

O'REILLY®

Neuronale Netze

selbst programmieren

EIN VERSTÄNDLICHER EINSTIEG MIT PYTHON



Tariq Rashid
Übersetzung von Frank Langenau

Do it yourself mit Python

»Um etwas wirklich zu verstehen, musst du es selbst machen.«

»Beginne klein ... und wachse dann«

In diesem Abschnitt schaffen wir uns ein eigenes neuronales Netz.

Ohne Computer geht es nicht, da – wie bereits erwähnt – viele Tausende von Berechnungen anfallen. Computer sind geradezu prädestiniert, wenn es darum geht, viele Berechnungen sehr schnell auszuführen, ohne dass sie ermüden oder an Genauigkeit verlieren.

Was der Computer tun soll, sagen wir ihm mit Anweisungen, die er verstehen kann. Allerdings ist es für den Computer recht schwer, menschliche Sprachen wie Englisch, Französisch oder Deutsch genau und widerspruchsfrei zu verstehen. Selbst Menschen haben Schwierigkeiten mit Genauigkeit und Mehrdeutigkeit, wenn sie miteinander reden, sodass es für Computer wenig Hoffnung gibt, es besser zu machen!

Python

Wir entscheiden uns für eine Computersprache namens *Python*. Diese Sprache ist leicht zu lernen und eignet sich daher gut für den Einstieg. Außerdem ist es einfach, die Python-Anweisungen anderer Programmierer zu lesen und zu verstehen. Darüber hinaus ist die Sprache sehr populär und wird in vielen verschiedenen Bereichen eingesetzt, unter anderem in Wissenschaft und Forschung, im Unterricht, für weltweite Infrastruktur sowie zur Datenanalyse und für die künstliche Intelligenz. Python wird vermehrt im Schulunterricht gelehrt, und der äußerst beliebte Raspberry Pi hat Python einem noch größeren Publikum zugänglich gemacht, Schüler und Studenten eingeschlossen.

Der Anhang beinhaltet eine Anleitung, um einen Raspberry Pi Zero für sämtliche Arbeiten einzurichten, die wir in diesem Buch behandeln, damit Sie Ihr eigenes

neuronales Netz mit Python erzeugen können. Ein Raspberry Pi Zero ist ein besonders preiswerter kleiner Computer, der nur etwa 5 Euro kostet. Das ist kein Druckfehler – er kostet wirklich lediglich 5 Euro.

Über Python (oder jede andere Computersprache) gibt es noch viel mehr zu lernen, doch ich beschränke mich hier auf das, was Sie für das Erstellen eines eigenen neuronalen Netzes brauchen. Somit lernen Sie gerade genug Python, um dieses Ziel zu erreichen.

Interaktives Python = IPython

Anstatt Python auf Ihrem Computer Schritt für Schritt einzurichten – einschließlich der verschiedenen Erweiterungen für Mathematik und Bilddarstellung –, greifen wir auf eine fertig gepackte Lösung zurück, genannt *IPython*.

IPython enthält die Programmiersprache Python und mehrere Erweiterungen für gebräuchliche numerische Datenverarbeitung und grafische Darstellung von Daten; dazu gehören auch diejenigen, die wir hier brauchen. Zudem besitzt IPython den Vorzug, interaktive Notebooks zu präsentieren, die sich wie Bleistift und Notizblock verhalten. Diese sind ideal dazu geeignet, Ideen auszuprobieren, Ergebnisse anzuzeigen und dann einige der Ideen erneut zu verändern, alles leicht und unkompliziert. Dadurch brauchen wir uns weder um Programmdateien noch um Interpreter oder Bibliotheken zu kümmern, was uns gegebenenfalls nur von unserem eigentlichen Anliegen ablenkt, insbesondere wenn nicht gleich alles wie erwartet funktioniert.

Die Site *ipython.org* bietet Ihnen ein paar Möglichkeiten an, um fertige IPython-Pakete zu beziehen. Ich verwende das Paket *Anaconda* von *www.continuum.io/downloads*, wie Abbildung 2-1 zeigt.

Möglicherweise hat sich das Erscheinungsbild der Site geändert, seit ich diesen Snapshot aufgenommen habe, lassen Sie sich also nicht abschrecken. Gehen Sie zunächst zu dem Ihrem Computer entsprechenden Abschnitt, der ein Windows-Computer, ein Apple Mac mit OS X oder ein Linux-Computer sein kann. Achten Sie im Abschnitt für Ihren Computer darauf, dass Sie die Python-Version 3.5 und nicht Version 2.7 herunterladen.

Die Akzeptanz von Python 3 nimmt ständig zu und ist zukunftsorientiert. Python 2.7 ist ebenfalls gut eingeführt, doch wir sollten uns an zukünftigen Aufgaben orientieren und Python 3 einsetzen, wann immer das möglich ist, speziell bei neuen Projekten. Da die meisten Computer inzwischen »64-Bit-Gehirne« haben, sollten Sie auch darauf achten, dass Sie diese Version auswählen. Nur Computer, die mehr als zehn Jahre alt sind, brauchen wahrscheinlich noch die veraltete 32-Bit-Version.

Folgen Sie den Anweisungen auf dieser Site, um das Paket auf Ihrem Computer zu installieren. Die Installation von IPython sollte leicht vonstattengehen und auch keine Probleme verursachen.

Anaconda for Windows

PYTHON 2.7	PYTHON 3.5
Windows 64-bit Graphical Installer 387M	Windows 64-bit Graphical Installer 392M
Windows 32-bit Graphical Installer 321M	Windows 32-bit Graphical Installer 316M
Behind a firewall? Use these zipped Windows installers .	

Windows Anaconda Installation

1. Download the installer.
2. Double-click the .exe file to install Anaconda and follow the instructions on the screen.
3. Optional: [Verify data integrity with MD5](#).

Anaconda for OS X

PYTHON 2.7	PYTHON 3.5
Mac OS X 64-bit Graphical Installer 274M (OS X 10.7 or higher)	Mac OS X 64-bit Graphical Installer 267M (OS X 10.7 or higher)
Mac OS X 64-bit Command-Line Installer 239M (OS X 10.7 or higher)	Mac OS X 64-bit Command-Line Installer 233M (OS X 10.7 or higher)

Abbildung 2-1: Das IPython-Paket Anaconda herunterladen

Ein sehr sanfter Start mit Python

Ich gehe jetzt davon aus, dass Sie auf IPython zugreifen können, wenn Sie die Anweisungen für die Installation befolgt haben.

Notebooks

Nachdem Sie Python gestartet und auf *New Notebook* geklickt haben, erscheint ein leeres *Notebook*, wie es Abbildung 2-2 zeigt.

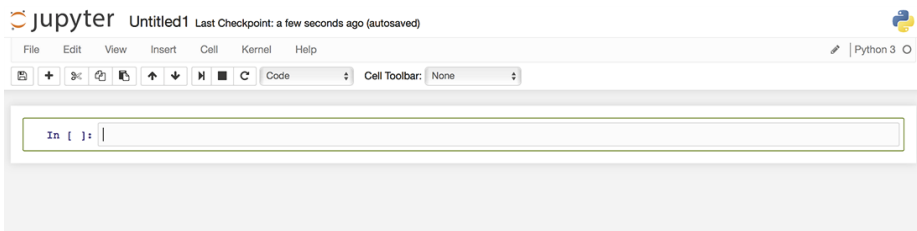


Abbildung 2-2: Ein neues Notebook

Das Notebook ist interaktiv, wartet also darauf, dass Sie eine Anweisung eingeben, führt sie aus, gibt dann die Antwort zurück und wartet erneut auf eine Anweisung oder Frage. Es verhält sich wie ein elektrischer Butler mit Talent für Arithmetik, der niemals müde wird.

Wenn Sie eine Aufgabe zu lösen haben, die schon ein wenig komplizierter ist, sollten Sie sie in Abschnitte zerlegen. Dadurch lassen sich die Gedanken leichter ordnen, und man findet auch schneller heraus, welcher Teil eines großen Projekts schiefgegangen ist. Bei IPython nennt man diese Abschnitte *Zellen* (Cells). Das in Abbildung 2-2 gezeigte IPython-Notebook hat anfangs eine leere Zelle, und eine blinkende Einfügemarke (ein sogenanntes Caret) zeigt die Eingabebereitschaft an.

