HANSER

Leseprobe

zu

Python für Ingenieure und Naturwissenschaftler

Einführung in die Programmierung, mathematische Anwendungen und Visualisierungen

von Hans-Bernhard Woyand

2., überarbeitete und erweiterte Auflage
Mit zahlreichen Bildern und Tabellen sowie 73 Aufgaben

ISBN (Buch): 978-3-446-45792-8
ISBN (E-Book): 978-3-446-45796-6

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-45792-8
sowie im Buchhandel

Vorwort

Seit vielen Jahren halte ich an der Bergischen Universität Wuppertal die Lehrveranstaltung „Informatik“ im Grundstudium Maschinenbau. Mehr als 10 Jahre haben wir in diesen Kursen die Programmiersprache C/C++ eingesetzt. Seit mehreren Jahren verwenden wir stattdessen nun *Python* als erste Programmiersprache und das mit großem Erfolg. Es zeigte sich, dass gerade bei Ingenieuren, die oftmals Schwierigkeiten mit dem algorithmischen Denken haben, Python einen leichteren Zugang ermöglicht.

Im Studiengang Maschinenbau wie auch in vielen anderen Studiengängen spielt das wissenschaftliche Rechnen und Visualisieren eine große Rolle. In der Literatur zu Python ist das nicht so: entweder werden die wissenschaftlichen Anwendungen gar nicht bzw. nur am Rande behandelt, oder die Bücher sind so umfangreich, dass sie als vorlesungsbegleitender Text ungeeignet sind. So entstand die Idee, das Skript zur Vorlesung zu diesem Buch auszuarbeiten.

Zielgruppe
Das Buch richtet sich an *Programmieranfänger*, die Python als erste Programmiersprache lernen möchten. Es wird also zuerst in Python eingeführt und dann werden die Grundlagen erläutert. Es folgen Vertiefungen und ein eigenes Kapitel über die objektorientierte Programmierung. Dann beginnt das wissenschaftliche Rechnen und Visualisieren. Hierzu wird in die Nutzung der wichtigsten Programmbibliotheken (Packages) eingeführt. Hauptzielgruppe sind also Studierende wissenschaftlicher Studiengänge, die Python erlernen wollen und bei denen die mathematischen und graphische Anwendungen eine wichtige Rolle spielen. Vorausgesetzt wird nur mathematisch-naturwissenschaftliches Wissen, wie es an höheren Schulen vermittelt wird.

Aufgabenorientierte Lehre
Mehr als 90 Aufgabenstellungen mit fast immer kommentierten Lösungen werden im Buch behandelt. Nach meiner Erfahrung lernen Studierende am meisten durch das selbstständige Lösen von Aufgaben.

Software
Die Software, die benötigt wird, um mit diesem Buch zu arbeiten, ist kostenfrei erhältlich. Es handelt sich dabei um die neuere Variante 3.2 bzw. 3.5 der Programmiersprache Python, sowie deren Vorgängerversionen 2.6 und 2.7. Die wissenschaftlichen Anwendungspakete sind nämlich *noch nicht alle* mit der neuen Python-Version kompatibel. Es ist zu erwarten, dass es auch noch einige Zeit dauern wird, bis die wissenschaftlichen Pakete unter der neuesten Version laufen. Für Programmieranfänger ist der Unterschied zwischen diesen Versionen sowieso nicht sehr bedeutsam.

Web-Seite zum Buch
Zu dem vorliegenden Buch existiert eine Web-Seite, auf der in loser Folge Ergänzungen, Fehlerberichtigungen usw. bereitgestellt werden. Weiterhin können alle Beispiele sowie die Lösungen der Aufgaben dort abgerufen werden. Für Nutzer des Betriebssystems MS-Windows sind auch Hinweise zur Installation der Software dort verfügbar. Die Web-Adresse ist

http://woyand.eu/python_buch

Falls sich diese Adresse einmal ändern sollte, so kann der Leser den neuen Standort über die Web-Seiten des Verlages *http://www.hanser-fachbuch.de* erfahren. Dort bestehen ebenfalls Download-Möglichkeiten.

Hinweise, Fehlermeldungen und Anregungen werden über die eMail-Adresse

info@woyand.eu

gern entgegen genommen.

