7. Eingabe und Ausgabe

Es gibt verschiedene Arten, die Ausgabe eines Programmes darzustellen: Daten können in menschenlesbarer Form ausgegeben werden oder in eine Datei für die spätere Verwendung geschrieben werden. Dieses Kapitel beschreibt einige Möglichkeiten.

7.1. Ausgefallenere Ausgabeformatierung

Bis jetzt sind uns zwei Arten der Ausgabe von Werten begegnet: Ausdrucksanweisungen (expression statements) und die `print()`-Funktion. (Eine dritte Möglichkeit ist die `write()`-Methode von Dateiobjekten; die Standardausgabedatei kann als `sys.stdout` referenziert werden. In der Bibliotheksreferenz gibt es dazu weitere Informationen.)

Oft will man mehr Kontrolle über die Formatierung der Ausgabe haben als nur Leerzeichen-getrennte Werte auszugeben. Es gibt zwei Arten die Ausgabe zu formatieren: Die erste Möglichkeit ist, dass man die gesamte Verarbeitung der Zeichenketten selbst übernimmt; indem man Slicing- und Verknüpfungsoperationen benutzt, kann man jede denkbare Anordnung zusammenstellen. Der Typ `string` hat einige Methoden, die ein paar nützliche Operationen ausführen, um Zeichenketten auf eine bestimmte Länge aufzufüllen; diese werden wir in Kürze behandeln. Die zweite Möglichkeit ist die Benutzung der `format()`-Methode.

Das `string`-Modul enthält eine Klasse `Template`, die noch einen Weg bietet, Werte in Zeichenketten zu ersetzen.

Eine Frage bleibt natürlich: Wie konvertiert man Werte zu Zeichenketten? Glücklicherweise kennt Python Wege, um jeden Wert in eine Zeichenkette umzuwandeln: Man übergibt den Wert an die `repr()`- oder `str()`-Funktion.

Die `str()`-Funktion ist dazu gedacht eine möglichst menschenlesbare Repräsentation des Wertes zurückzugeben, während `repr()` dazu gedacht ist, vom Interpreter lesbar zu sein (oder einen `SyntaxError` erzwingt, wenn es keine äquivalente Syntax gibt). Für Objekte, die keine besondere menschenlesbare Repräsentation haben, gibt `str()` denselben Wert wie `repr()` zurück. Viele Werte wie Nummern oder Strukturen wie Listen und Dictionaries benutzen für beide Funktionen dieselbe Repräsentation. Besonders Zeichenketten haben zwei verschiedene Repräsentationen.

Ein paar Beispiele:

```
>>> s = 'Hallo Welt!'
>>> str(s)
```

```
'Hallo Welt!'
>>> repr(s)
"'Hallo Welt!'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Der Wert von x ist ' + repr(x) + ', und y ist ' + repr(y) + '...'
>>> print(s)
Der Wert von x ist 32.5, und y ist 40000...
>>> #repr() bei einer Zeichenkette benutzt Anführungszeichen und Backslashes:
... hello = 'Hallo Welt\n'
>>> hellos = repr(hello)
>>> print(hellos)
'Hallo Welt\n'
>>> # Das Argument für repr() kann jedes Pythonobjekt sein:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Hier zwei Möglichkeiten, eine Tabelle von Quadrat- und Kubikzahlen zu erstellen:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Achte auf die Benutzung von 'end' in der vorherigen Zeile
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Beachte, dass im ersten Beispiel ein Leerzeichen pro Spalte durch die Funktionsweise von `print()` hinzugefügt wird: Sie trennt ihre Argumente mit Leerzeichen.)

Dieses Beispiel hat die `rjust()`-Methode von Zeichenkettenobjekten gezeigt, die eine Zeichenkette in einem Feld der gegebenen Breite rechtsbündig macht, indem sie diese links mit Leerzeichen auffüllt. Es gibt die ähnlichen Methoden `ljust()` und `center()`. Diese Methoden schreiben nichts, sondern geben eine neue Zeichenkette zurück. Ist die gegebene Zeichenkette zu lang, schneiden sie nichts, sondern geben diese unverändert zurück; dies wird die Anordnung durcheinanderbringen, aber ist meistens besser als die

Alternative, dass der Wert verfälscht wird. (Will man wirklich abschneiden, kann man immer noch eine Slicing-Operation hinzufügen, zum Beispiel `x.ljust(n)[:n]`.)

Es gibt noch eine weitere Methode, **`zfill()`**, die eine numerische Zeichenkette mit Nullen auffüllt. Sie versteht auch Plus- und Minuszeichen:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Die einfachste Benutzung der **`format()`**-Methode sieht so aus:

```
>>> print('Wir sind die {}, die "{}!" sagen.'.format('Ritter', 'Ni'))
Wir sind die Ritter, die "Ni!" sagen.
```

Die Klammern und die Zeichen darin (genannt Formatfelder - format fields) werden mit den Objekten ersetzt, die der **`format()`**-Methode übergeben werden. Eine Nummer in den Klammern bezieht sich auf die Position des Objektes, die der **`format()`**-Methode übergeben werden.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Werden Schlüsselwortargumente in der **`format()`**-Methode benutzt, können deren Werte durch die Benutzung des Argumentnamens referenziert werden.

```
>>>print('Dieses {Speise} ist {Adjektiv}.'.format(Speise='Spam',
          Adjektiv='absolut schrecklich'))
Dieses Spam ist absolut schrecklich.
```

Positionsabhängige und Schlüsselwortargumente können willkürlich kombiniert werden:

```
>>>print('Die Geschichte von {0}, {1} und {anderer}.'.format('Bill',
          'Manfred', anderer='Georg'))
Die Geschichte von Bill, Manfred und Georg.
```

`'!a'` (wendet **`ascii()`** an), `'!s'` (wendet **`str()`** an) und `'!r'` (wendet **`repr()`** an) können dazu benutzt werden den übergebenen Wert zu konvertieren bevor er formatiert wird:

```
>>> import math
>>> print('Der Wert von PI ist ungefähr {}.'.format(math.pi))
Der Wert von PI ist ungefähr 3.14159265359.
>>> print('Der Wert von PI ist ungefähr {!r}.'.format(math.pi))
Der Wert von PI ist ungefähr 3.141592653589793.
```

Ein optionales `':'` mit Formatspezifizierer (format specifier) können auf den Namen des Feldes folgen. Dies erlaubt einem eine größere Kontrolle darüber, wie der Wert formatiert wird. Das folgende Beispiel rundet Pi auf drei Stellen nach dem Komma.

```
>>> import math
```

```
>>> print('Der Wert von Pi ist ungefähr {0:.3f}.'.format(math.pi))
Der Wert von Pi ist ungefähr 3.142.
```

Übergibt man einen Integer nach dem `':'`, so legt man eine minimale Breite für dieses Feld an. Das ist nützlich um Tabellen schön aussehen zu lassen.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

Hat man einen wirklich langen Formatstring, den man nicht aufteilen will, wäre es nett, wenn man die zu formatierenden Variablen durch den Namen statt durch die Position referenzieren könnte. Dies kann einfach bewerkstelligt werden, indem man das Dictionary übergibt und auf die Schlüssel über eckige Klammern `'[]'` zugreift

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
          'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Das könnte auch genauso erreicht werden, indem man die Tabelle als Schlüsselwortargumente mit der '**'-Notation übergibt.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Das ist besonders nützlich in Verbindung mit der eingebauten Funktion **`vars()`**, die ein Dictionary mit allen lokalen Variablen zurückgibt.

Format String Syntax gibt eine komplette Übersicht zur Zeichenkettenformatierung mit **`format()`**.

7.1.1. Alte Zeichenkettenformatierung

Der `%`-Operator kann auch zur Zeichenkettenformatierung genutzt werden. Er interpretiert das linke Argument genauso wie einen **`sprintf()`**-artigen Formatstring, der auf das rechte Argument angewendet werden soll und gibt die resultierende Zeichenkette dieser Formatierungsoperation zurück. Zum Beispiel:

```
>>> import math
>>> print('Der Wert von Pi ist ungefähr %5.3f.' % math.pi)
Der Wert von Pi ist ungefähr 3.142.
```

Da **`format()`** ziemlich neu ist, benutzt viel Pythoncode noch den `%`-Operator. Jedoch sollte **`format()`** hauptsächlich benutzt werden, da die alte Art der Formatierung irgendwann aus der Sprache entfernt werden wird.

Mehr Informationen dazu gibt es in dem Abschnitt Old String Formatting Operations.

7.2. Lesen und Schreiben von Dateien

`open()` gibt ein Dateiobjekt (file object) zurück und wird meistens mit zwei Argumenten aufgerufen: `open(filename, mode)`

```
>>> f = open('/tmp/workfile', 'w')


>>> print(f)
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Das erste Argument ist eine Zeichenkette, die den Dateinamen enthält. Das zweite Argument ist eine andere Zeichenkette mit ein paar Zeichen, die die Art der Benutzung der Datei beschreibt. mode kann `'r'` sein, wenn die Datei nur gelesen wird, `'w'`, wenn sie nur geschrieben wird (eine existierende Datei mit demselben Namen wird gelöscht) und `'a'` öffnet die Datei zum Anhängen; alle Daten, die in die Datei geschrieben werden, werden automatisch ans Ende angehängt. `'r+'` öffnet die Datei zum Lesen und Schreiben. Das mode-Argument ist optional, fehlt es, so wird `'r'` angenommen.

Normalerweise werden Dateien im Textmodus (text mode) geöffnet, das heisst, dass man Zeichenketten von ihr liest beziehungsweise in sie schreibt, die in einer bestimmten Kodierung kodiert werden (der Standard ist UTF-8). Wird `'b'` an das mode-Argument angehängt, so öffnet man die Datei im Binärmodus (binary mode); in ihm werden Daten als Byteobjekte gelesen und geschrieben. Dieser Modus sollte für alle Dateien genutzt werden, die keinen Text enthalten.

Im Textmodus wird beim Lesen standardmäßig das plattformspezifische Zeilenende (`\n` unter Unixen, `\r\n` unter Windows) zu einem einfachen `\n` konvertiert und beim Schreiben `\n` zurück zum plattformspezifischen Zeilenende. Diese versteckte Modifikation ist klasse für Textdateien, wird aber binäre Dateiformate, wie `JPEG`- oder `EXE`-Dateien, beschädigen. Achte sehr sorgfältig darauf, dass Du den Binärmodus benutzt, wenn Du solche Dateien schreibst oder liest.

7.2.1. Methoden von Dateiobjekten

Die übrigen Beispiele in diesem Abschnitt nehmen an, dass ein Dateiobjekt namens `f` schon erstellt wurde.

Um den Inhalt einer Datei zu lesen, kann man `f.read(size)` aufrufen, was einen Teil der Daten ausliest und diese als Zeichenketten- oder Byteobjekt zurückgibt. size ist ein optionales, numerisches Argument. Wird es ausgelassen oder ist es negativ, so wird der gesamte Inhalt der Datei ausgelesen und zurückgegeben, falls die Datei doppelt so groß wie der Speicher Deiner Maschine ist, so ist das Dein Problem. Andernfalls werden

höchstens size Byte ausgelesen und zurückgegeben. Ist das Ende der Datei erreicht, so gibt `f.read()` eine leere Zeichenkette (`''`) zurück.

```
>>> f.read()
'Das ist die ganze Datei.\n'
>>> f.read()
''
```

`f.readline()` liest eine einzelne Zeile aus einer Datei; ein Zeilenumbruchszeichen (`\n`) bleibt am Ende der Zeichenkette und wird nur ausgelassen, falls die letzte Zeile nicht in einem Zeilenumbruch endet. Dies macht den Rückgabewert eindeutig:
Falls `f.readline()` eine leere Zeichenkette zurückgibt, so ist das Ende der Datei erreicht, während eine Leerzeile durch `'\n'`, eine Zeichenkette, die nur einen einzelnen Zeilenumbruch enthält, dargestellt wird.

```
>>> f.readline()
'Dies ist die erste Zeile der Datei\n'
>>> f.readline()
'Zweite Zeile der Datei\n'
>>> f.readline()
''
```

`f.readlines()` gibt eine Liste zurück die alle Zeilen der Datei enthält. Wird ein optionaler Paramenter sizehint übergeben, liest es mindestens so viele Bytes aus der Datei und zusätzlich noch so viele, dass die nächste Zeile komplett ist und gibt diese Zeilen zurück. Dies wird oft benutzt, um ein effizientes Einlesen der Datei anhand der Zeilen zu ermöglichen, ohne die gesamte Datei in den Speicher laden zu müssen. Nur komplette Zeilen werden zurückgegeben.

```
>>> f.readlines()
['Dies ist die erste Zeile der Datei\n', 'Zweite Zeile der Datei\n']
```

Ein alternativer Ansatz Zeilen auszulesen ist, über das Dateiobjekt zu iterieren. Das ist speichereffizient, schnell und führt zu einfacherem Code:

```
>>> for line in f:
...     print(line, end='')
...
Dies ist die erste Zeile der Datei.
Zweite Zeile der Datei
```

Der alternative Ansatz ist einfacher, bietet aber keine feinkörnige Kontrolle. Da beide Ansätze die Pufferung von Zeilen unterschiedlich handhaben, sollten sie nicht vermischt werden.

`f.write(string)` schreibt den Inhalt von string in die Datei und gibt die Anzahl der Zeichen, die geschrieben wurden, zurück.

```
>>> f.write('Dies ist ein Test\n')
18
```

Um etwas anderes als eine Zeichenkette zu schreiben, muss es erst in eine Zeichenkette konvertiert werden:

```
>>> value = ('Die Antwort', 42)
>>> s = str(value)
>>> f.write(s)
19
```

`f.tell()` gibt eine Ganzzahl zurück, die die aktuelle Position des Dateiobjektes innerhalb der Datei angibt, gemessen in Bytes vom Anfang der Datei. Um die Position des Dateiobjektes zu ändern, gibt es `f.seek(offset, from_what)`. Die Position wird berechnet indem offset zu einem Referenzpunkt addiert wird, dieser wird durch das Argument from_what festgelegt. Bei einem from_what des Wertes 0, wird von Beginn der Datei gemessen, bei 1 von der aktuellen Position, bei 2 vom Ende der Datei. from_what kann ausgelassen werden und hat den Standardwert 0, das den Anfang der Datei als Referenzpunkt benutzt.

```
>>> f = open('/tmp/workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # Gehe zum 6. Byte der Datei
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Gehe zum drittletzten Byte
13
>>> f.read(1)
b'd'
```

In Textdateien (die, die ohne ein `b` im Modus geöffnet werden) sind nur Positionierungen vom Anfang der Datei aus erlaubt (mit der Ausnahme, dass mit `f.seek(0, 2)` zum Ende der Datei gesprungen werden kann).

Wenn man mit einer Datei fertig ist, ruft man `f.close()` auf, um sie zu schließen und jegliche Systemressource freizugeben, die von der offenen Datei belegt wird. Nach dem Aufruf von `f.close()` schlägt automatisch jeder Versuch fehl das Objekt zu benutzen.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Die optimale Vorgehensweise ist es, das Schlüsselwort **`with`** zu benutzen, wenn man mit Dateiobjekten arbeitet. Das hat den Vorteil, dass die Datei richtig geschlossen wird, sobald die Befehle des Blocks abgearbeitet sind, auch wenn innerhalb eine Ausnahme verursacht wird. Das ist auch viel kürzer als einen äquivalenten **`try`** - **`finally`** -Block zu schreiben:

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Dateiobjekte haben noch ein paar zusätzliche Methoden, wie **`isatty()`** und **`truncate()`**, die weniger häufig genutzt werden. Ein komplettes Handbuch zu Dateiobjekten kann in der Bibliotheksreferenz gefunden werden.

7.2.2. Das `pickle`-Modul

Zeichenketten können einfach in eine Datei geschrieben und aus ihr gelesen werden. Zahlen sind ein bisschen aufwändiger, da die **`read()`**-Methode nur Zeichenketten zurückgibt. Diese müssen an eine Funktion wie **`int()`** übergeben werden, die eine Zeichenkette wie `'123'` nimmt und deren numerischen Wert 123 zurückgibt. Wenn man jedoch komplexere Datentypen wie Listen, Dictionaries oder Klasseninstanzen speichern will, wird die Angelegenheit viel komplizierter.

Anstatt die Benutzer ständig Code schreiben und debuggen zu lassen, um komplexere Datentypen zu speichern, stellt Python ein Standardmodul namens **`pickle`** bereit. Dies ist ein fantastisches Modul, das fast jedes Pythonobjekt (sogar ein paar Formen von Pythoncode!) nehmen kann und es in eine Zeichenkettenrepräsentation konvertieren kann; dieser Prozess wird pickling ("einwecken") genannt. Das Objekt aus der Zeichenkettenrepräsentation zu rekonstruieren wird unpickling genannt. Zwischen pickling und unpickling, kann die Zeichenkettenrepräsentation in Daten oder Dateien gespeichert werden oder über ein Netzwerk an eine entfernte Maschine geschickt werden.

Hat man ein Objekt `x` und ein Dateiobjekt `f`, das zum Schreiben geöffnet wurde, benötigt der einfachste Weg das Objekt zu picklen nur eine Zeile Code:

```
pickle.dump(x, f)
```

Um das Objekt wieder zu unpicklen reicht, wenn `f` ein Dateiobjekt ist, das zum Lesen geöffnet wurde:

```
x = pickle.load(f)
```

(Es gibt auch andere Varianten, die benutzt werden, wenn man viele Objekte pickled oder falls man gepicklete Daten nicht in einer Datei speichern will; siehe **`pickle`** in der Python Bibliotheksreferenz.)

`pickle` ist der normale Weg ein Pythonobjekt zu erzeugen, das gespeichert und von anderen Programmen oder demselben Programm wiederbenutzt werden kann; der Fachbegriff für so etwas ist ein persistentes Objekt. Weil **`pickle`** so weitläufig benutzt wird, stellen viele Programmierer, die Pythonerweiterungen schreiben sicher, dass neue Datentypen, wie Matrizen, richtig gepickled und unpickled werden können.

Next Previous