# 8

# Dateien lesen und schreiben

Variablen sind eine praktische Einrichtung, um Daten festzuhalten, während das Programm ausgeführt wird, aber wenn die Daten erhalten bleiben sollen, nachdem das Programm beendet ist, müssen Sie sie in einer Datei speichern. Sie können sich den Inhalt einer Datei als einen einzigen Stringwert vorstellen, dessen Größe durchaus in Gigabyte misst. In diesem Kapitel erfahren Sie, wie Sie in Python Dateien auf der Festplatte erstellen, lesen und speichern.

## Dateien und Dateipfade

Die beiden wichtigsten Merkmale einer Datei sind der *Dateiname* (der gewöhnlich als einzelnes Wort geschrieben wird) und der *Pfad*, der den Speicherort der Datei auf dem Computer angibt. Auf meinem Windows-7-Laptop habe ich beispielsweise eine Datei mit dem Namen *projects.docx* im Pfad *C:\Users\asweigart\Documents*. Der Teil des Dateinamens hinter dem Punkt wird als die *Dateinamenerweiterung* oder *Endung* bezeichnet und gibt den Typ der Datei an. Bei *project.docx* handelt es sich um ein Word-Dokument und *Users*, *asweigart* und

*Documents* sind *Ordner* (oder Verzeichnisse). Ordner können Dateien sowie andere Ordner enthalten. Beispielsweise befindet sich *projects.docx* im Ordner *Documents*, der wiederum im Ordner *asweigart* innerhalb des Ordners *Users* steckt. Diese Ordnerstruktur sehen Sie in Abb. 8–1.

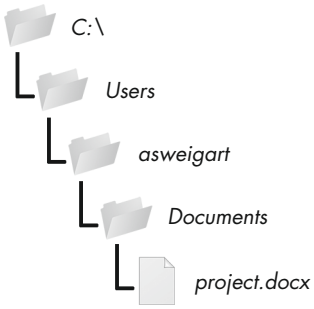


***Abb. 8–1*** *Eine Datei in einer Ordnerhierarchie*

Die Angabe *C:* im Pfad ist der *Wurzelordner* oder (unter Windows) *Stammordner*, der alle anderen Ordner enthält. Unter Windows heißt dieser Stammordner *C:* und wird auch als »Laufwerk *C:*« bezeichnet. Der Wurzelordner unter OS X und Linux ist /. In diesem Buch verwende ich den Windows-Stammordner *C:*. Wenn Sie die Beispiele in einer interaktiven Shell unter OS X oder Linux eingeben, müssen Sie diese Angabe durch / ersetzen.

Zusätzliche Datenträger oder *Volumes*, etwa ein DVD-Laufwerk oder ein USB-Stick, werden in den verschiedenen Betriebssystemen unterschiedlich dargestellt. Unter Windows erscheinen sie als zusätzliche, mit Buchstaben gekennzeichnete Stammlaufwerke wie *D:* oder *E:*, unter OS X als neue Ordner im Ordner */Volumes* und unter Linux als neue Ordner unter */mnt* (»mount«, was das Bereitstellen oder Einhängen eines solchen Laufwerks bezeichnet). Beachten Sie auch, dass bei Datei- und Ordnernamen unter Linux zwischen Groß- und Kleinschreibung unterschieden wird, unter Windows und OS X jedoch nicht.

### Backslash unter Windows und Schrägstrich unter OS X und Linux

Unter Windows werden Pfade mit Backslashes, also umgekehrtem Schrägstrich (\), als Trennzeichen zwischen den Ordnernamen geschrieben, unter OS X und Linux jedoch mit einem normalen Schrägstrich (/). Wenn Ihre Programme auf allen Betriebssystemen laufen sollen, müssen Sie Ihre Python-Skripts für beide Fälle auslegen.

Zum Glück lässt sich das mit der Funktion `os.path.join()` leicht erledigen. Wenn Sie ihr die Stringwerte der einzelnen Datei- und Ordnernamen eines Pfads übergeben,

gibt sie den Dateipfad als String mit den richtigen Trennzeichen für das vorliegende Betriebssystem zurück. Probieren Sie Folgendes in der interaktiven Shell aus:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

In der interaktiven Shell auf Windows gibt `os.path.join('usr', 'bin', spam')` wie oben angegeben `'usr\\bin\\spam'` zurück (mit doppelten Backslashes, da jeder Backslash mit einem zweiten maskiert werden muss). Auf OS X und Linux dagegen ergibt sich der String `'usr/bin/spam'`.

Die Funktion `os.path.join()` ist sehr nützlich, wenn Sie Strings für Dateinamen erstellen müssen – was häufig vorkommt, da Sie den in diesem Kapitel vorgestellten Funktionen zur Dateibearbeitung solche Strings übergeben müssen. Der folgende Beispielcode hängt die einzelnen Einträge aus einer Liste von Dateinamen an einen Ordnerpfad an:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
   print(os.path.join('C:\\Users\\asweigart', filename))

C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

### Das aktuelle Arbeitsverzeichnis

Jedes Programm, das auf einem Computer läuft, hat ein *aktuelles Arbeitsverzeichnis*. Bei allen Dateinamen und Pfaden, die nicht mit dem Stammordner beginnen, wird angenommen, dass sie sich in diesem Arbeitsverzeichnis befinden. Um den Stringwert des aktuellen Arbeitsverzeichnisses abzurufen, verwenden Sie die Funktion `os.getcwd()` (wobei `cwd` für *current working directory* steht), und um es zu ändern die Funktion `os.chdir()`:

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Hier ist das aktuelle Arbeitsverzeichnis zunächst *C:\Python34*, weshalb der Dateiname *project.docx* auf *C:\Python34\project.docx* verweist. Wenn wir das aktuelle

Arbeitsverzeichnis anschließend in *C:\Windows* ändern, wird *project.docx* als *C:\Windows\project.docx* interpretiert.

Wenn Sie versuchen, zu einem Verzeichnis zu wechseln, das gar nicht existiert, gibt Python eine Fehlermeldung aus:

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
   File "<pyshell#18>", line 1, in <module>
     os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

**Hinweis**

*Ordner* ist zwar die moderne Bezeichnung für ein Verzeichnis, doch wird das aktuelle Arbeitsverzeichnis (oder einfach das Arbeitsverzeichnis) normalerweise genauso bezeichnet und nicht als aktueller Arbeitsordner.

### Absolute und relative Pfade

Es gibt zwei Möglichkeiten, um einen Dateipfad anzugeben:

- Als *absoluter Pfad*, der stets mit dem Stammordner beginnt.
- Als *relativer Pfad*, der relativ zum aktuellen Arbeitsverzeichnis des Programms angegeben ist.

In Pfaden finden Sie oft auch die Angaben `.` und `...` Dabei handelt es sich nicht um echte Ordner, sondern um besondere Bezeichnungen. Der einzelne Punkt `.` steht dabei für das vorliegende Verzeichnis, zwei Punkte `..` für den Elternordner.

Abb. 8–2 zeigt Beispiele für Ordner und Dateien. Wenn *C:\bacon* das aktuelle Arbeitsverzeichnis ist, dann lauten die relativen Pfade für die dargestellten Ordner und Dateien wie in dem Bild angegeben.

