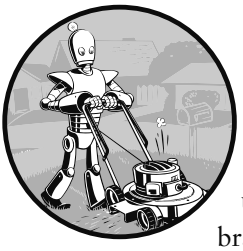
1

Grundlagen von Python



Die Programmiersprache Python bietet eine breite Palette von syntaktischen Konstruktionen, Standardbibliotheksfunktionen und Möglichkeiten zur interaktiven Entwicklung. Zum Glück brauchen Sie sich um das meiste davon nicht zu kümmern, sondern müssen nur so viel lernen, dass Sie damit praktische kleine Programme schreiben können.

Allerdings müssen Sie, bevor Sie irgendetwas tun können, zunächst einige Grundlagen der Programmierung erlernen. Wie ein Zauberlehrling werden Sie vielleicht denken, dass einige dieser Grundlagen ziemlich undurchsichtig sind und dass es viel Mühe macht, sie sich anzueignen, aber diese Kenntnisse und etwas Übung werden Sie in die Lage versetzen, Ihren Computer wie einen Zauberstab zu nutzen und damit unglaublich erscheinende Dinge zu tun.

In einigen Beispielen in diesem Kapitel werden Sie dazu aufgefordert, etwas in die *interaktive Shell,* auch *REPL* (Read-Evaluate-Print Loop, also etwa »Lesen-Auswerten-Ausgeben-Schleife«) genannt, einzugeben. Damit können Sie eine Python-Anweisung nach der anderen ausführen und die Ergebnisse unmittelbar

einsehen. Die Verwendung dieser Shell eignet sich hervorragend, um zu lernen, was die grundlegenden Python-Anweisungen bewirken. Nutzen Sie sie daher, während Sie das Buch durcharbeiten. Auf diese Weise können Sie sich den Stoff besser merken, als wenn Sie ihn nur lesen würden.

Ausdrücke in die interaktive Shell eingeben

Um die interaktive Shell auszuführen, können Sie den Editor Mu starten, den Sie beim Durcharbeiten der Installationsanleitungen im Vorwort heruntergeladen haben. Auf Windows öffnen Sie dazu das Startmenü, geben **Mu** ein und starten die gleichnamige Anwendung. Auf macOS öffnen Sie den Ordner *Programme* und doppelklicken darin auf *Mu*. Klicken Sie auf die Schaltfläche *New* und speichern Sie die leere Datei als *blank.py*. Wenn Sie diese leere Datei ausführen, indem Sie auf *Run* klicken oder F5 drücken, wird die interaktive Shell als neuer Bereich am unteren Rand des Mu-Fensters geöffnet. Dort sehen Sie die Eingabeaufforderung >>> der Shell.

Geben Sie dort **2 + 2** ein, um Python eine einfache Berechnung ausführen zu lassen. Das Mu-Fenster zeigt jetzt Folgendes an:

```
>>> 2 + 2
4
>>>
```

In Python wird etwas wie `2 + 2` als *Ausdruck* bezeichnet. Dies ist die einfachste Form von Programmieranweisungen in dieser Sprache. Ausdrücke setzen sich aus *Werten* (wie `2`) und *Operatoren* (wie `+`) zusammen. Sie können stets *ausgewertet*, also auf einen einzigen Wert reduziert werden. Daher können Sie an allen Stellen im Python-Code, an denen ein Wert stehen soll, auch einen Ausdruck verwenden.

Im vorstehenden Beispiel wurde `2 + 2` zu dem Wert `4` ausgewertet. Ein einzelner Wert ohne Operatoren wird ebenfalls als Ausdruck angesehen, wird aber nur zu sich selbst ausgewertet:

```
>>> 2
2
```

Fehler sind kein Beinbruch

Wenn ein Programm Code enthält, den der Computer nicht versteht, stürzt es ab, woraufhin Python eine Fehlermeldung anzeigt. Ihren Computer können Sie dadurch jedoch nicht beschädigen. Daher brauchen Sie auch keine Angst vor Fehlern zu haben. Bei einem Absturz hält das Programm nur unerwartet an.

Wenn Sie mehr über eine bestimmte Fehlermeldung wissen wollen, können Sie online nach dem genauen Text suchen. Auf *www.dpunkt.de/python_automatisieren_2/* finden Sie außerdem eine Liste häufig auftretender Python-Fehlermeldungen und ihrer Bedeutungen.

