Ist die Eingabe erfolgreich, wird die Variable `fehler` auf `0` gesetzt. Das führt dazu, dass die Schleife beendet wird und das Programm regulär fortfahren kann. Ist die Eingabe nicht erfolgreich, hat `fehler` nach wie vor den Wert `1`: Der Eingabevorgang wird wiederholt.

Übung u_fehler

Verbessern Sie das Programm zur Eingabe und Umrechnung eines beliebigen Inch-Werts in Zentimeter. Eingabefehler des Anwenders sollen abgefangen werden. Das Programm soll den Anwender so lange zur Eingabe auffordern, bis sie erfolgreich war (Datei *u_fehler.py*).

3.6.4 Exkurs: Schleifenfortsetzung mit »continue«

break

An dieser Stelle möchte ich auf ein Thema zurückkommen, das bereits in Abschnitt 3.4, »Schleifen«, behandelt wurde. Sie kennen aus diesem Abschnitt die Anweisung `break`, die zum unmittelbaren Abbruch einer Schleife führt.

continue

Die Anweisung `continue` dient zum unmittelbaren Abbruch des aktuellen Durchlaufs einer Schleife. Die Schleife wird anschließend mit dem nächsten Durchlauf fortgesetzt. Betrachten Sie hierzu das folgende Programm:

```
for i in range (1,7):
    print("Zahl:", i)
    if 3 <= i <= 5:
        continue
    print("Quadrat:", i*i)
```

Listing 3.27 Datei schleife_continue.py

Die Ausgabe dieses Programms ist:

```
Zahl: 1
Quadrat: 1
Zahl: 2
Quadrat: 4
Zahl: 3
Zahl: 4
Zahl: 5
Zahl: 6
Quadrat: 36
```

Rest übergehen

Die Schleife durchläuft alle Zahlen von 1 bis 6. Alle diese Zahlen werden auch ausgegeben. Liegt die aktuelle Zahl zwischen 3 und 5, wird der Rest der Schleife übergangen und unmittelbar der nächste Schleifendurchlauf begonnen. Anderenfalls wird das Quadrat der Zahl ausgegeben.

3.6.5 Spiel, Version mit Ausnahmebehandlung

Fehler abfangen

Die Ausnahmebehandlung und die Anweisung `continue` werden nun auch im Kopfrechenspiel eingesetzt. Damit kann ein Eingabefehler abgefangen und das Programm regulär fortgesetzt werden.

```
# Zufallsgenerator
import random
random.seed()

# Werte und Berechnung
a = random.randint(1,10)
b = random.randint(1,10)
c = a + b
print("Die Aufgabe:", a, "+", b)

# Schleife und Anzahl initialisieren
zahl = c + 1
versuch = 0

# Schleife mit while
while zahl != c:
    # Anzahl Versuche
    versuch = versuch + 1

    # Eingabe
    print("Bitte eine ganze Zahl eingeben:")
    z = input()

    # Versuch einer Umwandlung
    try:
        zahl = int(z)
    except:
        # Falls Umwandlung nicht erfolgreich
        print("Sie haben keine ganze Zahl eingegeben")
        # Schleife unmittelbar fortsetzen
        continue

    # Verzweigung
    if zahl == c:
        print(zahl, "ist richtig")
```

```
    else:
        print(zahl, "ist falsch")

# Anzahl Versuche
print("Ergebnis: ", c)
print("Anzahl Versuche:", versuch)
```

Listing 3.28 Datei spiel_ausnahme.py

Es wird die folgende Ausgabe erzeugt:

```
Die Aufgabe: 8 + 3
Bitte eine ganze Zahl eingeben:
12
12 ist falsch
Bitte eine ganze Zahl eingeben:
11a
Sie haben keine ganze Zahl eingegeben
Bitte eine ganze Zahl eingeben:
11
11 ist richtig
Ergebnis:  11
Anzahl Versuche: 3
```

Falsche Eingabe

Die Umwandlung der Eingabe steht in einem `try-except`-Block. Falls die Umwandlung aufgrund einer falschen Eingabe nicht gelingt, erscheint eine entsprechende Meldung. Der Rest der Schleife wird übergangen, und die nächste Eingabe wird unmittelbar angefordert.

3.7 Funktionen und Module

Modularisierung

Die Modularisierung, also die Zerlegung eines Programms in selbst geschriebene Funktionen, bietet besonders bei größeren Programmen unübersehbare Vorteile:

Mehrfach verwenden

- Programmteile, die mehrmals benötigt werden, müssen nur einmal definiert werden.
- Nützliche Programmteile können in mehreren Programmen verwendet werden.

Übersichtlicher

- Umfangreiche Programme können in übersichtliche Teile zerlegt werden.

- Pflege und Wartung von Programmen werden erleichtert.
- Der Programmcode ist für den Programmierer selbst (zu einem späteren Zeitpunkt) und für andere Programmierer leichter zu verstehen.

Verständlicher

Vordefinierte Funktionen

Neben den selbst geschriebenen Funktionen gibt es in Python, wie in jeder anderen Programmiersprache auch, zahlreiche vordefinierte Funktionen, die dem Entwickler viel Arbeit abnehmen können. Diese Funktionen sind entweder fest eingebaut oder über die Einbindung spezieller Module verfügbar.