Haftungsausschluss
Das Buch wurde mit größtmöglicher Sorgfalt erstellt. Trotzdem können Fehler nicht ganz ausgeschlossen werden. Aus diesem Grund sind die in diesem Buch dargestellten Verfahren mit keinerlei Garantie verbunden. Ich weise darauf hin, dass weder Autor noch Verlag eine Haftung für direkt oder indirekt entstandene Schäden übernehmen, die sich aus der Benutzung dieses Buches ergeben könnten.

Danksagung
Ich danke meiner Frau Annette Woyand für die sorgfältige Durchsicht des Manuskriptes. Frau Mirja Werner vom Carl Hanser Verlag danke ich für Ihr Engagement beim Zustandekommen dieses Buches.

Ich wünsche allen Lesern viel Erfolg und Spaß beim Einstieg in Python.

Wuppertal, im März 2017 *Hans-Bernhard Woyand*

Vorwort zur zweiten Auflage

Für die zweite Auflage wurden einige inhaltliche Verbesserungen vorgenommen und ein neues Kapitel hinzugefügt. In diesem Kapitel wird gezeigt, wie numerische Berechnungen mit dem Python-Bibliothek Scipy durchgeführt werden können. Scipy ist sehr umfangreich. Deshalb werden - dem Ansatz dieses Buches entsprechend - wichtige Teilbereiche dieser Software-Bibliothek auf wenigen Seiten vorgestellt. Behandelt wird die numerische Berechnung von Integralen, die Interpolation, die Berechnung von Nullstellen, die numerische Optimierung, die Signalanalyse mit der schnellen Fourier Transformation (FFT), sowie die numerische Integration gewöhnlicher Differenzialgleichungen.

Ich hoffe, dass diese Erweiterung des Buches für viele Leser hilfreich und anregend ist und wünsche viel Erfolg und Spaß mit Python.

Wuppertal, im Juli 2018 *Hans-Bernhard Woyand*

Inhaltsverzeichnis

2 Grundlagen

In diesem Kapitel werden die wichtigsten Datentypen, sowie die zugehörigen Operatoren und Standardfunktionen vorgestellt. Als wichtige Grundelemente von Programmen werden weiterhin die Sequenz, die Programmverzweigung und die Programmschleife eingeführt. Schließlich wird gezeigt, wie mithilfe von Funktionen Programme strukturiert werden können.

2.1 Einfache Objekttypen

Eigentlich sollte die Überschrift heißen: einfache Datentypen. Python ist jedoch eine durchgehend objektorientierte Programmiersprache. Aus diesem Grund werden auch einfache Datentypen in Python als Objekte abgebildet. Daher diese Überschrift. In diesem Kapitel geht es darum, diese grundlegenden Typen genauer kennen zu lernen.

2.1.1 Ganze Zahlen – Integer

Von unseren Handrechnungen her kennen wir ganze Zahlen (z.B. 12, -103, -22, +49027). Diese Zahlen werden in Python genauso geschrieben, wie wir es von den Handrechnungen gewohnt sind. Ein vorausgehendes Plus-Zeichen kann entfallen. Die Zahl +42097 ist also identisch mit der Zahl 42097. Ganzzahlen können in Python 3.x beliebig groß sein, d.h. sie können aus beliebig vielen Ziffern bestehen.

Mit der eingebauten Funktion type() kann der Typ einer Zahl ermittelt werden. Hier zwei Beispiele:

```
>>> type(3)
<class 'int'>
```

Die interne Bezeichnung für Ganzzahlen in Python ist *int*.

Oft kommt es vor, dass ein anderer Objekttyp (z.B. eine Zeichenkette oder eine Gleitpunktzahl) in eine Ganzzahl umgewandelt werden muss. Dies erledigt für uns die *Funktion int()*, deren Anwendung nachfolgend gezeigt wird. Wichtig ist bei der Anwendung dieser Funktion, dass das übergebene Argument auch wirklich in eine Ganzzahl umgewandelt werden kann! Wird eine Gleitpunktzahl übergeben, so werden deren „Nachkommastellen“ bei der Umwandlung abgeschnitten.

```
>>> int("235")
235
>>> int(235.57)
235
>>> int("255 Grad")
```