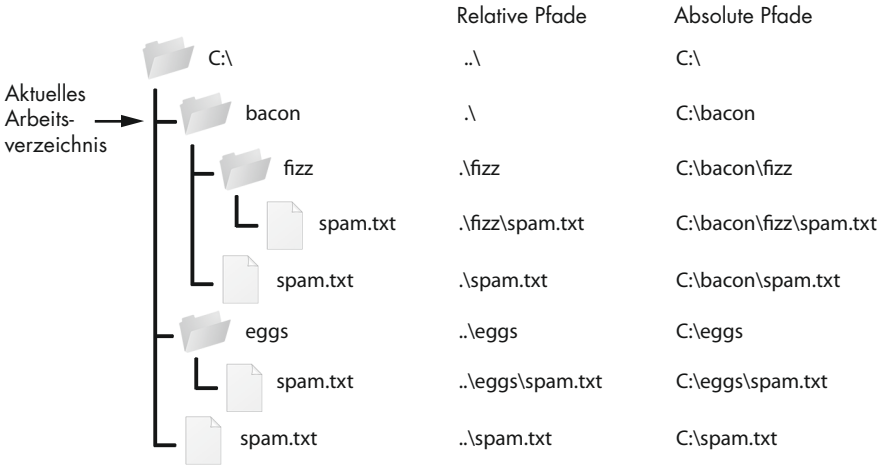


***Abb. 8–2*** *Relative und absolute Pfade für die Ordner und Dateien im Arbeitsverzeichnis C:\bacon*

Die Angabe `.\` zu Beginn eines relativen Pfads ist optional. Beispielsweise verweisen sowohl *.\spam.txt* als auch *spam.txt* auf dieselbe Datei.

### Neue Ordner mit os.makedirs() erstellen

Mithilfe der Funktion `os.makedirs()` können Ihre Programme neue Ordner erstellen (wobei sich *dirs* in dem Funktionsnamen auf die alternative Bezeichnung *directories* für Verzeichnisse bezieht):

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

Dadurch wird nicht nur der Ordner *C:\delicious* erstellt, sondern auch der Ordner *walnut* innerhalb von *C:\delicious* und der Ordner *waffles* innerhalb von *C:\delicious\walnut*. Die Funktion `os.makedirs()` erstellt also alle erforderlichen Zwischenordner, um dafür zu sorgen, dass der vollständige Pfad existiert. Diese Ordnerhierarchie sehen Sie in Abb. 8–3.

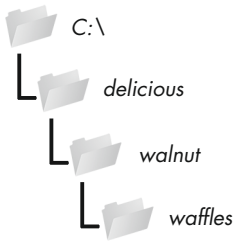


***Abb. 8–3*** *Das Ergebnis von* `os.makedirs('C:\\delicious\\walnut\\waffles')`

## Das Modul os.path

Das Modul `os.path` enthält viele nützliche Funktionen für den Umgang mit Datei- und Pfadnamen. Bereits kennengelernt haben Sie die Funktion `os.path.join()`, mit der Sie Pfade passend zum vorliegenden Betriebssystem zusammenstellen können. Da es sich bei `os.path` um ein Modul innerhalb des Moduls `os` handelt, können Sie es einfach durch `import os` importieren. Wenn Sie in Ihrem Programm mit Dateien, Ordnern oder Pfaden arbeiten müssen, schlagen Sie die Beispiele in diesem Abschnitt nach. Die komplette Dokumentation des Moduls `os.path` finden Sie auf der Python-Website unter *http://docs.python.org/3/library/os.path.html*.

> **Hinweis**
>
> Für die meisten Beispiele in diesem Abschnitt ist das Modul `os` erforderlich, weshalb Sie es zu Beginn jedes Skripts und bei jedem Neustart von IDLE importieren müssen. Anderenfalls erhalten Sie die Fehlermeldung `NameError: name 'os' is not defined`.

### Absolute und relative Pfade verwenden

Das Modul `os.path` enthält Funktionen, die den absoluten Pfad zu einem relativen Pfad zurückgeben oder prüfen, ob ein gegebener Pfad ein absoluter Pfad ist.

- Die Funktion `os.pathabspath(`*path*`)` gibt den absoluten Pfad des Arguments als String zurück. Das bietet eine einfache Möglichkeit, um einen relativen in einen absoluten Pfad umzuwandeln.
- Die Funktion `os.path.isabs(`*path*`)` gibt `True` zurück, wenn das Argument ein absoluter Pfad ist, und `False`, wenn es sich um einen relativen Pfad handelt.
- Die Funktion `os.path.relpath(`*path*`, `*start*`)` gibt den String des relativen Pfads vom Ausgangspunkt (*start*) bis zu *path* zurück. Wenn Sie *start* nicht angeben, wird das aktuelle Arbeitsverzeichnis als Ausgangspunkt genommen.

Probieren Sie diese Funktionen in der interaktiven Shell aus:

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Da *C:\Python34* beim Aufruf von `os.path.abspath()` das Arbeitsverzeichnis war, steht der Punkt für den absoluten Pfad *C:\Python34*.

### Hinweis

Die Ordner- und Dateistruktur auf Ihrem System unterscheidet sich natürlich von der auf meinem Computer. Daher werden Sie manche Beispiele in diesem Kapitel nicht exakt nachvollziehen können. Versuchen Sie aber, die Beispiele anhand der Ordner auf Ihrem Computer zu verfolgen.

Probieren Sie auch folgende Aufrufe von `os.path.relpath()` in der interaktiven Shell aus:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
```

Die Funktion `os.path.dirname(`*path*`)` gibt einen String mit allem zurück, was vor dem letzten (umgekehrten) Schrägstrich im Argument `path` steht, die Funktion `os.path.basename(`*path*`)` dagegen einen String mit allem, was hinter diesem letzten Schrägstrich steht. Verzeichnisname (*dir name*) und Grundname (*base name*) eines Pfads sind in Abb. 8–4 dargestellt.

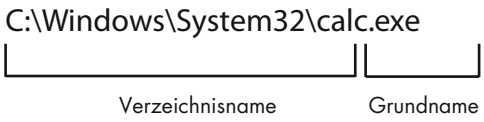


***Abb. 8–4*** *Der Grundname folgt auf den letzten Schrägstrich im Pfad und ist mit dem Dateinamen identisch. Alles, was links von dem letzten Schrägstrich steht, gehört dagegen zum Verzeichnisnamen.*

Geben Sie beispielsweise Folgendes in die interaktive Shell ein:

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

Wenn Sie sowohl den Verzeichnis- als auch den Grundnamen eines Pfads benötigen, können Sie `os.path.split()` aufrufen, um einen Tupelwert mit diesen beiden Strings zu erhalten:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Das gleiche Tupel können Sie auch erstellen, indem Sie `os.path.dirname()` und `os.path.basename()` aufrufen und die Rückgabewerte in ein Tupel aufnehmen:

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

Allerdings ist `os.path.split()` eine praktische Abkürzung.

Beachten Sie aber, dass `os.path.split()` keinen Dateipfad entgegennimmt und daraus eine Liste der Strings für die einzelnen Ordner zurückgibt. Für diesen Zweck müssen Sie den Pfad mit der Stringmethode `split()` zerlegen und ihr die Variable `os.path.sep` übergeben, in der das Ordnertrennzeichen für das vorliegende Betriebssystem gespeichert ist.

Betrachten Sie dazu das folgende Beispiel in der interaktiven Shell:

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

Unter OS X und Linux steht am Anfang der zurückgegebenen Liste ein leerer String:

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

Die Methode `split()` gibt eine Liste der einzelnen Teile des Pfads zurück. Damit sie auf allen Betriebssystemen korrekt funktioniert, müssen Sie ihr `os.path.sep` übergeben.

### Dateigrößen und Ordnerinhalte ermitteln

Nachdem Sie nun wissen, wie Sie mit Dateipfaden umgehen müssen, können Sie Informationen über einzelne Dateien und Ordner abrufen. Das Modul `os` enthält Funktionen, mit denen Sie die Größe eines Ordners in Byte ermitteln und feststellen können, welche Dateien und Ordner sich innerhalb eines gegebenen Ordners befinden.

- Die Funktion `os.path.getsize(`*path*`)` gibt die Größe der im Argument `path` übergebenen Datei in Byte zurück.
- Die Funktion `os.listdir(`*path*`)` gibt eine Liste der Namenstrings aller Dateien und Ordner zurück, die sich in dem als `path` übergebenen Pfad befinden. (Beachten Sie, dass sich diese Funktion nicht im Modul `os.path`, sondern in `os` befindet.)

Wenn ich diese Funktionen in der interaktiven Shell ausprobiere, erhalte ich folgende Ergebnisse:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--schnipp--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Wie Sie sehen, ist das Programm *calc.exe* auf meinem Computer 776.192 Bytes groß und im Ordner *C:\Windows\system32* befinden sich eine ganze Menge Dateien. Wenn ich die Gesamtgröße aller Dateien in diesem Verzeichnis herausfinden möchte, kann ich `os.path.getsize()` und `os.listdir()` zusammen einsetzen:

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
    totalSize = totalSize +
    os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(totalSize)
1117846456
```