Aufgabe einer Funktion

Als Beispiel für eine fest eingebaute Funktion wird bereits `input()` eingesetzt. Jede Funktion hat eine spezielle Aufgabe. So hält bspw. die Funktion `input()` das Programm an und nimmt eine Eingabe entgegen.

Rückgabewert

Wie viele (aber nicht alle) Funktionen hat `input()` einen sogenannten Rückgabewert, liefert also ein Ergebnis an die Stelle des Programms zurück, von der sie aufgerufen wird: die eingegebene Zeichenkette.

Es folgen zwei Hinweise für fortgeschrittene Leser, die bereits mit einer anderen Programmiersprache gearbeitet haben:

Nicht überladen

- Funktionen können in Python nicht überladen werden. Falls Sie eine Funktion mehrfach definieren, gegebenenfalls mit unterschiedlichen Parametern, so gilt nur die jeweils letzte Definition.

Typhinweise

- Seit Python 3.5 können Sie mithilfe eines Typhinweises angeben, welchen Datentyp der Rückgabewert einer Funktion haben soll. Beachten Sie dazu bitte Abschnitt 3.2.6.

3.7.1 Einfache Funktionen

Immer gleich

Einfache Funktionen führen bei Aufruf stets die gleiche Aktion aus. Im folgenden Beispiel führt jeder Aufruf der Funktion `stern()` dazu, dass eine optische Trennung auf dem Bildschirm ausgegeben wird:

```
# Definition der Funktion
def stern():
    print("-----------------")
    print("*** Trennung ****")
    print("-----------------")

# Hauptprogramm
x = 12
y = 5
```

```
stern()                         # 1. Aufruf
print("x =", x, ", y =", y)
stern()                         # 2. Aufruf
print("x + y =", x + y)
stern()                         # 3. Aufruf
print("x - y =", x - y)
stern()                         # 4. Aufruf
```

Listing 3.29 Datei funktion_einfach.py

Die Ausgabe sehen Sie in Abbildung 3.1.

```
-----------------
*** Trennung ****
-----------------
x = 12 , y = 5
-----------------
*** Trennung ****
-----------------
x + y = 17
-----------------
*** Trennung ****
-----------------
x - y = 7
-----------------
*** Trennung ****
-----------------
```

Abbildung 3.1 Einfache Funktionen

Definition mit def

Im oberen Teil des Programms wird die Funktion `stern()` definiert. Nach der Anweisung `def` folgt der Name der Funktion (hier `stern`), anschließend folgen runde Klammern und der bereits bekannte Doppelpunkt. Innerhalb der Klammern könnten Werte an die Funktion übergeben werden. Dazu mehr in Abschnitt 3.7.2, »Funktionen mit einem Parameter«, und in Abschnitt 3.7.3, »Funktionen mit mehreren Parametern«. Die nachfolgenden eingerückten Anweisungen werden jedes Mal durchgeführt, wenn die Funktion aufgerufen wird.

Aufruf

Eine Funktion wird zunächst nur definiert und nicht durchgeführt. Sie steht sozusagen zum späteren Gebrauch bereit. Im unteren Teil der Datei beginnt das eigentliche Programm. Es werden einige Rechenoperationen mit zwei Variablen durchgeführt. Mithilfe der Funktion `stern()` werden die Ausgabezeilen optisch voneinander getrennt. Sie wird insgesamt viermal aufgerufen. Nach Bearbeitung der Funktion fährt das Programm jedes Mal mit der Anweisung fort, die dem Aufruf der Funktion folgt.

Klammern

Eine Funktion wird aufgerufen, indem Sie ihren Namen, gefolgt von runden Klammern, notieren. Falls Sie Informationen an die Funktion übergeben möchten, notieren Sie sie innerhalb der runden Klammern.

Name

Den Namen einer Funktion können Sie weitgehend frei wählen – es gelten die gleichen Regeln wie bei den Namen von Variablen, siehe auch Abschnitt 2.1.5, »Variablen und Zuweisung«: Der Name kann aus den Buchstaben a bis z, A bis Z, aus Ziffern und dem Zeichen _ (Unterstrich) bestehen. Er darf nicht mit einer Ziffer beginnen und keinem reservierten Wort in Python entsprechen.

3.7.2 Funktionen mit einem Parameter

Parameter

Bei einem Aufruf können auch Informationen an Funktionen übermittelt werden, sogenannte *Parameter*. Diese Informationen können innerhalb der Funktion ausgewertet werden und führen gegebenenfalls bei jedem Aufruf zu unterschiedlichen Ergebnissen. Ein Beispiel:

```
# Definition der Funktion
def quadrat(x):
    q = x*x
    print("Zahl:", x, "Quadrat:", q)

# Hauptprogramm
quadrat(4.5)
a = 3
quadrat(a)
quadrat(2*a)
```

Listing 3.30 Datei parameter.py

Die Ausgabe lautet:

```
Zahl: 4.5 Quadrat: 20.25
Zahl: 3 Quadrat: 9
Zahl: 6 Quadrat: 36
```

Wertübermittlung

Die Definition der Funktion `quadrat()` enthält eine Variable innerhalb der Klammern. Beim Aufruf wird ein Wert an die Funktion übermittelt und dieser Variablen zugewiesen. Hier sind das die folgenden Werte:

- Die Zahl 4,5 – die Variable `x` erhält in der Funktion den Wert `4.5`.