Erteilen Sie nun dem Computer eine Anweisung! Lassen Sie ihn zwei Zahlen multiplizieren, beispielsweise 2 mal 3. Tippen Sie »2 * 3« in die Zelle ein (ohne die Anführungszeichen) und klicken Sie dann auf die Schaltfläche *run cell*, die aussieht wie ein Wiedergabesymbol. Der Computer wird schnell herausfinden, was Sie mit dieser Anweisung meinen, und das Ergebnis zurückgeben (siehe Abbildung 2-3).

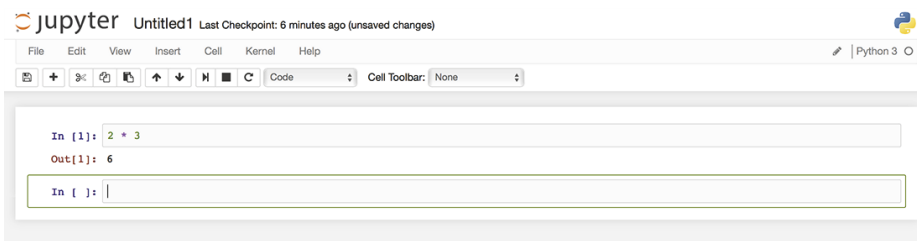


Abbildung 2-3: Eingabe einer Anweisung und Anzeige des Ergebnisses

Wie Abbildung 2-3 zeigt, erscheint die richtige Antwort »6«. Sie haben soeben Ihrem Computer mithilfe von Python Ihre erste Anweisung gegeben und ein richtiges Ergebnis erhalten – Ihr erstes Computerprogramm!

Lassen Sie sich nicht dadurch stören, dass IPython die Frage mit In [1] und die Antwort mit Out [1] kennzeichnet. Das dient lediglich als Erinnerung dafür, was Sie gefragt haben (In für Input = Eingabe) und was Python geantwortet hat (Out für Output = Ausgabe). Die Zahlen geben die Reihenfolge der Fragen und Antworten an.

ten an, was nützlich ist, wenn Sie in Ihrem Notebook etwas suchen, Anweisungen anpassen und erneut ausführen wollen.

Einfaches Python

Wir haben es ehrlich gemeint, als wir gesagt haben, dass Python eine einfache Computersprache sei. In der nächsten Zelle, die mit `In []` bezeichnet ist, geben Sie den folgenden Code ein und klicken auf *run cell*. Mit dem Begriff *Code* meint man üblicherweise Anweisungen, die in einer Computersprache geschrieben sind. Wenn es Ihnen (wie mir) zu umständlich ist, den Mauszeiger zu verschieben, um auf die Schaltfläche *run cell* zu klicken, können Sie stattdessen die Tastenkombination `[Strg] + [↵]` verwenden.

```
print("Hello World!")
```

Sie sollten eine Antwort erhalten, die einfach den Text `Hello World!` ausgibt, wie Abbildung 2-4 zeigt.

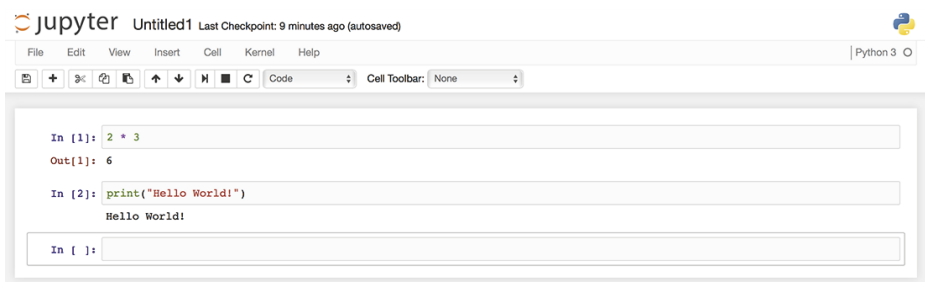


Abbildung 2-4: Ausgabe der eingegebenen Zeichenfolge

In Abbildung 2-4 sehen Sie, dass beim Ausführen der zweiten Anweisung zur Ausgabe von `Hello World!` die vorherige Zelle mit der eingegebenen Anweisung und der ausgegebenen Antwort nicht entfernt wurde. Das ist nützlich, wenn man eine Lösung schrittweise aus mehreren Teilen aufbaut.

Wir wollen nun sehen, was es mit dem folgenden Code auf sich hat, der ein zentrales Konzept einführt. Geben Sie ihn in eine neue Zelle ein und führen Sie ihn aus. Falls keine leere Zelle zu sehen ist, klicken Sie auf die Schaltfläche, die ein Pluszeichen zeigt und mit *insert cell below* (Zelle unten einfügen) beschriftet ist.

```
x = 10
print(x)
print(x+5)

y = x+7
print(y)

print(z)
```

Die erste Zeile, $x = 10$, sieht wie eine mathematische Anweisung aus, die besagt, dass x gleich 10 ist. In Python bedeutet sie, dass x auf 10 gesetzt wird, d. h. der Wert 10 in ein virtuelles Fach namens x gelegt wird. Das ist genauso einfach, wie es Abbildung 2-5 symbolisiert.

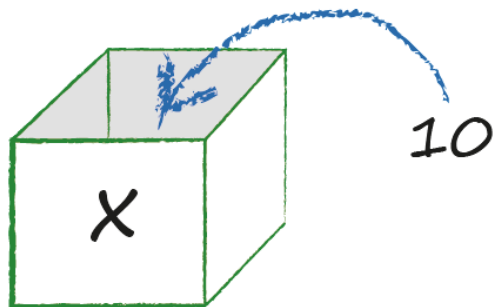


Abbildung 2-5: Eine Zahl für später aufbewahren

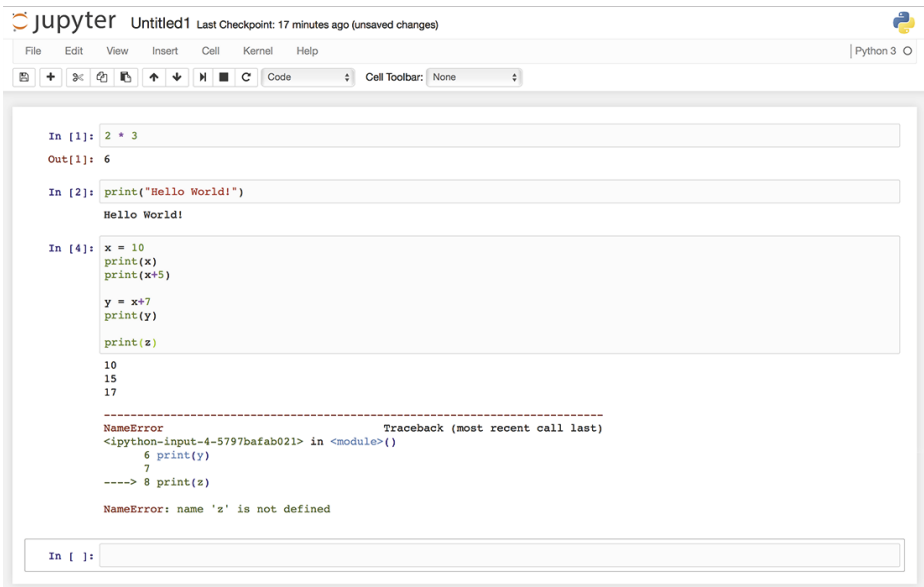
Diese 10 bleibt bis auf Weiteres dort drin. Die Anweisung `print(x)` haben wir schon weiter oben verwendet. Sie soll den Wert von x ausgeben, der 10 ist. Warum gibt sie nicht einfach den Buchstaben x aus? Weil Python immer bemüht ist, auszuwerten, was möglich ist, und x kann zum Wert 10 ausgewertet werden. Deshalb erscheint die 10 in der Ausgabe. Die nächste Zeile `print (x + 5)` wertet $x + 5$ aus, was $10 + 5$ oder 15 ergibt, sodass wir in der Ausgabe 15 erwarten.

