Reguläre Ausdrücke: Regex in Python (Teil 1)

Inhaltsverzeichnis

In diesem Tutorial lernen Sie **reguläre Ausdrücke** , auch bekannt als **Regexes** , in Python kennen. Eine Regex ist eine spezielle Folge von Zeichen, die ein Muster für komplexe String-Matching-Funktionen definiert.

Weiter oben in dieser Serie haben Sie im Lernprogramm Strings and Character Data in Python gelernt, wie Sie String-Objekte definieren und bearbeiten. Seitdem haben Sie einige Möglichkeiten gesehen, um festzustellen, ob zwei Zeichenfolgen übereinstimmen:

- Sie können testen, ob zwei Zeichenfolgen gleich sind, indem Sie die Gleichheit (`==`) Betreiber.
- Ob ein String ein Teilstring eines anderen ist, können Sie mit testen `in`-Operator oder die integrierten Zeichenfolgenmethoden `.find()`und `.index()`.

String-Matching wie dieses ist eine häufige Aufgabe beim Programmieren, und Sie können mit String-Operatoren und integrierten Methoden viel erreichen. Manchmal benötigen Sie jedoch möglicherweise ausgefeiltere Mustererkennungsfunktionen.

In diesem Tutorial lernen Sie:

- So greifen Sie auf die zu **`re`module** , das den Regex-Abgleich in Python implementiert
- Wie benutzt man **`re.search()`**um ein Muster mit einer Zeichenfolge abzugleichen
- So erstellen Sie komplexe Übereinstimmungsmuster mit Regex- **Metazeichen**

Schnall dich an! Die Regex-Syntax ist etwas gewöhnungsbedürftig. Aber sobald Sie sich damit vertraut gemacht haben, werden Sie feststellen, dass Regexes in Ihrer Python-Programmierung fast unverzichtbar sind.

Regexe in Python und ihre Verwendung

Stellen Sie sich vor, Sie haben ein String-Objekt `s`. Angenommen, Sie müssen Python-Code schreiben, um herauszufinden, ob `s`enthält den Teilstring `'123'`. Es gibt mindestens ein paar Möglichkeiten, dies zu tun. Du könntest die verwenden `in`Operator:

```
>>> s = 'foo123bar'
>>> '123' in s
True
```

Wenn Sie nicht nur wissen wollen, *ob* `'123'`existiert in `s`aber auch *wo* es vorhanden ist, kann man es verwenden `.find()`oder `.index()`. Jeder von diesen gibt die Zeichenposition innerhalb zurück `s`wo sich der Teilstring befindet:

```
>>> s = 'foo123bar'
>>> s.find('123')
3
>>> s.index('123')
3
```

In diesen Beispielen erfolgt der Abgleich durch einen einfachen zeichenweisen Vergleich. Das wird die Arbeit in vielen Fällen erledigen. Aber manchmal ist das Problem komplizierter.

Anstatt beispielsweise nach einer festen Teilzeichenfolge wie zu suchen `'123'`, nehmen Sie an, Sie möchten bestimmen, ob eine Zeichenfolge drei aufeinanderfolgende Dezimalziffern enthält , wie in den Zeichenfolgen `'foo123bar'`, `'foo456bar'`, `'234baz'`, und `'qux678'`.

Strenge Charaktervergleiche reichen hier nicht aus. Hier kommen Regexes in Python zur Rettung.

Eine (sehr kurze) Geschichte regulärer Ausdrücke

1951 beschrieb der Mathematiker Stephen Cole Kleene das Konzept einer regulären Sprache formal ausgedrückt werden regulären Ausdrücken . Mitte der 1960er Jahre implementierte der Informatikpionier Ken Thompson , einer der ursprünglichen Designer von Unix, den Mustervergleich im QED-Texteditor unter Verwendung der Kleene-Notation.

Seitdem sind Regexes in vielen Programmiersprachen, Editoren und anderen Tools aufgetaucht, um festzustellen, ob eine Zeichenfolge mit einem bestimmten Muster übereinstimmt. Python, Java und Perl unterstützen alle die Regex-Funktionalität, ebenso wie die meisten Unix-Tools und viele Texteditoren.

Das `re`Modul

Die Regex-Funktionalität in Python befindet sich in einem Modul namens `re`. Das `re`-Modul enthält viele nützliche Funktionen und Methoden, von denen Sie die meisten im nächsten Tutorial dieser Reihe kennenlernen werden.

Im Moment konzentrieren Sie sich hauptsächlich auf eine Funktion, `re.search()`.

`re.search(<regex>, <string>)`

Durchsucht eine Zeichenfolge nach einer Regex-Übereinstimmung.

`re.search(<regex>, <string>)`scannt `<string>`Suche nach dem ersten Ort, an dem das Muster `<regex>`Streichhölzer. Wenn eine Übereinstimmung gefunden wird, dann `re.search()`gibt ein **Match-Objekt** . Andernfalls kehrt es zurück `None`.

`re.search()`nimmt ein optionales Drittel `<flags>`Argument, das Sie am Ende dieses Tutorials kennenlernen werden.

So importieren Sie `re.search()`

Da `search()`wohnt in der `re`Modul, müssen Sie es importieren , bevor Sie es verwenden können. Eine Möglichkeit besteht darin, das gesamte Modul zu importieren und dann den Modulnamen als Präfix beim Aufruf der Funktion zu verwenden:

```
import re
re.search(...)
```

Alternativ können Sie die Funktion aus dem Modul nach Namen importieren und dann ohne das Präfix des Modulnamens darauf verweisen:

```
from re import search
search(...)
```

Sie müssen immer importieren `re.search()`auf die eine oder andere Weise, bevor Sie es verwenden können.

Die Beispiele im Rest dieses Lernprogramms gehen vom ersten gezeigten Ansatz aus – dem Importieren der `re`Modul und dann auf die Funktion mit dem Präfix des Modulnamens verweisen: `re.search()`. Der Kürze halber die `import re`-Anweisung wird normalerweise weggelassen, aber denken Sie daran, dass sie immer notwendig ist.

Weitere Informationen zum Importieren aus Modulen und Paketen finden Sie unter Python-Module und -Pakete – Eine Einführung .

Erstes Mustervergleichsbeispiel

Jetzt wissen Sie, wie Sie Zugriff erhalten `re.search()`, du kannst es versuchen:

```
>>> s = 'foo123bar'



>>> # One last reminder to import!

>>> import re



>>> re.search('123', s)

<_sre.SRE_Match object; span=(3, 6), match='123'>
```

Hier das Suchmuster `<regex>`ist `123`und `<string>`ist `s`. Das zurückgegebene Übereinstimmungsobjekt wird in **Zeile 7** . Match-Objekte enthalten eine Fülle nützlicher Informationen, die Sie bald erkunden werden.

Im Moment ist das der wichtige Punkt `re.search()`hat tatsächlich ein Übereinstimmungsobjekt zurückgegeben, anstatt `None`. Das sagt Ihnen, dass es eine Übereinstimmung gefunden hat. Mit anderen Worten, die angegebene `<regex>`Muster `123`ist dabei `s`.

Ein Match-Objekt ist **truthy** , sodass Sie es in einem booleschen Kontext wie eine bedingte Anweisung verwenden können:

```
>>> if re.search('123', s):
...     print('Found a match.')
... else:
...     print('No match.')
...
Found a match.
```

Der Interpreter zeigt das Übereinstimmungsobjekt als an `<_sre.SRE_Match object; span=(3, 6), match='123'>`. Diese enthält einige nützliche Informationen.

`span=(3, 6)`gibt den Anteil an `<string>`in dem die Übereinstimmung gefunden wurde. Dies bedeutet dasselbe wie in der Slice-Notation :

```
>>> s[3:6]
'123'
```

In diesem Beispiel beginnt die Übereinstimmung an der Zeichenposition `3`und erstreckt sich bis zu, aber nicht einschließlich Position `6`.

`match='123'`gibt an, aus welchen Zeichen `<string>`abgestimmt.

Das ist ein guter Anfang. Aber in diesem Fall die `<regex>`Muster ist nur die einfache Zeichenfolge `'123'`. Der Musterabgleich ist hier immer noch nur ein Zeichen-für-Zeichen-Vergleich, ziemlich genau derselbe wie der `in`Betreiber und `.find()`zuvor gezeigte Beispiele. Das Match-Objekt teilt Ihnen hilfreich mit, dass die übereinstimmenden Zeichen waren `'123'`, aber das ist keine große Offenbarung, da das genau die Zeichen waren, nach denen Sie gesucht haben.

Du wärmst dich gerade auf.

Python-Regex-Metazeichen

Die wahre Stärke des Regex-Abgleichs in Python zeigt sich, wenn `<regex>`enthält Sonderzeichen, die als **Metazeichen** . Diese haben eine einzigartige Bedeutung für die Regex-Matching-Engine und verbessern die Möglichkeiten der Suche erheblich.

Betrachten Sie noch einmal das Problem, wie festgestellt werden kann, ob eine Zeichenfolge drei aufeinanderfolgende Dezimalziffern enthält.

In einer Regex eine Reihe von Zeichen, die in eckigen Klammern (`[]`) bildet eine **Zeichenklasse** . Diese Metazeichenfolge entspricht jedem einzelnen Zeichen in der Klasse, wie im folgenden Beispiel gezeigt:

```
>>> s = 'foo123bar'
>>> re.search('[0-9][0-9][0-9]', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

`[0-9]`entspricht jedem einzelnen Dezimalziffernzeichen – jedem Zeichen dazwischen `'0'`und `'9'`, inklusive. Der volle Ausdruck `[0-9][0-9][0-9]`stimmt mit einer beliebigen Folge von drei Dezimalziffern überein. In diesem Fall, `s`stimmt überein, da es drei aufeinanderfolgende Dezimalziffern enthält, `'123'`.

Diese Zeichenfolgen passen auch:

```
>>> re.search('[0-9][0-9][0-9]', 'foo456bar')
<_sre.SRE_Match object; span=(3, 6), match='456'>

>>> re.search('[0-9][0-9][0-9]', '234baz')
<_sre.SRE_Match object; span=(0, 3), match='234'>

>>> re.search('[0-9][0-9][0-9]', 'qux678')
<_sre.SRE_Match object; span=(3, 6), match='678'>
```

Andererseits wird eine Zeichenfolge, die nicht drei aufeinanderfolgende Ziffern enthält, nicht übereinstimmen:

```
>>> print(re.search('[0-9][0-9][0-9]', '12foo34'))
None
```

Mit regulären Ausdrücken in Python können Sie Muster in einer Zeichenfolge identifizieren, die Sie mit der nicht finden würden `in`Operator oder mit String-Methoden.

Schauen Sie sich ein anderes Regex-Metazeichen an. Der Punkt (`.`) metacharacter entspricht jedem Zeichen außer einem Zeilenumbruch und funktioniert daher wie ein Platzhalter:

```
>>> s = 'foo123bar'
>>> re.search('1.3', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> s = 'foo13bar'
>>> print(re.search('1.3', s))
None
```

Im ersten Beispiel ist die Regex `1.3`Streichhölzer `'123'`weil die `'1'`und `'3'`Spiel buchstäblich, und die `.`entspricht dem `'2'`. Hier fragst du im Wesentlichen: „Tut es `s`enthalten a `'1'`, dann ein beliebiges Zeichen (außer einem Zeilenumbruch), dann a `'3'`?” Die Antwort ist ja für `'foo123bar'`aber nein für `'foo13bar'`.

Diese Beispiele veranschaulichen schnell die Leistungsfähigkeit von Regex-Metazeichen. Zeichenklasse und Punkt sind nur zwei der von der unterstützten Metazeichen `re`Modul. Es gibt viele mehr. Als Nächstes erkunden Sie sie vollständig.

Metazeichen Unterstützt von der `re`Modul

Die folgende Tabelle fasst kurz alle Metazeichen zusammen, die von unterstützt werden `re`Modul. Einige Charaktere dienen mehr als einem Zweck:

Figuren)	Bedeutung
`.`	Stimmt mit jedem einzelnen Zeichen außer Newline überein
`^`	· Verankert ein Match am Anfang einer Zeichenfolge · Ergänzt eine Zeichenklasse
`$`	Verankert ein Streichholz am Ende einer Zeichenfolge
`*`	Stimmt mit null oder mehr Wiederholungen überein
`+`	Entspricht einer oder mehreren Wiederholungen
`?`	· Stimmt mit null oder einer Wiederholung überein · Gibt die nicht gierigen Versionen von an `*`, `+`, und `?` · Führt eine Lookahead- oder Lookbehind-Assertion ein · Erstellt eine benannte Gruppe
`{}`	Entspricht einer explizit angegebenen Anzahl von Wiederholungen
`\`	· Escapezeichen für ein Metazeichen seiner besonderen Bedeutung · Führt eine Sonderzeichenklasse ein · Führt eine Gruppierungs-Rückreferenz ein
`[]`	Gibt eine Zeichenklasse an
`|`	Bezeichnet Abwechslung
`()`	Erstellt eine Gruppe

`:`
`#`
`=`
`!`
	Benennen Sie eine spezialisierte Gruppe

`<>`	Erstellt eine benannte Gruppe

Dies mag wie eine überwältigende Menge an Informationen erscheinen, aber keine Panik! In den folgenden Abschnitten wird jede davon im Detail behandelt.

Der Regex-Parser betrachtet alle Zeichen, die oben nicht aufgeführt sind, als gewöhnliche Zeichen, die nur mit sich selbst übereinstimmen. Im Mustervergleichsbeispiel haben Sie beispielsweise oben gezeigten

```
>>> s = 'foo123bar'
>>> re.search('123', s)
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

In diesem Fall, `123`ist technisch gesehen eine Regex, aber nicht sehr interessant, weil sie keine Metazeichen enthält. Es passt einfach zu der Zeichenfolge `'123'`.

Die Dinge werden viel spannender, wenn Sie Metazeichen in die Mischung werfen. In den folgenden Abschnitten wird ausführlich erläutert, wie Sie die einzelnen Metazeichen oder Metazeichenfolgen verwenden können, um die Mustererkennungsfunktionalität zu verbessern.

Metazeichen, die einem einzelnen Zeichen entsprechen

Die Metazeichenfolgen in diesem Abschnitt versuchen, ein einzelnes Zeichen aus der Suchzeichenfolge zu finden. Wenn der Regex-Parser auf eine dieser Metazeichensequenzen trifft, findet eine Übereinstimmung statt, wenn das Zeichen an der aktuellen Parsing-Position zu der Beschreibung passt, die die Sequenz beschreibt.

`[]`

Gibt einen bestimmten Satz von Zeichen an, die abgeglichen werden sollen.

Zeichen in eckigen Klammern (`[]`) stellen eine **Zeichenklasse** aufgezählte Menge von Zeichen, die abgeglichen werden sollen. Eine Zeichenklassen-Metazeichensequenz stimmt mit jedem einzelnen Zeichen überein, das in der Klasse enthalten ist.

Sie können die Zeichen einzeln wie folgt aufzählen:

```
>>> re.search('ba[artz]', 'foobarqux')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> re.search('ba[artz]', 'foobazqux')
<_sre.SRE_Match object; span=(3, 6), match='baz'>
```

Die Metazeichenfolge `[artz]`passt zu jeder Single `'a'`, `'r'`, `'t'`, oder `'z'`Charakter. Im Beispiel die Regex `ba[artz]`passt zu beiden `'bar'`und `'baz'`(und würde auch passen `'baa'`und `'bat'`).

Eine Zeichenklasse kann auch eine Reihe von Zeichen enthalten, die durch einen Bindestrich (`-`), in diesem Fall stimmt es mit jedem einzelnen Zeichen innerhalb des Bereichs überein. Zum Beispiel, `[a-z]`stimmt mit jedem Kleinbuchstaben dazwischen überein `'a'`und `'z'`, inklusive:

```
>>> re.search('[a-z]', 'FOObar')
<_sre.SRE_Match object; span=(3, 4), match='b'>
```

`[0-9]`passt zu jedem Ziffernzeichen:

```
>>> re.search('[0-9][0-9]', 'foo123bar')
<_sre.SRE_Match object; span=(3, 5), match='12'>
```

In diesem Fall, `[0-9][0-9]`entspricht einer Folge von zwei Ziffern. Der erste Teil der Zeichenfolge `'foo123bar'`das passt ist `'12'`.

`[0-9a-fA-F]`stimmt mit jedem hexadezimalen Ziffernzeichen überein:

```
>>> re.search('[0-9a-fA-f]', '--- a0 ---')
<_sre.SRE_Match object; span=(4, 5), match='a'>
```

Hier, `[0-9a-fA-F]`stimmt mit dem ersten hexadezimalen Ziffernzeichen in der Suchzeichenfolge überein, `'a'`.

Hinweis: In den obigen Beispielen ist der Rückgabewert immer die möglichst linke Übereinstimmung. `re.search()`scannt die Suchzeichenfolge von links nach rechts und sobald eine Übereinstimmung gefunden wird `<regex>`, stoppt es das Scannen und gibt die Übereinstimmung zurück.

Sie können eine Zeichenklasse durch Angabe ergänzen `^`als erstes Zeichen, in diesem Fall stimmt es mit jedem Zeichen überein, *das nicht* im Satz enthalten ist. Im folgenden Beispiel `[^0-9]`stimmt mit jedem Zeichen überein, das keine Ziffer ist:

```
>>> re.search('[^0-9]', '12345foo')
<_sre.SRE_Match object; span=(5, 6), match='f'>
```

Hier gibt das Match-Objekt an, dass das erste Zeichen in der Zeichenfolge, das keine Ziffer ist, eine ist `'f'`.

Wenn ein ^Zeichen in einer Zeichenklasse auftaucht, aber nicht das erste Zeichen ist, dann hat es keine besondere Bedeutung und passt auf ein Literal `'^'`Charakter:

```
>>> re.search('[#:^]', 'foo^bar:baz#qux')
<_sre.SRE_Match object; span=(3, 4), match='^'>
```

Wie Sie gesehen haben, können Sie einen Bereich von Zeichen in einer Zeichenklasse angeben, indem Sie Zeichen mit einem Bindestrich trennen. Was ist, wenn Sie möchten, dass die Zeichenklasse einen wörtlichen Bindestrich enthält? Sie können es als erstes oder letztes Zeichen platzieren oder mit einem Backslash (\):

```
>>> re.search('[-abc]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
>>> re.search('[abc-]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
>>> re.search('[ab\-c]', '123-456')
<_sre.SRE_Match object; span=(3, 4), match='-'>
```

Wenn Sie ein Literal einschließen möchten `']'`in einer Zeichenklasse, dann können Sie es als erstes Zeichen platzieren oder mit einem Backslash maskieren:

```
>>> re.search('[]]', 'foo[1]')
<_sre.SRE_Match object; span=(5, 6), match=']'>
>>> re.search('[ab\]cd]', 'foo[1]')
<_sre.SRE_Match object; span=(5, 6), match=']'>
```

Andere Regex-Metazeichen verlieren innerhalb einer Zeichenklasse ihre besondere Bedeutung:

```
>>> re.search('[)*+|]', '123*456')
<_sre.SRE_Match object; span=(3, 4), match='*'>
>>> re.search('[)*+|]', '123+456')
<_sre.SRE_Match object; span=(3, 4), match='+'>
```

Wie Sie in der obigen Tabelle gesehen haben, *und +haben besondere Bedeutungen in einer Regex in Python. Sie bezeichnen Wiederholungen, über die Sie in Kürze mehr erfahren werden. Aber in diesem Beispiel befinden sie sich innerhalb einer Zeichenklasse, also stimmen sie buchstäblich mit sich selbst überein.

Punkt (`.`)

> Gibt einen Platzhalter an.

Das `.`Metazeichen entspricht jedem einzelnen Zeichen außer einem Zeilenumbruch:

```
>>> re.search('foo.bar', 'fooxbar')
<_sre.SRE_Match object; span=(0, 7), match='fooxbar'>

>>> print(re.search('foo.bar', 'foobar'))
None
>>> print(re.search('foo.bar', 'foo\nbar'))
None
```

Als regulärer Ausdruck `foo.bar`bedeutet im Wesentlichen die Zeichen `'foo'`, dann ein beliebiges Zeichen außer Newline, dann die Zeichen `'bar'`. Die erste oben gezeigte Zeichenfolge, `'fooxbar'`, genau das Richtige, weil die `.`Metazeichen stimmt mit dem überein `'x'`.

Die zweite und dritte Saite passen nicht zusammen. Im letzten Fall steht zwar ein Zeichen dazwischen `'foo'` und `'bar'`, es ist ein Zeilenumbruch, und standardmäßig ist die `.` Metazeichen stimmt nicht mit einem Zeilenumbruch überein. Es gibt jedoch eine Möglichkeit, sie zu erzwingen `.` um einen Zeilenumbruch abzugleichen, was Sie am Ende dieses Tutorials erfahren werden.

`\w`
`\W`

Übereinstimmung basierend darauf, ob ein Zeichen ein Wortzeichen ist.

`\w` stimmt mit jedem alphanumerischen Wortzeichen überein. Wortzeichen sind Groß- und Kleinbuchstaben, Ziffern und der Unterstrich (_) Charakter, also `\w` ist im Wesentlichen eine Abkürzung für `[a-zA-Z0-9_]`:

```
>>> re.search('\w', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
>>> re.search('[a-zA-Z0-9_]', '#(.a$@&')
<_sre.SRE_Match object; span=(3, 4), match='a'>
```

In diesem Fall das erste Wortzeichen in der Zeichenfolge `'#(.a$@&'` ist `'a'`.

`\W` ist das Gegenteil. Es stimmt mit jedem Nichtwortzeichen überein und ist äquivalent zu `[^a-zA-Z0-9_]`:

```
>>> re.search('\W', 'a_1*3Qb')
<_sre.SRE_Match object; span=(3, 4), match='*'>
>>> re.search('[^a-zA-Z0-9_]', 'a_1*3Qb')
<_sre.SRE_Match object; span=(3, 4), match='*'>
```

Hier das erste Nichtwortzeichen in `'a_1*3!b'` ist `'*'`.

`\d`
`\D`

Übereinstimmung basierend darauf, ob ein Zeichen eine Dezimalziffer ist.

`\d` entspricht jedem Dezimalziffernzeichen. `\D` ist das Gegenteil. Es stimmt mit jedem Zeichen überein, *das keine* Dezimalziffer ist:

```
>>> re.search('\d', 'abc4def')
<_sre.SRE_Match object; span=(3, 4), match='4'>

>>> re.search('\D', '234Q678')
<_sre.SRE_Match object; span=(3, 4), match='Q'>
```

`\d` ist im Wesentlichen gleichbedeutend mit `[0-9]`, und `\D` ist äquivalent zu `[^0-9]`.

`\s`
`\S`

Übereinstimmung basierend darauf, ob ein Zeichen Leerzeichen darstellt.

`\s` passt zu jedem Leerzeichen:

```
>>> re.search('\s', 'foo\nbar baz')
<_sre.SRE_Match object; span=(3, 4), match='\n'>
```

Beachten Sie, dass im Gegensatz zum Punkt-Platzhalter-Metazeichen `\s`stimmt mit einem Zeilenumbruchzeichen überein.

`\S`ist das Gegenteil von `\s`. Es stimmt mit jedem Zeichen überein, *das kein* Leerzeichen ist:

```
>>> re.search('\S', '  \n foo  \n  ')
<_sre.SRE_Match object; span=(4, 5), match='f'>
```

Wieder, `\s`und `\S`Betrachten Sie einen Zeilenumbruch als Leerzeichen. Im obigen Beispiel ist das erste Nicht-Leerzeichen `'f'`.

Die Zeichenklassensequenzen `\w`, `\W`, `\d`, `\D`, `\s`, und `\S`kann auch in einer Zeichenklasse mit eckigen Klammern erscheinen:

```
>>> re.search('[\d\w\s]', '---3---')
<_sre.SRE_Match object; span=(3, 4), match='3'>
>>> re.search('[\d\w\s]', '---a---')
<_sre.SRE_Match object; span=(3, 4), match='a'>
>>> re.search('[\d\w\s]', '--- ---')
<_sre.SRE_Match object; span=(3, 4), match=' '>
```

In diesem Fall, `[\d\w\s]`stimmt mit jeder Ziffer, jedem Wort oder Leerzeichen überein. Und da `\w`beinhaltet `\d`, dieselbe Zeichenklasse könnte auch etwas kürzer als ausgedrückt werden `[\w\s]`.

Metazeichen entkommen

Gelegentlich werden Sie ein Metazeichen in Ihre Regex aufnehmen wollen, außer Sie möchten nicht, dass es seine spezielle Bedeutung trägt. Stattdessen möchten Sie, dass es sich selbst als wörtliches Zeichen darstellt.

Backslash (`\`)

> Entfernt die besondere Bedeutung eines Metazeichens.

Wie Sie gerade gesehen haben, kann der Backslash Sonderzeichenklassen wie Wörter, Ziffern und Leerzeichen einführen. Es gibt auch spezielle Metazeichenfolgen, sogenannte **Anker** , die mit einem umgekehrten Schrägstrich beginnen, über die Sie weiter unten mehr erfahren werden.

Wenn es keinem dieser Zwecke dient, maskiert der umgekehrte Schrägstrich **Metazeichen** . Ein Metazeichen, dem ein umgekehrter Schrägstrich vorangestellt ist, verliert seine besondere Bedeutung und entspricht stattdessen dem Literalzeichen. Betrachten Sie die folgenden Beispiele:

```
>>> re.search('.', 'foo.bar')

<_sre.SRE_Match object; span=(0, 1), match='f'>



>>> re.search('\.', 'foo.bar')

<_sre.SRE_Match object; span=(3, 4), match='.'>
```

In dem `<regex>`in **Zeile 1** der Punkt (`.`) fungiert als Platzhalter-Metazeichen, das mit dem ersten Zeichen in der Zeichenfolge übereinstimmt (`'f'`). Das `.`Charakter im `<regex>`in **Zeile 4** wird

durch einen umgekehrten Schrägstrich maskiert, es handelt sich also nicht um einen Platzhalter. Es wird wörtlich interpretiert und entspricht dem `'.'` am Index `3` des Suchstrings.

Die Verwendung von Backslashes für Escapezeichen kann unübersichtlich werden. Angenommen, Sie haben eine Zeichenfolge, die einen einzelnen umgekehrten Schrägstrich enthält:

```
>>> s = r'foo\bar'
>>> print(s)
foo\bar
```

Nehmen wir nun an, Sie möchten eine erstellen `<regex>` das wird mit dem Backslash dazwischen übereinstimmen `'foo'` und `'bar'`. Der umgekehrte Schrägstrich ist selbst ein Sonderzeichen in einer Regex. Um also einen wörtlichen umgekehrten Schrägstrich anzugeben, müssen Sie ihn mit einem anderen umgekehrten Schrägstrich maskieren. Wenn das der Fall ist, dann sollte Folgendes funktionieren:

```
>>> re.search('\\', s)
```

Nicht ganz. Das bekommen Sie, wenn Sie es versuchen:

```
>>> re.search('\\', s)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    re.search('\\', s)
  File "C:\Python36\lib\re.py", line 182, in search
    return _compile(pattern, flags).search(string)
  File "C:\Python36\lib\re.py", line 301, in _compile
    p = sre_compile.compile(pattern, flags)
  File "C:\Python36\lib\sre_compile.py", line 562, in compile
    p = sre_parse.parse(p, flags)
  File "C:\Python36\lib\sre_parse.py", line 848, in parse
    source = Tokenizer(str)
  File "C:\Python36\lib\sre_parse.py", line 231, in __init__
    self.__next()
  File "C:\Python36\lib\sre_parse.py", line 245, in __next
    self.string, len(self.string) - 1) from None
sre_constants.error: bad escape (end of pattern) at position 0
```

Hoppla. Was ist passiert?

Das Problem dabei ist, dass das Backslash-Escape zweimal erfolgt, zuerst vom Python-Interpreter für das String-Literal und dann noch einmal vom Regex-Parser für die empfangene Regex.

Hier ist die Abfolge der Ereignisse:

1. Der Python-Interpreter ist der erste, der das String-Literal verarbeitet `'\\'`. Es interpretiert das als einen Backslash mit Escapezeichen und übergibt nur einen einzigen Backslash an `re.search()`.
2. Der Regex-Parser erhält nur einen einzelnen umgekehrten Schrägstrich, der keine aussagekräftige Regex ist, sodass der unangenehme Fehler auftritt.

Es gibt zwei Möglichkeiten, dies zu umgehen. Erstens können Sie *beide* Backslashes im ursprünglichen String-Literal maskieren:

```
>>> re.search('\\\\', s)
<_sre.SRE_Match object; span=(3, 4), match='\\'>
```

Dadurch passiert Folgendes:

1. Der Dolmetscher sieht `'\\\\'` als ein Paar Backslashes mit Escapezeichen. Es reduziert jedes Paar auf einen einzelnen umgekehrten Schrägstrich und übergibt `'\\'` zum Regex-Parser.
2. Der Regex-Parser sieht dann `\\` als ein entgangener Backslash. Als ein `<regex>`, das einem einzelnen umgekehrten Schrägstrich entspricht. Sie können anhand des Übereinstimmungsobjekts sehen, dass es mit dem umgekehrten Schrägstrich am Index übereinstimmt `3` in `s` wie beabsichtigt. Es ist umständlich, aber es funktioniert.

Der zweite und wahrscheinlich sauberere Weg, dies zu handhaben, besteht darin, die zu spezifizieren `<regex>` Verwenden einer rohen Zeichenfolge :

```
>>> re.search(r'\\', s)
<_sre.SRE_Match object; span=(3, 4), match='\\'>
```

Dadurch wird das Escape auf der Interpreterebene unterdrückt. Die Saite `'\\'` wird unverändert an den Regex-Parser übergeben, der wie gewünscht wieder einen maskierten Backslash sieht.

Es hat sich bewährt, einen unformatierten String zu verwenden, um eine Regex in Python anzugeben, wenn sie Backslashes enthält.

Anker

Anker sind Matches ohne Breite. Sie stimmen mit keinen tatsächlichen Zeichen in der Suchzeichenfolge überein, und sie verbrauchen während der Analyse nichts von der Suchzeichenfolge. Stattdessen schreibt ein Anker eine bestimmte Stelle in der Suchzeichenfolge vor, an der eine Übereinstimmung auftreten muss.

`^`

`\A`

> Verankern Sie ein Streichholz am Anfang `<string>`.

Wenn der Regex-Parser trifft `^` oder `\A`, muss die aktuelle Position des Parsers am Anfang der Suchzeichenfolge stehen, damit er eine Übereinstimmung findet.

Mit anderen Worten, Regex `^foo` schreibt das vor `'foo'` muss nicht an irgendeiner alten Stelle im Suchstring stehen, sondern am Anfang:

```
>>> re.search('^foo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('^foo', 'barfoo'))
None
```

`\A` funktioniert ähnlich:

```
>>> re.search('\Afoo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('\Afoo', 'barfoo'))
None
```

`^` und `\A` verhalten sich etwas unterschiedlich voneinander in `MULTILINE` Modus. Sie erfahren mehr darüber `MULTILINE` Modus weiter unten im Abschnitt über Flags .

`$`
`\Z`

Verankern Sie ein Streichholz am Ende von `<string>`.

Wenn der Regex-Parser trifft `$`oder `\Z`, muss die aktuelle Position des Parsers am Ende der Suchzeichenfolge stehen, damit er eine Übereinstimmung findet. Was auch immer vorangeht `$`oder `\Z`muss das Ende des Suchstrings bilden:

```
>>> re.search('bar$', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> print(re.search('bar$', 'barfoo'))
None

>>> re.search('bar\Z', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
>>> print(re.search('bar\Z', 'barfoo'))
None
```

Als Sonderfall `$`(aber nicht `\Z`) passt auch kurz vor einem einzelnen Zeilenumbruch am Ende der Suchzeichenfolge:

```
>>> re.search('bar$', 'foobar\n')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
```

In diesem Beispiel `'bar'`steht technisch gesehen nicht am Ende der Suchzeichenfolge, da ihm ein zusätzliches Zeilenumbruchzeichen folgt. Aber der Regex-Parser lässt es gleiten und nennt es trotzdem eine Übereinstimmung. Diese Ausnahme gilt nicht für `\Z`.

`$`und `\Z`verhalten sich etwas unterschiedlich voneinander in `MULTILINE`Modus. finden Sie im Abschnitt unten über Flaggen Weitere Informationen `MULTILINE`Modus.

`\b`

Verankert eine Übereinstimmung an einer Wortgrenze.

`\b`behauptet, dass die aktuelle Position des Regex-Parsers am Anfang oder am Ende eines Wortes sein muss. Ein Wort besteht aus einer Folge von alphanumerischen Zeichen oder Unterstrichen (`[a-zA-Z0-9_]`), das gleiche wie für die `\w`Zeichenklasse:

```
>>> re.search(r'\bbar', 'foo bar')

<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> re.search(r'\bbar', 'foo.bar')

<_sre.SRE_Match object; span=(4, 7), match='bar'>



>>> print(re.search(r'\bbar', 'foobar'))

None
```

```
>>> re.search(r'foo\b', 'foo bar')

<_sre.SRE_Match object; span=(0, 3), match='foo'>

>>> re.search(r'foo\b', 'foo.bar')

<_sre.SRE_Match object; span=(0, 3), match='foo'>



>>> print(re.search(r'foo\b', 'foobar'))

None
```

In den obigen Beispielen findet eine Übereinstimmung in den **Zeilen 1 und 3** statt, da am Anfang von eine Wortgrenze steht `'bar'`. Dies ist in **Zeile 6** , daher schlägt die Übereinstimmung dort fehl.

In ähnlicher Weise gibt es Übereinstimmungen in **Zeilen 9 und 11** da am Ende von eine Wortgrenze existiert `'foo'`, aber nicht in **Zeile 14** .

Verwendung der `\b`Anker an beiden Enden des `<regex>`bewirkt, dass es übereinstimmt, wenn es in der Suchzeichenfolge als ganzes Wort vorhanden ist:

```
>>> re.search(r'\bbar\b', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search(r'\bbar\b', 'foo(bar)baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> print(re.search(r'\bbar\b', 'foobarbaz'))
None
```

Dies ist ein weiterer Fall, in dem es sich lohnt, das anzugeben `<regex>`als Rohstring, wie es die obigen Beispiele getan haben.

Da `'\b'`eine Escape-Sequenz sowohl für String-Literale als auch für reguläre Ausdrücke in Python ist, müsste jede obige Verwendung mit einem doppelten Escapezeichen versehen werden `'\\ b'`wenn Sie keine rohen Zeichenfolgen verwendet haben. Das wäre kein Weltuntergang, aber rohe Saiten sind aufgeräumter.

`\B`

> Verankert eine Übereinstimmung an einer Stelle, die keine Wortgrenze ist.

`\B`macht das Gegenteil von `\b`. Es behauptet, dass die aktuelle Position des Regex-Parsers *nicht* am Anfang oder Ende eines Wortes sein darf:

```
>>> print(re.search(r'\Bfoo\B', 'foo'))

None

>>> print(re.search(r'\Bfoo\B', '.foo.'))

None
```

```
>>> re.search(r'\Bfoo\B', 'barfoobaz')

<_sre.SRE_Match object; span=(3, 6), match='foo'>
```

In diesem Fall findet eine Übereinstimmung in **Zeile 7** da am Anfang oder Ende von keine Wortgrenze existiert `'foo'` im Suchstring `'barfoobaz'`.

Quantifizierer

Ein **Quantifizierer** -Metazeichen folgt unmittelbar auf einen Teil von a `<regex>` und gibt an, wie oft dieser Teil auftreten muss, damit die Übereinstimmung erfolgreich ist.

`*`

Stimmt mit null oder mehr Wiederholungen der vorhergehenden Regex überein.

Zum Beispiel, `a*` stimmt mit null oder mehr überein `'a'` Figuren. Das heißt, es würde eine leere Zeichenfolge finden, `'a'`, `'aa'`, `'aaa'`, usw.

Betrachten Sie diese Beispiele:

```
>>> re.search('foo-*bar', 'foobar')                    # Zero dashes

<_sre.SRE_Match object; span=(0, 6), match='foobar'>

>>> re.search('foo-*bar', 'foo-bar')                   # One dash

<_sre.SRE_Match object; span=(0, 7), match='foo-bar'>

>>> re.search('foo-*bar', 'foo--bar')                  # Two dashes

<_sre.SRE_Match object; span=(0, 8), match='foo--bar'>
```

Auf **Zeile 1** gibt es Null `'-'` Zeichen dazwischen `'foo'` und `'bar'`. In **Zeile 3** gibt es einen und in **Zeile 5** zwei. Die Metazeichenfolge `-*` Übereinstimmungen in allen drei Fällen.

Sie werden wahrscheinlich auf die Regex stoßen `.*` irgendwann in einem Python-Programm. Dies stimmt mit null oder mehr Vorkommen eines beliebigen Zeichens überein. Mit anderen Worten, es passt im Wesentlichen jede Zeichenfolge bis zu einem Zeilenumbruch. (Denken Sie daran, dass die `.` Platzhalter-Metazeichen stimmt nicht mit einem Zeilenumbruch überein.)

In diesem Beispiel `.*` passt alles dazwischen `'foo'` und `'bar'`:

```
>>> re.search('foo.*bar', '# foo $qux@grault % bar #')
<_sre.SRE_Match object; span=(2, 23), match='foo $qux@grault % bar'>
```

Hast du das bemerkt `span=` und `match=` Informationen, die im Match-Objekt enthalten sind?

Bisher haben die regulären Ausdrücke in den Beispielen, die Sie gesehen haben, Übereinstimmungen mit vorhersagbarer Länge angegeben. Sobald Sie anfangen, Quantifizierer wie zu verwenden `*`, kann die Anzahl der übereinstimmenden Zeichen sehr variabel sein, und die Informationen im Übereinstimmungsobjekt werden nützlicher.

Im nächsten Tutorial der Reihe erfahren Sie mehr darüber, wie Sie auf die in einem Übereinstimmungsobjekt gespeicherten Informationen zugreifen.

`+`

Stimmt mit einer oder mehreren Wiederholungen der vorhergehenden Regex überein.

Dies ist ähnlich wie `*`, aber die quantifizierte Regex muss mindestens einmal vorkommen:

```
>>> print(re.search('foo-+bar', 'foobar'))              # Zero dashes

None

>>> re.search('foo-+bar', 'foo-bar')                    # One dash

<_sre.SRE_Match object; span=(0, 7), match='foo-bar'>

>>> re.search('foo-+bar', 'foo--bar')                   # Two dashes

<_sre.SRE_Match object; span=(0, 8), match='foo--bar'>
```

Denken Sie daran von oben `foo-*bar`passte zur Saite `'foobar'`weil die `*`Metazeichen lässt kein Vorkommen von zu `'-'`. Das `+`Metazeichen hingegen erfordert mindestens ein Vorkommen von `'-'`. keine Übereinstimmung auf **Zeile 1** in diesem Fall

`?`

Stimmt mit null oder einer Wiederholung der vorhergehenden Regex überein.

Auch dies ist ähnlich wie `*`und `+`, aber in diesem Fall gibt es nur dann eine Übereinstimmung, wenn die vorhergehende Regex einmal oder gar nicht vorkommt:

```
>>> re.search('foo-?bar', 'foobar')                     # Zero dashes

<_sre.SRE_Match object; span=(0, 6), match='foobar'>

>>> re.search('foo-?bar', 'foo-bar')                    # One dash

<_sre.SRE_Match object; span=(0, 7), match='foo-bar'>

>>> print(re.search('foo-?bar', 'foo--bar'))            # Two dashes

None
```

In diesem Beispiel gibt es Übereinstimmungen in den **Zeilen 1 und 3** . Aber in **Zeile 5** , wo es zwei gibt `'-'`Zeichen, schlägt die Übereinstimmung fehl.

Hier sind einige weitere Beispiele, die die Verwendung aller drei Quantifizierer-Metazeichen zeigen:

```
>>> re.match('foo[1-9]*bar', 'foobar')
<_sre.SRE_Match object; span=(0, 6), match='foobar'>
>>> re.match('foo[1-9]*bar', 'foo42bar')
<_sre.SRE_Match object; span=(0, 8), match='foo42bar'>

>>> print(re.match('foo[1-9]+bar', 'foobar'))
None
>>> re.match('foo[1-9]+bar', 'foo42bar')
<_sre.SRE_Match object; span=(0, 8), match='foo42bar'>
```

```
>>> re.match('foo[1-9]?bar', 'foobar')
<_sre.SRE_Match object; span=(0, 6), match='foobar'>
>>> print(re.match('foo[1-9]?bar', 'foo42bar'))
None
```

Diesmal ist die quantifizierte Regex die Zeichenklasse `[1-9]`statt des einfachen Charakters `'-'`.

`*?`
`+?`
`??`

Die nicht gierigen (oder faulen) Versionen des `*`, `+`, und `?`Quantifizierer.

Bei alleiniger Verwendung die Metazeichen des Quantifizierers `*`, `+`, und `?`sind alle **gierig** , was bedeutet, dass sie die längstmögliche Übereinstimmung produzieren. Betrachten Sie dieses Beispiel:

```
>>> re.search('<.*>', '%<foo> <bar> <baz>%')
<_sre.SRE_Match object; span=(1, 18), match='<foo> <bar> <baz>'>
```

Die Regex `<.*>`effektiv bedeutet:

- EIN `'<'`Charakter
- Dann eine beliebige Folge von Zeichen
- Dann ein `'>'`Charakter

Aber welches `'>'`Charakter? Es gibt drei Möglichkeiten:

1. Der gleich danach `'foo'`
2. Der gleich danach `'bar'`
3. Der gleich danach `'baz'`

Seit der `*`Metazeichen ist gierig, es diktiert die längstmögliche Übereinstimmung, die alles bis einschließlich einschließt `'>'`Charakter, der folgt `'baz'`. Sie können anhand des Übereinstimmungsobjekts sehen, dass dies die erzeugte Übereinstimmung ist.

Wenn Sie stattdessen eine möglichst kurze Übereinstimmung wünschen, verwenden Sie die nicht gierige Metazeichenfolge `*?`:

```
>>> re.search('<.*?>', '%<foo> <bar> <baz>%')
<_sre.SRE_Match object; span=(1, 6), match='<foo>'>
```

In diesem Fall endet das Spiel mit dem `'>'`Charakter folgt `'foo'`.

Hinweis: Das Gleiche könnten Sie auch mit der Regex erreichen `<[^>]*>`, was bedeutet:

- EIN `'<'`Charakter
- Dann eine beliebige Folge von Zeichen außer `'>'`
- Dann ein `'>'`Charakter

Dies ist die einzige Option, die bei einigen älteren Parsern verfügbar ist, die Lazy Quantifiers nicht unterstützen. Glücklicherweise ist das beim Regex-Parser in Python nicht der Fall `re`Modul.

Es gibt faule Versionen der `+`und `?`auch Quantoren:

```
>>> re.search('<.+>', '%<foo> <bar> <baz>%')
```

```
<_sre.SRE_Match object; span=(1, 18), match='<foo> <bar> <baz>'>

>>> re.search('<.+?>', '%<foo> <bar> <baz>%')

<_sre.SRE_Match object; span=(1, 6), match='<foo>'>



>>> re.search('ba?', 'baaaa')

<_sre.SRE_Match object; span=(0, 2), match='ba'>

>>> re.search('ba??', 'baaaa')

<_sre.SRE_Match object; span=(0, 1), match='b'>
```

Die ersten beiden Beispiele in **Zeilen 1 und 3** ähneln den oben gezeigten Beispielen, verwenden aber nur `+`und `+?`Anstatt von `*`und `*?`.

Die letzten Beispiele in den **Zeilen 6 und 8** sind etwas anders. Im Allgemeinen ist die `?` Metazeichen stimmt mit null oder einem Vorkommen der vorangehenden Regex überein. Die gierige Version, `?`, stimmt mit einem Vorkommen überein, also `ba?`Streichhölzer `'b'`gefolgt von einer Single `'a'`. Die nicht gierige Version, `??`, stimmt mit null Vorkommen überein, also `ba??` passt einfach `'b'`.

`{m}`

> Passt genau `m`Wiederholungen der vorhergehenden Regex.

Dies ist ähnlich wie `*`oder `+`, aber es gibt genau an, wie oft die vorangehende Regex vorkommen muss, damit eine Übereinstimmung erfolgreich ist:

```
>>> print(re.search('x-{3}x', 'x--x'))               # Two dashes
None

>>> re.search('x-{3}x', 'x---x')                     # Three dashes
<_sre.SRE_Match object; span=(0, 5), match='x---x'>

>>> print(re.search('x-{3}x', 'x----x'))             # Four dashes
None
```

Hier, `x-{3}x`Streichhölzer `'x'`, gefolgt von genau drei Instanzen der `'-'`Charakter, gefolgt von einem anderen `'x'`. Die Übereinstimmung schlägt fehl, wenn weniger oder mehr als drei Bindestriche dazwischen liegen `'x'`Figuren.

`{m,n}`

> Stimmt mit einer beliebigen Anzahl von Wiederholungen der vorangehenden Regex überein `m`zu `n`, inklusive.

Im folgenden Beispiel wird die quantifiziert `<regex>`ist `-{2,4}`. Die Übereinstimmung ist erfolgreich, wenn zwei, drei oder vier Bindestriche dazwischen liegen `'x'`Zeichen, schlägt aber sonst fehl:

```
>>> for i in range(1, 6):
```

```
...     s = f"x{'-' * i}x"
...     print(f'{i}  {s:10}', re.search('x-{2,4}x', s))
...
1  x-x        None
2  x--x       <_sre.SRE_Match object; span=(0, 4), match='x--x'>
3  x---x      <_sre.SRE_Match object; span=(0, 5), match='x---x'>
4  x----x     <_sre.SRE_Match object; span=(0, 6), match='x----x'>
5  x-----x    None
```

Weglassen `m`impliziert eine Untergrenze von `0`, und weglassen `n`impliziert eine unbegrenzte Obergrenze:

Regulären Ausdruck	Streichhölzer	Identisch mit
`<regex>{,n}`	Beliebig viele Wiederholungen von `<regex>`weniger als oder gleich `n`	`<regex>{0,n}`
`<regex>{m,}`	Beliebig viele Wiederholungen von `<regex>`größer als oder gleich wie `m`	`----`
`<regex>{,}`	Beliebig viele Wiederholungen von `<regex>`	`<regex>{0,}` `<regex>*`

Wenn Sie alles weglassen `m`, `n`, und das Komma, dann fungieren die geschweiften Klammern nicht mehr als Metazeichen. `{}`stimmt nur mit der wörtlichen Zeichenfolge überein `'{}'`:

```
>>> re.search('x{}y', 'x{}y')
<_sre.SRE_Match object; span=(0, 4), match='x{}y'>
```

Tatsächlich muss eine Sequenz mit geschweiften Klammern, um eine besondere Bedeutung zu haben, in eines der folgenden Muster passen `m`und `n`sind nichtnegative ganze Zahlen:

- `{m,n}`
- `{m,}`
- `{,n}`
- `{,}`

Ansonsten passt es wörtlich:

```
>>> re.search('x{foo}y', 'x{foo}y')
<_sre.SRE_Match object; span=(0, 7), match='x{foo}y'>
>>> re.search('x{a:b}y', 'x{a:b}y')
<_sre.SRE_Match object; span=(0, 7), match='x{a:b}y'>
>>> re.search('x{1,3,5}y', 'x{1,3,5}y')
<_sre.SRE_Match object; span=(0, 9), match='x{1,3,5}y'>
>>> re.search('x{foo,bar}y', 'x{foo,bar}y')
<_sre.SRE_Match object; span=(0, 11), match='x{foo,bar}y'>
```

Später in diesem Lernprogramm, wenn Sie mehr über die erfahren `DEBUG`Flagge, Sie werden sehen, wie Sie dies bestätigen können.

`{m,n}?`

Die nicht-gierige (faule) Version von `{m,n}`.

`{m,n}`wird mit so vielen Zeichen wie möglich übereinstimmen, und `{m,n}?`passt so wenig wie möglich:

```
>>> re.search('a{3,5}', 'aaaaaaaa')
<_sre.SRE_Match object; span=(0, 5), match='aaaaa'>

>>> re.search('a{3,5}?', 'aaaaaaaa')
<_sre.SRE_Match object; span=(0, 3), match='aaa'>
```

In diesem Fall, `a{3,5}`erzeugt die längstmögliche Übereinstimmung, also passt es zu fünf `'a'`Figuren. `a{3,5}?`erzeugt die kürzeste Übereinstimmung, also drei Übereinstimmungen.

Gruppierungskonstrukte und Rückverweise

Gruppierungskonstrukte unterteilen einen regulären Ausdruck in Python in Teilausdrücke oder Gruppen. Dies dient zwei Zwecken:

1. **Gruppierung:** Eine Gruppe repräsentiert eine einzelne syntaktische Einheit. Zusätzliche Metazeichen gelten für die gesamte Gruppe als Einheit.
2. **Erfassen:** Einige Gruppierungskonstrukte erfassen auch den Teil der Suchzeichenfolge, der mit dem Teilausdruck in der Gruppe übereinstimmt. Sie können erfasste Übereinstimmungen später über verschiedene Mechanismen abrufen.

Hier sehen Sie, wie das Gruppieren und Erfassen funktioniert.

`(<regex>)`

Definiert einen Teilausdruck oder eine Gruppe.

Dies ist das grundlegendste Gruppierungskonstrukt. Eine Regex in Klammern entspricht einfach dem Inhalt der Klammern:

```
>>> re.search('(bar)', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> re.search('bar', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
```

Als regulärer Ausdruck `(bar)`stimmt mit der Zeichenfolge überein `'bar'`, das gleiche wie die Regex `bar`würde ohne die Klammern.

Eine Gruppe als Einheit behandeln

Ein Quantifizierer-Metazeichen, das einer Gruppe folgt, wirkt auf den gesamten Unterausdruck, der in der Gruppe als eine einzelne Einheit angegeben ist.

Das folgende Beispiel stimmt beispielsweise mit einem oder mehreren Vorkommen der Zeichenfolge überein `'bar'`:

```
>>> re.search('(bar)+', 'foo bar baz')
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search('(bar)+', 'foo barbar baz')
<_sre.SRE_Match object; span=(4, 10), match='barbar'>
>>> re.search('(bar)+', 'foo barbarbarbar baz')
<_sre.SRE_Match object; span=(4, 16), match='barbarbarbar'>
```

Hier ist eine Aufschlüsselung des Unterschieds zwischen den beiden regulären Ausdrücken mit und ohne gruppierende Klammern:

Regex	Deutung	Streichhölzer	Beispiele

`bar+`	Das +metacharacter gilt nur für das Zeichen `'r'`.	`'ba'` gefolgt von einem oder mehreren Vorkommen von `'r'`	`'bar'`
			`'barr'`
			`'barrr'`

`(bar)+`	Das +metazeichen gilt für die gesamte Zeichenfolge `'bar'`.	Ein oder mehrere Vorkommen von `'bar'`	`'bar'`
			`'barbar'`
			`'barbarbar'`

Schauen Sie sich nun ein komplizierteres Beispiel an. Die Regex `(ba[rz]){2,4}(qux)?` Streichhölzer `2`zu `4`Vorkommen von beidem `'bar'`oder `'baz'`, optional gefolgt von `'qux'`:

```
>>> re.search('(ba[rz]){2,4}(qux)?', 'bazbarbazqux')
<_sre.SRE_Match object; span=(0, 12), match='bazbarbazqux'>
>>> re.search('(ba[rz]){2,4}(qux)?', 'barbar')
<_sre.SRE_Match object; span=(0, 6), match='barbar'>
```

Das folgende Beispiel zeigt, dass Sie Gruppierungsklammern verschachteln können:

```
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoobar')
<_sre.SRE_Match object; span=(0, 9), match='foofoobar'>
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoobar123')
<_sre.SRE_Match object; span=(0, 12), match='foofoobar123'>
>>> re.search('(foo(bar)?)+(\d\d\d)?', 'foofoo123')
<_sre.SRE_Match object; span=(0, 9), match='foofoo123'>
```

Die Regex `(foo(bar)?)+(\d\d\d)?`ist ziemlich aufwendig, also lasst es uns in kleinere Stücke zerlegen:

Regex	Streichhölzer
`foo(bar)?`	`'foo'`optional gefolgt von `'bar'`
`(foo(bar)?)+`	Ein oder mehrere Vorkommen der oben genannten Punkte
`\d\d\d`	Drei Dezimalziffern
`(\d\d\d)?`	Kein oder ein Vorkommen der oben genannten Punkte

Wenn Sie alles aneinanderreihen, erhalten Sie: mindestens ein Vorkommen von `'foo'`optional gefolgt von `'bar'`, alle optional gefolgt von drei Dezimalziffern.

Wie Sie sehen können, können Sie in Python sehr komplizierte reguläre Ausdrücke erstellen, indem Sie Klammern zum Gruppieren verwenden.

Erfassungsgruppen

Das Gruppieren ist nicht der einzige nützliche Zweck, dem Gruppierungskonstrukte dienen. Die meisten (aber nicht alle) Gruppierungskonstrukte erfassen auch den Teil der Suchzeichenfolge, der mit der Gruppe übereinstimmt. Sie können den erfassten Teil abrufen oder später auf verschiedene Weise darauf verweisen.

Denken Sie an das Übereinstimmungsobjekt that `re.search()`kehrt zurück? Für ein Match-Objekt sind zwei Methoden definiert, die Zugriff auf erfasste Gruppen bieten: `.groups()`und `.group()`.

`m.groups()`

> Gibt ein Tupel zurück, das alle erfassten Gruppen aus einer Regex-Übereinstimmung enthält.

Betrachten Sie dieses Beispiel:

```
>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> m
<_sre.SRE_Match object; span=(0, 12), match='foo:quux:baz'>
```

Jeder der drei `(\w+)`expressions entspricht einer Folge von Wortzeichen. Die vollständige Regex `(\w+),(\w+),(\w+)`zerlegt die Suchzeichenfolge in drei durch Kommas getrennte Token.

Weil die `(\w+)`Ausdrücke gruppierende Klammern verwenden, werden die entsprechenden übereinstimmenden Token **erfasst** . Um auf die erfassten Übereinstimmungen zuzugreifen, können Sie verwenden `.groups()`zurückgibt Tupel , das alle erfassten Übereinstimmungen der Reihe nach enthält:

```
>>> m.groups()
('foo', 'quux', 'baz')
```

Beachten Sie, dass das Tupel die Token enthält, aber nicht die Kommas, die in der Suchzeichenfolge erschienen sind. Das liegt daran, dass die Wortzeichen, aus denen die Token bestehen, innerhalb der gruppierenden Klammern stehen, die Kommas jedoch nicht. Die Kommas, die Sie zwischen den zurückgegebenen Token sehen, sind die Standardtrennzeichen, die zum Trennen von Werten in einem Tupel verwendet werden.

`m.group(<n>)`

Gibt eine Zeichenfolge zurück, die die enthält `<n>`thgefangenes Spiel.

Mit einem Argument, `.group()`gibt eine einzelne erfasste Übereinstimmung zurück. Beachten Sie, dass die Argumente einsbasiert und nicht nullbasiert sind. So, `m.group(1)`bezieht sich auf das erste erfasste Spiel, `m.group(2)`zum zweiten usw.:

```
>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'quux', 'baz')

>>> m.group(1)
'foo'
>>> m.group(2)
'quux'
>>> m.group(3)
'baz'
```

Da die Nummerierung der erfassten Spiele einsbasiert ist und es keine Gruppe mit der Nummer Null gibt, `m.group(0)`hat eine besondere Bedeutung:

```
>>> m.group(0)
'foo,quux,baz'
>>> m.group()
'foo,quux,baz'
```

`m.group(0)`gibt die gesamte Übereinstimmung zurück, und `m.group()`macht das gleiche.

`m.group(<n1>, <n2>, ...)`

Gibt ein Tupel zurück, das die angegebenen erfassten Übereinstimmungen enthält.

Mit mehreren Argumenten `.group()`gibt ein Tupel zurück, das die angegebenen erfassten Übereinstimmungen in der angegebenen Reihenfolge enthält:

```
>>> m.groups()
('foo', 'quux', 'baz')

>>> m.group(2, 3)
('quux', 'baz')
>>> m.group(3, 2, 1)
('baz', 'quux', 'foo')
```

Dies ist nur eine bequeme Abkürzung. Sie könnten das Tupel von Übereinstimmungen stattdessen selbst erstellen:

```
>>> m.group(3, 2, 1)
('baz', 'qux', 'foo')
>>> (m.group(3), m.group(2), m.group(1))
('baz', 'qux', 'foo')
```

Die beiden gezeigten Anweisungen sind funktional gleichwertig.

Rückverweise

Sie können eine zuvor erfasste Gruppe später innerhalb derselben Regex abgleichen, indem Sie eine spezielle Metazeichenfolge verwenden, die als **Rückverweis** .

`\<n>`

> Gleicht den Inhalt einer zuvor erfassten Gruppe ab.

Innerhalb einer Regex in Python ist die Sequenz `\<n>`, wo `<n>`ist eine ganze Zahl von `1`zu `99`, entspricht dem Inhalt der `<n>`thgefangene Gruppe.

Hier ist eine Regex, die mit einem Wort übereinstimmt, gefolgt von einem Komma, gefolgt von demselben Wort noch einmal:

```
>>> regex = r'(\w+),\1'



>>> m = re.search(regex, 'foo,foo')

>>> m

<_sre.SRE_Match object; span=(0, 7), match='foo,foo'>

>>> m.group(1)

'foo'



>>> m = re.search(regex, 'qux,qux')

>>> m

<_sre.SRE_Match object; span=(0, 7), match='qux,qux'>
```

```
>>> m.group(1)

'qux'



>>> m = re.search(regex, 'foo,qux')

>>> print(m)

None
```

Im ersten Beispiel, in **Zeile 3** , `(\w+)`stimmt mit der ersten Instanz der Zeichenfolge überein `'foo'`und speichert sie als erste erfasste Gruppe. Das Komma passt wörtlich. Dann `\1`ist ein Rückverweis auf die erste erfasste Gruppe und Übereinstimmungen `'foo'`wieder. Das zweite Beispiel in **Zeile 9** ist identisch, außer dass die `(\w+)`Streichhölzer `'qux'`stattdessen.

Das letzte Beispiel in **Zeile 15** hat keine Übereinstimmung, weil das, was vor dem Komma steht, nicht dasselbe ist wie das, was danach kommt, also das `\1`Rückverweis passt nicht.

Hinweis: Jedes Mal, wenn Sie in Python einen regulären Ausdruck mit einer nummerierten Rückwärtsreferenz verwenden, ist es eine gute Idee, ihn als Rohzeichenfolge anzugeben. Andernfalls könnte der Interpreter die Rückwärtsreferenz mit einem Oktalwert .

Betrachten Sie dieses Beispiel:

```
>>> print(re.search('([a-z])#\1', 'd#d'))
None
```

Die Regex `([a-z])#\1`entspricht einem Kleinbuchstaben, gefolgt von `'#'`, gefolgt von demselben Kleinbuchstaben. Die Zeichenfolge ist in diesem Fall `'d#d'`, was passen sollte. Aber der Abgleich schlägt fehl, weil Python die Rückwärtsreferenz falsch interpretiert `\1`als das Zeichen, dessen Oktalwert eins ist:

```
>>> oct(ord('\1'))
'0o1'
```

Sie erhalten die richtige Übereinstimmung, wenn Sie die Regex als Rohzeichenfolge angeben:

```
>>> re.search(r'([a-z])#\1', 'd#d')
<_sre.SRE_Match object; span=(0, 3), match='d#d'>
```

Denken Sie daran, immer dann einen unformatierten String zu verwenden, wenn Ihre Regex eine Metazeichenfolge enthält, die einen umgekehrten Schrägstrich enthält.

Nummerierte Rückverweise sind wie die Argumente auf einsbasiert `.group()`. Nur die ersten neunundneunzig erfassten Gruppen sind per Rückverweis zugänglich. Der Dolmetscher wird prüfen `\100`als die `'@'`Zeichen, dessen Oktalwert 100 ist.

Andere Gruppierungskonstrukte

Das `(<regex>)`Die oben gezeigte Metazeichensequenz ist die einfachste Möglichkeit, eine Gruppierung innerhalb einer Regex in Python durchzuführen. Der nächste Abschnitt stellt Ihnen

einige erweiterte Gruppierungskonstrukte vor, mit denen Sie optimieren können, wann und wie die Gruppierung erfolgt.

`(?P<name><regex>)`

Erstellt eine benannte erfasste Gruppe.

Diese Metazeichenfolge ähnelt Gruppierungsklammern darin, dass sie eine Gruppenübereinstimmung erstellt `<regex>`auf die über das Match-Objekt oder eine nachfolgende Rückwärtsreferenz zugegriffen werden kann. Der Unterschied besteht in diesem Fall darin, dass Sie die übereinstimmende Gruppe über ihre angegebene Symbolik referenzieren `<name>`statt durch seine Nummer.

Sie haben vorhin dieses Beispiel mit drei nummerierten eroberten Gruppen gesehen `1`, `2`, und `3`:

```
>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'quux', 'baz')

>>> m.group(1, 2, 3)
('foo', 'quux', 'baz')
```

Das Folgende macht effektiv dasselbe, außer dass die Gruppen die symbolischen Namen haben `w1`, `w2`, und `w3`:

```
>>> m = re.search('(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'quux', 'baz')
```

Sie können auf diese erfassten Gruppen mit ihren symbolischen Namen verweisen:

```
>>> m.group('w1')
'foo'
>>> m.group('w3')
'baz'
>>> m.group('w1', 'w2', 'w3')
('foo', 'quux', 'baz')
```

Sie können weiterhin auf Gruppen mit symbolischen Namen per Nummer zugreifen, wenn Sie möchten:

```
>>> m = re.search('(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'foo,quux,baz')

>>> m.group('w1')
'foo'
>>> m.group(1)
'foo'

>>> m.group('w1', 'w2', 'w3')
('foo', 'quux', 'baz')
>>> m.group(1, 2, 3)
('foo', 'quux', 'baz')
```

Irgendein `<name>`angegeben mit diesem Konstrukt muss den Regeln für einen Python-Bezeichner , und each `<name>`kann nur einmal pro Regex vorkommen.

`(?P=<name>)`

Gleicht den Inhalt einer zuvor erfassten benannten Gruppe ab.

Das `(?P=<name>)` Metazeichenfolge ist eine Rückwärtsreferenz, ähnlich wie `\<n>`, außer dass es sich auf eine benannte Gruppe und nicht auf eine nummerierte Gruppe bezieht.

Hier ist noch einmal das Beispiel von oben, das eine nummerierte Rückwärtsreferenz verwendet, um ein Wort abzugleichen, gefolgt von einem Komma, gefolgt von demselben Wort noch einmal:

```
>>> m = re.search(r'(\w+),\1', 'foo,foo')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='foo,foo'>
>>> m.group(1)
'foo'
```

Der folgende Code macht dasselbe, indem er stattdessen eine benannte Gruppe und eine Rückwärtsreferenz verwendet:

```
>>> m = re.search(r'(?P<word>\w+),(?P=word)', 'foo,foo')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='foo,foo'>
>>> m.group('word')
'foo'
```

`(?P=<word>\w+)` Streichhölzer `'foo'` und speichert sie als erfasste Gruppe mit dem Namen `word`. Auch hier stimmt das Komma wörtlich überein. Dann `(?P=word)` ist ein Rückverweis auf die benannte Erfassung und Übereinstimmungen `'foo'` wieder.

Hinweis: Die spitzen Klammern (`<` und `>`) sind erforderlich `name` beim Erstellen einer benannten Gruppe, aber nicht, wenn später darauf verwiesen wird, entweder durch Rückverweis oder durch `.group()`:

```
>>> m = re.match(r'(?P<num>\d+)\.(?P=num)', '135.135')
>>> m
<_sre.SRE_Match object; span=(0, 7), match='135.135'>

>>> m.group('num')
'135'
```

Hier, `(?P<num>\d+)` erstellt die erfasste Gruppe. Aber die entsprechende Rückreferenz ist `(?P=num)` ohne die spitzen klammern.

`(?:<regex>)`

Erstellt eine nicht erfassende Gruppe.

`(?:<regex>)` ist genauso wie `(<regex>)`, dass es mit den angegebenen übereinstimmt `<regex>`. Aber `(?:<regex>)` erfasst die Übereinstimmung nicht für einen späteren Abruf:

```
>>> m = re.search('(\w+),(?:\w+),(\w+)', 'foo,quux,baz')
>>> m.groups()
('foo', 'baz')

>>> m.group(1)
'foo'
>>> m.group(2)
'baz'
```

In diesem Beispiel das mittlere Wort `'quux'` befindet sich in nicht erfassenden Klammern, fehlt also im Tupel der erfassten Gruppen. Es ist weder vom Match-Objekt abrufbar, noch wäre es durch Rückwärtsreferenz referenzierbar.

Warum möchten Sie eine Gruppe definieren, aber nicht erfassen?

Denken Sie daran, dass der Regex-Parser die behandeln wird `<regex>` innerhalb gruppierender Klammern als eine Einheit. Möglicherweise haben Sie eine Situation, in der Sie diese Gruppierungsfunktion benötigen, aber später nichts mit dem Wert tun müssen, sodass Sie ihn nicht wirklich erfassen müssen. Wenn Sie die nicht erfassende Gruppierung verwenden, wird das Tupel der erfassten Gruppen nicht mit Werten überladen, die Sie eigentlich nicht behalten müssen.

Außerdem braucht es einige Zeit und Speicher, um eine Gruppe zu erfassen. Wenn der Code, der den Abgleich durchführt, viele Male ausgeführt wird und Sie keine Gruppen erfassen, die Sie später nicht verwenden werden, sehen Sie möglicherweise einen leichten Leistungsvorteil.

`(?(<n>)<yes-regex>|<no-regex>)`
`(?(<name>)<yes-regex>|<no-regex>)`

> Gibt eine bedingte Übereinstimmung an.

Eine bedingte Übereinstimmung stimmt mit einem von zwei angegebenen regulären Ausdrücken überein, je nachdem, ob die angegebene Gruppe vorhanden ist:

- `(?(<n>)<yes-regex>|<no-regex>)` Spiele gegen `<yes-regex>` wenn eine Gruppe nummeriert `<n>` existiert. Ansonsten passt es gegen `<no-regex>`.
- `(?(<name>)<yes-regex>|<no-regex>)` Spiele gegen `<yes-regex>` wenn eine Gruppe namens `<name>` existiert. Ansonsten passt es gegen `<no-regex>`.

Bedingte Übereinstimmungen werden anhand eines Beispiels besser veranschaulicht. Betrachten Sie diese Regex:

```
regex = r'^(###)?foo(?(1)bar|baz)'
```

Hier sind die Teile dieser Regex aufgeschlüsselt mit einer Erklärung:

1. `^(###)?` gibt an, dass die Suchzeichenfolge optional mit beginnt `'###'`. Wenn dies der Fall ist, werden die Gruppierungsklammern herum gesetzt ###erstellt eine nummerierte Gruppe `1`. Andernfalls existiert keine solche Gruppe.
2. Die nächste Portion, `foo`, stimmt buchstäblich mit der Zeichenfolge überein `'foo'`.
3. Zuletzt, `(?(1)bar|baz)` Spiele gegen `'bar'` wenn Gruppe `1` besteht und `'baz'` wenn nicht.

Die folgenden Codeblöcke demonstrieren die Verwendung der obigen Regex in mehreren verschiedenen Python-Codeschnipseln:

Beispiel 1:

```
>>> re.search(regex, '###foobar')
<_sre.SRE_Match object; span=(0, 9), match='###foobar'>
```

Die Suchzeichenfolge `'###foobar'` beginnt mit `'###'`, also erstellt der Parser eine Gruppe mit der Nummer `1`. Die bedingte Übereinstimmung ist dann gegen `'bar'`, was passt.

Beispiel 2:

```
>>> print(re.search(regex, '###foobaz'))
None
```

Die Suchzeichenfolge `'###foobaz'` beginnt mit `'###'`, also erstellt der Parser eine Gruppe mit der Nummer `1`. Die bedingte Übereinstimmung ist dann gegen `'bar'`, was nicht passt.

Beispiel 3:

```
>>> print(re.search(regex, 'foobar'))
None
```

Die Suchzeichenfolge `'foobar'` beginnt nicht mit `'###'`, also ist keine Gruppe nummeriert `1`. Die bedingte Übereinstimmung ist dann gegen `'baz'`, was nicht passt.

Beispiel 4:

```
>>> re.search(regex, 'foobaz')
<_sre.SRE_Match object; span=(0, 6), match='foobaz'>
```

Die Suchzeichenfolge `'foobaz'` beginnt nicht mit `'###'`, also ist keine Gruppe nummeriert `1`. Die bedingte Übereinstimmung ist dann gegen `'baz'`, was passt.

Hier ist eine weitere bedingte Übereinstimmung mit einer benannten Gruppe anstelle einer nummerierten Gruppe:

```
>>> regex = r'^(?P<ch>\W)?foo(?(ch)(?P=ch)|)$'
```

Diese Regex stimmt mit der Zeichenfolge überein `'foo'`, dem ein einzelnes Nichtwortzeichen vorangestellt ist und dem dasselbe Nichtwortzeichen folgt, oder die Zeichenfolge `'foo'` von selbst.

Lassen Sie uns dies noch einmal in Stücke zerlegen:

Regex	Streichhölzer
`^`	Der Anfang der Zeichenfolge
`(?P<ch>\W)`	Ein einzelnes Nicht-Wort-Zeichen, das in einer Gruppe mit dem Namen erfasst ist `ch`
`(?P<ch>\W)?`	Kein oder ein Vorkommen der oben genannten Punkte
`foo`	Die wörtliche Zeichenfolge `'foo'`
`(?(ch)(?P=ch)|)`	Der Inhalt der benannten Gruppe `ch` wenn es existiert, oder die leere Zeichenfolge, wenn es nicht existiert
`$`	Das Ende der Zeichenfolge

Wenn ein Nicht-Wort-Zeichen vorangeht `'foo'`, dann erstellt der Parser eine Gruppe mit dem Namen `ch` die dieses Zeichen enthält. Die bedingte Übereinstimmung stimmt dann mit überein `<yes-regex>`, welches ist `(?P=ch)`, das gleiche Zeichen noch einmal. Das heißt, es muss auch das gleiche Zeichen folgen `'foo'` damit das gesamte Spiel gelingt.

Wenn `'foo'` kein Nichtwortzeichen vorangestellt ist, erstellt der Parser keine Gruppe `ch`. `<no-regex>` ist der leere String, was bedeutet, dass nichts folgen darf `'foo'` damit das gesamte Spiel

gelingt. Seit ^und $Verankern Sie die gesamte Regex, die Zeichenfolge muss gleich sein `'foo'`exakt.

Hier sind einige Beispiele für Suchen mit dieser Regex in Python-Code:

```
>>> re.search(regex, 'foo')

<_sre.SRE_Match object; span=(0, 3), match='foo'>

>>> re.search(regex, '#foo#')

<_sre.SRE_Match object; span=(0, 5), match='#foo#'>

>>> re.search(regex, '@foo@')

<_sre.SRE_Match object; span=(0, 5), match='@foo@'>


>>> print(re.search(regex, '#foo'))

None

>>> print(re.search(regex, 'foo@'))

None

>>> print(re.search(regex, '#foo@'))

None

>>> print(re.search(regex, '@foo#'))

None
```

Auf **Zeile 1** , `'foo'`ist von selbst. In **Zeilen 3 und 5** steht dasselbe Nicht-Wort-Zeichen davor und danach `'foo'`. Wie angekündigt, sind diese Übereinstimmungen erfolgreich.

In den verbleibenden Fällen schlagen die Übereinstimmungen fehl.

Bedingte reguläre Ausdrücke in Python sind ziemlich esoterisch und schwierig zu bearbeiten. Wenn Sie jemals einen Grund finden, einen zu verwenden, könnten Sie wahrscheinlich dasselbe Ziel mit mehreren separaten erreichen `re.search()`Anrufe, und Ihr Code wäre weniger kompliziert zu lesen und zu verstehen.

Lookahead- und Lookbehind-Zusicherungen

Lookahead- und **Lookbehin-** Assertionen bestimmen den Erfolg oder Misserfolg einer Regex-Übereinstimmung in Python basierend darauf, was direkt hinter (links) oder vor (rechts) der aktuellen Position des Parsers in der Suchzeichenfolge liegt.

Lookahead- und Lookbehind-Assertionen sind wie Anker Aussagen mit einer Breite von null, sodass sie nichts von der Suchzeichenfolge verbrauchen. Auch wenn sie Klammern enthalten und Gruppierungen durchführen, erfassen sie nicht, was sie finden.

`(?=<lookahead_regex>)`

Erstellt eine positive Lookahead-Assertion.

`(?=<lookahead_regex>)`behauptet, dass das, was auf die aktuelle Position des Regex-Parsers folgt, übereinstimmen muss `<lookahead_regex>`:

```
>>> re.search('foo(?=[a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

Die Lookahead-Behauptung `(?=[a-z])`präzisiert das Folgende `'foo'`muss ein Kleinbuchstabe sein. In diesem Fall ist es der Charakter `'b'`, also wird eine Übereinstimmung gefunden.

Im nächsten Beispiel hingegen schlägt die Vorausschau fehl. Das nächste Zeichen danach `'foo'`ist `'1'`, also gibt es keine Übereinstimmung:

```
>>> print(re.search('foo(?=[a-z])', 'foo123'))
None
```

Das Einzigartige an einem Lookahead ist der Teil der Suchzeichenfolge, der übereinstimmt `<lookahead_regex>`wird nicht verbraucht und ist nicht Teil des zurückgegebenen Übereinstimmungsobjekts.

Schauen Sie sich noch einmal das erste Beispiel an:

```
>>> re.search('foo(?=[a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

Der Regex-Parser schaut nur auf die voraus `'b'`dass folgt `'foo'`geht aber noch nicht darüber hinweg. Das kann man sagen `'b'`wird nicht als Teil der Übereinstimmung betrachtet, da das Übereinstimmungsobjekt angezeigt wird `match='foo'`.

Vergleichen Sie das mit einem ähnlichen Beispiel, das Gruppierungsklammern ohne Lookahead verwendet:

```
>>> re.search('foo([a-z])', 'foobar')
<_sre.SRE_Match object; span=(0, 4), match='foob'>
```

Diesmal verbraucht die Regex die `'b'`, und es wird Teil des eventuellen Spiels.

Hier ist ein weiteres Beispiel, das veranschaulicht, wie sich ein Lookahead von einer herkömmlichen Regex in Python unterscheidet:

```
>>> m = re.search('foo(?=[a-z])(?P<ch>.)', 'foobar')

>>> m.group('ch')

'b'



>>> m = re.search('foo([a-z])(?P<ch>.)', 'foobar')

>>> m.group('ch')

'a'
```

Bei der ersten Suche in **Zeile 1** geht der Parser wie folgt vor:

1. Der erste Teil der Regex, `foo`, Streichhölzer und verbraucht `'foo'` aus der Suchzeichenfolge `'foobar'`.
2. Die nächste Portion, `(?=[a-z])`, ist ein passendes Lookahead `'b'`, aber der Parser geht nicht über die hinaus `'b'`.
3. Zuletzt, `(?P<ch>.)` stimmt mit dem nächsten verfügbaren Einzelzeichen überein, das ist `'b'`, und erfasst es in einer Gruppe mit dem Namen `ch`.

Das `m.group('ch')` Anruf bestätigt, dass die genannte Gruppe `ch` enthält `'b'`.

Vergleichen Sie das mit der Suche in **Zeile 5** , die kein Lookahead enthält:

1. Wie im ersten Beispiel, der erste Teil der Regex, `foo`, Streichhölzer und verbraucht `'foo'` aus der Suchzeichenfolge `'foobar'`.
2. Die nächste Portion, `([a-z])`, Streichhölzer und verbraucht `'b'`, und der Parser rückt vor `'b'`.
3. Zuletzt, `(?P<ch>.)` stimmt mit dem nächsten verfügbaren Einzelzeichen überein, das jetzt ist `'a'`.

`m.group('ch')` bestätigt, dass in diesem Fall die genannte Gruppe `ch` enthält `'a'`.

`(?!<lookahead_regex>)`

> Erstellt eine negative Lookahead-Assertion.

`(?!<lookahead_regex>)` behauptet, dass das, was auf die aktuelle Position des Regex-Parsers folgt, *nicht* übereinstimmen `<lookahead_regex>`.

Hier sind die positiven Lookahead-Beispiele, die Sie zuvor gesehen haben, zusammen mit ihren negativen Lookahead-Gegenstücken:

```
>>> re.search('foo(?=[a-z])', 'foobar')

<_sre.SRE_Match object; span=(0, 3), match='foo'>

>>> print(re.search('foo(?![a-z])', 'foobar'))

None



>>> print(re.search('foo(?=[a-z])', 'foo123'))

None

>>> re.search('foo(?![a-z])', 'foo123')

<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

Die negativen Lookahead-Assertionen in den **Zeilen 3 und 8** legen das Folgende fest `'foo'` sollte kein Kleinbuchstabe sein. Dies schlägt in **Zeile 3** aber in **Zeile 8** . Dies ist das Gegenteil von dem, was mit den entsprechenden positiven Lookahead-Assertionen passiert ist.

Wie bei einem positiven Lookahead ist das, was mit einem negativen Lookahead übereinstimmt, nicht Teil des zurückgegebenen Match-Objekts und wird nicht verbraucht.

`(?<=<lookbehind_regex>)`

Erstellt eine positive Lookbehind-Assertion.

`(?<=<lookbehind_regex>)`behauptet, dass das, was vor der aktuellen Position des Regex-Parsers steht, übereinstimmen muss `<lookbehind_regex>`.

Im folgenden Beispiel gibt die Lookbehind-Assertion dies an `'foo'`muss vorangehen `'bar'`:

```
>>> re.search('(?<=foo)bar', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
```

Dies ist hier der Fall, sodass das Match erfolgreich ist. Wie bei Lookahead-Assertionen wird der Teil der Suchzeichenfolge, der mit dem Lookbehind übereinstimmt, nicht Teil der endgültigen Übereinstimmung.

Das nächste Beispiel passt nicht, weil das Lookbehind dies erfordert `'qux'`vorausgehen `'bar'`:

```
>>> print(re.search('(?<=qux)bar', 'foobar'))
None
```

Es gibt eine Einschränkung für Lookbehind-Assertionen, die nicht für Lookahead-Assertionen gilt. Das `<lookbehind_regex>`in einer Lookbehind-Assertion muss eine Übereinstimmung fester Länge angeben.

Folgendes ist beispielsweise nicht zulässig, da die Länge der Zeichenfolge übereinstimmt mit `a+`ist unbestimmt:

```
>>> re.search('(?<=a+)def', 'aaadef')
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    re.search('(?<=a+)def', 'aaadef')
  File "C:\Python36\lib\re.py", line 182, in search
    return _compile(pattern, flags).search(string)
  File "C:\Python36\lib\re.py", line 301, in _compile
    p = sre_compile.compile(pattern, flags)
  File "C:\Python36\lib\sre_compile.py", line 566, in compile
    code = _code(p, flags)
  File "C:\Python36\lib\sre_compile.py", line 551, in _code
    _compile(code, p.data, flags)
  File "C:\Python36\lib\sre_compile.py", line 160, in _compile
    raise error("look-behind requires fixed-width pattern")
sre_constants.error: look-behind requires fixed-width pattern
```

Das ist aber in Ordnung:

```
>>> re.search('(?<=a{3})def', 'aaadef')
<_sre.SRE_Match object; span=(3, 6), match='def'>
```

Alles was passt `a{3}`wird eine feste Länge von drei haben, also `a{3}`in einer Lookbehind-Assertion gültig ist.

`(?<!--<lookbehind_regex-->)`

Erstellt eine negative Lookbehind-Assertion.

`(?<!--<lookbehind_regex-->)`behauptet, dass das, was vor der aktuellen Position des Regex-Parsers steht, *nicht* übereinstimmen `<lookbehind_regex>`:

```
>>> print(re.search('(?<!foo)bar', 'foobar'))
None

>>> re.search('(?<!qux)bar', 'foobar')
<_sre.SRE_Match object; span=(3, 6), match='bar'>
```

Wie bei der positiven Lookbehind-Behauptung, `<lookbehind_regex>`muss eine Übereinstimmung mit fester Länge angeben.

Verschiedene Metazeichen

Es gibt noch ein paar weitere Metazeichenfolgen zu behandeln. Dies sind streunende Metazeichen, die offensichtlich in keine der bereits besprochenen Kategorien fallen.

`(?#...)`

> Gibt einen Kommentar an.

Der Regex-Parser ignoriert alles, was in der Sequenz enthalten ist `(?#...)`:

```
>>> re.search('bar(?#This is a comment) *baz', 'foo bar baz qux')
<_sre.SRE_Match object; span=(4, 11), match='bar baz'>
```

Auf diese Weise können Sie in Python eine Dokumentation innerhalb einer Regex angeben, was besonders nützlich sein kann, wenn die Regex besonders lang ist.

Vertikaler Balken oder Rohr (`|`)

> Gibt eine Reihe von Alternativen an, die abgeglichen werden sollen.

Ein Ausdruck des Formulars $<regex_1>|<regex_2>|...|<regex_n>$entspricht höchstens einer der angegebenen $<regex_i>$Ausdrücke:

```
>>> re.search('foo|bar|baz', 'bar')
<_sre.SRE_Match object; span=(0, 3), match='bar'>

>>> re.search('foo|bar|baz', 'baz')
<_sre.SRE_Match object; span=(0, 3), match='baz'>

>>> print(re.search('foo|bar|baz', 'quux'))
None
```

Hier, `foo|bar|baz`passt zu jedem von `'foo'`, `'bar'`, oder `'baz'`. Sie können eine beliebige Anzahl von regulären Ausdrücken mit trennen `|`.

Abwechslung ist nicht gierig. Der Regex-Parser betrachtet die durch getrennten Ausdrücke `|`in der Reihenfolge von links nach rechts und gibt die erste gefundene Übereinstimmung zurück. Die verbleibenden Ausdrücke werden nicht getestet, auch wenn einer von ihnen eine längere Übereinstimmung ergeben würde:

```
>>> re.search('foo', 'foograult')

<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

```
>>> re.search('grault', 'foograult')

<_sre.SRE_Match object; span=(3, 9), match='grault'>



>>> re.search('foo|grault', 'foograult')

<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

angegebene Muster **Zeile 6** , `'foo|grault'`, würde auf beiden übereinstimmen `'foo'`oder `'grault'`. Das zurückgegebene Match ist `'foo'`weil das beim Scannen von links nach rechts zuerst erscheint, obwohl `'grault'`wäre ein längeres Spiel.

Sie können Alternation, Gruppierung und beliebige andere Metazeichen kombinieren, um den gewünschten Grad an Komplexität zu erreichen. Im folgenden Beispiel `(foo|bar|baz)+`bedeutet eine Folge von einer oder mehreren der Zeichenfolgen `'foo'`, `'bar'`, oder `'baz'`:

```
>>> re.search('(foo|bar|baz)+', 'foofoofoo')
<_sre.SRE_Match object; span=(0, 9), match='foofoofoo'>
>>> re.search('(foo|bar|baz)+', 'bazbazbazbaz')
<_sre.SRE_Match object; span=(0, 12), match='bazbazbazbaz'>
>>> re.search('(foo|bar|baz)+', 'barbazfoo')
<_sre.SRE_Match object; span=(0, 9), match='barbazfoo'>
```

Im nächsten Beispiel `([0-9]+|[a-f]+)`bedeutet eine Folge von einem oder mehreren Dezimalziffern oder eine Folge von einem oder mehreren der Zeichen `'a-f'`:

```
>>> re.search('([0-9]+|[a-f]+)', '456')
<_sre.SRE_Match object; span=(0, 3), match='456'>
>>> re.search('([0-9]+|[a-f]+)', 'ffda')
<_sre.SRE_Match object; span=(0, 4), match='ffda'>
```

Mit all den Metazeichen, die die `re`Modul unterstützt, der Himmel ist praktisch die Grenze.

Das war's Leute!

Damit ist unsere Tour durch die von Python unterstützten Regex-Metazeichen abgeschlossen `re`Modul. (Eigentlich nicht ganz – es gibt noch ein paar Nachzügler, die Sie weiter unten in der Diskussion über Flaggen kennenlernen werden.)

Es ist eine Menge zu verdauen, aber sobald Sie sich mit der Regex-Syntax in Python vertraut gemacht haben, ist die Komplexität des Mustervergleichs, die Sie durchführen können, nahezu grenzenlos. Diese Tools sind sehr praktisch, wenn Sie Code schreiben, um Textdaten zu verarbeiten.

Wenn Sie mit Regexes noch nicht vertraut sind und mehr Übung im Umgang mit ihnen wünschen oder wenn Sie eine Anwendung entwickeln, die eine Regex verwendet, und Sie diese interaktiv testen möchten, besuchen Sie die Regular Expressions 101 Website Es ist wirklich cool!

Modifizierter Abgleich regulärer Ausdrücke mit Flags

Die meisten Funktionen in der `re`Modul nehmen Sie ein optionales `<flags>`Streit. Dazu gehört die Funktion, mit der Sie jetzt sehr vertraut sind, `re.search()`.

```
re.search(<regex>, <string>, <flags>)
```

Durchsucht eine Zeichenfolge nach einer Regex-Übereinstimmung und wendet den angegebenen Modifikator an `<flags>`.

Flags ändern das Regex-Parsing-Verhalten, sodass Sie Ihren Musterabgleich noch weiter verfeinern können.

Unterstützte Flags für reguläre Ausdrücke

Die folgende Tabelle fasst die verfügbaren Flags kurz zusammen. Alle Flaggen außer `re.DEBUG`haben einen kurzen, aus einem Buchstaben bestehenden Namen und auch einen längeren, aus ganzen Wörtern bestehenden Namen:

Kurzer Name	Langer Name	Wirkung
`re.I`	`re.IGNORECASE`	Bewirkt, dass bei der Suche nach alphabetischen Zeichen die Groß-/Kleinschreibung nicht beachtet wird
`re.M`	`re.MULTILINE`	Bewirkt, dass Anfangs- und Endanker von Zeichenfolgen mit eingebetteten Zeilenumbrüchen übereinstimmen
`re.S`	`re.DOTALL`	Bewirkt, dass das Punkt-Metazeichen einem Zeilenumbruch entspricht
`re.X`	`re.VERBOSE`	Ermöglicht das Einfügen von Leerzeichen und Kommentaren in einen regulären Ausdruck
`----`	`re.DEBUG`	Bewirkt, dass der Regex-Parser Debugging-Informationen in der Konsole anzeigt
`re.A`	`re.ASCII`	Gibt die ASCII-Codierung für die Zeichenklassifizierung an
`re.U`	`re.UNICODE`	Gibt die Unicode-Codierung für die Zeichenklassifizierung an
`re.L`	`re.LOCALE`	Gibt die Codierung für die Zeichenklassifizierung basierend auf dem aktuellen Gebietsschema an

In den folgenden Abschnitten wird ausführlicher beschrieben, wie sich diese Flags auf das Übereinstimmungsverhalten auswirken.

`re.I`
`re.IGNORECASE`

Macht übereinstimmende Groß-/Kleinschreibung unempfindlich.

Wann `IGNORECASE`aktiv ist, wird beim Zeichenabgleich die Groß-/Kleinschreibung nicht beachtet:

```
>>> re.search('a+', 'aaaAAA')

<_sre.SRE_Match object; span=(0, 3), match='aaa'>

>>> re.search('A+', 'aaaAAA')

<_sre.SRE_Match object; span=(3, 6), match='AAA'>



>>> re.search('a+', 'aaaAAA', re.I)

<_sre.SRE_Match object; span=(0, 6), match='aaaAAA'>
```

```
>>> re.search('A+', 'aaaAAA', re.IGNORECASE)

<_sre.SRE_Match object; span=(0, 6), match='aaaAAA'>
```

Bei der Suche in **Zeile 1** , `a+`stimmt nur mit den ersten drei Zeichen von überein `'aaaAAA'`. Ebenso in **Zeile 3** , `A+`stimmt nur mit den letzten drei Zeichen überein. Aber bei den nachfolgenden Suchen ignoriert der Parser die Groß-/Kleinschreibung, also beides `a+`und `A+`Übereinstimmung mit der gesamten Zeichenfolge.

`IGNORECASE`wirkt sich auch auf den alphabetischen Abgleich mit Zeichenklassen aus:

```
>>> re.search('[a-z]+', 'aBcDeF')
<_sre.SRE_Match object; span=(0, 1), match='a'>
>>> re.search('[a-z]+', 'aBcDeF', re.I)
<_sre.SRE_Match object; span=(0, 6), match='aBcDeF'>
```

Wenn die Groß-/Kleinschreibung signifikant ist, der längste Teil von `'aBcDeF'`das `[a-z]+`Übereinstimmungen ist nur der Anfang `'a'`. Angabe `re.I`macht die Groß- und Kleinschreibung bei der Suche unempfindlich, so `[a-z]+`stimmt mit der gesamten Zeichenfolge überein.

`re.M`
`re.MULTILINE`

> Bewirkt, dass Anfangs- und Endanker von Zeichenfolgen bei eingebetteten Zeilenumbrüchen übereinstimmen.

Standardmäßig ist die `^`(Anfang der Zeichenkette) und `$`(String-of-String)-Anker stimmen nur am Anfang und am Ende des Suchstrings überein:

```
>>> s = 'foo\nbar\nbaz'

>>> re.search('^foo', s)
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> print(re.search('^bar', s))
None
>>> print(re.search('^baz', s))
None

>>> print(re.search('foo$', s))
None
>>> print(re.search('bar$', s))
None
>>> re.search('baz$', s)
<_sre.SRE_Match object; span=(8, 11), match='baz'>
```

In diesem Fall, obwohl die Suchzeichenfolge `'foo\nbar\nbaz'`enthält nur eingebettete Newline-Zeichen `'foo'`Übereinstimmungen, wenn sie am Anfang der Zeichenfolge verankert sind, und nur `'baz'`passt, wenn am Ende verankert.

Wenn ein String jedoch Zeilenumbrüche eingebettet hat, können Sie sich vorstellen, dass er aus mehreren internen Zeilen besteht. In diesem Fall, wenn die `MULTILINE`Flag gesetzt ist, die `^`und `$`Anker-Metazeichen stimmen auch mit internen Zeilen überein:

- `^`Übereinstimmungen am Anfang der Zeichenfolge oder am Anfang einer beliebigen Zeile innerhalb der Zeichenfolge (d. h. unmittelbar nach einem Zeilenumbruch).

- **$**Übereinstimmungen am Ende der Zeichenfolge oder am Ende einer beliebigen Zeile innerhalb der Zeichenfolge (unmittelbar vor einem Zeilenumbruch).

Das Folgende sind die gleichen Suchanfragen wie oben gezeigt:

```
>>> s = 'foo\nbar\nbaz'
>>> print(s)
foo
bar
baz

>>> re.search('^foo', s, re.MULTILINE)
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search('^bar', s, re.MULTILINE)
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search('^baz', s, re.MULTILINE)
<_sre.SRE_Match object; span=(8, 11), match='baz'>

>>> re.search('foo$', s, re.M)
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search('bar$', s, re.M)
<_sre.SRE_Match object; span=(4, 7), match='bar'>
>>> re.search('baz$', s, re.M)
<_sre.SRE_Match object; span=(8, 11), match='baz'>
```

In der Schnur `'foo\nbar\nbaz'`, alle drei `'foo'`, `'bar'`, und `'baz'`treten entweder am Anfang oder Ende der Zeichenfolge oder am Anfang oder Ende einer Zeile innerhalb der Zeichenfolge auf. Mit dem `MULTILINE`Flaggensatz, alle drei passen zusammen, wenn sie mit einem von beiden verankert sind `^`oder `$`.

Hinweis: Die `MULTILINE`Flag ändert nur die `^`und `$`Anker auf diese Weise. Es hat keine Auswirkungen auf die `\A`und `\Z`Anker:

```
>>> s = 'foo\nbar\nbaz'



>>> re.search('^bar', s, re.MULTILINE)

<_sre.SRE_Match object; span=(4, 7), match='bar'>

>>> re.search('bar$', s, re.MULTILINE)

<_sre.SRE_Match object; span=(4, 7), match='bar'>



>>> print(re.search('\Abar', s, re.MULTILINE))

None

>>> print(re.search('bar\Z', s, re.MULTILINE))

None
```

In **Zeilen 3 und 5** wird die `^`und `$`Anker diktieren das `'bar'`muss am Anfang und am Ende einer Zeile stehen. Angabe der `MULTILINE`Flagge macht diese Spiele erfolgreich.

Die Beispiele in **Zeilen 8 und 10** verwenden die `\A`und `\Z`stattdessen Fahnen. Sie können sehen, dass diese Übereinstimmungen auch mit dem fehlschlagen `MULTILINE`Flagge in Kraft.

`re.S`
`re.DOTALL`

Verursacht den Punkt (`.`) Metazeichen, um einem Zeilenumbruch zu entsprechen.

Denken Sie daran, dass das Punkt-Metazeichen standardmäßig mit jedem Zeichen außer dem Zeilenumbruchzeichen übereinstimmt. Das `DOTALL`Flag hebt diese Einschränkung auf:

```
>>> print(re.search('foo.bar', 'foo\nbar'))

None

>>> re.search('foo.bar', 'foo\nbar', re.DOTALL)

<_sre.SRE_Match object; span=(0, 7), match='foo\nbar'>

>>> re.search('foo.bar', 'foo\nbar', re.S)

<_sre.SRE_Match object; span=(0, 7), match='foo\nbar'>
```

In diesem Beispiel stimmt das Punkt-Metazeichen in **Zeile 1** nicht mit dem Zeilenumbruch überein `'foo\nbar'`. In **den Zeilen 3 und 5** , `DOTALL`ist in Kraft, also stimmt der Punkt mit dem Zeilenumbruch überein. Beachten Sie, dass der Kurzname der `DOTALL`Flagge ist `re.S`, nicht `re.D`wie Du vielleicht erwartest.

`re.X`
`re.VERBOSE`

Ermöglicht das Einfügen von Leerzeichen und Kommentaren in eine Regex.

Das `VERBOSE`flag gibt einige spezielle Verhaltensweisen an:

- Der Regex-Parser ignoriert alle Leerzeichen, es sei denn, sie befinden sich innerhalb einer Zeichenklasse oder werden mit einem Backslash maskiert.
- Wenn die Regex eine enthält #Zeichen, das nicht in einer Zeichenklasse enthalten ist oder mit einem Backslash maskiert ist, dann ignoriert der Parser es und alle Zeichen rechts davon.

Was nützt das? Es ermöglicht Ihnen, eine Regex in Python zu formatieren, damit sie besser lesbar und selbstdokumentierend ist.

Hier ist ein Beispiel, das zeigt, wie Sie dies verwenden können. Angenommen, Sie möchten Telefonnummern mit dem folgenden Format parsen:

- Optionale dreistellige Vorwahl in Klammern
- Optionales Leerzeichen
- Dreistelliges Präfix
- Trennzeichen (entweder `'-'`oder `'.'`)
- Vierstellige Zeilennummer

Die folgende Regex macht den Trick:

```
>>> regex = r'^(\(\d{3}\))?\s*\d{3}[-.]\d{4}$'

>>> re.search(regex, '414.9229')
<_sre.SRE_Match object; span=(0, 8), match='414.9229'>
>>> re.search(regex, '414-9229')
<_sre.SRE_Match object; span=(0, 8), match='414-9229'>
>>> re.search(regex, '(712)414-9229')
<_sre.SRE_Match object; span=(0, 13), match='(712)414-9229'>
>>> re.search(regex, '(712) 414-9229')
<_sre.SRE_Match object; span=(0, 14), match='(712) 414-9229'>
```

Aber `r'^(\(\d{3}\))?\s*\d{3}[-.]\d{4}$'` ist ein Augenschmaus, nicht wahr? Verwendung der `VERBOSE`flag, können Sie dieselbe Regex stattdessen wie folgt in Python schreiben:

```
>>> regex = r'''^               # Start of string
...             (\(\d{3}\))?    # Optional area code
...             \s*             # Optional whitespace
...             \d{3}           # Three-digit prefix
...             [-.]            # Separator character
...             \d{4}           # Four-digit line number
...             $               # Anchor at end of string
...             '''

>>> re.search(regex, '414.9229', re.VERBOSE)
<_sre.SRE_Match object; span=(0, 8), match='414.9229'>
>>> re.search(regex, '414-9229', re.VERBOSE)
<_sre.SRE_Match object; span=(0, 8), match='414-9229'>
>>> re.search(regex, '(712)414-9229', re.X)
<_sre.SRE_Match object; span=(0, 13), match='(712)414-9229'>
>>> re.search(regex, '(712) 414-9229', re.X)
<_sre.SRE_Match object; span=(0, 14), match='(712) 414-9229'>
```

Das `re.search()`Die Aufrufe sind die gleichen wie die oben gezeigten, sodass Sie sehen können, dass diese Regex genauso funktioniert wie die zuvor angegebene. Aber es ist auf den ersten Blick weniger schwer zu verstehen.

Beachten Sie, dass dreifache Anführungszeichen es besonders bequem machen, eingebettete Zeilenumbrüche einzuschließen, die als ignorierte Leerzeichen gelten `VERBOSE`Modus.

Bei Verwendung der `VERBOSE`flag, achten Sie auf Leerzeichen, die Sie beabsichtigen, von Bedeutung zu sein. Betrachten Sie diese Beispiele:

```
>>> re.search('foo bar', 'foo bar')

<_sre.SRE_Match object; span=(0, 7), match='foo bar'>



>>> print(re.search('foo bar', 'foo bar', re.VERBOSE))

None



>>> re.search('foo\ bar', 'foo bar', re.VERBOSE)

<_sre.SRE_Match object; span=(0, 7), match='foo bar'>
```

```
>>> re.search('foo[ ]bar', 'foo bar', re.VERBOSE)

<_sre.SRE_Match object; span=(0, 7), match='foo bar'>
```

Nach allem, was Sie bisher gesehen haben, fragen Sie sich vielleicht, warum in **Zeile 4** die Regex `foo bar`stimmt nicht mit der Zeichenfolge überein `'foo bar'`. Es tut nicht, weil die `VERBOSE`flag bewirkt, dass der Parser das Leerzeichen ignoriert.

Um diese Übereinstimmung wie erwartet zu erzielen, maskieren Sie das Leerzeichen mit einem umgekehrten Schrägstrich oder fügen Sie es in eine Zeichenklasse ein, wie in den **Zeilen 7 und 9** .

Wie bei der `DOTALL`Flagge, beachten Sie, dass die `VERBOSE`flag hat einen nicht intuitiven Kurznamen: `re.X`, nicht `re.V`.

`re.DEBUG`

Zeigt Debugging-Informationen an.

Das `DEBUG`Flag bewirkt, dass der Regex-Parser in Python Debugging-Informationen über den Parsing-Prozess in der Konsole anzeigt:

```
>>> re.search('foo.bar', 'fooxbar', re.DEBUG)
LITERAL 102
LITERAL 111
LITERAL 111
ANY None
LITERAL 98
LITERAL 97
LITERAL 114
<_sre.SRE_Match object; span=(0, 7), match='fooxbar'>
```

Wenn der Parser anzeigt `LITERAL nnn`In der Debugging-Ausgabe zeigt es den ASCII-Code eines Literalzeichens in der Regex. In diesem Fall sind die Literalzeichen `'f'`, `'o'`, `'o'`und `'b'`, `'a'`, `'r'`.

Hier ist ein komplizierteres Beispiel. Dies ist die Telefonnummern-Regex, die in der Diskussion über die angezeigt wird `VERBOSE`Flagge früher:

```
>>> regex = r'^(\(\d{3}\))?\s*\d{3}[-.]\d{4}$'

>>> re.search(regex, '414.9229', re.DEBUG)
AT AT_BEGINNING
MAX_REPEAT 0 1
  SUBPATTERN 1 0 0
    LITERAL 40
    MAX_REPEAT 3 3
      IN
        CATEGORY CATEGORY_DIGIT
    LITERAL 41
MAX_REPEAT 0 MAXREPEAT
  IN
    CATEGORY CATEGORY_SPACE
MAX_REPEAT 3 3
  IN
    CATEGORY CATEGORY_DIGIT
IN
  LITERAL 45
  LITERAL 46
```

```
MAX_REPEAT 4 4
  IN
    CATEGORY CATEGORY_DIGIT
AT AT_END
<_sre.SRE_Match object; span=(0, 8), match='414.9229'>
```

Das sieht nach vielen esoterischen Informationen aus, die Sie nie brauchen würden, aber sie können nützlich sein. Sehen Sie sich den Deep Dive unten für eine praktische Anwendung an.

> **Deep Dive: Debuggen der Analyse regulärer Ausdrücke**
>
> Wie Sie von oben wissen, die Metazeichenfolge `{m,n}`gibt eine bestimmte Anzahl von Wiederholungen an. Es passt überall hin `m`zu `n`Wiederholungen des Vorhergehenden:

```
>>> re.search('x[123]{2,4}y', 'x222y')
<_sre.SRE_Match object; span=(0, 5), match='x222y'>
```

Sie können dies mit überprüfen `DEBUG`Flagge:

```
>>> re.search('x[123]{2,4}y', 'x222y', re.DEBUG)
LITERAL 120
MAX_REPEAT 2 4
  IN
    LITERAL 49
    LITERAL 50
    LITERAL 51
LITERAL 121
<_sre.SRE_Match object; span=(0, 5), match='x222y'>
```

`MAX_REPEAT 2 4`bestätigt, dass der Regex-Parser die Metazeichenfolge erkennt `{2,4}`und interpretiert es als Bereichsquantifizierer.

Aber wie bereits erwähnt, wenn ein Paar geschweifter Klammern in einer Regex in Python etwas anderes als eine gültige Zahl oder einen numerischen Bereich enthält, verliert es seine besondere Bedeutung.

Sie können dies auch überprüfen:

```
>>> re.search('x[123]{foo}y', 'x222y', re.DEBUG)
LITERAL 120
IN
  LITERAL 49
  LITERAL 50
  LITERAL 51
LITERAL 123
LITERAL 102
LITERAL 111
LITERAL 111
LITERAL 125
LITERAL 121
```

> Sie können sehen, dass es keine gibt `MAX_REPEAT`Token in der Debug-Ausgabe. Das `LITERAL`Token zeigen an, dass der Parser behandelt `{foo}`wörtlich und nicht als Quantifizierer-Metazeichenfolge. `123`, `102`, `111`, `111`, und `125`sind die ASCII-Codes für die Zeichen in der Literalzeichenfolge `'{foo}'`.
>
> Informationen, die von angezeigt werden `DEBUG`flag kann Ihnen bei der Fehlersuche helfen, indem es Ihnen zeigt, wie der Parser Ihre Regex interpretiert.

Kurioserweise die `re`-Modul definiert keine Ein-Buchstaben-Version von `DEBUG`Flagge. Sie könnten Ihre eigenen definieren, wenn Sie wollten:

```
>>> import re
>>> re.D
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 're' has no attribute 'D'

>>> re.D = re.DEBUG
>>> re.search('foo', 'foo', re.D)
LITERAL 102
LITERAL 111
LITERAL 111
<_sre.SRE_Match object; span=(0, 3), match='foo'>
```

Dies könnte jedoch eher verwirrend als hilfreich sein, da Leser Ihres Codes es möglicherweise als Abkürzung für das missverstehen könnten `DOTALL`Flagge. Wenn Sie diese Aufgabe gemacht haben, wäre es eine gute Idee, sie gründlich zu dokumentieren.

`re.A`
`re.ASCII`
`re.U`
`re.UNICODE`
`re.L`
`re.LOCALE`

> Geben Sie die Zeichencodierung an, die zum Parsen spezieller Regex-Zeichenklassen verwendet wird.

Einige der Regex-Metazeichenfolgen (`\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s`, und `\S`) erfordern, dass Sie bestimmten Klassen wie Wörtern, Ziffern oder Leerzeichen Zeichen zuweisen. Die Flags in dieser Gruppe bestimmen das Kodierungsschema, das verwendet wird, um diesen Klassen Zeichen zuzuweisen. Die möglichen Kodierungen sind ASCII, Unicode oder entsprechend dem aktuellen Gebietsschema.

Sie hatten eine kurze Einführung in die Zeichencodierung und Unicode im Tutorial zu Zeichenfolgen und Zeichendaten in Python unter der Diskussion der `ord()`eingebaute Funktion . Ausführlichere Informationen finden Sie in diesen Ressourcen:

- Unicode- und Zeichencodierungen in Python: Ein schmerzloser Leitfaden
- Pythons Unicode-Unterstützung

Warum ist die Zeichenkodierung im Kontext von Regexes in Python so wichtig? Hier ist ein kurzes Beispiel.

Das hast du früher gelernt `\d`gibt ein einstelliges Zeichen an. Die Beschreibung der `\d`Metazeichensequenz gibt an, dass sie der Zeichenklasse entspricht `[0-9]`. Das gilt für Englisch und westeuropäische Sprachen, aber für die meisten Sprachen der Welt, die Schriftzeichen `'0'`durch `'9'`stellen nicht alle oder auch nur *eine* der Ziffern dar.

Hier ist zum Beispiel eine Zeichenfolge, die aus drei Devanagari-Ziffern :

```
>>> s = '\u0967\u096a\u096c'
>>> s
```

```
'৭৪৬'
```

Damit der Regex-Parser das Devanagari-Skript richtig berücksichtigt, die Ziffern-Metazeichenfolge `\d`muss auch mit jedem dieser Zeichen übereinstimmen.

Das Unicode-Konsortium hat Unicode entwickelt, um dieses Problem zu lösen. Unicode ist ein Zeichencodierungsstandard, der alle Schriftsysteme der Welt repräsentieren soll. Alle Zeichenfolgen in Python 3, einschließlich regulärer Ausdrücke, sind standardmäßig Unicode.

Also zurück zu den oben aufgeführten Flaggen. Diese Flags helfen bei der Bestimmung, ob ein Zeichen in eine bestimmte Klasse fällt, indem sie angeben, ob die verwendete Kodierung ASCII, Unicode oder das aktuelle Gebietsschema ist:

- **`re.U`**und **`re.UNICODE`**Geben Sie die Unicode-Codierung an. Unicode ist der Standard, daher sind diese Flags überflüssig. Sie werden hauptsächlich aus Gründen der Abwärtskompatibilität unterstützt.
- **`re.A`**und **`re.ASCII`**Erzwingen Sie eine Bestimmung basierend auf der ASCII-Codierung. Wenn Sie zufällig auf Englisch arbeiten, geschieht dies sowieso, sodass das Flag keinen Einfluss darauf hat, ob eine Übereinstimmung gefunden wird oder nicht.
- **`re.L`**und **`re.LOCALE`**Treffen Sie die Bestimmung basierend auf dem aktuellen Gebietsschema. Gebietsschema ist ein veraltetes Konzept und gilt als nicht zuverlässig. Außer in seltenen Fällen werden Sie es wahrscheinlich nicht brauchen.

Mit der standardmäßigen Unicode-Codierung sollte der Regex-Parser in der Lage sein, jede Sprache zu verarbeiten, die Sie ihm zuwerfen. Im folgenden Beispiel werden alle Zeichen in der Zeichenfolge korrekt erkannt `'৭৪৬'`als Ziffer:

```
>>> s = '\u0967\u096a\u096c'
>>> s
'৭৪৬'
>>> re.search('\d+', s)
<_sre.SRE_Match object; span=(0, 3), match='৭৪৬'>
```

Hier ist ein weiteres Beispiel, das veranschaulicht, wie sich die Zeichenkodierung auf eine Regex-Übereinstimmung in Python auswirken kann. Betrachten Sie diese Zeichenfolge:

```
>>> s = 'sch\u00f6n'
>>> s
'schön'
```

`'schön'`(das deutsche Wort für *hübsch* oder *nett*) enthält die `'ö'`Zeichen, das den 16-Bit-Hexadezimal-Unicode-Wert hat `00f6`. Dieses Zeichen ist im herkömmlichen 7-Bit-ASCII nicht darstellbar.

Wenn Sie auf Deutsch arbeiten, sollten Sie davon ausgehen, dass der Regex-Parser alle darin enthaltenen Zeichen berücksichtigt `'schön'`Wortzeichen sein. Aber schau dir an, was passiert, wenn du suchst `s`für Wortzeichen mit dem `\w`Zeichenklasse und erzwinge eine ASCII-Kodierung:

```
>>> re.search('\w+', s, re.ASCII)
<_sre.SRE_Match object; span=(0, 3), match='sch'>
```

Wenn Sie die Kodierung auf ASCII beschränken, erkennt der Regex-Parser nur die ersten drei Zeichen als Wortzeichen. Das Spiel endet um `'ö'`.

Andererseits, wenn Sie angeben `re.UNICODE`oder zulassen, dass die Codierung standardmäßig auf Unicode eingestellt wird, dann alle Zeichen in `'schön'`gelten als Wortzeichen:

```
>>> re.search('\w+', s, re.UNICODE)
<_sre.SRE_Match object; span=(0, 5), match='schön'>
>>> re.search('\w+', s)
<_sre.SRE_Match object; span=(0, 5), match='schön'>
```

Das `ASCII`und `LOCALE`Flaggen sind verfügbar, falls Sie sie für besondere Umstände benötigen. Aber im Allgemeinen ist es die beste Strategie, die standardmäßige Unicode-Codierung zu verwenden. Dies sollte jede Weltsprache korrekt handhaben.

Kombinieren `<flags>`Argumente in einem Funktionsaufruf

Flag-Werte sind so definiert, dass Sie sie mit dem bitweisen OR (`|`) Operator. Dadurch können Sie mehrere Flags in einem einzigen Funktionsaufruf angeben:

```
>>> re.search('^bar', 'FOO\nBAR\nBAZ', re.I|re.M)
<_sre.SRE_Match object; span=(4, 7), match='BAR'>
```

Dies `re.search()`Aufruf verwendet bitweises ODER, um sowohl die `IGNORECASE`und `MULTILINE`Flaggen auf einmal.

Setzen und Löschen von Flags innerhalb eines regulären Ausdrucks

Neben der Möglichkeit, a `<flags>`Argument für die meisten `re`Modulfunktionsaufrufe können Sie auch Flag-Werte innerhalb einer Regex in Python ändern. Es gibt zwei Regex-Metazeichensequenzen, die diese Möglichkeit bieten.

`(?<flags>)`

> Legt Flag-Werte für die Dauer einer Regex fest.

Innerhalb einer Regex die Metazeichenfolge `(?<flags>)`setzt die angegebenen Flags für den gesamten Ausdruck.

Der Wert von `<flags>`ist ein oder mehrere Buchstaben aus der Menge `a`, `i`, `L`, `m`, `s`, `u`, und `x`. So entsprechen sie dem `re`Modulflags:

Brief	Flaggen
`a`	`re.A` `re.ASCII`
`i`	`re.I` `re.IGNORECASE`
`L`	`re.L` `re.LOCALE`
`m`	`re.M` `re.MULTILINE`
`s`	`re.S` `re.DOTALL`
`u`	`re.U` `re.UNICODE`
`x`	`re.X` `re.VERBOSE`

Das `(?<flags>)`Metazeichenfolge als Ganzes stimmt mit der leeren Zeichenfolge überein. Es stimmt immer erfolgreich überein und verbraucht nichts von der Suchzeichenfolge.

Die folgenden Beispiele sind äquivalente Möglichkeiten zum Einstellen von `IGNORECASE`und `MULTILINE`Flaggen:

```
>>> re.search('^bar', 'FOO\nBAR\nBAZ\n', re.I|re.M)
<_sre.SRE_Match object; span=(4, 7), match='BAR'>

>>> re.search('(?im)^bar', 'FOO\nBAR\nBAZ\n')
<_sre.SRE_Match object; span=(4, 7), match='BAR'>
```

Beachten Sie, dass a `(?<flags>)`Metazeichensequenz setzt die angegebenen Flags für die gesamte Regex, unabhängig davon, wo Sie sie im Ausdruck platzieren:

```
>>> re.search('foo.bar(?s).baz', 'foo\nbar\nbaz')
<_sre.SRE_Match object; span=(0, 11), match='foo\nbar\nbaz'>

>>> re.search('foo.bar.baz(?s)', 'foo\nbar\nbaz')
<_sre.SRE_Match object; span=(0, 11), match='foo\nbar\nbaz'>
```

In den obigen Beispielen stimmen beide Punkt-Metazeichen mit Zeilenumbrüchen überein, weil die `DOTALL`Flagge ist in Kraft. Dies gilt auch dann, wenn `(?s)`erscheint in der Mitte oder am Ende des Ausdrucks.

Ab Python 3.7 ist die Angabe veraltet `(?<flags>)`irgendwo in einer Regex außer am Anfang:

```
>>> import sys
>>> sys.version
'3.8.0 (default, Oct 14 2019, 21:29:03) \n[GCC 7.4.0]'

>>> re.search('foo.bar.baz(?s)', 'foo\nbar\nbaz')
<stdin>:1: DeprecationWarning: Flags not at the start
    of the expression 'foo.bar.baz(?s)'
<re.Match object; span=(0, 11), match='foo\nbar\nbaz'>
```

Es erzeugt immer noch die entsprechende Übereinstimmung, aber Sie erhalten eine Warnmeldung.

`(?<set_flags>-<remove_flags>:<regex>)`

Setzt oder entfernt Flag-Werte für die Dauer einer Gruppe.

`(?<set_flags>-<remove_flags>:<regex>)`definiert eine nicht einfangende Gruppe, gegen die gematcht wird `<regex>`. Für die `<regex>`in der Gruppe enthalten sind, setzt der Regex-Parser alle in angegebenen Flags `<set_flags>`und löscht alle Flags, die in angegeben sind `<remove_flags>`.

Werte für `<set_flags>`und `<remove_flags>`sind am häufigsten `i`, `m`, `s`oder `x`.

Im folgenden Beispiel ist die `IGNORECASE`Flag ist für die angegebene Gruppe gesetzt:

```
>>> re.search('(?i:foo)bar', 'FOObar')
<re.Match object; span=(0, 6), match='FOObar'>
```

Dies ergibt eine Übereinstimmung, weil `(?i:foo)`diktiert, dass das Spiel gegen `'FOO'`Groß- und Kleinschreibung wird nicht beachtet.

Vergleichen Sie dies nun mit diesem Beispiel:

```
>>> print(re.search('(?i:foo)bar', 'FOOBAR'))
None
```

Wie im vorherigen Beispiel, das Spiel gegen `'FOO'` würde gelingen, weil die Groß-/Kleinschreibung nicht beachtet wird. Aber einmal außerhalb der Gruppe, `IGNORECASE` ist nicht mehr in Kraft, also das Spiel gegen `'BAR'` unterscheidet zwischen Groß- und Kleinschreibung und schlägt fehl.

Hier ist ein Beispiel, das das Deaktivieren eines Flags für eine Gruppe zeigt:

```
>>> print(re.search('(?-i:foo)bar', 'FOOBAR', re.IGNORECASE))
None
```

Wieder gibt es keine Übereinstimmung. Obwohl `re.IGNORECASE` ermöglicht den Vergleich ohne Berücksichtigung der Groß-/Kleinschreibung für den gesamten Aufruf, die Metazeichenfolge `(?-i:foo)` schaltet sich aus `IGNORECASE` für die Dauer dieser Gruppe, also das Spiel gegen `'FOO'` scheitert.

Ab Python 3.7 können Sie angeben `u`, `a`, oder `L` wie `<set_flags>` um die Standardcodierung für die angegebene Gruppe zu überschreiben:

```
>>> s = 'sch\u00f6n'
>>> s
'schön'

>>> # Requires Python 3.7 or later
>>> re.search('(?a:\w+)', s)
<re.Match object; span=(0, 3), match='sch'>
>>> re.search('(?u:\w+)', s)
<re.Match object; span=(0, 5), match='schön'>
```

Sie können die Codierung jedoch nur auf diese Weise festlegen. Sie können es nicht entfernen:

```
>>> re.search('(?-a:\w+)', s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.8/re.py", line 199, in search
    return _compile(pattern, flags).search(string)
  File "/usr/lib/python3.8/re.py", line 302, in _compile
    p = sre_compile.compile(pattern, flags)
  File "/usr/lib/python3.8/sre_compile.py", line 764, in compile
    p = sre_parse.parse(p, flags)
  File "/usr/lib/python3.8/sre_parse.py", line 948, in parse
    p = _parse_sub(source, state, flags & SRE_FLAG_VERBOSE, 0)
  File "/usr/lib/python3.8/sre_parse.py", line 443, in _parse_sub
    itemsappend(_parse(source, state, verbose, nested + 1,
  File "/usr/lib/python3.8/sre_parse.py", line 805, in _parse
    flags = _parse_flags(source, state, char)
  File "/usr/lib/python3.8/sre_parse.py", line 904, in _parse_flags
    raise source.error(msg)
re.error: bad inline flags: cannot turn off flags 'a', 'u' and 'L' at
position 4
```

`u`, `a`, und `L` schließen sich gegenseitig aus. Pro Gruppe darf nur einer erscheinen.

Fazit

Damit ist Ihre Einführung in den Abgleich regulärer Ausdrücke und in Python abgeschlossen `re`Modul. Herzliche Glückwünsche! Sie haben eine enorme Menge an Material gemeistert.

Sie wissen jetzt, wie Sie:

- Verwenden `re.search()`Regex-Matching in Python durchzuführen
- Erstellen Sie komplexe Musterabgleichsuchen mit Regex- **Metazeichen**
- Optimieren Sie das Regex-Parsing-Verhalten mit **Flags**

Aber Sie haben immer noch nur eine Funktion im Modul gesehen: `re.search()`! Das `re`-Modul verfügt über viele weitere nützliche Funktionen und Objekte, die Sie Ihrem Mustererkennungs-Toolkit hinzufügen können. Das nächste Tutorial in der Reihe stellt Ihnen vor, was das Regex-Modul in Python sonst noch zu bieten hat.

Reguläre Ausdrücke: Regex in Python (Teil 2)

Inhaltsverzeichnis

Im vorherigen Tutorial dieser Reihe haben Sie viel Boden behandelt. Sie haben gesehen, wie man es benutzt `re.search()`Mustervergleich mit regulären Ausdrücken in Python durchzuführen und lernte die vielen Regex-Metazeichen und Parsing-Flags kennen, die Sie zur Feinabstimmung Ihrer Mustervergleichsfunktionen verwenden können.

Aber so großartig das alles ist, die `re`Modul hat noch viel mehr zu bieten.

In diesem Tutorial werden Sie:

- Entdecken Sie mehr Funktionen, darüber hinaus `re.search()`, dass die `re`Modul bietet
- Erfahren Sie, wann und wie Sie eine Regex in Python in ein **reguläres Ausdrucksobjekt**
- tun können, das **Match-Objekt** von den Funktionen in zurückgegeben wird `re`Modul

Bereit? Lassen Sie uns graben!

`re`Modulfunktionen

Zusätzlich zu `re.search()`, das `re`Das Modul enthält mehrere andere Funktionen, die Ihnen bei der Ausführung von Aufgaben im Zusammenhang mit Regex helfen.

Hinweis: Das haben Sie im vorherigen Tutorial gesehen `re.search()`kann eine optionale nehmen `<flags>`-Argument, das Flags ändern. Alle unten gezeigten Funktionen, mit Ausnahme von `re.escape()`, unterstütz die `<flags>`genauso argumentieren.

Sie können angeben `<flags>`entweder als Positionsargument oder als Schlüsselwortargument:

```
re.search(<regex>, <string>, <flags>)
re.search(<regex>, <string>, flags=<flags>)
```

Die Voreinstellung für `<flags>`ist immer `0`, was auf keine spezielle Modifikation des Übereinstimmungsverhaltens hinweist. Erinnern Sie sich an die Diskussion von Flags im vorherigen Tutorial , dass die `re.UNICODE`Das Flag ist standardmäßig immer gesetzt.

Die verfügbaren Regex-Funktionen in Python `re`Module fallen in die folgenden drei Kategorien:

1. Suchfunktionen
2. Substitutionsfunktionen
3. Dienstprogrammfunktionen

In den folgenden Abschnitten werden diese Funktionen näher erläutert.

Suchfunktionen

Suchfunktionen scannen einen Suchstring nach einer oder mehreren Übereinstimmungen der angegebenen Regex:

Funktion	Beschreibung
`re.search()`	Durchsucht eine Zeichenfolge nach einer Regex-Übereinstimmung
`re.match()`	Sucht nach einer Regex-Übereinstimmung am Anfang einer Zeichenfolge
`re.fullmatch()`	Sucht nach einer Regex-Übereinstimmung auf einer ganzen Zeichenfolge
`re.findall()`	Gibt eine Liste aller Regex-Übereinstimmungen in einem String zurück
`re.finditer()`	Gibt einen Iterator zurück, der Regex-Übereinstimmungen aus einer Zeichenfolge liefert

Wie Sie der Tabelle entnehmen können, sind diese Funktionen einander ähnlich. Aber jeder optimiert die Suchfunktion auf seine eigene Weise.

`re.search(<regex>, <string>, flags=0)`

Durchsucht eine Zeichenfolge nach einer Regex-Übereinstimmung.

durchgearbeitet vorherige Tutorial haben, sollten Sie mit dieser Funktion inzwischen gut vertraut sein. `re.search(<regex>, <string>)`sucht nach einem beliebigen Ort in `<string>`wo `<regex>`Streichhölzer:

```
>>> re.search(r'(\d+)', 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re.search(r'[a-z]+', '123FOO456', flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(3, 6), match='FOO'>

>>> print(re.search(r'\d+', 'foo.bar'))
None
```

Die Funktion gibt ein Übereinstimmungsobjekt zurück, wenn sie eine Übereinstimmung findet und `None`Andernfalls.

`re.match(<regex>, <string>, flags=0)`

Sucht nach einer Regex-Übereinstimmung am Anfang einer Zeichenfolge.

Dies ist identisch mit `re.search()`, außer dass `re.search()`gibt eine Übereinstimmung zurück, wenn `<regex>`Übereinstimmungen *überall* in `<string>`, wohingegen `re.match()`gibt nur dann eine Übereinstimmung zurück, wenn `<regex>`Spiele *Beginn* zu `<string>`:

```
>>> re.search(r'\d+', '123foobar')
<_sre.SRE_Match object; span=(0, 3), match='123'>
>>> re.search(r'\d+', 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> re.match(r'\d+', '123foobar')
<_sre.SRE_Match object; span=(0, 3), match='123'>
>>> print(re.match(r'\d+', 'foo123bar'))
None
```

Im obigen Beispiel `re.search()`stimmt überein, wenn die Ziffern sowohl am Anfang der Zeichenfolge als auch in der Mitte stehen, aber `re.match()`passt nur, wenn die Ziffern am Anfang stehen.

Erinnern Sie sich aus dem vorherigen Tutorial in dieser Reihe daran, dass if `<string>`enthält eingebettete Zeilenumbrüche, dann die `MULTILINE`Flagge verursacht `re.search()`passend zum Caretzeichen (`^`) Anker-Metazeichen entweder am Anfang von `<string>`oder am Anfang einer darin enthaltenen Zeile `<string>`:

```
>>> s = 'foo\nbar\nbaz'



>>> re.search('^foo', s)

<_sre.SRE_Match object; span=(0, 3), match='foo'>

>>> re.search('^bar', s, re.MULTILINE)

<_sre.SRE_Match object; span=(4, 7), match='bar'>
```

Das `MULTILINE`Flagge hat keinen Einfluss `re.match()`auf diese Weise:

```
>>> s = 'foo\nbar\nbaz'



>>> re.match('^foo', s)

<_sre.SRE_Match object; span=(0, 3), match='foo'>

>>> print(re.match('^bar', s, re.MULTILINE))
```

```
None
```

Auch mit der `MULTILINE`Flagge gesetzt, `re.match()`entspricht dem Caretzeichen (`^`) Anker nur am Anfang von `<string>`, nicht am Anfang der darin enthaltenen Zeilen `<string>`.

Beachten Sie, dass das Caretzeichen (`^`) Anker in **Zeile 3** im obigen Beispiel ist überflüssig. Mit `re.match()`, werden Übereinstimmungen grundsätzlich immer am Anfang der Zeichenfolge verankert.

`re.fullmatch(<regex>, <string>, flags=0)`

Sucht nach einer Regex-Übereinstimmung auf einer ganzen Zeichenfolge.

Dies ist ähnlich wie `re.search()`und `re.match()`, aber `re.fullmatch()`gibt nur dann eine Übereinstimmung zurück, wenn `<regex>`Streichhölzer `<string>`in seiner Gänze:

```
>>> print(re.fullmatch(r'\d+', '123foo'))

None

>>> print(re.fullmatch(r'\d+', 'foo123'))

None

>>> print(re.fullmatch(r'\d+', 'foo123bar'))

None

>>> re.fullmatch(r'\d+', '123')

<_sre.SRE_Match object; span=(0, 3), match='123'>



>>> re.search(r'^\d+$', '123')

<_sre.SRE_Match object; span=(0, 3), match='123'>
```

Beim Aufruf auf **Zeile 7** die Suchzeichenfolge `'123'`besteht von Anfang bis Ende ausschließlich aus Ziffern. Das ist also der einzige Fall, in dem `re.fullmatch()`gibt ein Spiel zurück.

Das `re.search()`Anruf auf **Leitung 10** , in dem die `\d+`regex explizit am Anfang und am Ende des Suchstrings verankert wird, ist funktional gleichwertig.

`re.findall(<regex>, <string>, flags=0)`

Gibt eine Liste aller Übereinstimmungen einer Regex in einem String zurück.

`re.findall(<regex>, <string>)`gibt eine Liste aller nicht überlappenden Übereinstimmungen von zurück `<regex>`in `<string>`. Es scannt die Suchzeichenfolge von links nach rechts und gibt alle Übereinstimmungen in der gefundenen Reihenfolge zurück:

```
>>> re.findall(r'\w+', '...foo,,,,bar:%$baz//|')
['foo', 'bar', 'baz']
```

Wenn `<regex>`eine einfangende Gruppe enthält, dann enthält die Rückgabeliste nur den Inhalt der Gruppe, nicht die gesamte Übereinstimmung:

```
>>> re.findall(r'#(\w+)#', '#foo#.#bar#.#baz#')
['foo', 'bar', 'baz']
```

In diesem Fall ist die angegebene Regex `#(\w+)#`. Die passenden Saiten sind `'#foo#'`, `'#bar#'`, und `'#baz#'`. Aber das Haschisch (#) Zeichen werden nicht in der Rückgabeliste angezeigt, da sie sich außerhalb der Gruppierungsklammern befinden.

Wenn `<regex>`enthält dann mehr als eine einfangende Gruppe `re.findall()`gibt eine Liste von Tupeln zurück, die die erfassten Gruppen enthalten. Die Länge jedes Tupels entspricht der Anzahl der angegebenen Gruppen:

```
>>> re.findall(r'(\w+),(\w+)', 'foo,bar,baz,qux,quux,corge')

[('foo', 'bar'), ('baz', 'qux'), ('quux', 'corge')]



>>> re.findall(r'(\w+),(\w+),(\w+)', 'foo,bar,baz,qux,quux,corge')

[('foo', 'bar', 'baz'), ('qux', 'quux', 'corge')]
```

Im obigen Beispiel enthält die Regex in **Zeile 1** zwei einfangende Gruppen, also `re.findall()`gibt eine Liste von drei Zwei-Tupeln zurück, die jeweils zwei erfasste Übereinstimmungen enthalten. **Zeile 4** enthält drei Gruppen, der Rückgabewert ist also eine Liste mit zwei Dreier-Tupeln.

`re.finditer(<regex>, <string>, flags=0)`

Gibt einen Iterator zurück, der Regex-Übereinstimmungen liefert.

`re.finditer(<regex>, <string>)`scannt `<string>`für nicht überlappende Übereinstimmungen von `<regex>`und gibt einen Iterator zurück, der die Übereinstimmungsobjekte von allen findet. Es scannt die Suchzeichenfolge von links nach rechts und gibt Übereinstimmungen in der Reihenfolge zurück, in der es sie findet:

```
>>> it = re.finditer(r'\w+', '...foo,,,,bar:%$baz//|')
>>> next(it)
<_sre.SRE_Match object; span=(3, 6), match='foo'>
>>> next(it)
<_sre.SRE_Match object; span=(10, 13), match='bar'>
>>> next(it)
<_sre.SRE_Match object; span=(16, 19), match='baz'>
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> for i in re.finditer(r'\w+', '...foo,,,,bar:%$baz//|'):
...     print(i)
...
<_sre.SRE_Match object; span=(3, 6), match='foo'>
<_sre.SRE_Match object; span=(10, 13), match='bar'>
<_sre.SRE_Match object; span=(16, 19), match='baz'>
```

`re.findall()`und `re.finditer()`sind sich sehr ähnlich, unterscheiden sich aber in zwei Punkten:

1. `re.findall()`gibt eine Liste zurück, wohingegen `re.finditer()`gibt einen Iterator zurück.
2. Die Elemente in der Liste, die `re.findall()`Returns sind die eigentlichen übereinstimmenden Strings, während die vom Iterator zurückgegebenen Elemente das sind `re.finditer()`Rückgaben sind Match-Objekte.

Jede Aufgabe, die Sie mit dem einen erledigen könnten, könnten Sie wahrscheinlich auch mit dem anderen bewältigen. Welche Sie wählen, hängt von den Umständen ab. Wie Sie später in diesem Lernprogramm sehen werden, können viele nützliche Informationen von einem Übereinstimmungsobjekt abgerufen werden. Wenn Sie diese Informationen benötigen, dann `re.finditer()`wird wohl die bessere wahl sein.

Substitutionsfunktionen

Substitutionsfunktionen ersetzen Teile einer Suchzeichenfolge, die mit einer bestimmten Regex übereinstimmen:

Funktion	Beschreibung
`re.sub()`	Durchsucht eine Zeichenfolge nach Regex-Übereinstimmungen, ersetzt die übereinstimmenden Teile der Zeichenfolge durch die angegebene Ersatzzeichenfolge und gibt das Ergebnis zurück
`re.subn()`	Verhält sich genauso `re.sub()`gibt aber auch Informationen über die Anzahl der vorgenommenen Ersetzungen zurück

Beide `re.sub()`und `re.subn()`Erstellen Sie eine neue Zeichenfolge mit den angegebenen Ersetzungen und geben Sie sie zurück. Der ursprüngliche String bleibt unverändert. (Denken Sie daran, dass Zeichenfolgen in Python unveränderlich sind, sodass es diesen Funktionen nicht möglich wäre, die ursprüngliche Zeichenfolge zu ändern.)

`re.sub(<regex>, <repl>, <string>, count=0, flags=0)`

> Gibt eine neue Zeichenfolge zurück, die sich aus der Ersetzung einer Suchzeichenfolge ergibt.

`re.sub(<regex>, <repl>, <string>)`findet die am weitesten links liegenden nicht überlappenden Vorkommen von `<regex>`in `<string>`, ersetzt jede Übereinstimmung wie durch angegeben `<repl>`, und gibt das Ergebnis zurück. `<string>`bleibt unverändert.

`<repl>`kann entweder ein String oder eine Funktion sein, wie unten erklärt.

Substitution durch Zeichenfolge

Wenn `<repl>`ist dann ein String `re.sub()`fügt es ein `<string>`anstelle aller übereinstimmenden Sequenzen `<regex>`:

```
>>> s = 'foo.123.bar.789.baz'
```

```
>>> re.sub(r'\d+', '#', s)

'foo.#.bar.#.baz'

>>> re.sub('[a-z]+', '(*)', s)

'(*).123.(*).789.(*)'
```

In **Zeile 3** die Zeichenfolge `'#'` ersetzt Ziffernfolgen in `s`. In **Zeile 5** die Zeichenfolge `'(*)'` ersetzt Folgen von Kleinbuchstaben. In beiden Fällen, `re.sub()` gibt den geänderten String wie immer zurück.

`re.sub()` ersetzt nummerierte Rückverweise (`\<n>`) in `<repl>` mit dem Text der entsprechenden erfassten Gruppe:

```
>>> re.sub(r'(\w+),bar,baz,(\w+)',
...         r'\2,bar,baz,\1',
...         'foo,bar,baz,qux')
'qux,bar,baz,foo'
```

Hier enthalten die erfassten Gruppen 1 und 2 `'foo'` und `'qux'`. In der Ersetzungszeichenfolge `'\2,bar,baz,\1'`, `'foo'` ersetzt `\1` und `'qux'` ersetzt `\2`.

Sie können auch auf benannte Rückverweise verweisen, die mit erstellt wurden `(?P<name><regex>)` in der Ersetzungszeichenfolge mithilfe der Metazeichenfolge `\g<name>`:

```
>>> re.sub(r'foo,(?P<w1>\w+),(?P<w2>\w+),qux',
...         r'foo,\g<w2>,\g<w1>,qux',
...         'foo,bar,baz,qux')
'foo,baz,bar,qux'
```

Tatsächlich können Sie auf diese Weise auch auf *nummerierte* Rückverweise verweisen, indem Sie die Gruppennummer in den spitzen Klammern angeben:

```
>>> re.sub(r'foo,(\w+),(\w+),qux',
...         r'foo,\g<2>,\g<1>,qux',
...         'foo,bar,baz,qux')
'foo,baz,bar,qux'
```

Möglicherweise müssen Sie diese Technik verwenden, um Mehrdeutigkeiten in Fällen zu vermeiden, in denen auf eine nummerierte Rückwärtsreferenz unmittelbar eine wörtliche Ziffer folgt. Angenommen, Sie haben eine Zeichenfolge wie `'foo 123 bar'` und möchte eine hinzufügen `'0'` am Ende der Ziffernfolge. Sie könnten dies versuchen:

```
>>> re.sub(r'(\d+)', r'\10', 'foo 123 bar')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.6/re.py", line 191, in sub
    return _compile(pattern, flags).sub(repl, string, count)
  File "/usr/lib/python3.6/re.py", line 326, in _subx
    template = _compile_repl(template, pattern)
  File "/usr/lib/python3.6/re.py", line 317, in _compile_repl
    return sre_parse.parse_template(repl, pattern)
  File "/usr/lib/python3.6/sre_parse.py", line 943, in parse_template
    addgroup(int(this[1:]), len(this) - 1)
  File "/usr/lib/python3.6/sre_parse.py", line 887, in addgroup
    raise s.error("invalid group reference %d" % index, pos)
```

```
sre_constants.error: invalid group reference 10 at position 1
```

Leider interpretiert der Regex-Parser in Python `\10`als Rückverweis auf die zehnte erfasste Gruppe, die in diesem Fall nicht existiert. Stattdessen können Sie verwenden `\g<1>`um auf die Gruppe zu verweisen:

```
>>> re.sub(r'(\d+)', r'\g<1>0', 'foo 123 bar')
'foo 1230 bar'
```

Die Rückreferenz `\g<0>`bezieht sich auf den Text des gesamten Spiels. Dies gilt auch dann, wenn keine Gruppierungsklammern vorhanden sind `<regex>`:

```
>>> re.sub(r'\d+', '/\g<0>/', 'foo 123 bar')
'foo /123/ bar'
```

Wenn `<regex>`gibt dann eine Übereinstimmung mit der Länge null an `re.sub()`wird ersetzen `<repl>`in jede Zeichenposition in der Zeichenfolge:

```
>>> re.sub('x*', '-', 'foo')
'-f-o-o-'
```

Im obigen Beispiel ist die Regex `x*`stimmt mit jeder Sequenz der Länge Null überein, also `re.sub()`fügt die Ersetzungszeichenfolge an jeder Zeichenposition in der Zeichenfolge ein – vor dem ersten Zeichen, zwischen jedem Zeichenpaar und nach dem letzten Zeichen.

Wenn `re.sub()`keine Übereinstimmungen findet, kehrt es immer zurück `<string>`unverändert.

Substitution durch Funktion

Wenn Sie angeben `<repl>`dann als Funktion `re.sub()`ruft diese Funktion für jede gefundene Übereinstimmung auf. Es übergibt jedes entsprechende Übereinstimmungsobjekt als Argument an die Funktion, um Informationen über die Übereinstimmung bereitzustellen. Der Rückgabewert der Funktion wird dann zum Ersetzungsstring:

```
>>> def f(match_obj):
...     s = match_obj.group(0)  # The matching string
...
...     # s.isdigit() returns True if all characters in s are digits
...     if s.isdigit():
...         return str(int(s) * 10)
...     else:
...         return s.upper()
...
>>> re.sub(r'\w+', f, 'foo.10.bar.20.baz.30')
'FOO.100.BAR.200.BAZ.300'
```

In diesem Beispiel `f()`wird für jedes Spiel aufgerufen. Als Ergebnis, `re.sub()`konvertiert jeden alphanumerischen Teil von `<string>`in Großbuchstaben und multipliziert jeden numerischen Teil mit `10`.

Begrenzung der Anzahl der Ersetzungen

Wenn Sie eine positive Ganzzahl für das optionale angeben `count`Parameter, dann `re.sub()`führt höchstens so viele Ersetzungen durch:

```
>>> re.sub(r'\w+', 'xxx', 'foo.bar.baz.qux')
```

```
'xxx.xxx.xxx.xxx'
>>> re.sub(r'\w+', 'xxx', 'foo.bar.baz.qux', count=2)
'xxx.xxx.baz.qux'
```

Wie bei den meisten `re`Modulfunktionen, `re.sub()`akzeptiert optional `<flags>`Argument auch.

`re.subn(<regex>, <repl>, <string>, count=0, flags=0)`

> Gibt eine neue Zeichenfolge zurück, die sich aus der Durchführung von Ersetzungen an einer Suchzeichenfolge ergibt, und gibt auch die Anzahl der vorgenommenen Ersetzungen zurück.

`re.subn()`ist identisch mit `re.sub()`, außer dass `re.subn()`gibt ein Zwei-Tupel zurück, das aus der geänderten Zeichenfolge und der Anzahl der vorgenommenen Ersetzungen besteht:

```
>>> re.subn(r'\w+', 'xxx', 'foo.bar.baz.qux')
('xxx.xxx.xxx.xxx', 4)
>>> re.subn(r'\w+', 'xxx', 'foo.bar.baz.qux', count=2)
('xxx.xxx.baz.qux', 2)

>>> def f(match_obj):
...     m = match_obj.group(0)
...     if m.isdigit():
...         return str(int(m) * 10)
...     else:
...         return m.upper()
...
>>> re.subn(r'\w+', f, 'foo.10.bar.20.baz.30')
('FOO.100.BAR.200.BAZ.300', 6)
```

In allen anderen Belangen `re.subn()`verhält sich genauso `re.sub()`.

Dienstprogrammfunktionen

Es gibt zwei verbleibende Regex-Funktionen in Python `re`Modul, das Sie noch abdecken müssen:

Funktion	Beschreibung
`re.split()`	Teilt einen String mit einem regulären Ausdruck als Trennzeichen in Teilstrings auf
`re.escape()`	Maskiert Zeichen in einer Regex

Dies sind Funktionen, die einen Regex-Abgleich beinhalten, aber nicht eindeutig in eine der oben beschriebenen Kategorien fallen.

`re.split(<regex>, <string>, maxsplit=0, flags=0)`

> Teilt einen String in Teilstrings.

`re.split(<regex>, <string>)`spaltet `<string>`in Teilstrings mit `<regex>`als Trennzeichen und gibt die Teilstrings als Liste zurück.

Das folgende Beispiel teilt die angegebene Zeichenfolge in Teilzeichenfolgen, die durch ein Komma getrennt sind (`,`), Semikolon (`;`) oder Schrägstrich (`/`) Zeichen, umgeben von beliebig viel Leerzeichen:

```
>>> re.split('\s*[,;/]\s*', 'foo,bar  ;  baz / qux')
['foo', 'bar', 'baz', 'qux']
```

Wenn `<regex>`einfangende Gruppen enthält, enthält die Rückgabeliste auch die passenden Trennzeichenfolgen:

```
>>> re.split('(\s*[,;/]\s*)', 'foo,bar  ;  baz / qux')
['foo', ',', 'bar', '  ;  ', 'baz', ' / ', 'qux']
```

Diesmal enthält die Rückgabeliste nicht nur die Teilstrings `'foo'`, `'bar'`, `'baz'`, und `'qux'`sondern auch mehrere Trennzeichenfolgen:

- `','`
- `' ; '`
- `' / '`

Dies kann nützlich sein, wenn Sie aufteilen möchten `<string>`Zerlegen Sie sie in getrennte Token, verarbeiten Sie die Token auf irgendeine Weise und setzen Sie die Zeichenfolge dann wieder zusammen, indem Sie dieselben Trennzeichen verwenden, die sie ursprünglich getrennt haben:

```
>>> string = 'foo,bar  ;  baz / qux'
>>> regex = r'(\s*[,;/]\s*)'
>>> a = re.split(regex, string)

>>> # List of tokens and delimiters
>>> a
['foo', ',', 'bar', '  ;  ', 'baz', ' / ', 'qux']

>>> # Enclose each token in <>'s
>>> for i, s in enumerate(a):
...
...     # This will be True for the tokens but not the delimiters
...     if not re.fullmatch(regex, s):
...         a[i] = f'<{s}>'
...

>>> # Put the tokens back together using the same delimiters
>>> ''.join(a)
'<foo>,<bar>  ;  <baz> / <qux>'
```

Wenn Sie Gruppen verwenden müssen, aber nicht möchten, dass die Trennzeichen in die Rückgabeliste aufgenommen werden, können Sie nicht erfassende Gruppen verwenden:

```
>>> string = 'foo,bar  ;  baz / qux'
>>> regex = r'(?:\s*[,;/]\s*)'
>>> re.split(regex, string)
['foo', 'bar', 'baz', 'qux']
```

Wenn die optional `maxsplit`argument ist vorhanden und größer als Null, dann `re.split()`führt höchstens so viele Splits aus. Das letzte Element in der Rückgabeliste ist der Rest von `<string>`Nachdem alle Splits aufgetreten sind:

```
>>> s = 'foo, bar, baz, qux, quux, corge'

>>> re.split(r',\s*', s)
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> re.split(r',\s*', s, maxsplit=3)
['foo', 'bar', 'baz', 'qux, quux, corge']
```

Explizit angeben `maxsplit=0`ist gleichbedeutend mit dem vollständigen Weglassen. Wenn `maxsplit`ist dann negativ `re.split()`kehrt zurück `<string>`unverändert (falls Sie auf der Suche nach einer ziemlich aufwändigen Möglichkeit waren, gar nichts zu tun).

Wenn `<regex>`enthält Erfassungsgruppen, sodass die Rückgabeliste Trennzeichen enthält, und `<regex>`entspricht dem Beginn von `<string>`, dann `re.split()`fügt einen leeren String als erstes Element in die Rückgabeliste ein. Ebenso ist das letzte Element in der Rückgabeliste eine leere Zeichenfolge if `<regex>`entspricht dem Ende von `<string>`:

```
>>> re.split('(/)', '/foo/bar/baz/')
['', '/', 'foo', '/', 'bar', '/', 'baz', '/', '']
```

In diesem Fall ist die `<regex>`Trennzeichen ist ein einzelner Schrägstrich (`/`) Charakter. In gewisser Weise befindet sich also links vom ersten Trennzeichen und rechts vom letzten ein leerer String. Das macht also Sinn `re.split()`platziert leere Zeichenfolgen als erstes und letztes Element der Rückgabeliste.

`re.escape(<regex>)`

> Maskiert Zeichen in einer Regex.

`re.escape(<regex>)`gibt eine Kopie von zurück `<regex>`wobei jedem Nichtwortzeichen (alles andere als ein Buchstabe, eine Ziffer oder ein Unterstrich) ein umgekehrter Schrägstrich vorangestellt ist.

Dies ist nützlich, wenn Sie einen der anrufen `re`Modulfunktionen und die `<regex>`Sie übergeben eine Menge Sonderzeichen, die der Parser buchstäblich statt als Metazeichen auffassen soll. Es erspart Ihnen die Mühe, alle Backslash-Zeichen manuell einzugeben:

```
>>> print(re.match('foo^bar(baz)|qux', 'foo^bar(baz)|qux'))

None

>>> re.match('foo\^bar\(baz\)\|qux', 'foo^bar(baz)|qux')

<_sre.SRE_Match object; span=(0, 16), match='foo^bar(baz)|qux'>



>>> re.escape('foo^bar(baz)|qux') == 'foo\^bar\(baz\)\|qux'

True

>>> re.match(re.escape('foo^bar(baz)|qux'), 'foo^bar(baz)|qux')

<_sre.SRE_Match object; span=(0, 16), match='foo^bar(baz)|qux'>
```

In diesem Beispiel gibt es keine Übereinstimmung in **Zeile 1,** weil die regex `'foo^bar(baz)|qux'`enthält Sonderzeichen, die sich wie Metazeichen verhalten. In **Zeile 3** werden sie explizit mit Backslashes maskiert, sodass eine Übereinstimmung auftritt. **Zeilen 6 und 8** zeigen, dass Sie denselben Effekt erzielen können, indem Sie verwenden `re.escape()`.

Kompilierte Regex-Objekte in Python

Das `re`-Modul unterstützt die Fähigkeit, eine Regex in Python in ein **reguläres Ausdrucksobjekt vorzukompilieren** , das später wiederholt verwendet werden kann.

`re.compile(<regex>, flags=0)`

> Kompiliert einen regulären Ausdruck in ein reguläres Ausdrucksobjekt.

`re.compile(<regex>)`kompiliert `<regex>`und gibt das entsprechende reguläre Ausdrucksobjekt zurück. Wenn Sie a `<flags>`Wert, dann gelten die entsprechenden Flags für alle Suchen, die mit dem Objekt durchgeführt werden.

Es gibt zwei Möglichkeiten, ein kompiliertes reguläres Ausdrucksobjekt zu verwenden. Sie können es als erstes Argument für angeben `re`Modulfunktionen anstelle von `<regex>`:

```
re_obj = re.compile(<regex>, <flags>)
result = re.search(re_obj, <string>)
```

Sie können eine Methode auch direkt aus einem regulären Ausdrucksobjekt aufrufen:

```
re_obj = re.compile(<regex>, <flags>)
result = re_obj.search(<string>)
```

Die beiden obigen Beispiele sind äquivalent dazu:

```
result = re.search(<regex>, <string>, <flags>)
```

Hier ist eines der Beispiele, die Sie zuvor gesehen haben, neu formuliert mit einem kompilierten Objekt für reguläre Ausdrücke:

```
>>> re.search(r'(\d+)', 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> re_obj = re.compile(r'(\d+)')
>>> re.search(re_obj, 'foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re_obj.search('foo123bar')
<_sre.SRE_Match object; span=(3, 6), match='123'>
```

Hier ist eine andere, die auch die verwendet `IGNORECASE`Flagge:

```
>>> r1 = re.search('ba[rz]', 'FOOBARBAZ', flags=re.I)



>>> re_obj = re.compile('ba[rz]', flags=re.I)

>>> r2 = re.search(re_obj, 'FOOBARBAZ')

>>> r3 = re_obj.search('FOOBARBAZ')



>>> r1

<_sre.SRE_Match object; span=(3, 6), match='BAR'>
```

```
>>> r2

<_sre.SRE_Match object; span=(3, 6), match='BAR'>

>>> r3

<_sre.SRE_Match object; span=(3, 6), match='BAR'>
```

In diesem Beispiel gibt die Anweisung in **Zeile 1** Regex an `ba[rz]` direkt zu `re.search()` als erstes Argument. In **Zeile 4** das erste Argument für `re.search()` ist das kompilierte reguläre Ausdrucksobjekt `re_obj`. In **Zeile 5** , `search()` wird direkt aufgerufen `re_obj`. Alle drei Fälle erzeugen die gleiche Übereinstimmung.

Warum sich die Mühe machen, eine Regex zu kompilieren?

Was nützt das Vorkompilieren? Es gibt ein paar mögliche Vorteile.

Wenn Sie häufig einen bestimmten Regex in Ihrem Python-Code verwenden, können Sie durch Vorkompilieren die Regex-Definition von ihrer Verwendung trennen. Dies verbessert die Modularität . Betrachten Sie dieses Beispiel:

```
>>> s1, s2, s3, s4 = 'foo.bar', 'foo123bar', 'baz99', 'qux & grault'

>>> import re
>>> re.search('\d+', s1)
>>> re.search('\d+', s2)
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re.search('\d+', s3)
<_sre.SRE_Match object; span=(3, 5), match='99'>
>>> re.search('\d+', s4)
```

Hier die Regex `\d+` taucht mehrfach auf. Wenn Sie im Laufe der Pflege dieses Codes entscheiden, dass Sie eine andere Regex benötigen, müssen Sie sie an jeder Stelle ändern. Das ist bei diesem kleinen Beispiel gar nicht so schlimm, denn die Nutzungen liegen nah beieinander. Aber in einer größeren Anwendung könnten sie weit verstreut und schwer aufzuspüren sein.

Das Folgende ist modularer und wartbarer:

```
>>> s1, s2, s3, s4 = 'foo.bar', 'foo123bar', 'baz99', 'qux & grault'
>>> re_obj = re.compile('\d+')

>>> re_obj.search(s1)
>>> re_obj.search(s2)
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re_obj.search(s3)
<_sre.SRE_Match object; span=(3, 5), match='99'>
>>> re_obj.search(s4)
```

Andererseits können Sie eine ähnliche Modularität ohne Vorkompilierung erreichen, indem Sie die Variablenzuweisung verwenden:

```
>>> s1, s2, s3, s4 = 'foo.bar', 'foo123bar', 'baz99', 'qux & grault'
>>> regex = '\d+'

>>> re.search(regex, s1)
>>> re.search(regex, s2)
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> re.search(regex, s3)
```

```
<_sre.SRE_Match object; span=(3, 5), match='99'>
>>> re.search(regex, s4)
```

Theoretisch könnten Sie erwarten, dass die Vorkompilierung auch zu einer schnelleren Ausführungszeit führt. Angenommen, Sie rufen an `re.search()`viele tausend Male auf der gleichen Regex. Es mag den Anschein haben, als wäre es effizienter, die Regex einmal im Voraus zu kompilieren, als sie jedes Mal neu zu kompilieren, wenn sie tausende Male verwendet wird.

In der Praxis ist das aber nicht der Fall. Die Wahrheit ist, dass die `re`-Modul kompiliert und cachet eine Regex, wenn sie in einem Funktionsaufruf verwendet wird. Wenn dieselbe Regex anschließend im selben Python-Code verwendet wird, wird sie nicht neu kompiliert. Der kompilierte Wert wird stattdessen aus dem Cache abgerufen. Der Leistungsvorteil ist also minimal.

Alles in allem gibt es keinen zwingenden Grund, eine Regex in Python zu kompilieren. Wie vieles in Python ist es nur ein weiteres Tool in Ihrem Toolkit, das Sie verwenden können, wenn Sie glauben, dass es die Lesbarkeit oder Struktur Ihres Codes verbessert.

Objektmethoden für reguläre Ausdrücke

Ein kompiliertes reguläres Ausdrucksobjekt `re_obj`unterstützt die folgenden Methoden:

- `re_obj.search(<string>[, <pos>[, <endpos>]])`
- `re_obj.match(<string>[, <pos>[, <endpos>]])`
- `re_obj.fullmatch(<string>[, <pos>[, <endpos>]])`
- `re_obj.findall(<string>[, <pos>[, <endpos>]])`
- `re_obj.finditer(<string>[, <pos>[, <endpos>]])`

Diese verhalten sich alle gleich wie die entsprechenden `re`Funktionen, denen Sie bereits begegnet sind, mit der Ausnahme, dass sie auch die optionalen unterstützen `<pos>`und `<endpos>`Parameter. Sind diese vorhanden, dann bezieht sich die Suche nur auf den Teil von `<string>`angezeigt durch `<pos>`und `<endpos>`, die sich wie Indizes in Slice-Notation verhalten :

```
>>> re_obj = re.compile(r'\d+')

>>> s = 'foo123barbaz'



>>> re_obj.search(s)

<_sre.SRE_Match object; span=(3, 6), match='123'>



>>> s[6:9]

'bar'

>>> print(re_obj.search(s, 6, 9))

None
```

Im obigen Beispiel lautet die Regex `\d+`, eine Folge von Ziffern. Das `.search()`Anruf auf **Leitung 4** sucht alle `s`, also gibt es eine Übereinstimmung. Auf **Zeile 9** , die `<pos>`und `<endpos>`Parameter schränken die Suche effektiv auf die Teilzeichenfolge ein, die mit Zeichen 6 beginnt und bis zu Zeichen 9 reicht, diese jedoch nicht einschließt (die Teilzeichenfolge `'bar'`), die keine Ziffern enthält.

Wenn Sie angeben `<pos>`aber weglassen `<endpos>`, dann gilt die Suche für den Teilstring von `<pos>`bis zum Ende der Zeichenfolge.

Beachten Sie, dass Anker wie Caret (`^`) und Dollarzeichen (`$`) beziehen sich immer noch auf den Anfang und das Ende der gesamten Zeichenfolge, nicht auf die Teilzeichenfolge, die durch bestimmt wird `<pos>`und `<endpos>`:

```
>>> re_obj = re.compile('^bar')
>>> s = 'foobarbaz'

>>> s[3:]
'barbaz'

>>> print(re_obj.search(s, 3))
None
```

Hier allerdings `'bar'`am Anfang des Teilstrings ab Zeichen 3 steht, aber nicht am Anfang des gesamten Strings, also das Caretzeichen (`^`)-Anker stimmt nicht überein.

Die folgenden Methoden sind für ein kompiliertes reguläres Ausdrucksobjekt verfügbar `re_obj`auch:

- `re_obj.split(<string>, maxsplit=0)`
- `re_obj.sub(<repl>, <string>, count=0)`
- `re_obj.subn(<repl>, <string>, count=0)`

Auch diese verhalten sich analog zu den entsprechenden `re`Funktionen, aber sie unterstützen nicht die `<pos>`und `<endpos>`Parameter.

Objektattribute für reguläre Ausdrücke

Das `re`Das Modul definiert mehrere nützliche Attribute für ein kompiliertes reguläres Ausdrucksobjekt:

Attribut	Bedeutung
`re_obj.flags`	Irgendein `<flags>`die für die Regex gelten
`re_obj.groups`	Die Anzahl der einfangenden Gruppen in der Regex
`re_obj.groupindex`	Ein Wörterbuch, das jeden symbolischen Gruppennamen abbildet, der durch definiert wird `(?P<name>)`Konstrukt (falls vorhanden) auf die entsprechende Gruppennummer
`re_obj.pattern`	Das `<regex>`Muster, das dieses Objekt erzeugt hat

Der folgende Code demonstriert einige Verwendungen dieser Attribute:

```
>>> re_obj = re.compile(r'(?m)(\w+),(\w+)', re.I)

>>> re_obj.flags
```

```
42

>>> re.I|re.M|re.UNICODE

<RegexFlag.UNICODE|MULTILINE|IGNORECASE: 42>

>>> re_obj.groups

2

>>> re_obj.pattern

'(?m)(\\w+),(\\w+)'


>>> re_obj = re.compile(r'(?P<w1>),(?P<w2>)')

>>> re_obj.groupindex

mappingproxy({'w1': 1, 'w2': 2})

>>> re_obj.groupindex['w1']

1

>>> re_obj.groupindex['w2']

2
```

Beachten Sie, dass `.flags`enthält alle Flags, die als Argumente für angegeben sind `re.compile()`, alle innerhalb der Regex mit dem angegebenen `(?flags)`Metazeichenfolge und alle, die standardmäßig in Kraft sind. definierten regulären Ausdrucksobjekt **Zeile 1** sind drei Flags definiert:

1. **`re.I`:** Angegeben als `<flags>`Wert in der `re.compile()`Anruf
2. **`re.M`:** Angegeben als `(?m)`innerhalb der Regex
3. **`re.UNICODE`:** Standardmäßig aktiviert

Sie können in **Zeile 4** , dass der Wert von `re_obj.flags`ist das **logische ODER** dieser drei Werte, was gleich ist `42`.

Der Wert der `.groupindex`Das Attribut für das in **Zeile 11** ist technisch gesehen ein Objekt vom Typ `mappingproxy`. Aus praktischen Gründen funktioniert es wie ein Wörterbuch .

Objektmethoden und -attribute abgleichen

Wie Sie gesehen haben, sind die meisten Funktionen und Methoden in der `re`-Modul gibt ein **Übereinstimmungsobjekt** wenn es eine erfolgreiche Übereinstimmung gibt. Da ein Match-Objekt truthy , können Sie es in einer Bedingung verwenden:

```
>>> m = re.search('bar', 'foo.bar.baz')
>>> m
<_sre.SRE_Match object; span=(4, 7), match='bar'>
```

```
>>> bool(m)
True

>>> if re.search('bar', 'foo.bar.baz'):
...     print('Found a match')
...
Found a match
```

Aber Match-Objekte enthalten auch ziemlich viele nützliche Informationen über das Match. Sie haben bereits einige davon gesehen – die `span=`und `match=`Daten, die der Interpreter anzeigt, wenn er ein Übereinstimmungsobjekt anzeigt. Mit seinen Methoden und Attributen können Sie viel mehr aus einem Match-Objekt herausholen.

Objektmethoden abgleichen

Die folgende Tabelle fasst die Methoden zusammen, die für ein Übereinstimmungsobjekt verfügbar sind `match`:

Methode	Kehrt zurück
`match.group()`	Die angegebene erfasste Gruppe oder Gruppen aus `match`
`match.__getitem__()`	Eine gefangene Gruppe aus `match`
`match.groups()`	Alle erfassten Gruppen aus `match`
`match.groupdict()`	Ein Wörterbuch benannter erfasster Gruppen aus `match`
`match.expand()`	Das Ergebnis der Durchführung von Rückverweissubstitutionen von `match`
`match.start()`	Der Startindex von `match`
`match.end()`	Der Endindex von `match`
`match.span()`	Sowohl der Anfangs- als auch der Endindex von `match`als Tupel

In den folgenden Abschnitten werden diese Methoden ausführlicher beschrieben.

`match.group([<group1>, ...])`

Gibt die angegebene(n) erfasste(n) Gruppe(n) aus einer Übereinstimmung zurück.

Für nummerierte Gruppen, `match.group(n)`gibt die zurück n^{th}Gruppe:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(1)
'foo'
>>> m.group(3)
'baz'
```

Denken Sie daran: Nummerierte erfasste Gruppen sind einsbasiert, nicht nullbasiert.

Wenn Sie Gruppen mit erfassen `(?P<name><regex>)`, dann `match.group(<name>)`gibt die entsprechende benannte Gruppe zurück:

```
>>> m = re.match(r'(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'quux,corge,grault')
>>> m.group('w1')
'quux'
>>> m.group('w3')
'grault'
```

Mit mehr als einem Argument `.group()` gibt ein Tupel aller angegebenen Gruppen zurück. Eine bestimmte Gruppe kann mehrmals erscheinen, und Sie können alle erfassten Gruppen in beliebiger Reihenfolge angeben:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(1, 3)
('foo', 'baz')
>>> m.group(3, 3, 1, 1, 2, 2)
('baz', 'baz', 'foo', 'foo', 'bar', 'bar')

>>> m = re.match(r'(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'quux,corge,grault')
>>> m.group('w3', 'w1', 'w1', 'w2')
('grault', 'quux', 'quux', 'corge')
```

Wenn Sie eine Gruppe angeben, die außerhalb des Bereichs liegt oder nicht vorhanden ist, dann `.group()` erhebt ein `IndexError` Ausnahme:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group

>>> m = re.match(r'(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'quux,corge,grault')
>>> m.group('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
```

Es ist möglich, dass eine Regex in Python als Ganzes übereinstimmt, aber eine Gruppe enthält, die nicht an der Übereinstimmung teilnimmt. In diesem Fall, `.group()` kehrt zurück `None` für die nicht teilnehmende Gruppe. Betrachten Sie dieses Beispiel:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')
>>> m
<_sre.SRE_Match object; span=(0, 8), match='foo,bar,'>
>>> m.group(1, 2)
('foo', 'bar')
```

Diese Regex passt, wie Sie am Match-Objekt sehen können. Die ersten beiden erfassten Gruppen enthalten `'foo'` und `'bar'`, beziehungsweise.

Ein Fragezeichen (`?`) Quantifizierer-Metazeichen folgt jedoch auf die dritte Gruppe, sodass diese Gruppe optional ist. Eine Übereinstimmung tritt auf, wenn nach dem zweiten Komma eine dritte Folge von Wortzeichen folgt (`,`), sondern auch, wenn es keine gibt.

In diesem Fall gibt es das nicht. Es gibt also insgesamt ein Spiel, aber die dritte Gruppe nimmt daran nicht teil. Als Ergebnis, `m.group(3)` ist immer noch definiert und eine gültige Referenz, kehrt aber zurück `None`:

```
>>> print(m.group(3))
None
```

Es kann auch vorkommen, dass eine Gruppe mehrfach am Gesamtspiel teilnimmt. Wenn Sie anrufen `.group()` für diese Gruppennummer, dann wird nur der Teil der Suchzeichenfolge zurückgegeben, der beim letzten Mal übereinstimmte. Die früheren Spiele sind nicht zugänglich:

```
>>> m = re.match(r'(\w{3},)+', 'foo,bar,baz,qux')
>>> m
<_sre.SRE_Match object; span=(0, 12), match='foo,bar,baz,'>
>>> m.group(1)
'baz,'
```

In diesem Beispiel ist die vollständige Übereinstimmung `'foo,bar,baz,'`, wie durch das angezeigte Übereinstimmungsobjekt gezeigt. Jeder von `'foo,'`, `'bar,'`, und `'baz,'`stimmt mit dem überein, was in der Gruppe ist, aber `m.group(1)`gibt nur das letzte Spiel zurück, `'baz,'`.

Wenn Sie anrufen `.group()`mit einem Argument von `0`oder überhaupt kein Argument, dann wird die gesamte Übereinstimmung zurückgegeben:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')

>>> m

<_sre.SRE_Match object; span=(0, 11), match='foo,bar,baz'>



>>> m.group(0)

'foo,bar,baz'

>>> m.group()

'foo,bar,baz'
```

Dies sind dieselben Daten, die der Interpreter im Folgenden anzeigt `match=`wenn es das Übereinstimmungsobjekt anzeigt, wie Sie in **Zeile 3** oben sehen können.

`match.__getitem__(<grp>)`

> Gibt eine erfasste Gruppe aus einem Spiel zurück.

`match.__getitem__(<grp>)`ist identisch mit `match.group(<grp>)`und gibt die durch angegebene einzelne Gruppe zurück `<grp>`:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(2)
'bar'
>>> m.__getitem__(2)
'bar'
```

Wenn `.__getitem__()`repliziert einfach die Funktionalität von `.group()`, warum würden Sie es dann verwenden? Sie würden wahrscheinlich nicht direkt, aber Sie könnten indirekt. Lesen Sie weiter, um zu sehen, warum.

Eine kurze Einführung in magische Methoden

`.__getitem__()`gehört zu einer Sammlung von Methoden in Python, die **magische Methoden** . Dies sind spezielle Methoden, die der Interpreter aufruft, wenn eine Python-Anweisung bestimmte entsprechende syntaktische Elemente enthält.

Hinweis: auch als **Dunder-Methoden** wegen des **doppelten** Unterstrichs **Methodennamens** am Anfang und am Ende des

Später in dieser Serie gibt es mehrere Tutorials zur objektorientierten Programmierung. Dort erfährst du noch viel mehr über magische Methoden.

Die besondere Syntax, die `.__getitem__()`entspricht der Indizierung mit eckigen Klammern. Für jedes Objekt `obj`, wenn Sie den Ausdruck verwenden `obj[n]`, hinter den Kulissen übersetzt Python es leise in einen Aufruf an `.__getitem__()`. Die folgenden Ausdrücke sind effektiv äquivalent:

```
obj[n]
obj.__getitem__(n)
```

Die Syntax `obj[n]`ist nur sinnvoll, wenn a `.__getitem()__`Methode existiert für die Klasse oder den Typ, zu dem `obj`gehört. Genau wie Python interpretiert `obj[n]`hängt dann von der Umsetzung ab `.__getitem__()`für diese Klasse.

Zurück zu Objekten anpassen

Ab Python-Version 3.6 ist die `re`Modul implementiert `.__getitem__()`für Spielobjekte. Die Umsetzung ist so, dass `match.__getitem__(n)`ist das gleiche wie `match.group(n)`.

Das Ergebnis von all dem ist, dass, anstatt anzurufen `.group()`direkt können Sie von einem Match-Objekt aus auf erfasste Gruppen zugreifen, indem Sie stattdessen die Indizierungssyntax mit eckigen Klammern verwenden:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.group(2)
'bar'
>>> m.__getitem__(2)
'bar'
>>> m[2]
'bar'
```

Dies funktioniert auch mit benannten erfassten Gruppen:

```
>>> m = re.match(
...     r'foo,(?P<w1>\w+),(?P<w2>\w+),qux',
...     'foo,bar,baz,qux')
>>> m.group('w2')
'baz'
>>> m['w2']
'baz'
```

Dies ist etwas, das Sie erreichen könnten, indem Sie einfach anrufen `.group()`ausdrücklich, aber es ist trotzdem eine ziemlich verkürzte Schreibweise.

Wenn eine Programmiersprache eine alternative Syntax bereitstellt, die nicht unbedingt erforderlich ist, aber den Ausdruck von etwas auf eine sauberere, leichter lesbare Weise ermöglicht, wird sie als **syntaktischer Zucker** . Für ein Übereinstimmungsobjekt `match[n]`ist syntaktischer Zucker für `match.group(n)`.

Hinweis: Viele Objekte in Python haben eine `.__getitem()__`-Methode definiert, die die Verwendung der Indizierungssyntax mit eckigen Klammern ermöglicht. Diese Funktion ist jedoch nur für Regex-Match-Objekte in Python Version 3.6 oder höher verfügbar.

`match.groups(default=None)`

Gibt alle erfassten Gruppen eines Spiels zurück.

`match.groups()`gibt ein Tupel aller erfassten Gruppen zurück:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.groups()
('foo', 'bar', 'baz')
```

Wie Sie zuvor gesehen haben, wenn eine Gruppe in einer Regex in Python nicht am Gesamtmatch teilnimmt, `.group()`kehrt zurück `None`für diese Gruppe. Standardmäßig, `.groups()`tut es ebenso.

Falls Sie es wollen `.groups()`Um in dieser Situation etwas anderes zurückzugeben, können Sie die verwenden `default`Stichwortargument:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')

>>> m

<_sre.SRE_Match object; span=(0, 8), match='foo,bar,'>

>>> print(m.group(3))

None



>>> m.groups()

('foo', 'bar', None)

>>> m.groups(default='---')

('foo', 'bar', '---')
```

Hier der dritte `(\w+)`Gruppe nimmt nicht am Spiel teil, weil das Fragezeichen (`?`) Metazeichen macht es optional, und die Zeichenfolge `'foo,bar,'`enthält keine dritte Folge von Wortzeichen. Standardmäßig, `m.groups()`kehrt zurück `None`für die dritte Gruppe, wie in **Zeile 8** . In **Zeile 10** können Sie diese Angabe sehen `default='---'`bewirkt, dass es die Zeichenfolge zurückgibt `'---'`stattdessen.

Es gibt keine entsprechende `default`Stichwort für `.group()`. Es kehrt immer zurück `None`für nicht teilnehmende Gruppen.

`match.groupdict(default=None)`

Gibt ein Wörterbuch benannter erfasster Gruppen zurück.

`match.groupdict()` gibt ein Wörterbuch aller benannten Gruppen zurück, die mit erfasst wurden `(?P<name><regex>)` Metazeichenfolge. Die Wörterbuchschlüssel sind die Gruppennamen und die Wörterbuchwerte sind die entsprechenden Gruppenwerte:

```
>>> m = re.match(
...     r'foo,(?P<w1>\w+),(?P<w2>\w+),qux',
...     'foo,bar,baz,qux')
>>> m.groupdict()
{'w1': 'bar', 'w2': 'baz'}
>>> m.groupdict()['w2']
'baz'
```

Wie bei `.groups()`, zum `.groupdict()` das `default` Argument bestimmt den Rückgabewert für nicht teilnehmende Gruppen:

```
>>> m = re.match(
...     r'foo,(?P<w1>\w+),(?P<w2>\w+)?,qux',
...     'foo,bar,,qux')
>>> m.groupdict()
{'w1': 'bar', 'w2': None}
>>> m.groupdict(default='---')
{'w1': 'bar', 'w2': '---'}
```

Wieder die letzte Gruppe `(?P<w2>\w+)` nimmt wegen des Fragezeichens (`?`) Metazeichen. Standardmäßig, `m.groupdict()` kehrt zurück `None` für diese Gruppe, aber Sie können es mit ändern `default` Streit.

`match.expand(<template>)`

Führt Rückverweissubstitutionen aus einer Übereinstimmung durch.

`match.expand(<template>)` gibt die Zeichenfolge zurück, die sich aus der Ausführung der Rückverweissubstitution ergibt `<template>` Genau wie `re.sub()` würdest du:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')

>>> m

<_sre.SRE_Match object; span=(0, 11), match='foo,bar,baz'>

>>> m.groups()

('foo', 'bar', 'baz')



>>> m.expand(r'\2')

'bar'

>>> m.expand(r'[\3] -> [\1]')

'[baz] -> [foo]'



>>> m = re.search(r'(?P<num>\d+)', 'foo123qux')
```

```
>>> m

<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> m.group(1)

'123'



>>> m.expand(r'--- \g<num> ---')

'--- 123 ---'
```

Dies funktioniert für numerische Rückverweise, wie in den **Zeilen 7 und 9** oben, und auch für benannte Rückverweise, wie in **Zeile 18** .

`match.start([<grp>])`

`match.end([<grp>])`

Gibt die Anfangs- und Endindizes der Übereinstimmung zurück.

`match.start()`gibt den Index in der Suchzeichenfolge zurück, wo die Übereinstimmung beginnt, und `match.end()`gibt den Index unmittelbar nach dem Ende der Übereinstimmung zurück:

```
>>> s = 'foo123bar456baz'

>>> m = re.search('\d+', s)

>>> m

<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> m.start()

3

>>> m.end()

6
```

Wenn Python ein Übereinstimmungsobjekt anzeigt, sind dies die mit aufgelisteten Werte `span=`Schlüsselwort, wie in **Zeile 4** oben gezeigt. Sie verhalten sich wie String-Slicing- Werte. Wenn Sie sie also zum Slicen der ursprünglichen Suchzeichenfolge verwenden, sollten Sie die passende Teilzeichenfolge erhalten:

```
>>> m
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> s[m.start():m.end()]
'123'
```

`match.start(<grp>)` und `match.end(<grp>)` gibt die Anfangs- und Endindizes der übereinstimmenden Teilzeichenfolge zurück `<grp>`, die eine nummerierte oder benannte Gruppe sein kann:

```
>>> s = 'foo123bar456baz'
>>> m = re.search(r'(\d+)\D*(?P<num>\d+)', s)

>>> m.group(1)
'123'
>>> m.start(1), m.end(1)
(3, 6)
>>> s[m.start(1):m.end(1)]
'123'

>>> m.group('num')
'456'
>>> m.start('num'), m.end('num')
(9, 12)
>>> s[m.start('num'):m.end('num')]
'456'
```

Wenn die angegebene Gruppe mit einer Nullzeichenfolge übereinstimmt, dann `.start()` und `.end()` sind gleich:

```
>>> m = re.search('foo(\d*)bar', 'foobar')
>>> m[1]
''
>>> m.start(1), m.end(1)
(3, 3)
```

Das macht Sinn, wenn Sie sich daran erinnern `.start()` und `.end()` wirken wie Slicing-Indizes. Jeder String-Slice, bei dem Anfangs- und Endindex gleich sind, ist immer ein leerer String.

Ein Sonderfall tritt auf, wenn die Regex eine Gruppe enthält, die nicht am Match teilnimmt:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')
>>> print(m.group(3))
None
>>> m.start(3), m.end(3)
(-1, -1)
```

Wie Sie zuvor gesehen haben, nimmt die dritte Gruppe in diesem Fall nicht teil. `m.start(3)` und `m.end(3)` sind hier nicht wirklich sinnvoll, also kehren sie zurück `-1`.

`match.span([<grp>])`

Gibt sowohl den Anfangs- als auch den Endindex des Spiels zurück.

`match.span()` gibt sowohl den Anfangs- als auch den Endindex der Übereinstimmung als Tupel zurück. Wenn Sie angegeben haben `<grp>`, dann gilt das Rückgabetupel für die angegebene Gruppe:

```
>>> s = 'foo123bar456baz'
>>> m = re.search(r'(\d+)\D*(?P<num>\d+)', s)
>>> m
<_sre.SRE_Match object; span=(3, 12), match='123bar456'>

>>> m[0]
'123bar456'
```

```
>>> m.span()
(3, 12)

>>> m[1]
'123'
>>> m.span(1)
(3, 6)

>>> m['num']
'456'
>>> m.span('num')
(9, 12)
```

Die folgenden sind effektiv äquivalent:

- `match.span(<grp>)`
- `(match.start(<grp>), match.end(<grp>))`

`match.span()`bietet nur eine bequeme Möglichkeit, beides zu erhalten `match.start()`und `match.end()`in einem Methodenaufruf.

Objektattribute abgleichen

Wie ein kompiliertes reguläres Ausdrucksobjekt verfügt auch ein Übereinstimmungsobjekt über mehrere nützliche Attribute:

Attribut	Bedeutung
`match.pos` `match.endpos`	Die Effektivwerte der `<pos>`und `<endpos>`Argumente für das Spiel
`match.lastindex`	Der Index der letzten erfassten Gruppe
`match.lastgroup`	Der Name der letzten gefangenen Gruppe
`match.re`	Das kompilierte reguläre Ausdrucksobjekt für die Übereinstimmung
`match.string`	Die Suchzeichenfolge für die Übereinstimmung

Die folgenden Abschnitte enthalten weitere Einzelheiten zu diesen Übereinstimmungsobjektattributen.

`match.pos`

`match.endpos`

Enthalten die effektiven Werte von `<pos>`und `<endpos>`für die Suche.

Denken Sie daran, dass einige Methoden, wenn sie für eine kompilierte Regex aufgerufen werden, optional akzeptieren `<pos>`und `<endpos>`Argumente, die die Suche auf einen Teil der angegebenen Suchzeichenfolge beschränken. Auf diese Werte kann über das Match-Objekt zugegriffen werden `.pos`und `.endpos`Attribute:

```
>>> re_obj = re.compile(r'\d+')
>>> m = re_obj.search('foo123bar', 2, 7)
>>> m
<_sre.SRE_Match object; span=(3, 6), match='123'>
>>> m.pos, m.endpos
(2, 7)
```

Wenn die `<pos>`und `<endpos>`Argumente nicht in den Aufruf aufgenommen werden, entweder weil sie weggelassen wurden oder weil die betreffende Funktion sie nicht akzeptiert, dann die `.pos`und `.endpos`Attribute zeigen effektiv den Anfang und das Ende der Zeichenfolge an:

```
>>> re_obj = re.compile(r'\d+')

>>> m = re_obj.search('foo123bar')

>>> m

<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> m.pos, m.endpos

(0, 9)



>>> m = re.search(r'\d+', 'foo123bar')

>>> m

<_sre.SRE_Match object; span=(3, 6), match='123'>

>>> m.pos, m.endpos

(0, 9)
```

Das `re_obj.search()`Anruf oben auf **Leitung 2** nehmen könnte `<pos>`und `<endpos>`Argumente, aber sie sind nicht spezifiziert. Das `re.search()`Anruf auf **Leitung 8** kann sie überhaupt nicht annehmen. In beiden Fällen, `m.pos`und `m.endpos`sind `0`und `9`, die Start- und Endindizes der Suchzeichenfolge `'foo123bar'`.

`match.lastindex`

Enthält den Index der zuletzt erfassten Gruppe.

`match.lastindex`ist gleich dem ganzzahligen Index der letzten erfassten Gruppe:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.lastindex
3
>>> m[m.lastindex]
'baz'
```

In Fällen, in denen die Regex möglicherweise nicht teilnehmende Gruppen enthält, können Sie so feststellen, wie viele Gruppen tatsächlich an der Übereinstimmung teilgenommen haben:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,baz')
>>> m.groups()
('foo', 'bar', 'baz')
>>> m.lastindex, m[m.lastindex]
(3, 'baz')

>>> m = re.search(r'(\w+),(\w+),(\w+)?', 'foo,bar,')
>>> m.groups()
('foo', 'bar', None)
```

```
>>> m.lastindex, m[m.lastindex]
(2, 'bar')
```

Im ersten Beispiel ist die dritte Gruppe, die wegen des Fragezeichens optional ist (`?`) Metazeichen, nimmt an der Übereinstimmung teil. Aber im zweiten Beispiel nicht. Sie können sagen, weil `m.lastindex`ist `3`im ersten Fall u `2`in dieser Sekunde.

Es gibt einen subtilen Punkt, der zu beachten ist `.lastindex`. Es ist nicht immer so, dass die letzte gefundene Gruppe auch die letzte syntaktisch gefundene Gruppe ist. Die Python-Dokumentation gibt dieses Beispiel:

```
>>> m = re.match('((a)(b))', 'ab')
>>> m.groups()
('ab', 'a', 'b')
>>> m.lastindex
1
>>> m[m.lastindex]
'ab'
```

Die äußerste Gruppe ist `((a)(b))`, was passt `'ab'`. Dies ist die erste Gruppe, auf die der Parser trifft, also wird sie zu Gruppe 1. Aber es ist auch die letzte Gruppe, die übereinstimmt, weshalb `m.lastindex`ist `1`.

Die zweite und dritte Gruppe, die der Parser erkennt, sind `(a)`und `(b)`. Das sind Gruppen `2`und `3`, aber sie stimmen vor der Gruppe überein `1`tut.

`match.lastgroup`

> Enthält den Namen der zuletzt erfassten Gruppe.

Wenn die letzte erfasste Gruppe aus der stammt `(?P<name><regex>)`Metazeichenfolge, dann `match.lastgroup`gibt den Namen dieser Gruppe zurück:

```
>>> s = 'foo123bar456baz'
>>> m = re.search(r'(?P<n1>\d+)\D*(?P<n2>\d+)', s)
>>> m.lastgroup
'n2'
```

`match.lastgroup`kehrt zurück `None`wenn die letzte erfasste Gruppe keine benannte Gruppe ist:

```
>>> s = 'foo123bar456baz'

>>> m = re.search(r'(\d+)\D*(\d+)', s)
>>> m.groups()
('123', '456')
>>> print(m.lastgroup)
None

>>> m = re.search(r'\d+\D*\d+', s)
>>> m.groups()
()
>>> print(m.lastgroup)
None
```

Wie oben gezeigt, kann dies entweder daran liegen, dass die letzte erfasste Gruppe keine benannte Gruppe ist, oder daran, dass es überhaupt keine erfassten Gruppen gab.

`match.re`

Enthält das reguläre Ausdrucksobjekt für die Übereinstimmung.

`match.re`enthält das reguläre Ausdrucksobjekt, das die Übereinstimmung erzeugt hat. Dies ist dasselbe Objekt, das Sie erhalten würden, wenn Sie die Regex übergeben würden `re.compile()`:

```
>>> regex = r'(\w+),(\w+),(\w+)'


>>> m1 = re.search(regex, 'foo,bar,baz')

>>> m1

<_sre.SRE_Match object; span=(0, 11), match='foo,bar,baz'>

>>> m1.re

re.compile('(\\w+),(\\w+),(\\w+)')


>>> re_obj = re.compile(regex)

>>> re_obj

re.compile('(\\w+),(\\w+),(\\w+)')

>>> re_obj is m1.re

True


>>> m2 = re_obj.search('qux,quux,corge')

>>> m2

<_sre.SRE_Match object; span=(0, 14), match='qux,quux,corge'>

>>> m2.re

re.compile('(\\w+),(\\w+),(\\w+)')

>>> m2.re is re_obj is m1.re

True
```

Denken Sie daran, dass die `re`-Modul speichert reguläre Ausdrücke, nachdem es sie kompiliert hat, sodass sie bei erneuter Verwendung nicht erneut kompiliert werden müssen. Aus diesem Grund sind, wie die Identitätsvergleiche in den **Zeilen 12 und 20** zeigen, alle verschiedenen regulären Ausdrucksobjekte im obigen Beispiel genau dasselbe Objekt.

Sobald Sie Zugriff auf das reguläre Ausdrucksobjekt für die Übereinstimmung haben, sind auch alle Attribute dieses Objekts verfügbar:

```
>>> m1.re.groups
3
>>> m1.re.pattern
'(\\w+),(\\w+),(\\w+)'
>>> m1.re.pattern == regex
True
>>> m1.re.flags
32
```

Sie können auch jede der Methoden aufrufen, die für ein kompiliertes reguläres Ausdrucksobjekt darauf definiert sind:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.re
re.compile('(\\w+),(\\w+),(\\w+)')

>>> m.re.match('quux,corge,grault')
<_sre.SRE_Match object; span=(0, 17), match='quux,corge,grault'>
```

Hier, `.match()`wird aufgerufen `m.re`um eine weitere Suche mit derselben Regex, aber einer anderen Suchzeichenfolge durchzuführen.

`match.string`

> Enthält die Suchzeichenfolge für eine Übereinstimmung.

`match.string`enthält die Suchzeichenfolge, die das Ziel der Übereinstimmung ist:

```
>>> m = re.search(r'(\w+),(\w+),(\w+)', 'foo,bar,baz')
>>> m.string
'foo,bar,baz'

>>> re_obj = re.compile(r'(\w+),(\w+),(\w+)')
>>> m = re_obj.search('foo,bar,baz')
>>> m.string
'foo,bar,baz'
```

Wie Sie dem Beispiel entnehmen können, ist die `.string`-Attribut ist auch verfügbar, wenn das Übereinstimmungsobjekt von einem kompilierten regulären Ausdrucksobjekt abgeleitet ist.

Fazit

Damit ist Ihre Tour durch Python's abgeschlossen `re`Modul!

Diese Einführungsserie enthält zwei Tutorials zur Verarbeitung regulärer Ausdrücke in Python. durchgearbeitet haben vorherige als auch dieses Tutorial , sollten Sie jetzt wissen, wie Sie:

- Nutzen Sie alle Funktionen, die der `re`Modul bietet
- Vorkompilieren einer Regex in Python
- Extrahieren Sie Informationen aus Übereinstimmungsobjekten

Reguläre Ausdrücke sind extrem vielseitig und leistungsfähig – buchstäblich eine eigene Sprache. Sie werden sie in Ihrer Python-Codierung von unschätzbarem Wert finden.

Hinweis: Die `re`Das Modul ist großartig und wird Ihnen wahrscheinlich in den meisten Fällen gute Dienste leisten. Es gibt jedoch ein alternatives Python-Modul eines Drittanbieters namens `regex`das bietet eine noch größere Übereinstimmung mit regulären Ausdrücken. Mehr dazu erfahren Sie unter `regex`Projektseite .

Als Nächstes in dieser Reihe erfahren Sie, wie Python Konflikte zwischen Bezeichnern in verschiedenen Codebereichen vermeidet. Wie Sie bereits gesehen haben, hat jede Funktion in Python ihren eigenen Namensraum , der sich von denen anderer Funktionen unterscheidet. Im nächsten Tutorial erfahren Sie, wie Namespaces in Python implementiert werden und wie sie den **Gültigkeitsbereich** .