- Die Variable `a` – die Variable `x` erhält in der Funktion den aktuellen Wert von `a`, nämlich 3.
- Das Ergebnis einer Berechnung – die Variable `x` erhält in der Funktion den aktuellen Wert von `2 * a`, also 6.

Anzahl der Werte

Hinweis

Die Funktion erwartet genau einen Wert. Sie darf also nicht ohne einen Wert oder mit mehr als einem Wert aufgerufen werden, sonst bricht das Programm mit einer Fehlermeldung ab.

Übung u_parameter

Es soll wiederum die Steuer für verschiedene Gehälter berechnet werden (Datei *u_parameter.py*). Liegt das Gehalt über 2.500 Euro, sind 22 % Steuern zu zahlen, ansonsten 18 %. Die Berechnung und die Ausgabe der Steuer sollen diesmal innerhalb einer Funktion mit dem Namen `steuer()` stattfinden. Die Funktion soll für die folgenden Gehälter aufgerufen werden: 1.800 Euro, 2.200 Euro, 2.500 Euro, 2.900 Euro.

3.7.3 Funktionen mit mehreren Parametern

Anzahl, Reihenfolge

Eine Funktion kann noch vielseitiger werden, wenn Sie ihr mehrere Parameter übermitteln. Dabei ist auf die übereinstimmende Anzahl und die richtige Reihenfolge der Parameter zu achten. Ein Beispiel:

```
# Definition der Funktion
def berechnung(x,y,z):
    ergebnis = (x+y) * z
    print("Ergebnis:", ergebnis)

# Hauptprogramm
berechnung(2,3,5)
berechnung(5,2,3)
```

Listing 3.31 Datei parameter_mehrere.py

Die Ausgabe lautet:

Ergebnis: 25
Ergebnis: 21

Reihenfolge

Es werden genau drei Parameter erwartet, bei beiden Aufrufen werden auch drei Werte übermittelt. Wie Sie am Ergebnis erkennen, ist die Reihenfolge der Parameter wichtig.

- Beim ersten Aufruf erhält `x` den Wert `2`, `y` den Wert `3` und `z` den Wert `5`. Dies ergibt die Rechnung: (2 + 3) * 5 = 25.
- Beim zweiten Aufruf werden dieselben Zahlen übergeben, aber in anderer Reihenfolge. Es ergibt sich die Rechnung: (5 + 2) * 3 = 21.

3.7.4 Funktionen mit Rückgabewert

Ergebnis

Funktionen werden häufig zur Berechnung von Ergebnissen eingesetzt. Zu diesem Zweck können Funktionen ihre Ergebnisse als sogenannte *Rückgabewerte* zurückliefern.

Rückgabewert

Im Unterschied zu vielen anderen Programmiersprachen können Funktionen in Python mehr als einen Rückgabewert liefern, siehe Abschnitt 5.7.4, »Mehrere Rückgabewerte«. In diesem Abschnitt werden aber zunächst nur Funktionen betrachtet werden, die genau einen Rückgabewert zur Verfügung stellen.

Im folgenden Beispiel wird eine Funktion, die einen Rückgabewert liefert, auf verschiedene Arten vom Hauptprogramm aus aufgerufen.

```
# Definition der Funktion
def mittelwert(x,y):
    ergebnis = (x+y) / 2
    return ergebnis

# Hauptprogramm
c = mittelwert(3, 9)
print("Mittelwert:", c)
x = 5
print("Mittelwert:", mittelwert(x,4))
```

Listing 3.32 Datei rueckgabewert.py

Diese Ausgabe wird erzeugt:

Mittelwert: 6.0
Mittelwert: 4.5

return Innerhalb der Funktion wird zunächst das Ergebnis berechnet. Es wird anschließend mithilfe der Anweisung `return` an die aufrufende Stelle zurückgeliefert. Die Anweisung `return` beendet außerdem unmittelbar den Ablauf der Funktion.

Rückgabe speichern Beim ersten Aufruf wird der Rückgabewert in der Variablen `c` zwischengespeichert. Er kann im weiteren Verlauf des Programms an beliebiger Stelle verwendet werden.

Rückgabe ausgeben Beim zweiten Aufruf geschehen zwei Dinge gleichzeitig: Die Funktion `mittelwert()` wird aufgerufen und liefert ein Ergebnis. Außerdem wird dieses Ergebnis unmittelbar ausgegeben.

3.7.5 Spiel, Version mit Funktionen

Mit Funktionen Das Programm mit dem Kopfrechenspiel umfasst nun zwei Funktionen; die erste Funktion dient zur Ermittlung der Aufgabe, die zweite zur Bewertung der Eingabe:

```
# Aufgabe
def aufgabe():
    a = random.randint(1,10)
    b = random.randint(1,10)
    erg = a + b
    print("Die Aufgabe:", a, "+", b)
    return erg

# Kommentar
def kommentar(eingabezahl, ergebnis):
    if eingabezahl == ergebnis:
        print(eingabezahl, "ist richtig")
    else:
        print(eingabezahl, "ist falsch")

# Zufallsgenerator
import random
random.seed()

# Aufgabe
c = aufgabe()
```



```
# Schleife und Anzahl initialisieren
zahl = c + 1
versuch = 0

# Schleife mit while
while zahl != c:
    # Anzahl Versuche
    versuch = versuch + 1

    # Eingabe
    print("Bitte eine Zahl eingeben:")
    z = input()

    # Versuch einer Umwandlung
    try:
        zahl = int(z)
    except:
        # Falls Umwandlung nicht erfolgreich
        print("Sie haben keine Zahl eingegeben")
        # Schleife unmittelbar fortsetzen
        continue

    # Kommentar
    kommentar(zahl,c)

# Anzahl Versuche
print("Ergebnis: ", c)
print("Anzahl Versuche:", versuch)
```