Bei der nächsten Anweisung $y = x + 7$ sollte es nicht schwierig sein, das Ergebnis herauszufinden. Wir halten uns wieder an die Konvention, dass Python alles auswertet, was möglich ist. Die Anweisung soll einem neuen Fach mit der Bezeichnung y einen Wert zuweisen. Doch welchen Wert? Der Ausdruck auf der rechten Seite des Gleichheitszeichens lautet $x + 7$, was $10 + 7$ oder 17 ist. Somit enthält y den Wert 17, und die Anweisung in der nächsten Zeile sollte ihn ausgeben.

Was passiert in der Zeile `print(z)`, wenn wir z (im Unterschied zu x und y) noch keinen Wert zugewiesen haben? Wir bekommen eine höfliche Fehlermeldung, die uns über den Fehler informiert, wobei Python versucht, möglichst hilfreich zu sein, damit wir den Fehler korrigieren können. Leider muss man sagen, dass die Fehlermeldungen der meisten Computersprachen zwar darauf abzielen, hilfreich zu sein, damit aber nicht immer erfolgreich sind.

Abbildung 2-6 zeigt die Ergebnisse des obigen Codes, einschließlich der hilfreichen, höflichen Fehlermeldung `name 'z' is not defined` (Name ' z ' ist nicht definiert).

Diese Fächer mit Beschriftungen wie x und y , die Werte wie 10 und 17 aufnehmen, werden *Variablen* genannt. Variablen dienen in Computersprachen dazu, eine Folge von Anweisungen möglichst allgemein formulieren zu können, genauso wie Mathematiker zu x und y greifen, um allgemeine Aussagen zu treffen.



```
In [1]: 2 * 3
Out[1]: 6

In [2]: print("Hello World!")
Hello World!

In [4]: x = 10
        print(x)
        print(x+5)

        y = x+7
        print(y)

        print(z)

10
15
17

-----
NameError                                Traceback (most recent call last)
<ipython-input-4-5797bafab021> in <module>()
      6 print(y)
      7
----> 8 print(z)

NameError: name 'z' is not defined

In [ ]:
```

Abbildung 2-6: Die Ergebnisse des oben angegebenen Codebeispiels inklusive einer Fehlermeldung

Arbeiten automatisieren

Computer sind hervorragend dafür geeignet, ähnliche Aufgaben viele Male auszuführen – das macht ihnen nichts aus, und sie sind sehr schnell im Vergleich zu Menschen mit Taschenrechnern!

Probieren wir einmal, ob wir einen Computer dazu bringen können, die ersten zehn Quadratzahlen auszugeben, beginnend mit 0 quadriert, 1 quadriert, dann 2 quadriert usw. Wir erwarten eine Ausgabe wie 0, 1, 4, 9, 16, 25 usw.

Wir könnten die Berechnungen natürlich selbst durchführen und sie mit einem Satz von Anweisungen wie `print(0)`, `print(1)`, `print(4)` usw. ausgeben. Das würde funktionieren, doch wir hätten es versäumt, den Computer die Berechnungen für uns ausführen zu lassen. Darüber hinaus hätten wir die Gelegenheit verpasst, einen generischen Satz von Anweisungen zu erzeugen, um die Quadrate der Zahlen bis zu jedem vorgegebenen Wert auszugeben. Dazu müssen wir einige neue Ideen aufgreifen. Gehen wir es also behutsam an.

Geben Sie den folgenden Code in die nächste Zelle ein und führen Sie ihn aus:

```
list( range(10) )
```

Das Ergebnis sollte eine Liste mit zehn Zahlen von 0 bis 9 sein (siehe Abbildung 2-7). Das ist großartig, weil wir dem Computer die Aufgabe übertragen haben, die Liste zu erstellen. Wir mussten das nicht selbst tun. Wir sind der Boss, und der Computer ist unser Diener!


```
In [8]: list( range(10) )  
Out[8]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Abbildung 2-7: Vom Computer erzeugte Liste mit zehn Zahlen

Vielleicht fragen Sie sich, warum die Liste von 0 bis 9 und nicht von 1 bis 10 reicht. Das hängt damit zusammen, dass viele Computer 0-basiert arbeiten, d. h. bei 0 und nicht bei 1 anfangen zu zählen. Ich habe schon mehrfach fehlerhaften Code produziert, weil ich davon ausgegangen bin, dass eine Computerliste bei 1 und nicht bei 0 beginnt.

Solche geordneten Listen sind beispielsweise nützlich, um mitzuzählen, wenn Berechnungen viele Male laufen oder iterative Funktionen anzuwenden sind.

Sicherlich haben Sie bemerkt, dass hier das Schlüsselwort `print` fehlt, das bei der Ausgabe des Texts `Hello World!` erforderlich war, das wir aber auch bei der Auswertung von `2 * 3` weglassen konnten. Die Verwendung des Schlüsselworts `print` kann optional sein, wenn wir mit Python im interaktiven Modus arbeiten. Denn Python weiß dann, dass wir das Ergebnis der ausgeführten Anweisungen sehen wollen.

Um den Computer zu veranlassen, Dinge wiederholt auszuführen, verwendet man meistens spezielle Codestrukturen – die sogenannten *Schleifen*. Dieser Begriff deutet bereits an, dass etwas wiederholt wird, gegebenenfalls endlos. Anstatt eine Schleife zu definieren, ist es aber am besten, ein einfaches Beispiel zu zeigen. Geben Sie den folgenden Code in eine neue Zelle ein und führen Sie ihn aus:

```
for n in range(10):  
    print(n)  
    pass  
print("done")
```

In diesem Code sind drei neue Elemente enthalten. Die erste Zeile enthält den Ausdruck `range(10)`, den wir weiter oben bereits gesehen haben. Damit wird die schon bekannte Liste der Zahlen von 0 bis 9 erzeugt.

Der Code `for n in` leitet eine Schleife ein. In diesem Fall führt sie etwas für jede Zahl in der Liste aus und zählt die Durchläufe mit, indem sie den aktuellen Wert der Variablen `n` zuweist. Variablen haben Sie bereits weiter oben kennengelernt. Die Schleife führt hier also beim ersten Durchlauf die Anweisung `n=0` aus, dann `n=1` usw. bis `n=9`, dem letzten Element in der Liste.

Die Anweisung `print(n)` in der nächsten Zeile ist ebenfalls schon bekannt. Sie gibt den Wert von `n` aus. Wir erwarten, dass alle Zahlen in der Liste ausgegeben werden. Allerdings ist die Anweisung `print(n)` eingerückt. In Python ist dies wichtig, da Einrückungen anzeigen, welche Anweisungen anderen untergeordnet sind. Hier ist `print(n)` der mit `for n in` erzeugten Schleife untergeordnet. Die Anweisung

pass signalisiert das Ende der Schleife. Die nächste Anweisung kehrt zurück zur normalen Einrückungsebene und ist somit nicht Teil der Schleife. Folglich erwarten wir, dass »done« nur einmal und nicht zehnmal ausgegeben wird. Abbildung 2-8 zeigt die Ausgabe, die wie erwartet aussieht.

```
Out[8]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [12]: for n in range(10):
          print(n)
          pass
          print("done")

0
1
2
3
4
5
6
7
8
9
done
```

Abbildung 2-8: Die Ausgabe der Beispielschleife

Es sollte inzwischen klar sein, dass sich die Quadrate mit $n*n$ ausgeben lassen. Um die Ausgabe verständlicher zu formulieren, kann man sie in der Art »The square of 3 is 9« gestalten. Der folgende Code zeigt diese Änderung an der print-Anweisung, die in der Schleife wiederholt wird. Die Variablen sind nicht in Anführungszeichen eingeschlossen und werden folglich ausgewertet.

```
for n in range(10):
    print("The square of", n, "is", n*n)
    pass
print("done")
```

Abbildung 2-9 zeigt das Ergebnis.

