Kapitel 4
Der Weg zum ersten Programm

Nachdem wir im interaktiven Modus spielerisch einige Grundelemente der Sprache Python behandelt haben, möchten wir dieses Wissen jetzt auf ein tatsächliches Programm übertragen. Im Gegensatz zum interaktiven Modus, der eine wechselseitige Interaktion zwischen Ihnen und dem Interpreter ermöglicht, wird der Quellcode eines Programms in eine Datei geschrieben. Diese wird als Ganzes vom Interpreter eingelesen und ausgeführt.

In den folgenden Abschnitten lernen Sie die Grundstrukturen eines Python-Programms kennen und werden Ihr erstes einfaches Beispielprogramm schreiben.

4.1 Tippen, kompilieren, testen

In diesem Abschnitt werden die Arbeitsabläufe besprochen, die nötig sind, um ein Python-Programm zu erstellen und auszuführen. Ganz allgemein sollten Sie sich darauf einstellen, dass wir in einem Großteil des Buchs ausschließlich *Konsolenanwendungen* schreiben werden. Eine Konsolenanwendung hat eine rein textbasierte Schnittstelle zum Benutzer und läuft in der *Konsole* (auch *Shell*) des jeweiligen Betriebssystems ab. Für die meisten Beispiele und auch für viele reale Anwendungsfälle reicht eine solche textbasierte Schnittstelle aus.[1]

Grundsätzlich besteht ein Python-Programm aus einer oder mehreren Programmdateien. Diese Programmdateien haben die Dateiendung *.py* und enthalten den Python-Quelltext. Dabei handelt es sich um nichts anderes als um Textdateien. Programmdateien können also mit einem normalen Texteditor bearbeitet werden.

Nachdem eine Programmdatei geschrieben wurde, besteht der nächste Schritt darin, sie auszuführen. Wenn Sie IDLE verwenden, kann die Programmdatei bequem über den Menüpunkt RUN • RUN MODULE ausgeführt werden. Sollten Sie einen Editor verwenden, der keine vergleichbare Funktion unterstützt, müssen Sie in einer Kommandozeile in das Verzeichnis der Programmdatei wechseln und – abhängig von Ihrem Betriebssystem – verschiedene Kommandos ausführen.

1 Selbstverständlich ermöglicht Python auch die Programmierung grafischer Benutzeroberflächen. Dies wird in Kapitel 41 behandelt.

Windows

Unter Windows wechseln Sie in das Verzeichnis, in dem die Programmdatei liegt, und starten den Python-Interpreter mit dem Kommando `python`, gefolgt von dem Namen der auszuführenden Programmdatei.[2]

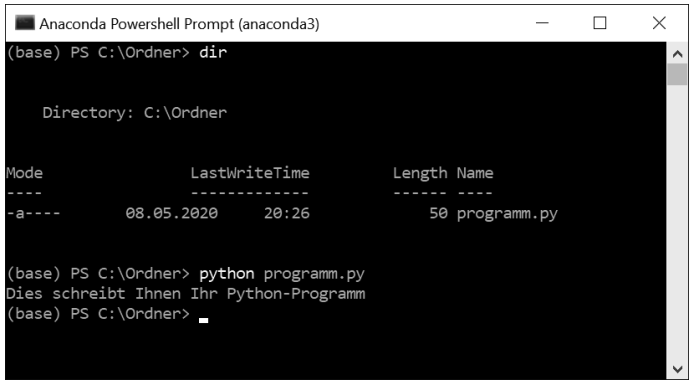

Abbildung 4.1 Ausführen eines Python-Programms unter Windows

Bei »Dies schreibt Ihnen Ihr Python-Programm« handelt es sich um eine Ausgabe des Python-Programms in der Datei *programm.py*, die beweist, dass das Python-Programm tatsächlich ausgeführt wurde.

> **Hinweis**
>
> Unter Windows ist es auch möglich, ein Python-Programm durch einen Doppelklick auf die jeweilige Programmdatei auszuführen. Das hat aber den Nachteil, dass sich das Konsolenfenster sofort nach Beenden des Programms schließt und die Ausgaben des Programms somit nicht erkennbar sind.

Linux und macOS

Unter Unix-ähnlichen Betriebssystemen wie Linux oder macOS wechseln Sie ebenfalls in das Verzeichnis, in dem die Programmdatei liegt, und starten dann den Python-Interpreter mit dem Kommando `python`, gefolgt von dem Namen der auszuführenden Programmdatei. Im folgenden Beispiel wird die Programmdatei *programm.py* unter Linux ausgeführt, die sich im Verzeichnis */home/user/ordner* befindet:

```
user@HOST ~ $ cd ordner
user@HOST ~/ordner $ python programm.py
Dies schreibt Ihnen Ihr Python-Programm
```

2 In älteren Windows-Versionen finden Sie die Konsole unter Start • Programme • Zubehör • Eingabeaufforderung. In neueren Windows-Versionen starten Sie die *PowerShell*.

Bitte beachten Sie den Hinweis in Abschnitt 2.2.2, der besagt, dass das Kommando, mit dem Sie Python starten, je nach Distribution von dem hier demonstrierten `python` abweichen kann.

4.1.1 Shebang

Unter einem Unix-ähnlichen Betriebssystem wie beispielsweise Linux können Python-Programmdateien mithilfe eines *Shebangs*, auch *Magic Line* genannt, direkt ausführbar gemacht werden. Dazu muss die erste Zeile der Programmdatei in der Regel folgendermaßen lauten:

```
#!/usr/bin/python
```

In diesem Fall wird das Betriebssystem dazu angehalten, diese Programmdatei immer mit dem Python-Interpreter auszuführen. Unter anderen Betriebssystemen, beispielsweise Windows, wird die Shebang-Zeile ignoriert.

Beachten Sie, dass der Python-Interpreter auf Ihrem System in einem anderen Verzeichnis als dem hier angegebenen installiert sein könnte. Allgemein ist daher folgende Shebang-Zeile besser, da sie vom tatsächlichen Installationsort von Python unabhängig ist:

```
#!/usr/bin/env python
```

Weitere Details zum Zusammenspiel zwischen der Shebang-Zeile und den virtuellen Umgebungen von Anaconda finden Sie in Abschnitt 39.2. Beachten Sie außerdem, dass das Executable-Flag der Programmdatei gesetzt werden muss, bevor die Datei tatsächlich ausführbar ist. Das geschieht mit folgendem Befehl:

```
$ chmod +x dateiname
```

Die in diesem Buch gezeigten Beispiele enthalten aus Gründen der Übersichtlichkeit keine Shebang-Zeile. Das bedeutet aber ausdrücklich nicht, dass vom Einsatz einer Shebang-Zeile abzuraten wäre.

4.1.2 Interne Abläufe

Bislang haben Sie eine ungefähre Vorstellung davon, was Python ausmacht und wo die Stärken dieser Programmiersprache liegen. Außerdem haben wir Ihnen das Grundwissen zum Erstellen und Ausführen einer Python-Programmdatei vermittelt. Doch in den vorangegangenen Abschnitten sind Begriffe wie »Compiler« oder »Interpreter« gefallen, ohne erklärt worden zu sein. In diesem Abschnitt möchten wir uns daher den internen Vorgängen widmen, die beim Ausführen einer Python-Programmdatei ablaufen. Abbildung 4.2 veranschaulicht, was beim Ausführen einer Programmdatei namens *programm.py* geschieht.

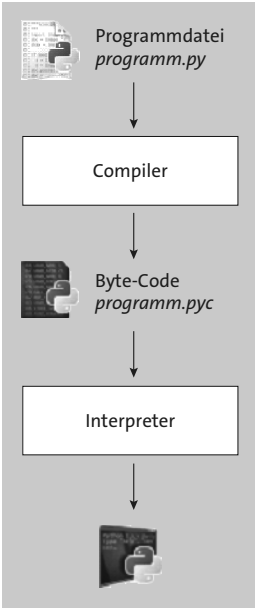

Abbildung 4.2 Kompilieren und Interpretieren einer Programmdatei

Wenn die Programmdatei *programm.py*, wie zu Beginn des Kapitels beschrieben, ausgeführt wird, passiert sie zunächst den *Compiler*. Als Compiler wird ein Programm bezeichnet, das von einer formalen Sprache in eine andere übersetzt. Im Falle von Python übersetzt der Compiler von der Sprache Python in den *Byte-Code*. Dabei steht es dem Compiler frei, den generierten Byte-Code im Arbeitsspeicher zu behalten oder als *programm.pyc* auf der Festplatte zu speichern.

Beachten Sie, dass der vom Compiler generierte Byte-Code nicht direkt auf dem Prozessor ausgeführt werden kann, im Gegensatz etwa zu C- oder C++-Kompilaten. Zur Ausführung des Byte-Codes wird eine weitere Abstraktionsschicht, der *Interpreter*, benötigt. Der Interpreter, häufig auch *virtuelle Maschine* (engl. *virtual machine*) genannt, liest den vom Compiler erzeugten Byte-Code ein und führt ihn aus.

Dieses Prinzip einer interpretierten Programmiersprache hat verschiedene Vorteile. So kann derselbe Python-Code beispielsweise unmodifiziert auf allen Plattformen ausgeführt werden, für die ein Python-Interpreter existiert. Allerdings laufen Programme interpretierter Programmiersprachen aufgrund des zwischengeschalteten Interpreters in der Regel auch langsamer als ein vergleichbares C-Programm, das direkt auf dem Prozessor ausgeführt wird.[3]

3 Diese Aussage stimmt nicht notwendigerweise, wenn der Interpreter Optimierungen zur Laufzeit des Programms durchführt, beispielsweise eine Just-in-time-Kompilierung. Aktuelle Versionen von CPython und der alternative Interpreter *PyPy* (siehe Abschnitt 40.1) führen solche Optimierungen durch, um die Programmausführung zu beschleunigen.

4.2 Grundstruktur eines Python-Programms

Um Ihnen ein Gefühl für die Sprache Python zu vermitteln, möchten wir Ihnen zunächst einen Überblick über ihre Syntax geben. 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 Sie sich zu halten haben. Tun Sie es nicht, so verursachen Sie den allseits bekannten *Syntax-Error*.

Python macht Ihnen sehr genaue Vorgaben, wie Sie Ihren Quellcode strukturieren müssen. Obwohl erfahrene Programmierer und Programmiererinnen darin eine Einschränkung sehen mögen, kommt diese Eigenschaft gerade Neulingen zugute, denn unstrukturierter und unübersichtlicher Code ist eine der größten Fehlerquellen in der Programmierung.

Grundsätzlich besteht ein Python-Programm aus einzelnen *Anweisungen*, die im einfachsten Fall genau eine Zeile im Quelltext einnehmen. Folgende Anweisung gibt beispielsweise einen Text auf dem Bildschirm aus:

```
print("Hallo Welt")
```

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

```
Anweisungskopf:
    Anweisung
    …
    Anweisung
```

Das kann in einem konkreten Python-Programm etwa so aussehen:

```
if x > 10:
    print("x ist größer als 10")
    print("Zweite Zeile!")
```

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 Einrückung kann sowohl über Tabulatoren als auch über Leerzeichen erfolgen, wobei Sie gut beraten sind, beides nicht zu vermischen. Wir empfehlen eine Einrückungstiefe von jeweils vier Leerzeichen.

Python unterscheidet sich hier von vielen gängigen Programmiersprachen, in denen die Zuordnung von Anweisungskopf und Anweisungskörper durch geschweifte Klammern oder Schlüsselwörter wie »Begin« und »End« erreicht wird.

Hinweis

Ein Programm, in dem sowohl Leerzeichen als auch Tabulatoren verwendet wurden, kann vom Python-Compiler anstandslos übersetzt werden, da jeder Tabulator intern durch acht Leerzeichen ersetzt wird. Dies kann aber zu schwer auffindbaren Fehlern führen, denn viele Editoren verwenden standardmäßig eine Tabulatorweite von vier Leerzeichen. Dadurch scheinen bestimmte Quellcodeabschnitte gleich weit eingerückt zu sein, obwohl sie es *de facto* nicht sind.

Bitte stellen Sie Ihren Editor so ein, dass jeder Tabulator automatisch durch Leerzeichen ersetzt wird, oder verwenden Sie ausschließlich Leerzeichen zur Einrückung Ihres Codes.

Möglicherweise fragen Sie sich jetzt, wie Anweisungen, die über mehrere Zeilen gehen, mit dem interaktiven Modus vereinbar sind, in dem ja immer nur eine Zeile bearbeitet werden kann. Nun, generell werden wir versuchen, den interaktiven Modus zu vermeiden, wenn ein Codebeispiel mehrere Zeilen lang ist. Dennoch ist die Frage berechtigt. Die Antwort: Es wird ganz intuitiv zeilenweise eingegeben. Wenn der Interpreter erkennt, dass eine Anweisung noch nicht vollendet ist, ändert er den Prompt von >>> in ... Geben wir einmal unser oben dargestelltes Beispiel in den interaktiven Modus ein:

```
>>> x = 123
>>> if x > 10:
...     print("Der Interpreter leistet gute Arbeit")
...     print("Zweite Zeile!")
...
Der Interpreter leistet gute Arbeit
Zweite Zeile!
>>>
```

Beachten Sie, dass Sie die aktuelle Einrückungstiefe berücksichtigen müssen, auch wenn eine Zeile mit ... beginnt. Darüber hinaus kann der Interpreter das Ende des Anweisungskörpers nicht automatisch erkennen, da dieser beliebig viele Anweisungen enthalten kann. Deswegen muss ein Anweisungskörper im interaktiven Modus durch Drücken der ↵-Taste beendet werden.