Dieser Code durchläuft alle Dateien im Ordner *C:\Windows\System32* und erhöht dabei die Variable `totalSize` um die Größe der jeweiligen Datei. Beachten Sie, dass ich hier beim Aufruf von `os.path.getsize()` die Funktion `os.path.join()` verwende, um den Ordnernamen mit dem Namen der aktuellen Datei zu verknüpfen. Der von `os.path.getsize()` zurückgegebene Integer wird dann zu dem Wert in `totalSize` addiert. Nachdem alle Dateien durchlaufen wurden, gibt das Programm `totalSize` aus, um die Gesamtgröße aller Dateien im Ordner *C:\Windows\System32* anzuzeigen.

### Die Gültigkeit von Pfaden prüfen

Viele Python-Funktionen werden mit einer Fehlermeldung beendet, wenn Sie ihnen einen Pfad übergeben, den es gar nicht gibt. Um das zu verhindern, finden Sie im Modul `os.path` Funktionen, mit denen Sie prüfen können, ob ein gegebener Pfad existiert und ob es sich dabei um eine Datei oder einen Ordner handelt.

- Die Funktion `os.path.exists(`*path*`)` gibt `True` zurück, wenn die Datei oder der Ordner im Argument vorhanden ist, und anderenfalls `False`.
- Die Funktion `os.path.isfile(`*path*`)` gibt `True` zurück, wenn das Pfadargument vorhanden und eine Datei ist, und anderenfalls `False`.
- Die Funktion `os.path.isdir (`*path*`)` gibt `True` zurück, wenn das Pfadargument vorhanden und ein Ordner ist, und anderenfalls `False`.

Beim Ausprobieren in der interaktiven Shell habe ich folgende Ergebnisse erhalten:

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

Mit der Funktion `os.path.exists()` können Sie auch prüfen, ob an den Computer ein DVD- oder Flash-Laufwerk angeschlossen ist. Beispielsweise kann ich auf meinem Windows-Rechner wie folgt herausfinden, ob ein Flash-Laufwerk mit dem Volume-Namen *D:* vorhanden ist:

```
>>> os.path.exists('D:\\')
False
```

Hoppla! Da habe ich wohl vergessen, mein Flash-Laufwerk anzuschließen.

## Dateien lesen und schreiben

Wenn Sie im Umgang mit Ordnern und relativen Pfaden sicher sind, können Sie die Speicherorte der Daten angeben, in denen Sie lesen und schreiben wollen. Die in diesem Abschnitt beschriebenen Funktionen dienen zur Arbeit mit *reinen Textdateien*. Sie enthalten nur einfache Textzeichen ohne Informationen über Schriftart, Schriftgröße oder Farbe. Beispiele dafür sind Textdateien mit der Endung *.txt* und Python-Skriptdateien mit der Erweiterung *.py*. Geöffnet werden können sie mit Programmen wie dem Windows-Editor oder der OS-X-Anwendung TextEdit. In Python können Sie Programme schreiben, die die Inhalte solcher reinen Textdateien lesen und als ganz normale Stringwerte behandeln können.

*Binärdateien* dagegen sind alle anderen Dateitypen, z. B. die Dokumente von Textverarbeitungsprogrammen, PDF-Dokumente, Bilder, Arbeitsblätter und ausführbare Programme. Wenn Sie eine Binärdatei im Editor oder in TextEditor öffnen, sehen Sie nur einen Haufen wirrer, sinnloser Zeichen wie in Abb. 8–5.

Da die verschiedenen Arten von Binärdateien jeweils auf ihre eigene Art und Weise behandelt werden müssen, gehe ich in diesem Buch nicht darauf ein, wie rohe Binärdateien gelesen und geschrieben werden. Zum Glück gibt es eine Reihe von


***Abb. 8–5*** *Das Windows-Programm calc.exe im Editor*

Modulen, die die Arbeit mit Binärdateien erleichtern. Eines davon, das Modul `shelve`, lernen Sie in diesem Kapitel noch kennen.

Um Dateien in Python zu lesen oder zu schreiben, sind drei Schritte erforderlich:

1. Rufen Sie die Funktion `open()` auf, um ein `File`-Objekt zurückzugeben.
2. Rufen Sie die Funktion `read()` oder `write()` für das `File`-Objekt auf.
3. Schließen Sie die Datei, indem Sie die Methode `close()` für das `File`-Objekt aufrufen.

## Dateien mit der Funktion open() öffnen

Um mit der Funktion `open()` eine Datei zu öffnen, müssen Sie ihr den String mit dem Pfad der gewünschten Datei übergeben. Dabei kann es sich sowohl um einen absoluten als auch um einen relativen Pfad handeln. Die Funktion `open()` gibt ein `File`-Objekt zurück.

Probieren Sie das aus, indem Sie mit dem Editor oder mit TextEdit eine Textdatei namens *hello.txt* erstellen. Geben Sie **Hello world!** als Inhalt dieser Datei ein und speichern Sie sie in Ihrem Benutzerordner. Unter Windows geben Sie dann Folgendes in die interaktive Shell ein:

```
>>> helloFile = open('C:\\Users\\Benutzerordner\\hello.txt')
```

Unter OS X verwenden Sie stattdessen folgenden Befehl:

```
>>> helloFile = open('/Users/Benutzerordner/hello.txt')
```

Ersetzen Sie dabei *Benutzerordner* durch Ihren Benutzernamen auf dem Computer. Beispielsweise verwende ich auf meinem Windows-Rechner, wo mein Benutzername *asweigart* lautet, die Eingabe `'C:\\Users\\asweigart\\hello.txt'`.

Beide Befehle öffnen die Datei im Reintext-Lesemodus oder kurz *Lesemodus*. In diesem Modus erlaubt Python Ihnen nur, Daten in der Datei zu lesen; Daten schreiben oder auf irgendeine Weise ändern können Sie dagegen nicht. Dies ist der Standardmodus für Dateien, die Sie in Python öffnen. Wenn Sie sich nicht auf diese Standardeinstellung verlassen wollen, können Sie den Modus auch ausdrücklich angeben, indem Sie als zweites Argument von `open()` den Stringwert `'r'` übergeben. Die Funktionsaufrufe `open('/Users/asweigart/hello.txt', 'r')` und `open('/Users/asweigart/hello.txt')` bewirken daher das Gleiche.

Der Aufruf von `open()` gibt ein `File`-Objekt zurück. Ein solches Objekt steht für eine Datei auf Ihrem Computer. Es handelt sich dabei einfach nur um eine weitere Art von Werten in Python, ebenso wie Listen oder Dictionarys. Im vorstehenden Beispiel wird das `File`-Objekt in der Variablen `helloFile` gespeichert. Wenn Sie nun in der Datei lesen oder schreiben wollen, können Sie das tun, indem Sie entsprechende Methoden für das `File`-Objekt in `helloFile` aufrufen.

### Die Inhalte einer Datei lesen

Da Sie nun über ein `File`-Objekt verfügen, können Sie darin lesen. Wenn Sie den gesamten Inhalt der Datei als Stringwert lesen möchten, verwenden Sie die Methode `read()` des `File`-Objekts. Bleiben wir bei unserem Beispiel, bei dem das `File`-Objekt für die Datei *hello.txt* in `helloFile` gespeichert ist. Geben Sie nun Folgendes in die interaktive Shell ein:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

Stellen Sie sich den Inhalt einer Datei als einen einzigen, umfangreichen Stringwert vor. Die Methode `read()` gibt diesen String zurück.

Alternativ können Sie mit der Methode `readlines()` auch eine Liste von Stringwerten aus der Datei abrufen, wobei jeder String für eine Textzeile steht. Um das auszuprobieren, legen Sie die Datei *sonnet29.txt* in demselben Verzeichnis an wie *hello.txt* und schreiben Folgendes hinein:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Achten Sie darauf, die vier Zeilen durch Zeilenumbrüche zu trennen. Geben Sie in der interaktiven Shell dann Folgendes ein:

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

Alle einzelnen Stringwerte enden mit dem Zeilenumbruchzeichen `\n`, ausgenommen die letzte Zeile der Datei. Häufig ist es einfacher, mit einer Liste von Strings zu arbeiten als mit einem einzigen, riesigen String.

### Dateien schreiben

In Python können Sie auch Inhalte in eine Datei schreiben, ähnlich wie Sie mit `print()` Strings auf den Bildschirm »schreiben«. Allerdings ist es nicht möglich, in eine Datei zu schreiben, die im Lesemodus geöffnet ist. Stattdessen müssen Sie sie im *Schreibmodus* oder im *Anhängemodus* öffnen.

Im Schreibmodus wird die vorhandene Datei überschrieben und ganz neu mit Text gefüllt, ähnlich wie Sie einen Variablenwert mit einem neuen Wert überschreiben. Um eine Datei in diesem Modus zu öffnen, übergeben Sie `'w'` als zweites Argument der Funktion `open()`. Im Anhängemodus dagegen wird neuer Text am Ende der Datei angehängt, so wie Sie neue Elemente hinten zu einer Liste hinzufügen, anstatt sie komplett zu überschreiben. Hierzu verwenden Sie `'a'` als zweites Argument von `open()`.

Wenn es keine Datei mit dem an `open()` übergebenen Namen gibt, wird sowohl im Schreib- als auch im Anhängemodus eine neue, leere Datei erstellt. Nach dem Lesen oder Schreiben in einer Datei müssen Sie die Methode `close()` aufrufen, bevor Sie die Datei wieder öffnen können.

Sehen wir uns all diese verschiedenen Möglichkeiten nun im Zusammenhang an. Dazu probieren Sie Folgendes in der interaktiven Shell aus:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

Als Erstes öffnen wir hier *bacon.txt* im Schreibmodus. Da es eine Datei dieses Namens noch nicht gibt, wird sie von Python erstellt. Danach rufen wir die Funktion `write()` für die geöffnete Datei auf und übergeben ihr das Stringargument `'Hello world! /n'`, wodurch dieser String in die Datei geschrieben wird. Außerdem wird die Anzahl der geschriebenen Zeichen einschließlich des Zeilenumbruchs ausgegeben. Daraufhin schließen wir die Datei.

Um Text zu den bestehenden Inhalten der Datei hinzuzufügen, ohne den eben geschriebenen String zu ersetzen, öffnen wir die Datei anschließend im Anhängemodus, schreiben `'Bacon is not a vegetable.'` in die Datei und schließen sie. Als Letztes geben wir den Inhalt der Datei auf dem Bildschirm aus. Dazu öffnen wir sie im Standardmodus, dem Lesemodus, rufen `read()` auf, speichern das resultierende `File`-Objekt in `content`, schließen die Datei und geben `content` aus.

Beachten Sie, dass die Methode `write()` im Gegensatz zu `print()` am Ende eines Strings nicht automatisch einen Zeilenumbruch einfügt. Das müssen Sie manuell tun.

## Variablen mit dem Modul shelve speichern

Mit dem Modul `shelve` können Sie in Ihren Python-Programmen Variablen in binären »Shelf-Dateien« speichern. Dadurch kann das Programm die Daten der Variablen von der Festplatte wiederherstellen. Dieses Modul ermöglicht es Ihnen, Speichern- und Öffnen-Funktionen zu Ihren Programmen hinzuzufügen. Wenn ein Benutzer in einem Programm beispielsweise Konfigurationseinstellungen vorgenommen hat, kann er sie in einer Shelf-Datei speichern, sodass das Programm sie bei der nächsten Ausführung lädt.

Geben Sie Folgendes in die interaktive Shell ein:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

Um Daten mithilfe des Moduls `shelve` zu lesen und zu schreiben, müssen Sie es zunächst importieren. Rufen Sie dann die Funktion `shelve.open()` auf, übergeben Sie ihr einen Dateinamen und speichern Sie den zurückgegebenen Shelf-Wert in einer Variablen. Nun können Sie an diesem Shelf-Wert Änderungen vornehmen wie an einem Dictionary. Abschließend rufen Sie `close()` für den Shelf-Wert auf. In unserem Beispiel wird der Shelf-Wert in `shelfFile` gespeichert. Wir erstellen die Liste `cats` und speichern diese Liste mit `shelfFile['cats']` als Wert zu dem Schlüs-

sel `'cats'` in `shelfFile` (wie in einem Dictionary). Danach rufen wir `close()` für `shelfFile` auf.

Wenn Sie diesen Code unter Windows ausführen, stehen anschließend drei neue Dateien in Ihrem Arbeitsverzeichnis: *mydata.bak*, *mydata.dat* und *mydata.dir*. Unter OS X dagegen wird nur eine einzige Datei namens *mydata.db* erstellt.

Diese Binärdateien enthalten die Daten, die Sie in dem Shelf gespeichert haben. Das genaue Format dieser Dateien brauchen Sie nicht zu kennen. Sie müssen nur wissen, was das Modul `shelve` tut, aber nicht, wie es das im Einzelnen macht. Dieses Modul nimmt Ihnen die Verantwortung dafür ab, die Programmdaten in einer Datei zu speichern.

Mit dem Modul `shelve` können Sie die Shelf-Dateien in Ihrem Programm auch wieder öffnen und die Daten daraus lesen. Dabei ist es nicht erforderlich, ausdrücklich den Lese- oder Schreibmodus zu verlangen, denn nach dem Öffnen ist beides möglich. Probieren Sie das wie folgt in der interaktiven Shell aus:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Hier öffnen wir die Shelf-Dateien, um zu prüfen, ob sie nach wie vor die richtigen Daten enthalten. `shelfFile['cats']` gibt genau die Liste zurück, die wir zuvor gespeichert haben. Da wir nun wissen, dass die richtigen Daten gespeichert sind, rufen wir `close()` auf.

Wie für Dictionarys gibt es auch für Shelf-Werte die Methoden `keys()` und `values()`, die listenähnliche Werte der Schlüssel bzw. Werte in dem Shelf zurückgeben. Um daraus echte Listen zu machen, müssen Sie sie an die Funktion `list()` übergeben, wie das folgende Beispiel zeigt:

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

Reiner Text ist sinnvoll für Dateien, die in einem Texteditor wie dem Windows-Editor oder TextEdit gelesen werden sollen. Wenn Sie dagegen Daten aus Ihrem Python-Programm speichern wollen, sollten Sie das Modul `shelve` verwenden.

## Variablen mit der Funktion pprint.pformat() speichern

Im Abschnitt »Saubere Ausgabe« in Kapitel 5 haben Sie zwei Funktionen des Moduls `pprint` kennengelernt. Während `pprint.pprint()` den Inhalt einer Liste oder eines Dictionarys in übersichtlicher Form auf dem Bildschirm anzeigt, gibt `pprint.pformat()` den Text als String aus, der nicht nur übersichtlich formatiert ist, sondern auch syntaktisch korrekter Python-Code ist. Wenn Sie eine Variable, die ein Dictionary enthält, für den späteren Gebrauch speichern wollen, können Sie mit `pprint.pformat()` einen String ausgeben und diesen dann in eine *.py*-Datei schreiben. Diese Datei ist dann ein eigenes Modul, das Sie später jederzeit importieren können, um die darin gespeicherte Variable zu verwenden.

Um das zu veranschaulichen, geben Sie Folgendes in die interaktive Shell ein:

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc':
            'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Als Erstes importieren wir hier `pprint`, damit wir `pprint.pformat()` verwenden können. In der Variablen `cats` ist eine Liste gespeichert, deren einzelne Elemente wiederum Dictionarys sind. Wenn wir die Shell schließen, würde diese Liste normalerweise verloren gehen. Um das zu verhindern, geben wir sie mithilfe von `pprint.pformat()` als String aus, den wir dann ganz einfach als Datei namens *myCats.py* speichern können.

Die Module, die wir mit der Anweisung `import` importieren, sind nichts anderes als Python-Skripte. Wenn wir den von `pprint.pformat()` zurückgegebenen String in einer *.py*-Datei speichern, ist diese Datei ebenfalls ein Modul, das wir wie alle anderen importieren können.

Python-Skripte wiederum sind nichts anderes als Textdateien mit der Endung *.py*. Daher können Python-Programme andere Python-Programme generieren, die Sie dann in Ihre Skripts importieren können.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

Gegenüber der Speicherung von Variablen mit dem Modul `shelve` bietet die Speicherung als *.py*-Datei den Vorteil, dass sich der Inhalt dieser Datei mit einem einfachen Texteditor lesen und bearbeiten lässt, da es sich schließlich um eine Textdatei handelt. In den meisten Anwendungen ist die Verwendung von `shelve` trotzdem die bevorzugte Möglichkeit, um Variablen zu speichern, denn nur einfache Datentypen wie Integer, Fließkommazahlen, Strings, Listen und Dictionarys können in eine einfache Textdatei geschrieben werden. Bei anderen Datentypen, beispielsweise bei `File`-Objekten, ist das dagegen nicht möglich.

## Projekt: Zufallsgenerator für Tests

Stellen Sie sich vor, Sie unterrichten Erdkunde in einer Klasse mit 35 Schülern und wollen einen Test über die Hauptstädte der US-Bundesstaaten schreiben. Allerdings sind in dieser Klasse ein paar Tunichtgute, sodass Sie befürchten müssen, dass einige der Schüler schummeln. Daher ordnen Sie die Fragen in jedem Test zufällig an, sodass kein Test dem anderen gleicht und daher niemand von seinem Nachbarn abschreiben kann. Es wäre allerdings ziemlich langwierig und langweilig, das manuell zu tun. Zum Glück kennen Sie sich aber schon ein bisschen mit Python aus.

Das Programm soll folgende Aufgaben erfüllen:

- 35 verschiedene Tests zusammenstellen
- Für jeden Test 50 Multiple-Choice-Fragen in zufälliger Reihenfolge erstellen
- Zu jeder Frage die korrekte Antwort und drei zufällige falsche Antworten in zufälliger Reihenfolge bereitstellen
- Die Testfragebogen in 35 Textdateien schreiben
- Die Lösungsbogen in 35 Textdateien schreiben

Der Code muss also Folgendes tun:

- Die Namen der Bundesstaaten und der zugehörigen Hauptstädte in einem Dictionary speichern
- Die Funktionen `open()`, `write()` und `close()` für die Textdateien mit den Frage- und den Lösungsbogen aufrufen
- Die Fragen und die möglichen Antworten mit `random.shuffle()` in eine zufällige Reihenfolge bringen

### Schritt 1: Die Daten für den Test in einem Dictionary speichern

Der erste Schritt besteht darin, ein Skelett für das Skript zu erstellen und darin die Daten aufzunehmen, die in dem Test abgefragt werden sollen. Erstellen Sie die Datei *randomQuizGenerator.py* und geben Sie darin folgenden Code ein:

```
#! python3
# randomQuizGenerator.py - Erstellt Testfragebogen mit Fragen und Antworten
# in zufälliger Reihenfolge sowie die zugehörigen Lösungsbogen

import random    ❶

# Die abzufragenden Daten. Die Schlüssel sind die Bundesstaaten, die Werte
# deren Hauptstädte.
capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona':    ❷
'Phoenix', 'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado':
'Denver', 'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Talla-
hassee', 'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise',
'Illinois': 'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines',
'Kansas': 'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge',
'Maine': 'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston',
'Michigan': 'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson',
'Missouri': 'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln',
'Nevada': 'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton',
'New Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Provi-
dence', 'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont': 'Montpe-
lier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West Virginia':
'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Erstellt 35 Testfragebogen
for quizNum in range(35):    ❸
   # TODO: Dateien für Frage- und Lösungsbogen erstellen

   # TODO: Kopf für den Test schreiben

   # TODO: Die Reihenfolge der Bundesstaaten durcheinanderwürfeln

   # TODO: Alle 50 Staaten durchlaufen und für jeden eine Frage erstellen
```

Da dieses Programm die Fragen und Antworten willkürlich anordnen soll, müssen Sie das Modul `random` importieren (❶), um dessen Funktionen nutzen zu können. Die Variable `capitals` (❷) enthält ein Dictionary mit den Namen der US-Bundesstaaten als Schlüssel und deren Hauptstädten als Werte. Da Sie 35 Tests erstellen wollen, muss der Code, der die Dateien für die Frage- und Lösungsbogen generieren soll (zurzeit nur durch die `TODO`-Kommentare angezeigt), in einer `for`-Schleife mit 35 Iterationen stehen (❸). Wenn Sie eine andere Anzahl individueller Tests benötigen, können Sie die Zahl der Iterationen einfach ändern.

### Schritt 2: Die Fragebogendatei erstellen und die Fragen mischen

Beginnen wir nun damit, die `TODO`-Platzhalter zu ersetzen.

Der Code in der Schleife wird 35 Mal ausgeführt – einmal für jeden individuellen Test –, sodass Sie sich in der Schleife immer nur um einen Test auf einmal kümmern müssen. Als Erstes müssen Sie die Datei für den Test erstellen. Sie braucht einen eindeutigen Dateinamen und einen Standardkopf, in dem die Schüler später Name, Datum und Schuljahr eintragen. Anschließend brauchen Sie eine Liste der Bundesstaaten in zufälliger Reihenfolge, aus der Sie dann später die Fragen und Antworten für den Test entnehmen.

Ergänzen Sie *randomQuizGenerator.py* auf folgende Weise:

```
#! python3
# randomQuizGenerator.py - Erstellt Testfragebogen mit Fragen und Antworten
# in zufälliger Reihenfolge sowie die zugehörigen Lösungsbogen

-- schnipp --

# Erstellt 35 Testfragebogen
for quizNum in range(35):
    # Erstellt die Dateien für die Frage- und Lösungsbogen
    quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')   ❶
    answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')
❷

    # Schreibt den Kopf für den Test
    quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')   ❸
    quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' %
                   (quizNum + 1))
    quizFile.write('\n\n')

    # Würfelt die Reihenfolge der Bundesstaaten durcheinander
    states = list(capitals.keys())
    random.shuffle(states)   ❹

    # TODO: Alle 50 Staaten durchlaufen und für jeden eine Frage erstellen
```

Die Dateinamen für die Tests folgen dem Muster *capitalsquiz<N>.txt*, wobei *<N>* die laufende Nummer des Tests ist. Abgeleitet wird diese Nummer von `quizNum`, dem Zähler der `for`-Schleife. Der Lösungsbogen für die Datei *capitalsquiz<N>.txt* befindet sich in der Textdatei *capitalsquiz_answers<N>.txt*. Bei jedem Schleifendurchlauf wird der Platzhalter `%s` in `'capitalsquiz%s.txt'` und `'capitalsquiz_answers%s.txt'` durch `(quizNum + 1)` ersetzt, sodass das erste Paar von Frage- und Lösungsbogen die Namen *capitalsquiz1.txt* und *capitalsquiz_answers1.txt* trägt. Erstellt werden diese Dateien mit Aufrufen von `open()` in (❶) und (❷), wobei als zweites Argument `'w'` übergeben wird, um sie im Schreibmodus zu öffnen.

Die Anweisung `write()` bei (❸) erstellt den Kopf, den die Schüler später ausfüllen müssen. Schließlich wird mithilfe der Funktion `random.shuffle()` eine willkürlich durcheinandergewürfelte Liste der US-Bundesstaaten erstellt (❹). Diese Funktion ordnet die Werte in der ihr übergebenen Liste in zufälliger Reihenfolge an.

### Schritt 3: Die Auswahl der möglichen Antworten zusammenstellen

Als Nächstes müssen Sie die angebotenen Antworten A bis D für die einzelnen Fragen zusammenstellen. Sie benötigen eine weitere `for`-Schleife, um den Inhalt jeder der 50 Fragen in dem Test zu bestimmen, und darin verschachtelt eine dritte `for`-Schleife, um die Multiple-Choice-Antworten für die einzelnen Fragen zu generiert. Nach diesem Schritt sieht der Code wie folgt aus:

```
#! python3
# randomQuizGenerator.py - Erstellt Testfragebogen mit Fragen und Antworten
# in zufälliger Reihenfolge sowie die zugehörigen Lösungsbogen

-- schnipp --

    # Durchläuft alle 50 Staaten und erstellt eine Frage für jeden
    for questionNum in range(50):

        # Ruft die richtigen und falschen Antworten ab
        correctAnswer = capitals[states[questionNum]]   ❶
        wrongAnswers = list(capitals.values())   ❷
        del wrongAnswers[wrongAnswers.index(correctAnswer)]   ❸
        wrongAnswers = random.sample(wrongAnswers, 3)   ❹
        answerOptions = wrongAnswers + [correctAnswer]   ❺
        random.shuffle(answerOptions)   ❻

        # TODO: Fragen und mögliche Antworten in die Testdatei schreiben

        # TODO: Lösungsschlüssel in eine Datei schreiben
```

An die richtige Antwort kommen wir ganz leicht, denn sie ist als Wert im Dictionary `capitals` gespeichert (❶). Die Schleife durchläuft die Bundesstaaten in der durcheinandergewürfelten Liste `states` von `states[0]` bis `states[49]`, ruft die einzelnen Staaten in `capitals` ab und speichert die zugehörige Hauptstadt in `correctAnswer`.

Die Liste der angebotenen falschen Antworten zusammenzustellen, ist dagegen etwas kniffliger. Dazu können Sie *sämtliche* Werte aus dem Dictionary `capitals` kopieren (❷), die richtige Antwort entfernen (❸) und anschließend drei zufällige Werte aus dieser Liste auswählen (❹). Diese zufällige Auswahl lässt sich mit `random.sample()` ganz leicht treffen. Das erste Argument dieser Funktion ist die Liste, aus der Sie auswählen wollen, das zweite die Anzahl der gewünschten Werte. Die

komplette Liste der angebotenen Antwortmöglichkeiten besteht aus diesen drei falschen und der richtigen Antwort (❺). Als Letztes werden diese Antworten zufällig geordnet, damit die richtige Antwort nicht immer D ist.

### Schritt 4: Den Inhalt der Dateien für die Frage- und Lösungsbogen schreiben

Jetzt müssen wir nur noch die Fragen in die Fragebogen und die richtigen Antworten in die Lösungsbogen schreiben. Ergänzen Sie den Code wie folgt:

```
#! python3
# randomQuizGenerator.py - Erstellt Testfragebogen mit Fragen und Antworten
# in zufälliger Reihenfolge sowie die zugehörigen Lösungsbogen

-- schnipp --

   # Durchläuft alle 50 Staaten und erstellt eine Frage für jeden
   for questionNum in range(50):

-- schnipp --

      # Write the question and the answer options to the quiz file.
      quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,
         states[questionNum]))
      for i in range(4):   ❶
         quizFile.write('    %s. %s\n' % ('ABCD'[i], answerOptions[i]))   ❷
      quizFile.write('\n')

      # Write the answer key to a file.
      answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[   ❸
         answerOptions.index(correctAnswer)]))
   quizFile.close()
   answerKeyFile.close()
```

Eine Schleife, die die Integerzahlen von `0` bis `3` durchläuft, schreibt die Antwortmöglichkeiten in die Liste `answerOptions` (❶). Der Ausdruck `'ABCD'[i]` bei (❷) behandelt den String `'ABCD'` als Array und wird in den aufeinanderfolgenden Schleifeniterationen nacheinander zu `'A'`, `'B'`, `C'` und schließlich `'D'` ausgewertet.

In der letzten Zeile (❸) sucht der Ausdruck `answerOptions.index(correctAnswer)` den Integerindex der zufällig geordneten Antwortoptionen. Der Ausdruck `'ABCD'[answerOptions.index(correctAnswer)]` ergibt den Kennbuchstaben der richtigen Antwort, der dann in die Lösungsdatei geschrieben wird.

Die Datei *capitalsquiz1.txt*, die dieses Programm ausgibt, sieht ähnlich aus wie im folgenden Beispiel, wobei die Reihenfolge der Fragen und die angebotenen Antworten natürlich von Fall zu Fall abweichen, da sie vom Ergebnis der Aufrufe von `random.shuffle()` abhängen:

```
Name:

Date:

Period:

                    State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?
    A. Hartford
    B. Santa Fe
    C. Harrisburg
    D. Charleston

2. What is the capital of Colorado?
    A. Raleigh
    B. Harrisburg
    C. Denver
    D. Lincoln
-- schnipp --
```

Die zugehörige Lösungsdatei *capitalsquiz_answers1.txt* sieht wie folgt aus:

```
1. D
2. C
3. A
4. C
--snip--
```

## Projekt: Mehrfach-Zwischenablage

Stellen Sie sich vor, Sie haben die langweilige Aufgabe, auf einer Webseite oder in einer Software viele Formulare mit mehreren Textfeldern auszufüllen. Dank der Zwischenablage müssen Sie ein und denselben Text nicht immer wieder neu eingeben. Allerdings kann sich in der Zwischenablage immer nur ein Text auf einmal befinden. Wenn Sie mehrere verschiedene Texte kopieren und einfügen müssen, bleibt Ihnen trotzdem nichts anderes übrig, als immer wieder die gleichen Texte zu markieren und zu kopieren.

Allerdings können Sie ein Python-Programm schreiben, das sich mehrere Texte merken kann. Diese »Mehrfach-Zwischenablage« werden wir *mcb.pyw* nennen (wobei *mcb* für *multiclipboard* steht). Die Erweiterung *.pyw* bedeutet, dass Python bei der Ausführung dieses Programms kein Terminal-Fenster anzeigt. (Mehr darüber erfahren Sie in Anhang B.)

Das Programm legt jeden einzelnen in die Zwischenablage kopierten Text unter einem Schlüsselwort ab. Wenn Sie beispielsweise `py mcb.pyw save spam` ausführen, wird der aktuelle Inhalt der Zwischenablage unter dem Schlüsselwort `spam` gespeichert. Diesen Text können Sie dann später mit `py mcb.pyw spam` wieder in die Zwischenablage laden. Wenn Sie zwischendurch vergessen sollten, welche Schlüsselwörter Sie verwendet haben, können Sie mit `py mcb.pyw list` eine Liste aller Schlüsselwörter in die Zwischenablage kopieren.

Das Programm soll folgende Aufgaben erledigen:

- Die Befehlszeilenargumente untersuchen
- Bei dem Argument `save` den Inhalt der Schlüsselablage unter dem angegebenen Schlüsselwort speichern
- Bei dem Argument `list` alle Schlüsselwörter in die Zwischenablage kopieren
- Anderenfalls den unter dem übergebenen Schlüsselwort gespeicherten Text in die Zwischenablage übertragen

Der Code muss daher Folgendes tun:

- Die Befehlszeilenargumente in `sys.argv` lesen
- In der Zwischenablage lesen und schreiben
- Eine Shelf-Datei speichern und laden

Unter Windows können Sie das Skript ganz einfach über das Fenster *Ausführen* starten, wenn Sie die Batchdatei *mcb.bat* mit dem folgenden Inhalt erstellen:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

### Schritt 1: Kommentare und Vorbereitungen für die Shelf-Daten

Als Erstes erstellen wir ein Skelett für das Skript, das einige Kommentare enthält und grundlegende Aufgaben zur Einrichtung erfüllt. Der Code sieht vorläufig wie folgt aus:

```
#! python3
# mcb.pyw - Speichert Text und lädt ihn in die Zwischenablage
# Nutzung: py.exe mcb.pyw save <Schlüssel> - Speichert den Inhalt der    ❶
#                                            Zwischenablage unter dem
#                                            Schlüssel
#          py.exe mcb.pyw <Schlüssel> - Lädt den Wert zu dem Schlüssel in
#                                       die Zwischenablage
#          py.exe mcb.pyw list - Lädt alle Schlüsselwörter in die
#                                Zwischenablage
```

```
import shelve, pyperclip, sys    ❷

mcbShelf = shelve.open('mcb')    ❸

# TODO: Inhalt der Zwischenablage speichern

# TODO: Schlüsselwörter auflisten und Inhalt laden

mcbShelf.close()
```

Es ist gängige Praxis, am Anfang einer Skriptdatei Kommentare mit allgemeinen Hinweisen zur Nutzung anzugeben (❶). Sollten Sie vergessen, wie Sie das Skript ausführen müssen, können Sie immer auf diese Gedächtnisstütze zurückgreifen. Nach den Kommentaren importieren Sie die benötigten Module (❷). Für das Kopieren und Einfügen mithilfe der Zwischenablage brauchen Sie das Modul `pyperclip` und zum Lesen der Befehlszeilenargumente ist das Modul `sys` erforderlich. Auch das `shelve`-Modul wird benötigt, denn wenn der Benutzer einen neuen Zwischenablagentext speichern möchte, so geschieht das mithilfe einer Shelf-Datei. Soll der Text später wieder in die Zwischenablage zurückkopiert werden, öffnen Sie einfach die Shelf-Datei und laden den Inhalt in das Programm. Die Namen der Shelf-Dateien erhalten das Präfix *mcb* (❸).

### Schritt 2: Den Inhalt der Zwischenablage unter einem Schlüsselwort speichern

Je nachdem, ob der Benutzer Text unter einem Schlüsselwort speichern, Text in die Zwischenablage laden oder alle vorhandenen Schlüsselwörter auflisten möchte, führt das Programm verschiedene Aufgaben aus. Beginnen wir mit der ersten dieser Aufgaben. Ergänzen Sie den Code wie folgt:

```
#! python3
# mcb.pyw - Speichert Text und lädt ihn in die Zwischenablage
-- schnipp --

# Speichert den Inhalt der Zwischenablage
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':    ❶
        mcbShelf[sys.argv[2]] = pyperclip.paste()    ❷
elif len(sys.argv) == 2:
   # TODO: Schlüsselwörter auflisten und Inhalt laden    ❸

mcbShelf.close()
```

Wenn das erste Befehlszeilenargument (das in der Liste `sys.argv` immer am Index 1 steht) `'save'` lautet (❶), dann ist das zweite Argument das Schlüsselwort, unter dem der aktuelle Inhalt der Zwischenablage gespeichert werden soll. Dieses Schlüs-

selwort wird als Schlüssel für `mcbShelf` verwendet und der Wert ist der Text, der sich zurzeit in der Zwischenablage befindet (❷).

Gibt es nur ein Befehlszeilenargument, dann handelt es sich dabei entweder um `'list'` oder um ein Schlüsselwort, dessen zugehöriger Wert in die Zwischenablage geladen werden soll. Diesen Code werden Sie später schreiben. Vorläufig steht hier nur der `TODO`-Kommentar als Platzhalter und Erinnerungsstütze (❸).

### Schritt 3: Schlüsselwörter auflisten und Inhalte laden

Implementieren wir nun die restlichen beiden Fälle, in denen der Benutzer über ein Schlüsselwort Text in die Zwischenablage laden oder die Liste der verfügbaren Schlüsselwörter abrufen möchte. Ergänzen Sie den Code wie folgt:

```
#! python3
# mcb.pyw - Speichert Text und lädt ihn in die Zwischenablage
-- schnipp --

# Speichert den Inhalt der Zwischenablage
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':   1
        mcbShelf[sys.argv[2]] = pyperclip.paste()   2
elif len(sys.argv) == 2:
   # Listet Schlüsselwörter auf und lädt Inhalte
   if sys.argv[1].lower() == 'list':   ❶
      pyperclip.copy(str(list(mcbShelf.keys())))   ❷
   elif sys.argv[1] in mcbShelf:
      pyperclip.copy(mcbShelf[sys.argv[1]])   ❸

mcbShelf.close()
```

Wenn es nur ein Befehlszeilenargument gibt, prüfen wir zunächst, ob es `'list'` ist (❶). Ist das tatsächlich der Fall, wird eine Stringdarstellung der Liste von Shelf-Schlüsseln in die Zwischenablage kopiert (❷). Der Benutzer kann diese Liste dann in einen Texteditor kopieren, um sie zu lesen.

Anderenfalls können Sie davon ausgehen, dass es sich bei dem Befehlszeilenargument um ein Schlüsselwort handelt. Wenn es in `mcbShelf` als Schlüssel vorkommt, laden Sie den zugehörigen Wert in die Zwischenablage (❸).

Das war es auch schon! Je nachdem, welches Betriebssystem auf Ihrem Computer läuft, sind zum Starten dieses Programms unterschiedliche Schritte erforderlich. Weitere Hinweise zu den verschiedenen Betriebssystemen erhalten Sie in Anhang B.

In dem Passwortmanager aus Kapitel 6 haben Sie Passwörter in einem Dictionary gespeichert. Um Passwörter hinzuzufügen oder zu ändern, mussten Sie den Quellcode des Programms ändern. Das ist jedoch keine gute Vorgehensweise, da

sich durchschnittliche Benutzer nicht trauen, in den Quellcode einzugreifen, nur um Daten zu ändern. Außerdem laufen Sie bei jeder Änderung am Quellcode Gefahr, versehentlich irgendwelche Bugs hervorzurufen. Wenn Sie die Daten für ein Programm statt im Code an anderer Stelle speichern, lässt sich das Programm von anderen einfacher benutzen und ist weniger fehleranfällig.

## Zusammenfassung

Dateien werden in Ordnern (Verzeichnissen) abgelegt, wobei der Pfad den Speicherort der Datei angibt. Da jedes Programm, das auf einem Computer läuft, über ein Arbeitsverzeichnis verfügt, können Dateipfade relativ zu diesem aktuellen Speicherort angegeben werden, anstatt jedes Mal den vollständigen (absoluten) Pfad auszuschreiben. Das Modul `os.path` verfügt über viele Funktionen für den Umgang mit Dateipfaden.

Programme können auch direkt mit den Inhalten von Textdateien umgehen. Mit der Funktion `open()` öffnen Sie solche Dateien, um ihren Inhalt zu lesen, entweder mit `read()` in Form eines einzigen, langen Strings oder mit `readlines()` als Liste von Strings. Die Funktion `open()` kann Dateien auch im Schreib- oder Anhängemodus öffnen, um neue Textdateien zu erstellen oder Inhalte zu vorhandenen Textdateien hinzuzufügen.

In den vorherigen Kapiteln haben Sie die Zwischenablage verwendet, um dem Programm große Mengen von Texten zu übergeben, anstatt den ganzen Text manuell einzugeben. Wie Sie jetzt wissen, können Ihre Programme Dateien auch direkt von der Festplatte lesen. Das ist ein großer Vorteil, da solche Dateien nicht so flüchtig sind wie der Inhalt der Zwischenablage.

Im nächsten Kapitel lernen Sie den Umgang mit Dateien – wie Sie sie kopieren, löschen, umbenennen, verschieben usw.

## Wiederholungsfragen

1. Wozu ist ein relativer Pfad relativ?
2. Womit beginnt ein absoluter Pfad?
3. Was machen die Funktionen `os.getcwd()` und `os.chdir()`?
4. Worum handelt es sich bei den Ordnern `.` und `..`?
5. Welcher Teil von *C:\bacon\eggs\spam.txt* ist der Verzeichnisname und welcher der Grundname?
6. Welche drei Modusargumente können Sie an die Funktion `open()` übergeben?
7. Was geschieht, wenn Sie eine bereits vorhandene Datei im Schreibmodus öffnen?

8. Was ist der Unterschied zwischen den Methoden `read()` und `readlines()`?
9. Welche Datenstruktur stellt ein Shelf-Wert dar?

## Übungsprojekte

Entwerfen und schreiben Sie zur Übung die folgenden Programme:

### Erweiterte Mehrfach-Zwischenablage

Ergänzen Sie das Programm für die Mehrfach-Zwischenablage aus diesem Kapitel um das Befehlszeilenargument `delete` *<schlüsselwort>*, mit dem Sie ein Schlüsselwort aus dem Shelf löschen. Fügen Sie außerdem das Befehlszeilenargument `delete` hinzu, das *alle* Schlüsselwörter löscht.

### Lückentextspiel

Erstellen Sie ein Programm für ein Lückentextspiel, bei dem der Benutzer an den Stellen, an denen in der Textdatei *ADJECTIVE*, *NOUN*, *ADVERB* oder *VERB* vorkommt, ein eigenes Wort der entsprechenden Art eingeben kann. Die Textdatei kann dabei beispielsweise wie folgt aussehen:

```
The ADJECTIVE panda walked to the NOUN and then VERB. A nearby NOUN was
unaffected by these events.
```

Das Programm findet die zu ersetzenden Stellen und fordert den Benutzer auf, Wörter dafür einzugeben:

```
Enter an adjective:
silly
Enter a noun:
chandelier
Enter a verb:
screamed
Enter a noun:
pickup truck
```

Dadurch entsteht folgender Text:

```
The silly panda walked to the chandelier and then screamed. A nearby pickup
truck was unaffected by these events.
```

Das Ergebnis soll auf dem Bildschirm ausgegeben und in einer neuen Textdatei gespeichert werden.

### Regex-Suche

Schreiben Sie ein Programm, das alle *.txt*-Dateien in einem Ordner öffnet und nach sämtlichen Zeilen sucht, die mit dem vom Benutzer angegebenen regulären Ausdruck übereinstimmen. Die Ergebnisse sollen auf dem Bildschirm ausgegeben werden.