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int("255 Grad")
ValueError: invalid literal for int() with base 10: '255 Grad'
```

In der Datenverarbeitung spielen auch andere Zahlensysteme eine bedeutendere Rolle: das Dualsystem, das Oktalsystem sowie das Hexadezimalsystem. Dualzahlen bestehen aus nur aus den zwei Ziffern 0 und 1. Die Zahl 2 bildet die Basis dieses Systems. In Python können wir Dualzahlen direkt eingeben und in das Dezimalsystem umrechnen lassen. Hierzu wird die Zeichenfolge 0b vor einer Dualzahl codiert. Mit der Funktion *bin()* können wir eine Dezimalzahl in eine Dualzahl umrechnen lassen. Dies ist in folgendem Shell-Dialog dargestellt:

```
>>> x = 0b110011
>>> print(x)
51
>>> y = bin(51)
>>> print(y)
0b110011
```

Das Oktalsystem weist die Basis 8 auf. Oktalzahlen werden durch die Ziffern 0 bis 7 aufgebaut. Python erkennt Oktalzahlen an den beiden führenden Ziffern 0o (oder auch die Ziffern 0O). Mit der Funktion *oct()* können Dezimalzahlen ins Oktalsystem umgerechnet werden. Das Hexadezimalsystem weist die Basis 16 auf und kommt deshalb nicht mit den Ziffern 0 bis 9 aus. Es werden die zusätzlichen Zahlzeichen A, B, C, D, E und F verwendet. Die Funktion *hex()* berechnet die hexadezimale Darstellung aus Dezimalzahlen. Mit den führenden Ziffern 0x wird die Darstellung von Hexadezimalzahlen eingeleitet. Dies ist in dem folgenden Shell-Dialog gezeigt:

```
>>> x = oct(51); y = hex(51)
>>> print(x,y)
0o63 0x33
>>> u = 0o63; v = 0x33
>>> print(u,v)
51 51
```

2.1.2 Gleitpunktzahlen – Float

Bei wissenschaftlichen Berechnungen wird oft mit reellen Zahlen gearbeitet. Dies ist jene Zahlenmenge, die sich aus den rationalen Zahlen (also jenen Zahlen, die durch Brüche dargestellt werden können) und den irrationalen Zahlen (Beispiele hierfür sind die Zahl π und die Eulersche Zahl e) aufbaut. Irrationale Zahlen können potentiell unendlich viele Nachkommastellen haben und sind damit natürlich *nicht* exakt in einem Digitalrechner nachbildbar. Man begnügt sich dann mit einer angenäherten Darstellung durch eine rationale Zahl.

Bei Handrechnungen sind wir es gewohnt, Dezimalzahlen mit einem Komma zu trennen. Beispiele hierfür sind die Zahlen 3,14159, -100,2 und 22,3 10^{-5}. Hier muss der Programmieranfänger gründlich umdenken. Denn in fast allen modernen Programmiersprachen wird statt des Kommas ein Punkt verwendet. Die oben genannten Zahlen werden in Python deshalb folgendermaßen geschrieben: 3.14159, -100.2 und 22.3e-5. Bei der Exponentialdarstellung von Dezimalzahlen (z.B. 22,3 10^{-5}) wird die Zahl aus zwei Teilen aufgebaut: der Mantisse (22,3) und dem Exponenten (-5) zur Basis 10.

Da der Ausdruck 10^{-5} in Programmiersprachen nicht direkt codiert werden kann, umschreibt man ihn durch die Notation e-5. Der Buchstabe e darf in diesem Zusammenhang nicht mit der Eulerzahl verwechselt werden! Er steht hier stellvertretend für die Zahl 10 (Basis des Exponenten). Statt eines Kleinbuchstabens kann auch ein Großbuchstabe verwendet werden: 22.3E-5.

Wegen der Verwendung eines Dezimalpunktes, werden diese Zahlen auch *Gleitpunktzahlen* genannt. In manchen Büchern findet man auch die Begriffe *Gleitkommazahl* oder *Fließkommazahl*. Dies ist aber nicht korrekt, da ja eben *kein* Komma Verwendung findet. In dem folgenden Code können Sie erkennen, dass die Zahl 3000,0 auf unterschiedlichste Art und Weise dargestellt werden kann.