Listing 3.33 Datei spiel_funktion.py

Die Ausgabe hat sich nicht geändert.

aufgabe() In der Funktion `aufgabe()` werden die beiden Zufallszahlen ermittelt, und die Aufgabe wird auf dem Bildschirm ausgegeben. Außerdem wird das Ergebnis der Aufgabe als Rückgabewert an das Hauptprogramm zurückgeliefert.

kommentar() Der Funktion `kommentar()` werden zwei Zahlen als Parameter übermittelt: die Lösung des Anwenders und das richtige Ergebnis. Innerhalb der Funktion wird die eingegebene Lösung untersucht, und ein entsprechender Kommentar wird ausgegeben. Die Funktion hat keinen Rückgabewert.

Übung u_rueckgabewert

Schreiben Sie das Programm aus Übung *u_parameter* um. Die Steuer soll innerhalb der Funktion `steuer()` berechnet und an das Hauptprogramm zurückgeliefert werden. Die Ausgabe des Werts soll im Hauptprogramm stattfinden (Datei *u_rueckgabewert.py*).

3.8 Das fertige Spiel

Erweiterungen

Zum Abschluss des Programmierkurses erweitern wir das Kopfrechenspiel noch etwas – dabei nutzen wir die Programmiermittel, die Ihnen inzwischen zur Verfügung stehen. Die Erweiterungen:

- Es werden bis zu zehn Aufgaben nacheinander gestellt. Der Benutzer kann dabei die Anzahl selbst bestimmen.
- Zusätzlich zur Addition kommen die weiteren Grundrechenarten zum Einsatz: Subtraktion, Multiplikation und Division.
- Die Bereiche, aus denen die zufälligen Zahlen gewählt werden, hängen von der Rechenart ab. Bei der Multiplikation wird z. B. mit kleineren Zahlen gerechnet als bei der Addition.
- Der Benutzer hat maximal drei Versuche pro Aufgabe.
- Die Anzahl der richtig gelösten Aufgaben wird ermittelt.

Die einzelnen Abschnitte des Programms sind nummeriert. Diese Nummern finden sich in der Erläuterung wieder. In späteren Kapiteln kommen weitere Ergänzungen hinzu. Es folgt das Programm:

```
# 1: Zufallsgenerator
import random
random.seed()

# 2: Anzahl Aufgaben
anzahl = -1
while anzahl<0 or anzahl>10:
    try:
        print("Wie viele Aufgaben (1 bis 10):")
        anzahl = int(input())
    except:
        continue
```



```
# 3: Anzahl richtige Ergebnisse
richtig = 0

# 4: Schleife mit "anzahl" Aufgaben
for aufgabe in range(1,anzahl+1):

    # 5: Operatorauswahl
    opzahl = random.randint(1,4)

    # 6: Operandenauswahl
    if opzahl == 1:
        a = random.randint(-10,30)
        b = random.randint(-10,30)
        op = "+"
        c = a + b
    elif opzahl == 2:
        a = random.randint(1,30)
        b = random.randint(1,30)
        op = "-"
        c = a - b
    elif opzahl == 3:
        a = random.randint(1,10)
        b = random.randint(1,10)
        op = "*"
        c = a * b

    # 7: Sonderfall Division
    elif opzahl == 4:
        c = random.randint(1,10)
        b = random.randint(1,10)
        op = "/"
        a = c * b

    # 8: Aufgabenstellung
    print("Aufgabe", aufgabe, "von",
          anzahl, ":", a, op, b)

    # 9: Schleife mit 3 Versuchen
    for versuch in range(1,4):
```

```
            # 10: Eingabe
            try:
                print("Bitte eine Zahl eingeben:")
                zahl = int(input())
            except:
                # Falls Umwandlung nicht erfolgreich
                print("Sie haben keine Zahl eingegeben")
                # Schleife unmittelbar fortsetzen
                continue

            # 11: Kommentar
            if zahl == c:
                print(zahl, "ist richtig")
                richtig = richtig + 1
                break
            else:
                print(zahl, "ist falsch")

        # 12: Richtiges Ergebnis der Aufgabe
        print("Ergebnis: ", c)

# 13: Anzahl richtige Ergebnisse
print("Richtig:", richtig, "von", anzahl)
```

Listing 3.34 Datei spiel_fertig.py

Es wird die folgende Ausgabe erzeugt:

```
Wie viele Aufgaben (1 bis 10):
2
Aufgabe 1 von 2 : 26 + 18
Bitte eine Zahl eingeben:
44
44 ist richtig
Ergebnis:  44
Aufgabe 2 von 2 : 27 – 2
Bitte eine Zahl eingeben:
24
24 ist falsch
Bitte eine Zahl eingeben:
23
```



```
23 ist falsch
Bitte eine Zahl eingeben:
22
22 ist falsch
Ergebnis:  25
Richtig: 1 von 2
```

Fehler abfangen

Nach der Initialisierung des Zufallsgenerators (1) wird die gewünschte Anzahl der Aufgaben eingelesen (2). Da der Benutzer einen Fehler bei der Eingabe machen könnte, findet eine Ausnahmebehandlung statt.