4.2.1 Umbrechen langer Zeilen

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. Geläufige maximale Zei-

lenlängen sind 80 oder 120 Zeichen. Innerhalb von Klammern dürfen Sie Quellcode beliebig umbrechen:

```
>>> var = (
... 10
... +
... 10
... )
>>> var
20
```

An vielen anderen Stellen, an denen keine Klammern gesetzt werden dürfen, sind Sie an die strengen syntaktischen Regeln von Python gebunden. Durch den Einsatz der Backslash-Notation ist es möglich, Quellcode an nahezu beliebigen Stellen in eine neue Zeile zu umbrechen:

```
>>> var \
... = \
... 10
>>> var
10
```

Grundsätzlich kann ein Backslash überall da stehen, wo auch ein Leerzeichen hätte stehen können. Daher ist auch ein Backslash innerhalb eines Strings möglich:

```
>>> "Hallo \
... Welt"
'Hallo Welt'
```

Beachten Sie dabei aber, dass eine Einrückung des umbrochenen Teils des Strings Leerzeichen in den String schreibt. Aus diesem Grund sollten Sie die folgende Variante, einen String in mehrere Zeilen zu schreiben, vorziehen:

```
>>> "Hallo " \
... "Welt"
'Hallo Welt'
```

4.2.2 Zusammenfügen mehrerer Zeilen

Genauso, wie Sie eine einzeilige Anweisung mithilfe des Backslashs auf mehrere Zeilen umbrechen, können Sie mehrere einzeilige Anweisungen in einer Zeile zusammenfassen. Dazu werden die Anweisungen durch ein Semikolon voneinander getrennt:

```
>>> print("Hallo"); print("Welt")
Hallo
Welt
```

Anweisungen, die aus einem Anweisungskopf und einem Anweisungskörper bestehen, können auch ohne Einsatz eines Semikolons in eine Zeile gefasst werden, sofern der Anweisungskörper selbst aus nicht mehr als einer Zeile besteht:

```
>>> x = True
>>> if x: print("Hallo Welt")
...
Hallo Welt
```

Sollte der Anweisungskörper mehrere Zeilen lang sein, können diese durch ein Semikolon zusammengefasst werden:

```
>>> x = True
>>> if x: print("Hallo"); print("Welt")
...
Hallo
Welt
```

Alle durch ein Semikolon zusammengefügten Anweisungen werden so behandelt, als wären sie gleich weit eingerückt. Allein ein Doppelpunkt vergrößert die Einrückungstiefe. Aus diesem Grund gibt es im oben genannten Beispiel keine Möglichkeit, in derselben Zeile eine Anweisung zu schreiben, die nicht mehr im Körper der `if`-Anweisung steht.

Hinweis

Beim Einsatz des Backslashs und vor allem des Semikolons entsteht schnell unleserlicher Code. Verwenden Sie beide Notationen daher nur, wenn Sie meinen, dass es der Lesbarkeit und Übersichtlichkeit dienlich ist.

4.3 Das erste Programm

Als Einstieg in die Programmierung mit Python erstellen wir ein kleines Beispielprogramm, das Spiel »Zahlenraten«. Die Spielidee ist folgende: Der Spieler soll eine im Programm festgelegte Zahl erraten. Dazu stehen ihm beliebig viele Versuche zur Verfügung. Nach jedem Versuch informiert ihn das Programm darüber, ob die geratene Zahl zu groß, zu klein oder genau richtig gewesen ist. Sobald der Spieler die Zahl erraten hat, gibt das Programm die Anzahl der Versuche aus und wird beendet. Aus Sicht des Spielers soll das Ganze folgendermaßen aussehen:

```
Raten Sie: 42
Zu klein
Raten Sie: 10000
Zu groß
Raten Sie: 999
Zu klein
Raten Sie: 1337
Super, Sie haben es in 4 Versuchen geschafft!
```

Kommen wir vom Ablaufprotokoll zur konkreten Implementierung in Python.

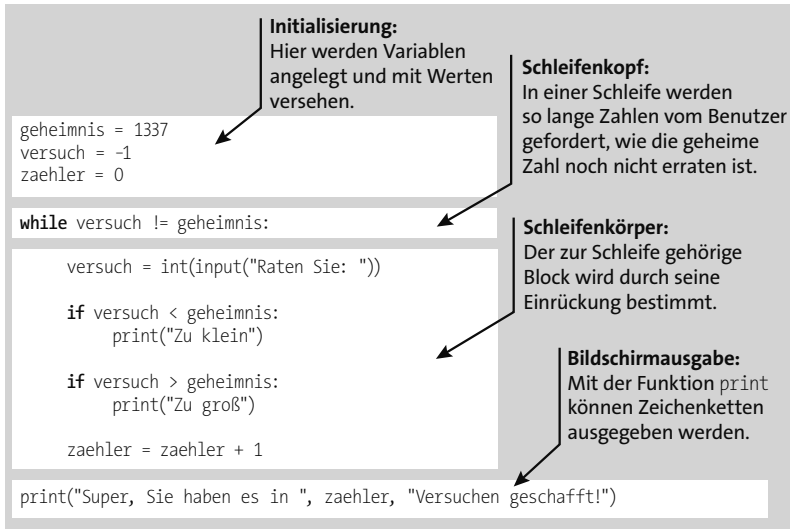

Abbildung 4.3 Zahlenraten, ein einfaches Beispiel

Die in Abbildung 4.3 hervorgehobenen Bereiche des Programms werden im Folgenden noch einmal ausführlich diskutiert.

Initialisierung

Bei der Initialisierung werden die für das Spiel benötigten Variablen angelegt. Python unterscheidet zwischen verschiedenen Datentypen, wie etwa Zeichenketten, Ganz- oder Fließkommazahlen. Der Typ einer Variablen wird zur Laufzeit des Programms anhand des ihr zugewiesenen Wertes bestimmt. Es ist also nicht nötig, einen Datentyp explizit anzugeben. Eine Variable kann im Laufe des Programms ihren Typ ändern.

In unserem Spiel werden Variablen für die gesuchte Zahl (`geheimnis`), die Benutzereingabe (`versuch`) und den Versuchszähler (`zaehler`) angelegt und mit Anfangswerten versehen. Dadurch, dass `versuch` und `geheimnis` zu Beginn des Programms verschiedene Werte haben, ist sichergestellt, dass die Schleife anläuft.

Schleifenkopf

Eine `while`-Schleife wird eingeleitet. Eine `while`-Schleife läuft so lange, wie die im Schleifenkopf genannte Bedingung (`versuch != geheimnis`) erfüllt ist, also in diesem Fall, bis die Variablen `versuch` und `geheimnis` den gleichen Wert haben. Aus Benutzersicht bedeutet dies: Die Schleife läuft so lange, bis die Benutzereingabe mit der zu erratenden Zahl übereinstimmt.

Den zum Schleifenkopf gehörigen Schleifenkörper erkennt man daran, dass die nachfolgenden Zeilen um eine Stufe weiter eingerückt wurden. Sobald die Einrückung wieder um einen Schritt nach links geht, endet der Schleifenkörper.

Schleifenkörper

In der ersten Zeile des Schleifenkörpers wird eine vom Spieler eingegebene Zahl eingelesen und in der Variablen `versuch` gespeichert. Dabei wird mithilfe von `input("Raten Sie: ")` die Eingabe des Benutzers eingelesen und mit `int` in eine ganze Zahl konvertiert (von engl. *integer*, »ganze Zahl«). Diese Konvertierung ist wichtig, da Benutzereingaben generell als String eingelesen werden. In unserem Fall möchten wir die Eingabe jedoch als Zahl weiterverwenden. Der String `"Raten Sie: "` wird vor der Eingabe ausgegeben und dient dazu, den Benutzer zur Eingabe der Zahl aufzufordern.

Nach dem Einlesen wird einzeln geprüft, ob die eingegebene Zahl `versuch` größer oder kleiner als die gesuchte Zahl `geheimnis` ist, und mittels `print` eine entsprechende Meldung ausgegeben. Schließlich wird der Versuchszähler `zaehler` um eins erhöht.

Nach dem Hochzählen des Versuchszählers endet der Schleifenkörper, da die nächste Zeile nicht mehr unter dem Schleifenkopf eingerückt ist.

Bildschirmausgabe

Die letzte Programmzeile gehört nicht mehr zum Schleifenkörper. Das bedeutet, dass sie erst ausgeführt wird, wenn die Schleife vollständig durchlaufen, das Spiel also gewonnen ist. In diesem Fall werden eine Erfolgsmeldung sowie die Anzahl der benötigten Versuche ausgegeben. Das Spiel ist beendet.

Erstellen Sie jetzt Ihr erstes Python-Programm, indem Sie den Programmcode in eine Datei namens *spiel.py* schreiben und ausführen. Ändern Sie den Startwert von `geheimnis`, und spielen Sie das Spiel.

4.4 Kommentare

Sie können sich sicherlich vorstellen, dass es nicht das Ziel ist, Programme zu schreiben, die auf eine Postkarte passen würden. Mit der Zeit wird der Quelltext Ihrer Programme umfangreicher und komplexer werden. Irgendwann ist der Zeitpunkt er-

reicht, da bloßes Gedächtnistraining nicht mehr ausreicht, um die Übersicht zu bewahren. Spätestens dann kommen Kommentare ins Spiel.

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:

```
# Ein Beispiel mit Kommentaren
print("Hallo Welt!") # Simple Hallo-Welt-Ausgabe
```

Für längere Kommentare bietet sich ein *Blockkommentar* an. Ein Blockkommentar beginnt und endet mit drei aufeinanderfolgenden Anführungszeichen:[4]

```
""" Dies ist ein Blockkommentar,
er kann sich über mehrere Zeilen erstrecken. """
```

Kommentare sollten nur gesetzt werden, wenn sie zum Verständnis des Quelltextes beitragen oder wertvolle Informationen enthalten. Jede noch so unwichtige Zeile zu kommentieren, führt dazu, dass man den Wald vor lauter Bäumen nicht mehr sieht.

4.5 Der Fehlerfall

Vielleicht haben Sie bereits mit dem Beispielprogramm aus Abschnitt 4.3, »Das erste Programm«, gespielt und sind dabei auf eine solche oder ähnliche Ausgabe des Interpreters gestoßen:

```
  File "hallo_welt.py", line 10
    if versuch < geheimnis
                           ^
SyntaxError: expected ':'
```

Es handelt sich dabei um eine Fehlermeldung, die in diesem Fall auf einen Syntaxfehler im Programm hinweist. Können Sie erkennen, welcher Fehler hier vorliegt? Richtig, es fehlt der Doppelpunkt am Ende der Zeile.

Python stellt bei der Ausgabe einer Fehlermeldung wichtige Informationen bereit, die bei der Fehlersuche hilfreich sind:

4 Eigentlich wird mit dieser Notation kein Blockkommentar erzeugt, sondern ein mehrzeiliger String, der sich aber auch dazu eignet, größere Quellcodebereiche »auszukommentieren«.

- Die erste Zeile der Fehlermeldung gibt Aufschluss darüber, in welcher Zeile innerhalb welcher Datei der Fehler aufgetreten ist. In diesem Fall handelt es sich um Zeile 10 in der Datei *spiel.py*.
- Der mittlere Teil zeigt den betroffenen Ausschnitt des Quellcodes, wobei die genaue Stelle, auf die sich die Meldung bezieht, mit einem kleinen Pfeil markiert ist. Wichtig ist, dass dies die Stelle ist, an der der Interpreter den Fehler erstmalig feststellen konnte. Das ist nicht unbedingt gleichbedeutend mit der Stelle, an der der Fehler gemacht wurde.
- Die letzte Zeile spezifiziert den Typ der Fehlermeldung, in diesem Fall einen `SyntaxError`. Dies sind die am häufigsten auftretenden Fehlermeldungen. Sie zeigen an, dass der Compiler das Programm aufgrund eines formalen Fehlers nicht weiter übersetzen konnte.

Neben dem Syntaxfehler gibt es eine Reihe weiterer Fehlertypen, die an dieser Stelle nicht alle im Detail besprochen werden sollen.[5] Wir möchten jedoch noch auf den `IndentationError` (dt. »Einrückungsfehler«) hinweisen, da er gerade bei Python-Anfängern und -Anfängerinnen häufig auftritt. Versuchen Sie dazu einmal, folgendes Programm auszuführen:

```
i = 10
if i == 10:
print("Falsch eingerückt")
```

Sie sehen, dass die letzte Zeile eigentlich einen Schritt weiter eingerückt sein müsste. So, wie das Programm jetzt geschrieben wurde, hat die `if`-Anweisung keinen Anweisungskörper. Das ist nicht zulässig, und daher tritt ein `IndentationError` auf:

```
  File "indent.py", line 3
    print("Falsch eingerückt")
    ^
IndentationError: expected an indented block after 'if' statement on line 2
```

Nachdem wir uns mit diesen Grundlagen vertraut gemacht haben, kommen wir zu den Kontrollstrukturen, die es Ihnen erlauben, den Programmfluss zu steuern.

Hinweis

Mit Python 3.10 und 3.11 wurden viele übliche Fehlermeldungen und auch deren allgemeine Formatierung überarbeitet. Sollten Sie also eine ältere Python-Version einsetzen, können die Ausgaben von den im Buch abgedruckten Beispielen abweichen.

5 Sie finden eine Übersicht über alle Fehlertypen in Anhang A.4.

Kapitel 11
Numerische Datentypen

In diesem Kapitel besprechen wir mit den numerischen Datentypen die erste große Gruppe von Datentypen in Python. Tabelle 11.1 listet alle zu dieser Gruppe gehörigen Datentypen auf und nennt ihren Zweck.

Datentyp	Beschreibung	Veränderlichkeit*	Abschnitt
`int`	ganze Zahlen	*unveränderlich*	Abschnitt 11.4
`float`	Gleitkommazahlen	*unveränderlich*	Abschnitt 11.5
`bool`	boolesche Werte	*unveränderlich*	Abschnitt 11.6
`complex`	komplexe Zahlen	*unveränderlich*	Abschnitt 11.7

* Alle numerischen Datentypen sind unveränderlich. Das bedeutet nicht, dass es keine Operatoren gibt, die Zahlen verändern, sondern vielmehr, dass nach jeder Veränderung eine neue Instanz des jeweiligen Datentyps erzeugt werden muss. Aus Sicht des Programmierers besteht also zunächst kaum ein Unterschied. Näheres zum Unterschied zwischen veränderlichen und unveränderlichen Datentypen erfahren Sie in Abschnitt 7.3.

Tabelle 11.1 Numerische Datentypen

Die numerischen Datentypen bilden eine Gruppe, weil sie thematisch zusammengehören. Diese Zusammengehörigkeit schlägt sich auch darin nieder, dass die numerischen Datentypen viele gemeinsame Operatoren haben. In den folgenden Abschnitten werden wir diese gemeinsamen Operatoren behandeln und im Anschluss daran die numerischen Datentypen `int`, `float`, `bool` und `complex` detailliert besprechen.

11.1 Arithmetische Operatoren

Unter einem *arithmetischen Operator* wird ein Operator verstanden, der eine arithmetische Berechnung vornimmt, beispielsweise eine Addition oder eine Multiplikation. Für alle numerischen Datentypen sind die in Tabelle 11.2 aufgeführten arithmetischen Operatoren definiert.

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	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*

* Die Operatoren % und // haben für komplexe Zahlen keine mathematische Bedeutung und sind deshalb für den Datentyp `complex` nicht definiert.

Tabelle 11.2 Gemeinsame Operatoren numerischer Datentypen

Hinweis

Zwei Anmerkungen für Leser und Leserinnen, die bereits mit einer C-ähnlichen Programmiersprache vertraut sind:

Es gibt in Python keine Entsprechungen für die Inkrementierungs- und Dekrementierungsoperatoren ++ und -- aus C.

Die Operatoren % und // können folgendermaßen beschrieben werden:

- x // y = `runden`(x / y)
- x % y = x - y * `runden`(x / y)

Python rundet dabei stets ab, während C zur Null hin rundet. Dieser Unterschied tritt nur auf, wenn die Operanden gegensätzliche Vorzeichen haben.

11.1.1 Erweiterte Zuweisungen

Neben diesen grundlegenden Operatoren gibt es in Python eine Reihe zusätzlicher Operatoren. Oftmals möchte man beispielsweise die Summe von x und y berechnen und das Ergebnis in x speichern, x also um y erhöhen. Dazu ist mit den oben genannten Operatoren folgende Anweisung nötig:

```
x = x + y
```

Für solche Fälle gibt es in Python sogenannte *erweiterte Zuweisungen* (engl. *augmented assignments*), die als eine Art Abkürzung für die oben genannte Anweisung angesehen werden können. Tabelle 11.3 listet die in Python definierten erweiterten Zuweisungen auf.

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 11.3 Gemeinsame Operatoren numerischer Datentypen

Wichtig ist, dass Sie hier für y einen beliebigen arithmetischen Ausdruck einsetzen können, während x ein Ausdruck sein muss, der auch als Ziel einer normalen Zuweisung eingesetzt werden könnte, also zum Beispiel ein symbolischer Name oder ein Element einer Liste oder eines Dictionarys.

11.2 Vergleichende Operatoren

Ein *vergleichender Operator* ist ein Operator, der aus zwei Instanzen einen Wahrheitswert berechnet. Tabelle 11.4 listet die vergleichenden Operatoren auf, die für numerische Datentypen definiert sind.

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

Tabelle 11.4 Gemeinsame Operatoren numerischer Datentypen

Operator	Ergebnis
x > y	wahr, wenn x größer ist als y*
x >= y	wahr, wenn x größer oder gleich y ist*
* Da komplexe Zahlen prinzipiell nicht sinnvoll anzuordnen sind, lässt der Datentyp complex nur die Verwendung der ersten beiden Operatoren zu.	

Tabelle 11.4 Gemeinsame Operatoren numerischer Datentypen (Forts.)

Jeder dieser vergleichenden Operatoren liefert als Ergebnis einen Wahrheitswert. Ein solcher Wert wird zum Beispiel als Bedingung einer `if`-Anweisung erwartet. Die Operatoren könnten also folgendermaßen verwendet werden:

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

Sie können beliebig viele der vergleichenden Operatoren zu einer Reihe verketten. Das obere Beispiel ist genau genommen nur ein Spezialfall dieser Regel – mit lediglich zwei Operanden. Die Bedeutung einer solchen Verkettung entspricht der mathematischen Sichtweise und ist am folgenden Beispiel zu erkennen:

```
if 2 < x < 4:
    print("x liegt zwischen 2 und 4")
```

Mehr zu booleschen Werten folgt in Abschnitt 11.6.

11.3 Konvertierung zwischen numerischen Datentypen

Numerische Datentypen können über die eingebauten Funktionen `int`, `float`, `bool` und `complex` ineinander umgeformt werden. Dabei können je nach Umformung Informationen verloren gehen. Als Beispiel betrachten wir einige Konvertierungen im interaktiven Modus:

```
>>> float(33)
33.0
>>> int(33.5)
33
>>> bool(12)
True
>>> complex(True)
(1+0j)
```

Anstelle eines konkreten Literals kann auch eine Referenz eingesetzt bzw. eine Referenz mit dem entstehenden Wert verknüpft werden:

```
>>> var1 = 12.5
>>> int(var1)
12
>>> var2 = int(40.25)
>>> var2
40
```

Hinweis

Der Datentyp `complex` nimmt bei den oben vorgestellten Konvertierungen eine Sonderstellung ein, da er sich nicht sinnvoll in einer pauschalen Weise auf einen einzelnen Zahlenwert reduzieren lässt. Aus diesem Grund schlägt eine Konvertierung wie beispielsweise `int(1+2j)` fehl.

So viel zur allgemeinen Einführung in die numerischen Datentypen. Die folgenden Abschnitte werden jeden Datentyp dieser Gruppe im Detail behandeln.

11.4 Ganzzahlen – int

Für die Arbeit mit ganzen Zahlen gibt es in Python den Datentyp `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.[1]

Wir haben bereits viel mit ganzen Zahlen gearbeitet, sodass die Verwendung von `int` eigentlich keiner Demonstration mehr bedarf. Der Vollständigkeit halber sehen Sie hier dennoch ein kleines Beispiel:

```
>>> i = 1234
>>> i
1234
>>> p = int(5678)
>>> p
5678
```

1 In Python 2 existierten noch zwei Datentypen für ganze Zahlen: `int` für den begrenzten Zahlenraum von 32 Bit bzw. 64 Bit sowie `long` mit einem unbegrenzten Wertebereich.

Seit Python 3.6 kann ein Unterstrich verwendet werden, um die Ziffern eines Literals zu gruppieren:

```
>>> 1_000_000
1000000
>>> 1_0_0
100
```

Die Gruppierung ändert nichts am Zahlenwert des Literals, sondern dient dazu, die Lesbarkeit von Zahlenliteralen zu erhöhen. Ob und wie Sie die Ziffern gruppieren, bleibt Ihnen überlassen.

11.4.1 Zahlensysteme

Ganze Zahlen können in Python in mehreren *Zahlensystemen*[2] geschrieben werden:

- Zahlen, die wie im oben dargestellten Beispiel ohne ein spezielles Präfix geschrieben sind, werden im *Dezimalsystem* (Basis 10) interpretiert. Beachten Sie, dass einer solchen Zahl keine führenden Nullen vorangestellt werden dürfen:

  ```
  v_dez = 1337
  ```

- Das Präfix `0o` (»Null-o«) kennzeichnet eine Zahl, die im *Oktalsystem* (Basis 8) geschrieben wurde. Beachten Sie, dass hier nur Ziffern von `0` bis `7` erlaubt sind:

  ```
  v_okt = 0o2471
  ```

  Das kleine »o« im Präfix kann auch durch ein großes »O« ersetzt werden. Wir empfehlen Ihnen jedoch, stets ein kleines »o« zu verwenden, da das große »O« in vielen Schriftarten von der Null kaum zu unterscheiden ist.

- Die nächste und weitaus gebräuchlichere Variante ist das *Hexadezimalsystem* (Basis 16), das durch das Präfix `0x` bzw. `0X` (Null-x) gekennzeichnet wird. Die Zahl selbst darf aus den Ziffern `0`–`9` und den Buchstaben `A`–`F` bzw. `a`–`f` gebildet werden:

  ```
  v_hex = 0x5A3F
  ```

- Neben dem Hexadezimalsystem ist in der Informatik das *Dualsystem*, auch *Binärsystem* (Basis 2), von entscheidender Bedeutung. Zahlen im Dualsystem werden analog zu den vorangegangenen Literalen durch das Präfix `0b` eingeleitet:

  ```
  v_bin = 0b1101
  ```

  Im Dualsystem dürfen nur die Ziffern `0` und `1` verwendet werden.

Vielleicht möchten Sie sich nicht auf diese vier Zahlensysteme beschränken, die von Python explizit unterstützt werden, sondern ein exotischeres verwenden. Natürlich

2 Sollten Sie nicht wissen, was ein Zahlensystem ist, können Sie diesen Abschnitt problemlos überspringen.

gibt es in Python nicht für jedes mögliche Zahlensystem ein eigenes Literal. Stattdessen können Sie sich folgender Schreibweise bedienen:

```
v_6 = int("54425", 6)
```

Es handelt sich um eine alternative Methode, eine Instanz des Datentyps `int` zu erzeugen und mit einem Anfangswert zu versehen. Dazu werden in den Klammern ein String, der den gewünschten Initialwert in dem gewählten Zahlensystem enthält, sowie die Basis dieses Zahlensystems als ganze Zahl geschrieben. Beide Werte müssen durch ein Komma getrennt werden. Im Beispiel wurde das Sechsersystem verwendet.

Python unterstützt Zahlensysteme mit einer Basis von 2 bis 36. Wenn ein Zahlensystem mehr als zehn verschiedene Ziffern zur Darstellung einer Zahl benötigt, werden zusätzlich zu den Ziffern 0 bis 9 die Buchstaben A bis Z des englischen Alphabets verwendet.

Die Variable `v_6` hat jetzt den Wert `7505` im Dezimalsystem.

Für alle Zahlensystem-Literale ist die Verwendung eines negativen Vorzeichens möglich:

```
>>> -1234
-1234
>>> -0o777
-511
>>> -0xFF
-255
>>> -0b1010101
-85
```

Beachten Sie, dass es sich bei den Zahlensystemen nur um eine alternative Schreibweise des gleichen Wertes handelt. Der Datentyp `int` springt beispielsweise nicht in eine Art Hexadezimalmodus, sobald er einen solchen Wert enthält, sondern das Zahlensystem ist nur bei Zuweisungen oder Ausgaben von Bedeutung. Standardmäßig werden alle Zahlen im Dezimalsystem ausgegeben:

```
>>> v1 = 0xFF
>>> v2 = 0o777
>>> v1
255
>>> v2
511
```

Wir werden später in Abschnitt 12.5 im Zusammenhang mit Strings darauf zurückkommen, wie sich Zahlen in anderen Zahlensystemen ausgeben lassen.

11.4.2 Bit-Operationen

Wie bereits gesagt, hat das Dualsystem oder auch Binärsystem in der Informatik eine große Bedeutung. Für den Datentyp `int` sind daher einige zusätzliche Operatoren definiert, die sich explizit auf die binäre Darstellung der Zahl beziehen. Tabelle 11.5 fasst diese *Bit-Operatoren* zusammen.

Operator	Erweiterte 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 11.5 Bit-Operatoren des Datentyps int

Da vielleicht nicht jedem unmittelbar klar ist, was die einzelnen Operationen bewirken, möchten wir sie im Folgenden im Detail besprechen.

Bitweises UND

Das *bitweise UND* zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verknüpft werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau dort eine 1, wo beide der jeweiligen Bits der Operanden 1 sind, und sonst eine 0. Dies veranschaulicht Abbildung 11.1.

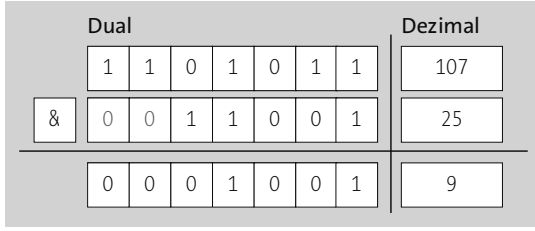

Abbildung 11.1 Bitweises UND

Im interaktiven Modus von Python probieren wir aus, ob das bitweise UND mit den in der Grafik gewählten Operanden tatsächlich das erwartete Ergebnis zurückgibt:

```
>>> 107 & 25
9
>>> 0b1101011 & 0b11001
9
>>> bin(0b1101011 & 0b11001)
'0b1001'
```

Im Beispiel verwenden wir die eingebaute Funktion `bin` (siehe Abschnitt 17.14.5), um das Ergebnis des bitweisen UND im Binärsystem darzustellen.

Bitweises ODER

Das *bitweise ODER* zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verglichen werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine 1, wo mindestens eines der jeweiligen Bits der Operanden 1 ist. Abbildung 11.2 veranschaulicht dies.

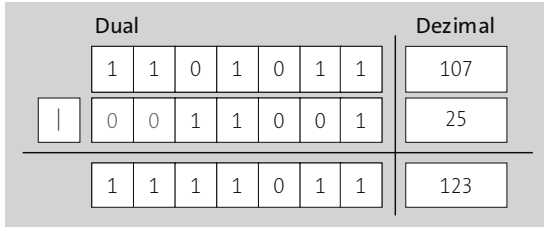

Abbildung 11.2 Bitweises nicht ausschließendes ODER

Im interaktiven Modus von Python probieren wir aus, ob das bitweise ODER mit den in der Grafik gewählten Operanden tatsächlich das erwartete Ergebnis zurückgibt:

```
>>> 107 | 25
123
>>> 0b1101011 | 0b11001
123
>>> bin(0b1101011 | 0b11001)
'0b1111011'
```

Im Beispiel verwenden wir die eingebaute Funktion `bin` (siehe Abschnitt 17.14.5), um das Ergebnis des bitweisen ODER im Binärsystem darzustellen.

Bitweises ausschließendes ODER

Das *bitweise ausschließende ODER* (auch *exklusives ODER*) zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verglichen werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine 1, wo sich

die jeweiligen Bits der Operanden voneinander unterscheiden, und eine 0, wo sie gleich sind. Dies zeigt Abbildung 11.3.

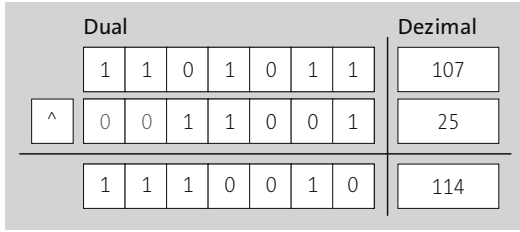

Abbildung 11.3 Bitweises exklusives ODER

Im interaktiven Modus von Python probieren wir aus, ob das bitweise ausschließende ODER mit den in der Grafik gewählten Operanden tatsächlich das erwartete Ergebnis zurückgibt:

```
>>> 107 ^ 25
114
>>> 0b1101011 ^ 0b11001
114
>>> bin(0b1101011 ^ 0b11001)
'0b1110010'
```

Im Beispiel verwenden wir die eingebaute Funktion `bin` (siehe Abschnitt 17.14.5), um das Ergebnis des bitweisen ausschließenden ODER im Binärsystem darzustellen.

Bitweises Komplement

Das *bitweise Komplement* bildet das sogenannte *Einerkomplement* einer Dualzahl, das der Negation aller vorkommenden Bits entspricht. In Python ist dies auf Bit-Ebene nicht möglich, da eine ganze Zahl in ihrer Länge unbegrenzt ist und das Komplement immer in einem abgeschlossenen Zahlenraum gebildet werden muss. Deswegen wird die eigentliche Bit-Operation zur arithmetischen Operation und ist folgendermaßen definiert:[3]

$\sim x = -x - 1$

Im interaktiven Modus lässt sich die Funktionsweise des bitweisen Komplements experimentell erproben:

3 Das ist sinnvoll, da man zur Darstellung negativer Zahlen in abgeschlossenen Zahlenräumen das sogenannte *Zweierkomplement* verwendet. Dieses erhalten Sie, indem Sie zum Einerkomplement 1 addieren.
Also: $-x$ = Zweierkomplement von $x = \sim x + 1$ Daraus folgt: $\sim x = -x - 1$

```
>>> ~9
-10
>>> ~0b1001
-10
>>> bin(~0b1001)
'-0b1010'
```

Im Beispiel verwenden wir die eingebaute Funktion `bin` (siehe Abschnitt 17.14.5), um das Ergebnis des bitweisen Komplements im Binärsystem darzustellen.