```
# Mehrere Wege zur Darstellung der Zahl 3000,0
x1 = 3000.0
x2 = 3e+3
x3 = 3.0e3
x4 = 3E3
x5 = 3.e+3
print(x1,x2,x3,x4,x5)
```

Die Mantisse kann also auch als Ganzzahl geschrieben werden. Eine Null nach dem Dezimalpunkt kann entfallen. Ebenso das Pluszeichen in dem Exponenten. Zwischen Mantisse und Exponent darf kein Leerzeichen stehen. Und auch nicht vor dem Exponenten! Falls Sie versehentlich einmal ein Leerzeichen codieren, merkt dies IDLE sofort und markiert die fehlerhafte Stelle im Programm mit roter Farbe. Das Programm ist dann nicht ausführbar. Der Exponent muss eine ganze Zahl sein. Ein Ausdruck der Form 3.0e+3.0 ist also falsch und führt ebenfalls zu einer Fehlermeldung. Das folgende Programmfragment enthält einige Fehler. Welche?

```
# Erkennen Sie die Fehler? Korrigieren Sie diese!
x1 = 3000,0
x2 = 3 e+3
x3 = 3.0e 3
x4 = 3E3.0
print(x1,x2,x3,x4)
```

Da ist zunächst das etwas merkwürdige Verhalten von Python im Falle der Zuweisung x1 = 3000,0. Python erkennt dies nicht als Fehler, sondern weist der Variablen x1 ein so genanntes *Tupel* (siehe Kapitel 3.2) zu! Ein Tupel ist ein Verbund aus zwei Zahlen. Als Ergebnis enthält die Variable x1 die beiden Zahlen: 3000 *und* 0.

Auch bei Gleitpunktzahlen können wir den Objekttyp mittels der eingebauten Funktion type() ermitteln. Er heißt intern *float*.

```
>>> type(3000.0)
<class 'float'>
>>> type(3e3)
<class 'float'>
```

Mit der Funktion *float(),* deren Anwendung nachfolgend gezeigt wird, können andere Objekttypen (z.B. Ganzzahlen oder Strings) in Gleitpunktzahlen umgewandelt werden. Wichtig ist bei der Anwendung dieser Funktion, dass das übergebene Argument auch wirklich in eine Gleitpunktzahl umgewandelt werden kann!

```
>>> float(235)
235.0
>>> float("235")
235.0
>>> float("235 Grad")

Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    float("235 Grad")
ValueError: could not convert string to float: '235 Grad'

>>> float("0.754e-3")
0.000754
```

2.1.3 Komplexe Zahlen - Complex

Eine komplexe Zahl z kann als Zusammenfassung zweier reeller Zahlen a und b aufgefasst werden: z = a + bj. Dabei ist a der so genannte Realteil und b der so genannte Imaginärteil. Das Symbol j wird als imaginäre Einheit bezeichnet. Dabei gilt: j = $\sqrt{-1}$. Komplexe Zahlen sind in Python fest eingebaut. Das folgende Programmfragment zeigt, wie man komplexe Zahlen zuweist und anschließend ausgibt. Statt z1 = 3.0 – 5.0j kann auch mittels der Funktion *complex()* geschrieben werden: z1 = complex(3.0, -5.0). Als Ergebnis erhält man die gleiche komplexe Zahl.

```
# Komplexe Zahlen
z1 = 3.0 - 5.0j
z2 = complex(3.0,-5.0)
z3 = 3 - 5j
print(z1, z2, z3)
```

Wie bei Gleitpunktzahlen können wir die Null nach dem Dezimalpunkt (und auch den Dezimalpunkt) weglassen.

Die imaginäre Einheit j (oder J) muss unmittelbar nach der zweiten Zahl folgen. Es ist also unzulässig, ein Leerzeichen einzufügen. Auch die intuitive Schreibweise z = 2 + 3*j führt zu einem Fehler, weil Python dann die Zahl 3 mit einer (vordefinierten) Vari-

ablen j multiplizieren will. Für den Fall, dass diese Variable vorher verwendet wurde, wird diese Multiplikation dann auch ausgeführt! In dem folgenden Programmfragment ist ein Fehler enthalten. Finden und korrigieren Sie diesen! Was geschieht beim Aufruf von *complex(3.0,-5.0j)?*

```
# Komplexe Zahlen - Die Fehler suchen und finden!
j = 3.0 - 5.0j                #1
k = complex(3.0,-5.0j)    #2
m = -2 - 4 j                  #3
print(j, k, m)
```

Die erste Zuweisung (Zeile #1) ist ungewöhnlich, aber nicht falsch. Dort wird einer Variablen j eine komplexe Zahl 3.0-5.0j zugewiesen. Der Buchstabe j tritt also doppelt auf: in Form eines Variablennamens und in Form der imaginären Einheit. Die zweite Zeile (#2) ist ebenfalls nicht falsch. Dort wird eine komplexe Zahl mittels der Funktion complex() erzeugt. Als Realteil wird die Zahl 3.0 und als Imaginärteil die Zahl -5.0j, die jedoch ebenfalls eine komplexe Zahl ist. Python wertet diese Funktion folgendermaßen aus: k = 3.0 + (-5.0j) j = 3.0 + 5.0 = 8.0. Bedenken Sie, dass j*j = -1 ist! Der Fehler findet sich in Form eines Leerzeichens in Zeile #3. Richtig muss es dort heißen: m = -2 - 4j.

Schließlich rufen wir wieder die Funktion type() auf:

```
# Komplexe Zahlen - Funktion type()
j = 3.0 - 5.0j                    #1
k = complex(3.0,-5.0j)        #2
m = -2 - 4j                       #3
print type(j), type(k), type(m)
```

Die Ausgabe dieses Programmfragments lautet:

```
<class 'complex'> <class 'complex'> <class 'complex'>
```

2.1.4 Zeichenketten - Strings

Zeichenketten hatten wir schon im ersten Kapitel kennen gelernt. Eine Zeichenkette wird entweder durch zwei obere Anführungszeichen (“) oder durch zwei Apostrophe (’) eingeschlossen. Zulässige Zuweisungen mit Strings sind also x = “Wuppertal“ oder x = ’Wuppertal’. Ein häufiges Problem besteht darin, dass unsere Programme Texte ausgeben sollen, die ebenfalls Apostrophe oder Anführungszeichen enthalten können. Das kann dann dadurch realisiert werden, dass wir die beiden möglichen Begrenzungszeichen abwechseln lassen. In dem folgenden Beispiel wird auch gezeigt, wie man mit der Funktion type() den Typ eines Strings abfragen kann.

```
>>> x = "Der Programmierer sagte: 'Python ist ein tolles Werkzeug' "
>>> print(x)
Der Programmierer sagte: 'Python ist ein tolles Werkzeug'
>>>type(x)
<class 'str'>
```

Oder als Alternative: x= ’Der Programmierer sagte: “Python ist ein tolles Werkzeug“ ’.

Im ersten Kapitel war schon kurz gezeigt worden, dass Strings addiert (aneinander gehängt, engl. concatenation) werden können:

```
>>> x = "Python"
>>> y = "Programmierung"
>>> leer = " "
>>> z = x+y
>>> u = x+leer+y
>>> print(z)
PythonProgrammierung
>>> print(u)
Python Programmierung
```

Strings können also auch Leerzeichen enthalten, oder – wie oben gezeigt – auch nur aus Leerzeichen bestehen. Weiterhin können Strings auch mit ganzen Zahlen multipliziert werden. Bei der Multiplikation mit n wird der String dann n-mal wiederholt. Hier ein Beispiel:

```
>>> x = "Python"
>>> y = 5*x
>>> print(y)
PythonPythonPythonPythonPython
```

Hierfür gibt es sinnvolle Anwendungen. Viele Programmierer möchten Ausgaben auf dem Computerbildschirm in der folgenden Form erstellen:

```
------------------------------
 Hauptprogramm
------------------------------



------------------------------
 Programm-Ende
------------------------------
>>>
```

Die Überschriften sollen also durch Über- und Unterstriche aus Minuszeichen eingerahmt werden. Natürlich können diese Zeilen durch print-Anweisungen der Form

```
print(“---------------------------------------“)
```

erzeugt werden. Dies ist aber umständlich. Viel einfacher ist folgende Lösung:

```
anzahl_zeichen = 30
strich = anzahl_zeichen * "-"
print(strich)
print(" Hauptprogramm")
print(strich)
```