```
In [13]: for n in range(10):
          print("The square of", n, "is", n*n)
          pass
          print("done")

The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
done
```

Abbildung 2-9: Das Ergebnis der Schleifenanweisung mit verständlicherer Ausgabeanweisung

Das ist schon recht eindrucksvoll! Mit einem ziemlich knappen Satz von Anweisungen bringen wir den Computer dazu, viele Aufgaben sehr schnell auszuführen.

Es ist auch ganz einfach, die Anzahl der Schleifendurchläufe zu vergrößern, und zwar mit der Anweisung `range(50)` oder sogar `range(1000)`. Probieren Sie es aus!

Kommentare

Bevor wir weitere interessante Python-Befehle untersuchen, sollten Sie sich den folgenden Beispielcode ansehen:

```
# the following prints out the cube of 2
print(2**3)
```

Die erste Zeile beginnt mit dem Hashsymbol `#`. Python ignoriert alle Zeilen, die mit diesem Symbol beginnen. Solche Zeilen sind keineswegs nutzlos. Hier kann man hilfreiche Kommentare in den Code einfügen, um ihn verständlicher zu machen für andere Leser oder auch für sich selbst, wenn man den Code später überarbeiten muss.

Glauben Sie mir, Sie werden dankbar sein dafür, Ihren Code kommentiert zu haben, vor allem bei komplexeren oder weniger durchsichtigen Codeabschnitten. Ich habe schon oft versucht, meinen eigenen Code wieder zu entschlüsseln, und mich dabei gefragt: »Was habe ich mir eigentlich dabei gedacht?«

Funktionen

Kapitel 1 dieses Buchs ist ausführlich auf mathematische Funktionen eingegangen. Wir haben sie uns als Maschinen vorgestellt, die Eingaben übernehmen, in bestimmter Weise verarbeiten und das Ergebnis ausgeben. Diese Funktionen waren eigenständig und konnten immer wieder verwendet werden.

Viele Computersprachen, Python eingeschlossen, erleichtern es, wiederverwendbare Computeranweisungen zu erzeugen. Wie mathematische Funktionen sind diese wiederverwendbaren Codeabschnitte eigenständig, wenn man sie ordnungsgemäß definiert, und sie erlauben es, kürzeren und eleganteren Code zu schreiben. Warum kürzeren Code? Weil es besser ist, eine Funktion mehrfach unter ihrem Namen aufzurufen, als den gesamten Funktionscode entsprechend oft auszusprechen.

Und was meinen wir mit »ordnungsgemäß definiert«? Es bedeutet, sich klar darüber zu sein, welche Arten von Eingabe eine Funktion erwartet und welche Art von Ausgabe sie produziert. Manche Funktionen übernehmen ausschließlich Zahlen als Eingabe, sodass man ihnen kein Wort anbieten kann, das aus Buchstaben besteht.

Auch hier ist es am besten, dieses einfache Konzept einer *Funktion* an einem leicht verständlichen Beispiel deutlich zu machen. Geben Sie den folgenden Code ein und führen Sie ihn aus:

```
# function that takes 2 numbers as input
# and outputs their average
def avg(x,y):
    print("first input is", x)
    print("second input is", y)
    a = (x + y) / 2.0
    print("average is", a)
    return a
```

Was bewirkt dieser Code? Die beiden ersten Zeilen, die mit # beginnen, ignoriert Python. Sie können hier jedoch Kommentare für zukünftige Leser hinterlassen. Die nächste Anweisung, `def avg(x,y)`, teilt Python mit, dass wir eine neue wiederverwendbare Funktion definieren. Dabei steht das Schlüsselwort `def` für »definieren«, und `avg` (als Abkürzung des englischen Worts *average*, Mittelwert) ist der Name, den wir der Funktion geben. Wir hätten sie auch »banana« oder »pluto« nennen können, doch sinnvoller sind Namen, die die Aufgabe der Funktion andeuten. Der geklammerte Abschnitt nach dem Funktionsnamen, `(x,y)`, sagt Python, dass diese Funktion zwei Eingaben übernimmt, die in der sich anschließenden Definition der Funktion `x` und `y` heißen. In manchen Programmiersprachen muss man angeben, welchen Typ diese Eingaben haben. In Python ist das nicht erforderlich. Allerdings beschwert sich Python später, wenn man eine Variable missbräuchlich verwenden will, beispielsweise ein Wort wie eine Zahl behandelt oder ähnlich unsinnige Dinge anstellt.

Da wir Python angekündigt haben, dass wir eine Funktion definieren wollen, müssen wir nun auch angeben, was die Funktion tun soll. Wie der obige Code zeigt, ist diese Definition der Funktion eingerückt. Manche Sprachen spezifizieren durch Klammern, welche Anweisungen zu welchen Teilen eines Programms gehören, doch die Python-Entwickler waren der Ansicht, dass Unmengen von Klammern unangenehm für das Auge sind und Einrückungen die Struktur eines Programms verständlicher darstellen. Die Meinungen gehen hier weit auseinander – manche Programmierer fühlen sich durch eine derartige Einrückungsphilosophie eingeengt. Ich aber liebe sie! Diese Methode gehört zu den menschenfreundlichsten Konzepten, die in der manchmal eigenwilligen Welt der Computerprogrammierung aufgetaucht sind!

Die Definition der Funktion `avg(x,y)` ist leicht zu verstehen, da sie nur zwei bereits bekannte Elemente verwendet. Sie gibt die beiden Zahlen aus, die die Funktion bei ihrem Aufruf übergeben bekommt. Um den Mittelwert zu bilden, ist es gar nicht notwendig, die beiden Zahlen auszugeben, doch ich möchte damit verdeutlichen, was innerhalb der Funktion passiert. Die Anweisung in der nächsten Codezeile berechnet $(x+y) / 2.0$ und weist den Ergebniswert der Variablen `a` zu. Die nächste Anweisung gibt den Mittelwert aus, um wieder den Ablauf im Code nachvollziehbar zu machen. Schließlich bildet die letzte Codezeile das Ende der Funktion. Mit `return a` wird Python gesagt, was die Funktion zurückgeben soll, genau wie bei den weiter oben betrachteten Maschinen.

Wenn Sie diesen Code ausführen, passiert praktisch nichts. Er erzeugt keine Zahlen, denn wir haben die Funktion lediglich definiert, aber noch nicht verwendet. Immerhin hat Python diese Funktion zur Kenntnis genommen und hält sie bereit, wenn wir sie nutzen wollen.

Geben Sie in die nächste Zelle `avg(2,4)` ein, um diese Funktion mit den Eingabewerten 2 und 4 auszuführen. Übrigens: In der Welt der Computerprogrammierung sagt man, wenn der Code einer Funktion ausgeführt werden soll, »eine Funktion aufrufen«. Die Ausgabe sollte dem entsprechen, was wir erwarten, d. h., die Funktion gibt die beiden Eingabewerte und den berechneten Mittelwert aus. Außerdem erscheint die Antwort der Funktion in einer eigenen Ausgabezeile, weil eine interaktive Python-Sitzung den Wert ausgibt, den eine aufgerufene Funktion als Ergebnis liefert. Abbildung 2-10 zeigt die Funktionsdefinition und die Ergebnisse für die beiden Aufrufe der Funktion mit `avg(2,4)` und `avg(200, 301)`. Probieren Sie die Funktion mit eigenen Eingabewerten selbst einmal aus.