Es gibt eine Menge verschiedener Operatoren, die Sie in Python-Ausdrücken verwenden können. Tabelle 1–1 führt die arithmetischen Operatoren auf.

Operator	Operation	Beispiel	Ergebnis
**	Exponent	`2 ** 3`	8
%	Modulo/Rest	`22 % 8`	6
//	Integerdivision/abgerundeter Quotient	`22 // 8`	2
/	Division	`22 / 8`	2.75
*	Multiplikation	`3 * 5`	15
-	Subtraktion	`5 - 2`	3
+	Addition	`2 + 2`	4

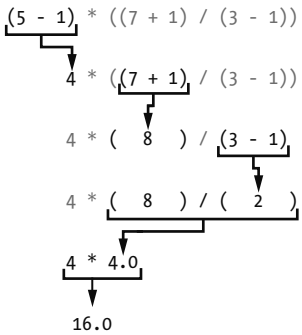
Tab. 1–1 *Arithmetische Operatoren, geordnet vom höchsten zum niedrigsten Rang*

Die Auswertungsreihenfolge oder *Rangfolge* der arithmetischen Operatoren in Python entspricht ihrer gewöhnlichen Rangfolge in der Mathematik: Als Erstes wird der Operator ** ausgewertet, dann die Operatoren *, /, // und % von links nach rechts, und schließlich die Operatoren + und - (ebenfalls von links nach rechts). Um die Auswertungsreihenfolge zu ändern, können Sie bei Bedarf Klammern setzen. Der Weißraum zwischen den Operatoren und Werten spielt in Python keine Rolle (außer bei den Einrückungen am Zeilenanfang). Ein Abstand von einem Leerzeichen ist jedoch üblich. Zur Übung geben Sie die folgenden Ausdrücke in die interaktive Shell ein:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
```

```
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2 + 2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Die Ausdrücke müssen Sie jeweils selbst eingeben, aber Python nimmt Ihnen die Arbeit ab, sie auf einen einzelnen Wert zu reduzieren. Wie die folgende Grafik zeigt, wertet es dabei die einzelnen Teile eines Ausdrucks nacheinander aus, bis ein einziger Wert übrig ist:



Die Regeln, nach denen Operatoren und Werte zu Ausdrücken zusammengestellt werden, bilden einen grundlegenden Bestandteil der Programmiersprache Python, vergleichbar mit den Grammatikregeln einer natürlichen Sprache. Betrachten Sie das folgende Beispiel:

Dies ist ein grammatikalisch korrekter deutscher Satz.

Dies grammatikalisch ist Satz kein deutscher korrekter.

Der zweite Satz lässt sich nur schwer verstehen (»parsen«, wie es bei einer Programmiersprache heißt), da er nicht den Regeln der deutschen Grammatik folgt. Genauso ist es, wenn Sie eine schlecht formulierte Python-Anweisung eingeben. Python versteht sie nicht und zeigt die Fehlermeldung `SyntaxError` an, wie die folgenden Beispiele zeigen:

```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

Um herauszufinden, ob eine Anweisung funktioniert oder nicht, können Sie sie einfach in die interaktive Shell eingeben. Keine Angst, dadurch können Sie nichts kaputt machen. Schlimmstenfalls zeigt Python eine Fehlermeldung an. Für professionelle Softwareentwickler gehören Fehlermeldungen zum Alltag.

Die Datentypen für ganze Zahlen, Fließkommazahlen und Strings

Ein *Datentyp* ist eine Kategorie für Werte, wobei jeder Wert zu genau einem Datentyp gehört. Die gebräuchlichsten Datentypen in Python finden Sie in Tabelle 1–2. Werte wie `-2` und `-30` sind beispielsweise *Integerwerte*. Dieser Datentyp (`int`) steht für ganze Zahlen. Zahlen mit Dezimalpunkt, z. B. `3.14`, sind dagegen *Fließkommazahlen* und weisen den Typ `float` auf. Beachten Sie, dass ein Wert wie `42` ein Integer ist, `42.0` dagegen eine Fließkommazahl.

Datentyp	Beispiele
Integer	`-2`, `-1`, `0`, `1`, `2`, `3`, `4`, `5`
Fließkommazahlen	`-1.25`, `-1.0`, `-0.5`, `0.0`, `0.5`, `1.0`, `1.25`
Strings	`'a'`, `'aa'`, `'aaa'`, `'Hello!'`, `'11 cats'`

Tab. 1–2 *Häufig verwendete Datentypen*

In Python-Programmen können auch Textwerte vorkommen, sogenannte *Strings* (`str`). Schließen Sie Strings immer in einfache Anführungszeichen ein (z. B. `'Hello'` oder `'Goodbye cruel world!'`), damit Python weiß, wo der String anfängt und wo er endet. Sie können sogar einen String erstellen, der gar keine Zeichen enthält, nämlich den *leeren String* `''`. In Kapitel 4 werden Strings ausführlicher behandelt.

Wenn Sie die Fehlermeldung `SyntaxError: EOL while scanning string literal` erhalten, haben Sie wahrscheinlich wie im folgenden Beispiel das schließende einfache Anführungszeichen am Ende eines Strings vergessen:

```
>>> 'Hello world!
SyntaxError: EOL while scanning string literal
```

Stringverkettung und -wiederholung

Die Bedeutung eines Operators kann sich in Abhängigkeit von den Datentypen der Werte ändern, die rechts und links von ihm stehen. Beispielsweise fungiert + zwischen zwei Integer- oder Fließkommawerten als Additionsoperator, zwischen zwei Strings aber als *Stringverkettungsoperator*. Probieren Sie Folgendes in der interaktiven Shell aus:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

Dieser Ausdruck wird zu einem einzigen neuen String ausgewertet, der den Text der beiden Originalstrings enthält. Wenn Sie jedoch versuchen, den Operator + zwischen einem String und einem Integerwert einzusetzen, weiß Python nicht, wie es damit umgehen soll, und gibt eine Fehlermeldung aus:

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Alice' + 42
TypeError: can only concatenate str (not "int") to str
```

Die Fehlermeldung `can only concatenate str (not "int") to str` bedeutet, dass Python glaubt, Sie wollten einen Integer mit dem String `'Alice'` verketten. Dazu aber müssten Sie den Integerwert ausdrücklich in einen String umwandeln, da Python dies nicht automatisch tun kann. (Die Umwandlung von Datentypen werden wir im Abschnitt »Analyse des Programms« weiter hinten in diesem Kapitel erklären und uns dabei mit den Funktionen `str()`, `int()` und `float()` beschäftigen.)

Zwischen zwei Integer- oder Fließkommawerten dient * als Multiplikationsoperator, doch zwischen einem String und einem Integerwert wird er zum *Stringwiederholungsoperator*. Um das auszuprobieren, geben Sie in die interaktive Shell Folgendes ein:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

Der Ausdruck wird zu einem einzigen String ausgewertet, der den ursprünglichen String so oft enthält, wie der Integerwert angibt. Die Stringwiederholung ist zwar ein nützlicher Trick, wird aber längst nicht so häufig angewendet wie die Stringverkettung.

Den Operator * können Sie nur zwischen zwei numerischen Werten (zur Multiplikation) oder zwischen einem String- und einem Integerwert einsetzen (zur Stringwiederholung). In allen anderen Fällen zeigt Python eine Fehlermeldung an:

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Es ist sinnvoll, dass Python solche Ausdrücke nicht auswertet. Schließlich ist es nicht möglich, zwei Wörter miteinander zu multiplizieren, und es dürfte auch ziemlich schwierig sein, einen willkürlichen String eine gebrochene Anzahl von Malen zu wiederholen.

Werte in Variablen speichern

Eine *Variable* können Sie sich wie eine Kiste im Arbeitsspeicher des Computers vorstellen, in der einzelne Werte abgelegt werden. Wenn Sie das Ergebnis eines ausgewerteten Ausdrucks an einer späteren Stelle in Ihrem Programm noch brauchen, können Sie es in einer Variablen festhalten.

Zuweisungsanweisungen

Um einen Wert in einer Variablen zu speichern, verwenden Sie eine *Zuweisungsanweisung*. Sie besteht aus einem Variablennamen, einem Gleichheitszeichen (das hier nicht als Gleichheitszeichen dient, sondern als *Zuweisungsoperator*) und dem zu speichernden Wert. Wenn Sie die Zuweisungsanweisung `spam = 42` eingeben, wird der Wert `42` in der Variablen `spam` gespeichert.

Sie können sich eine Variable als eine beschriftete Kiste vorstellen, in der der Wert abgelegt wird (siehe Abb. 1–1).