```
print(); print()
print(strich)
print(" Programm-Ende")
print(strich)
```

Falls die Anzahl der Minuszeichen pro Zeile im gesamten Programm verändert werden sollen, so kann dies in dieser Programmvariante ganz einfach durch die Veränderung der Variablen *anzahl_zeichen* geschehen.

Strings sind so genannte sequentielle Objekttypen. Das bedeutet, dass die Zeichen, die einen String bilden, gewissermaßen durchnummeriert sind. Die folgende Abbildung soll dies erläutern. Wir betrachten die Zeichenkette “Python“.

P	y	t	h	o	n
0	1	2	3	4	5

Abbildung 2.1 Sequentielle Anordnung von Zeichen

Von links beginnend erhält jedes Zeichen einen Index. Das erste Zeichen hat den Index Null; das letzte Zeichen den Index 5. Der letzte Index ist immer um die Zahl 1 kleiner als die Länge der Zeichenkette. Diese interne Organisation der Zeichenketten wird nachfolgend eine wichtige Rolle spielen.

In dem folgenden Abschnitt sollen die wichtigsten Funktionen zum Arbeiten mit Zeichenketten vorgestellt werden.

len(s)
Diese Funktion ermittelt die Länge einer Zeichenkette s.

```
>>> x = "Python Programmierung"
>>> len(x)
21
```

str(o)
Die Funktion str() wandelt beliebige Objekte o in Zeichenketten um. Wir zeigen dies am Beispiel der Umwandlung einer Gleitpunktzahl.

```
>>> y = 120.45
>>> str(y)
'120.45'
```

Neben den genannten *Funktionen* gibt es auch *Methoden* zur Behandlung von Zeichenketten. Methoden sind Funktionen, die im Zusammenhang mit einem bestimmten Objekt aufgerufen werden. Näheres hierzu im Kapitel „Objektorientierte Programmierung“. Hier sind die wichtigsten Methoden für Strings:

s.count(z)
Die Methode count() stellt fest, wie oft die Zeichenkette z in der Zeichenkette s enthalten ist.

```
>>> x = "Hallo Hallo HalloHallo"
>>> x.count("Hallo")
4
```

s.center(w)
Die Methode center() wandelt die Zeichenkette in einen String mit der Länge w um. Die Zeichenkette ist dabei in diesem String zentriert. Dies eignet sich z.B. zur Erzeugung von Überschriften. Der Parameter w muss dabei größer als die Länge des Strings s sein; also w>len(s).

```
>>> x = "Python"
>>> print(x)
Python
>>> y = x.center(14)
>>> print(y)
   Python
```

s.ljust(w), s.rjust(w)
Diese Methoden richten eine Zeichenkette linksbündig (ljust) bzw. rechtsbündig (rjust) in einem String der Länge w aus. Diese Länge w muss größer als die Länge des vorgegebenen Strings s sein; also w>len(s).

```
>>> x = "Python"
>>> y = x.ljust(20)
>>> z = x.rjust(20)
>>> print(y)
Python
>>> print(z)
             Python
```

s.endswith(z), s.startswith(z)
Diese Methoden geben *True* (wahr) zurück, wenn die Zeichenkette s mit der Zeichenkette z endet, bzw. mit der Zeichenkette z beginnt. In dem folgenden Beispiel wird getestet, ob der String x mit “\n“ endet. Zur Erinnerung: diese Zeichenfolge erzeugt bei der Ausgabe des Strings eine neue Zeile („carriage return“). Wird dagegen darauf getestet, ob der String mit „on“ endet, so wird *False* (falsch) ausgegeben. Zuletzt wird getestet, ob der String x mit “Py“ beginnt.

```
>>> x = "Python\n"
>>> x.endswith("on")
False
>>> x.endswith("\n")
True
>>> x.startswith("Py")
True
```

s.find(z)
Die Methode find() sucht den String z in der Zeichenkette s. Wird er gefunden, so gibt diese Methode den Index des ersten Zeichens von z in s zurück. Diese Methode verwendet also die oben dargestellte, sequentielle Organisation der Zeichenketten.

Sachwortverzeichnis

G

H

I

J

K

L

M

T

U

V

W

X

Y

Z