9

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 String vorstellen, dessen Größe sich 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-Laptop habe ich beispielsweise eine Datei mit dem Namen *projects.docx* im Pfad *C:\Users\Al\Documents*. Der Teil des Dateinamens hinter dem Punkt wird als *Dateinamenerweiterung* oder *Endung* bezeichnet und gibt den Typ der Datei an. Bei *project.docx* handelt es sich um ein Word-Dokument, und *Users*, *Al* und *Documents* sind *Ordner* (oder *Ver-*

zeichnisse). Ordner können Dateien sowie andere Ordner enthalten. Beispielsweise befindet sich *projects.docx* im Ordner *Documents*, der wiederum im Ordner *Al* innerhalb des Ordners *Users* steckt. Diese Ordnerstruktur sehen Sie in Abb. 9–1.

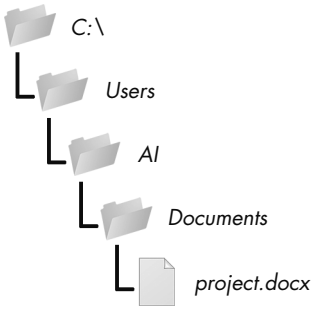


Abb. 9–1 *Eine Datei in einer Ordnerhierarchie*

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 macOS und Linux ist /. In diesem Buch verwende ich den Windows-Stammordner *C:*. Wenn Sie die Beispiele in einer interaktiven Shell unter macOS 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 weitere, mit Buchstaben gekennzeichnete Stammlaufwerke wie *D:* oder *E:*, unter macOS 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 macOS jedoch nicht.

Hinweis

Auf Ihrem System befinden sich natürlich andere Dateien und Ordner als auf meinem. Die Ergebnisse der Beispiele in diesem Kapitel werden daher nicht exakt gleich aussehen. Versuchen Sie aber, sie mithilfe Ihrer Dateien und Ordner nachzuvollziehen.

Backslash unter Windows und Schrägstrich unter macOS und Linux

Unter Windows werden Pfade mit Backslashes, also umgekehrten Schrägstrichen (\), als Trennzeichen zwischen den Ordnernamen geschrieben, unter macOS und Linux jedoch mit einem normalen Schrägstrich (/). Wenn Ihre Programme auf

allen Betriebssystemen laufen sollen, müssen Sie Ihre Python-Skripte für beide Fälle auslegen.

Zum Glück lässt sich das mit der Funktion `Path()` aus dem Modul `pathlib` leicht erledigen. Wenn Sie ihr die Strings der einzelnen Datei- und Ordnernamen eines Pfades übergeben, gibt sie den Dateipfad als String mit den richtigen Trennzeichen für das vorliegende Betriebssystem zurück. Probieren Sie Folgendes in der interaktiven Shell aus:

```
>>> from pathlib import Path
>>> Path('spam', 'bacon', 'eggs')

WindowsPath('spam/bacon/eggs')

>>> str(Path('spam', 'bacon', 'eggs'))
'spam\\bacon\\eggs'
```

Die übliche Vorgehensweise für den Import von `pathlib` besteht darin, `from pathlib import Path` anzugeben, da Sie sonst überall, wo Sie `Path` in Ihrem Code brauchen, `pathlib.Path` schreiben müssten, was ziemlich viel Tipparbeit wäre.

In der interaktiven Shell auf Windows gibt `Path('spam', 'bacon', 'eggs')` ein `WindowsPath`-Objekt für den zusammengefügten Pfad zurück. Dargestellt wird dieses Objekt als `WindowsPath('spam/bacon/eggs')`. Obwohl Windows-Dateipfade Backslashes enthalten, werden bei der Anzeige in der interaktiven Shell normale Schrägstriche verwendet, da Open-Source-Softwareentwickler nun mal Linux bevorzugen.