Eingabe, Umwandlung

Der Rückgabewert der Funktion `input()` wird unmittelbar als Parameter der Funktion `int()` genutzt. So werden zwei Schritte auf einmal erledigt.

Der Zähler für die Anzahl der richtig gelösten Aufgaben (`richtig`) wird auf `0` gestellt (3). Es wird eine äußere `for`-Schleife mit der gewünschten Anzahl gestartet (4).

Zufälliger Operator

Der Operator wird per Zufallsgenerator ermittelt (5). Für jeden Operator gibt es andere Bereiche, aus denen die Zahlen ausgewählt werden (6). Der Operator selbst und das Ergebnis werden gespeichert.

Nur ganze Zahlen

Eine Besonderheit ist bei der Division zu beachten (7): Es sollen nur ganze Zahlen vorkommen. Die beiden zufälligen Operanden (`a` und `b`) werden daher aus dem Ergebnis einer Multiplikation ermittelt.

Die Aufgabe wird gestellt (8). Dabei werden zur besseren Orientierung des Benutzers auch die laufende Nummer und die Gesamtanzahl der Aufgaben ausgegeben. Es wird eine innere `for`-Schleife für maximal drei Versuche gestartet (9).

Maximal drei Versuche

Die Eingaben des Benutzers (10) werden kommentiert (11). Nach maximal drei Versuchen wird das richtige Ergebnis ausgegeben (12). Zuletzt wird die Anzahl der richtig gelösten Aufgaben ausgegeben (13).

Auf einen Blick

1 Einführung 17

2 Erste Schritte 23

3 Programmierkurs 35

4 Datentypen 85

5 Weiterführende Programmierung 143

6 Objektorientierte Programmierung 209

7 Verschiedene Module 233

8 Dateien 267

9 Internet 311

10 Datenbanken 345

11 Benutzeroberflächen 375

12 Unterschiede in Python 2 453

13 Raspberry Pi 457

Inhalt

Materialien zum Buch 16

1 Einführung 17

1.1 Vorteile von Python 17
1.2 Verbreitung von Python 18
1.3 Aufbau des Buchs 18
1.4 Übungen 20
1.5 Installation von Python unter Windows 20
1.6 Installation von Python unter Ubuntu Linux 21
1.7 Installation von Python unter macOS 21

2 Erste Schritte 23

2.1 Python als Taschenrechner 23
2.1.1 Eingabe von Berechnungen 23
2.1.2 Addition, Subtraktion und Multiplikation 24
2.1.3 Division, Ganzzahldivision und Modulo 24
2.1.4 Rangfolge und Klammern 25
2.1.5 Variablen und Zuweisung 26
2.2 Erstes Programm 28
2.2.1 Hallo Welt 28
2.2.2 Eingabe eines Programms 28
2.3 Speichern und Ausführen 29
2.3.1 Speichern 29
2.3.2 Ausführen unter Windows 29
2.3.3 Ausführen unter Ubuntu Linux und unter macOS 31
2.3.4 Kommentare 32
2.3.5 Verkettung von Ausgaben 33
2.3.6 Lange Ausgaben 33

3 Programmierkurs 35

3.1 Ein Spiel programmieren ... 35

3.2 Variablen und Operatoren ... 36
- 3.2.1 Berechnung und Zuweisung ... 36
- 3.2.2 Eingabe einer Zeichenkette ... 37
- 3.2.3 Eingabe einer Zahl ... 37
- 3.2.4 Spiel, Version mit Eingabe ... 38
- 3.2.5 Zufallszahlen ... 40
- 3.2.6 Typhinweise ... 41

3.3 Verzweigungen ... 41
- 3.3.1 Vergleichsoperatoren ... 41
- 3.3.2 Einfache Verzweigung ... 42
- 3.3.3 Spiel, Version mit Bewertung der Eingabe ... 43
- 3.3.4 Mehrfache Verzweigung ... 44
- 3.3.5 Logische Operatoren ... 46
- 3.3.6 Mehrere Vergleichsoperatoren ... 49
- 3.3.7 Spiel, Version mit genauer Bewertung der Eingabe ... 50
- 3.3.8 Rangfolge der Operatoren ... 51

3.4 Schleifen ... 52
- 3.4.1 for-Schleife ... 52
- 3.4.2 Schleifenabbruch mit »break« ... 53
- 3.4.3 Geschachtelte Kontrollstrukturen ... 54
- 3.4.4 Spiel, Version mit for-Schleife und Abbruch ... 55
- 3.4.5 for-Schleife mit »range()« ... 56
- 3.4.6 Spiel, Version mit »range()« ... 59
- 3.4.7 while-Schleife ... 61
- 3.4.8 Spiel, Version mit while-Schleife und Zähler ... 62
- 3.4.9 Kombinierte Zuweisungsausdrücke ... 64