Bit-Verschiebung

Bei der *Bit-Verschiebung* (engl. *bit shift*) wird die Bit-Folge in der binären Darstellung des ersten Operanden um die durch den zweiten Operanden gegebene Anzahl Stellen nach links bzw. rechts verschoben. Auf der rechten Seite entstehende Lücken werden mit Nullen gefüllt, und das Vorzeichen des ersten Operanden bleibt erhalten. Abbildung 11.4 und Abbildung 11.5 veranschaulichen eine Verschiebung um zwei Stellen nach links bzw. nach rechts.

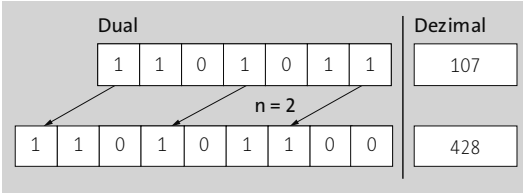

Abbildung 11.4 Bit-Verschiebung um zwei Stellen nach links

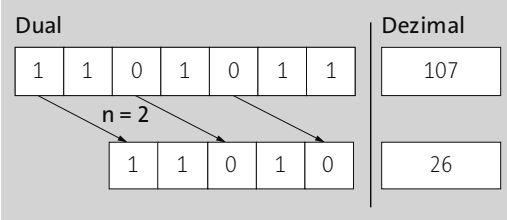

Abbildung 11.5 Bit-Verschiebung um zwei Stellen nach rechts

Die in der Bit-Darstellung entstehenden Lücken auf der rechten bzw. linken Seite werden mit Nullen aufgefüllt.

Die Bit-Verschiebung ist in Python ähnlich wie der Komplementoperator arithmetisch implementiert. Ein Shift um *x* Stellen nach rechts entspricht einer ganzzahligen Division durch 2^x. Ein Shift um *x* Stellen nach links entspricht einer Multiplikation mit 2^x.

Auch für die bitweisen Verschiebungen können wir die in den Grafiken gezeigten Beispiele im interaktiven Modus nachvollziehen:

```
>>> 107 << 2
428
>>> 107 >> 2
26
>>> bin(0b1101011 << 2)
'0b110101100'
>>> bin(0b1101011 >> 2)
'0b11010'
```

Im Beispiel verwenden wir die eingebaute Funktion `bin` (siehe Abschnitt 17.14.5), um die Ergebnisse der Bit-Verschiebungen im Binärsystem darzustellen.

11.4.3 Die Methode bit_length

Der Datentyp `int` verfügt über eine Methode, die sich auf die Binärdarstellung der ganzen Zahl bezieht. Die Methode `bit_length` berechnet die Anzahl Stellen, die für die Binärdarstellung der Zahl benötigt werden:

```
>>> (36).bit_length()
6
>>> (4345).bit_length()
13
```

Die Binärdarstellung der `36` ist `100100`, und die der `4345` ist `1000011111001`. Damit benötigen die beiden Zahlen `6` bzw. `13` Stellen für ihre Binärdarstellung.

Hinweis

Beachten Sie, dass die Klammern um die Zahlenliterale bei ganzen Zahlen benötigt werden, da es sonst zu Doppeldeutigkeiten mit der Syntax für Gleitkommazahlen kommen könnte.

11.5 Gleitkommazahlen – float

Zu Beginn dieses Teils sind wir bereits oberflächlich auf Gleitkommazahlen eingegangen, was wir in diesem Abschnitt ein wenig vertiefen möchten. Zum Speichern einer Gleitkommazahl mit begrenzter Genauigkeit[4] wird der Datentyp `float` verwendet.

4 In Abschnitt 11.5.2 besprechen wir einige Details zur Genauigkeit des Datentyps.

Wie bereits besprochen wurde, sieht das Literal für eine Gleitkommazahl im einfachsten Fall folgendermaßen aus:

```
v = 3.141
```

Vor- und Nachkommaanteil können dabei weggelassen werden, wenn sie den Wert 0 haben:

```
>>> -3.
-3.0
>>> .001
0.001
```

Achten Sie dabei darauf, dass der Punkt ein essenzielles Element eines Gleitkommazahl-Literals ist und als solches nicht weggelassen werden darf.

Seit Python 3.6 kann zudem ein Unterstrich verwendet werden, um die Ziffern eines Gleitkommazahl-Literals zu gruppieren:

```
>>> 3.000_000_1
3.0000001
```

11.5.1 Exponentialschreibweise

Python unterstützt außerdem eine Notation, die es ermöglicht, die Exponentialschreibweise zu verwenden:

```
v = 3.141e-12
```

Durch ein kleines oder großes `e` wird die *Mantisse* (`3.141`) vom *Exponenten* (`-12`) getrennt. Übertragen in die mathematische Schreibweise, entspricht dies dem Wert $3{,}141 \cdot 10^{-12}$. Beachten Sie, dass sowohl die Mantisse als auch der Exponent im Dezimalsystem angegeben werden müssen. Andere Zahlensysteme sind nicht vorgesehen, was die gefahrlose Verwendung führender Nullen ermöglicht:

```
v = 03.141e-0012
```

11.5.2 Genauigkeit

Eventuell haben Sie gerade schon etwas mit den Gleitkommazahlen experimentiert und sind dabei auf einen vermeintlichen Fehler des Interpreters gestoßen:

```
>>> 1.1 + 2.2
3.3000000000000003
```

Reelle Zahlen können im Datentyp `float` nicht unendlich präzise gespeichert werden, sondern werden stattdessen mit einer bestimmten Genauigkeit angenähert.

Wenn Sie technisch versiert sind und jetzt von anderen Programmiersprachen zu Python wechseln, wird es Sie interessieren, dass `float`-Instanzen in Python IEEE-754-Gleitkommazahlen mit doppelter Genauigkeit sind. Der Datentyp `float` in Python ist damit mit dem Datentyp `double` in C, C++ und Java vergleichbar.

Falls Sie explizit Gleitkommazahlen mit einfacher Genauigkeit verwenden möchten, können Sie auf den Datentyp `float32` der Drittanbieterbibliothek *NumPy* (siehe Kapitel 43) zurückgreifen.

11.5.3 Unendlich und Not a Number

Gleitkommazahlen können als `float` nicht beliebig genau gespeichert werden. Das impliziert auch, dass es sowohl eine Ober- als auch eine Untergrenze für diesen Datentyp geben muss. Und tatsächlich können Gleitkommazahlen, die in ihrer Größe ein bestimmtes Limit überschreiten, in Python nicht mehr dargestellt werden. Wenn das Limit überschritten ist, wird die Zahl als `inf` gespeichert[5] bzw. als `-inf`, wenn das untere Limit unterschritten wurde. Es kommt also zu keinem Fehler, und es ist immer noch möglich, eine übergroße Zahl mit anderen zu vergleichen:

```
>>> 3.0e999
inf
>>> -3.0e999
-inf
>>> 3.0e999 < 12.0
False
>>> 3.0e999 > 12.0
True
>>> 3.0e999 == 3.0e999999999999
True
```

Es ist zwar möglich, zwei unendlich große Gleitkommazahlen miteinander zu vergleichen, jedoch lässt sich nur bedingt mit ihnen rechnen. Dazu folgendes Beispiel:

```
>>> 3.0e999 + 1.5e999999
inf
>>> 3.0e999 - 1.5e999999
nan
>>> 3.0e999 * 1.5e999999
inf
```

5 »inf« steht für *infinity* (dt. »unendlich«).

```
>>> 3.0e999 / 1.5e999999
nan
>>> 5 / 1e9999
0.0
```

Zwei unendlich große Gleitkommazahlen lassen sich problemlos addieren oder multiplizieren. Das Ergebnis ist in beiden Fällen wieder `inf`. Ein Problem gibt es aber, wenn versucht wird, zwei solche Zahlen zu subtrahieren bzw. zu dividieren. Da diese Rechenoperationen nicht sinnvoll sind, ergeben sie `nan`. Der Status `nan` ist vergleichbar mit `inf`, bedeutet jedoch »not a number«, also so viel wie »nicht berechenbar«.

Beachten Sie, dass weder `inf` noch `nan` eine Konstante ist, die Sie selbst in einem Python-Programm verwenden könnten. Stattdessen können Sie `float`-Instanzen mit den Werten `inf` und `nan` folgendermaßen erzeugen:

```
>>> float("inf")
inf
>>> float("nan")
nan
>>> float("inf") / float("inf")
nan
```

11.6 Boolesche Werte – bool

Eine Instanz des Datentyps `bool`[6] kann nur zwei verschiedene Werte annehmen: »wahr« oder »falsch« oder, um innerhalb der Python-Syntax zu bleiben, `True` oder `False`. Deshalb ist es auf den ersten Blick absurd, `bool` den numerischen Datentypen zuzuordnen. Wie in vielen Programmiersprachen üblich, wird in Python `True` analog zur `1` und `False` analog zur `0` gesehen, sodass sich mit booleschen Werten genauso rechnen lässt wie beispielsweise mit den ganzen Zahlen. Bei den Namen `True` und `False` handelt es sich um Konstanten, die im Quelltext verwendet werden können. Beachten Sie besonders, dass die Konstanten mit einem Großbuchstaben beginnen:

```
v1 = True
v2 = False
```

11.6.1 Logische Operatoren

Ein oder mehrere boolesche Werte lassen sich mithilfe bestimmter Operatoren zu einem booleschen Ausdruck kombinieren. Ein solcher Ausdruck resultiert, wenn er

6 Der Name `bool` geht zurück auf den britischen Mathematiker und Logiker George Boole (1815–1864).

ausgewertet wurde, wieder in einem booleschen Wert, also in `True` oder `False`. Bevor es zu theoretisch wird, folgt hier zunächst die Tabelle der sogenannten *logischen Operatoren*[7], und darunter sehen Sie weitere Erklärungen mit konkreten Beispielen.

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 11.6 Logische Operatoren des Datentyps bool

Logische Negierung

Die *logische Negierung* eines booleschen Wertes ist schnell erklärt: Der entsprechende Operator `not` macht `True` zu `False` und `False` zu `True`. In einem konkreten Beispiel würde das folgendermaßen aussehen:

```
if not x:
    print("x ist False")
else:
    print("x ist True")
```

Logisches UND

Das *logische UND* zwischen zwei Wahrheitswerten ergibt nur dann `True`, wenn beide Operanden bereits `True` sind. In Tabelle 11.7 sind alle möglichen Fälle aufgelistet.

x	y	x and y
`True`	`True`	`True`
`False`	`True`	`False`
`True`	`False`	`False`
`False`	`False`	`False`

Tabelle 11.7 Mögliche Fälle des logischen UND

7 Beachten Sie, dass es einen Unterschied gibt zwischen den logischen Operatoren, die im Zusammenhang mit booleschen Werten stehen, und den binären Operatoren, die sich auf die Binärdarstellung einer Zahl beziehen.

In einem konkreten Beispiel würde die Anwendung des logischen UND so aussehen:

```
if x and y:
    print("x und y sind True")
```

Logisches ODER

Das *logische ODER* zwischen zwei Wahrheitswerten ergibt genau dann eine wahre Aussage, wenn mindestens einer der beiden Operanden wahr ist. Es handelt sich demnach um ein nicht ausschließendes ODER. Ein Operator für ein logisches ausschließendes (exklusives) ODER existiert in Python nicht[8]. Tabelle 11.8 listet alle möglichen Fälle auf.

x	y	x or y
True	True	True
False	True	True
True	False	True
False	False	False

Tabelle 11.8 Mögliche Fälle des logischen ODER

Ein logisches ODER könnte folgendermaßen implementiert werden:

```
if x or y:
    print("x oder y ist True")
```

Selbstverständlich können Sie all diese Operatoren miteinander kombinieren und in einem komplexen Ausdruck verwenden. Das könnte etwa folgendermaßen aussehen:

```
if x and y or ((y and z) and not x):
    print("Holla die Waldfee")
```

Wir möchten diesen Ausdruck hier nicht im Einzelnen besprechen. Es sei nur gesagt, dass der Einsatz von Klammern den erwarteten Effekt hat, nämlich dass umklammerte Ausdrücke zuerst ausgewertet werden. Tabelle 11.9 zeigt den Wahrheitswert des Ausdrucks auf, und zwar in Abhängigkeit von den drei Parametern `x`, `y` und `z`.

8 Ein logisches exklusives ODER zwischen `x` und `y` lässt sich über `(x or y) and not (x and y)` nachbilden.

x	y	z	x and y or ((y and z) and not x)
True	True	True	True
False	True	True	True
True	False	True	False
True	True	False	True
False	False	True	False
False	True	False	False
True	False	False	False
False	False	False	False

Tabelle 11.9 Mögliche Ergebnisse des Ausdrucks

Die Kombination von logischen und vergleichenden Operatoren

Zu Beginn des Abschnitts über numerische Datentypen haben wir einige vergleichende Operatoren eingeführt, die eine Wahrheitsaussage in Form eines booleschen Wertes ergeben. Das folgende Beispiel zeigt, dass sie ganz selbstverständlich zusammen mit den logischen Operatoren verwendet werden können:

```
if x > y or (y > z and x != 0):
    print("Mein lieber Schwan")
```

In diesem Fall muss es sich bei `x`, `y` und `z` um Instanzen vergleichbarer Typen handeln, wie zum Beispiel `int`, `float` oder `bool`.

11.6.2 Wahrheitswerte nicht-boolescher Datentypen

Mithilfe der Built-in Function `bool` lassen sich Instanzen eines jeden Basisdatentyps in einen booleschen Wert überführen.

```
>>> bool([1,2,3])
True
>>> bool("")
False
>>> bool(-7)
True
```

Dies ist eine sinnvolle Eigenschaft, da sich eine Instanz der Basisdatentypen häufig in zwei Stadien befinden kann: »leer« und »nicht leer«. Oftmals möchte man beispielsweise testen, ob ein String Buchstaben enthält oder nicht. Da ein String in einen

booleschen Wert konvertiert werden kann, wird ein solcher Test sehr einfach durch logische Operatoren möglich:

```
>>> not ""
True
>>> not "abc"
False
```

Durch Verwendung eines logischen Operators wird der Operand automatisch als Wahrheitswert interpretiert.