```
In [20]: # function that takes 2 numbers as input
# and outputs their average
def avg(x,y):
    print("first input is", x)
    print("second input is", y)
    a = (x + y) / 2.0
    print("average is", a)
    return a
```

```
In [21]: avg(2,4)

first input is 2
second input is 4
average is 3.0
```

```
Out[21]: 3.0
```

```
In [23]: avg(200,301)

first input is 200
second input is 301
average is 250.5
```

```
Out[23]: 250.5
```

Abbildung 2-10: Definition der Funktion `avg` und zwei Beispiele für den Aufruf der Funktion

Der Funktionscode, der den Mittelwert berechnet, dividiert die Summe der beiden Eingaben durch 2.0 und nicht einfach durch 2. Warum muss das so sein? Das hängt mit einer Eigenheit von Python zusammen, die mir allerdings nicht gefällt. Wenn man lediglich 2 schreibt, rundet Python das Ergebnis auf die nächste ganze Zahl ab, weil Python die einfache 2 als Ganzzahl betrachtet. Bei `avg(2,4)` spielt das zwar keine Rolle, weil $6 / 2$ gleich 3 ist, also eine ganze Zahl. Doch bei `avg(200,301)` ergibt sich der Mittelwert zu $501 / 2$, was 250,5 sein sollte, aber zu 250 abgerundet würde. Behalten Sie dies im Hinterkopf, wenn sich der eigene Code nicht wie erwartet verhält. Mit der Division durch 2.0 teilen wir Python mit,

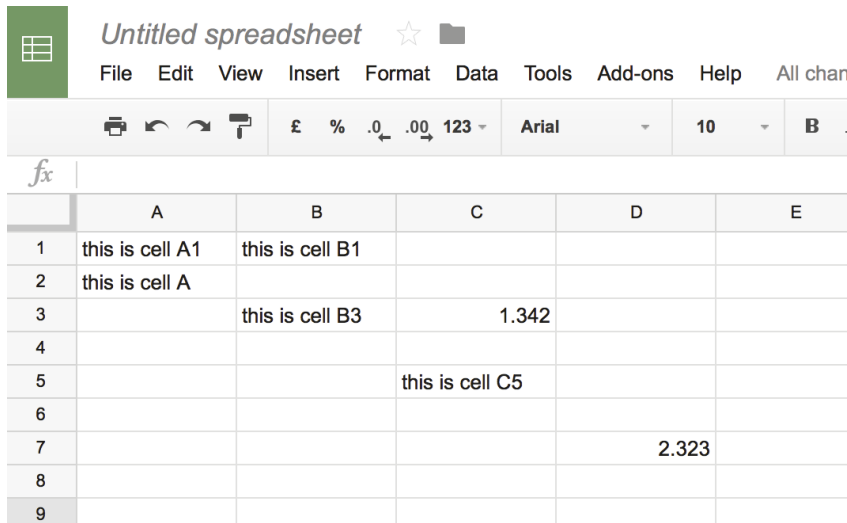
dass wir Zahlen mit Nachkommastellen verarbeiten und nicht auf ganze Zahlen runden möchten.

Gehen wir einen Schritt zurück und gratulieren wir uns. Denn wir haben eine wiederverwendbare Funktion definiert – eines der wichtigsten und leistungsfähigsten Elemente sowohl in der Mathematik als auch in der Computerprogrammierung.

Wir nutzen wiederverwendbare Funktionen, wenn wir unser neuronales Netz codieren. Zum Beispiel ist eine wiederverwendbare Funktion sinnvoll, die den Wert der Sigmoid-Aktivierungsfunktion berechnet, sodass wir diese Funktion viele Male aufrufen können.

Arrays

Arrays sind einfach Tabellen von Werten, und sie erweisen sich als eine wirklich praktische Einrichtung. Wie bei Tabellen üblich, spricht man bestimmte Zellen unter ihrer Zeilen- und Spaltennummer an. Diese Methode kennen Sie sicherlich aus Tabellenkalkulationen, in denen Sie auf Zellen zum Beispiel mit B1 oder C5 verweisen und die Werte dieser Zellen auch in Berechnungen verwenden können, beispielsweise mit C3+D7 (siehe Abbildung 2-11).



	A	B	C	D	E
1	this is cell A1	this is cell B1			
2	this is cell A				
3		this is cell B3	1.342		
4					
5			this is cell C5		
6					
7				2.323	
8					
9					

Abbildung 2-11: Spalten und Zeilen in einer Tabellenkalkulation

Wenn wir unser neuronales Netz codieren, stellen wir die Matrizen der Eingangssignale, der Gewichte und der Ausgangssignale ebenfalls als Arrays dar. Doch damit nicht genug. Arrays verwenden wir auch, um die Signale innerhalb der neuronalen Netze zu repräsentieren, wenn sie vorwärts weitergeleitet werden. Das Gleiche gilt für die Fehler, die in Rückwärtsrichtung durch das Netz laufen.

Machen wir uns also mit ihnen vertraut. Geben Sie den folgenden Code ein und führen Sie ihn aus:

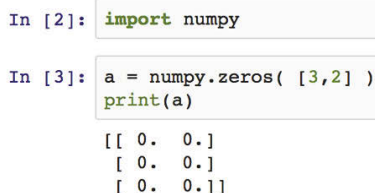
```
import numpy
```

Was bewirkt diese Anweisung? Der `import`-Befehl weist Python an, zusätzliche Kräfte von einer anderen Stelle zu mobilisieren, d. h. neue Tools in das Repertoire einzufügen. Manchmal sind diese Zusatztools zwar schon Bestandteil von Python, sie stehen aber nicht unmittelbar zur Verfügung, um Python möglichst schlank und klein zu halten. Zusätzliche Funktionalität bindet man nur dann ein, wenn man sie auch wirklich braucht. Oftmals sind diese zusätzlichen Tools keine Kernbestandteile von Python, sondern werden von anderen Programmierern als nützliche Erweiterungen geschaffen und bereitgestellt, sodass sie jedermann nutzen kann. Der obige Code importiert einen zusätzlichen Satz von Tools, die in einem Modul namens `numpy` verpackt sind. Das beliebte Paket `numpy` enthält zweckmäßige Erweiterungen wie Arrays und die Funktionalität, um Berechnungen mit ihnen anzustellen.

In die nächste Zelle geben Sie den folgenden Code ein:

```
a = numpy.zeros( [3,2] )
print(a)
```

Dieser Code greift auf das importierte `numpy`-Modul zurück, um ein Array mit drei Zeilen und zwei Spalten zu erzeugen, bei dem alle Werte auf null gesetzt sind. Das gesamte Array wird dann der Variablen `a` zugewiesen. Die zweite Anweisung gibt `a` aus. Wie Abbildung 2-12 zeigt, ist das Array voll von Nullen und erscheint in einer Anordnung, die wie eine Tabelle mit drei Zeilen und zwei Spalten aussieht.



```
In [2]: import numpy

In [3]: a = numpy.zeros( [3,2] )
        print(a)

[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

Abbildung 2-12: Ein Array erzeugen, initialisieren und ausgeben

Als Nächstes modifizieren wir den Inhalt dieses Arrays und ändern einige dieser Nullen in andere Werte. Der folgende Code zeigt, wie man auf bestimmte Zellen zugreifen kann, um sie mit neuen Werten zu überschreiben. Das geschieht in der gleichen Weise, wie man auf Zellen in einer Tabellenkalkulation oder auf Felder einer Straßenkarte verweist.

```
a[0,0] = 1
a[0,1] = 2
a[1,0] = 9
a[2,1] = 12
print(a)
```

Die erste Codezeile aktualisiert die Zelle in Zeile 0 und Spalte 0 des Arrays mit dem Wert 1 und überschreibt dabei, was vorher in dieser Zelle stand. Die anderen Codezeilen sind ähnliche Aktualisierungen. Die Anweisung in der letzten Codezeile gibt das geänderte Array mit `print(a)` aus. Abbildung 2-13 zeigt, wie das Array nach diesen Änderungen aussieht.

```
In [2]: import numpy
```

```
In [3]: a = numpy.zeros( [3,2] )  
print(a)
```

```
[[ 0.  0.]  
 [ 0.  0.]  
 [ 0.  0.]
```

```
In [4]: a[0,0] = 1  
a[0,1] = 2  
a[1,0] = 9  
a[2,1] = 12  
print(a)
```

```
[[ 1.  2.]  
 [ 9.  0.]  
 [ 0. 12.]
```

Abbildung 2-13: Anweisungen zum Aktualisieren (Überschreiben) einzelner Arrayzellen und das geänderte Array