3.5 Entwicklung eines Programms ... 64

3.6 Fehler und Ausnahmen ... 65
- 3.6.1 Basisprogramm ... 66
- 3.6.2 Fehler abfangen ... 67
- 3.6.3 Eingabe wiederholen ... 68
- 3.6.4 Exkurs: Schleifenfortsetzung mit »continue« ... 70
- 3.6.5 Spiel, Version mit Ausnahmebehandlung ... 71

3.7 Funktionen und Module ... 72
- 3.7.1 Einfache Funktionen ... 73
- 3.7.2 Funktionen mit einem Parameter ... 75
- 3.7.3 Funktionen mit mehreren Parametern ... 76
- 3.7.4 Funktionen mit Rückgabewert ... 77
- 3.7.5 Spiel, Version mit Funktionen ... 78

3.8 Das fertige Spiel ... 80

4 Datentypen 85

4.1 Zahlen ... 85
- 4.1.1 Ganze Zahlen ... 85
- 4.1.2 Zahlen mit Nachkommastellen ... 87
- 4.1.3 Typ ermitteln ... 88
- 4.1.4 Operator ** ... 88
- 4.1.5 Rundung und Konvertierung ... 89
- 4.1.6 Winkelfunktionen ... 91
- 4.1.7 Weitere mathematische Funktionen ... 91
- 4.1.8 Bitoperatoren ... 94
- 4.1.9 Brüche ... 96

4.2 Zeichenketten ... 99
- 4.2.1 Eigenschaften ... 99
- 4.2.2 Operatoren ... 101
- 4.2.3 Operationen ... 102
- 4.2.4 Funktionen ... 104
- 4.2.5 Umwandlung einer Zeichenkette in eine Zahl ... 108
- 4.2.6 Umwandlung einer Zahl in eine Zeichenkette ... 110
- 4.2.7 Datentyp »bytes« ... 110

4.3 Listen ... 111
- 4.3.1 Eigenschaften ... 112
- 4.3.2 Operatoren ... 114
- 4.3.3 Funktionen und Operationen ... 114

4.4 Tupel ... 118
- 4.4.1 Eigenschaften ... 118
- 4.4.2 Operationen ... 118
- 4.4.3 Tupel entpacken ... 120

4.5 Dictionarys ... 122
4.5.1 Eigenschaften ... 122
4.5.2 Operatoren und Funktionen ... 124
4.5.3 Views ... 125
4.5.4 Vergleiche ... 127

4.6 Mengen, Sets ... 128
4.6.1 Eigenschaften ... 128
4.6.2 Funktionen ... 129
4.6.3 Operatoren ... 130
4.6.4 Frozenset ... 132

4.7 Wahrheitswerte und Nichts ... 133
4.7.1 Wahrheitswerte True und False ... 133
4.7.2 Nichts, None ... 137

4.8 Referenz, Identität und Kopie ... 138
4.8.1 Referenz und Identität ... 138
4.8.2 Ressourcen sparen ... 140
4.8.3 Objekte kopieren ... 142

5 Weiterführende Programmierung 143

5.1 Allgemeines ... 143
5.1.1 Kombinierte Zuweisungsoperatoren ... 143
5.1.2 Programmzeile in mehreren Zeilen ... 145
5.1.3 Eingabe mit Hilfestellung ... 146
5.1.4 Anweisung »pass« ... 147
5.1.5 Funktionen »eval()« und »exec()« ... 149

5.2 Ausgabe und Formatierung ... 150
5.2.1 Funktion »print()« ... 150
5.2.2 Formatierung mit String-Literalen ... 152
5.2.3 Formatierung mit »format()« ... 156
5.2.4 Formatierung wie in C ... 157

5.3 Conditional Expression ... 159

5.4 Iterierbare Objekte ... 160
5.4.1 Funktion »zip()« ... 160
5.4.2 Funktion »map()« ... 161
5.4.3 Funktion »filter()« ... 163

5.5 List Comprehension ... 164

5.6 Fehler und Ausnahmen ... 166
5.6.1 Allgemeines ... 166
5.6.2 Syntaxfehler ... 166
5.6.3 Laufzeitfehler ... 168
5.6.4 Logische Fehler und Debugging ... 169
5.6.5 Fehler erzeugen ... 173
5.6.6 Unterscheidung von Ausnahmen ... 175

5.7 Funktionen ... 176
5.7.1 Variable Anzahl von Parametern ... 177
5.7.2 Benannte Parameter ... 178
5.7.3 Parameter mit Vorgabewerten ... 179
5.7.4 Mehrere Rückgabewerte ... 180
5.7.5 Übergabe von Kopien und Referenzen ... 181
5.7.6 Lokal, global ... 184
5.7.7 Rekursive Funktionen ... 186
5.7.8 Lambda-Funktion ... 187
5.7.9 Funktionsname als Parameter ... 187

5.8 Eingebaute Funktionen ... 189
5.8.1 Funktionen »max()«, »min()« und »sum()« ... 191
5.8.2 Funktionen »chr()« und »ord()« ... 191
5.8.3 Funktionen »reversed()« und »sorted()« ... 193

