mitp

Brett Slatkin

Effektiv Python programmieren

59 Wege für bessere Python-Programme

- Private Instanzattribute sollten mit `__doppelten_führenden_unterstrichen` geschrieben werden.
- Klassen und Exceptions sollten als `GroßgeschriebeneWörter` geschrieben werden.
- Modul-weite Konstanten sollten `IN_GROSSBUCHSTABEN` geschrieben werden.
- Instanzmethoden einer Klasse sollten als Bezeichnung des ersten Parameters (der auf das Objekt verweist) `self` verwenden.
- Klassenmethoden sollten als Bezeichnung des ersten Parameters (der auf die Klasse verweist) `cls` verwenden.

- **Ausdrücke und Anweisungen:** In *The Zen of Python* heißt es: »Es sollte einen – und vorzugsweise *nur* einen – offensichtlichen Weg geben, ein Problem zu lösen.« PEP 8 schreibt daher für Ausdrücke und Anweisungen Folgendes fest:
  - Verwenden Sie Negierungen möglichst inmitten eines Ausdrucks (`if a is not b`) statt positive Ausdrücke zu negieren (`if not a is b`).
  - Testen Sie leere Werte wie `[]` oder `''` nicht, indem Sie deren Länge überprüfen (`if len(eineliste) == 0`). Verwenden Sie stattdessen `if not somelist`. Leere Werte werden implizit als `False` bewertet.
  - Gleiches gilt für nicht leere Werte wie `[1]` oder `'hallo'`. Die Anweisung `if somelist` wird für nicht leere Werte implizit als `True` bewertet.
  - Verzichten Sie auf einzeilige `if`-Anweisungen, `for`- bzw. `while`-Schleifen und zusammengesetzte `except`-Anweisungen. Verteilen Sie sie der Übersichtlichkeit halber auf mehrere Zeilen.
  - Platzieren Sie `import`-Anweisungen immer am Anfang der Datei.
  - Verwenden Sie beim Import von Modulen stets absolute Pfadbezeichnungen, nicht die zum aktuellen Modul relativen. Wenn Sie beispielsweise das Modul `foo` des Pakets `bar` importieren, sollten Sie nicht `import foo`, sondern `from bar import foo` verwenden.
  - Verwenden Sie die ausdrückliche Syntax `from . import foo`, falls Sie auf relative Pfade angewiesen sind.
  - Importe sollten in folgender Reihenfolge stattfinden: Zuerst Module der Standardbibliothek, dann Module Dritter, dann die eigenen Module. Die jeweilige Gruppe sollte alphabetisch sortiert sein.

Hinweis

Pylint (`http://www.pylint.org`) ist ein beliebtes Werkzeug zur statischen Analyse von Python-Quelltexten. Pylint erzwingt automatisch die Einhaltung der Stilregeln gemäß PEP 8 und kann viele weitere typische Fehler in Python-Programmen entdecken.

1.2.1 Kompakt

- Halten Sie sich beim Schreiben von Python-Code an die Stilregeln gemäß PEP 8.
- Ein einheitlicher Stil erleichtert die Zusammenarbeit mit anderen Programmierern der Python-Community.
- Außerdem erleichtert ein einheitlicher Stil nachträgliche Änderungen des eigenen Codes.

1.3 Punkt 3: Unterschiede zwischen bytes, str und unicode

In Python 3 gibt es zwei Datentypen, die Zeichenfolgen repräsentieren: `bytes` und `str`. `bytes`-Instanzen enthalten reine 8-Bit-Werte, in `str`-Instanzen hingegen sind Unicode-Zeichen gespeichert.

In Python 2 gibt es ebenfalls zwei Datentypen, die Zeichenfolgen repräsentieren: `str` und `unicode`. Im Gegensatz zu Python 3 enthalten `str`-Instanzen reine 8-Bit-Werte und Unicode-Zeichen werden in `unicode`-Instanzen gespeichert.

Es gibt viele verschiedene Möglichkeiten, Unicode-Zeichen als Binärdaten (reine 8-Bit-Werte) zu repräsentieren. Die am häufigsten verwendete Textkodierung ist UTF-8. Den `str`-Instanzen in Python 3 und den `unicode`-Instanzen in Python 2 ist jedoch keine Binärkodierung zugeordnet. Um Unicode-Zeichen als Binärdaten zu speichern, müssen Sie die `encode`-Methode verwenden. Zur Umwandlung der Binärdaten in Unicode-Zeichen dient die Methode `decode`.

Beim Schreiben Ihres Python-Programms sollten Sie darauf achten, die Kodierung und Dekodierung von Unicode an den äußersten Schnittstellen vorzunehmen. Ihr eigentliches Programm sollte die Unicode-Datentypen (`str` in Python 3 bzw. `unicode` in Python 2) verwenden und keinerlei Annahmen über die Zeichenkodierung machen. Auf diese Weise können Sie auch alternative Textkodierungen (wie Latin-1, Shift JIS oder Big5) einlesen, gleichzeitig aber bei der Ausgabe strikt auf eine bestimmte Textkodierung setzen (idealerweise UTF-8).

Das Vorhandensein der beiden unterschiedlichen Zeichendatentypen führt zu zwei typischen Situationen:

- Sie möchten reine 8-Bit-Werte verarbeiten, die als UTF-8-Zeichen kodiert sind (oder eine andere Textkodierung aufweisen).
- Sie möchten Unicode-Zeichen verarbeiten, denen keine Textkodierung zugeordnet ist.

Sie benötigen dann zwei Hilfsfunktionen zur Konvertierung der verschiedenen Zeichentypen, die gewährleisten, dass die Eingabewerte den Erwartungen Ihres Codes entsprechen.

In Python 3 ist eine Methode erforderlich, die eine `str`- oder eine `bytes`-Instanz entgegennimmt und stets eine `str`-Instanz zurückliefert.

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value # str-Instanz
```

Sie benötigen eine weitere Methode, die eine `str`- oder eine `bytes`-Instanz entgegennimmt und stets eine `bytes`-Instanz zurückgibt.

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value # bytes-Instanz
```

In Python 2 muss die Methode eine `str`- oder eine `unicode`-Instanz entgegennehmen und stets eine `unicode`-Instanz zurückliefern.

```
# Python 2
def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value # unicode-Instanz
```

Und schließlich fehlt eine letzte Methode, die eine `str`- oder eine `unicode`-Instanz entgegennimmt und stets eine `str`-Instanz zurückgibt.

```
# Python 2
def to_str(unicode_or_str):
    if isinstance(unicode_or_str, unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    return value # str-Instanz
```

Es gibt bei der Verwendung von reinen 8-Bit-Werten und Unicode-Zeichen in Python zwei große Vorbehalte.

Zum einen scheinen in Python 2 `unicode`- und `str`-Instanzen vom selben Typ zu sein, sofern die `str`-Instanz nur 7-Bit-ASCII-Zeichen enthält.

- Eine solche `str`-Instanz kann mit der `unicode`-Instanz durch den »+«-Operator verknüpft werden.
- Eine solche `str`-Instanz und die `unicode`-Instanz können mittels der Operatoren zum Test auf Gleichheit und Ungleichheit miteinander verglichen werden.
- Sie können `unicode`-Instanzen für Formatstrings wie `'%s'` verwenden.

Dieses Verhalten bedeutet, dass Sie einer Funktion, die eine `str`- oder `unicode`-Instanz erwartet, oftmals beide Datentypen übergeben können und alles problemlos funktioniert (sofern es sich nur um 7-Bit-ASCII-Zeichen handelt). In Python 3 hingegen sind `bytes`- und `str`-Instanzen niemals äquivalent – nicht mal leere Strings. Sie müssen daher beim Herumreichen von Zeichenfolgen viel genauer auf den Datentyp achten.

Der zweite Vorbehalt ergibt sich dadurch, dass in Python 3 Operationen mit Datei-Handles (die von der integrierten `open`-Funktion zurückgegeben werden) standardmäßig von einer UTF-8-Kodierung ausgehen, während Python 2 eine Binärkodierung annimmt. Das kann zu überraschenden Fehlschlägen führen, insbesondere dann, wenn die Programmierer an Python 2 gewohnt sind.

Wenn Sie beispielsweise einige zufällige Werte als Binärdaten in eine Datei schreiben möchten, funktioniert der nachstehende Code in Python 2, schlägt in Python 3 jedoch fehl:

```
with open('/tmp/random.bin', 'w') as f:
    f.write(os.urandom(10))
>>>
TypeError: must be str, not bytes
```

Der Grund für diesen Fehler ist das in Python 3 beim `open`-Befehl neu hinzugekommene Argument `encoding`, das standardmäßig den Wert `'utf-8'` besitzt. Die `read`- und `write`-Operationen des Datei-Handles erwarten daher `str`-Instanzen, die Unicode-Zeichen enthalten, nicht `bytes`-Instanzen, in denen Binärdaten gespeichert sind.

Damit das Ganze wieder funktioniert, müssen Sie angeben, dass die zu lesende Datei im binären Schreibmodus (write binary, `'wb'`) statt im zeichenweisen Schreibmodus (`'w'`) geöffnet werden soll. Hier verwende ich `open` so, dass der Aufruf sowohl in Python 2 als auch in Python 3 korrekt funktioniert:

```
with open('/tmp/random.bin', 'wb') as f:
    f.write(os.urandom(10))
```

Dieses Problem tritt beim Lesen von Dateien ebenfalls auf. Die Lösung ist ebenfalls dieselbe: Geben Sie beim Öffnen der Datei statt `'r'` mittels `'rb'` an, dass die Datei im binären Modus geöffnet werden soll.

1.3.1 Kompakt

- In Python 3 enthalten `bytes`-Instanzen reine 8-Bit-Werte und `str`-Instanzen Unicode-Zeichenfolgen. `bytes`- und `str`-Instanzen können *nicht* durch Operatoren wie > oder + miteinander verknüpft werden.
- In Python 2 enthalten `str`-Instanzen reine 8-Bit-Werte und `unicode`-Instanzen Unicode-Zeichenfolgen. `str`- und `unicode`-Instanzen *können* durch Operatoren miteinander verknüpft werden, sofern die `str`-Instanz ausschließlich 7-Bit-ASCII-Zeichen enthält.
- Verwenden Sie Hilfsfunktionen, um zu gewährleisten, dass die zu verarbeitenden Eingaben im erwarteten Format (8-Bit-Werte, UTF-8-kodierte Zeichen, Unicode-Zeichen usw.) vorliegen.
- Beim Lesen oder Schreiben von Binärdaten müssen Sie die Datei im binären Modus (wie z.B. `'rb'` oder `'wb'`) öffnen.

1.4 Punkt 4: Hilfsfunktionen statt komplizierter Ausdrücke

Pythons prägnante Syntax ermöglicht es, in einzeiligen Ausdrücken eine Menge Programmlogik unterzubringen. Nehmen Sie beispielsweise an, Sie möchten den Abfrage-String einer URL dekodieren. Hier sind sämtliche Parameter Ganzzahlen:

```
from urllib.parse import parse_qs
values = parse_qs('red=5&blue=0&green=',
                  keep_blank_values=True)
print(repr(values))
>>>
{'red': ['5'], 'green': [''], 'blue': ['0']}
```

Manche Parameter könnten mehrere Werte besitzen, einige auch nur einen, weitere sind zwar vorhanden, aber leer, und wieder andere fehlen womöglich ganz. Die `get`-Methode des Dictionarys kann also verschiedene Werte zurückliefern:

```
print('Rot:       ', values.get('red'))
print('Grün:      ', values.get('green'))
```