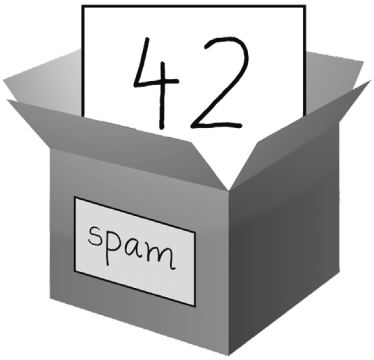


Abb. 1–1 *Die Anweisung* `spam = 42` *sagt dem Programm: »Die Variable* `spam` *enthält jetzt die Ganzzahl* 42.*«*

Geben Sie beispielsweise Folgendes in die interaktive Shell ein:

```
>>> spam = 40    ❶
>>> spam
40
>>> eggs = 2
>>> spam + eggs    ❷
42
>>> spam + eggs + spam
82
>>> spam = spam + 2    ❸
>>> spam
42
```

Eine Variable wird *initialisiert* (erstellt), wenn zum ersten Mal ein Wert in ihr gespeichert wird (❶). Danach können Sie sie zusammen mit anderen Variablen und Werten in Ausdrücken verwenden (❷). Wenn Sie der Variablen einen neuen Wert zuweisen (❸), geht der alte Wert verloren. Daher wird `spam` am Ende dieses Beispiels nicht mehr zu `40` ausgewertet, sondern zu `42`. Die Variable ist also *überschrieben* worden. Versuchen Sie in der interaktiven Shell wie folgt einen String zu überschreiben:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Wie die Kiste in Abb. 1–2 enthält die Variable `spam` in diesem Beispiel den Wert `Hello`, bis er durch `Goodbye` ersetzt wird.

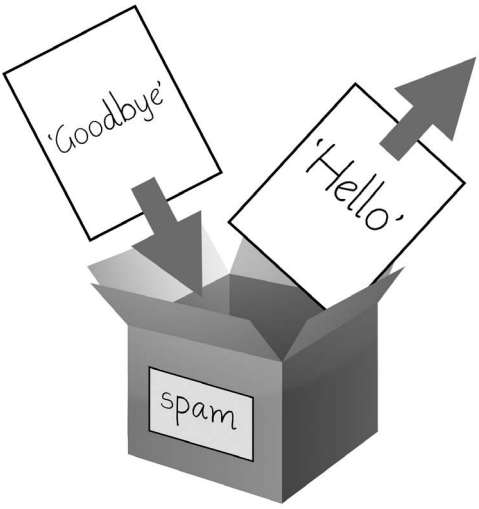


Abb. 1–2 *Wird einer Variablen ein neuer Wert zugewiesen, so wird der alte Wert vergessen.*

Variablennamen

Ein guter Variablenname beschreibt die enthaltenen Daten. Stellen Sie sich vor, Sie ziehen um und beschriften alle Kartons mit »Sachen«. Dann würden Sie Ihre Sachen nie wiederfinden! In diesem Buch und in einem Großteil der Python-Dokumentation werden allgemeine Variablennamen wie `spam`, `eggs` und `bacon` verwendet (in Anlehnung an den Spam-Sketch von Monty Python), aber in Ihren eigenen Programmen sollten Sie beschreibende Namen verwenden, um den Code leichter lesbar zu machen.

Sie können Variablen in Python fast beliebig benennen, allerdings gibt es einige Einschränkungen. Tabelle 1–3 führt Beispiele für gültige Variablennamen auf. Sie müssen die folgenden drei Regeln erfüllen:

- Der Name muss ein einzelnes Wort sein, darf also keine Leerzeichen enthalten.
- Der Name darf nur aus Buchstaben, Ziffern und dem Unterstrich bestehen.
- Der Name darf nicht mit einer Zahl beginnen.

Gültige Variablennamen	Ungültige Variablennamen
`current_balance`	`current-balance` (Bindestriche sind nicht zulässig)
`currentBalance`	`current balance` (Leerzeichen sind nicht zulässig)
`account4`	`4account` (der Name darf nicht mit einer Zahl beginnen)
`_42`	`42` (der Name darf nicht mit einer Zahl beginnen)
`TOTAL_SUM`	`TOTAL_$UM` (Sonderzeichen wie `$` sind nicht zulässig)
`hello`	`'hello'` (Sonderzeichen wie `'` sind nicht zulässig)

Tab. 1–3 *Gültige und ungültige Variablennamen*

Bei Variablennamen wird zwischen Groß- und Kleinschreibung unterschieden, sodass `spam`, `SPAM`, `Spam` und `sPaM` vier verschiedene Variablen bezeichnen. Zwar ist `Spam` ein zulässiger Variablenname, aber verabredungsgemäß sollten Variablennamen in Python mit einem Kleinbuchstaben beginnen.

In diesem Buch wird für Variablennamen die CamelCase-Schreibweise verwendet, also die Schreibung mit Binnenmajuskel statt mit einem Unterstrich. Variablennamen sehen also aus wie `lookLikeThis` und nicht wie `look_like_this`. Erfahrene Programmierer mögen einwenden, dass die offizielle Python-Stilrichtlinie PEP 8 Unterstriche verlangt. Ich bevorzuge allerdings die CamelCase-Schreibweise und möchte dazu auf den Abschnitt »Sinnlose Übereinstimmung ist die Plage kleiner Geister« aus PEP 8 verweisen:

> *»Übereinstimmung mit der Stilrichtlinie ist wichtig. Am wichtigsten ist es jedoch zu wissen, wann man diese Übereinstimmung aufgeben muss. Für manche Fälle ist die Stilrichtlinie einfach ungeeignet. Urteilen Sie dann selbst nach bestem Wissen und Gewissen.«*

Ihr erstes Programm

In der interaktiven Shell können Sie einzelne Python-Anweisungen nacheinander ausführen, aber um ein vollständiges Python-Programm zu schreiben, müssen Sie die Anweisungen in den *Dateieditor* eingeben. Er ähnelt Texteditoren wie dem Windows-Editor oder TextMate, verfügt aber zusätzlich über einige Sonderfunktionen für die Eingabe von Quellcode. Um in Mu eine neue Datei anzulegen, klicken Sie in der obersten Zeile auf *New*.

In dem Fenster, das jetzt erscheint, sehen Sie einen Cursor, der auf Ihre Eingaben wartet. Dieses Fenster unterscheidet sich jedoch von der interaktiven Shell, in der Python-Anweisungen ausgeführt werden, sobald Sie die Eingabetaste drücken. Im Dateieditor können Sie viele Anweisungen eingeben, die Datei speichern und

dann das Programm ausführen. Anhand der folgenden Merkmale können Sie erkennen, in welchem der beiden Fenster Sie sich gerade befinden:

- Das Fenster der interaktiven Shell zeigt die Eingabeaufforderung >>> an.
- Im Dateieditorfenster gibt es die Eingabeaufforderung >>> nicht.

Nun ist es an der Zeit, Ihr erstes Programm zu schreiben! Geben Sie im Fenster des Dateieditors Folgendes ein:

```
# Dieses Programm sagt "Hallo" und fragt nach Ihrem Namen.    ❶

print('Hello world!')    ❷
print('What is your name?')    # Fragt nach dem Namen
myName = input()    ❸
print('It is good to meet you, ' + myName)    ❹
print('The length of your name is:')    ❺
print(len(myName))
print('What is your age?')     # Fragt nach dem Alter    ❻
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Nachdem Sie den Quellcode eingegeben haben, speichern Sie ihn, damit Sie ihn nicht jedes Mal neu eingeben müssen, wenn Sie Mu starten. Klicken Sie auf *Save*, geben Sie im Feld *File Name* den Namen **hello.py** ein und klicken Sie auf *Save*.

Während Sie ein Programm eingeben, sollten Sie es zwischendurch immer mal wieder speichern. Sollte Ihr Computer abstürzen oder sollten Sie versehentlich Mu beenden, verlieren Sie dann keinen Code. Als Tastaturkürzel zum Speichern einer Datei drücken Sie Strg + S auf Windows und Linux bzw. Cmd + S auf macOS.

Nachdem Sie das Programm gespeichert haben, führen Sie es aus. Drücken Sie dazu F5. Das Programm läuft jetzt im Fenster der interaktiven Shell. Beachten Sie aber, dass Sie F5 im Editorfenster drücken müssen, nicht im Shell-Fenster. Geben Sie Ihren Namen ein, wenn das Programm Sie danach fragt. Die Programmausgabe im Fenster der interaktiven Shell sieht wie folgt aus:

```
Python 3.7.0b4 (v3.7.0b4:eb96c37699, May 2 2018, 19:02:22) [MSC v.1913 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
```