Sie wissen nun, wie man den Wert von Zellen in einem Array festlegt. Doch wie lässt sich der Inhalt der jeweiligen Zellen lesen, ohne das gesamte Array ausgeben zu müssen? Das kennen Sie bereits. Sie verwenden einfache Ausdrücke wie `a[1,2]` oder `a[2,1]`, um auf den Inhalt dieser Zellen zu verweisen. Das können Sie dann ausgeben oder anderen Variablen zuweisen. Dieser Code zeigt, wie das geht:

```
print(a[0,1])  
v = a[1,0]  
print(v)
```

Wie die Ausgabe in Abbildung 2-14 zeigt, liefert die erste `print`-Anweisung den Wert 2.0, der in der Zelle `[0,1]` steht. Dann weist der Code den Wert von `a[1,0]` der Variablen `v` zu und gibt den Wert dieser Variablen aus. Wie erwartet, erscheint 9.0 als Ausgabe.

```
In [5]: print(a[0,1])  
v = a[1,0]  
print(v)
```

```
2.0  
9.0
```

Abbildung 2-14: Auf einzelne Elemente eines Arrays zugreifen

Die Nummerierung der Spalten und Zeilen beginnt bei 0 statt bei 1. Links oben befindet sich also Zelle [0,0] und nicht Zelle [1,1]. Dementsprechend hat die Zelle rechts unten die Indizes [2,1] und nicht [3,2]. Das hat mich so manches Mal kalt erwischt, weil ich regelmäßig vergesse, dass viele Dinge in der Computerwelt mit 0 und nicht mit 1 beginnen. Hätten wir versucht, auf `a[3,2]` zu verweisen, hätte uns eine Fehlermeldung darauf hingewiesen, dass wir auf eine nicht vorhandene Zelle zuzugreifen versuchen. Das Gleiche passiert, wenn wir die Spalten und Zeilen vertauschen. Um zu sehen, wie die Fehlermeldung aussieht, versuchen wir, auf `a[0,2]` zuzugreifen (siehe Abbildung 2-15).

```
In [6]: # trying to look up an array element that doesn't exist
a[0,2]

-----
IndexError                                Traceback (most recent call last)
<ipython-input-6-489d1c44975f> in <module>()
      1 # trying to look up an array element that doesn't exist
----> 2 a[0,2]

IndexError: index 2 is out of bounds for axis 1 with size 2
```

Abbildung 2-15: Fehlermeldung beim Zugriff auf ein nicht vorhandenes Arrayelement

Arrays – oder Matrizen – bieten sich für neuronale Netze an, weil wir damit die Anweisungen für unzählige Berechnungen vereinfachen können, um die Signale vorwärts und die Fehler rückwärts durch ein Netz zu leiten. Kapitel 1 dieses Buchs hat das bereits ausführlich erläutert.

Arrays grafisch darstellen

Genau wie bei großen Tabellen oder Listen ist es auch bei großen Arrays wenig aufschlussreich, das reine Zahlenmaterial zu betrachten. Um auf einen Blick erfassen zu können, was die Zahlen bedeuten, sollte man die Daten visualisieren. So kann man sich zweidimensionale Arrays mit Zahlen als zweidimensionale Oberflächen vorstellen, die abhängig vom Wert in jeder Zelle eingefärbt ist. Dabei lässt sich festlegen, wie in welcher Farbe Sie den Wert einer Zelle anzeigen möchten. Das kann anhand einer einfachen Farbskala geschehen, oder Sie färben alles weiß ein außer bei den Werten, die über einem bestimmten Schwellwert liegen und dann in Schwarz erscheinen.

Wir wollen nun das oben erzeugte kleine 3-mal-2-Array grafisch darstellen.

Vorher müssen wir Python noch um die Fähigkeit erweitern, Grafiken zu zeichnen. Dazu *importieren* wir zusätzlichen Python-Code, den andere Programmierer geschrieben haben. Man kann sich das so vorstellen, dass wir uns ein Rezeptbuch von einem Freund borgen, um es in unser Bücherregal zu stellen. Somit bietet unser Bücherregal nun zusätzliche Inhalte, die uns in die Lage versetzen, ein größeres Speisenangebot anzubieten, als es vorher möglich war.

Die folgende Anweisung importiert die Funktionalität zur Darstellung von Grafiken:

```
import matplotlib.pyplot
```

Hier ist `matplotlib.pyplot` der Name des neuen »Rezeptbuchs«, das wir uns ausborgen. Man sagt auch »ein Modul importieren« oder »eine Bibliothek importieren«. Dabei geht es immer um den zusätzlichen Python-Code, den Sie in Ihr Programm einbinden, d. h. importieren. Wenn Sie professionell mit Python arbeiten, importieren Sie oftmals zusätzliche Funktionalität, um sich das Leben einfacher zu machen, indem Sie nützlichen Code von anderen wiederverwenden. Aber Sie können auch selbst nützlichen Code schreiben, um ihn für andere bereitzustellen!

Des Weiteren müssen wir dafür sorgen, dass IPython die Grafiken im Notebook zeichnet und nicht in einem separaten externen Fenster. Das spezifizieren wir explizit wie folgt:

```
%matplotlib inline
```

Nun können wir dieses Array darstellen. Geben Sie den folgenden Code ein und führen Sie ihn aus:

```
matplotlib.pyplot.imshow(a, interpolation="nearest")
```

Die Anweisung `imshow()` erzeugt eine Grafik, und der erste Parameter ist das Array, das wir ausgeben wollen. Der Parameter `interpolation` weist Python an, die Farben nicht zu überblenden. In der Standardeinstellung versucht Python, durch Überblenden ein weicheres Bild zu erzeugen. Abbildung 2-16 zeigt die Ausgabe.

```
In [8]: import matplotlib.pyplot
        %matplotlib inline
```

```
In [9]: matplotlib.pyplot.imshow(a, interpolation="nearest")
```

```
Out[9]: <matplotlib.image.AxesImage at 0x108917710>
```

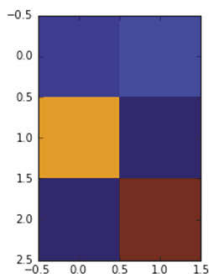


Abbildung 2-16: Die grafische Ausgabe des Arrays

Das ist spannend! Unsere erste Grafikausgabe zeigt das 3-mal-2-Array als farbige Felder. Arrayelemente, die den gleichen Wert enthalten, erscheinen auch in der

gleichen Farbe. Später visualisieren wir mit genau der gleichen `imshow()`-Anweisung ein Array von Werten, die wir in unser neuronales Netz einspeisen.

Das IPython-Paket bringt zahlreiche Werkzeuge für die Datenvisualisierung mit. Sie sollten sich diese Tools ansehen und einige davon auch ausprobieren, um ein Gefühl für das breite Spektrum möglicher Grafikoperationen zu bekommen. Selbst die `imshow()`-Anweisung bietet viele Optionen, mit denen Sie sich befassen sollten, beispielsweise die Verwendung verschiedener Farbpaletten.

Objekte

Wir kommen nun zu einem weiteren Python-Konzept: *Objekte*. Objekte ähneln wiederverwendbaren Funktionen, weil wir sie einmal definieren und viele Male verwenden. Doch Objekte können viel mehr leisten als eine einfache Funktion.

Das Wesen von Objekten versteht man am besten, wenn man sie in Aktion sieht, anstatt sich mit einem abstrakten Konzept herumzuschlagen. Sehen Sie sich den folgenden Code an:

```
# class for a dog object
class Dog:

    # dogs can bark()
    def bark(self):
        print("woof!")
        pass

    pass
```

Los geht es mit etwas Bekanntem. Der Code enthält Anweisungen, die eine Funktion `bark()` (engl. für bellen) definieren. Diese Funktion liefert die Ausgabe »woof!«. Das ist leicht zu erkennen.

Sehen wir uns nun außerhalb dieser bekannten Funktionsdefinition um. In der zweiten Zeile stehen das Schlüsselwort `class` und der Name `Dog`. Die sich anschließende Codestruktur ähnelt der einer Funktion. Wie in einer Funktionsdefinition ist auch hier ein Name vorhanden. Im Unterschied zu einer Funktionsdefinition sehen Sie hier aber nicht das Schlüsselwort `def`, um eine Funktion zu definieren, sondern das Schlüsselwort `class`, das eine Klassendefinition einleitet.