5.9 Statistikfunktionen ... 194

5.10 Eigene Module ... 197
5.10.1 Eigene Module erzeugen ... 197
5.10.2 Standard-Import eines Moduls ... 198
5.10.3 Import eines Moduls mit Umbenennung ... 198
5.10.4 Import von Funktionen ... 198

5.11 Parameter der Kommandozeile ... 199
5.11.1 Übergabe von Zeichenketten ... 200
5.11.2 Übergabe von Zahlen ... 200
5.11.3 Beliebige Anzahl von Parametern ... 201

5.12 Programm »Bruchtraining« ... 201
5.12.1 Der Ablauf des Programms ... 202
5.12.2 Hauptprogramm ... 203
5.12.3 Eine leichte Aufgabe ... 204

5.12.4 Eine mittelschwere Aufgabe ... 205
5.12.5 Eine schwere Aufgabe ... 207

6 Objektorientierte Programmierung 209

6.1 Was ist OOP? ... 209
6.2 Klassen, Objekte und eigene Methoden ... 210
6.3 Konstruktor und Destruktor ... 212
6.4 Besondere Methoden ... 214
6.5 Operatormethoden ... 216
6.6 Referenz, Identität und Kopie ... 217
6.7 Vererbung ... 220
6.8 Mehrfachvererbung ... 223
6.9 Datenklassen ... 225
6.10 Enumerationen ... 227
6.11 Spiel, objektorientierte Version ... 229

7 Verschiedene Module 233

7.1 Datum und Zeit ... 233
7.1.1 Ausgabe der Zeit mit »localtime()« ... 233
7.1.2 Ausgabe der Zeit mit »strftime()« ... 235
7.1.3 Zeitangabe erzeugen ... 237
7.1.4 Mit Zeitangaben rechnen ... 238
7.1.5 Programm anhalten ... 240
7.1.6 Spiel, Version mit Zeitmessung ... 242
7.1.7 Spiel, objektorientierte Version mit Zeitmessung ... 243
7.2 Warteschlangen ... 244
7.2.1 Klasse SimpleQueue ... 245
7.2.2 Klasse LifoQueue ... 246
7.2.3 Klasse PriorityQueue ... 247
7.2.4 Klasse deque ... 247

7.3 Multithreading ... 251
7.3.1 Wozu dient Multithreading? ... 251
7.3.2 Erzeugung eines Threads ... 252
7.3.3 Identifizierung eines Threads ... 253
7.3.4 Gemeinsame Daten und Objekte ... 254
7.3.5 Threads und Exceptions ... 256
7.4 Reguläre Ausdrücke ... 257
7.4.1 Suchen von Teiltexten ... 258
7.4.2 Ersetzen von Teiltexten ... 262
7.5 Audioausgabe ... 265

8 Dateien 267

8.1 Dateitypen ... 267
8.2 Öffnen und Schließen einer Datei ... 268
8.3 Sequenzielle Dateien ... 269
8.3.1 Sequenzielles Schreiben ... 269
8.3.2 Sequenzielles Lesen ... 271
8.3.3 CSV-Datei schreiben ... 276
8.3.4 CSV-Datei lesen ... 278
8.4 Dateien mit festgelegter Struktur ... 280
8.4.1 Formatiertes Schreiben ... 281
8.4.2 Lesen an beliebiger Stelle ... 282
8.4.3 Schreiben an beliebiger Stelle ... 284
8.5 Serialisierung ... 285
8.5.1 Objekte in Datei schreiben ... 286
8.5.2 Objekte aus Datei lesen ... 287
8.6 Bearbeitung mehrerer Dateien ... 289
8.6.1 Funktion »glob.glob()« ... 289
8.6.2 Funktion »os.scandir()« ... 291
8.7 Informationen über Dateien ... 292
8.8 Dateien und Verzeichnisse verwalten ... 293
8.9 Beispielprojekt Morsezeichen ... 294
8.9.1 Morsezeichen aus Datei lesen ... 295

8.9.2 Ausgabe auf dem Bildschirm ... 296
8.9.3 Ausgabe mit Tonsignalen ... 297
8.10 Spiel, Version mit Highscore-Datei ... 300
8.10.1 Eingabebeispiel ... 300
8.10.2 Aufbau des Programms ... 301
8.10.3 Code des Programms ... 301
8.11 Spiel, objektorientierte Version mit Highscore-Datei ... 306

9 Internet 311

9.1 Laden und Senden von Internetdaten ... 311
9.1.1 Daten lesen ... 312
9.1.2 Daten kopieren ... 314
9.1.3 Daten senden per »GET« ... 315
9.1.4 Daten senden per »POST« ... 318
9.2 Webserver-Programmierung ... 320
9.2.1 Erstes Programm ... 321
9.2.2 Beantworten einer Benutzereingabe ... 322
9.2.3 Formularelemente mit mehreren Werten ... 325
9.2.4 Typen von Formularelementen ... 327
9.3 Browser aufrufen ... 333
9.4 Spiel, Version für das Internet ... 334
9.4.1 Eingabebeispiel ... 334
9.4.2 Aufbau des Programms ... 336
9.4.3 Code des Programms ... 337

10 Datenbanken 345

10.1 Aufbau von Datenbanken ... 345
10.2 SQLite ... 346
10.2.1 Datenbank, Tabelle und Datensätze ... 347
10.2.2 Daten anzeigen ... 349
10.2.3 Daten auswählen, Operatoren ... 350
10.2.4 Operator »LIKE« ... 353

10.2.5 Sortierung der Ausgabe ... 355
10.2.6 Auswahl nach Eingabe ... 356
10.2.7 Datensätze ändern ... 357
10.2.8 Datensätze löschen ... 360
10.3 SQLite auf dem Webserver ... 361
10.4 MySQL ... 363
10.4.1 XAMPP und Connector/Python ... 364
10.4.2 Datenbank erzeugen ... 364
10.4.3 Tabelle anlegen ... 366
10.4.4 Datensätze anlegen ... 367
10.4.5 Daten anzeigen ... 369
10.5 Spiel, Version mit Highscore-Datenbank ... 370
10.6 Spiel, objektorientierte Version mit Highscore-Datenbank ... 373

11 Benutzeroberflächen 375

11.1 Einführung ... 375
11.1.1 Eine erste GUI-Anwendung ... 376
11.1.2 Ändern von Eigenschaften ... 378
11.2 Widget-Typen ... 379
11.2.1 Anzeigefeld, Label ... 379
11.2.2 Eigenschaften von Bildern ... 382
11.2.3 Einzeilige Textbox, Entry ... 385
11.2.4 Versteckte Eingabe ... 387
11.2.5 Mehrzeilige Textbox, Text ... 388
11.2.6 Scrollende Textbox, ScrolledText ... 390
11.2.7 Listbox mit einfacher Auswahl ... 392
11.2.8 Listbox mit mehrfacher Auswahl ... 394
11.2.9 Spinbox ... 395
11.2.10 Scrollbar, scrollende Widgets ... 398
11.2.11 Radiobuttons zur Auswahl, Widget-Variablen ... 400
11.2.12 Radiobuttons zur Auswahl und Ausführung ... 402
11.2.13 Checkbuttons zur mehrfachen Auswahl ... 403
11.2.14 Schieberegler, Scale ... 406
11.2.15 Mausereignisse ... 408
11.2.16 Tastaturereignisse ... 411

11.3 Geometrische Anordnung von Widgets ... 413
11.3.1 Frame-Widget, Methode »pack()« ... 414
11.3.2 Ein einfacher Taschenrechner ... 416
11.3.3 Methode »grid()« ... 420
11.3.4 Methode »place()«, absolute Koordinaten ... 422
11.3.5 Methode »place()«, relative Koordinaten ... 424
11.3.6 Absolute Veränderung von Koordinaten ... 426
11.3.7 Relative Veränderung von Koordinaten ... 427
11.4 Menüs, Messageboxen und Dialogfelder ... 431
11.4.1 Menüleisten ... 432
11.4.2 Kontextmenüs ... 437
11.4.3 Messageboxen ... 439
11.4.4 Eigene Dialogfelder ... 444
11.4.5 Ausführung verhindern ... 446
11.5 Spiel, GUI-Version ... 447

12 Unterschiede in Python 2 453

12.1 Neue und geänderte Eigenschaften ... 453
12.1.1 Auffällige Änderungen ... 453
12.1.2 Weitere Änderungen ... 454
12.2 Konvertierung von Python 2 zu Python 3 ... 455

13 Raspberry Pi 457

13.1 Einzelteile und Installation ... 457
13.1.1 Einzelteile ... 457
13.1.2 Weitere Bausätze ... 459
13.1.3 Sicherheit und Schäden ... 460
13.1.4 Zusammenbau ... 460
13.1.5 Erster Start ... 461
13.1.6 Raspberry Desktop ... 462
13.1.7 Terminal ... 462

13.2 Elektronische Schaltungen ... 463
13.2.1 Gleichspannungs-Stromkreis ... 463
13.2.2 Spannung ist Information ... 464
13.2.3 Bauelemente und Ausrüstung ... 465
13.2.4 Widerstände ... 465
13.2.5 Aufbau des GPIO-Anschlusses ... 467
13.3 Lüftersteuerung ... 468
13.3.1 Temperatur ermitteln ... 468
13.3.2 Leuchtdioden ... 469
13.3.3 Leuchtdiode ansteuern ... 470
13.3.4 Leuchtdiode blinken lassen ... 471
13.3.5 Mehrere Leuchtdioden ... 471
13.3.6 Lüfter ansteuern ... 472
13.3.7 Temperaturabhängige Lüftersteuerung ... 474
13.3.8 Temperatur in Datenbank speichern ... 475
13.4 Roboter AlphaBot2-Pi ... 477
13.4.1 Demo-Programme ... 477
13.4.2 SSH-Verbindung zur Befehlseingabe ... 478
13.4.3 SSH-Server auf Raspberry Pi ... 478
13.4.4 PuTTY als SSH-Client ... 479
13.4.5 Montage des Roboters ... 481
13.4.6 Erstes Programm ... 482
13.4.7 Alle Richtungen ... 484
13.4.8 Dateien übertragen mit PSCP ... 485
13.4.9 Steuerung per Tastatur ... 486
13.4.10 Umfahren von Hindernissen ... 488

Anhang 491

A.1 Erstellen von EXE-Dateien ... 491
A.2 Installation von XAMPP ... 492
A.3 UNIX-Befehle ... 494
A.4 Lösungen ... 497

Index ... 505