Für jeden Basisdatentyp ist ein bestimmter Wert als `False` definiert. Alle davon abweichenden Werte sind `True`. Tabelle 11.10 listet für jeden Datentyp den entsprechenden `False`-Wert auf. Einige der Datentypen wurden noch nicht eingeführt, woran Sie sich an dieser Stelle jedoch nicht weiter stören sollten.

Basisdatentyp	False-Wert	Beschreibung
`NoneType`	`None`	der Wert `None`
Numerische Datentypen		
`int`	`0`	der numerische Wert Null
`float`	`0.0`	der numerische Wert Null
`bool`	`False`	der boolesche Wert `False`
`complex`	`0+0j`	der numerische Wert Null
Sequenzielle Datentypen		
`str`	`""`	ein leerer String
`list`	`[]`	eine leere Liste
`tuple`	`()`	ein leeres Tupel
Assoziative Datentypen		
`dict`	`{}`	ein leeres Dictionary
Mengen		
`set`	`set()`	eine leere Menge
`frozenset`	`frozenset()`	eine leere Menge

Tabelle 11.10 Wahrheitswerte der Basisdatentypen

Alle anderen Werte ergeben `True`.

11.6.3 Auswertung logischer Operatoren

Python wertet logische Ausdrücke grundsätzlich von links nach rechts aus, also im folgenden Beispiel zuerst `a` und dann `b`:

```
if a or b:
    print("a oder b sind True")
```

Es wird aber nicht garantiert, dass jeder Teil des Ausdrucks tatsächlich ausgewertet wird. Aus Optimierungsgründen bricht Python die Auswertung des Ausdrucks sofort ab, wenn das Ergebnis feststeht. Wenn im Beispiel oben also `a` bereits den Wert `True` hat, ist der Wert von `b` nicht weiter von Belang; `b` würde dann nicht mehr ausgewertet. Das folgende Beispiel demonstriert dieses Verhalten, das *Lazy Evaluation* (dt. »faule Auswertung«) genannt wird.

```
>>> a = True
>>> if a or print("Lazy "):
...     print("Evaluation")
...
Evaluation
```

Obwohl in der Bedingung der `if`-Anweisung die `print`-Funktion aufgerufen wird, wird diese Bildschirmausgabe nie durchgeführt, da der Wert der Bedingung bereits nach der Auswertung von `a` feststeht. Dieses Detail scheint unwichtig, kann aber insbesondere im Zusammenhang mit seiteneffektbehafteten[9] Funktionen zu schwer auffindbaren Fehlern führen.

In Abschnitt 11.6.1 wurde gesagt, dass ein boolescher Ausdruck stets einen booleschen Wert ergibt, wenn er ausgewertet wurde. Das ist nicht ganz korrekt, denn auch hier wurde die Arbeitsweise des Interpreters in einer Weise optimiert, über die man Bescheid wissen sollte. Deutlich wird dies an folgendem Beispiel aus dem interaktiven Modus:

```
>>> 0 or 1
1
```

Nach dem, was wir bisher besprochen haben, sollte das Ergebnis des Ausdrucks `True` sein, was nicht der Fall ist. Stattdessen gibt Python hier den ersten Operanden mit dem Wahrheitswert `True` zurück. In vielen Fällen macht das keinen Unterschied, denn der zurückgegebene Wert wird problemlos automatisch in den Wahrheitswert `True` überführt.

Die Auswertung der beiden Operatoren `or` und `and` läuft dabei folgendermaßen ab:

9 siehe dazu Abschnitt 17.10

Das logische ODER (`or`) nimmt den Wert des ersten Operanden an, der den Wahrheitswert `True` besitzt, oder – wenn es einen solchen nicht gibt – den Wert des letzten Operanden.

Das logische UND (`and`) nimmt den Wert des ersten Operanden an, der den Wahrheitswert `False` besitzt, oder – wenn es einen solchen nicht gibt – den Wert des letzten Operanden.

Diese Details haben dabei auch durchaus ihren unterhaltsamen Wert:

```
>>> "Python" or "Java"
'Python'
```

11.7 Komplexe Zahlen – complex

Überraschenderweise findet sich ein Datentyp zur Speicherung komplexer Zahlen unter Pythons Basisdatentypen. In vielen Programmiersprachen würden komplexe Zahlen eher eine Randnotiz in der Standardbibliothek darstellen oder ganz außen vor bleiben. Sollten Sie nicht mit komplexen Zahlen vertraut sein, können Sie diesen Abschnitt gefahrlos überspringen. Er behandelt nichts, was für das weitere Erlernen von Python vorausgesetzt würde.

Komplexe Zahlen bestehen aus einem reellen Realteil und einem Imaginärteil. Der Imaginärteil ist eine reelle Zahl, die mit der imaginären Einheit *j* multipliziert wird.[10] Die imaginäre Einheit *j* ist als Lösung der Gleichung

$j^2 = -1$

definiert. Im folgenden Beispiel weisen wir einer komplexen Zahl den Namen `v` zu:

```
v = 4j
```

Wenn man wie im Beispiel nur einen Imaginärteil angibt, wird der Realteil automatisch als `0` angenommen. Um den Realteil festzulegen, wird dieser zum Imaginärteil addiert. Die beiden folgenden Schreibweisen sind äquivalent:

```
v1 = 3 + 4j
v2 = 4j + 3
```

Anstelle des kleinen `j` ist auch ein großes `J` als Literal für den Imaginärteil einer komplexen Zahl zulässig. Entscheiden Sie hier ganz nach Ihren Vorlieben, welche der beiden Möglichkeiten Sie verwenden möchten.

10 Das in der Mathematik eigentlich übliche Symbol der imaginären Einheit ist *i*. Python hält sich hier an die Notationen der Elektrotechnik.

Sowohl der Real- als auch der Imaginärteil können eine beliebige reelle Zahl sein. Folgende Schreibweise ist demnach auch korrekt:

```
v3 = 3.4 + 4e2j
```

Zu Beginn des Abschnitts über numerische Datentypen wurde bereits angedeutet, dass sich komplexe Zahlen von den anderen numerischen Datentypen unterscheiden. Da für komplexe Zahlen keine mathematische Anordnung definiert ist, können Instanzen des Datentyps `complex` nur auf Gleichheit oder Ungleichheit überprüft werden. Die Menge der vergleichenden Operatoren ist also auf `==` und `!=` beschränkt.

Darüber hinaus haben sowohl der Modulo-Operator `%` als auch der Operator `//` für eine ganzzahlige Division im Komplexen keinen mathematischen Sinn und stehen deswegen in Kombination mit komplexen Zahlen nicht zur Verfügung.

Der Datentyp `complex` besitzt zwei Attribute, die die Arbeit mit ihm erleichtern. Es kommt zum Beispiel vor, dass man Berechnungen nur mit dem Realteil oder nur mit dem Imaginärteil der gespeicherten Zahl anstellen möchte. Um einen der beiden Teile zu isolieren, stellt eine `complex`-Instanz die in Tabelle 11.11 aufgeführten Attribute bereit.

Attribut	Beschreibung
`x.real`	Realteil von `x` als Gleitkommazahl
`x.imag`	Imaginärteil von `x` als Gleitkommazahl

Tabelle 11.11 Attribute des Datentyps complex

Diese können wie im folgenden Beispiel verwendet werden:

```
>>> c = 23 + 4j
>>> c.real
23.0
>>> c.imag
4.0
```

Außer über seine zwei Attribute verfügt der Datentyp `complex` über eine Methode, die in Tabelle 11.12 exemplarisch für eine Referenz auf eine komplexe Zahl namens `x` erklärt wird.

Methode	Beschreibung
`x.conjugate()`	Liefert die zu `x` konjugierte komplexe Zahl.

Tabelle 11.12 Methoden des Datentyps complex

Das folgende Beispiel demonstriert die Verwendung der Methode `conjugate`:

```
>>> c = 23 + 4j
>>> c.conjugate()
(23-4j)
```

Das Ergebnis von `conjugate` ist wieder eine komplexe Zahl und verfügt daher ebenfalls über die Methode `conjugate`:

```
>>> c = 23 + 4j
>>> c2 = c.conjugate()
>>> c2
(23-4j)
>>> c3 = c2.conjugate()
>>> c3
(23+4j)
```

Das Konjugieren einer komplexen Zahl ist eine selbstinverse Operation. Das bedeutet, dass das Ergebnis einer zweifachen Konjugation wieder die Ausgangszahl ist.

Kapitel 20
Ausnahmebehandlung

Stellen Sie sich einmal ein Programm vor, das über eine vergleichsweise tiefe Aufrufhierarchie verfügt. Das heißt, dass Funktionen weitere Unterfunktionen aufrufen, die ihrerseits wieder Funktionen aufrufen. Es ist häufig so, dass die übergeordneten Funktionen nicht korrekt weiterarbeiten können, wenn in einer ihrer Unterfunktionen ein Fehler aufgetreten ist. Die Information, dass ein Fehler aufgetreten ist, muss also durch die Aufrufhierarchie nach oben geschleust werden, damit jede übergeordnete Funktion auf den Fehler reagieren und sich daran anpassen kann.

20.1 Exceptions

Bislang konnten wir Fehler, die innerhalb einer Funktion aufgetreten sind, allein anhand des Rückgabewertes der Funktion kenntlich machen. Es ist mit viel Aufwand verbunden, einen solchen Rückgabewert durch die Funktionshierarchie nach oben durchzureichen, zumal es sich dabei um Ausnahmen handelt. Wir würden also sehr viel Code dafür aufwenden, um seltene Fälle zu behandeln.

Für solche Fälle unterstützt Python ein Programmierkonzept, das *Exception Handling* (dt. »Ausnahmebehandlung«) genannt wird. Im Fehlerfall erzeugt unsere 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.

Bisher haben wir bei einer solchen Ausgabe

```
>>> abc
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
NameError: name 'abc' is not defined
```

ganz allgemein von einem »Fehler« oder einer »Fehlermeldung« gesprochen. Dies ist nicht ganz korrekt: Im Folgenden möchten wir diese Ausgabe als *Traceback* bezeichnen. Welche Informationen ein Traceback enthält und wie sie interpretiert werden können, wurde bereits in Abschnitt 4.5 behandelt. Ein Traceback wird immer dann angezeigt, wenn eine Exception bis nach ganz oben durchgereicht wurde, ohne abgefangen zu werden. Doch was genau ist eine Exception?

Eine *Exception* ist ein Objekt, das Attribute und Methoden zur Klassifizierung und Bearbeitung eines Fehlers enthält. Einige dieser Informationen werden im Traceback angezeigt, so etwa die Beschreibung des Fehlers (`name 'abc' is not defined`). Eine Exception kann im Programm selbst abgefangen und behandelt werden, ohne dass der Benutzer etwas davon mitbekommt. Näheres zum Abfangen einer Exception erfahren Sie im weiteren Verlauf dieses Kapitels. Sollte eine Exception nicht abgefangen werden, wird sie in Form eines Tracebacks ausgegeben, und der Programmablauf wird beendet.

20.1.1 Eingebaute Exceptions

In Python existiert eine Reihe eingebauter Exceptions, zum Beispiel die bereits bekannten Exceptions `SyntaxError`, `NameError` oder `TypeError`. Solche Exceptions werden von Funktionen der Standardbibliothek oder vom Interpreter selbst geworfen. Sie sind eingebaut, das bedeutet, dass sie zu jeder Zeit im Quelltext verwendet werden können:

```
>>> NameError
<class 'NameError'>
>>> SyntaxError
<class 'SyntaxError'>
```

Die eingebauten Exceptions sind hierarchisch organisiert, das heißt, sie erben von gemeinsamen Basisklassen. Sie sind deswegen in ihrem Attribut- und Methodenumfang weitestgehend identisch. Im Anhang (in Abschnitt A.4) finden Sie eine Liste der eingebauten Exception-Typen mit kurzer Erklärung.

BaseException

Die Klasse `BaseException` ist die Basisklasse aller Exceptions und stellt damit eine Grundfunktionalität bereit, die für alle Exception-Typen vorhanden ist. Aus diesem Grund soll sie hier besprochen werden.

Die Grundfunktionalität, die `BaseException` bereitstellt, besteht aus einem wesentlichen Attribut namens `args`. Dabei handelt es sich um ein Tupel, in dem alle Parameter abgelegt werden, die der Exception bei ihrer Instanziierung übergeben wurden. Über diese Parameter ist es dann später beim Fangen der Exception möglich, detaillierte Informationen über den aufgetretenen Fehler zu erhalten. Das folgende Beispiel demonstriert nun die Verwendung des Attributs `args`:

```
>>> e = BaseException("Hallo Welt")
>>> e.args
('Hallo Welt',)
>>> e = BaseException("Hallo Welt",1,2,3,4,5)
>>> e.args
('Hallo Welt', 1, 2, 3, 4, 5)
```

So viel zunächst zur direkten Verwendung der Exception-Klassen.

20.1.2 Das Werfen einer Exception

Bisher haben wir nur Exceptions betrachtet, die in einem Fehlerfall vom Python-Interpreter geworfen wurden. Es ist jedoch auch möglich, mithilfe der `raise`-Anweisung selbst eine Exception zu werfen:

```
>>> raise SyntaxError("Hallo Welt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: Hallo Welt
```

Dazu wird das Schlüsselwort `raise` geschrieben, gefolgt von einer Instanz. Diese darf nur Instanz einer von `BaseException` abgeleiteten Klasse sein. Darüber hinaus ist auch das Werfen einer von `BaseException` abgeleiteten Klasse möglich, ohne zunächst eine Instanz zu erstellen. Eine auf diesem Wege geworfene Exception beinhaltet dann keine Fehlermeldung:

```
>>> raise SyntaxError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: None
```

Das Werfen von Instanzen anderer Datentypen, insbesondere von Strings, ist nicht möglich:

```
>>> raise "Hallo Welt"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exceptions must derive from BaseException
```

Im folgenden Abschnitt möchten wir besprechen, wie Exceptions im Programm abgefangen werden können, sodass sie nicht in einem Traceback enden, sondern zur Ausnahmebehandlung eingesetzt werden können. Wir werden sowohl in diesem als auch im nächsten Abschnitt bei den eingebauten Exceptions bleiben. Selbst definierte Exception-Typen werden das Thema von Abschnitt 20.1.4 sein.