Bevor wir uns damit beschäftigen, was eine *Klasse* – verglichen mit einem Objekt – ist, sehen wir uns wieder einen realen, wenn auch sehr einfachen Code an, der diese abstrakten Konzepte zum Leben erweckt:

```
sizzles = Dog()
sizzles.bark()
```

Der Code in der ersten Zeile erzeugt eine Variable `sizzles`. Die Anweisung sieht wie ein Funktionsaufruf aus. Tatsächlich ist `Dog()` eine spezielle Funktion, die eine Instanz der Klasse `Dog` erzeugt. Wir wissen nun, wie man Dinge aus Klassendefini-

tionen erzeugt. Und diese Dinge heißen *Objekte*. Wir haben ein Objekt `sizzles` aus der Klassendefinition von `Dog` erzeugt – dieses Objekt können wir als Hund (engl. `dog`) betrachten!

Die nächste Zeile ruft die Funktion `bark()` auf dem Objekt `sizzles` auf. Das ist nicht ganz neu, da wir Funktionen bereits kennengelernt haben. Noch nicht geläufig ist der Aufruf der Funktion `bark()`, als wäre sie ein Teil des `sizzles`-Objekts. Das hängt damit zusammen, dass alle von der Klasse `Dog` erzeugten Objekte über eine Funktion `bark()` verfügen. Das ist so in der Definition der Klasse `Dog` zu sehen.

Drücken wir es ganz einfach aus: Wir haben `sizzles` erzeugt, eine Art von `Dog`. Dieses `sizzles` ist ein Objekt, das in Gestalt einer `Dog`-Klasse erzeugt wurde. Objekte sind Instanzen einer Klasse.

Abbildung 2-17 zeigt unsere bisherigen Schritte und bestätigt auch, dass `sizzles.bark()` tatsächlich ein »woof!« ausgibt.

```
In [7]: # class for a dog object
class Dog:

    # dogs can bark()
    def bark(self):
        print("woof!")
        pass

    pass

In [8]: sizzles = Dog()

In [9]: sizzles.bark()

woof!
```

Abbildung 2-17: Von der Klasse zum Objekt

Vermutlich ist Ihnen das `self` in der Definition der Funktion `bark(self)` aufgefallen. Das erscheint Ihnen, so wie auch mir, vielleicht ungewöhnlich. So sehr ich Python mag, ich glaube nicht, dass es perfekt ist. Das `self` steht hier, damit Python die erzeugte Funktion dem richtigen Objekt zuweist. Eigentlich liegt das doch auf der Hand, weil `bark()` innerhalb der Klassendefinition steht und Python deshalb wissen müsste, mit welchen Objekten die Funktion zu verbinden ist. Doch das ist eben nur meine Meinung.

Objekte und Klassen lassen sich in vielen nützlichen Kontexten einsetzen. Sehen Sie sich den folgenden Code an:

```
sizzles = Dog()
mutley = Dog()

sizzles.bark()
mutley.bark()
```

Führen Sie den Code aus und sehen Sie sich das Ergebnis an (siehe Abbildung 2-18).

```
In [4]: sizzles = Dog()
        mutley = Dog()

        sizzles.bark()
        mutley.bark()

woof!
woof!
```

```
In [ ]:
```

Abbildung 2-18: Eine Funktion auf zwei verschiedenen Objekten aufrufen

Das ist interessant! Wir erzeugen zwei Objekte namens `sizzles` und `mutley`. Dabei werden beide Objekte von derselben `Dog()`-Klassendefinition erzeugt. Das ist ein leistungsfähiges Instrument! Wir definieren, wie die Objekte aussehen und wie sie sich verhalten sollten, und dann erstellen wir reale Instanzen von ihnen.

Hier liegt auch der Unterschied zwischen *Klasse* und *Objekt* – das eine ist eine Definition und das andere eine wirkliche Instanz dieser Definition. Eine Klasse ist ein Kuchenrezept in einem Buch, ein Objekt ist ein Kuchen, der nach diesem Rezept gebacken wurde. Abbildung 2-19 veranschaulicht, wie Objekte aus einem Klassenrezept entstehen.

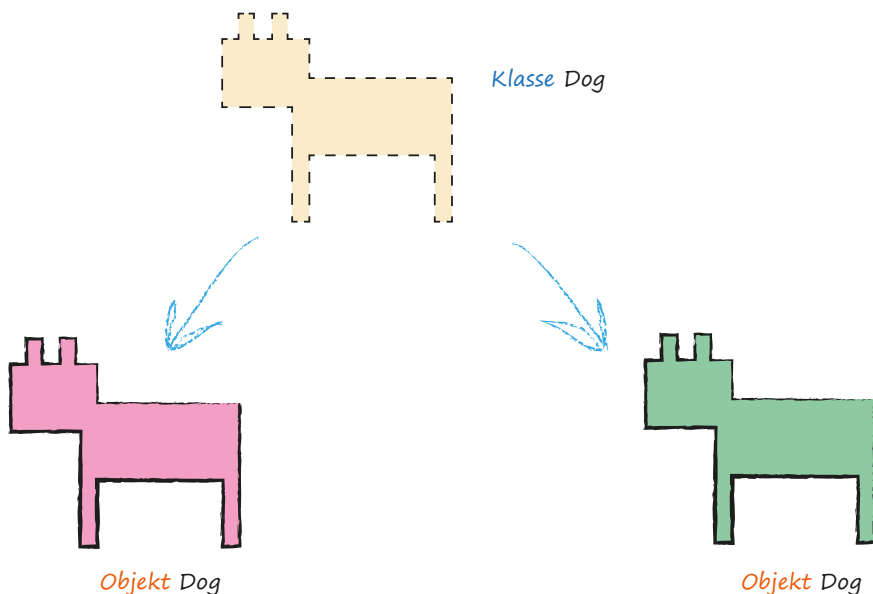


Abbildung 2-19: Objekte sind Instanzen einer Klasse.

Wozu dienen diese Objekte, die aus einer Klasse erzeugt werden? Wozu der ganze Aufwand? Wäre es nicht einfacher, ohne den ganzen zusätzlichen Code lediglich das Wort »woof!« auszugeben?

Zum einen ist es nützlich, viele gleichartige Objekte zu erzeugen, die alle von derselben Vorlage stammen. Man spart sich die Zeit, jedes einzelne Objekt vollständig erzeugen zu müssen. Doch der eigentliche Nutzen von Objekten besteht darin, dass Daten und Funktionen in den Objekten ordentlich eingehüllt sind. Hiervon profitiert der Programmierer. Komplizierte Probleme lassen sich leichter erfassen, wenn Codeabschnitte um Objekte herum organisiert sind, zu denen sie ihrem Wesen nach gehören. Hunde bellen. Schaltflächen klicken. Lautsprecher geben Töne wieder. Drucker drucken oder beschweren sich über Papiermangel. In vielen Computersystemen werden Schaltflächen, Lautsprecher und Drucker als Objekte dargestellt, deren Funktionen der Benutzer aufruft.

Diese Objektfunktionen bezeichnet man auch als *Methoden*. So haben wir weiter oben der Klasse Dog eine bark()-Funktion hinzugefügt, und die beiden Objekte sizzles und mutley, die von dieser Klasse erzeugt wurden, besitzen eine bark()-Methode. Im Beispiel haben Sie gesehen, dass beide bellen!

Neuronale Netze übernehmen eine Eingabe, führen bestimmte Berechnungen aus und liefern eine Ausgabe. Wir wissen auch, dass sie trainiert werden können. Und sicherlich erkennen Sie, dass diese Aktionen – trainieren und eine Antwort liefern – natürliche Funktionen eines neuronalen Netzes sind. Anders ausgedrückt, sind es Funktionen des Objekts »Neuronales Netz«. Des Weiteren enthalten neuronale Netze Daten, die natürlicherweise dorthin gehören – die Verknüpfungsgewichte. Deshalb werden wir unser neuronales Netz als Objekt erstellen.

Der Vollständigkeit halber sehen wir uns an, wie wir einer Klasse Datenvariablen hinzufügen sowie einige Methoden, um diese Daten anzuzeigen und zu ändern. Sehen Sie sich die folgende Definition einer neuen Klasse Dog an. Die noch unbekannten Elemente werden wir nacheinander untersuchen.

```
# class for a dog object
class Dog:

    # initialisation method with internal data
    def __init__(self, petname, temp):
        self.name = petname;
        self.temperature = temp;

    # get status
    def status(self):
        print("dog name is ", self.name)
        print("dog temperature is ", self.temperature)
        pass

    # set temperature
    def setTemperature(self,temp):
```

```

        self.temperature = temp;
        pass

# dogs can bark()
def bark(self):
    print("woof!")
    pass

pass

```

Zunächst einmal haben wir der Klasse Dog drei neue Funktionen hinzugefügt. Zur bereits vorhandenen Funktion `bark()` sind die Funktionen `__init__()`, `status()` und `setTemperature()` hinzugekommen. Wie man Funktionen hinzufügt, dürfte ohne Weiteres verständlich sein. Denkbar wäre auch gewesen, passend zu `bark()` eine Funktion `sneeze()` (engl. für niesen) vorzusehen, wenn wir gewollt hätten.

Doch diese neuen Funktionen enthalten offenbar Variablennamen innerhalb der Funktionsnamen. Die Funktion `setTemperature` heißt eigentlich `setTemperature(self, temp)`. Die eigenwillig benannte Funktion `__init__` ist eigentlich `__init__(self, petname, temp)`. Was bedeuten diese zusätzlichen Elemente innerhalb der Klammern? Das sind die Variablen, die die Funktion bei ihrem Aufruf erwartet – sogenannte *Parameter*. Haben Sie noch die Mittelwertfunktion `avg(x,y)`, die weiter oben gezeigt wurde, auf dem Schirm? Die Definition von `avg()` hat klargemacht, dass die Funktion zwei Zahlen erwartet. Die Funktion `__init__()` benötigt also einen Parameter **petname** und einen Parameter **temp**, die Funktion `setTemperature()` nur einen Parameter **temp**.

Sehen wir uns nun die neuen Funktionen genauer an, zuerst die Funktion mit dem ungewöhnlichen Namen `__init__()`. Weshalb hat sie so einen eigenwilligen Namen bekommen? Dieser Name hat für Python eine spezielle Bedeutung. Wird ein Objekt erstellt, ruft Python eine Funktion `__init__()` auf. Das ist wirklich praktisch, um ein Objekt vorzubereiten, bevor wir es tatsächlich verwenden. Was stellen wir also in dieser magischen Initialisierungsfunktion an? Anscheinend erzeugen wir nur zwei neue Variablen namens `self.name` und `self.temperature`. Ihre Werte erhalten sie von den Variablen **petname** und **temp**, die an die Funktion übergeben werden. Das Schlüsselwort `self` bedeutet, dass die Variablen Teil des Objekts selbst sind. Das heißt, sie gehören nur zu diesem Objekt und sind unabhängig von einem anderen Dog-Objekt oder allgemeinen Variablen in Python. Wir wollen den Namen dieses Hundes nicht mit dem eines anderen Hundes verwechseln! Falls Ihnen das kompliziert erscheint, keine Bange. Anhand eines realen Beispiels werden Sie es sicher leicht verstehen.

Die `status()`-Funktion ist ganz einfach gehalten. Sie übernimmt keine Parameter und gibt lediglich die Variablen `name` und `temperature` des Dog-Objekts aus.

Die letzte Funktion ist `setTemperature()`. Als Parameter übernimmt sie die Temperatur `temp` und setzt die Variable `self.temperature` auf diesen Wert. Das heißt, Sie

können die Temperatur des Objekts jederzeit ändern, nachdem Sie das Objekt erstellt haben, und zwar sooft Sie wollen.

Bisher ist offengeblieben, warum diese Funktionen – `bark()` eingeschlossen – ein `self` als ersten Parameter enthalten. Es handelt sich um eine Eigenheit von Python, die ich unschön finde, aber Python hat sich eben in der Form entwickelt. Der Parameter `self` soll Python klarmachen, dass die von Ihnen definierte Funktion zu dem Objekt gehört, auf das mit `self` verwiesen wird. Man könnte meinen, dass das ohnehin klar ist, weil wir ja die Funktion innerhalb der Klasse schreiben. Und es dürfte nicht überraschen, dass dies zu Debatten selbst unter erfahrenen Python-Programmierern geführt hat. Wenn Sie sich über diesen Parameter wundern, befinden Sie sich also in guter Gesellschaft.

An einem praktischen Beispiel lassen sich die beschriebenen Konzepte am besten veranschaulichen. Der Code in Abbildung 2-20 zeigt die neue `Dog`-Klasse, die mit diesen neuen Funktionen definiert wird, und ein neues `Dog`-Objekt namens `lassie`, das mit Parametern erzeugt wird, die seinen Namen als »Lassie« und seine Anfangstemperatur mit 37 festlegen.

```
In [18]: # class for a dog object
class Dog:

    # initialisation method with internal data
    def __init__(self, petname, temp):
        self.name = petname;
        self.temperature = temp;

    # get status
    def status(self):
        print("dog name is ", self.name)
        print("dog temperature is ", self.temperature)
        pass

    # set temperature
    def setTemperature(self,temp):
        self.temperature = temp;
        pass

    # dogs can bark()
    def bark(self):
        print("woof!")
        pass

    pass

In [19]: # create a new dog object from the Dog class
lassie = Dog("Lassie", 37)

In [20]: lassie.status()

dog name is  Lassie
dog temperature is  37
```

Abbildung 2-20: Ein neues `Dog`-Objekt initialisieren

Die Funktion `status()` gibt beim Aufruf auf diesem `Dog`-Objekt `lassie` den Namen des Hundes und dessen aktuelle Körpertemperatur aus. Diese Temperatur hat sich nicht geändert, seitdem `lassie` erstellt worden ist.

Wir legen nun eine andere Temperatur fest und kontrollieren mit einer `status()`-Abfrage, ob sie sich wirklich innerhalb des Objekts geändert hat.

```
lassie.setTemperature(40)
lassie.status()
```

Abbildung 2-21 zeigt die Ergebnisse.

```
In [19]: # create a new dog object from the Dog class
lassie = Dog("Lassie", 37)

In [20]: lassie.status()
dog name is Lassie
dog temperature is 37

In [22]: lassie.setTemperature(40)

In [23]: lassie.status()
dog name is Lassie
dog temperature is 40
```

Abbildung 2-21: Die Temperatur des Objekts *lassie* ändern

Wie die Ausgabe zeigt, hat der Aufruf von `setTemperature(40)` auf dem *lassie*-Objekt tatsächlich die im Objekt verzeichnete Temperatur geändert.

Sie können sich wirklich freuen, weil Sie ziemlich viel über Objekte gelernt haben – ein vermeintlich anspruchsvolles Thema, das aber doch gar nicht so schwer war, oder!

Vorerst wissen Sie genügend über Python, um mit der Programmierung eines neuronalen Netzes beginnen zu können.

Neuronales Netz mit Python

Wir beginnen nun damit, ein neuronales Netz zu programmieren, und zwar auf Basis des Wissens, das Sie sich in diesem Kapitel über Python angeeignet haben. Dabei gehen wir schrittweise vor und erstellen Stück für Stück ein Python-Programm.

Klein zu beginnen und dann zu wachsen, ist ein sinnvoller Ansatz, um einigermaßen komplexen Computercode zu erstellen.

Nach den bisherigen Vorarbeiten ergibt es sich gewissermaßen von selbst, zunächst das Gerüst eines neuronalen Netzes zu entwerfen. Los geht's!

Der Gerüstcode

Dieser Abschnitt skizziert, wie eine Klasse »Neuronales Netz« (`neuralNetwork`) aussehen sollte. Sie wissen bereits, dass sie mindestens drei Funktionen erfüllen sollte: