HANSER

Leseprobe

Bernd Klein

Einführung in Python 3

Für Ein- und Umsteiger

ISBN (Buch): 978-3-446-44133-0

ISBN (E-Book): 978-3-446-44151-4

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44133-0

sowie im Buchhandel.

Inhalt

Vorwort

Bedingt durch den traumhaften Anstieg der Bedeutung von Python in der Wissenschaft und Wirtschaft in den letzten Jahren gibt es auch ein wachsendes Interesse an geeigneten Python-Büchern. Gerade für Programmieranfänger besteht die Schwierigkeit darin, ein geeignetes Buch zu finden. Der unüberlegte Griff ins Bücherregal führt schnell zum Kauf eines Buches, was zwar viele Vorzüge haben mag, aber für Anfängerinnen und Anfänger völlig ungeeignet ist.

Dies ist besonders schade, da es sich bei Python ja um eine einfache Programmiersprache handelt. Das war bereits beim anfänglichen Design der Sprache eines der wesentlichen Ziele ihres „Erfinders" Guido van Rossum, als er Python Anfang der 1990er Jahre am Zentrum für Mathematik und Informatik (Centrum voor Wiskunde en Informatica) in Amsterdam entwarf. Python ist einfach, weil es mit erstaunlich wenigen Schlüsselwörtern auskommt und seine Syntax, also der Aufbau der Sprache, auf Übersichtlichkeit optimiert ist.

Auch im Bereich objektorientierte Programmierung ist Python sehr konsequent, in manchen Dingen sogar konsequenter als Java. So sind in Python alles Objekte, d.h. es gibt keine primitiven Typen. Selbst Integer und Float-Zahlen sind in Python als Klassen realisiert. Ein weiterer Unterschied zu Java besteht darin, dass in Python Mehrfachvererbung möglich ist.

Auch Programme in anderen Sprachen lassen sich einfach als Module in Python einbetten. So kann man beispielsweise zeitkritische Algorithmen in C programmieren und sie dann in Python einbinden.

Zusammenfassend kann man sagen, dass es sich bei Python um eine Programmiersprache handelt, die sich bestens zum Einstieg in die Programmierung eignet, aber auch die optimale Lösung für Spezialisten aus zahlreichen Problemfeldern ist. In diesem Sinne finden Sie in diesem Buch den idealen Einstieg.

Brigitte Bauer-Schiewek, Lektorin

Danksagung

Zum Schreiben eines Buches benötigt es neben der nötigen Erfahrung und Kompetenz im Fachgebiet vor allem viel Zeit. Zeit außerhalb des üblichen Rahmens. Zeit, die vor allem die Familie mitzutragen hat. Deshalb gilt mein besonderer Dank meiner Frau Karola, die mich während dieser Zeit tatkräftig unterstützt hat.

Außerdem danke ich den zahlreichen Teilnehmern an meinen Python-Kursen, die mir geholfen haben, meine didaktischen und fachlichen Kenntnisse kontinuierlich zu verbessern. Ebenso möchte ich den Besuchern meiner Online-Tutorials unter www.python-kurs.eu und www.python-course.eu danken, vor allem denjenigen, die sich mit konstruktiven Anmerkungen bei mir gemeldet haben. Allen voran Stefan Günther, der mir auch wertvolle Anregungen zur Erstauflage des Buches lieferte.

Zuletzt danke ich auch ganz herzlich dem Hanser Verlag, der dieses Buch – nun auch in der zweiten Auflage – ermöglicht hat. Vor allem danke ich Frau Brigitte Bauer-Schiewek, Programmplanung Computerbuch, und Frau Sarah Merz, Lektorat und Online-Marketing, für die kontinuierliche ausgezeichnete Unterstützung. Für die technische Unterstützung bei LaTex-Problemen danke ich Herrn Uwe Hartmann und Herrn Stephan Korell. Herrn Jürgen Dubau danke ich fürs Lektorat.

Für die Hilfe zur zweiten – weitestgehend überarbeiteten – Auflage möchte ich im besonderen Maße Herrn Jan Lendertse und Herrn Vincent Bermel Dank sagen. Von ihnen kamen viele wertvolle Anregungen und Änderungsvorschläge.

Bernd Klein, Singen

10 Schleifen

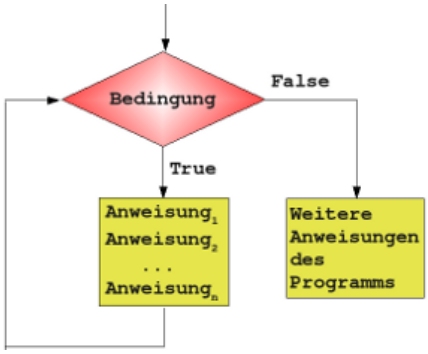
10.1 Übersicht

Schleifen werden benötigt, um einen Codeblock, also eine oder mehrere Python-Anweisungen, wiederholt auszuführen. Einen solchen Codeblock bezeichnet man auch als Schleifenkörper oder Body. In Python gibt es zwei Schleifentypen: die while-Schleife und die for-Schleife.

Bild 10.1 Karussell

Die meisten Schleifenarten, die in Programmiersprachen Verwendung finden, enthalten einen Zähler oder ganz allgemein Variablen, die im Verlauf der Berechnungen innerhalb des Schleifenkörpers ihre Werte ändern. Außerhalb, das heißt noch vor dem Beginn der Schleife, werden diese Variablen initialisiert. Vor jedem Schleifendurchlauf wird geprüft, ob ein Ausdruck, in dem diese Variable oder Variablen vorkommen, wahr ist. Dieser Ausdruck bestimmt das Endekriterium der Schleife. Solange die Berechnung dieses Ausdrucks wahr ist, d.h. „True" liefert, wird der Rumpf der Schleife ausgeführt. Nachdem alle Anweisungen des Schleifenkörpers durchgeführt worden sind, springt die Programmsteuerung automatisch zum Anfang der Schleife, also zur Prüfung des Endekriteriums zurück und prüft wieder, ob diese nochmals erfüllt ist. Wenn ja, geht es wie oben beschrieben weiter, ansonsten wird der Schleifenkörper nicht mehr ausgeführt, und es wird mit dem Rest des Skripts fortgefahren. Das unten stehende Diagramm zeigt dies schematisch.

10.2 while-Schleife



Das folgende Skript, das wir in der interaktiven Shell direkt eintippen können, gibt die Zahlen von 1 bis 4, gefolgt von ihrem Quadrat, unter Benutzung einer while-Schleife aus:

```
>>> i = 1
>>> while i <= 4:
...     print(i, i**2)
...     i += 1
...
1 1
2 4
3 9
4 16
```

Auch die Summe der Zahlen von 1 bis 100 lässt sich mittels einer while-Schleife leicht berechnen, wie wir im folgenden Programm sehen können:

```
n = 100
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i + 1
print("Summe von 1 bis " +str(n) + ": " + str(sum) )
```

Zu obigen Programm lässt sich Folgendes sagen. Zur Aufnahme der Summe haben wir eine Variable „sum" benutzt. „sum" ist aber auch eine Python-Funktion, die die Summe einer aus numerischen Werten bestehenden Liste oder eines Tupels berechnet:

```
>>> sum([3, 4.5, 9])
16.5
```

Damit hätten wir uns natürlich auch das obige Programm zur Berechnung der Summe der Zahlen von 1 bis 100 sparen können, indem wir „sum" und „range" benutzen:

```
>>> n = 100
>>> sum(range(1,n+1))
5050
```

Einige kennen sicherlich auch die Gaußsche Summenformel, mit der wir die Aufgabenstellung auch ohne sum und range direkt lösen können:

```
>>> n = 100
>>> summe = n * (n + 1) / 2
>>> summe
5050.0
```

10.3 break und continue

Für die Schleifen existieren zwei wichtige Anweisungen: „break" zum vorzeitigen Abbruch der Schleife und „continue", um einen Durchlauf zu beenden und mit dem nächsten Durchlauf bzw. der Überprüfung der Abbruchbedingung weiterzumachen. Die break-Anweisung steht innerhalb des Schleifenrumpfes meist in Verbindung mit einer if-Abfrage. Stößt der Programmablauf auf eine break-Anweisung, wird die Schleife unmittelbar abgebrochen. Das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt. Bei geschachtelten Schleifen wird mittels break nur die innerste Schleife abgebrochen. Trifft der Programmablauf auf eine continue-Anweisung, wird der Schleifendurchlauf abgebrochen, und der Programmablauf kehrt zum Schleifenkopf zurück, wo geprüft wird, ob die Bedingung für einen weiteren Durchlauf erfüllt ist.

Im folgenden Beispiel benutzen wir sowohl break als auch continue. Falls ein doppelter Eintrag in der Liste steht, wird mittels continue auf das nächste Listenelement weiter gegangen. Falls die Liste eine negative Zahl oder eine Null enthält, wird die Schleife komplett abgebrochen:

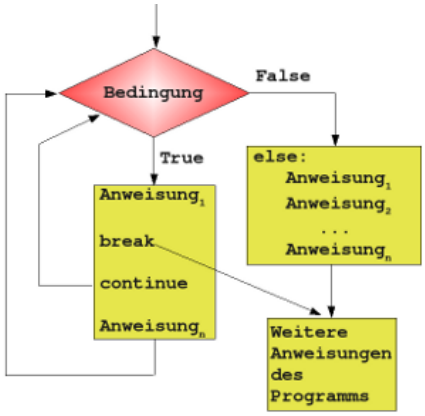
```
liste = eval(input("Liste mit positiven Zahlen eingeben: "))
n = len(liste)
i = 0
previous = None
erg = []
while i < n:
    current = liste[i]
    i += 1
    if current == previous:
        continue
    if current <= 0:
        print("Abbruch: Nicht-positive Zahl gefunden!")
        break
    erg.append(current)
    previous = current

print(erg)
```

Zum besseren Verständnis des obigen Programmes zeigen wir einen Beispiellauf:

```
$ python3 break_example.py
Liste mit positiven Zahlen eingeben: [37, 99, 123, 17, 17, 17, 89, 32,
    -3]
Abbruch: Nicht-positive Zahl gefunden!
[37, 99, 123, 17, 89, 32]
```

10.4 else-Teil



C-Programmierern[1] wie auch Programmierern anderer Programmiersprachen kommt es meist sehr merkwürdig vor, wenn sie ein else ohne zugehöriges if finden. Bei Schleifen kann ein else-Teil folgen, das muss aber nicht so sein. Die Anweisungen im else-Teil werden ausgeführt, sobald die Bedingung nicht mehr erfüllt ist. Sicherlich fragen sich einige nun, worin dann der Unterschied zu einer normalen while-Schleife liegt. Hätte man die Anweisungen nicht in den else-Teil gesteckt, sondern einfach hinter die while-Schleife gestellt, wären sie ja auch genauso ausgeführt worden. Der else-Teil einer while-Schleife wird erst zusammen mit dem break-Kommando sinnvoll. Normalerweise wird eine Schleife nur beendet, wenn die Bedingung im Schleifenkopf erfüllt ist. Mit break kann man aber eine Schleife vorzeitig – also gewissermaßen als Notausstieg – verlassen.

Im folgenden Beispiel, einem einfachen Zahlenratespiel, kann man erkennen, dass in Kombination mit einem break der else-Zweig durchaus sinnvoll sein kann. Nur wenn die while-Schleife regulär beendet wird, d.h. der Spieler die Zahl erraten hat, gibt es einen Glückwunsch. Gibt der Spieler auf, d.h. break, dann wird der else-Zweig der while-Schleife nicht ausgeführt.

```
import random
n = 20
to_be_guessed = int(n * random.random()) + 1
```

[1] Es gibt auch eine else-Schleife in C und C++. Diese wird aber nur selten verwendet.

```
guess = 0
while guess != to_be_guessed:
    guess = int(input("New number: "))
    if guess > 0:
        if guess > to_be_guessed:
            print("Number too large")
        else:
            print("Number too small")
    else:
        print("Sorry, that you're giving up!")
        break
else:
    print("Congratulation. You made it!")
```

Die Ausgabe einer Spielsitzung könnte beispielsweise so aussehen:

```
$ python3 number_game.py
New number: 12
Number too small
New number: 15
Number too small
New number: 18
Number too large
New number: 17
Number too small
Congratulation. You made it!
$
```

10.5 For-Schleife

Wie auch die while-Schleife ist die for-Schleife eine Kontrollstruktur, mit der eine Gruppe von Anweisungen (ein Block) wiederholt ausführt werden kann. Die Syntax der for-Schleifen unterscheidet sich in den verschiedenen Programmiersprachen. Ebenso ist die Semantik einer for-Schleife, also wie sie vom Compiler oder Interpreter zu verstehen bzw. auszuführen ist, von Programmiersprache zu Programmiersprache unterschiedlich. Die „klassische" numerische Schleife, wie sie C und C++ kennt, besitzt eine Schleifenvariable, die mit einem Startwert initialisiert wird und sich nach jedem Durchlauf des Schleifenkörpers verändert, d.h. meistens um einen bestimmten Wert (z.B. 1) erhöht oder vermindert wird, bis der definierte Zielwert erreicht ist. Man nennt diese Schleifenform auch Zählschleife, weil die Schleifenvariable und damit auch der Startwert, der Endwert und die Schrittweite numerisch sein müssen.

Im folgenden Beispiel sehen wir eine for-Schleife in C, die die Zahlen von 1 bis 100 ausdruckt:

```
for( i = 0; i < 100; i++)
   printf("i: %d\n", i);
```

Auch wenn Sie diese Schleifenform bereits in C oder einer anderen Sprache liebgewonnen haben, müssen wir Sie leider enttäuschen: Python kennt keine solche for-Schleife. Wohl-

gemerkt „keine solche", aber sehr wohl eine for-Schleife. Die in Python benutzte Art von for-Schleife entspricht der in der Bash-Shell oder in Perl verwendeten foreach-Schleife. Bei dieser Schleifenart handelt es sich um ein Sprachkonstrukt, mit dessen Hilfe nacheinander die Elemente einer Menge oder Liste bearbeitet werden können. Dazu werden sie einer Variable zugewiesen.

Im Folgenden sehen wir die allgemeine Syntax der for-Schleife in Python. Sequenz steht für ein iterierbares Objekt.

```
for Variable in Sequenz:
        Anweisung_1
        Anweisung_2
        ...
        Anweisung_n
else:
        Else-Anweisung_1
        Else-Anweisung_2
        ...
        Else-Anweisung_m
```

Wie bereits gesagt, dient in Python die for-Schleife zur Iteration über eine Sequenz von Objekten, während sie in vielen anderen Sprachen meist nur „eine etwas andere while-Schleife" ist.

Beispiel einer for-Schleife in Python:

```
>>> languages = ["C", "C++", "Perl", "Python"]
>>> for language in languages:
...     print(language)
...
C
C++
Perl
Python
>>>
```

Wie die while-Schleife besitzt auch die for-Schleife einen optionalen else-Block. Wie bei der while-Schleife wird der else-Block nur ausgeführt, wenn die Schleife nicht durch eine break-Anweisung abgebrochen wurde. Das bedeutet, dass der else-Block nur dann ausgeführt wird, wenn alle Elemente der Sequenz abgearbeitet worden sind.

Trifft der Programmablauf auf eine break-Anweisung, so wird die Schleife sofort verlassen und das Programm nach der Anweisung fortgesetzt, die der for-Schleife folgt, falls es überhaupt noch Anweisungen nach der for-Schleife gibt.

Üblicherweise befindet sich die break-Anweisung wie im folgenden Beispiel innerhalb einer Konditionalanweisung:

```
edibles = ["ham", "spam","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
```

```
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

Wenn wir obiges Beispiel unter for.py speichern und aufrufen, erhalten wir folgende Ausgaben:

```
$ python for.py
Great, delicious ham
No more spam please!
Finally, I finished stuffing myself
$
```

Wenn wir „spam" aus der Liste der essbaren Dinge entfernen, erhalten wir folgende Ausgabe:

```
$ python for.py
Great, delicious ham
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
$
```

Vielleicht ist unsere Abscheu vor dem Dosenfutter „spam" nicht so groß, dass wir sofort aufhören zu essen. In diesem Fall kommt die continue-Anweisung ins Spiel. Im folgenden kleinen Skript benutzen wir continue, um mit dem nächsten Artikel der essbaren Artikel weiterzumachen. „continue" schützt uns davor, „spam" essen zu müssen:

```
edibles = ["ham", "spam", "eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        continue
    print("Great, delicious " + food)
    # here can be the code for enjoying our food :-)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

Die Ausgabe sieht dann wie folgt aus:

```
$ python for.py
Great, delicious ham
No more spam please!
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
$
```

10.6 Aufgaben

Für die erste Aufgabe benötigen wir die römischen Zahlen. Für diejenigen, die sich nicht so ganz sicher sind mit römischen Zahlen, geben wir hier die Zuordnungen in der folgenden Tabelle.

Tabelle 10.1 Römische Zahlen

Römische Zahl	Wert (Dezimalzahl)
I	1
II	2
III	3
IV	4
V	5
VI	6
VII	7
VIII	8
IX	9
X	10
XI	11
XIV	14
XV	15
XVI	16
…	…
XIX	19
XX	20
XXI	21
…	…
XXIX	29
XXX	30
XL	40
L	50
LX	60
XC	90
C	100
CC	200
CD	400
D	500
CM	900
M	1000
MM	2000

1. Aufgabe:

Schreiben Sie ein Python-Programm, das eine beliebige römische Zahl in eine „gewöhnliche" Dezimalzahl umrechnet.

Lösung: 35.4 (1. Aufgabe), Seite 442

2. Aufgabe:

Bild 10.2 Achtung: Frösche

In der nächsten Aufgabe lernen wir einen besonderen Frosch kennen, so wie ihn sich nur Mathematiker ausdenken können. Besonders seine Art, eine Straße zu überqueren, macht es zweifelhaft, ob er in der realen Welt lange überleben könnte. Er überquert eine 2,50 Meter breite Straße wie folgt: Mit dem ersten Sprung legt er die erstaunliche Distanz von einem Meter zurück, dann springt er wegen zunehmender Erschöpfung mit jedem weiteren Schritt immer nur noch halb so weit wie vorher.

Die Entfernung, die er dabei zurücklegt, berechnet sich also als Summe der Werte $1+0,5+0.25+0,125$ und so weiter.

Dies entspricht natürlich in mathematischer Schreibweise der folgenden Notation:

$$\sum_{i=0}^{n} \frac{1}{i^2} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \ldots$$

Versuchen Sie, mittels eines Python-Programms herauszubekommen, ob der Frosch es auf die andere Straßenseite schafft.

Lösung: 35.4 (2. Aufgabe), Seite 443

3. Aufgabe:

Der indische Herrscher Shihram tyrannisierte seine Untertanen und stürzte sein Land in Not und Elend. Um die Aufmerksamkeit des Königs auf seine Fehler zu lenken, ohne seinen Zorn zu entfachen, schuf Dahers Sohn, der weise Brahmane Sissa, ein Spiel, in dem der König als wichtigste Figur ohne Hilfe anderer Figuren und Bauern nichts ausrichten kann. Der Unterricht im Schachspiel machte auf Shihram einen starken Eindruck. Er wurde milder und ließ das Schachspiel verbreiten, damit alle davon Kenntnis nähmen. Um sich für die anschauliche Lehre von Lebensweisheit und zugleich Unterhaltung zu bedanken, gewährte er dem Brahmanen einen freien Wunsch. Dieser wünschte sich Weizenkörner: Auf das erste Feld eines Schachbretts wollte er ein Korn, auf das zweite Feld die doppelte Menge, also zwei, auf das dritte wiederum

doppelt so viele, also vier und so weiter. Der König lachte und war gleichzeitig erbost über die vermeintliche Bescheidenheit des Brahmanen.

Als sich Shihram einige Tage später erkundigte, ob Sissa seine Belohnung in Empfang genommen habe, musste er hören, dass die Rechenmeister die Menge der Weizenkörner noch nicht berechnet hätten. Der Vorsteher der Kornkammer meldete nach mehreren Tagen ununterbrochener Arbeit, dass er diese Menge Getreidekörner im ganzen Reich nicht aufbringen könne. Auf allen Feldern eines Schachbretts zusammen wären es 18.446.744.073.709.551.615 Weizenkörner. Nun stellte er sich die Frage, wie das Versprechen eingelöst werden könne. Der Rechenmeister half dem Herrscher aus der Verlegenheit, indem er ihm empfahl, er solle Sissa ibn Dahir ganz einfach das Getreide Korn für Korn zählen lassen.[2]

Berechnen Sie mithilfe eines Python-Programms, ohne eine Formel zu Hilfe zu nehmen, wie viele Weizenkörner angehäuft werden müssen.

Lösung: 35.4 (3. Aufgabe), Seite 444

[2] Die Weizenkornlegende wurde wörtlich von Wikipedia entnommen.

Stichwortverzeichnis

N

O

P

S