20.1.3 Das Abfangen einer Exception

In diesem Abschnitt geht es darum, wie eine in einer Unterfunktion geworfene Exception in den darüberliegenden Aufrufebenen abgefangen werden kann. Das Fangen einer Exception ist notwendig, um auf den aufgetretenen Fehler reagieren zu können. Stellen Sie sich ein Programm vor, das Daten aus einer vom Benutzer festgelegten Datei liest. Dazu verwendet das Programm die folgende, im Moment noch sehr simple Funktion `get`, die das geöffnete Dateiobjekt zurückgibt:

```
def get(name):
    return open(name)
```

Sollte keine Datei mit dem angegebenen Namen existieren, wirft die eingebaute Funktion `open` eine `FileNotFoundError`-Exception. Da die Funktion `get` nicht auf diese Exception reagiert, wird sie in der Aufrufhierarchie weiter nach oben gereicht und verursacht schließlich ein vorzeitiges Beenden des Programms.

Nun sind fehlerhafte Benutzereingaben Probleme, die Sie beim Schreiben eines interaktiven Programms berücksichtigen sollten. Die folgende Variante der Funktion `get_file` fängt eine von `open` geworfene `FileNotFoundError`-Exception ab und gibt in diesem Fall anstelle des geöffneten Dateiobjekts den Wert `None` zurück:

```
def get_file(name):
    try:
        return open(name)
    except FileNotFoundError:
        return None
```

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 Codeblock, der um eine Ebene weiter eingerückt ist. Dieser Codeblock wird zunächst ausgeführt. Wenn in diesem Codeblock 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. Beachten Sie, dass mehrere Exception-Typen in Form eines Tupels angegeben werden müssen. Dazu werden Sie später noch ein Beispiel sehen. Hinter der Liste der Exception-Typen kann, ebenfalls optional, das Schlüsselwort as stehen, gefolgt von einem frei wählbaren Bezeichner. Hier legen Sie fest, unter welchem Namen Sie auf die gefangene Exception-Instanz im except-Zweig zugreifen können. Auf diesem Weg können Sie beispielsweise auf die in dem args-Attribut der Exception-Instanz abgelegten Informationen zugreifen. Auch dazu werden Sie im Verlauf dieses Kapitels noch Beispiele sehen.

  Danach folgen ein Doppelpunkt und, um eine Ebene weiter eingerückt, ein beliebiger Codeblock. Dieser Codeblock wird nur dann ausgeführt, wenn innerhalb des try-Blocks eine der aufgelisteten Exceptions geworfen wurde.

Eine grundlegende try/except-Anweisung hat also folgende Struktur:

```
try:
    Anweisung
    Anweisung
except ExceptionTyp as Bezeichner:
    Anweisung
    Anweisung
```

Kommen wir zurück zu unserer Beispielfunktion get_file. Es ist durchaus möglich, dass bei einem Funktionsaufruf für name fälschlicherweise kein String, sondern zum Beispiel eine Liste übergeben wird. In einem solchen Fall wird kein FileNotFoundError, sondern ein TypeError geworfen, der von der try/except-Anweisung bislang nicht abgefangen wird:

```
>>> get([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in get
TypeError: expected str, bytes or os.PathLike object, not list
```

Die Funktion soll nun dahingehend erweitert werden, dass auch ein TypeError abgefangen und dann ebenfalls None zurückgegeben wird. Dazu haben wir im Wesentlichen drei Möglichkeiten. Die erste besteht darin, die Liste der abzufangenden Exception-Typen im vorhandenen except-Zweig um den TypeError zu erweitern. Beachten Sie dabei, dass zwei oder mehr Exception-Typen im Kopf eines except-Zweiges als Tupel angegeben werden müssen:

```
def get(name):
    try:
        return open(name)
    except (FileNotFoundError, TypeError):
        return None
```

Dies ist einfach und führt im gewählten Beispiel zum gewünschten Resultat. Stellen Sie sich jedoch vor, Sie wollten je nach Exception-Typ unterschiedlichen Code ausführen. Um ein solches Verhalten zu erreichen, kann eine `try`/`except`-Anweisung über beliebig viele `except`-Zweige verfügen:

```
def get(name):
    try:
        return open(name)
    except FileNotFoundError:
        return None
    except TypeError:
        return None
```

Die dritte – weniger elegante – Möglichkeit besteht darin, alle Arten von Exceptions auf einmal abzufangen. Dazu wird ein `except`-Zweig ohne Angabe eines Exception-Typs geschrieben:

```
def get(name):
    try:
        return open(name)
    except:
        return None
```

Hinweis

Es ist nur in wenigen Fällen sinnvoll, alle möglichen Exceptions auf einmal abzufangen. Durch diese Art Exception Handling kann es vorkommen, dass unabsichtlich auch Exceptions abgefangen werden, die nichts mit dem oben dargestellten Code zu tun haben. Das betrifft zum Beispiel die `KeyInterrupt`-Exception, die bei einem Programmabbruch per Tastenkombination geworfen wird.

Sollten Sie einmal jede beliebige Exception fangen wollen, verwenden Sie `except Exception`, da `Exception` die Basisklasse aller Exceptions ist, die das Programm nicht zwingend beenden.

Eine Exception ist nichts anderes als eine Instanz einer bestimmten Klasse. Darum stellt sich die Frage, ob und wie man innerhalb eines `except`-Zweiges Zugriff auf die

geworfene Instanz erlangt. Das ist durch Angabe des bereits angesprochenen `as Bezeichner`-Teils im Kopf des `except`-Zweiges möglich. Unter dem dort angegebenen Namen können Sie nun innerhalb des Codeblocks auf die geworfene Exception-Instanz zugreifen:[1]

```
try:
    print([1,2,3][10])
except (IndexError, TypeError) as e:
    print("Fehlermeldung:", e.args[0])
```

Die Ausgabe des oben angeführten Beispiels lautet:

```
Fehlermeldung: list index out of range
```

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 Codeblock wird ausgeführt, wenn keine Exception aufgetreten ist, und der dem `finally`-Zweig zugehörige Codeblock 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.

Sowohl der `else`- als auch der `finally`-Zweig müssen ans Ende der `try`/`except`-Anweisung geschrieben werden. Wenn beide Zweige vorkommen, muss der `else`-Zweig vor dem `finally`-Zweig stehen.

Abbildung 20.1 zeigt eine vollständige `try`/`except`-Anweisung.

Abschließend noch einige Bemerkungen dazu, wie eine `try`/`except`-Anweisung ausgeführt wird: Zunächst wird der Code ausgeführt, der zum `try`-Zweig gehört. Sollte innerhalb dieses Codes eine Exception geworfen werden, wird der Code ausgeführt, der zu dem entsprechenden `except`-Zweig gehört. Ist kein passender `except`-Zweig vorhanden, wird die Exception nicht abgefangen und endet, wenn sie auch anderswo nicht abgefangen wird, als Traceback auf dem Bildschirm. Sollte im `try`-Zweig keine Exception geworfen werden, wird keiner der `except`-Zweige ausgeführt, sondern der `else`-Zweig. Der `finally`-Zweig wird in jedem Fall zum Schluss ausgeführt.

1 Die möglicherweise verwirrende Schreibweise `print([1,2,3][10])` ist gleichbedeutend mit:
```
lst = [1,2,3]
print(lst[10])
```

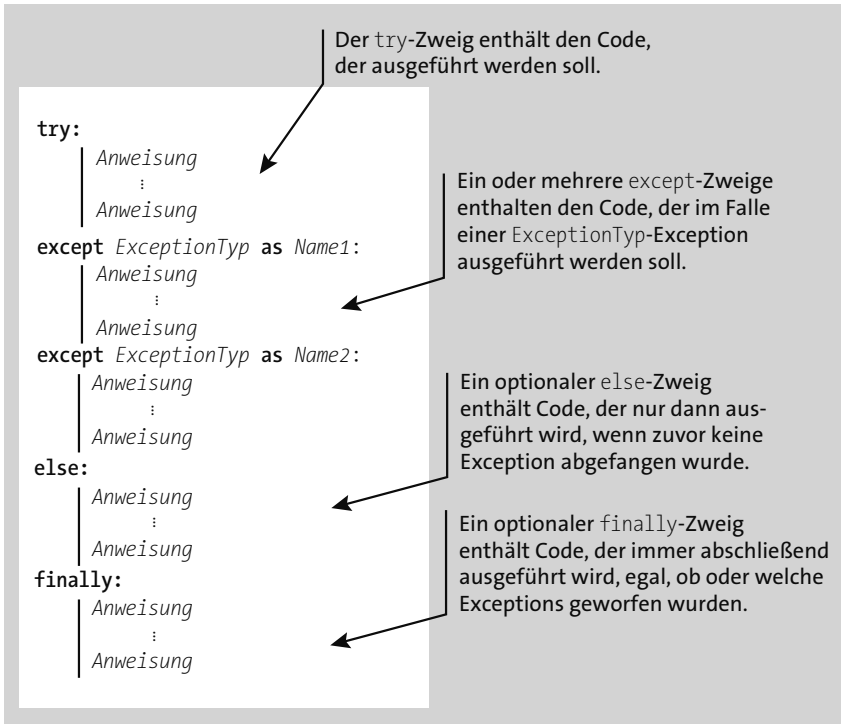

Abbildung 20.1 Eine vollständige try/except-Anweisung

Exceptions, die innerhalb eines `except`-, `else`- oder `finally`-Zweiges geworfen werden, können nicht von folgenden `except`-Zweigen der gleichen Anweisung wieder abgefangen werden. Es ist jedoch möglich, `try`/`except`-Anweisungen zu verschachteln:

```
try:
    try:
        raise TypeError
    except IndexError:
        print("Ein IndexError ist aufgetreten")
except TypeError:
    print("Ein TypeError ist aufgetreten")
```

Im `try`-Zweig der inneren `try`/`except`-Anweisung wird ein `TypeError` geworfen, der von der Anweisung selbst nicht abgefangen wird. Die Exception wandert dann, bildlich gesprochen, eine Ebene höher und durchläuft die nächste `try`/`except`-Anweisung. In dieser wird der geworfene `TypeError` abgefangen und eine entsprechende Meldung ausgegeben. Die Ausgabe des Beispiels lautet also: `Ein TypeError ist aufgetreten`, es wird kein Traceback angezeigt.

20.1.4 Eigene Exceptions

Beim Werfen und Abfangen von Exceptions sind Sie nicht auf den eingebauten Satz von Exception-Typen beschränkt, vielmehr können Sie selbst neue Typen erstellen. Viele Drittanbieterbibliotheken nutzen diese Möglichkeit, um speziell auf die jeweilige Anwendung zugeschnittene Exception-Typen anzubieten.

Zum Definieren eines eigenen Exception-Typs brauchen Sie lediglich eine eigene Klasse zu erstellen, die von der Exception-Basisklasse `Exception` erbt, und dann ganz nach Anforderung weitere Attribute und Methoden zum Umgang mit Ihrer Exception hinzuzufügen.

Im Folgenden definieren wir zunächst eine rudimentäre Kontoklasse, die als einzige Operation das Abheben eines bestimmten Geldbetrags unterstützt:

```
class Konto:
    def __init__(self, betrag):
        self.kontostand = betrag
    def abheben(self, betrag):
        self.kontostand -= betrag
```

In dieser Implementierung der Klasse ist es möglich, das Konto beliebig zu überziehen. In einer etwas raffinierteren Variante soll das Überziehen des Kontos unterbunden werden, und beim Versuch, mehr Geld abzuheben, als vorhanden ist, soll eine selbst definierte Exception geworfen werden. Dazu definieren wir zunächst eine von der Basisklasse `Exception` abgeleitete Klasse und fügen Attribute für den Kontostand und den abzuhebenden Betrag hinzu:

```
class KontostandException(Exception):
    def __init__(self, kontostand, betrag):
        super().__init__(kontostand, betrag)
        self.kontostand = kontostand
        self.betrag = betrag
```

Dann modifizieren wir die Methode `abheben` der Klasse `Konto` dahingehend, dass bei einem ungültigen Abhebevorgang eine `KontostandException`-Instanz geworfen wird:

```
class Konto:
    def __init__(self, betrag):
        self.kontostand = betrag
    def abheben(self, betrag):
        if betrag > self.kontostand:
            raise KontostandException(self.kontostand, betrag)
        self.kontostand -= betrag
```

Die dem Konstruktor der Klasse übergebenen zusätzlichen Informationen werden im Traceback nicht angezeigt:

```
>>> k = Konto(1000)
>>> k.abheben(2000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in abheben
KontostandException: (1000, 2000)
```

Sie kommen erst zum Tragen, wenn die Exception abgefangen und bearbeitet wird:

```
try:
    k.abheben(2000)
except KontostandException as e:
    print("Kontostand: {}€".format(e.kontostand))
    print("Abheben von {}€ nicht möglich.".format(e.betrag))
```

Dieser Code fängt die entstandene Exception ab und gibt daraufhin eine Fehlermeldung aus. Anhand der zusätzlichen Informationen, die die Klasse durch die Attribute `kontostand` und `betrag` bereitstellt, lässt sich der vorausgegangene Abhebevorgang rekonstruieren. Die Ausgabe des Beispiels lautet:

```
Kontostand: 1000€
Abheben von 2000€ nicht möglich.
```

Damit eine selbst definierte Exception mit weiterführenden Informationen auch eine Fehlermeldung enthalten kann, muss sie die Magic Method `__str__` implementieren:

```
class KontostandException(Exception):
    def __init__(self, kontostand, betrag):
        self.kontostand = kontostand
        self.betrag = betrag
    def __str__(self):
        return "Kontostand zu niedrig: Es werden {}€ mehr benötigt".format(
                    self.betrag - self.kontostand)
```

Ein Traceback, der durch diese Exception verursacht wird, sieht folgendermaßen aus:

```
>>> k = Konto(1000)
>>> k.abheben(2000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in abheben
KontostandException: Kontostand zu niedrig: Es werden 1000€ mehr benötigt
```

20.1.5 Erneutes Werfen einer Exception

In manchen Fällen, gerade bei einer tiefen Funktionshierarchie, ist es sinnvoll, eine Exception abzufangen, die für diesen Fall vorgesehene Fehlerbehandlung zu starten und die Exception danach erneut zu werfen. Betrachten wir dazu folgendes Beispiel:

```
def funktion3():
    raise TypeError
def funktion2():
    funktion3()
def funktion1():
    funktion2()
funktion1()
```

Im Beispiel wird die Funktion `funktion1` aufgerufen, die ihrerseits `funktion2` aufruft, in der die Funktion `funktion3` aufgerufen wird. Es handelt sich also um insgesamt drei verschachtelte Funktionsaufrufe. Im Innersten dieser Funktionsaufrufe, in `funktion3`, wird eine `TypeError`-Exception geworfen. Diese Exception wird nicht abgefangen, deshalb sieht der dazugehörige Traceback so aus:

```
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    funktion1()
  File "test.py", line 8, in funktion1
    return funktion2()
  File "test.py", line 5, in funktion2
    return funktion3()
  File "test.py", line 2, in funktion3
    raise TypeError
TypeError
```

Der Traceback beschreibt erwartungsgemäß die Funktionshierarchie zum Zeitpunkt der `raise`-Anweisung. Diese Liste wird auch *Callstack* genannt.

Hinter dem Exception-Prinzip steht der Gedanke, dass sich eine Exception in der Aufrufhierarchie nach oben arbeitet und an jeder Station abgefangen werden kann. In unserem Beispiel soll die Funktion `funktion1` die `TypeError`-Exception abfangen, damit sie eine spezielle, auf den `TypeError` zugeschnittene Fehlerbehandlung durchführen kann. Nachdem `funktion1` ihre funktionsinterne Fehlerbehandlung durchgeführt hat, soll die Exception weiter nach oben gereicht werden. Dazu wird sie erneut geworfen, und zwar wie im folgenden Beispiel:

```
def funktion3():
    raise TypeError
```

```
def funktion2():
    funktion3()
def funktion1():
    try:
        funktion2()
    except TypeError:
        # Fehlerbehandlung
        raise TypeError
funktion1()
```

Im Gegensatz zum vorangegangenen Beispiel sieht der nun auftretende Traceback so aus:

```
Traceback (most recent call last):
  File "test.py", line 14, in <module>
    funktion1()
  File "test.py", line 12, in funktion1
    raise TypeError
TypeError
```

Sie sehen, dass dieser Traceback Informationen über den Kontext der zweiten `raise`-Anweisung enthält.[2] Diese sind aber gar nicht von Belang, sondern eher ein Nebenprodukt der Fehlerbehandlung innerhalb der Funktion `funktion1`. Optimal wäre es, wenn trotz des temporären Abfangens der Exception in `funktion1` der resultierende Traceback den Kontext der ursprünglichen `raise`-Anweisung beschriebe. Um das zu erreichen, wird eine `raise`-Anweisung ohne Angabe eines Exception-Typs geschrieben:

```
def funktion3():
    raise TypeError
def funktion2():
    funktion3()
def funktion1():
    try:
        funktion2()
    except TypeError as e:
        # Fehlerbehandlung
        raise
funktion1()
```

2 Tatsächlich enthält der ausgegebene Traceback aufgrund des *Exception Chainings* (siehe Abschnitt 20.1.6) auch noch Informationen über die ursprüngliche Exception. Das soll uns an dieser Stelle aber nicht weiter interessieren.

Der in diesem Beispiel ausgegebene Traceback sieht folgendermaßen aus:

```
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    funktion1()
  File "test.py", line 11, in funktion1
    funktion2()
  File "test.py", line 7, in funktion2
    funktion3()
  File "test.py", line 4, in funktion3
    raise TypeError
TypeError
```

Sie sehen, dass es sich dabei um den Stacktrace der Stelle handelt, an der die Exception ursprünglich geworfen wurde. Der Traceback enthält damit die gewünschten Informationen über die Stelle, an der der Fehler tatsächlich aufgetreten ist.

20.1.6 Exception Chaining

Gelegentlich kommt es vor, dass man innerhalb eines `except`-Zweiges in die Verlegenheit kommt, eine weitere Exception zu werfen – entweder weil bei der Behandlung der Exception ein weiterer Fehler aufgetreten ist oder um die entstandene Exception »umzubenennen«.

Wenn innerhalb eines `except`-Zweiges eine weitere Exception geworfen wird, wendet Python automatisch das sogenannte *Exception Chaining* an. Dabei wird die vorangegangene Exception als Kontext an die neu geworfene Exception angehängt, sodass ein Maximum an Information weitergegeben wird. Zum Beispiel erzeugt der Code

```
try:
    [1,2,3][128]
except IndexError:
    raise RuntimeError("Schlimmer Fehler")
```

die Ausgabe:

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    [1,2,3][128]
IndexError: list index out of range
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
```

```
  File "test.py", line 5, in <module>
    raise RuntimeError("Schlimmer Fehler") from e
RuntimeError: Schlimmer Fehler
```

Es wird auf das 128. Element einer dreielementigen Liste zugegriffen, was eine `IndexError`-Exception provoziert. Diese Exception wird gefangen, und bei der Behandlung wird eine `RuntimeError`-Exception geworfen. Anhand des ausgegebenen Tracebacks sehen Sie, dass die ursprüngliche `IndexError`-Exception an die neue `RuntimeError`-Exception angehängt wurde.

Mithilfe der `raise`/`from`-Syntax lässt sich das Exception-Chaining-Verhalten steuern. Beim Werfen einer Exception kann ein Kontext angegeben werden, der dann im resultierenden Traceback berücksichtigt wird. Dieser Kontext kann zum Beispiel eine zweite Exception sein:

```
>>> raise IndexError from ValueError
ValueError
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError
```

Es zeigt sich, dass wir mit der `raise`/`from`-Syntax das Exception Chaining auslösen können. Alternativ kann mit der `raise`/`from`-Syntax das automatische Anhängen einer Exception verhindert werden:

```
try:
    [1,2,3][128]
except IndexError:
    raise RuntimeError("Schlimmer Fehler") from None
```

In diesem Fall enthält der resultierende Traceback nur die neu entstandene `RuntimeError`-Exception. Die ursprüngliche `IndexError`-Exception geht verloren.

20.2 Zusicherungen – assert

Mithilfe des Schlüsselworts `assert` lassen sich Zusicherungen in ein Python-Programm integrieren. Durch das Schreiben einer `assert`-Anweisung legt der Programmierer eine Bedingung fest, die für die Ausführung des Programms essenziell ist und die bei Erreichen der `assert`-Anweisung zu jeder Zeit `True` ergeben muss. Wenn die Bedingung einer `assert`-Anweisung `False` ergibt, wird eine `AssertionError`-Exception geworfen. In der folgenden Sitzung im interaktiven Modus wurden mehrere `assert`-Anweisungen eingegeben:

```
>>> lst = [7, 1, 3, 5, -12]
>>> assert max(lst) == 7
>>> assert min(lst) == -12
>>> assert sum(lst) == 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

In der `assert`-Anweisung kann auch eine Fehlermeldung spezifiziert werden, die im Falle eines Fehlschlags in die `AssertionError`-Exception eingetragen wird. Diese Fehlermeldung kann, durch ein Komma getrennt, hinter die Bedingung geschrieben werden:

```
>>> assert max(lst) == 7, "max ist kaputt"
>>> assert min(lst) == -12, "min ist kaputt"
>>> assert sum(lst) == 0, "sum ist kaputt"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: sum ist kaputt
```

Die `assert`-Anweisung ist damit ein praktisches Hilfsmittel zum Aufspüren von Fehlern und ermöglicht es, den Programmlauf zu beenden, wenn bestimmte Voraussetzungen nicht erfüllt sind. Häufig prüft man an Schlüsselstellen im Programm mit `assert`, ob alle Referenzen die erwarteten Werte referenzieren, um eventuelle Fehlberechnungen rechtzeitig erkennen zu können.

Beachten Sie, dass `assert`-Anweisungen üblicherweise nur während der Entwicklung eines Programms benötigt werden und in einem fertigen Programm eher stören würden. Deswegen werden `assert`-Anweisungen nur dann ausgeführt, wenn die globale Konstante `__debug__` den Wert `True` referenziert. Diese Konstante ist nur dann `False`, wenn der Interpreter mit der Kommandozeilenoption `-O` gestartet wurde. Wenn die Konstante `__debug__` den Wert `False` referenziert, werden `assert`-Anweisungen ignoriert und haben damit keinen Einfluss mehr auf die Laufzeit Ihres Programms.

Hinweis

Beachten Sie, dass Sie den Wert von `__debug__` im Programm selbst nicht verändern dürfen, sondern nur über die Kommandozeilenoption `-O` bestimmen können, ob `assert`-Anweisungen ausgeführt oder ignoriert werden sollen.

20.3 Warnungen

Unter einer *Warnung* wird eine Exception verstanden, die den Programmablauf nicht verändert, sondern nur auf dem Standardfehlerstrom `stderr` (siehe Abschnitt 29.2.3) erscheint, um Sie über einen bedenklichen, aber nicht kritischen Umstand zu informieren.

Ein typisches Beispiel für eine Warnung ist die `DeprecationWarning`, die den Entwickler oder die Anwenderin darüber informiert, dass das laufende Programm eine Funktionalität verwendet, die in zukünftigen Python-Versionen oder zukünftigen Versionen einer Bibliothek nicht mehr zur Verfügung stehen wird. Diese Feststellung stellt für den aktuellen Programmlauf kein Problem dar, ist jedoch wichtig genug, um darüber zu informieren.

Hinweis

Abschnitt A.4 im Anhang listet die in Python standardmäßig definierten Typen von Warnungen auf und erklärt ihre Bedeutung.

Das Modul `warnings` der Standardbibliothek ermöglicht es, über komplexe Filterregeln das Anzeigen bzw. Ignorieren von Warnungen verschiedenen Inhalts und verschiedener Quellen zu steuern. Standardmäßig unterdrückt Python einige Warnungen, insbesondere wenn sie aus importierten Modulen stammen. Diese Filterregeln werden von den Python-Entwicklern jedoch häufig an neue Gegebenheiten angepasst.

Das Modul `warnings` enthält die Funktion `simplefilter`, die die voreingestellten Filterregeln mit einer allgemeinen Regel überschreiben kann. Auf diese Weise lassen sich Warnungen beispielsweise universell unterdrücken:

```
>>> import warnings
>>> warnings.simplefilter("ignore")
```

Analog können alle Warnungen zu Exceptions gemacht werden, die den Programmablauf unterbrechen. In diesem Fall können Warnungen auch gefangen und behandelt werden:

```
>>> warnings.simplefilter("error")
```

Weitere mögliche Argumente sind `"default"` für das Unterdrücken von erneut auftretenden Warnungen aus derselben Quelle, `"always"` für das Ausgeben aller Warnungen, `"module"` für das Ausgeben nur der jeweils ersten Warnung eines Moduls und `"once"` für das Unterdrücken von erneut auftretenden Warnungstypen.

Hinweis

Warnungen können auch über den Kommandozeilenparameter `-W` des Python-Interpreters zu Fehlern gemacht werden. Auf diese Weise lässt sich das Verhalten eines Python-Programms in Bezug auf Warnungen verändern, ohne den Code anpassen zu müssen:

```
$ python -W error programm.py
```

Analog sind die Argumente `default`, `always`, `module` und `once` möglich.

Kapitel 27
Bildschirmausgaben und Logging

An dieser Stelle möchten wir uns mit Modulen der Standardbibliothek befassen, die die Möglichkeiten der Bildschirmausgabe sinnvoll ergänzen. Dabei handelt es sich um das Modul `pprint` zur übersichtlich formatierten Ausgabe komplexer Instanzen, das wir auch schon in Abschnitt 3.10 kurz verwendet haben, sowie um das Modul `logging` zur formatierten Ausgabe von Log-Nachrichten auf dem Bildschirm oder in Logdateien.

Die in diesem Kapitel besprochenen Module verstehen sich als Ergänzung zur normalerweise verwendeten Built-in Function `print`, die ausführlich in Abschnitt 17.14.36 beschrieben wird.

27.1 Übersichtliche Ausgabe komplexer Objekte – pprint

In der Standardbibliothek existiert das Modul `pprint` (für *pretty print*), das für eine übersichtlich formatierte Repräsentation eines Python-Datentyps auf dem Bildschirm verwendet werden kann. Das Modul macht insbesondere die Ausgabe komplexer Datentypen, zum Beispiel langer Listen, besser lesbar. Bevor Beispiele ausgeführt werden können, muss das Modul eingebunden werden:

```
>>> import pprint
```

Das Modul `pprint` enthält im Wesentlichen eine gleichnamige Funktion, die zur Ausgabe einer Instanz aufgerufen werden kann.

pprint(object, [stream, indent, width, depth], {compact})

Die Funktion `pprint` gibt die Instanz `object`, formatiert auf dem Stream `stream`, aus. Wenn Sie den Parameter `stream` nicht übergeben, wird in den Standardausgabestrom `sys.stdout` geschrieben. Über die Parameter `indent`, `width` und `depth` lässt sich die Formatierung der Ausgabe steuern. Dabei kann für `indent` die Anzahl der Leerzeichen übergeben werden, die für eine Einrückung verwendet werden sollen. Der Parameter `indent` ist mit `1` vorbelegt.

Über den optionalen Parameter `width` kann die maximale Anzahl an Zeichen angegeben werden, die die Ausgabe breit sein darf. Dieser Parameter ist mit 80 Zeichen vorbelegt.

Im folgenden Beispiel erzeugen wir zunächst mit einer willkürlichen Methode unserer Wahl eine Liste von Strings und geben diese mithilfe von `pprint` formatiert aus:

```
>>> strings = [f"Der Wert von {i}**2 ist {i**2}" for i in range(10)]
>>> pprint.pprint(strings)
['Der Wert von 0**2 ist 0',
 'Der Wert von 1**2 ist 1',
 'Der Wert von 2**2 ist 4',
 'Der Wert von 3**2 ist 9',
 'Der Wert von 4**2 ist 16',
 'Der Wert von 5**2 ist 25',
 'Der Wert von 6**2 ist 36',
 'Der Wert von 7**2 ist 49',
 'Der Wert von 8**2 ist 64',
 'Der Wert von 9**2 ist 81']
```

Zum Vergleich geben wir `strings` noch einmal unformatiert mit `print` aus:

```
>>> print(strings)
['Der Wert von 0**2 ist 0', 'Der Wert von 1**2 ist 1', 'Der Wert von 2**2 ist
4', 'Der Wert von 3**2 ist 9', 'Der Wert von 4**2 ist 16', 'Der Wert von 5**2
ist 25', 'Der Wert von 6**2 ist 36', 'Der Wert von 7**2 ist 49', 'Der Wert von
8**2 ist 64', 'Der Wert von 9**2 ist 81']
```

Der Parameter `depth` ist eine ganze Zahl und bestimmt, bis zu welcher Tiefe Unterinstanzen, beispielsweise also verschachtelte Listen, ausgegeben werden sollen. Falls für `depth` ein anderer Wert als `None` übergeben wird, deutet `pprint` tiefer verschachtelte Elemente durch drei Punkte ... an.

Über den Schlüsselwortparameter `compact` lässt sich steuern, wie kompakt umfangreiche Strukturen (z. B. lange Listen) dargestellt werden. Wird hier `True` übergeben, wird beispielsweise nicht jedes Element von `strings` in eine eigene Zeile geschrieben.

Sollten Sie die Ausgabe von `pprint` weiterverarbeiten wollen, verwenden Sie die Funktion `pformat`, die die formatierte Repräsentation in Form eines Strings zurückgibt:

```
>>> s = pprint.pformat(strings)
>>> print(s)
['Der Wert von 0**2 ist 0',
 'Der Wert von 1**2 ist 1',
 'Der Wert von 2**2 ist 4',
 'Der Wert von 3**2 ist 9',
 'Der Wert von 4**2 ist 16',
 'Der Wert von 5**2 ist 25',
 'Der Wert von 6**2 ist 36',
 'Der Wert von 7**2 ist 49',
 'Der Wert von 8**2 ist 64',
 'Der Wert von 9**2 ist 81']
```

Die Funktion `pformat` hat die gleiche Schnittstelle wie `pprint` – mit dem Unterschied, dass der Parameter `stream` fehlt.

27.2 Logdateien – logging

Das Modul `logging` stellt ein flexibles Interface zum Protokollieren des Programmablaufs bereit. Protokolliert wird der Programmablauf, indem an verschiedenen Stellen im Programm Meldungen an das `logging`-Modul abgesetzt werden. Diese Meldungen können unterschiedliche Dringlichkeitsstufen haben. So gibt es beispielsweise Fehlermeldungen, Warnungen oder Debug-Informationen. Das Modul `logging` kann diese Meldungen auf vielfältige Weise verarbeiten. Üblich ist es, die Meldung mit einem Zeitstempel zu versehen und entweder auf dem Bildschirm auszugeben oder in eine Datei zu schreiben.

In diesem Abschnitt wird die Verwendung des Moduls `logging` anhand mehrerer Beispiele im interaktiven Modus gezeigt. Um die Beispielprogramme korrekt ausführen zu können, muss zuvor das Modul `logging` eingebunden sein:

```
>>> import logging
```

Bevor Meldungen an den *Logger* geschickt werden können, muss dieser durch Aufruf der Funktion `basicConfig` initialisiert werden. Im folgenden Beispiel wird ein Logger eingerichtet, der alle eingehenden Meldungen in die Logdatei `programm.log` schreibt:

```
>>> logging.basicConfig(filename="programm.log")
```

Jetzt können mithilfe der im Modul enthaltenen Funktion `log` Meldungen an den Logger übergeben werden. Die Funktion `log` bekommt dabei die Dringlichkeitsstufe der Meldung als ersten und die Meldung selbst in Form eines Strings als zweiten Parameter übergeben:

```
>>> logging.log(logging.ERROR, "Ein Fehler ist aufgetreten")
>>> logging.log(logging.INFO, "Dies ist eine Information")
```

Durch das Aufrufen der Funktion `shutdown` wird der Logger korrekt deinitialisiert, und eventuell noch anstehende Schreiboperationen werden durchgeführt:

```
>>> logging.shutdown()
```

Natürlich sind nicht nur die Dringlichkeitsstufen `ERROR` und `INFO` verfügbar. Tabelle 27.1 listet alle vordefinierten Stufen auf, aus denen Sie wählen können. Die Tabelle ist dabei nach Dringlichkeit geordnet, wobei die dringendste Stufe zuletzt aufgeführt wird.

Level	Beschreibung
NOTSET	keine Dringlichkeitsstufe
DEBUG	eine Meldung, die nur für den Programmierer zur Fehlersuche interessant ist
INFO	eine Informationsmeldung über den Programmstatus
WARNING	eine Warnmeldung, die auf einen möglichen Fehler hinweist
ERROR	eine Fehlermeldung, nach der das Programm weiterarbeiten kann
CRITICAL	eine Meldung über einen kritischen Fehler, der das sofortige Beenden des Programms oder der aktuell durchgeführten Operation zur Folge hat

Tabelle 27.1 Vordefinierte Dringlichkeitsstufen

Aus Gründen des Komforts existiert zu jeder Dringlichkeitsstufe eine eigene Funktion. So sind die beiden Funktionsaufrufe von `log` aus dem letzten Beispiel äquivalent zu:

```
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
```

Wenn Sie sich die Logdatei nach dem Aufruf dieser beiden Funktionen ansehen, werden Sie feststellen, dass es lediglich einen einzigen Eintrag gibt:

```
ERROR:root:Ein Fehler ist aufgetreten
```

Das liegt daran, dass der Logger in seiner Basiskonfiguration nur Meldungen loggt, deren Dringlichkeit größer oder gleich der einer Warnung ist. Um auch Debug- und Info-Meldungen mitzuloggen, müssen Sie beim Aufruf der Funktion `basicConfig` im Schlüsselwortparameter `level` einen geeigneten Wert übergeben:

```
logging.basicConfig(
    filename="programm.log",
    level=logging.DEBUG)
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
```

In diesem Beispiel wurde die Mindestdringlichkeit auf `DEBUG` gesetzt. Das bedeutet, dass alle Meldungen geloggt werden, die mindestens eine Dringlichkeit von `DEBUG` haben. Folglich erscheinen auch beide Meldungen in der Logdatei:

```
ERROR:root:Ein Fehler ist aufgetreten
INFO:root:Dies ist eine Information
```

Tabelle 27.2 listet die wichtigsten Schlüsselwortparameter auf, die der Funktion `basicConfig` übergeben werden können.

Parameter	Beschreibung
`datefmt`	Spezifiziert das Datumsformat. Näheres dazu erfahren Sie im folgenden Abschnitt.
`filemode`	Gibt den Modus* an, in dem die Logdatei geöffnet werden soll (Standardwert: `"a"`).
`filename`	Gibt den Dateinamen der Logdatei an.
`format`	Spezifiziert das Meldungsformat. Näheres dazu erfahren Sie im folgenden Abschnitt.
`handlers`	Gibt eine Liste von Handlern an, die registriert werden sollen. Näheres dazu erfahren Sie in Abschnitt 27.2.2.
`level`	Legt die Mindestdringlichkeit für Meldungen fest, damit diese in der Logdatei erscheinen.
`stream`	Gibt einen Stream an, in den die Logmeldungen geschrieben werden sollen. Wenn die Parameter `stream` und `filename` gemeinsam angegeben werden, wird `stream` ignoriert.
`style`	Bestimmt die Formatierungssyntax für die Meldung. Der voreingestellte Wert `"%"` bedingt die alte %-Syntax aus Python 2, während ein Wert von `"{"` die neue Syntax zur String-Formatierung** erzwingt.

* Die verschiedenen Modi, in denen Dateien geöffnet werden können, sind in Abschnitt 6.4 aufgeführt.

** Näheres zur String-Formatierung erfahren Sie in Abschnitt 12.5.3.

Tabelle 27.2 Schlüsselwortparameter der Funktion basicConfig

27.2.1 Das Meldungsformat anpassen

Wie in den vorangegangenen Beispielen zu sehen war, wird ein Eintrag in einer Logdatei standardmäßig nicht mit einem Zeitstempel versehen. Es gibt eine Möglichkeit, das Format der geloggten Meldung anzupassen. Dazu übergeben Sie beim Funktionsaufruf von `basicConfig` den Schlüsselwortparameter `format`:

```
logging.basicConfig(
    filename="programm.log",
    level=logging.DEBUG,
    style="{",
    format="{asctime} [{levelname:8}] {message}")
```

```
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
logging.error("Und schon wieder ein Fehler")
```

Sie sehen, dass ein Format-String übergeben wurde, der die Vorlage für eine Meldung enthält, wie sie später in der Logdatei stehen soll. Dabei stehen die Bezeichner `asctime` für den Timestamp, `levelname` für die Dringlichkeitsstufe und `message` für die Meldung. Die von diesem Beispiel generierten Meldungen sehen folgendermaßen aus:

```
2020-02-05 14:28:55,811 [ERROR   ] Ein Fehler ist aufgetreten
2020-02-05 14:29:00,690 [INFO    ] Dies ist eine Information
2020-02-05 14:29:12,686 [ERROR   ] Und schon wieder ein Fehler
```

Tabelle 27.3 listet die wichtigsten Bezeichner auf, die innerhalb des `format`-Format-Strings verwendet werden dürfen. Je nach Kontext, in dem die Meldung erzeugt wird, haben einige der Bezeichner keine Bedeutung.

Bezeichner	Beschreibung
`asctime`	Zeitpunkt der Meldung. Das Datums- und Zeitformat kann beim Funktionsaufruf von `basicConfig` über den Parameter `datefmt` angegeben werden. Näheres dazu folgt im Anschluss an diese Tabelle.
`filename`	der Dateiname der Programmdatei, in der die Meldung abgesetzt wurde
`funcName`	der Name der Funktion, in der die Meldung abgesetzt wurde
`levelname`	die Dringlichkeitsstufe der Meldung
`lineno`	die Quellcodezeile, in der die Meldung abgesetzt wurde
`message`	der Text der Meldung
`module`	Der Name des Moduls, in dem die Meldung abgesetzt wurde. Der Modulname entspricht dem Dateinamen ohne Dateiendung.
`pathname`	der Pfad zur Programmdatei, in der die Meldung abgesetzt wurde
`process`	die ID des Prozesses, in dem die Meldung abgesetzt wurde
`thread`	die ID des Threads, in dem die Meldung abgesetzt wurde

Tabelle 27.3 Bezeichner im Format-String

Es ist möglich, das Format anzupassen, in dem Zeitstempel ausgegeben werden. Beispielsweise können wir ein in Deutschland übliches Datumsformat setzen und außerdem die Ausgabe der Milisekundenanteile abschalten. Das Format des Timestamps

kann beim Aufruf von `basicConfig` über den Schlüsselwortparameter `datefmt` angegeben werden:

```
logging.basicConfig(
    filename="programm.log",
    level=logging.DEBUG,
    style="{",
    format="{asctime} [{levelname:8}] {message}",
    datefmt="%d.%m.%Y %H:%M:%S")
logging.error("Ein Fehler ist aufgetreten")
```

Die in der Vorlage für das Datumsformat verwendeten Platzhalter wurden in Abschnitt 15.1 eingeführt. Die von diesem Beispiel erzeugte Meldung sieht folgendermaßen aus:

```
05.02.2020 14:38:49 [ERROR   ] Ein Fehler ist aufgetreten
```

27.2.2 Logging-Handler

Bisher haben wir ausschließlich besprochen, wie das Modul `logging` dazu verwendet werden kann, alle eingehenden Meldungen in eine Datei zu schreiben. Tatsächlich ist das Modul in dieser Beziehung sehr flexibel und erlaubt es, nicht nur in Dateien, sondern beispielsweise auch in Streams zu schreiben oder die Meldungen über eine Netzwerkverbindung zu schicken. Dafür werden sogenannte *Logging-Handler* verwendet. Um genau zu sein, haben wir in den vorangegangenen Abschnitten bereits einen impliziten Handler verwendet, ohne uns darüber im Klaren zu sein.

Um einen speziellen Handler einzurichten, muss eine Instanz der Handler-Klasse erzeugt werden. Diese kann dann vom Logger verwendet werden. Im folgenden Beispiel sollen alle Meldungen auf einen Stream, nämlich `sys.stdout`, geschrieben werden; dazu wird die Handler-Klasse `logging.StreamHandler` verwendet:

```
import logging
import sys
handler = logging.StreamHandler(sys.stdout)
frm = logging.Formatter("{asctime} {levelname}: {message}",
                        "%d.%m.%Y %H:%M:%S", style="{")
handler.setFormatter(frm)
logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)
logger.critical("Ein wirklich kritischer Fehler")
logger.warning("Und eine Warnung hinterher")
logger.info("Dies hingegen ist nur eine Info")
```

Zunächst wird der Handler, in diesem Fall ein `StreamHandler`, instanziiert. Im nächsten Schritt wird eine Instanz der Klasse `Formatter` erzeugt. Diese Klasse kapselt die Formatierungsanweisungen, die wir in den vorangegangenen Beispielen beim Aufruf der Funktion `basicConfig` übergeben haben. Mithilfe der Methode `setFormatter` werden dem Handler die Formatierungsanweisungen bekannt gegeben.

Um den Handler beim Logger zu registrieren, benötigen wir Zugriff auf die bisher implizit verwendete Logger-Instanz. Diesen Zugriff erlangen wir über die Funktion `getLogger`. Danach wird über `addHandler` der Handler hinzugefügt und über `setLevel` die gewünschte Dringlichkeitsstufe eingestellt.

Die Meldungen werden im Folgenden nicht über Funktionen des Moduls `logging`, sondern über die Methoden `critical`, `warning` und `info` der `Logger`-Instanz `logger` abgesetzt. Das Beispielprogramm gibt folgenden Text auf dem Bildschirm aus:

```
05.02.2020 17:21:46 CRITICAL: Ein wirklich kritischer Fehler
05.02.2020 17:21:46 WARNING: Und eine Warnung hinterher
05.02.2020 17:21:46 INFO: Dies hingegen ist nur eine Info
```

Im Folgenden sollen die wichtigsten zusätzlichen Handler-Klassen beschrieben werden, die im Paket `logging` bzw. `logging.handlers` enthalten sind.

logging.FileHandler(filename, [mode, encoding, delay])

Dieser Handler schreibt die Logeinträge in die Datei `filename`. Dabei wird die Datei im Modus `mode` geöffnet. Der Handler `FileHandler` kann auch implizit durch Angabe der Schlüsselwortparameter `filename` und `filemode` beim Aufruf der Funktion `basicConfig` verwendet werden.

Der Parameter `encoding` kann dazu verwendet werden, das zum Schreiben der Datei verwendete Encoding festzulegen. Wenn Sie für den `delay`-Parameter `True` übergeben, wird mit dem Öffnen der Datei so lange gewartet, bis tatsächlich Daten geschrieben werden sollen.

logging.StreamHandler([stream])

Dieser Handler schreibt die Logeinträge in den Stream `stream`. Beachten Sie, dass der Handler `StreamHandler` auch implizit durch Angabe des Schlüsselwortparameters `stream` beim Aufruf der Funktion `basicConfig` verwendet werden kann.

logging.handlers.SocketHandler(host, port) logging.handlers.DatagramHandler(host, port)

Diese Handler senden die Logeinträge über eine TCP-Schnittstelle (`SocketHandler`) bzw. über eine UDP-Netzwerkschnittstelle (`DatagramHandler`) an den Rechner mit dem Hostnamen `host` unter Verwendung des Ports `port`.

logging.handlers.SMTPHandler(mailhost, from, to, subject, [credentials])

Dieser Handler sendet die Logeinträge als E-Mail an die Adresse `to`. Dabei werden `subject` als Betreff und `from` als Absenderadresse eingetragen. Über den Parameter `mailhost` geben Sie den zu verwendenden SMTP-Server an. Sollte dieser Server eine Authentifizierung verlangen, können Sie ein Tupel, das Benutzername und Passwort enthält, für den optionalen letzten Parameter `credentials` übergeben.