Wenn Sie aus diesem Pfad einen einfachen Textstring gewinnen wollen, können Sie ihn an die Funktion `str()` übergeben. In unserem Fall führt das zu dem Rückgabewert `'spam\\bacon\\eggs'` (mit doppelten Backslashes, da jeder Backslash mit einem zweiten maskiert werden muss). Auf Linux dagegen gibt `Path()` ein `PosixPath`-Objekt zurück, dessen Übergabe an `str()` den Rückgabewert `'spam/bacon/eggs'` ergibt. (Bei *POSIX* handelt es sich um einen Satz von Standards für Unix-artige Betriebssysteme wie Linux.)

Diese `Path`-Objekte (je nach Betriebssystem also `WindowsPath`- oder `PosixPath`-Objekte) können Sie an verschiedene der Funktionen zur Dateibearbeitung übergeben, die Sie in diesem Kapitel noch kennenlernen werden. Beispielsweise hängt der folgende Code Dateinamen aus einer Liste hinten an den Ordnernamen an:

```
>>> from pathlib import Path
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(Path(r'C:\Users\Al', filename))
C:\Users\Al\accounts.txt
```

```
C:\Users\Al\details.csv
C:\Users\Al\invite.docx
```

Auf Windows trennt der Backslash Verzeichnisse voneinander, weshalb Sie ihn nicht in Dateinamen verwenden können. Dagegen sind Backslashes in Dateinamen auf macOS und Linux zulässig. Während `Path(r'spam\eggs')` auf Windows zwei verschiedene Ordner bezeichnet (oder die Datei *eggs* im Ordner *spam*), steht derselbe Befehl auf macOS und Linux für einen einzelnen Ordner (oder eine Datei) namens *spam\eggs*. Aus diesem Grund ist es sinnvoll, in Python-Code nur normale Schrägstriche zu verwenden (was ich auch im Rest dieses Kapitels so handhaben werde). Das Modul `pathlib` stellt sicher, dass der Code auf allen Betriebssystemen funktioniert.

Das Modul `pathlib` wurde in Python 3.4 als Ersatz für die älteren `os.path`-Funktionen eingeführt. Seit Python 3.6 wird es von den Modulen der Python-Standardbibliothek unterstützt, aber wenn Sie noch mit Python 2 arbeiten, sollten Sie `pathlib2` verwenden. Damit steht Ihnen die Funktionalität von `pathlib` auch auf Python 2.7 zur Verfügung. In Anhang A finden Sie eine Anleitung, um `pathlib2` mithilfe von `pip` zu installieren. Immer wenn ich eine ältere `os.path`-Funktion durch `pathlib` ersetzt habe, mache ich mir eine Notiz dazu. Mehr über die älteren Funktionen erfahren Sie auf *https://docs.python.org/3/library/os.path.html*.

Pfade mit dem Operator / zusammenfügen

Den Operator `+` können wir verwenden, um zwei Integer- oder Fließkommazahlen wie in dem Ausdruck `2 + 2` zu addieren, der zu `4` ausgewertet wird, aber auch, um zwei Strings zu verketten, etwa in `'Hello' + 'World'`, was `'HelloWorld'` ergibt. Ebenso ist es auch möglich, den Operator /, der gewöhnlich zur Division dient, zur Verbindung von `Path`-Objekten und Strings zu nutzen. Das ist praktisch, um ein von der Funktion `Path()` erstelltes `Path`-Objekt zu bearbeiten.

Probieren Sie das wie folgt in der interaktiven Shell aus:

```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

Die Anwendung des Operators / auf `Path`-Objekte macht die Kombination von Pfaden so einfach wie die Stringverkettung. Außerdem ist es sicherer als eine Stringverkettung oder die Methode `join()`. Betrachten Sie dazu die folgenden Beispiele:

```
>>> homeFolder = r'C:\Users\Al'
>>> subFolder = 'spam'
>>> homeFolder + '\\' + subFolder
'C:\\Users\\Al\\spam'
>>> '\\'.join([homeFolder, subFolder])
'C:\\Users\\Al\\spam'
```

Ein Skript, das diesen Code nutzt, ist nicht sicher, da die Backslashes nur auf Windows funktionieren. Sie könnten zwar eine `if`-Anweisung hinzufügen, die `sys.platform` überprüft (eine Systemvariable, die einen String mit der Angabe des Betriebssystems enthält), um zu entscheiden, welche Art von Schrägstrich zu verwenden ist, aber solchen Code überall dort einzufügen, wo er gebraucht wird, kann mühsam und fehleranfällig sein.

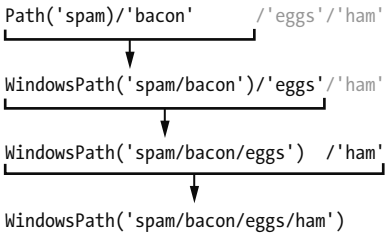
Das Modul `pathlib` löst all diese Probleme durch die Verwendung des Operators /, der die Pfade unabhängig vom Betriebssystem korrekt kombiniert. Im folgenden Beispiel werden damit dieselben Pfade konstruiert wie im vorherigen Code:

```
>>> homeFolder = Path('C:/Users/Al')
>>> subFolder = Path('spam')
>>> homeFolder / subFolder
WindowsPath('C:/Users/Al/spam')
>>> str(homeFolder / subFolder)
'C:\\Users\\Al\\spam'
```

Bei der Verwendung des Operators / zur Kombination von Pfaden müssen Sie lediglich darauf achten, dass einer der beiden ersten Werte ein `path`-Objekt ist. Python gibt eine Fehlermeldung aus, wenn Sie versuchen, Folgendes in die interaktive Shell einzugeben:

```
>>> 'spam' / 'bacon' / 'eggs'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Python wertet einen Ausdruck mit dem Operator / von links nach rechts zu einem `Path`-Objekt aus. Damit sich ein solches Objekt ergeben kann, muss entweder der erste oder der zweite Wert von links selbst ein `Path`-Objekt sein. Die Auswertung läuft wie in der folgenden Grafik gezeigt ab:



Wenn Sie die Fehlermeldung `TypeError: unsupported operand type(s) for /: 'str' and 'str'` erhalten, müssen Sie auf die linke Seite des Ausdrucks ein `Path`-Objekt stellen.

Der Operator / ersetzt die ältere Funktion `os.path.join()`, die auf *https://docs.python.org/3/library/os.path.html#os.path.join* genauer beschrieben wird.

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.

Hinweis

Auch wenn »Ordner« die modernere Bezeichnung für ein Verzeichnis ist, lautet der übliche Begriff *aktuelles Arbeitsverzeichnis* und nicht etwa »aktueller Arbeitsordner«.

Um den String des aktuellen Arbeitsverzeichnisses abzurufen, verwenden Sie die Funktion `Path.cwd()` (wobei `cwd` für *current working directory* steht), und um es zu ändern die Funktion `os.chdir()`:

```
>>> from pathlib import Path
>>> import os
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')'
>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')
```

Hier ist das aktuelle Arbeitsverzeichnis zunächst *C:\Users\Al\AppData\Local\Programs\Python\Python37*, weshalb der Dateiname *project.docx* auf *C:\Users\Al\AppData\Local\Programs\Python\Python37\project.docx* verweist. Wenn wir das aktuelle Arbeitsverzeichnis anschließend in *C:\Windows\System32* ändern, wird *project.docx* als *C:\Windows\System32\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 "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:/ThisFolderDoesNotExist'
```

Es gibt keine `pathlib`-Funktion, um das Arbeitsverzeichnis zu ändern, da ein solcher Wechsel bei laufendem Programm zu schwer auffindbaren Bugs führen kann.

Die ältere Funktion, um das aktuelle Arbeitsverzeichnis als String abzurufen, ist `os.getcwd()`.

Das Benutzerverzeichnis

Alle Benutzer eines Computers haben einen Ordner für ihre eigenen Dateien, der als *Benutzerordner* oder *Benutzerverzeichnis* bezeichnet wird. Mit `Path.home()` können Sie ein `Path`-Objekt für diesen Ordner abrufen:

```
>>> Path.home()
WindowsPath('C:/Users/Al')
```

An welchem Speicherort sich die Benutzerverzeichnisse befinden, hängt vom Betriebssystem ab:

- Auf Windows befinden sich Benutzerverzeichnisse in *C:\Users*.
- Auf macOS befinden sich Benutzerverzeichnisse in */Users*.
- Auf Linux befinden sich Benutzerverzeichnisse meistens in */home*.

Da Ihre Skripte sehr wahrscheinlich die Berechtigung haben, Dateien in Ihrem Benutzerverzeichnis zu lesen und zu schreiben, stellt es den idealen Speicherort für die Dateien dar, mit denen Ihre Python-Programme arbeiten.

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. 9–2 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.

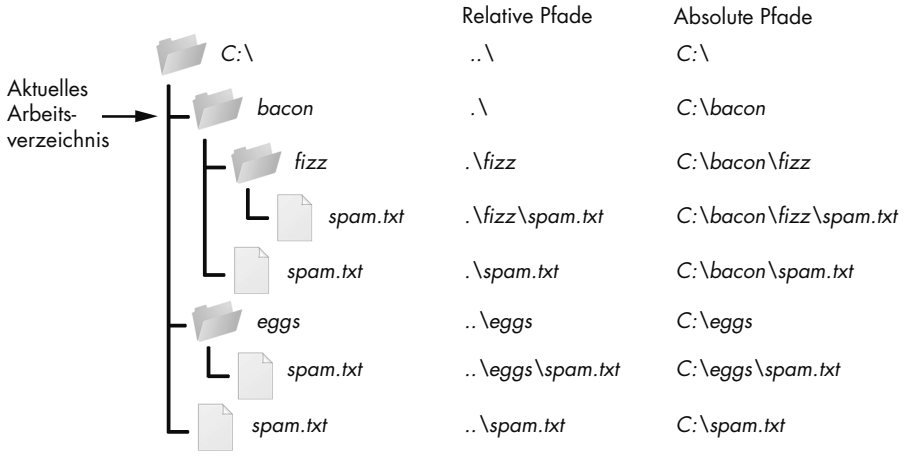


Abb. 9–2 *Relative und absolute Pfade für die Ordner und Dateien im Arbeitsverzeichnis C:\bacon*

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. 9–3.

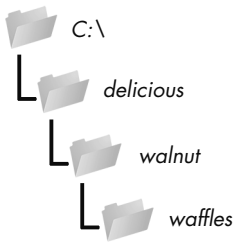


Abb. 9–3 *Das Ergebnis von* `os.makedirs('C:\\delicious\\walnut\\waffles')`

Um ein Verzeichnis aus einem `Path`-Objekt zu erstellen, rufen Sie die Methode `mkdir()` auf. Zum Beispiel legt der folgende Code den Ordner *spam* im Benutzerordner auf meinem Computer an:

```
>>> from pathlib import Path
>>> Path(r'C:\Users\Al\spam').mkdir()
```

Beachten Sie, dass `mkdir()` immer nur ein Verzeichnis auf einmal erstellt, also nicht mehrere Unterverzeichnisse wie `os.makedirs()`.

Absolute und relative Pfade verwenden

Das Modul `pathlib` enthält Methoden, um zu prüfen, ob ein gegebener Pfad ein absoluter Pfad ist, und um den absoluten Pfad zu einem relativen Pfad zurückzugeben.

Der Aufruf der Methode `is_absolute()` für ein `Path`-Objekt gibt `True` zurück, wenn das Objekt einen absoluten Pfad darstellt, anderenfalls `False`. Probieren Sie das wie folgt in der interaktiven Shell aus, wobei Sie jedoch Dateien und Ordner verwenden müssen, die auf Ihrem System vorhanden sind, und nicht die hier angegebenen:

```
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().is_absolute()
True
>>> Path('spam/bacon/eggs').is_absolute()
False
```

Um den absoluten Pfad zu einem relativen zu erhalten, können Sie `Path.cwd() /` vor das `Path`-Objekt für den relativen Pfad stellen. Schließlich meinen wir mit »relativer Pfad« fast ausschließlich einen Pfad relativ zum aktuellen Arbeitsverzeichnis. Probieren Sie das wie folgt aus:

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.cwd() / Path('my/relative/path')
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37/my/relative/
path')
```

Ist der relative Pfad auf einen anderen Pfad als das aktuelle Arbeitsverzeichnis bezogen, ersetzen Sie einfach `Path.cwd()` durch diesen anderen Pfad. Das folgende Beispiel ruft einen absoluten Pfad im Benutzerverzeichnis statt im aktuellen Arbeitsverzeichnis ab:

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.home() / Path('my/relative/path')
WindowsPath('C:/Users/Al/my/relative/path')
```

Auch das Modul `os.path` enthält nützliche Funktionen für absolute und relative Pfade:

- Die Funktion `os.path.abspath(`*`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:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'
>>> os.path.abspath('.\\Scripts')
'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Da *C:\Users\Al\AppData\Local\Programs\Python\Python37* beim Aufruf von `os.path.abspath()` das Arbeitsverzeichnis war, steht der Punkt für den absoluten Pfad `'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'`.

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'
```

Wenn sich der relative Pfad zwar im selben Elternordner befindet wie der gegebene Pfad, aber im Unterverzeichnis eines anderen Pfades, z. B. `'C:\\Windows'` und `'C:\\spam\\eggs'`, können Sie mit der Zwei-Punkte-Schreibweise zu dem Elternordner zurückkehren.

Die Komponenten eines Dateipfads abrufen

Aus den Attributen eines gegebenen `Path`-Objekts können Sie die einzelnen Komponenten des Dateipfads als Strings abrufen. Das kann praktisch sein, um einen neuen Dateipfad auf der Grundlage eines vorhandenen zusammenzustellen. Die verschiedenen Attribute sehen Sie in Abb. 9–4.

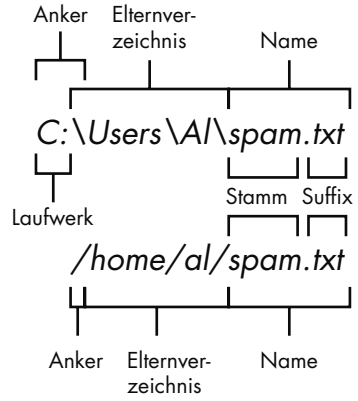


Abb. 9–4 *Komponenten von Dateipfaden auf Windows (oben) und macOS und Linux (unten)*

Ein Dateipfad setzt sich aus folgenden Komponenten zusammen:

- Der *Anker* (`anchor`) ist der Wurzelordner des Dateisystems (auf Windows Stammordner genannt).
- Auf Windows wird als *Laufwerksangabe* (`drive`) ein einzelner Buchstabe verwendet, der häufig eine physische Festplatte oder ein anderes Speichermedium bezeichnet.
- Der *Elternordner* (`parent`) ist das Verzeichnis, in dem sich die Datei befindet.
- Der *Name* (`name`) der Datei besteht aus dem *Stamm* (`stem`) oder *Grundnamen* (nicht zu verwechseln mit dem Stammordner auf Windows!) und dem *Suffix* (`suffix`), also der Erweiterung.

Windows-`Path`-Objekte haben das Attribut `drive`, macOS- und Linux-`Path`-Objekte dagegen nicht. Der erste Backslash ist im Attribut `drive` nicht enthalten.

Wie Sie die einzelnen Attribute aus dem Dateipfad abrufen, zeigen die folgenden Beispiele:

```
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.anchor
'C:\\'
>>> p.parent # Dies ist ein Path-Objekt, kein String
WindowsPath('C:/Users/Al')
>>> p.name
'spam.txt'
>>> p.stem
'spam'
>>> p.suffix
'.txt'
>>> p.drive
'C:'
```

Der Abruf dieser Attribute ergibt jeweils einen einfachen String. Die einzige Ausnahme bildet `parent`, das zu einem weiteren `Path`-Objekt ausgewertet wird.

Das Attribut `parents` (nicht zu verwechseln mit `parent`!) wird mithilfe eines Integerindex zu den einzelnen Vorfahrenordnern des `Path`-Objekts ausgewertet:

```
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().parents[0]
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python')
>>> Path.cwd().parents[1]
WindowsPath('C:/Users/Al/AppData/Local/Programs')
>>> Path.cwd().parents[2]
WindowsPath('C:/Users/Al/AppData/Local')
>>> Path.cwd().parents[3]
WindowsPath('C:/Users/Al/AppData')
>>> Path.cwd().parents[4]
WindowsPath('C:/Users/Al')
>>> Path.cwd().parents[5]
WindowsPath('C:/Users')
>>> Path.cwd().parents[6]
WindowsPath('C:/')
```

Das ältere Modul `os.path` verfügt über ähnliche Funktionen, um die einzelnen Teile eines Pfades als String zurückzugeben. So gibt `os.path.dirname(`*`path`*`)` 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 Pfades sind in Abb. 9–5 dargestellt.

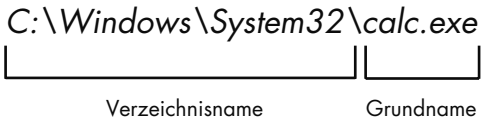


Abb. 9–5 *Der Grundname folgt auf den letzten Schrägstrich im Pfad und ist mit dem Dateinamen identisch. Alles, was links von dem letzten Schrägstrich steht, gehört dagegen zum Verzeichnisnamen.*

Geben Sie beispielsweise Folgendes in die interaktive Shell ein:

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

Wenn Sie sowohl den Verzeichnis- als auch den Grundnamen eines Pfades 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 es nicht möglich ist, `os.path.split()` einen Dateipfad zu übergeben und daraus eine Liste der Strings für die einzelnen Ordner zu gewinnen. Für diesen Zweck müssen Sie den Pfad mit der Stringmethode `split()` zerlegen und ihr die Variable `os.sep` übergeben (beachten Sie, dass sich `sep` in `os` befindet, nicht in `os.path`), in der das Ordnertrennzeichen für das vorliegende Betriebssystem gespeichert ist, also `'\\'` für Windows und `'/'` für macOS und Linux. Eine Aufteilung an diesem Trennzeichen ergibt eine Liste der einzelnen Ordner.

Betrachten Sie dazu das folgende Beispiel in der interaktiven Shell:

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

Dies gibt alle Teile des Pfades als Strings zurück.

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

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

Die Stringmethode `split()` gibt eine Liste der einzelnen Teile des Pfades zurück.

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')
27648
>>> 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 27.648 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)
2559970473
```

Dieser Code durchläuft alle Dateien im Ordner *C:\Windows\System32* und erhöht 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.

Eine Dateiliste mit Glob-Mustern bearbeiten

Wenn Sie mit bestimmten Dateien arbeiten wollen, geht das mit der Methode `glob()` einfacher als mit `listdir()`. `Path`-Objekte verfügen über die Methode `glob()`, die den Inhalt eines Ordners anhand von *Glob-Mustern* auflistet. Dabei handelt es sich um eine vereinfachte Form von regulären Ausdrücken, die oftmals an der Befehlszeile verwendet werden. Die Methode `glob()` gibt ein Generatorobjekt zurück (dessen Erklärung hier zu weit führen würde), das Sie zur einfachen Anzeige in der interaktiven Shell an `list()` übergeben müssen:

```
>>> p = Path('C:/Users/Al/Desktop')
>>> p.glob('*')
<generator object Path.glob at 0x000002A6E389DED0>
>>> list(p.glob('*')) # Erstellt eine Liste aus dem Generator.
[WindowsPath('C:/Users/Al/Desktop/1.png'), WindowsPath('C:/Users/Al/
Desktop/22-ap.pdf'), WindowsPath('C:/Users/Al/Desktop/cat.jpg'),
-- schnipp --
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

Das Sternchen steht für »mehrere beliebige Zeichen«, weshalb `p.glob('*')` einen Generator für alle Dateien in dem in `p` gespeicherten Pfad zurückgibt.

Ebenso wie bei regulären Ausdrücken können Sie auch anspruchsvolle Glob-Ausdrücke zusammenstellen:

```
>>> list(p.glob('*.txt') # Listet alle Textdateien auf
[WindowsPath('C:/Users/Al/Desktop/foo.txt'),
-- schnipp --
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

Das Glob-Muster `'*.txt'` gibt Dateien zurück, deren Namen mit einer beliebigen Kombination von Zeichen beginnen, aber mit dem String `'.txt'` enden, also der Erweiterung für Textdateien.

Im Gegensatz zum Sternchen steht das Fragezeichen nur für ein einzelnes beliebiges Zeichen:

```
>>> list(p.glob('project?.docx')
[WindowsPath('C:/Users/Al/Desktop/project1.docx'), WindowsPath('C:/Users/Al/
Desktop/project2.docx'),
-- schnipp --
WindowsPath('C:/Users/Al/Desktop/project9.docx')]
```

Der Glob-Ausdruck `'project?.docx'` findet sowohl `'project1.docx'` als auch `'project5.docx'`, aber nicht `'project10.docx'`, da `?` nur für ein einziges Zeichen steht und nicht für einen zweistelligen String wie `'10'`.

Um noch komplexere Glob-Ausdrücke zu erstellen, können Sie das Sternchen und das Fragezeichen auch kombinieren:

```
>>> list(p.glob('*.?x?')
[WindowsPath('C:/Users/Al/Desktop/calc.exe'), WindowsPath('C:/Users/Al/
Desktop/foo.txt'),
-- schnipp --

WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

Der Glob-Ausdruck `'*.?x?'` findet Dateien mit einem beliebigen Namen und einer dreistelligen Erweiterung, in deren Mitte ein `x` steht.

Wenn Sie mit `glob()` Dateien mit bestimmten Attributen auswählen, können Sie sehr leicht die Dateien in einem Verzeichnis angeben, an denen Sie bestimmte Operationen durchführen wollen. Dazu können Sie in einer `for`-Schleife den von `glob()` zurückgegebenen Generator durchlaufen:

```
>>> p = Path('C:/Users/Al/Desktop')
>>> for textFilePathObj in p.glob('*.txt'):
...     print(textFilePathObj) # Gibt das Path-Objekt als String aus
...     # Macht irgendetwas mit der Textdatei
...
C:\Users\Al\Desktop\foo.txt
C:\Users\Al\Desktop\spam.txt
C:\Users\Al\Desktop\zzz.txt
```

Wenn Sie eine Operation auf allen Dateien in einem Verzeichnis ausführen wollen, können Sie dazu `os.listdir(p)` oder `p.glob('*')` verwenden.

Die Gültigkeit von Pfaden prüfen

Viele Python-Funktionen werden mit einer Fehlermeldung beendet, wenn Sie ihnen einen Pfad übergeben, den es gar nicht gibt. Zum Glück verfügen `Path`-Objekte über Methoden, die prüfen, ob ein gegebener Pfad existiert und ob es sich dabei um eine Datei oder einen Ordner handelt. Unter der Voraussetzung, dass die Variable `p` ein `Path`-Objekt enthält, gilt Folgendes: