



**Python 3 - Intensivkurs: Projekte erfolgreich realisieren**

350 Seiten

ISBN: 978-3-642-04376-5

49,95 Euro

<http://www.python3-intensivkurs.de>

# Kapitel 6

## Reguläre Ausdrücke

### 6.1 Los geht's

Jede moderne Programmiersprache besitzt integrierte Funktionen zur Bearbeitung von Strings. In Python besitzen Strings Methoden zum Suchen und Ersetzen: `index()`, `find()`, `split()`, `count()`, `replace()` usw. Diese Methoden sind jedoch auf die simpelsten aller Fälle beschränkt. Die `index()`-Methode z. B. sucht nach einem einzelnen fest einprogrammierten Unterstring. Die Methode unterscheidet dabei immer zwischen Groß- und Kleinschreibung. Um einen String `s` ohne Beachtung der Groß-/Kleinschreibung zu suchen, müssen Sie `s.lower()` oder `s.upper()` aufrufen, um dafür zu sorgen, dass alle Buchstaben klein bzw. groß geschrieben sind. Denselben Beschränkungen unterliegen auch die `replace()`- und die `split()`-Methode.

Kann Ihr Ziel mithilfe dieser Stringmethoden erreicht werden, so sollten Sie sie nutzen. Sie sind schnell, einfach und leicht zu lesen, und ich könnte Ihnen einiges über schnellen, einfachen, leicht lesbaren Code erzählen. Wenn Sie jedoch innerhalb von `if`-Anweisungen sehr viele Stringfunktionen nutzen, um Spezialfälle zu behandeln, oder `split()` und `join()` verketteten, um Ihre Strings zu zerstückeln, dann sollten Sie vielleicht doch besser reguläre Ausdrücke in Betracht ziehen.

Reguläre Ausdrücke sind eine mächtige und (weitgehend) standardisierte Möglichkeit, Text mithilfe komplexer Zeichenmuster zu durchsuchen, zu ersetzen und zu parsen. Auch wenn die Syntax der regulären Ausdrücke sehr unübersichtlich ist und nicht wie gewöhnlicher Code aussieht, kann das Ergebnis lesbarer sein, als eine handgemachte Lösung, die eine Menge Stringfunktionen einsetzt. Es besteht sogar die Möglichkeit, innerhalb regulärer Ausdrücke Kommentare zu nutzen. So können Sie Ihre regulären Ausdrücke mit einer erklärenden Dokumentation versehen.

☞ Haben Sie bereits in anderen Sprachen (wie Perl 5) reguläre Ausdrücke eingesetzt, so wird Ihnen die Python-Syntax sehr bekannt vorkommen.

## 6.2 Fallbeispiel: Adresse

Zu diesen Beispielen inspirierte mich ein Alltagsproblem, das ich vor einigen Jahren während meiner Arbeit hatte. Ich musste Adressen bereinigen und vereinheitlichen, um sie von einem alten System in ein neueres zu übertragen. (Sehen Sie, ich erfinde dieses Zeug nicht einfach, es ist sogar nützlich.) Das folgende Beispiel zeigt, wie ich an das Problem herangegangen bin.

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.') ①
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.') ②
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') ③
'100 NORTH BROAD RD.'
>>> import re ④
>>> re.sub('ROAD$', 'RD.', s) ⑤
'100 NORTH BROAD RD.'
```

① Mein Ziel ist es, die Adresse so zu vereinheitlichen, dass 'ROAD' immer mit 'RD.' abgekürzt wird. Auf den ersten Blick nahm ich an, dass ich dies recht einfach mit der Stringmethode `replace()` erreichen könnte. Alle Daten lagen bereits in Großschreibung vor, das sollte also nicht das Problem sein. Der Suchstring, 'ROAD', war außerdem eine Konstante. Und tatsächlich, in diesem einfachen Fall funktioniert `s.replace()`.

② Doch das Leben ist leider voller Gegenbeispiele und so fand ich sehr schnell dieses. Das Problem besteht hier darin, dass 'ROAD' zweimal innerhalb der Adresse auftritt, einmal als Teil des Straßennamens 'BROAD' und einmal als eigenes Wort. Die `replace()`-Methode ersetzt nun beide Vorkommen, während ich zu sehe, wie meine Adresse zerstört wird.

③ Um das Problem mehrerer Vorkommen von 'ROAD' innerhalb von Adressen zu lösen, könnten Sie auf etwas dieser Art zurückgreifen: Suche und ersetze 'ROAD' nur dann, wenn es innerhalb der letzten vier Zeichen der Adresse (`s[-4:]`) vorkommt und kümmere dich nicht um den Rest des Strings (`s[:-4]`). Doch Sie sehen sicher bereits, dass dies jetzt schon ziemlich unflexibel wird. Das Muster ist z. B. abhängig von der Länge des Strings, den es zu ersetzen gilt. (Wollten Sie 'STREET' durch 'ST.' ersetzen, müssten Sie `s[:-6]` und `s[-6:].replace(...)` benutzen.) Würden Sie sich dies gerne in sechs Monaten noch einmal ansehen und Fehler beseitigen? Ich jedenfalls nicht.

④ Es ist an der Zeit, reguläre Ausdrücke einzusetzen. In Python befindet sich die gesamte Funktionalität der regulären Ausdrücke im Modul `re`.

⑤ Sehen Sie sich den ersten Parameter an: 'ROAD\$'. Dies ist ein einfacher regulärer Ausdruck, der 'ROAD' nur dann entspricht, wenn es am Ende eines Strings

auftaucht. Das `$` bedeutet „Ende des Strings“. (Das Gegenstück dazu ist das Zeichen `^`, das „Anfang des Strings“ bedeutet.) Durch Verwendung der `re.sub()`-Funktion durchsuchen Sie den String `s` nach dem regulären Ausdruck `'ROAD$'` und ersetzen ihn mit `'RD.'`. Damit wird das `'ROAD'` am Ende des Strings `s` ersetzt, jedoch nicht das `'ROAD'`, das Teil des Wortes `'BROAD'` ist, denn dieses befindet sich ja in der Mitte von `s`.

Weiterhin mit der Bereinigung der Adressen beschäftigt, entdeckte ich, dass das vorherige Beispiel nicht gut genug war. Nicht alle Adressen enthalten eine Straßenbezeichnung. Einige Adressen enden einfach mit dem Straßennamen. Die meiste Zeit über klappte meine Methode, doch wenn der Straßename `'BROAD'` wäre, würde der reguläre Ausdruck `'ROAD'` als Teil von `'BROAD'` am Ende des Strings finden. Das ist nicht das was ich wollte.

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s) ①
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s) ②
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s) ③
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) ④
'100 BROAD RD. APT 3'
```

① Eigentlich wollte ich, dass `'ROAD'` nur dann erkannt wird, wenn es am Ende des Strings steht und ein eigenes Wort ist (und nicht Teil eines längeren Wortes). Um dies als regulären Ausdruck darzustellen, verwendet man `\b`, was so viel bedeutet wie „genau hier muss sich eine Wortgrenze befinden“. In Python ist dies etwas kompliziert, da dem `\`-Zeichen innerhalb eines Strings zuerst die Sonderbedeutung genommen werden muss. Dieser Umstand wird manchmal als *Backslash-Plage* bezeichnet und ist einer der Gründe dafür, dass reguläre Ausdrücke in Perl einfacher zu verwenden sind als in Python. Andererseits vermischt Perl normale Syntax und reguläre Ausdrücke, was dazu führt, dass Bugs schwer zu finden sein können, da man nicht weiß, ob sich der Fehler in der Syntax oder im regulären Ausdruck befindet.

② Um die *Backslash-Plage* zu umgehen, können Sie einen sogenannten *Raw-String* verwenden, indem Sie vor den String den Buchstaben `r` setzen. Dies teilt Python mit, dass er bei diesem String nichts *escapen* (Sonderbedeutung des `\`-Zeichens) soll. `'\t'` ist das Zeichen für einen Tabulator, doch `r'\t'` ist ein Backslash gefolgt vom Buchstaben `t`. Ich empfehle Ihnen, immer *Raw-Strings* zu verwenden, wenn Sie mit regulären Ausdrücken arbeiten; andernfalls werden sie schnell zu verwirrend (und reguläre Ausdrücke sind von Haus aus schon verwirrend genug).

③ *\*Seufz\** Leider fand ich recht bald weitere Fälle, die meiner Logik widersprachen. In einem der Fälle enthielt die Adresse zwar 'ROAD' als ein eigenes Wort, doch es befand sich nicht am Ende, da nach der Straße noch eine Apartmentnummer folgte. Da sich 'ROAD' aber eben nicht am Ende des Strings befindet, stimmt es nicht mit dem Suchmuster überein und wird nicht gefunden. `re.sub()` ersetzt also rein gar nichts und Sie erhalten den ursprünglichen String, was sie nicht wollen.

④ Um dieses Problem zu lösen entfernte ich das `$`-Zeichen und fügte ein weiteres `\b` hinzu. So bedeutet der reguläre Ausdruck nun „finde 'ROAD', wenn es ein ganzes eigenes Wort irgendwo innerhalb des Strings ist“, also entweder am Ende, am Anfang, oder irgendwo dazwischen.

### 6.3 Fallbeispiel: römische Zahlen

Sie haben sicher schon römische Zahlen gesehen, auch wenn Sie sie gar nicht bemerkt haben. Sie könnten sie beim Copyright-Vermerk alter Filme oder Fernsehserien gesehen haben („Copyright MCMXLVI“ anstatt „Copyright 1946“), oder bei Bibliotheken oder Universitäten („errichtet MDCCCLXXXVIII“ anstatt „errichtet 1888“). Sie könnten sie auch auf Grundrissen oder bei bibliografischen Angaben gesehen haben. Das System hinter dieser Zahldarstellung stammt aus dem antiken Römischen Reich (daher der Name).

Es gibt insgesamt sieben Zeichen, mit denen durch Wiederholung und Kombination Zahlen dargestellt werden können.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

Im Folgenden einige allgemeine Regeln zum Bilden römischer Zahlen:

- Die Zeichen werden zusammengezählt. I ist 1, II ist 2 und III ist 3. VI ist 6 („5 und 1“), VII ist 7 und VIII ist 8.
- Die Zehner-Zeichen (I, X, C und M) können bis zu drei Mal wiederholt werden. Bei 4 müssen Sie vom nächsthöheren Fünfer-Zeichen subtrahieren. Sie können 4 nicht als IIII darstellen, sondern nur als IV („1 weniger als 5“). Die Zahl 40 wird als XL geschrieben („10 weniger als 50“), 41 als XLI, 42 als XLII, 43 als XLIII und schließlich 44 als XLIV („10 weniger als 50, und dann 1 weniger als 5“).
- Genauso müssen Sie auch bei 9 verfahren. Hier müssen Sie vom nächsthöheren Zehner-Zeichen subtrahieren: 8 ist VIII, doch 9 ist IX („1 weniger als 10“), nicht

VIII (da das Zeichen I nicht vier Mal wiederholt werden darf). Die Zahl 90 wird somit als XC dargestellt und 900 als CM.

- Die Fünfer-Zeichen dürfen niemals wiederholt werden. Die Zahl 10 wird immer als X dargestellt, niemals als VV. Die Zahl 100 ist immer C, niemals LL.
- Römische Zahlen werden von hoch nach niedrig geschrieben und von links nach rechts gelesen. Die Reihenfolge der Zeichen spielt daher eine sehr wichtige Rolle. DC ist 600; CD ist eine völlig andere Zahl (400, „100 weniger als 500“). CI ist 101; IC ist nicht einmal eine gültige römische Zahl (man kann 1 nicht direkt von 100 subtrahieren, sondern müsste XCIX schreiben, „10 weniger als 100, und dann 1 weniger als 10“).

### 6.3.1 Prüfen der Tausender

Was benötigt man, um herauszufinden, ob ein beliebiger String eine gültige römische Zahl ist? Lassen Sie uns eine Ziffer nach der anderen ansehen. Da römische Zahlen immer mit der höchsten Ziffer beginnen, sollten wir es auch so machen. Wir fangen also mit den Tausendern an. Für Zahlen ab 1.000 werden die Tausender durch eine Reihe von M-Zeichen dargestellt.

```
>>> import re
>>> pattern = '^M?M?M?M?$'           ①
>>> re.search(pattern, 'M')          ②
<_sre.SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')        ③
<_sre.SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')       ④
<_sre.SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')     ⑤
>>> re.search(pattern, '')          ⑥
<_sre.SRE_Match object at 0106F4A8>
```

① Dieses Suchmuster besteht aus drei Teilen. Das `^`-Zeichen sorgt dafür, dass der folgende Ausdruck nur am Anfang eines Strings gefunden wird. Wäre dies nicht angegeben, so würde das Muster jedes M finden, egal an welcher Stelle es sich befindet. Das wollen Sie nicht. Sie wollen sicherstellen, dass das M-Zeichen nur gefunden wird, wenn es am Anfang eines Strings steht. `M?` sucht ein optionales M-Zeichen. Da dies dreimal wiederholt wird, findet das Muster 0 bis 3 M-Zeichen hintereinander. Das `$`-Zeichen steht für das Ende des Strings. In Kombination mit dem `^`-Zeichen am Anfang sorgt dies dafür, dass nur Strings erkannt werden, die einzig und allein aus M-Zeichen bestehen.

② Das Herzstück des `re`-Moduls ist die `search()`-Funktion, die einen regulären Ausdruck (`pattern`) und einen String (`'M'`) übernimmt und versucht, das

Muster innerhalb des Strings zu finden. Hat `search()` etwas gefunden, gibt die Funktion ein Objekt zurück, das verschiedene Methoden zur Beschreibung des Fundes besitzt; findet `search()` nichts, wird `None` zurückgegeben, der Null-Wert von Python. Alles was Sie momentan interessieren sollte ist, ob das Muster gefunden wurde. Das sehen Sie schon am Rückgabewert von `search()`. 'M' stimmt hier mit dem regulären Ausdruck überein, da das erste optionale M-Zeichen passt und sowohl das zweite als auch das dritte ignoriert werden.

③ 'MM' wird gefunden, da das erste und das zweite optionale M-Zeichen passen und das dritte ignoriert wird.

④ 'MMM' wird gefunden, da alle drei M-Zeichen passen.

⑤ 'MMMM' wird nicht gefunden. Alle drei M-Zeichen sind vorhanden, doch dann besteht der reguläre Ausdruck auf dem Ende des Strings (wegen des `$`-Zeichens), doch der String endet noch nicht (aufgrund des vierten M-Zeichens). `search()` gibt also `None` zurück.

⑥ Interessanterweise stimmt auch ein leerer String mit dem Muster überein, da alle M-Zeichen optional sind.

### 6.3.2 Prüfen der Hunderter

Die Hunderter sind schwieriger zu handhaben als die Tausender, da es je nach dem Wert verschiedene Möglichkeiten der Darstellung gibt, die sich gegenseitig ausschließen.

- 100 = C
- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

Es gibt also vier mögliche Muster:

- CM
- CD
- Null bis drei C-Zeichen (null, wenn die Hunderterstelle 0 ist)
- D, gefolgt von null bis drei C-Zeichen

Die beiden letzten Muster können kombiniert werden.

- Ein optionales D, gefolgt von null bis drei C-Zeichen

Dieses Beispiel zeigt, wie man die Hunderterstelle einer römischen Zahl überprüft.

```

>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' ①
>>> re.search(pattern, 'MCM') ②
<_sre.SRE_Match object at 01070390>
>>> re.search(pattern, 'MD') ③
<_sre.SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC') ④
<_sre.SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC') ⑤
>>> re.search(pattern, '') ⑥
<_sre.SRE_Match object at 01071D98>

```

① Dieses Muster beginnt genau wie das vorherige. Zuerst prüft es den Anfang des Strings (^), dann die Tausenderstelle (M?M?M?). Darauf folgt – in Klammern – der neue Teil, der drei sich gegenseitig ausschließende Muster angibt, welche wiederum durch vertikale Balken getrennt sind: CM, CD und D?C?C?C? (das ist ein optionales D gefolgt von null bis drei optionalen C-Zeichen). Der Parser für reguläre Ausdrücke sucht der Reihe nach jedes der Muster (von links nach rechts), benutzt das erste das passt und ignoriert den Rest.

② 'MCM' wird gefunden, da das erste M passt, das zweite und dritte M-Zeichen ignoriert wird und das CM ebenfalls passt (die Muster CD und D?C?C?C? werden also gar nicht erst in die Prüfung mit einbezogen). MCM ist die römische Zahl für 1900.

③ 'MD' wird gefunden, weil das erste M passt, das zweite und dritte M-Zeichen ignoriert wird und das Muster D?C?C?C? zu D passt (jedes der C-Zeichen ist optional und wird ignoriert). MD ist die römische Zahl für 1500.

④ 'MMMCCC' wird gefunden, da alle drei M-Zeichen passen und das Muster D?C?C?C? zu CCC passt (das D ist optional und wird ignoriert). MMMCCC ist die römische Zahl für 3300.

⑤ 'MCMC' wird nicht gefunden. Das erste M passt, das zweite und dritte M-Zeichen wird ignoriert und das CM passt. Doch dann passt das \$-Zeichen nicht, da wir uns noch nicht am Ende des Strings befinden (es ist immer noch ein C-Zeichen vorhanden). Das C passt nicht zum Muster D?C?C?C?, da sich die Muster gegenseitig ausschließen und das Muster CM bereits gefunden wurde.

⑥ Interessanterweise passt auch ein leerer String zu dem Muster, da alle M-Zeichen optional sind und ignoriert werden. Der leere String passt außerdem auch zum Muster D?C?C?C?, da auch hier alle Zeichen optional sind und ignoriert werden.

Puh! Haben Sie bemerkt, wie schnell reguläre Ausdrücke wirklich unangenehm werden können? Und Sie haben gerade mal die Tausender und Hunderter hinter sich gebracht. Wenn Sie aber all dem gefolgt sind, dann werden die Zehner und Einer ein Spaziergang, da sie genau dasselbe Muster verwenden. Lassen Sie uns aber eine andere Möglichkeit zur Darstellung des Musters ansehen.

## 6.4 Verwenden der {n,m}-Syntax

Im vorherigen Abschnitt hatten Sie es mit einem Muster zu tun, bei dem dasselbe Zeichen bis zu dreimal wiederholt werden konnte. Bei den regulären Ausdrücken gibt es noch eine andere Möglichkeit dies auszudrücken. Diese Methode finden einige Leute lesbarer. Sehen Sie sich zuerst die im vorherigen Beispiel angewandte Methode an.

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')      ①
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')    ②
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')  ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM') ④
>>>
```

① Hier passen der Beginn des Strings und das erste optionale M, nicht jedoch das zweite und dritte M (das ist aber in Ordnung, da sie optional sind). Das Ende des Strings passt dann wieder.

② Hier passen der Beginn des Strings und das erste und das zweite optionale M, nicht aber das dritte M (das ist aber in Ordnung, da es optional ist). Das Ende des Strings passt dann wieder.

③ Hier passen der Beginn des Strings und alle drei optionalen M-Zeichen. Das Ende des Strings passt hier ebenfalls.

④ Hier passen der Beginn des Strings und alle drei optionalen M-Zeichen. Das Ende des Strings passt hier jedoch nicht (da immer noch ein M übrig ist). Das Muster wird also nicht gefunden und gibt None zurück.

```
>>> pattern = '^M{0,3}$'      ①
>>> re.search(pattern, 'M')  ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM') ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM') ④
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') ⑤
>>>
```

① Dieses Muster bedeutet: „Suche den Beginn des Strings, dann null bis drei M-Zeichen und dann das Ende des Strings.“ 0 und 3 können beliebige Zahlen sein; wollen Sie, dass mindestens ein M vorhanden ist, aber nicht mehr als drei, könnten Sie sagen `M{1,3}`.

② Hier passen der Beginn des Strings, eines der möglichen drei M-Zeichen und das Ende des Strings.

③ Hier passen der Beginn des Strings, zwei der möglichen drei M-Zeichen und das Ende des Strings.

④ Hier passen der Beginn des Strings, drei der möglichen drei M-Zeichen und das Ende des Strings.

⑤ Hier passen der Beginn des Strings und drei der möglichen drei M-Zeichen, doch nicht das Ende des Strings. Der reguläre Ausdruck erlaubt nur bis zu drei M-Zeichen vor dem Ende des Strings, doch hier sind es vier. Das Muster passt hier also nicht und gibt `None` zurück.

### 6.4.1 Prüfen der Zehner und Einer

Lassen Sie uns den regulären Ausdruck nun so erweitern, dass er auch die Zehner und Einer umfasst. Dieses Beispiel zeigt das Prüfen der Zehner.

```

>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?) (XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL') ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML') ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX') ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX') ④
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX') ⑤
>>>

```

① Hier passen der Beginn des Strings, das erste optionale M, außerdem CM, XL und schließlich das Ende des Strings. Denken Sie daran, dass `(A|B|C)` bedeutet „finde A oder B oder C“. Hier passt XL, also wird XC und `L?X?X?X?` ignoriert und zum Ende des Strings gegangen. MCML ist die römische Zahl für 1940.

② Hier passen der Beginn des Strings, das erste optionale M, CM und `L?X?X?X?`. Vom Muster `L?X?X?X?` passt hier das L; die optionalen X-Zeichen werden ignoriert. Dann wird zum Ende des Strings gegangen. MCML ist die römische Zahl für 1950.

③ Hier passen der Beginn des Strings sowie das erste optionale M, CM, das optionale L und das optionale X. Das zweite und dritte optionale X wird ignoriert. Das Ende des Strings passt ebenfalls. MCMLX ist die römische Zahl für 1960.

④ Hier passen der Beginn des Strings, das erste optionale M, CM, das optionale L, alle drei optionalen X-Zeichen und das Ende des Strings. MCMLXXX ist die römische Zahl für 1980.

⑤ Hier passen der Beginn des Strings, das erste optionale M, CM, das optionale L und alle drei optionalen X-Zeichen. Das Ende des Strings passt nicht, da es noch ein unbeachtetes X gibt. Das komplette Muster wird also nicht erkannt und gibt None zurück. MCMLXXXX ist keine gültige römische Zahl.

Der Ausdruck für die Einer folgt demselben Muster. Ich erspare Ihnen die Details und zeige Ihnen das Ergebnis.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

Wie sieht dies jetzt also unter Verwendung der {n,m}-Syntax aus? Dieses Beispiel zeigt die neue Syntax.

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV') ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI') ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCLXXXVIII') ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I') ④
<_sre.SRE_Match object at 0x008EEB48>
```

① Hier passen der Beginn des Strings, eines von möglichen drei M-Zeichen und D?C{0,3}. Davon passt das optionale D und null der möglichen drei C-Zeichen. Außerdem wird L?X{0,3} erkannt, da das optionale L und null der möglichen drei X-Zeichen passen. Auch V?I{0,3} wird gefunden, da das optionale V und null von möglichen drei I-Zeichen passen. Das Ende des Strings passt ebenso. MDLV ist die römische Zahl für 1555.

② Hier passen der Beginn des Strings, zwei von möglichen drei M-Zeichen und D?C{0,3} mit einem D und einem von möglichen drei C-Zeichen. Außerdem wird L?X{0,3} mit einem L und einem von möglichen drei X-Zeichen erkannt. Auch V?I{0,3} wird mit einem V und einem von möglichen drei I-Zeichen erkannt. Das Ende des Strings passt ebenso. MMDCLXVI ist die römische Zahl für 2666.

③ Hier passen der Beginn des Strings, drei von möglichen drei M-Zeichen und D?C{0,3} mit einem D und drei von möglichen drei C-Zeichen. Außerdem wird L?X{0,3} mit einem L und drei von möglichen drei X-Zeichen erkannt. Auch V?I{0,3} wird mit einem V und drei von möglichen drei I-Zeichen erkannt. Das Ende des Strings passt ebenfalls. MMMDCCLXXXVIII ist die römische Zahl

für 3888 und die längste mit römischen Zahlen ohne Erweiterung darstellbare Zahl.

④ Sehen Sie her. (Ich fühle mich wie ein Zauberer. „Seht her, Kinder! Ich werde ein Kaninchen aus meinem Hut zaubern.“) Hier passen der Beginn des Strings, null von drei  $M$ -Zeichen,  $D?C\{0, 3\}$ , da das optionale  $D$  ignoriert wird und null von drei  $C$ -Zeichen erkannt werden,  $L?X\{0, 3\}$ , da das optionale  $L$  ignoriert wird und null von drei  $X$ -Zeichen erkannt werden,  $V?I\{0, 3\}$ , da das optionale  $V$  ignoriert wird und null von drei  $I$ -Zeichen erkannt werden. Auch das Ende des Strings passt hier. Whoa!

Wenn Sie all das verstanden haben, sind Sie besser dran als ich es war. Stellen Sie sich nun vor, Sie versuchten die regulären Ausdrücke eines anderen inmitten einer kritischen Funktion eines großen Programms zu verstehen. Oder stellen Sie sich bloß vor, dass Sie nach einigen Monaten Ihre eigenen regulären Ausdrücke ansehen. Ich habe es getan und kann Ihnen sagen, dass es kein schöner Anblick ist.

Lassen Sie uns nun eine alternative Syntax ansehen, die Ihnen behilflich sein kann, Ihre Ausdrücke wartungsfähig zu halten.

## 6.5 Ausführliche reguläre Ausdrücke

Bisher haben Sie es nur mit – wie ich sie nenne – „kompakten“ regulären Ausdrücken zu tun gehabt. Wie Sie gesehen haben, sind diese schwer zu lesen und selbst wenn Sie herausfinden, was ein Ausdruck tut, ist das keine Garantie dafür, dass Sie es auch nach sechs Monaten noch wissen. Was Sie also brauchen ist eine beigefügte Dokumentation.

Python erlaubt Ihnen genau das durch sogenannte *verbose regular expressions* (ausführliche reguläre Ausdrücke; Anm. d. Übers.). Ein ausführlicher regulärer Ausdruck unterscheidet sich von einem kompakten regulären Ausdruck auf zweierlei Arten:

- Whitespace wird ignoriert. Leerzeichen, Tabulatoren und Zeilenumbrüche werden nicht als Leerzeichen, Tabulatoren und Zeilenumbrüche erkannt. Sie werden überhaupt nicht erkannt. (Möchten Sie, dass ein Leerzeichen in einem ausführlichen regulären Ausdruck erkannt wird, müssen Sie es escapen, indem Sie einen Backslash davor setzen.)
- Kommentare werden ignoriert. Ein Kommentar innerhalb eines ausführlichen regulären Ausdrucks ist genau wie ein Kommentar im Python-Code: er beginnt mit einem #-Zeichen und geht bis zum Ende der Zeile. In diesem Fall liegt ein Kommentar in einem mehrzeiligen String vor statt im Quellcode, doch die Funktionsweise ist die gleiche.

Ein Beispiel wird dies verdeutlichen. Sehen wir uns den bisher genutzten kompakten regulären Ausdruck an und machen daraus einen ausführlichen regulären Ausdruck. Dieses Beispiel zeigt wie.

```

>>> pattern = '''
    ^                # beginning of string
    M{0,3}           # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3}) # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                    # or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3}) # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                    # or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3}) # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                    # or 5-8 (V, followed by 0 to 3 Is)
    $                # end of string
    ...
>>> re.search(pattern, 'M', re.VERBOSE)           ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCLXXXIX', re.VERBOSE)   ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCLXXXVIII', re.VERBOSE) ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')                       ④

```

① Das Wichtigste das Sie beim Verwenden von ausführlichen regulären Ausdrücken bedenken müssen, ist dass Sie ein zusätzliches Argument übergeben müssen: `re.VERBOSE` ist eine im `re`-Modul definierte Konstante, die anzeigt, dass das Muster als ausführlicher regulärer Ausdruck behandelt werden soll. Wie Sie sehen, besitzt dieses Muster sehr viel Whitespace (der komplett ignoriert wird) und einige Kommentare (die komplett ignoriert werden). Vernachlässigen Sie den Whitespace und die Kommentare, so ist dies exakt der gleiche reguläre Ausdruck wie Sie ihn im vorherigen Abschnitt gesehen haben. Er ist jedoch viel lesbarer.

② Hier wird der Beginn des Strings, eines von möglichen drei M-Zeichen, CM, L und drei von möglichen drei X-Zeichen, IX und schließlich das Ende des Strings erkannt.

③ Hier wird der Beginn des Strings, drei von möglichen drei M-Zeichen, D und drei von möglichen drei C-Zeichen, L und drei von möglichen drei X-Zeichen, V und drei von möglichen drei I-Zeichen und schließlich das Ende des Strings erkannt.

④ Hier wird nichts erkannt. Warum nicht? Es ist kein `re.VERBOSE`-Flag vorhanden. Die `re.search()`-Funktion behandelt das Muster wie einen kompakten regulären Ausdruck mit erheblichem Whitespace und Raute-Symbolen. Python kann nicht automatisch feststellen, ob ein regulärer Ausdruck ausführlich ist oder nicht. Stattdessen nimmt Python an, dass jeder reguläre Ausdruck kompakt ist, es sei denn Sie geben explizit an, dass er ausführlich ist.

## 6.6 Fallbeispiel: Telefonnummern gliedern

Bis jetzt haben Sie sich darauf konzentriert, ganze Muster zu erkennen. Entweder das Muster passt, oder es passt nicht. Doch reguläre Ausdrücke sind noch weit mächtiger. Passt ein regulärer Ausdruck, dann können Sie bestimmte Teile davon herauspicken. Sie können herausfinden, was wo erkannt wurde.

Dieses Beispiel stammt von einem weiteren wirklichen Problem dem ich mich während meiner früheren Arbeit gegenüber sah. Das Problem: eine US-amerikanische Telefonnummer gliedern. Der Kunde wollte die Möglichkeit haben, die Nummer in jeder beliebigen Form einzugeben (in ein einzelnes Eingabefeld). In der Datenbank seines Unternehmens sollte die Nummer dann jedoch aufgeteilt in Ortsvorwahl, Amt, Teilnehmernummer und eine eventuelle Durchwahl gespeichert werden. Ich durchkämmte das Internet und habe viele Beispiele für reguläre Ausdrücke gefunden, die behaupteten genau dies zu können, doch keines davon war tolerant genug.

Hier sind die Telefonnummern, die ich erkennen musste:

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212
- 1-800-555-1212
- 800-555-1212-1234
- 800-555-1212x1234
- 800-555-1212 ext. 1234
- work 1-(800) 555.1212 #1234

Ziemlich viele Möglichkeiten! In jedem der Fälle musste ich wissen, dass die Ortsvorwahl 800 lautete, das Amt 555 und der Rest der Telefonnummer 1212. Bei den Nummern die eine Durchwahl hatten, musste ich außerdem wissen, dass diese 1234 war.

Lassen Sie uns eine Lösung zum Gliedern von Telefonnummern erarbeiten. Dieses Beispiel zeigt den ersten Schritt.

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$') ①
>>> phonePattern.search('800-555-1212').groups() ②
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234') ③
>>> phonePattern.search('800-555-1212-1234').groups() ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

① Lesen Sie reguläre Ausdrücke immer von links nach rechts. Dieser hier vergleicht den Beginn des Strings und  $(\{\d\{3\}\})$ . Was bedeutet  $\d\{3\}$ ? Nun,  $\d$  bedeutet „jede beliebige Ziffer“ (0 bis 9).  $\{3\}$  heißt „vergleiche genau drei Ziffern“; es stellt eine Variante der eben gesehenen  $\{n, m\}$ -Syntax dar. Setzt man dies alles in Klammern heißt das „suche genau drei Ziffern und behalte Sie als Gruppe, nach der ich später fragen kann“. Suche dann einen Bindestrich. Dann erneute eine Gruppe von genau drei Ziffern. Dann wieder einen Bindestrich. Dann eine weitere Gruppe von genau drei Ziffern. Dann das Ende des Strings.

② Um Zugang zu den Gruppen zu erhalten, die der Parser für reguläre Ausdrücke sich gemerkt hat, verwenden Sie die `groups()`-Methode des Objekts, das die

`search()`-Methode zurückgibt. Die Methode gibt dann ein Tupel dieser Gruppen zurück. Im vorliegenden Fall haben Sie drei Gruppen definiert, eine mit drei Ziffern, eine weitere mit drei Ziffern und eine mit vier Ziffern.

③ Dieser reguläre Ausdruck ist nicht die endgültige Lösung, da er nicht mit einer Durchwahl am Ende einer Telefonnummer umgehen kann. Um dies zu erreichen, müssen Sie den regulären Ausdruck erweitern.

④ Hier sehen Sie, warum Sie im fertigen Code niemals die `search()`-Methode und die `groups()`-Methode „verketten“ sollten. Gibt die `search()`-Methode keinen Treffer zurück, wird `None` geliefert. `None.groups()` aber verursacht eine eindeutige Ausnahme: `None` besitzt keine Methode namens `groups()`. (Natürlich ist dies weniger eindeutig, wenn die Ausnahme irgendwo versteckt in Ihrem Code auftritt. Ja, ich spreche aus Erfahrung.)

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$') ①
>>> phonePattern.search('800-555-1212-1234').groups()          ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                    ③
>>>
>>> phonePattern.search('800-555-1212')                          ④
>>>
```

① Dieser reguläre Ausdruck ist beinahe mit dem vorherigen identisch. Genau wie vorhin wird auch hier der Beginn des Strings verglichen, dann eine zu merkende Gruppe von drei Ziffern, ein Bindestrich, wieder eine zu merkende Gruppe von drei Ziffern, ein weiterer Bindestrich und eine zu merkende Gruppe von vier Ziffern. Neu ist hier, dass hier ein weiterer Bindestrich und schließlich eine zu merkende Gruppe von einer oder mehr Ziffern gesucht werden. Dann erst das Ende des Strings.

② Die `groups()`-Methode gibt nun ein Tupel von vier Elementen zurück, da der reguläre Ausdruck jetzt vier Gruppen angibt, die der Parser sich merken soll.

③ Leider ist auch dieser reguläre Ausdruck keine endgültige Lösung. Er geht davon aus, dass die verschiedenen Teile der Telefonnummer durch Bindestriche getrennt sind. Was also passiert, wenn sie durch Leerzeichen, Kommas, oder Punkte getrennt werden. Sie benötigen eine allgemeinere Lösung, die die verschiedenen Arten der Trennung berücksichtigt.

④ Huch! Dieser reguläre Ausdruck tut nicht nur nicht alles was Sie wollen, er ist sogar ein Rückschritt, weil Sie nun keine Telefonnummern ohne eine Durchwahl gliedern können. Das ist absolut nicht das was Sie wollten. Wenn eine Durchwahl vorhanden ist, wollen Sie diese wissen. Wenn Sie jedoch nicht vorhanden ist, wollen Sie trotzdem die einzelnen Teile der Hauptnummer erhalten.

Das nächste Beispiel zeigt, wie man den regulären Ausdruck so gestaltet, dass er mit verschiedenen Trennzeichen zwischen den einzelnen Teilen der Telefonnummer umgehen kann.

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$') ①
>>> phonePattern.search('800 555 1212 1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups() ③
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234') ④
>>>
>>> phonePattern.search('800-555-1212') ⑤
>>>
```

① Halten Sie sich fest! Hier vergleichen Sie den Beginn des Strings, eine Gruppe von drei Ziffern und `\D+`. Was ist das nun wieder? Nun, `\D` entspricht jedem beliebigen Zeichen ausgenommen einer Ziffer. Das `+` bedeutet „1 oder mehr“. `\D+` entspricht also einem oder mehr Zeichen, die keine Ziffern sind. Dies benutzen Sie anstelle eines Bindestrichs, um verschiedene Trennzeichen zu erkennen.

② Die Verwendung von `\D+` führt dazu, dass Sie nun Telefonnummern erkennen können, bei denen das Trennzeichen ein Leerzeichen ist.

③ Natürlich werden auch Bindestriche weiterhin erkannt.

④ Leider ist dies immer noch nicht die endgültige Lösung, weil davon ausgegangen wird, dass überhaupt ein Trennzeichen vorhanden ist. Was passiert nun aber, wenn die Telefonnummer ohne Leerzeichen oder Bindestriche eingegeben wird?

⑤ Huch! Dies hat immer noch nicht unser Problem mit der Durchwahl beseitigt. Sie stehen nun vor zwei Problemen, doch diese können Sie beide mit derselben Technik lösen.

Das folgende Beispiel zeigt den regulären Ausdruck, der auch Telefonnummern ohne Trennzeichen verarbeiten kann.

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d+)$') ①
>>> phonePattern.search('80055512121234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups() ③
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ④
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234') ⑤
>>>
```

① Die einzige Änderung die Sie gegenüber dem letzten Schritt durchgeführt haben besteht darin, dass alle `+`-Zeichen durch `*`-Zeichen ersetzt wurden. Statt zwischen den Teilen der Telefonnummer nach `\D+` zu suchen, suchen Sie nun nach `\D*`. Erinnern Sie sich, dass `+` „1 oder mehr“ bedeutet? Nun, `*` bedeutet „0 oder mehr“. Sie sollten jetzt also auch Telefonnummern gliedern können, die gar kein Trennzeichen beinhalten.

② Und siehe da, es funktioniert! Warum? Sie haben den Beginn des Strings verglichen, dann eine zu merkende Gruppe von drei Ziffern (800), null nicht-nu-

merische Zeichen, wieder eine zu merkende Gruppe von drei Ziffern (555), null nicht-numerische Zeichen, eine zu merkende Gruppe von vier Ziffern (1212), null nicht-numerische Zeichen, eine zu merkende Gruppe von einer variablen Anzahl an Ziffern (1234) und dann das Ende des Strings.

③ Auch andere Varianten funktionieren nun: Punkte anstelle von Bindestrichen, und sowohl ein Leerzeichen wie auch ein `x` vor der Durchwahl.

④ Endlich haben Sie unser Problem gelöst: eine Durchwahl ist nun wieder optional. Ist keine Durchwahl vorhanden gibt die `groups()`-Methode zwar weiterhin ein Tupel mit vier Elementen zurück, doch das vierte Element ist nur ein leerer String.

⑤ Ich hasse es der Überbringer schlechter Nachrichten zu sein, aber Sie sind noch nicht fertig. Was ist das Problem? Hier gibt es ein zusätzliches Zeichen vor der Ortsvorwahl. Der reguläre Ausdruck geht jedoch davon aus, dass die Ortsvorwahl das Erste ist womit der String beginnt. Kein Problem. Sie können dieselbe Technik der „0 oder mehr nicht-numerischen Zeichen“ nutzen, um führende Zeichen vor der Ortsvorwahl zu ignorieren.

Das nachfolgende Beispiel zeigt, wie man führende Zeichen in Telefonnummern handhabt.

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('(800)5551212 ext. 1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ③
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234') ④
>>>
```

① Dies ist dasselbe wie im vorherigen Beispiel, abgesehen davon, dass Sie jetzt vor der ersten zu merkenden Gruppe (die Ortsvorwahl) nach `\D*` suchen. Beachten Sie, dass der Parser sich diese nicht-numerischen Zeichen nicht merkt (sie sind nicht in Klammern eingeschlossen). Werden Sie gefunden, dann werden sie einfach ignoriert und der Parser merkt sich die Ortsvorwahl wann immer sie beginnt.

② Sie können nun die Telefonnummer auch mit der führenden linken Klammer gliedern. (Die rechte Klammer nach der Ortsvorwahl wurde schon vorher verarbeitet; sie wird als nicht-numerisches Trennzeichen behandelt, das von `\D*` nach der ersten zu merkenden Gruppe erkannt wird.)

③ Dies ist nur ein Funktionstest, um sicherzustellen, dass nichts kaputt gegangen ist, was vorher funktioniert hat. Da die führenden Zeichen optional sind, werden hier der Beginn des Strings, null nicht-numerische Zeichen, eine zu merkende Gruppe von drei Ziffern (800), ein nicht-numerisches Zeichen (der Bindestrich), eine zu merkende Gruppe von drei Ziffern (555), ein nicht-numerisches Zeichen (der Bindestrich), eine zu merkende Gruppe von vier Ziffern (1212), null nicht-numerische Zeichen, eine zu merkende Gruppe von null Ziffern und schließlich das Ende des Strings erkannt.

④ Hier sehen wir, warum ich mir angesichts regulärer Ausdrücke manchmal die Augen mit einem stumpfen Gegenstand ausstechen will. Warum wird diese Telefon-

nummer nicht erkannt? Ganz einfach: es befindet sich eine 1 vor der Ortsvorwahl, doch Sie sind davon ausgegangen, dass alle führenden Zeichen vor der Ortsvorwahl nicht-numerische Zeichen sind (`\D*`). Oh je!

Lassen Sie uns kurz zurückblicken. Bisher haben all unsere regulären Ausdrücke nach dem Beginn des Strings gesucht. Sie sehen nun aber, dass am Beginn des Strings eine unbestimmte Anzahl an Dingen stehen kann die Sie ignorieren wollen. Statt nun zu versuchen, all diese Dinge zu erkennen und zu überspringen, gehen wir es anders an: suchen Sie gar nicht erst den Beginn des Strings. Diese Vorgehensweise wird im nächsten Beispiel gezeigt.

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()      ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                               ③
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234')                             ④
('800', '555', '1212', '1234')
```

① Beachten Sie bei diesem regulären Ausdruck das Fehlen des `^`-Zeichens. Sie suchen nicht länger den Beginn des Strings. Es gibt kein Gesetz, das Sie dazu zwingt, die komplette Eingabe mit Ihrem regulären Ausdruck zu erkennen. Die Engine für reguläre Ausdrücke wird die harte Arbeit erledigen und herausfinden, ab wo der Eingabestring passt. Von dort aus wird er dann weiter verarbeitet.

② Nun können Sie jede beliebige Telefonnummer die führende Zeichen oder eine führende Ziffer enthält und außerdem jedwedes Trennzeichen in beliebiger Anzahl beinhaltet zergliedern.

③ Funktionstest. Es funktioniert immer noch.

④ Auch dies funktioniert noch.

Haben Sie bemerkt, wie schnell reguläre Ausdrücke außer Kontrolle geraten können? Sehen Sie sich noch einmal kurz alle vorherigen aufeinander aufbauenden Ausdrücke an. Können Sie den Unterschied zwischen einem und dem nächsten benennen?

Während Sie die endgültige Lösung noch verstehen (und es ist die endgültige Lösung; wenn Sie einen Fall entdecken, bei dem es nicht funktioniert, will ich es nicht wissen), sollten wir sie nun als ausführlichen regulären Ausdruck schreiben, bevor Sie vergessen, warum Sie die Entscheidungen getroffen haben, die Sie getroffen haben.

```
>>> phonePattern = re.compile(r'''
                                # don't match beginning of string, number can start anywhere
    (\d{3}) # area code is 3 digits (e.g. '800')
    \D*    # optional separator is any number of non-digits
    (\d{3}) # trunk is 3 digits (e.g. '555')
    \D*    # optional separator
    (\d{4}) # rest of number is 4 digits (e.g. '1212')
```

```

\D*      # optional separator
(\d*)    # extension is optional and can be any number of digits
$        # end of string
'', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ①
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                          ②
('800', '555', '1212', '')

```

① Abgesehen davon, dass sich dieser reguläre Ausdruck über mehrere Zeilen erstreckt, ist es immer noch derselbe wie der im letzten Schritt. Es ist daher keine Überraschung, dass er auch dieselben Eingaben verarbeitet.

② Abschließender Funktionstest. Ja, es funktioniert noch. Sie haben es geschafft!

## 6.7 Zusammenfassung

Dies ist erst die sehr kleine Spitze des Eisbergs. Reguläre Ausdrücke können sehr viel mehr als das. Anders ausgedrückt: auch wenn Sie nun völlig überwältigt von ihnen sind, glauben Sie mir, dass Sie noch nichts davon gesehen haben.

Die folgenden Techniken sollten Sie nun kennen:

- `^` steht für den Beginn eines Strings
- `$` steht für das Ende eines Strings
- `\b` steht für eine Wortgrenze
- `\d` steht für eine Ziffer
- `\D` steht für ein nicht-numerisches Zeichen
- `x?` steht für ein optionales `x`-Zeichen (es steht also für kein oder für ein `x`)
- `x*` steht für kein oder mehrere `x`-Zeichen
- `x+` steht für ein oder mehrere `x`-Zeichen
- `x{n,m}` steht für ein `x`-Zeichen, das mindestens `n`-mal, aber nicht öfter als `m`-mal vorkommen darf
- `(a|b|c)` steht für `a` oder `b` oder `c`
- `(x)` ist allgemein eine zu merkende Gruppe. Sie erhalten den gefundenen Wert, indem Sie die `groups()`-Methode des von `re.search` zurückgegebenen Objekts aufrufen.

Auch wenn reguläre Ausdrücke sehr mächtig sind, sind sie dennoch nicht für jedes Problem die passende Lösung. Sie sollten so viel über sie lernen, dass Sie beurteilen können, wann sie angebracht sind, wann sie Ihre Probleme lösen können und wann Sie mehr Probleme verursachen als sie lösen.