Dan Bader

Python Tricks

Praktische Tipps
für Fortgeschrittene

dpunkt.verlag

Das Komma hinter `Dilbert` macht es möglich, neue Elemente hinzuzufügen und zu entfernen, ohne die Kommasetzung ändern zu müssen. Alle Zeilen sind einheitlich, die Differenzanzeige ist übersichtlicher, und diejenigen, die deinen Code überprüfen, werden sehr zufrieden sein. Manchmal sind es eben die kleinen Dinge, die große Wirkung entfalten.

Kernpunkte

- Durch geschickte Formatierung und Kommasetzung kannst du Listen, Dictionarys und Mengen wartungsfreundlicher gestalten.
- Die Stringliteralverkettung von Python kannst du zu deinem Vorteil nutzen, aber sie kann auch zu schwer zu findenden Bugs führen.

2.3 Kontextmanager und die Anweisung with

Manche sehen die Python-Anweisung `with` als eher obskur an. Wenn du aber einen Blick hinter die Kulissen wirfst, wirst du feststellen, dass sie tatsächlich sehr hilfreich ist, um saubereren und besser lesbaren Python-Code zu schreiben. Wozu ist die Anweisung `with` also gut? Sie ermöglicht es, einige gängige Muster der Ressourcenverwaltung zu vereinfachen, indem sie deren Funktion verbirgt und es erlaubt, sie wiederzuverwenden. Ein gutes Beispiel für die wirkungsvolle Anwendung dieser Anweisung findest du in der Standardbibliothek von Python, und zwar in der integrierten Funktion `open()`:

```
with open('hello.txt', 'w') as f:
    f.write('hello, world!')
```

Es wird gewöhnlich empfohlen, Dateien mit `with` zu öffnen, da dadurch garantiert wird, dass offene Dateideskriptoren automatisch geschlossen werden, wenn die Programmausführung den Kontext der `with`-Anweisung verlässt. Intern wird der vorstehende Beispielcode in etwas wie das Folgende umgewandelt:

```
f = open('hello.txt', 'w')
try:
    f.write('hello, world')
finally:
    f.close()
```

Dieser Code ist deutlich umfangreicher. Von besonderer Bedeutung ist hier die `try...finally`-Anweisung. Es würde nicht ausreichen, einfach Folgendes zu schreiben:

```
f = open('hello.txt', 'w')
f.write('hello, world')
f.close()
```

Bei dieser Implementierung ist nicht garantiert, dass die Datei geschlossen wird, wenn während des Aufrufs von `f.write()` eine Ausnahme auftritt. Das Programm kann dann in den Dateideskriptor überlaufen. Aus diesem Grund ist die Anweisung `with` so praktisch: Damit wird es ganz einfach, Ressourcen ordnungsgemäß zu erwerben und freizugeben.

Ein weiteres gutes Beispiel für den wirkungsvollen Einsatz der Anweisung `with` in der Python-Standardbibliothek ist die Klasse `threading.Lock`:

```
some_lock = threading.Lock()

# Gefährlich:
some_lock.acquire()
try:
    # Macht irgendetwas ...
finally:
    some_lock.release()

# Besser:
with some_lock:
    # Macht irgendetwas ...
```

In beiden Fällen kapselt die Anweisung `with` den Großteil der Logik zur Handhabung der Ressource. Du musst nicht jedes Mal ausdrückliche `try...finally`-Anweisungen schreiben, da sich `with` darum kümmert. Die Anweisung `with` kann Code für den Umgang mit Systemressourcen besser lesbar machen. Sie hilft auch, Bugs und Lecks zu vermeiden, da sie es praktisch unmöglich macht, das Aufräumen oder die Freigabe von Ressourcen zu vergessen, die nicht mehr benötigt werden.

with in eigenen Objekten unterstützen

Dass du die Funktion `open()` und die Klasse `threading.Lock` zusammen mit `with` einsetzen kannst, ist keine besondere Sache. Durch die Implementierung des *Kontextmanagers*[6] kannst du das Gleiche auch bei deinen eigenen Klassen und Funktionen erreichen.

Was ist ein Kontextmanager? Es handelt sich dabei um ein einfaches »Protokoll« (eine Schnittstelle), dem deine Objekte entsprechen müssen, um die Anweisung `with` nutzen zu können. Du musst ihm dazu im Grunde genommen nur die Methoden `__enter__` und `__exit__` hinzufügen. Python ruft diese beiden Methoden während des Ressourcenverwaltungszyklus jeweils zum richtigen Zeitpunkt auf. Sehen wir uns an, wie das in der Praxis abläuft. Eine einfache Implementierung des `open()`-Kontextmanagers sieht wie folgt aus:

```
class ManagedFile:
    def __init__(self, name):
        self.name = name
```

6 Siehe Python-Dokumentation, »With Statement Context Managers«

```
    def __enter__(self):
        self.file = open(self.name, 'w')
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
```

Die Klasse `ManagedFile` erfüllt das Kontextmanagerprotokoll und unterstützt ebenso wie die ursprüngliche Funktion `open()` die Anweisung `with`:

```
>>> with ManagedFile('hello.txt') as f:
...     f.write('hello, world!')
...     f.write('bye now')
```

Wenn die Ausführung in den Kontext der `with`-Anweisung eintritt und es an der Zeit ist, die Ressource zu sammeln, ruft Python `__enter__` auf. Verlässt die Ausführung den Kontext wieder, wird dagegen `__exit__` aufgerufen, um die Ressource freizugeben.

Du kannst mit der Anweisung `with` nicht nur dadurch arbeiten, dass du einen Kontextmanager auf der Grundlage einer Klasse schreibst. Das Modul `contextlib`[7] aus der Standardbibliothek bietet noch weitere Abstraktionen, die auf dem grundlegenden Kontextmanagerprotokoll aufbauen. Wenn die Möglichkeiten von `contextlib` für deinen Anwendungsfall geeignet sind, kannst du dir damit die Arbeit ein wenig erleichtern.

Beispielsweise kannst du den Dekorator `contextlib.contextmanager` verwenden, um eine generatorgestützte *Factory-Funktion* für eine Ressource zu definieren, die dann automatisch `with` unterstützt. Wenn wir diese Technik anwenden, sieht unser Kontextmanager `ManagedFile` wie folgt aus:

```
from contextlib import contextmanager

@contextmanager
def managed_file(name):
    try:
        f = open(name, 'w')
        yield f
    finally:
        f.close()

>>> with managed_file('hello.txt') as f:
...     f.write('hello, world!')
...     f.write('bye now')
```

Bei `managed_file()` handelt es sich um einen Generator, der zunächst die Ressource anfragt. Anschließend aber hält er seine eigene Ausführung an und tritt die Ressource ab, sodass der Aufrufer sie verwenden kann. Wenn der Aufrufer den Kon-

7 Siehe *https://docs.python.org/3/library/contextlib.html*

text von `with` verlässt, nimmt der Generator die Ausführung wieder auf, sodass er alle noch verbleibenden Aufräumvorgänge ausführen und die Ressource wieder freigeben kann.

Die Klassen- und die Generatorimplementierung des Kontextmanagers sind praktisch gleichwertig. Vielleicht ist die eine Variante für dich besser lesbar als die andere, aber das ist Geschmackssache.

Ein Nachteil der Implementierung mit `@contextmanager` besteht darin, dass dazu Kenntnisse anspruchsvollerer Python-Konzepte wie Dekorator und Generatoren erforderlich sind. Wenn du dein Wissen zu diesen Konzepten noch einmal vertiefen musst, schlage ruhig in den entsprechenden Kapiteln dieses Buches nach. Was die richtige Implementierung ist, hängt jedoch davon ab, womit du und dein Team am besten zurechtkommen und was du für übersichtlicher hältst.

APIs mit Kontextmanagern schreiben

Kontextmanager sind sehr flexibel. Mit dem kreativen Einsatz der Anweisung `with` kannst du bequem APIs für deine Module und Klassen schreiben. Nehmen wir beispielsweise an, bei der »Ressource«, die wir verwalten wollen, handelt es sich um die Einrückungsebenen in einem Berichtsgenerator. Stell dir vor, wir würden dazu Code wie den folgenden schreiben:

```
with Indenter() as indent:
    indent.print('hi!')
    with indent:
        indent.print('hello')
        with indent:
            indent.print('bonjour')
    indent.print('hey')
```

Das liest sich fast wie eine fachbereichsspezifische Sprache (Domain-Specific Language, DSL) für die Einrückung von Text. Beachte auch, dass dieser Code mehrmals in denselben Kontextmanager eintritt und ihn wieder verlässt, um die Einrückungsebenen zu ändern. Die Ausführung dieses Codeausschnitts führt dazu, dass der folgende sauber formatierte Text an der Konsole ausgegeben wird:

```
hi!
    hello
        bonjour
hey
```

Wie kannst du einen Kontextmanager implementieren, der so etwas ermöglicht? Das ist übrigens eine hervorragende Übung, um die Funktionsweise von Kontextmanagern genau kennenzulernen. Bevor du dir meine nachfolgende Implementierung ansiehst, solltest du dir daher etwas Zeit nehmen, um dich an einer eigenen Implementierung zu versuchen.

Hast du das gemacht? Dann siehst du hier eine mögliche Implementierung mithilfe eines klassenbasierten Kontextmanagers:

```
class Indenter:
    def __init__(self):
        self.level = 0

    def __enter__(self):
        self.level += 1
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.level -= 1

    def print(self, text):
        print('    ' * self.level + text)
```

Das war gar nicht so schwer, oder? Ich hoffe, dass es dir jetzt schon leichterfällt, Kontextmanager und die Anweisung `with` in deinen eigenen Python-Programmen einzusetzen. Das ist eine hervorragende Möglichkeit, um die Ressourcenverwaltung auf eine viel phytonischerer und wartungsfreundlichere Weise zu erledigen.

Wenn du gern noch weitere Übungen durchführen möchtest, um deine Kenntnisse zu vertiefen, versuche einen Kontextmanager zu implementieren, der die Ausführungszeit eines Codeblocks mit der Funktion `time.time` misst. Schreibe dabei sowohl eine Dekorator- als auch eine Klassenvariante, um die Unterschiede dazwischen genau kennenzulernen.

Kernpunkte

- Die Anweisung `with` vereinfacht die Ausnahmebehandlung, da sie die reguläre Verwendung von `try/finally`-Anweisungen in sogenannten Kontextmanagern kapselt.
- Am häufigsten wird `with` für die sichere Verwaltung von Systemressourcen verwendet. Die Ressource wird durch `with` angefragt, und die Freigabe erfolgt, wenn die Ausführung den Kontext von `with` verlässt.
- Die korrekte Verwendung von `with` kann dabei helfen, Ressourcenlecks zu vermeiden und deinen Code lesbarer zu gestalten.

2.4 Einfache und doppelte Unterstriche

Einfache und doppelte Unterstriche in Variablen- und Methodennamen haben eine besondere Bedeutung. Einige dieser Bedeutungen sind lediglich Konventionen und dienen als Hinweise für Programmierer, während andere tatsächlich vom Python-Interpreter genutzt werden.

In diesem Abschnitt besprechen wir fünf Varianten von Unterstrichen in Namen und ihre Auswirkungen auf das Verhalten deiner Python-Programme:

- Führender einfacher Unterstrich: `_var`
- Angehängter einfacher Unterstrich: `var_`