Kostenloses eBook

LERNEN Regular Expressions

Free unaffiliated eBook created from
Stack Overflow contributors.

#regex

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: regular-expressions

It is an unofficial and free Regular Expressions ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Regular Expressions.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit regulären Ausdrücken

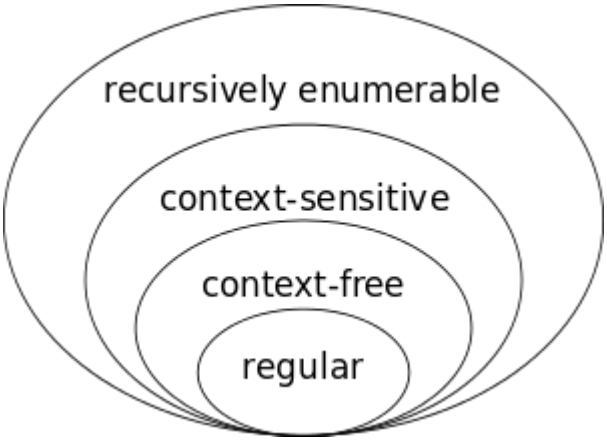
Bemerkungen

Für viele Programmierer ist der *Regex* eine Art magisches Schwert, das sie werfen, um jede Art von Textanalyse zu lösen. Aber dieses Tool ist nichts Magisches, und obwohl es großartig ist bei dem, was es tut, ist es keine vollständige Programmiersprache (*dh* es ist **nicht** Turing-vollständig).

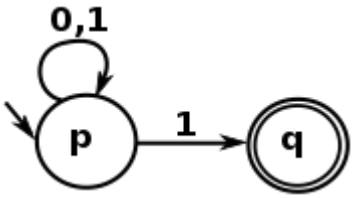
Was bedeutet "regulärer Ausdruck"?

Reguläre Ausdrücke drücken eine durch eine *reguläre Grammatik* definierte Sprache aus, die von einem *nichtdeterministischen endlichen Automaten* (NFA) gelöst werden kann, wobei die Übereinstimmung durch die Zustände dargestellt wird.

Eine *reguläre Grammatik* ist die einfachste Grammatik, wie sie von der *Chomsky-Hierarchie* ausgedrückt wird.



Einfach gesagt, eine reguläre Sprache wird visuell durch das ausgedrückt, was eine NFA ausdrücken kann, und hier ist ein sehr einfaches Beispiel für NFA:



Und die Sprache für *reguläre Ausdrücke* ist eine Textdarstellung eines solchen Automaten. Das letzte Beispiel wird durch die folgende Regex ausgedrückt:

```
^[01]*1$
```

Welches ist eine Zeichenfolge, die mit `0` oder `1` beginnt und `0` oder mehrmals wiederholt wird, die mit einer `1` endet. Mit anderen Worten, es ist ein Regex, um ungerade Zahlen aus ihrer binären Darstellung abzugleichen.

Sind alle Regex tatsächlich eine *reguläre* Grammatik?

Eigentlich nicht. Viele Regex-Engines haben sich verbessert und verwenden *Push-Down-Automaten* , die sich während des Betriebs stapeln lassen und Informationen abrufen können. Diese Automaten definieren, was in Chomskys Hierarchie *kontextfreie* Grammatiken genannt wird . Die häufigste Verwendung von nicht regulären *Regex* ist die Verwendung eines rekursiven Musters für das Übereinstimmen von Klammern.

Ein rekursiver Regex wie der folgende (der in Klammern steht) ist ein Beispiel für eine solche Implementierung:

```
{((?>[^\(\)]+|(?R))*)}
```

(Dieses Beispiel funktioniert nicht mit der `re` Engine von Python, sondern mit der `regex` Engine oder mit der PCRE-Engine).

Ressourcen

Weitere Informationen zur Theorie der regulären Ausdrücke finden Sie in den folgenden vom MIT zur Verfügung gestellten Kursen:

- Automaten, Berechenbarkeit und Komplexität
- Reguläre Ausdrücke und Grammatiken
- Festlegen von Sprachen mit regulären Ausdrücken und kontextfreien Grammatiken

Wenn Sie einen komplexen Regex schreiben oder debuggen, gibt es Online-Tools, mit denen Sie Regex als Automaten visualisieren können, wie zum Beispiel die Debuggex-Site .

Versionen

PCRE

Ausführung	Veröffentlicht
2	2015-01-05
1	1997-06-01

Wird von PHP 4.2.0 (und höher), Delphi XE (und höher), Julia , Notepad ++ verwendet

Perl

Ausführung	Veröffentlicht
1	1987-12-18
2	1988-06-05
3	1989-10-18
4	1991-03-21
5	1994-10-17
6	2009-07-28

.NETZ

Ausführung	Veröffentlicht
1	2002-02-13
4	2010-04-12

Sprachen: C #

Java

Ausführung	Veröffentlicht
4	2002-02-06
5	2004-10-04
7	2011-07-07
SE8	2014-03-18

JavaScript

Ausführung	Veröffentlicht
1.2	1997-06-11

Ausführung	Veröffentlicht
1.8.5	2010-07-27

Python

Ausführung	Veröffentlicht
1.4	1996-10-25
2,0	2000-10-16
3,0	2008-12-03
3.5.2	2016-06-07

Oniguruma

Ausführung	Veröffentlicht
Initiale	2002-02-25
5.9.6	2014-12-12
Onigmo	2015-01-20

Boost

Ausführung	Veröffentlicht
0	1999-12-14
1.61.0	2016-05-13

POSIX

Ausführung	Veröffentlicht
BRE	1997-01-01
EHE	2008-01-01

Sprachen: Bash

Examples

Character Guide

Beachten Sie, dass sich einige Syntaxelemente je nach Ausdruck unterschiedlich verhalten.

Syntax	Beschreibung
`?`	Stimmt mit dem vorhergehenden Zeichen oder Unterausdruck 0 oder 1 überein. Wird auch für nicht erfassende Gruppen und benannte Erfassungsgruppen verwendet.
`*`	Stimmt mit dem vorhergehenden Zeichen oder Unterausdruck mindestens 0 mal überein.
`+`	Ordnen Sie den vorhergehenden Buchstaben oder Unterausdruck mindestens ein Mal zu.
`{n}`	Passen Sie das vorhergehende Zeichen oder den Unterausdruck genau *n*- mal an.
`{min,}`	Entspricht die vorhergehenden Zeichen oder subexpression *min* oder mehr Male.
`{,max}`	Entspricht dem vorhergehenden Zeichen oder Unterausdruck *maximal* oder weniger.
`{min,max}`	Mindestens *min*- mal, maximal jedoch `max` .
`_`	Wenn es in eckigen Klammern steht `to` ; Beispiel: [3-6] stimmt mit den Zeichen 3, 4, 5 oder 6 überein.
`^`	Anfang der Zeichenfolge (oder Zeilenanfang, wenn die Option multiline `/m` angegeben ist) oder negiert eine Liste von Optionen (z. B. in eckigen Klammern `[]`)
`$`	Ende der Zeichenfolge (oder Ende einer Zeile, wenn die Option multiline `/m` angegeben ist).
`( ... )`	Gruppiert Unterausdrücke und erfasst übereinstimmenden Inhalt in speziellen Variablen (`\1` , `\2` usw.), die später innerhalb desselben regulären Ausdrucks verwendet werden können. Beispielsweise stimmt `(\w+)\s\1\s` mit der Wortwiederholung überein
`(?<name> ... )`	Gruppiert Unterausdrücke und erfasst sie in einer benannten Gruppe
`(?: ... )`	Gruppiert Unterausdrücke ohne Erfassung
`.`	Stimmt mit jedem Zeichen außer Zeilenumbrüchen (`\n` und normalerweise `\r`) `\r`

Syntax	Beschreibung
	.
`[ ... ]`	Jedes Zeichen zwischen diesen Klammern sollte einmal übereinstimmen. NB: `^` Nach der offenen Klammer wird dieser Effekt aufgehoben. `-` Wenn Sie in den Klammern vorkommen, können Sie einen Wertebereich angeben (es sei denn, es handelt sich um das erste oder letzte Zeichen, in diesem Fall handelt es sich lediglich um einen regulären Bindestrich).
`\`	Escape das folgende Zeichen. Wird auch in Meta-Sequenzen verwendet - Regex-Token mit besonderer Bedeutung.
`\$`	Dollar (dh ein entkommenes Sonderzeichen)
`\(`	offene Klammer (dh ein Escape-Sonderzeichen)
`\)`	schließende Klammer (dh ein Escape-Sonderzeichen)
`\*`	Sternchen (dh ein Escape-Sonderzeichen)
`\.`	Punkt (dh ein Escape-Sonderzeichen)
`\?`	Fragezeichen (dh ein Escape-Sonderzeichen)
`\[`	linke (offene) eckige Klammer (dh ein nicht zugeordnetes Sonderzeichen)
`\\`	Backslash (dh ein Escape-Sonderzeichen)
`\]`	rechtes (enges) eckiges Klammerzeichen (dh ein hinterlegtes Sonderzeichen)
`\^`	Caret (dh ein entfliehenes Sonderzeichen)
`\{`	linke (offene) geschweifte Klammer / Klammer
`|`	Pipe (dh ein entgangenes Sonderzeichen)
`\}`	rechtes (enges) geschweiftes Bracket / Brace (dh ein entgangenes Sonderzeichen)
`\+`	Plus (dh ein Escape-Sonderzeichen)
`\A`	Anfang einer Zeichenfolge
`\Z`	Ende einer Zeichenfolge
`\z`	absolut von einer schnur
`\b`	Wort (alphanumerische Reihenfolge)
`\1` , `\2`	Rückverweise auf zuvor übereinstimmende Unterausdrücke, gruppiert nach `()` ,

Syntax	Beschreibung
usw.	`\1` bedeutet die erste Übereinstimmung, `\2` bedeutet zweite Übereinstimmung usw.
`[\b]`	Rücktaste - Wenn sich `\b` innerhalb einer Zeichenklasse (`[]`) befindet, entspricht die Rücktaste
`\B`	negated `\b` - passt an jeder Position zwischen Zwei-Wort-Zeichen sowie an einer beliebigen Position zwischen zwei Nicht-Wort-Zeichen
`\D`	Nicht-Ziffer
`\d`	Ziffer
`\e`	Flucht
`\f`	Formularvorschub
`\n`	Zeilenvorschub
`\r`	Wagenrücklauf
`\S`	Nicht-Weißraum
`\s`	Leerraum
`\t`	Tab
`\v`	vertikale Registerkarte
`\W`	Nicht-Wort
`\w`	Wort (dh alphanumerisches Zeichen)
`{ ... }`	benannter Zeichensatz
`|`	oder; dh umreißt die vorherigen und vorhergehenden Optionen.

Erste Schritte mit regulären Ausdrücken online lesen:
https://riptutorial.com/de/regex/topic/259/erste-schritte-mit-regularen-ausdrucken

Kapitel 2: Ankerzeichen: Caret (^)

Bemerkungen

Terminologie

Das Caret-Zeichen (^) wird auch mit folgenden Begriffen bezeichnet:

- Hut
- Steuerung
- Aufwärtspfeil
- Chevron
- Akzent auf die Zirkulation

Verwendungszweck

Es hat zwei Verwendungsmöglichkeiten in regulären Ausdrücken:

- Bezeichnet den Zeilenanfang
- Wird es unmittelbar nach einer eckigen Klammer (`[^`) verwendet, negiert es die Menge der zulässigen Zeichen (dh `[123]` bedeutet, dass das Zeichen 1, 2 oder 3 zulässig ist, während die Anweisung `[^123]` jedes andere Zeichen als 1 bedeutet , 2 oder 3 ist erlaubt.

Zeichenflucht

Um ein Caret ohne besondere Bedeutung auszudrücken, sollte es durch einen Backslash vorangestellt werden. dh `\^` .

Examples

Anfang der Zeile

Bei mehreren Leitungen `(?m)` Modifier ausgeschaltet ist, `^` entspricht nur der Anfang der Eingabezeichenfolge:

Für die Regex

```
^He
```

Die folgenden Eingabezeichenfolgen stimmen überein:

- `Hedgehog\nFirst line\nLast line`
- `Help me, please`
- `He`

Und die folgenden Eingabezeichenfolgen stimmen **nicht** überein:

- `First line\nHedgehog\nLast line`
- `IHedgehog`
- `Hedgehog` (aufgrund von Leerzeichen)

Wenn mit mehreren Leitungen `(?m)` Modifikator eingeschaltet wird , `^` entspricht Anfang der jede Zeile:

```
^He
```

Das Obige würde mit jeder Eingabezeichenfolge übereinstimmen, die eine mit `He` beginnende Zeile enthält.

Wenn Sie `\n` als neues Zeilenzeichen betrachten, stimmen die folgenden Zeilen überein:

- `Hello`
- `First line\nHedgehog\nLast line` (nur zweite Zeile)
- `My\nText\nIs\nHere` (nur letzte Zeile)

Und die folgenden Eingabezeichenfolgen stimmen **nicht** überein:

- `Camden Hells Brewery`
- `Helmet` (wegen Leerzeichen))

Leerzeilen mit `^`

Ein anderer typischer Anwendungsfall für Caret ist das Abgleichen von Leerzeilen (oder eine leere Zeichenfolge, wenn der Mehrzeilenmodifikator deaktiviert ist).

Um eine leere Zeile abzugleichen (mehrzeilige Zeile **ein**), wird ein Caret-Zeichen neben einem `$` das ein anderes Ankerzeichen ist, das die Position am Zeilenende darstellt (Ankerzeichen: Dollar ($)). Daher entspricht der folgende reguläre Ausdruck einer leeren Zeile:

```
^$
```

Dem Caret-Charakter entkommen

Wenn Sie das Zeichen `^` in einer Zeichenklasse (Zeichenklassen) verwenden müssen, setzen Sie es entweder an einen anderen Ort als den Anfang der Klasse:

```
[12^3]
```

Oder entgehen Sie dem `^` mit einem umgekehrten Schrägstrich `\` :

```
[\^123]
```

Wenn Sie das Caret-Zeichen außerhalb einer Zeichenklasse abgleichen möchten, müssen Sie es umgehen:

```
\^
```

Dies verhindert, dass das `^` Ankerzeichen interpretiert wird, das den Anfang der Zeichenfolge / Zeile darstellt.

Vergleich Anfang des Linienankers und Beginn des Schnurankers

Während viele Leute denken, dass `^` den Beginn einer Zeichenfolge bedeutet, bedeutet dies eigentlich den Beginn einer Zeile. Verwenden Sie für einen tatsächlichen Start des String-Ankers `\A`

Die Zeichenfolge `hello\nworld` (oder deutlicher)

```
hello
world
```

Wäre durch die regulären Ausdrücke `^h` , `^w` und `\Ah` , nicht aber durch `\Aw`

Multiline-Modifikator

Standardmäßig ist die caret `^` entspricht metacharacter die **Position** vor dem ersten Zeichen in der Zeichenkette.

In Anbetracht der Zeichenfolge " **charsequence** ", die auf die folgenden Muster angewendet wird: `/^char/` & `/^sequence/` , versucht die Engine wie folgt zu passen:

- `/^char/`
    - **^** - Zeichenfolge
    - **c** - `c` harsequenz
    - **h** - `ch` arsequenz
    - **a** - `cha` rsequenz
    - **r** - `char` Sequenz

  Übereinstimmung gefunden

- `/^sequence/`
    - **^** - Zeichenfolge
    - **s** - Zeichenfolge

  Spiel nicht gefunden

Das gleiche Verhalten wird auch dann angewandt werden , wenn die Zeichenfolge enthält *Leitungsabschlüsse,* wie beispielsweise `\r?\n` . Es wird nur die Position am Anfang der

Zeichenfolge abgeglichen.

Zum Beispiel:

```
/^/g
```

┊ char \ r \ n
\ r \ n
Sequenz

Wenn Sie jedoch nach jedem Zeilenabschlusszeichen eine Übereinstimmung herstellen müssen, müssen Sie den **Multilinienmodus** (`//m` , `(?m)`) innerhalb Ihres Musters einstellen. Auf diese Weise stimmt das Caret `^` mit "dem Anfang jeder Zeile" überein, was der Position am Anfang der Zeichenkette und den Positionen **unmittelbar nach** [1] den Zeilenabschlüssen entspricht.

[1] In einigen Versionen (Java, PCRE, ...) stimmt `^` nach dem Zeilenabschlusszeichen nicht überein, wenn das Zeilenendezeichen das letzte in der Zeichenfolge ist.

Zum Beispiel:

```
/^/gm
```

┊ char \ r \ n
\ R \ n
Folge

Einige reguläre Ausdrücke, die den Mehrzeiligen-Modifikator unterstützen:

- Java

```
Pattern pattern = Pattern.compile("(?m)^abc");
Pattern pattern = Pattern.compile("^abc", Pattern.MULTILINE);
```

- .NETZ

```
var abcRegex = new Regex("(?m)^abc");
var abdRegex = new Regex("^abc", RegexOptions.Multiline)
```

- PCRE

```
/(?m)^abc/
/^abc/m
```

- Python 2 & 3 (eingebautes `re` Modul)

```
abc_regex = re.compile("(?m)^abc");
abc_regex = re.compile("^abc", re.MULTILINE);
```

Ankerzeichen: Caret (^) online lesen: https://riptutorial.com/de/regex/topic/452/ankerzeichen--

caret----

Kapitel 3: Ankerzeichen: Dollar ($)

Bemerkungen

Viele Regex-Engines verwenden einen "mehrzeiligen" Modus , um mehrere Zeilen in einer Datei unabhängig voneinander zu durchsuchen.

Wenn Sie also `$` , stimmen diese Engines mit den Endungen aller Zeilen überein. Engines, die diesen Mehrzeilenmodus nicht verwenden, stimmen jedoch nur mit der letzten Position der für die Suche angegebenen Zeichenfolge überein.

Examples

Bringen Sie einen Buchstaben am Ende einer Zeile oder Zeichenfolge zusammen

```
g$
```

Das obige stimmt mit einem Buchstaben (dem Buchstaben `g`) am Ende eines *Strings* in den meisten Regex-Engines überein (nicht in Oniguruma , wo der Anker `$` standardmäßig mit dem *Zeilenende übereinstimmt* und der Modifizierer `m` (*MULTILINE*) verwendet wird, um ein `.` in den meisten anderen NFA Regex - Varianten) alle Zeichen einschließlich Zeilenumbruch Zeichen, als DOTALL Modifikator entsprechen. Der Anker `$` stimmt mit dem ersten Vorkommen eines Buchstaben `g` vor dem Ende der folgenden Zeichenfolgen überein:

In den folgenden Sätzen stimmen nur die **fett** gedruckten Buchstaben überein:

> Anker sind Zeichen , die in der Tat keine Zeichen in einem strin **g** entsprechen
>
> Ihr Ziel ist es, eine bestimmte Position in dieser Saite zu finden.
>
> Bob war helpin **g**
>
> Aber seine Bearbeitung führte Beispiele ein, die nicht übereinstimmten!

In den meisten regulären Ausdrucksvarianten kann `$` Anchor auch vor einem Zeilenumbruch- oder Zeilenumbruchzeichen (Sequenz) im *MULTILINE*- Modus *übereinstimmen* , wobei `$` am Ende jeder Zeile und nicht nur am Ende einer Zeichenfolge übereinstimmt. Wenn Sie beispielsweise `g$` als Regex verwenden, stimmen die kursiven Zeichen in der folgenden Zeichenfolge im mehrzeiligen Modus überein:

```
tvxlt obofh necpu riist *g*\n aelxk zlhdx lyogu vcbke pzyay wtsea wbrju jzt*g*\n drosf ywhed bykie
lqmzg wgyhc lg\n qewrx ozrvm jwenx
```

Ankerzeichen: Dollar ($) online lesen: https://riptutorial.com/de/regex/topic/1603/ankerzeichen--dollar----

Kapitel 4: Atomische Gruppierung

Einführung

Bei regulären, nicht erfassenden Gruppen kann die Engine erneut in die Gruppe eintreten und versuchen, etwas anderes abzugleichen (z. B. eine andere Alternative oder weniger Zeichen, wenn ein Quantifizierer verwendet wird).

Atomare Gruppen unterscheiden sich von regulären Gruppen ohne Capturing darin, dass das Zurückverfolgen verboten ist. Sobald die Gruppe beendet ist, werden alle Backtracking-Informationen verworfen, sodass keine alternativen Übereinstimmungen versucht werden können.

Bemerkungen

Ein Possessiv-Quantifizierer verhält sich wie eine atomare Gruppe, da die Engine keinen Token oder eine Gruppe zurückverfolgen kann.

Die folgenden Funktionen sind äquivalent, auch wenn einige schneller sind als andere:

```
a*+abc
(?>a*)abc
(?:a+)*+abc
(?:a)*+abc
(?:a*)*+abc
(?:a*)++abc
```

Examples

Gruppieren mit (?>)

Verwenden einer Atomic Group

Atomgruppen haben das Format `(?>...)` mit einem `?>` Nach dem geöffneten Paren.

Betrachten Sie den folgenden Beispieltext:

```
ABC
```

Der reguläre Ausdruck versucht, an Position 0 des Textes, die vor dem `A` in `ABC` liegt, zu passen.

Wenn ein Ausdruck verwendet wird, bei dem die Groß- / Kleinschreibung nicht beachtet wird `(?>a*)abc` , würde das Zeichen `(?>a*)` mit dem Zeichen 1 `A` übereinstimmen

```
BC
```

als verbleibender Text Die Gruppe `(?>a*)` wird beendet, und `abc` wird für den verbleibenden Text versucht, der nicht übereinstimmt.

Die Engine kann sich nicht in die Atomgruppe zurückverfolgen, so dass der aktuelle Durchlauf fehlschlägt. Die Engine bewegt sich zur nächsten Position im Text, die sich an Position 1 befinden würde, dh hinter dem `A` und vor dem `B` von `ABC` .

Der Regex `(?>a*)abc` wird erneut versucht, und `(?>a*)` entspricht 0 mal `A` und verlässt den Vorgang

```
BC
```

als verbleibender Text Die Gruppe `(?>a*)` wird beendet und es wird versucht, `abc` auszuführen, was fehlschlägt.

Wieder kann sich die Engine nicht in die Atomgruppe zurückverfolgen, so dass der aktuelle Durchlauf fehlschlägt. Der Regex schlägt weiterhin fehl, bis alle Positionen im Text erschöpft sind.

Verwenden einer nicht-atomaren Gruppe

Regelmäßige, nicht erfassende Gruppen haben das Format `(?:...)` mit einem `?:` Nach dem offenen Paren.

Wenn der gleiche Beispieltext verwendet wird, aber stattdessen der Groß- / Kleinschreibung-abhängige Ausdruck `(?:a*)abc` verwendet wird, würde eine Übereinstimmung auftreten, da das Zurückverfolgen zulässig ist.

Zuerst wird `(?:a*)` den Buchstaben `A` im Text verbrauchen

```
ABC
```

Verlassen

```
BC
```

als verbleibender Text Die Gruppe `(?:a*)` wird beendet, und `abc` wird für den verbleibenden Text versucht, der nicht übereinstimmt.

Die Engine geht zurück in die Gruppe `(?:a*)` und versucht, 1 weniger Zeichen zu finden: Anstatt 1 `A` Zeichen zu finden, versucht sie, 0 `A` Zeichen zu finden, und die Gruppe `(?:a*)` wird beendet. Diese Blätter

```
ABC
```

als verbleibender Text Der reguläre Ausdruck `abc` ist jetzt in der Lage, den verbleibenden Text erfolgreich abzugleichen.

Anderer Beispieltext

Betrachten Sie diesen Beispieltext mit sowohl atomaren als auch nicht atomaren Gruppen (wiederum ohne Berücksichtigung der Groß- und Kleinschreibung):

```
AAAABC
```

Der `AAAABC` versucht, an Position 0 des Textes, die vor dem ersten `A` in `AAAABC` .

Das Muster, das die Atomgruppe `(?>a*)abc` , kann **nicht** übereinstimmen und verhält sich fast identisch mit dem oben genannten atomaren `ABC` Beispiel: Alle 4 der `A` Zeichen werden zuerst mit `(?>a*)` abgeglichen (wobei `BC` als verbleibender Text, der abgeglichen werden soll), und `abc` nicht mit diesem Text übereinstimmen. Die Gruppe kann **nicht** erneut eingegeben werden, daher schlägt die Übereinstimmung fehl.

Das Muster, das die nicht-atomare Gruppe `(?:a*)abc` , **kann** übereinstimmen und verhält sich ähnlich wie das nicht-atomare `ABC` Beispiel oben: Alle 4 der `A` Zeichen werden zuerst mit `(?:a*)` abgeglichen `BC` als verbleibender Text) und `abc` nicht mit diesem Text übereinstimmen. Die Gruppe **ist** in der Lage neu eingegeben werden, so dass man weniger `A` versucht wird: 3 `A` - Zeichen werden anstelle von 4 (Abfahrt abgestimmt `ABC` als verbleibender Text Übereinstimmen) und `abc` in der Lage , auf diesem Text erfolgreich übereinstimmen.

Atomische Gruppierung online lesen: https://riptutorial.com/de/regex/topic/8770/atomische-gruppierung

Kapitel 5: Benannte Fanggruppen

Syntax

- Erstellen Sie eine benannte Erfassungsgruppe (`x` ist das Muster, das Sie erfassen möchten):

  (? 'name'X) (? X) (? PX)

- Verweisen Sie auf eine benannte Capture-Gruppe:

  $ {name} \ {name} g \ {name}

Bemerkungen

Python und Java erlauben nicht, dass mehrere Gruppen denselben Namen verwenden.

Examples

Wie eine benannte Capture-Gruppe aussieht

In Anbetracht der Geschmacksrichtungen kann die benannte Capture-Gruppe folgendermaßen aussehen:

```
(?'name'X)
(?<name>X)
(?P<name>X)
```

Dabei ist `x` das Muster, das Sie erfassen möchten. Betrachten wir die folgende Zeichenfolge:

> Es war einmal ein *hübsches kleines Mädchen* ...
>
> Es war einmal ein *Einhorn mit Hut* ...
>
> Es war einmal ein *Boot mit Piratenflagge* ...

In dem ich das Thema (*kursiv)* jeder Zeile erfassen möchte. Ich verwende den folgenden Ausdruck `.* was a (?<subject>[\w ]+)[.]{3} .`

Das übereinstimmende Ergebnis gilt:

```
MATCH 1
subject    [29-47]    `pretty little girl`
MATCH 2
subject    [80-99]    `unicorn with an hat`
MATCH 3
subject    [132-155]    `boat with a pirate flag`
```

Verweisen Sie auf eine benannte Capture-Gruppe

Wie Sie vielleicht wissen (oder nicht wissen), können Sie auf eine Capture-Gruppe verweisen mit:

```
$1
```

`1` ist die Gruppennummer.

Auf dieselbe Weise können Sie auf eine benannte Capture-Gruppe mit folgendem verweisen:

```
${name}
\{name}
g\{name}
```

Nehmen wir das vorstehende Beispiel und ersetzen Sie die Übereinstimmungen mit

```
The hero of the story is a ${subject}.
```

Das Ergebnis, das wir erhalten werden, ist:

```
The hero of the story is a pretty little girl.
The hero of the story is a unicorn with an hat.
The hero of the story is a boat with a pirate flag.
```

Benannte Fanggruppen online lesen: https://riptutorial.com/de/regex/topic/744/benannte-fanggruppen

Kapitel 6: Charakterklassen

Bemerkungen

Einfache Klassen

Regex	Streichhölzer
`[abc]`	Jedes der folgenden Zeichen: `a` , `b` oder `c`
`[az]`	Jedes Zeichen von `a` bis `z` , **inklusive** (dies wird als ein *Bereich)*
`[0-9]`	Jede Ziffer von `0` bis **einschließlich** `9`

Gemeinsame Klassen

Einige Gruppen / Zeichenbereiche werden so oft verwendet, dass sie spezielle Abkürzungen haben:

Regex	Streichhölzer
`\w`	Alphanumerische Zeichen plus Unterstrich (auch "Wortzeichen" genannt)
`\W`	Nicht-Wort-Zeichen (wie `[^\w]`)
`\d`	Ziffern (*breiter* als `[0-9]` da persische Ziffern, indische usw. enthalten sind)
`\D`	Nicht-Ziffern (*kürzer* als `[^0-9]` da persische Ziffern abgelehnt werden, indische usw.)
`\s`	Leerzeichen (Leerzeichen, Tabulatoren usw.) **Hinweis** : kann je nach Engine / Kontext variieren
`\S`	Zeichen, die keine Leerzeichen sind

Klassen negieren

Ein Caret (^) nach der öffnenden eckigen Klammer wirkt als Negation der nachfolgenden Zeichen. Dies entspricht allen Zeichen, die nicht zur Zeichenklasse gehören.

Negierte Zeichenklassen stimmen auch mit Zeilenumbruchzeichen überein. Wenn diese nicht übereinstimmen sollen, müssen die spezifischen Zeilenumbruchzeichen der Klasse hinzugefügt

werden (\ r und / oder \ n).

Regex	Streichhölzer
`[^AB]`	Jedes **andere** Zeichen als `A` und `B`
`[^\d]`	Jedes Zeichen **außer** Ziffern

Examples

Die Grundlagen

Angenommen, wir haben eine Liste von Teams, die wie folgt benannt werden: `Team A` , `Team B` , ..., `Team Z` Dann:

- `Team [AB]` : Dies entspricht entweder `Team A` oder `Team B`
- `Team [^AB]` : Dies passt zu jedem Team **außer** `Team A` oder `Team B`

Oft müssen wir Zeichen finden, die in irgendeinem oder einem anderen Kontext (wie Buchstaben von `A` bis `Z`) "zusammengehören".

Passen Sie verschiedene, ähnliche Wörter an

Betrachten Sie die Zeichenklasse `[aeiou]` . Diese Zeichenklasse kann in einem regulären Ausdruck verwendet werden, um einen Satz ähnlich geschriebener Wörter zu finden.

`b[aeiou]t` passt zu:

- Fledermaus
- Wette
- bisschen
- bot
- aber

Es passt nicht:

- Kampf
- btt
- bt

Charakterklassen entsprechen für sich jeweils nur einen Charakter.

Nicht-Alphanumerik-Abgleich (negierte Zeichenklasse)

```
[^0-9a-zA-Z]
```

Dies entspricht allen Zeichen, die weder Ziffern noch Buchstaben sind (alphanumerische

Zeichen). Wenn der Unterstrich _ auch negiert werden soll, kann der Ausdruck folgendermaßen verkürzt werden:

```
[^\w]
```

Oder:

```
\W
```

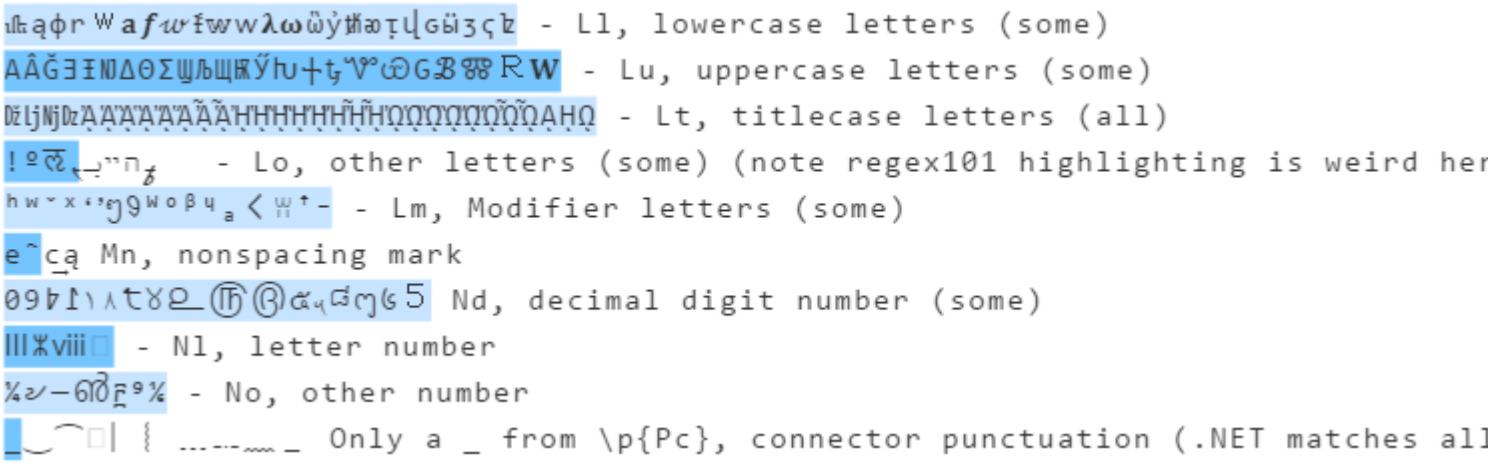
In den folgenden Sätzen:

1. Hi was geht?
2. Ich kann nicht bis 2017 warten !!!

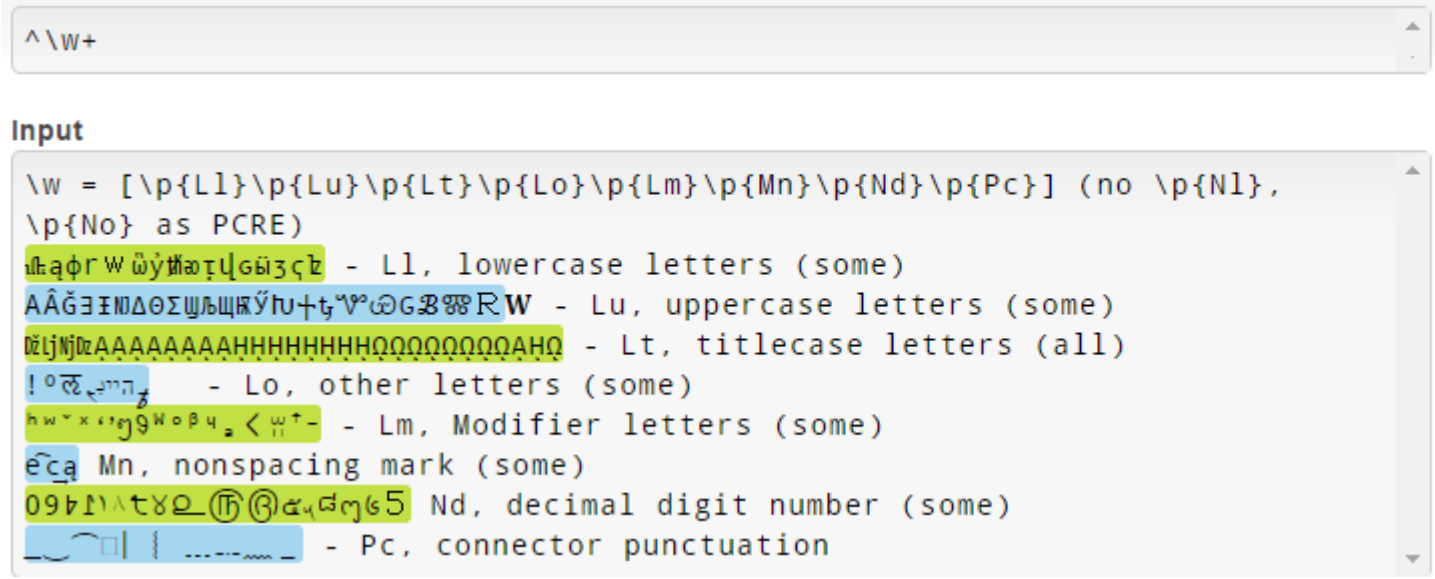
Die folgenden Zeichen stimmen überein:

1. , , , ' , ? und das Zeilenende-Zeichen.
2. ' , ! und das Zeilenende-Zeichen.

UNICODE HINWEIS
Beachten Sie, dass einige Varianten mit Unterstützung für Unicode-Zeichensätze `\w` und `\W` möglicherweise als `[\p{L}\p{N}_]` und `[^\p{L}\p{N}_]` interpretieren, was andere Unicode-Buchstaben bedeutet und numerische Zeichen werden ebenfalls eingeschlossen (siehe PCRE-Dokumente). Hier ist ein PCRE `\w` Test :

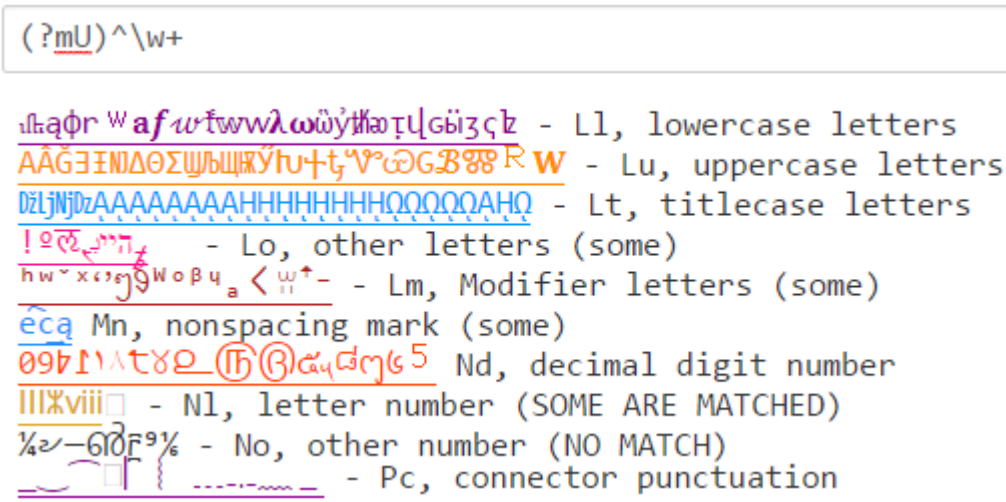


In .NET `\w` = `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Mn}\p{Nd}\p{Pc}]` , Beachten Sie, dass es im Gegensatz zu PCRE nicht zu `\p{Nl}` und `\p{No}` passt (siehe die Dokumentation zu `\w` .NET):



Beachten Sie, dass Unicode 3.1-Kleinbuchstaben (wie ᵫᵬᵭᵮᵯᵰᵱᵲᵳᵴ) aus irgendeinem Grund nicht übereinstimmen.

Java `(?U)\w` `\w` stimmt mit einer Mischung von `\w` überein, die in PCRE und .NET übereinstimmen:



Nicht-Ziffern-Übereinstimmung (negierte Zeichenklasse)

```
[^0-9]
```

Dies entspricht allen Zeichen, die keine ASCII-Ziffern sind.

Wenn Unicode-Ziffern auch negiert werden sollen, kann der folgende Ausdruck verwendet werden, je nach Ihren Geschmacks- / Spracheinstellungen:

```
[^\d]
```

Dies kann verkürzt werden auf:

```
\D
```

Möglicherweise müssen Sie die Unterstützung der Unicode-Zeichen-Eigenschaften explizit mithilfe des Modifizierers `u` oder in einigen Sprachen programmgesteuert aktivieren. Dies ist jedoch möglicherweise nicht offensichtlich. Um die Absicht explizit zu vermitteln, kann das folgende Konstrukt verwendet werden (wenn Support verfügbar ist):

```
\P{N}
```

Was *definitionsgemäß* bedeutet: jedes Zeichen, das in keinem Skript ein numerisches Zeichen ist. In einem negierten Zeichenbereich können Sie Folgendes verwenden:

```
[^\p{N}]
```

In den folgenden Sätzen:

1. Hi was geht?
2. Ich kann nicht bis 2017 warten !!!

Die folgenden Zeichen werden übereinstimmen:

1. `,` , `'` , `?` , das Zeilenendezeichen und alle Buchstaben (Klein- und Großbuchstaben).
2. `'` , `!` , das Zeilenendezeichen und alle Buchstaben (Klein- und Großbuchstaben).

Charakterklasse und häufige Probleme des Anfängers

1. Zeichenklasse

Die Zeichenklasse wird mit `[]` . Inhalt innerhalb einer Zeichenklasse wird `single character separately` als `single character separately` behandelt. Nehmen wir beispielsweise an, wir verwenden

```
[12345]
```

Im obigen Beispiel bedeutet es Übereinstimmung `1 or 2 or 3 or 4 or 5` . In einfachen Worten kann es als `or condition for single characters` (**Betonung auf einzelne Zeichen**)

1.1 Vorsicht

- In der Zeichenklasse gibt es kein Konzept, eine Zeichenfolge zu finden. Wenn Sie also Regex `[cat]` , bedeutet dies nicht, dass es wörtlich mit dem Wort `cat` übereinstimmt, sondern mit `c` oder `a` oder `t` . Dies ist ein sehr verbreitetes Missverständnis, das bei Menschen besteht, die neuerdings Regex sind.
- Manchmal benutzen Leute `|` (alternativ) innerhalb der Charakterklasse wird angenommen, dass sie als `OR condition` die falsch ist. zB bedeutet `[a|b]` eigentlich Übereinstimmung mit `a` oder `|` (wörtlich) oder `b` .

2. Bereich in Zeichenklasse

Der Bereich in der Zeichenklasse wird mit `-` sign bezeichnet. Nehmen wir an, wir möchten ein beliebiges Zeichen in den englischen Alphabeten `A` bis `Z` . Dies kann mit der folgenden Zeichenklasse erfolgen

```
[A-Z]
```

Dies kann für jeden gültigen ASCII- oder Unicode-Bereich erfolgen. Zu den am häufigsten verwendeten Bereichen gehören `[AZ]` , `[az]` oder `[0-9]` . Darüber hinaus können diese Bereiche in der Zeichenklasse als kombiniert werden

```
[A-Za-z0-9]
```

Dies bedeutet, dass jedes Zeichen im Bereich von `A to Z` oder von `a to z` oder von `0 to 9` übereinstimmt. Die Bestellung kann alles sein. Das obige ist also äquivalent zu `[a-zA-Z0-9]` , solange der von Ihnen definierte Bereich korrekt ist.

2.1 Vorsicht

- Beim Schreiben von Bereichen für `A` bis `Z` schreiben die Leute es manchmal als `[Az]` . Dies ist in den meisten Fällen falsch, weil wir `z` anstelle von `Z` . Dies bedeutet also, dass jedes Zeichen aus dem ASCII-Bereich `65` (von A) bis `122` (von z) übereinstimmt, das viele unbeabsichtigte Zeichen nach dem ASCII-Bereich `90` (von Z) enthält. **JEDOCH** `[Az]` kann verwendet werden , um alle entsprechen `[a-zA-Z]` Buchstaben in POSIX-Stil regex , wenn Sortierung für eine bestimmte Sprache eingestellt ist. `[[ "ABCEDEF[]_abcdef" =~ ([Az]+) ]] && echo "${BASH_REMATCH[1]}"` auf Cygwin mit `LC_COLLATE="en_US.UTF-8"` ergibt `ABCEDF` . Wenn Sie `LC_COLLATE` auf `C` (auf Cygwin, mit `export`), wird der erwartete `ABCEDEF[]_abcdef` .

- Bedeutung `-` innerhalb von Zeichenklasse ist etwas Besonderes. Es bezeichnet den oben erläuterten Bereich. *Was ist, wenn wir `-` buchstäblich übereinstimmen wollen?* Wir können es nicht irgendwo einsetzen, sonst werden Bereiche angegeben, wenn es zwischen zwei Zeichen steht. In diesem Fall müssen Sie `-` am Anfang einer Zeichenklasse wie `[-AZ]` oder am Ende einer Zeichenklasse wie `[AZ-]` oder es `escape it` wenn Sie es wie `[AZ\-az]` in der Mitte verwenden `[AZ\-az]` .

3. Negierte Zeichenklasse

Negierte Zeichenklasse wird mit `[^..]` . Das Caret-Zeichen `^` für ein beliebiges Zeichen außer dem in der Zeichenklasse vorhandenen Zeichen. z.B

```
[^cat]
```

bedeutet, dass jedes Zeichen außer `c` oder `a` oder `t` übereinstimmt.

3.1 Wort der Vorsicht

- Die Bedeutung des Caret-Zeichens `^` wird der Negation nur dann zugeordnet, wenn die Zeichenklasse beginnt. Wenn es an einer anderen Stelle in der Zeichenklasse steht, wird es als wörtliche Einfügemarke ohne besondere Bedeutung behandelt.

- Einige Leute schreiben Regex wie `[^]` . In den meisten Regex-Engines gibt dies einen Fehler aus. Wenn Sie `^` in der Startposition verwenden, erwartet Sie mindestens ein Zeichen, das negiert werden soll. In *JavaScript* ist dies jedoch ein gültiges Konstrukt, das auf *alles andere als auf nichts* passt, dh auf jedes mögliche Symbol (außer Diakritik, zumindest in ES5).

POSIX-Zeichenklassen

POSIX-Zeichenklassen sind vordefinierte Sequenzen für einen bestimmten Zeichensatz.

Zeichenklasse	Beschreibung
`[:alpha:]`	Alphabetische Zeichen
`[:alnum:]`	Alphabetische Zeichen und Ziffern
`[:digit:]`	Ziffern
`[:xdigit:]`	Hexadezimal-Ziffern
`[:blank:]`	Leerzeichen und Tab
`[:cntrl:]`	Steuerzeichen
`[:graph:]`	Sichtbare Zeichen (außer Leerzeichen und Steuerzeichen)
`[:print:]`	Sichtbare Zeichen und Leerzeichen
`[:lower:]`	Kleinbuchstaben
`[:upper:]`	Großbuchstaben
`[:punct:]`	Interpunktion und Symbole
`[:space:]`	Alle Leerzeichen, einschließlich Zeilenumbrüche

Je nach Implementierung und / oder Gebietsschema können zusätzliche Zeichenklassen verfügbar sein.

Zeichenklasse	Beschreibung
`[:<:]`	Anfang des Wortes
`[:>:]`	Ende des Wortes
`[:ascii:]`	ASCII-Zeichen
`[:word:]`	Buchstaben, Ziffern und Unterstrich. Entspricht `\w`

Um die innere Klammerfolge (auch als Zeichenklasse bezeichnet) zu verwenden, sollten Sie auch die eckigen Klammern verwenden. Beispiel:

```
[[:alpha:]]
```

Dies entspricht einem Buchstaben.

```
[[:digit:]-]{2}
```

Dies entspricht 2 Zeichen, entweder Ziffern oder `-` . Folgendes wird passen:

- `--`
- `11`
- `-2`
- `3-`

Weitere Informationen finden Sie unter: Regular-expressions.info

Charakterklassen online lesen: https://riptutorial.com/de/regex/topic/1757/charakterklassen

Kapitel 7: Einfache Muster abgleichen

Examples

Ein einzelnes Zeichen mit [0-9] oder \ d (Java) abgleichen

`[0-9]` und `\d` sind äquivalente Muster (es sei denn, Ihre Regex-Engine ist Unicode-fähig und `\d` stimmt auch mit überein). Sie stimmen beide mit einem einstelligen Zeichen überein, sodass Sie die jeweils lesbarere Schreibweise verwenden können.

Erstellen Sie eine Zeichenfolge des Musters, das Sie zuordnen möchten. Wenn Sie die \ d-Notation verwenden, müssen Sie einen zweiten Backslash hinzufügen, um den ersten Backslash zu umgehen.

```
String pattern = "\\d";
```

Erstellen Sie ein Musterobjekt. Übergeben Sie die Musterzeichenfolge an die Methode compile ().

```
Pattern p = Pattern.compile(pattern);
```

Erstellen Sie ein Matcher-Objekt. Übergeben Sie die Zeichenfolge, die Sie suchen, um das Muster in der matcher () -Methode zu finden. Prüfen Sie, ob das Muster gefunden wurde.

```
Matcher m1 = p.matcher("0");
m1.matches(); //will return true

Matcher m2 = p.matcher("5");
m2.matches(); //will return true

Matcher m3 = p.matcher("12345");
m3.matches(); //will return false since your pattern is only for a single integer
```

Verschiedene Zahlen abgleichen

`[ab]` wobei a und b Ziffern im Bereich von `0` bis `9`

```
[3-7] will match a single digit in the range 3 to 7.
```

Mehrere Ziffern übereinstimmen

```
\d\d       will match 2 consecutive digits
\d+        will match 1 or more consecutive digits
\d*        will match 0 or more consecutive digits
\d{3}      will match 3 consecutive digits
\d{3,6}    will match 3 to 6 consecutive digits
\d{3,}     will match 3 or more consecutive digits
```

Das `\d` in den obigen Beispielen kann durch einen Zahlenbereich ersetzt werden:

```
[3-7][3-7]    will match 2 consecutive digits that are in the range 3 to 7
[3-7]+        will match 1 or more consecutive digits that are in the range 3 to 7
[3-7]*        will match 0 or more consecutive digits that are in the range 3 to 7
[3-7]{3}      will match 3 consecutive digits that are in the range 3 to 7
[3-7]{3,6}    will match 3 to 6 consecutive digits that are in the range 3 to 7
[3-7]{3,}     will match 3 or more consecutive digits that are in the range 3 to 7
```

Sie können auch bestimmte Ziffern auswählen:

```
[13579]       will only match "odd" digits
[02468]       will only match "even" digits
1|3|5|7|9     another way of matching "odd" digits - the | symbol means OR
```

Übereinstimmende Zahlen in Bereichen, die mehr als eine Ziffer enthalten:

```
\d|10        matches 0 to 10    single digit OR 10.  The | symbol means OR
[1-9]|10     matches 1 to 10    digit in range 1 to 9 OR 10
[1-9]|1[0-5] matches 1 to 15    digit in range 1 to 9 OR 1 followed by digit 1 to 5
\d{1,2}|100  matches 0 to 100   one to two digits OR 100
```

Übereinstimmende Zahlen, die durch andere Zahlen geteilt werden:

```
\d*0         matches any number that divides by 10  - any number ending in 0
\d*00        matches any number that divides by 100 - any number ending in 00
\d*[05]      matches any number that divides by 5   - any number ending in 0 or 5
\d*[02468]   matches any number that divides by 2   - any number ending in 0,2,4,6 or 8
```

übereinstimmende Zahlen, die sich durch 4 teilen - jede Zahl, die 0, 4 oder 8 ist oder auf 00, 04, 08, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60 endet 64, 68, 72, 76, 80, 84, 88, 92 oder 96

```
[048]|\d*(00|04|08|12|16|20|24|28|32|36|40|44|48|52|56|60|64|68|72|76|80|84|88|92|96)
```

Dies kann verkürzt werden. Anstelle von `20|24|28` wir beispielsweise `2[048]` . Da die 40er, 60er und 80er Jahre das gleiche Muster haben, können wir sie einschließen: `[02468][048]` und die anderen haben auch ein Muster `[13579][26]` . So kann die ganze Sequenz reduziert werden auf:

```
[048]|\d*([02468][048]|[13579][26])    - numbers divisible by 4
```

Übereinstimmende Zahlen, deren Muster nicht durch 2,4,5,10 teilbar ist, können nicht immer präzise ausgeführt werden. In der Regel müssen Sie auf eine Reihe von Zahlen zurückgreifen. Zum Beispiel kann das Vergleichen aller Zahlen, die durch 7 im Bereich von 1 bis 50 geteilt werden, einfach durch Auflisten aller Zahlen erfolgen:

```
7|14|21|28|35|42|49

or you could do it this way

7|14|2[18]|35|4[29]
```

Nachgestellte Räume

`\s*$` : Dies entspricht einem beliebigen (`*`) Whitespace (`\s`) am Ende (`$`) des Texts

Führende Räume

`^\s*` : Dies entspricht einem beliebigen (`*`) Leerzeichen (`\s`) am Anfang (`^`) des Textes

Bemerkungen

`\s` ist ein allgemeines Metazeichen für mehrere RegExp-Engines und dient zum Erfassen von Leerzeichen (Leerzeichen, Zeilenumbrüche und Registerkarten). **Hinweis** : Wahrscheinlich *werden nicht* alle Unicode-Leerzeichen erfasst. Überprüfen Sie die Dokumentation Ihrer Motoren, um sich darüber sicher zu sein.

Passen Sie jeden Schwimmer an

```
[\+\-]?\d+(\.\d*)?
```

Dies wird mit jedem vorzeichenbehafteten Float übereinstimmen, wenn Sie keine Zeichen haben möchten oder eine Gleichung analysieren `[\+\-]?` entfernen `[\+\-]?` also hast du `\d+(\.\d+)?`

Erläuterung:

- `\d+` entspricht einer beliebigen Ganzzahl
- `()?` bedeutet, dass der Inhalt der Klammern optional ist, aber immer zusammen angezeigt werden muss
- '\.' Stimmt mit '.' überein, müssen wir dem seit '.' entspricht normalerweise einem beliebigen Zeichen

Also wird dieser Ausdruck passen

```
5
+5
-5
5.5
+5.5
-5.5
```

Auswählen einer bestimmten Zeile aus einer Liste basierend auf einem Wort an einem bestimmten Ort

Ich habe folgende Liste:

```
1. Alon Cohen
2. Elad Yaron
3. Yaron Amrani
4. Yogev Yaron
```

Ich möchte den Vornamen der Jungs mit dem Nachnamen Yaron auswählen.

Da es mir egal ist, um welche Nummer es sich handelt, lege ich es einfach als Zeichen und einen passenden Punkt und Leerzeichen vom Anfang der Zeile an wie folgt: `^[\d]+\.\s` .

Jetzt müssen wir das Leerzeichen und den Vornamen `[a-zA-Z]+\s` , da wir nicht erkennen können, ob es sich um `[a-zA-Z]+\s` oder Kleinbuchstaben handelt, wir passen nur zu beidem: `[a-zA-Z]+\s` oder `[aZ]+\s` und kann auch `[\w]+\s` .

Jetzt geben wir den erforderlichen Nachnamen an, um nur die Zeilen zu erhalten, die Yaron als Nachnamen enthalten (am Ende der Zeile): `\sYaron$` .

Alles zusammen setzen `^[\d]+\.\s[\w]+\sYaron$` .

Live-Beispiel: https://regex101.com/r/nW4fH8/1

Einfache Muster abgleichen online lesen: https://riptutorial.com/de/regex/topic/343/einfache-muster-abgleichen

Kapitel 8: Ersetzungen mit regulären Ausdrücken

Parameter

In der Reihe	Beschreibung
$ Nummer	Ersetzt die durch die Gruppennummer übereinstimmende Unterzeichenfolge.
$ {name}	Ersetzt den durch einen benannten Gruppennamen übereinstimmenden Teilstring.
$$	Escape-Zeichen '$' in der Ergebniszeichenfolge (Ersetzung).
$ & *($ 0)*	Ersetzt mit der gesamten übereinstimmenden Zeichenfolge.
$ + *($ &)*	Ersetzt den übereinstimmenden Text durch die zuletzt erfasste Gruppe.
$ `	Ersetzt den gesamten übereinstimmenden Text vor dem Match mit jedem nicht übereinstimmenden Text.
$ '	Ersetzt den gesamten abgeglichenen Text nach dem Abgleich mit jedem nicht abgeglichenen Text.
$ _	Ersetzt den gesamten übereinstimmenden Text durch die gesamte Zeichenfolge.
Hinweis:	*Kursive* Ausdrücke bedeuten, dass die Zeichenfolgen flüchtig sind (kann je nach Regex-Geschmack variieren).

Examples

Grundlagen der Substitution

Eine der häufigsten und nützlichsten Methoden zum Ersetzen von Text durch reguläre Ausdrücke ist die Verwendung von Capture-Gruppen .
Oder sogar eine benannte Capture-Gruppe als Referenz zum Speichern oder Ersetzen der Daten.

Es gibt zwei Ausdrücke, die in regexs Dokumenten ziemlich gleich aussehen. Daher kann es wichtig sein, **Substitutionen** (dh `$1`) **niemals** mit **Backreferences** (dh `\1`) zu **verwechseln** . Ersetzungsbedingungen werden in einem Ersetzungstext verwendet. Rückreferenzen, im reinen Regex-Ausdruck. Obwohl einige Programmiersprachen beide für Substitutionen akzeptieren, ist dies nicht ermutigend.

Nehmen wir an, wir haben diese Regex: `/hello(\s+)world/i` Wenn auf `$number` verwiesen wird (in diesem Fall `$1`), werden stattdessen die mit `\s+` übereinstimmenden Whitespaces ersetzt.
Das gleiche Ergebnis wird mit der Regex: `/hello(?<spaces>\s+)world/i` . Und da wir hier eine benannte Gruppe haben, können wir auch `${spaces}` .

In diesem Beispiel können wir auch `$0` oder `$&` (**Hinweis:** Stattdessen kann `$&` als `$+` verwendet werden, um die **LAST-** Capture-Gruppe in anderen Regex-Engines abzurufen.) um den gesamten passenden Text zu erhalten. (dh `$&` wird `hEllo woRld` für die Zeichenfolge zurückgeben: `hEllo woRld of Regex!`)

Sehen Sie sich dieses einfache Beispiel für die Ersetzung mit dem angepassten Zitat von John Lennon mithilfe der `$number` und der `${name}` -Syntax an:

Einfaches Beispiel für eine Capture-Gruppe:

```
/(Happy)\./g
```

Test String

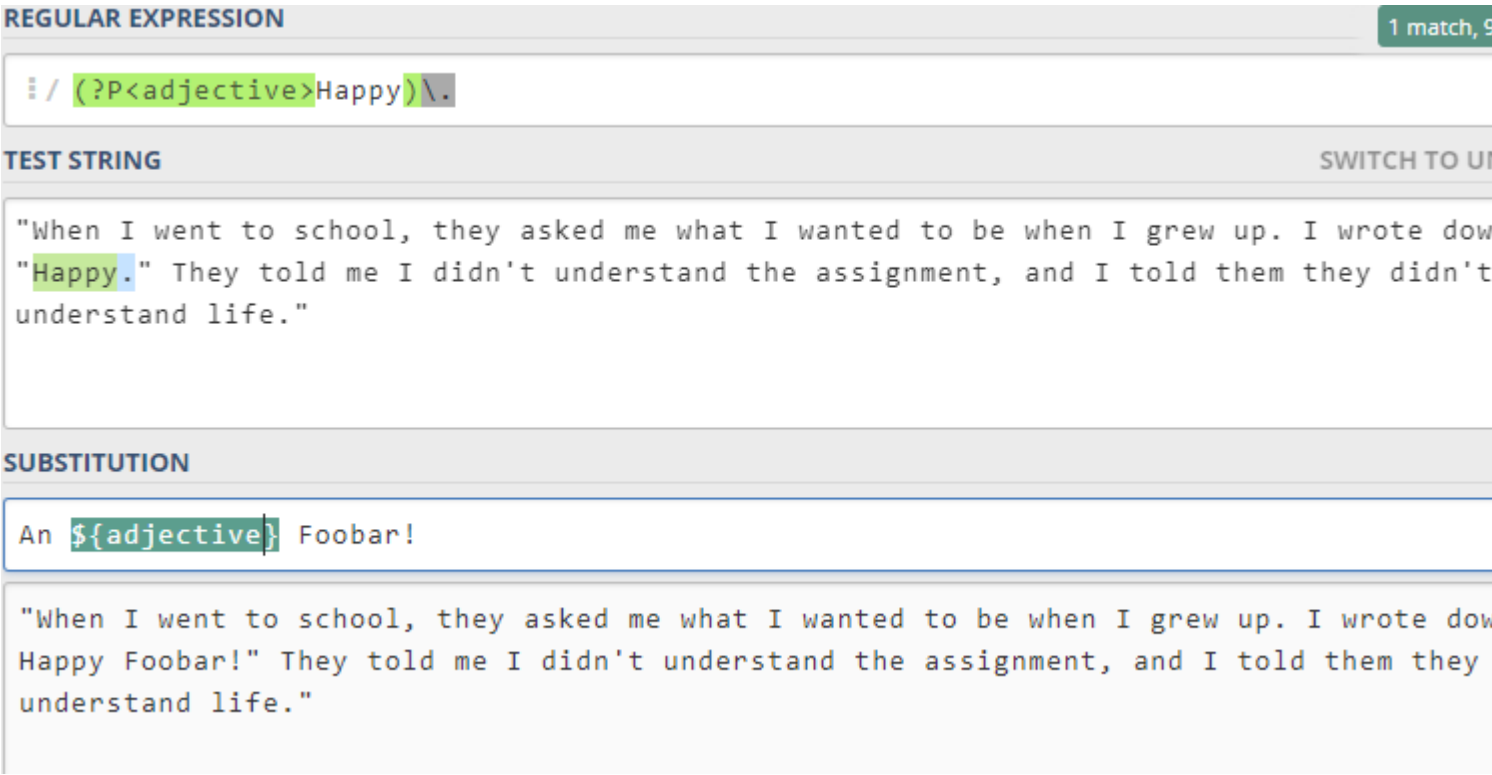
```
"When I went to school, they asked me what I wanted to be when I grew up. I w
me I didn't understand the assignment, and I told them they didn't understand
```

Substitution

```
An $1 Foobar!
```

```
"When I went to school, they asked me what I wanted to be when I grew up. I w
They told me I didn't understand the assignment, and I told them they didn't
```

Benanntes Erfassungsgruppenbeispiel:



Erweiterter Ersatz

Einige Programmiersprachen haben eigene Regex-Besonderheiten, beispielsweise den `$+` Ausdruck (in C #, Perl, VB usw.), der den übereinstimmenden Text der zuletzt erfassten Gruppe ersetzt.

Beispiel:

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//       The dog jumped over the fence.
```

Beispiel aus dem Developer Network von Microsoft Official [1]

Andere seltene Substitutionsbedingungen sind `$`` und `$'` :

`$`` = Ersetzt Übereinstimmungen mit dem Text **vor** der übereinstimmenden Zeichenfolge
`$'` = Ersetzt Übereinstimmungen mit dem Text **nach** der übereinstimmenden Zeichenfolge

Aus diesem Grund sollten diese Ersetzungszeichenfolgen ihre Arbeit so ausführen:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$`"
Output: "part1part1part3" //Note that part2 was replaced with part1, due &` term
--------------------------------------------------------------------------------
Regex: /part2/
Input: "part1part2part3"
Replacement: "$'"
Output: "part1part3part3" //Note that part2 was replaced with part3, due &' term
```

Hier ist ein Beispiel für diese Ersetzungen, die an einem Javascript arbeiten:

```
var rgx = /middle/;
var text = "Your story must have a beginning, middle, and end"
console.log(text.replace(rgx, "$`"));
//Logs: "Your story must have a beginning, Your story must have a beginning, , and end"
console.log(text.replace(rgx, "$'"))
//Logs: "Your story must have a beginning, , and end, and end"
```

Es gibt auch den Begriff `$_` der stattdessen den gesamten übereinstimmenden Text abruft:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$_"
Output: "part1part1part2part3part3" //Note that part2 was replaced with part1part2part3,
                                                                    // due $_ term
```

Das Konvertieren in VB würde uns folgendes geben:

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "ABC123DEF456"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$_"
      Console.WriteLine("Original string:          {0}", input)
      Console.WriteLine("String with substitution: {0}", _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'       Original string:          ABC123DEF456
'       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

Beispiel aus dem Developer Network von Microsoft Official [2]

Und der letzte und nicht zuletzt ersetzende Begriff ist `$$` , der in einen Regex-Ausdruck übersetzt würde, wäre derselbe wie `\$` (Eine Escape-Version des Buchstabens **$**).

Wenn Sie eine Zeichenfolge wie `USD: $3.99` möchten: `USD: $3.99` zum Beispiel und die `3.99` speichern, aber als `$3.99` durch nur einen Regex ersetzen, können Sie Folgendes verwenden:

```
Regex: /USD:\s+\$([\d.]+)/
Input: "USD: $3.99"
Replacement: "$$$1"
To Store: "$1"
Output: "$3.99"
Stored: "3.99"
```

Wenn Sie dies mit Javascript testen möchten, können Sie den Code verwenden:

```
var rgx = /USD:\s+\$([\d.]+)/;
var text = "USD: $3.99";
var stored = parseFloat(rgx.exec(text)[1]);
console.log(stored); //Logs 3.99
console.log(text.replace(rgx, "$$$1")); //Logs $3.99
```

Verweise

[1]: Ersetzen der zuletzt erfassten Gruppe
[2]: Ersetzen der gesamten Eingabezeichenfolge

Ersetzungen mit regulären Ausdrücken online lesen:
https://riptutorial.com/de/regex/topic/9852/ersetzungen-mit-regularen-ausdrucken

Kapitel 9: Flucht

Examples

Raw String Literals

Es ist am besten für die Lesbarkeit (und Ihre Vernunft), um zu vermeiden, dass die Flucht entkommt. Hier kommen rohe Zeichenkettenliterale ins Spiel. (Beachten Sie, dass einige Sprachen Trennzeichen zulassen, die normalerweise gegenüber Zeichenketten bevorzugt werden. Dies ist jedoch ein anderer Abschnitt.)

Sie funktionieren normalerweise auf dieselbe Weise wie diese Antwort beschreibt :

> [A] backslash, `\` bedeutet "nur ein Backslash" (außer wenn es direkt vor einem Zitat steht, das sonst das Literal beenden würde) - keine "Escape-Sequenzen" zur Darstellung von Zeilenumbrüchen, Registerkarten, Backspaces, Formular-Feeds , und so weiter.

Nicht alle Sprachen haben sie und die, die unterschiedliche Syntax verwenden. C # nennt sie eigentlich *wörtliche String-Literale* , aber es ist dasselbe.

Python

```
pattern = r"regex"
```

```
pattern = r'regex'
```

C ++ (11+)

Die Syntax hier ist äußerst vielseitig. Die einzige Regel ist die Verwendung eines Trennzeichens, das an keiner Stelle im regulären Ausdruck erscheint. Wenn Sie dies tun, ist für etwas in der Zeichenfolge kein zusätzliches Escape erforderlich. Beachten Sie, dass die Klammern `()` nicht Teil der Regex sind:

```
pattern = R"delimiter(regex)delimiter";
```

VB.NET

Verwenden Sie einfach eine normale Zeichenfolge. Backslashes sind IMMER Literale .

C

```
pattern = @"regex";
```

Beachten Sie, dass diese Syntax auch `""` (zwei Anführungszeichen) als Escape-Form von `" ""` erlaubt .

Zeichenketten

In den meisten Programmiersprachen muss jeder Backslash im String-Literal verdoppelt werden, damit in einem String ein Backslash aus einem String-Literal generiert wird. Andernfalls wird es als Flucht für das nächste Zeichen interpretiert.

Leider muss jeder von der Regex benötigte Backslash ein wörtlicher Backslash sein. Aus diesem Grund wird es notwendig, "Escape-Fluchten" (`\\`) zu haben, wenn Regex aus String-Literalen generiert wird.

Außerdem müssen Anführungszeichen (`"` oder `'`) im Zeichenfolgenliteral möglicherweise mit Escapezeichen versehen werden, je nachdem, um welches Zeichenkettenliteral es sich handelt. In einigen Sprachen ist es möglich, einen Anführungszeichenstil für eine Zeichenfolge zu verwenden (wählen Sie den am besten lesbaren aus das gesamte String-Literal zu umgehen).

In einigen Sprachen (zB: Java <= 7) können reguläre Ausdrücke nicht direkt als Literale ausgedrückt werden, z. B. `/\w/` ; Sie müssen aus Strings generiert werden, und normalerweise werden String-Literale verwendet - in diesem Fall `"\\w"` . In diesen Fällen müssen Literalzeichen wie Anführungszeichen, umgekehrte Schrägstriche usw. mit Escapezeichen versehen werden. Dies lässt sich am einfachsten mit einem Tool (wie *RegexPlanet*) erreichen. Dieses spezielle Tool ist für Java konzipiert, funktioniert jedoch für jede Sprache mit einer ähnlichen Zeichenfolgensyntax.

Welche Zeichen müssen entkommen werden?

Durch Zeichen-Escape-Zeichen können bestimmte Zeichen (die von der Regex-Engine für die Bearbeitung von Suchen reserviert werden) buchstäblich gesucht und in der Eingabezeichenfolge gefunden werden. Die Escape-Funktion hängt vom Kontext ab. In diesem Beispiel wird daher keine Zeichenfolge- oder Trennzeichen- Escapezeichen behandelt.

Backslashes

Zu sagen, dass Backslash das "Escape" -Zeichen ist, ist etwas irreführend. Backslash entkommt und Backslash bringt; es schaltet das Metazeichen tatsächlich ein oder aus, verglichen mit dem wörtlichen Status der Figur, die sich davor befindet.

Um einen literalen Backslash überall in einer Regex verwenden zu können, muss dieser durch einen anderen Backslash ersetzt werden.

Flucht (außerhalb von Zeichenklassen)

Es gibt mehrere Zeichen, für die eine Escape-Funktion erforderlich ist (zumindest außerhalb von Zeichenklassen):

- Halterungen: `[]`
- Klammern: `()`
- Geschweifte Klammern: `{}`
- Betreiber: `*` , `+` `?` , `|`
- Anker: `^` , `$`
- Andere: `.` , `\`
- Um ein Literal `^` am Anfang oder ein Literal `$` am Ende einer Regex zu verwenden, muss das Zeichen mit Escapezeichen versehen werden.
- Einige Varianten verwenden `^` und `$` als Metazeichen, wenn sie sich am Anfang bzw. Ende der Regex befinden. In diesen Geschmacksrichtungen ist keine zusätzliche Flucht notwendig. Es ist normalerweise nur das Beste, ihnen trotzdem zu entkommen.

Flucht innerhalb von Zeichenklassen

- Es empfiehlt sich, eckige Klammern (`[` und `]`) zu umgehen, wenn sie als Literale in einer Zeichenklasse erscheinen. Unter bestimmten Umständen ist dies je nach Geschmack nicht erforderlich, beeinträchtigt jedoch die Lesbarkeit.
- Das Caret `^` ist ein Meta-Zeichen, wenn es als erstes Zeichen einer Zeichenklasse gesetzt wird: `[^aeiou]` . Überall in der Zeichenklasse ist es nur ein buchstäblicher Charakter.
- Der Bindestrich `-` ist ein Meta-Zeichen, es sei denn, er steht am Anfang oder Ende einer Zeichenklasse. Wenn das erste Zeichen in der Zeichenklasse ein Caret `^` , ist es ein Literal, wenn es das zweite Zeichen in der Zeichenklasse ist.

Dem Ersatz entgehen

Es gibt auch Regeln für die Flucht innerhalb der Ersetzung, aber keine der oben genannten Regeln trifft zu. Die einzigen Metazeichen sind `$` und `\` , zumindest wenn `$` verwendet werden kann, um Capture-Gruppen zu referenzieren (zB `$1` für Gruppe 1). Um ein wörtliches `$` , entgehen Sie es: `\$5.00` . Ebenso `\` : `C:\\Program Files\\` .

BRE-Ausnahmen

Während ERE (erweiterte reguläre Ausdrücke) die typische Perl-Stil-Syntax widerspiegelt, weist BRE (Basic Regular Expressions) grundlegende Unterschiede beim Fluchtweg auf:

- Es gibt unterschiedliche Abkürzungssyntax. Alle `\d` , `\s` , `\w` und so weiter sind verschwunden. Stattdessen hat es eine eigene Syntax (die POSIX verwirrend "Zeichenklassen" nennt), wie `[:digit:]` . Diese Konstrukte müssen sich innerhalb einer Zeichenklasse befinden.

- Es gibt wenige Metazeichen (`.` , `*` , `^` , `$`), Die normal verwendet werden können. ALLE anderen Metazeichen müssen unterschiedlich maskiert werden:

Zahnspangen `{}`

- `a{1,2}` entspricht `a{1,2}` . Verwenden Sie zum Abgleichen von `a` oder `aa` `a\{1,2\}`

Klammern `()`

- `(ab)\1` ist ungültig, da keine Capture-Gruppe 1 vorhanden ist. Verwenden Sie `\(ab\)\1` um das `abab` zu beheben

Backslash

- Innerhalb von Zeichenklassen (die in POSIX Klammerausdrücke genannt werden) ist der Backslash kein Metazeichen (und muss nicht umgangen werden). `[\d]` stimmt entweder mit `\` oder `d` überein.
- Überall sonst wie üblich entkommen.

Andere

- `+` und `?` sind Literale. Wenn die BRE-Engine sie als Metazeichen unterstützt, müssen sie als `\?` und `\+` .

/ Trennzeichen /

In vielen Sprachen können reguläre Ausdrücke zwischen bestimmten Zeichen eingeschlossen werden, normalerweise dem Schrägstrich `/` .

Begrenzungszeichen wirken sich auf das Escapeing aus: Wenn das Begrenzungszeichen `/` und der Regex nach `/` Liter suchen muss, muss der Schrägstrich vor dem Literal (`\/`) geschützt werden.

Übermäßiger Fluchtweg schadet der Lesbarkeit. Daher müssen die verfügbaren Optionen berücksichtigt werden:

Javascript ist einzigartig, da es Schrägstrich als Trennzeichen zulässt, aber sonst nichts (obwohl es stringifizierte Regexes zulässt).

Perl 1

Perl erlaubt zum Beispiel, dass fast alles ein Trennzeichen ist. Sogar arabische Zeichen:

```
$str =~ m ش ش
```

Spezifische Regeln werden in der Perl-Dokumentation erwähnt .

PCRE erlaubt zwei Arten von Trennzeichen: Übereinstimmende Trennzeichen und Trennzeichen im Klammerstil. Übereinstimmende Trennzeichen verwenden das Paar eines einzelnen Charakters, während Trennzeichen im Klammer-Stil ein paar Zeichen verwenden, die ein

öffnendes und ein schließendes Paar darstellen.

- Passende Trennzeichen `!"#$%&'*+,./:;=?@^_`|~-`
- Trennzeichen im Klammerstil: `()` , `{}` , `[]` , `<>`

Flucht online lesen: https://riptutorial.com/de/regex/topic/4524/flucht

Kapitel 10: Gierige und faule Quantifizierer

Parameter

Quantifizierer	Beschreibung
`?`	Passen Sie das vorangehende Zeichen oder den Unterausdruck 0 oder 1 Mal an (vorzugsweise 1).
`*`	Stimmt mit dem vorhergehenden Zeichen oder Unterausdruck mindestens 0 mal überein (so viele wie möglich).
`+`	Ordnen Sie den vorhergehenden Buchstaben oder Unterausdruck mindestens ein Mal (so viele wie möglich) zu.
`{n}`	Passen Sie das vorhergehende Zeichen oder den Unterausdruck genau *n*- mal an.
`{min,}`	Entspricht die vorangehenden Zeichen oder subexpression *min* oder mehrmals (so viele wie möglich).
`{0,max}`	Passen Sie das vorhergehende Zeichen oder den untergeordneten Ausdruck *maximal* oder weniger oft an (so nahe wie möglich an das *Maximum*).
`{min,max}`	Passen Sie den vorhergehenden Buchstaben oder Unterausdruck mindestens *min* , jedoch nicht mehr als *max. An* (so nahe wie möglich an *max*).
Faule Quantifizierer	**Beschreibung**
`??`	Passen Sie das vorangehende Zeichen oder den Unterausdruck 0 oder 1 Mal an (vorzugsweise 0).
`*?`	Stimmt mit dem vorhergehenden Zeichen oder Unterausdruck mindestens 0 mal überein (so wenig wie möglich).
`+?`	Ordnen Sie den vorhergehenden Buchstaben oder Unterausdruck mindestens ein Mal (so wenig wie möglich) zu.
`{n}?`	Passen Sie das vorhergehende Zeichen oder den Unterausdruck genau *n*- mal an. Kein Unterschied zwischen gieriger und fauler Version.
`{min,}?`	Entspricht die vorhergehenden Zeichen oder subexpression *min* oder mehrmals (so nahe wie möglich *min).*

Quantifizierer	Beschreibung
`{0,max}?`	Passen Sie das vorhergehende Zeichen oder den untergeordneten Ausdruck *maximal* oder seltener an (so wenig wie möglich).
`{min,max}?`	Passen Sie den vorhergehenden Buchstaben oder Unterausdruck mindestens *min* , jedoch nicht mehr als *max* (möglichst nahe an *min*) an.

Bemerkungen

Gier

Ein *gieriger* Quantifizierer versucht immer, das Untermuster so oft wie möglich zu wiederholen, bevor er durch Backtracking kürzere Übereinstimmungen untersucht.

Im Allgemeinen entspricht ein gieriges Muster der längsten möglichen Zeichenfolge.

Standardmäßig sind alle Quantifizierer gierig.

Faulheit

Ein *faule* (auch *nicht-gierig* oder *nur ungern* genannt) quantifier immer versucht , das Untermuster so *wenig* wie möglich zu wiederholen, vor längeres Matches durch Expansion zu erkunden.

Im Allgemeinen entspricht ein Lazy-Pattern der kürzesten möglichen Zeichenfolge.

Um Quantifikatoren faul zu machen, einfach anhängen `?` zu dem vorhandenen Quantifizierer, z. B. `+? {0,5}?` .

Das Konzept von Gier und Faulheit existiert nur in rückverfolgbaren Motoren

Die Vorstellung von gierigen / faulen Quantifizierern existiert nur in Regex-Engines. In Regex-Engines ohne Backtracking oder POSIX-kompatiblen Regex-Engines geben die Quantifizierer nur die Ober- und Untergrenze der Wiederholung an, ohne anzugeben, wie die Übereinstimmung gefunden werden soll.

Examples

Gierigkeit gegen Faulheit

Gegeben die folgende Eingabe:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
```

Wir werden zwei Muster verwenden: eines gierig: `A.*Z` und eines faul: `A.*?Z` Diese Muster ergeben folgende Übereinstimmungen:

- `A.*Z` ergibt 1 Treffer: `AlazyZgreeedyAlaaazyZ` (Beispiele: Regex101 , Rubular)
- `A.*?Z` ergibt 2 Treffer: `AlazyZ` und `AlaaazyZ` (Beispiele: Regex101 , Rubular)

Konzentriere dich zuerst auf das, was `A.*Z` macht. Wenn es mit dem ersten `A` übereinstimmt, versucht das `.*` , Gierig zu sein, dann so viele zu finden `.` wie möglich.

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_________________________/
      A.* matched, Z can't match
```

Da das `Z` nicht übereinstimmt, zieht sich die Engine zurück, und `.*` Muss dann um eins geringer sein `.` :

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_________________________/
      A.* matched, Z can't match
```

Das passiert noch ein paar Mal, bis es endlich so weit ist:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________/
      A.* matched, Z can now match
```

Jetzt kann `Z` passen, also stimmt das Gesamtmuster überein:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________/
      A.*Z matched
```

Im Gegensatz dazu trifft die widerwillige (faule) Wiederholung in `A.*?Z` zunächst auf wenige `.` wie möglich und dann mehr nehmen `.` wie nötig. Dies erklärt, warum es zwei Übereinstimmungen in der Eingabe findet.

Hier ist eine visuelle Darstellung der Übereinstimmung der beiden Muster:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____/l      \______/l      l = lazy
     \_________g_________/       g = greedy
```

Beispiel basierend auf einer Antwort von Polygenschmiermitteln .

Der POSIX-Standard enthält nicht das `?` Betreiber, so dass viele POSIX-Regex-Engines nicht faul passen. Während das Refactoring, vor allem mit dem "größten Trick aller Zeiten" , in manchen Fällen zum Vergleich beitragen kann, besteht die einzige Möglichkeit, ein echtes Lazy-Matching zu haben, in der Verwendung einer Engine, die dies unterstützt.

Grenzen mit mehreren Übereinstimmungen

Wenn Sie eine Eingabe mit genau definierten Grenzen haben und mehr als eine Übereinstimmung in Ihrer Zeichenfolge erwarten, haben Sie zwei Möglichkeiten:

- Verwendung fauler Quantifizierer;
- Verwendung einer negierten Zeichenklasse

Folgendes berücksichtigen:

Sie haben eine einfache Template-Engine, Sie möchten Teilzeichenfolgen wie `$[foo]` ersetzen, wobei `foo` eine beliebige Zeichenfolge sein kann. Sie möchten diese Teilzeichenfolge durch das ersetzen, das auf dem Teil zwischen den `[]` basiert.

Sie können etwas wie `\$\[(.*)\]` Versuchen und dann die erste Capture-Gruppe verwenden.

Das Problem dabei ist, wenn Sie eine Zeichenfolge wie `something $[foo] lalala $[bar] something else` ist

```
something $[foo] lalala $[bar] something else
          | \_______CG1_______/|
          \________Match_______/
```

Die Erfassungsgruppe ist `foo] lalala $[bar` die möglicherweise nicht gültig ist.

Sie haben zwei Lösungen

1. Faulheit verwenden: In diesem Fall ist das `*` Faulenzen eine Möglichkeit, die richtigen Dinge zu finden. Sie ändern also Ihren Ausdruck in `\$\[(.*?)\]`
2. Bei Verwendung einer negierten Zeichenklasse: `[^\]]` ändern Sie Ihren Ausdruck in `\$\[([^\]]*)\]` .

In beiden Lösungen ist das Ergebnis das gleiche:

```
something $[foo] lalala $[bar] something else
          | \_/|         | \_/|
          \____/         \____/
```

Die Capture-Gruppe ist jeweils `foo` und `bar` .

Die Verwendung einer negierten Zeichenklasse reduziert das Backtracking-Problem und kann Ihre CPU bei großen Eingaben viel Zeit sparen.

Gierige und faule Quantifizierer online lesen: https://riptutorial.com/de/regex/topic/429/gierige-und-faule-quantifizierer

Kapitel 11: Gruppen erfassen

Examples

Grundlegende Capture-Gruppen

Eine *Gruppe* ist ein Abschnitt eines regulären Ausdrucks in Klammern `()` . Dies wird üblicherweise als "Unterausdruck" bezeichnet und dient zwei Zwecken:

- Sie macht den Unterausdruck atomar, dh er passt sich als Ganzes an, schlägt fehl oder wiederholt sich.
- Der übereinstimmende Teil des Textes ist im Rest des Ausdrucks und im Rest des Programms verfügbar.

Gruppen werden in Regex-Engines nummeriert, beginnend mit 1. Die maximale Gruppennummer beträgt 9, aber viele moderne Regex-Varianten unterstützen höhere Gruppenzahlen. Gruppe 0 stimmt immer mit dem gesamten Muster überein, genauso wie die gesamte reguläre Ausdehnung mit Klammern.

Die Ordnungszahl erhöht sich mit jeder öffnenden Klammer, unabhängig davon, ob die Gruppen hintereinander angeordnet oder verschachtelt sind:

```
foo(bar(baz)?) (qux)+|(bla)
   1   2       3      4
```

Gruppen und ihre Anzahl

Nachdem ein Ausdruck eine Gesamtübereinstimmung erzielt hat, werden alle seine Gruppen verwendet - unabhängig davon, ob eine bestimmte Gruppe irgendetwas gefunden hat oder nicht.

Eine Gruppe kann optional sein, wie `(baz)?` oben oder in einem alternativen Teil des Ausdrucks, der nicht von der Übereinstimmung verwendet wurde, wie `(bla)` oben. In diesen Fällen enthalten nicht übereinstimmende Gruppen einfach keine Informationen.

Wenn ein Quantifizierer wie in `(qux)+` oben hinter einer Gruppe platziert wird, `(qux)+` die Gesamtzahl der Gruppen des Ausdrucks gleich. Wenn eine Gruppe mehr als einmal übereinstimmt, ist ihr Inhalt der letzte Treffer. Moderne Regex-Varianten erlauben jedoch den Zugriff auf alle Vorkommnisse.

Wenn Sie das Datum und die Fehlerstufe eines Protokolleintrags wie diesem abrufen möchten:

```
2012-06-06 12:12.014 ERROR: Failed to connect to remote end
```

Sie könnten so etwas verwenden:

```
^(\d{4}-\d{2}-\d{2}) \d{2}:\d{2}.\d{3} (\w*): .*$
```

Dies würde das Datum des Protokolleintrags `2012-06-06` als Erfassungsgruppe 1 und die Fehlerstufe `ERROR` als Erfassungsgruppe 2 extrahieren.

Rückreferenzen und nicht erfassende Gruppen

Da Groups "nummeriert" sind, unterstützen einige Engines auch das Übereinstimmen mit dem, was eine Gruppe zuvor wieder gefunden hat.

Angenommen, Sie wollten etwas finden, bei dem zwei gleiche Zeichenfolgen der Länge drei durch ein `$` geteilt werden, das Sie verwenden würden:

```
(.{3})\$\1
```

Dies würde einer der folgenden Zeichenfolgen entsprechen:

```
"abc$abc"
"a b$a b"
"af $af "
"   $   "
```

Wenn Sie möchten, dass eine Gruppe nicht von der Engine nummeriert wird, können Sie sie für nicht erfassend festlegen. Eine nicht erfassende Gruppe sieht folgendermaßen aus:

```
(?:)
```

Sie sind besonders nützlich, um ein bestimmtes Muster beliebig oft zu wiederholen, da eine Gruppe auch als "Atom" verwendet werden kann. Erwägen:

```
(\d{4}(?:-\d{2}){2} \d{2}:\d{2}.\d{3}) (.*)[\r\n]+\1 \2
```

Dadurch werden zwei Protokolleinträge in den benachbarten Zeilen übereinstimmen, die denselben Zeitstempel und denselben Eintrag haben.

Benannte Erfassungsgruppen

Einige reguläre Ausdrucksvarianten ermöglichen *benannte Erfassungsgruppen* . Anstelle eines numerischen Index können Sie diese Gruppen im folgenden Code nach Namen referenzieren, dh in Rückverweisen, im Ersetzungsmuster sowie in den folgenden Zeilen des Programms.

Numerische Indizes ändern sich, wenn sich die Anzahl oder Anordnung von Gruppen in einem Ausdruck ändert, sodass sie im Vergleich spröder sind.

Um beispielsweise ein Wort (`\w+`) in einfachen oder doppelten Anführungszeichen (`['"]`) zu finden, können wir Folgendes verwenden:

```
(?<quote>['"])\w+\k{quote}
```

Welches ist äquivalent zu:

```
(['"])\w+\1
```

In einer einfachen Situation wie dieser hat eine normale, nummerierte Erfassungsgruppe keine Nachteile.

In komplexeren Situationen wird die Verwendung von benannten Gruppen die Struktur des Ausdrucks für den Leser deutlicher machen, was die Wartbarkeit verbessert.

Die Analyse von Protokolldateien ist ein Beispiel für eine komplexere Situation, die von Gruppennamen profitiert. Dies ist das Apache Common Log Format (CLF):

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

Der folgende Ausdruck fängt die Teile in benannten Gruppen ein:

```
(?<ip>\S+) (?<logname>\S+) (?<user>\S+) (?<time>\[[^]]+\]) (?<request>"[^"]+") (?<status>\S+)
(?<bytes>\S+)
```

Die Syntax hängt vom Flavour ab. Folgende sind üblich:

- `(?<name>...)`
- `(?'name'...)`
- `(?P<name>...)`

Rückreferenzen:

- `\k<name>`
- `\k{name}`
- `\k'name'`
- `\g{name}`
- `(?P=name)`

In der .NET-Variante können mehrere Gruppen denselben Namen haben. Sie verwenden Capture-Stapel .

In PCRE müssen Sie es explizit aktivieren, indem Sie den Modifizierer `(?J)` (`PCRE_DUPNAMES`) oder die `PCRE_DUPNAMES (?|)` Verwenden. Es ist jedoch nur der zuletzt erfasste Wert verfügbar.

```
(?J)(?<a>...)(?<a>...)
(?|(?<a>...)|(?<a>...))
```

Gruppen erfassen online lesen: https://riptutorial.com/de/regex/topic/660/gruppen-erfassen

Kapitel 12: Lookahead und Lookbehind

Syntax

- **Positiver Lookahead:** `(?=pattern)`
- **Negativer Lookahead:** `(?!pattern)`
- **Positives Aussehen** : `(?<=pattern)`
- **Negatives Aussehen** : `(?<!pattern)`

Bemerkungen

Wird nicht von allen Regex-Engines unterstützt.

Darüber hinaus beschränken viele Regex-Engines die Muster in Lookhinds auf Saiten mit fester Länge. Zum Beispiel sollte das Muster `(?<=a+)b` mit dem `b` in `aaab` , wirft jedoch einen Fehler in Python.

Erfassungsgruppen sind erlaubt und funktionieren wie erwartet, einschließlich Rückreferenzen. Das Lookahead / Lookbehind selbst ist jedoch keine Fanggruppe.

Examples

Grundlagen

Ein **positiver Lookahead** `(?=123)` , dass dem Text das angegebene Muster folgt, ohne dass das Muster in die Übereinstimmung einbezogen wird. In ähnlicher Weise behauptet ein **positiver Look hinter** `(?<=123)` , dass dem Text das angegebene Muster vorangeht. Ersetzen des `=` durch `!` negiert die Behauptung.

Eingabe : `123456`

- `123(?=456)` entspricht `123` (*positiver Lookahead*)
- `(?<=123)456` Treffer `456` (*positiver Blick hinter*)
- `123(?!456)` schlägt fehl (*negativer Lookahead*)
- `(?<!123)456` schlägt fehl (*negativer Lookbehind*)

Eingabe : `456`

- `123(?=456)` schlägt fehl
- `(?<=123)456` schlägt fehl
- `123(?!456)` schlägt fehl
- `(?<!123)456` entspricht `456`

Verwenden von lookbehind zum Testen von Endungen

Ein Lookbehind kann am Ende eines Musters verwendet werden, um sicherzustellen, dass es endet oder nicht in einer bestimmten Weise.

`([az ]+|[AZ ]+)(?<! )` entspricht Sequenzen von nur Kleinbuchstaben oder nur Großbuchstaben, wobei nachfolgende Leerzeichen ausgeschlossen werden.

Simulieren von LookLang mit variabler Länge mit \ K

Einige reguläre Ausdrücke (Perl, PCRE, Oniguruma, Boost) unterstützen nur Lookbehinds mit fester Länge, bieten jedoch die Funktion `\K` , mit der der Lookbehind mit variabler Länge zu Beginn eines Musters simuliert werden kann. Nach einer Begegnung mit `\K` , der angepasste Text bis zu diesem Punkt wird verworfen, und nur der Text den Teil des Musters *folgenden* passenden `\K` wird im Endergebnis gehalten.

```
ab+\Kc
```

Ist äquivalent zu:

```
(?<=ab+)c
```

Im Allgemeinen ein Muster der Form:

```
(subpattern A)\K(subpattern B)
```

Am Ende ähnlich sein:

```
(?<=subpattern A)(subpattern B)
```

Außer, wenn das B-Untermuster mit demselben Text wie das A-Untermuster übereinstimmen kann, kann dies zu geringfügig unterschiedlichen Ergebnissen führen, da das A-Untermuster den Text im Gegensatz zu einem echten Lookbehind immer noch verbraucht.

Lookahead und Lookbehind online lesen: https://riptutorial.com/de/regex/topic/639/lookahead-und-lookbehind

Kapitel 13: Match zurücksetzen: \ K

Bemerkungen

Regex101 definiert \ K-Funktionalität als:

> `\K` setzt den Startpunkt des gemeldeten Matches zurück. Alle zuvor verbrauchten Charaktere werden nicht mehr in das endgültige Spiel aufgenommen

Die `\K` Escape-Sequenz wird von mehreren Engines, Sprachen oder Tools unterstützt, z.

- Boost (seit ???)
- grep -P ← *verwendet PCRE*
- Oniguruma (seit 5.13.3)
- PCRE (seit 7.2)
- Perl (seit 5.10.0)
- PHP (seit 5.2.4)
- Ruby (seit 2.0.0)

... und (bisher) nicht unterstützt von:

- .NETZ
- awk
- bash
- GNU
- ICU
- Java
- Javascript
- Notepad ++
- Ziel c
- POSIX
- Python
- Qt / QRegExp
- sed
- Tcl
- vim
- XML
- XPath

Examples

Suchen und Ersetzen mit \ K-Operator

In Anbetracht des Textes:

Foo: Bar

Ich möchte alles, was "foo:" folgt, durch "baz" ersetzen, aber ich möchte "foo:" behalten. Dies könnte mit einer Capturing-Gruppe wie folgt durchgeführt werden:

```
s/(foo: ).*/$1baz/
```

Welche Ergebnisse ergeben sich aus dem Text:

foo: baz

Beispiel 1

oder wir könnten `\K` , was alles, was zuvor abgeglichen wurde, mit einem Muster wie folgt "vergisst":

```
s/foo: \K.*/baz/
```

Der reguläre Ausdruck stimmt mit "foo:" überein und trifft dann auf `\K` Die zuvor übereinstimmenden Zeichen werden als selbstverständlich betrachtet und vom regulären Ausdruck zurückgelassen. Dies bedeutet, dass nur die mit `.*` Übereinstimmende Zeichenfolge durch "baz" ersetzt wird.

foo: baz

Beispiel 2

Match zurücksetzen: \ K online lesen: https://riptutorial.com/de/regex/topic/1338/match-zurucksetzen----k

Kapitel 14: Nützlicher Regex Showcase

Examples

Übereinstimmung mit einem Datum

Sie sollten sich daran erinnern, dass Regex dazu gedacht ist, ein Datum abzugleichen (oder nicht). Zu sagen, dass ein Datum *gültig* ist , ist ein viel komplizierterer Kampf, da es eine Menge Ausnahmebehandlung erfordert (siehe Bedingungen für das Schaltjahr).

Beginnen wir mit dem Abgleichen des Monats (1 - 12) mit einer optionalen führenden 0:

```
0?[1-9]|1[0-2]
```

Um den Tag abzugleichen, auch mit einer optionalen führenden 0:

```
0?[1-9]|[12][0-9]|3[01]
```

Und um das Jahr abzugleichen (nehmen wir einfach an, dass der Bereich 1900 - 2999 reicht):

```
(?:19|20)[0-9]{2}
```

Das Trennzeichen kann ein Leerzeichen, ein Bindestrich, ein Schrägstrich, ein Leerzeichen usw. sein. Sie können alles hinzufügen, was Sie als Trennzeichen verwenden:

```
[-\\/ ]?
```

Jetzt verketten Sie das Ganze und erhalten:

```
(0?[1-9]|1[0-2])[-\\/ ]?(0?[1-9]|[12][0-9]|3[01])[-/ ]?(?:19|20)[0-9]{2} // MMDDYYYY
(0?[1-9]|[12][0-9]|3[01])[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Wenn Sie etwas pedantischer sein möchten, können Sie eine Rückwärtsreferenz verwenden, um sicherzustellen, dass die beiden Trennzeichen gleich sind:

```
(0?[1-9]|1[0-2])([-\\/ ]?)(0?[1-9]|[12][0-9]|3[01])\2(?:19|20)[0-9]{2} // MMDDYYYY
                                                  ^ refer to [-/ ]
(0?[1-9]|[12][0-9]|3[01])([-\\/ ]?)(0?[1-9]|1[0-2])\2(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}([-\\/ ]?)(0?[1-9]|1[0-2])\2(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Übereinstimmung mit einer E-Mail-Adresse

> Das Abgleichen einer E-Mail-Adresse innerhalb eines Strings ist eine schwierige Aufgabe, da die Spezifikation, die sie definiert, RFC2822 , komplex ist und die Implementierung als Regex schwierig gestaltet. Weitere Informationen dazu, warum es

> nicht ratsam ist, eine E-Mail mit einem regulären Ausdruck abzugleichen, finden Sie im Antipattern-Beispiel, wenn Sie keinen regulären Ausdruck für das Übereinstimmen von E-Mails verwenden . Der beste Hinweis, den Sie auf dieser Seite beachten sollten, ist die Verwendung einer von Experten geprüften und weit verbreiteten Bibliothek in Ihrer bevorzugten Sprache, um dies zu implementieren.

Bestätigen Sie ein E-Mail-Adressformat

Wenn Sie einen Eintrag schnell überprüfen müssen, um sicherzugehen, dass er *wie* eine E-Mail *aussieht* , sollten Sie ihn möglichst einfach halten:

```
^\S{1,}@\S{2,}\.\S{2,}$
```

Dieser reguläre Ausdruck überprüft, ob es sich bei der Mail-Adresse um eine nicht durch Leerzeichen getrennte Folge von Zeichen mit einer Länge von mehr als Eins handelt, gefolgt von einem `@` , gefolgt von zwei Folgen von Nicht-Leerzeichen mit der Länge zwei oder mehr, die durch ein Zeichen getrennt werden `.` . Sie ist nicht perfekt und validiert möglicherweise ungültige Adressen (je nach Format). Vor allem aber werden gültige Adressen nicht ungültig.

Überprüfen Sie, ob die Adresse vorhanden ist

Die einzige zuverlässige Methode, um die Gültigkeit einer E-Mail zu überprüfen, besteht darin, auf ihre Existenz zu prüfen. Früher gab es den `VRFY` SMTP-Befehl, der für diesen Zweck entwickelt wurde, aber leider ist er nach dem Missbrauch durch Spammer jetzt nicht mehr verfügbar .

Die einzige Möglichkeit, um zu überprüfen, ob die Mail gültig ist und existiert, ist das Senden einer E-Mail an diese Adresse.

Riesige Regex-Alternativen

Es ist jedoch nicht unmöglich, eine Adressen-E-Mail mit einem regulären Ausdruck zu überprüfen. Das einzige Problem ist, dass je näher diese Ausdrücke sind, desto größer werden sie und als Folge sind sie unmöglich schwer zu lesen und zu pflegen. Im Folgenden finden Sie Beispiele für solche genaueren Regex, die in einigen Bibliotheken verwendet werden.

> **Re** Die folgenden **regulären Ausdrücke** werden zu Dokumentations- und Lernzwecken angegeben. Das Kopieren in Ihren Code ist eine schlechte Idee. Verwenden Sie diese Bibliothek stattdessen direkt, sodass Sie sich auf Upstream-Code- und Peer-Entwickler verlassen können, um den Code für die E-Mail-Analyse auf dem neuesten Stand zu halten und zu pflegen.

Perl-Adressenanpassungsmodul

Die besten Beispiele für solche Regex sind in einigen Sprachen Standardbibliotheken. In der Perl-Bibliothek befindet sich beispielsweise ein Modul aus dem Modul `RFC::RFC822::Address` , das nach RFC so genau wie möglich zu sein versucht. Für Ihre Neugier finden Sie unter dieser URL eine

Version dieser regulären Ausdrücke , die aus der Grammatik generiert wurde. Wenn Sie versucht sind, diese zu kopieren, fügen Sie sie hier ein. Hier ein Zitat des Regex-Autors:

> " *Ich pflege den regulären Ausdruck [linked] nicht. Es können Fehler darin enthalten sein, die bereits im Perl-Modul behoben wurden.* "

NET-Adressenanpassungsmodul

Eine andere, kürzere Variante ist diejenige, die von der .Net-Standardbibliothek im `EmailAddressAttribute` Modul verwendet wird :

```
^((([a-z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])+(\.([a-
z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])+)*)|((\x22)((((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(([\x01-\x08\x0b\x0c\x0e-
\x1f\x7f]|\x21|[\x23-\x5b]|[\x5d-\x7e]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(\\([\x01-
\x09\x0b\x0c\x0d-\x7f]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])))*(((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(\x22)))@((([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])|(([a-z]|\d|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-
|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])*([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])))\.)+(([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(([a-z]|[\u00A0-
\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])*([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])))\.?$
```

Aber selbst wenn es *kürzer ist, ist* es immer noch zu groß, um lesbar und leicht zu pflegen.

Ruby-Adressenanpassungsmodul

In Ruby wird im Modul rfc822 eine Zusammensetzung von Regex verwendet, um eine Adresse zu finden. Dies ist eine nette Idee, denn wenn Fehler gefunden werden, ist es einfacher, den Regex-Teil zu lokalisieren, um ihn zu ändern und zu beheben.

Python-Adressenanpassungsmodul

Als Gegenbeispiel verwendet das Python- E-Mail-Parsing-Modul keinen regulären Ausdruck, sondern implementiert es mit einem Parser.

Übereinstimmung mit einer Telefonnummer

So passen Sie einen Präfixcode an (a `+` oder (00), dann eine Zahl von 1 bis 1939 mit optionalem Leerzeichen):
Es wird nicht nach einem *gültigen* Präfix gesucht, sondern nach einem Präfix. Siehe die vollständige Liste der Präfixe

```
(?:00|\+)?[0-9]{4}
```

Da die Länge der gesamten Telefonnummer höchstens 15 beträgt, können wir bis zu 14 Ziffern suchen:
Für das Präfix wird mindestens eine Ziffer ausgegeben

```
[0-9]{1,14}
```

Die Zahlen können Leerzeichen, Punkte oder Bindestriche enthalten und können mit 2 oder 3 gruppiert werden.

```
(?:[ .-][0-9]{3}){1,5}
```

Mit dem optionalen Präfix:

```
(?:(?:00|\+)?[0-9]{4})?(?:[ .-][0-9]{3}){1,5}
```

Wenn Sie ein bestimmtes Länderformat verwenden möchten, können Sie diese Suchabfrage verwenden und das Land hinzufügen. Die Frage wurde sicherlich bereits gestellt.

Stimmen Sie eine IP-Adresse ab

IPv4

Um das IPv4-Adressformat abzugleichen, müssen Sie die Nummern `[0-9]{1,3}` dreimal `{3}` getrennt durch Punkte `\.` , überprüfen `\.` und endet mit einer anderen Nummer.

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

Dieser reguläre Ausdruck ist zu einfach. Wenn Sie möchten, dass er genau ist, müssen Sie überprüfen, ob die Zahlen zwischen `0` und `255` , wobei der Regex oben in einer beliebigen Position `444` akzeptiert. Sie möchten auf 250-255 mit `25[0-5]` oder einem anderen Wert 200 `2[0-4][0-9]` oder einem Wert von 100 oder weniger mit `[01]?[0-9][0-9]` prüfen. `[01]?[0-9][0-9]` . Sie möchten prüfen, ob ein Punkt `\.` folgt `\.` dreimal `{3}` und dann einmal ohne Punkt.

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

IPv6

IPv6 - Adressen haben die Form von 8 16-Bit - Worten hex begrenzt mit dem Doppelpunkt (`:`) Charakter. In diesem Fall prüfen wir nach 7 Wörtern, gefolgt von Doppelpunkten, gefolgt von einem, das nicht ist. Wenn ein Wort führende Nullen hat, werden diese *möglicherweise* abgeschnitten. Dies bedeutet, dass jedes Wort zwischen 1 und 4 Hexadezimalstellen enthalten kann.

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$
```

Dies reicht jedoch nicht aus. Da IPv6-Adressen ziemlich "wortreich" werden können, gibt der Standard an, dass nur Zero-Words durch `::` ersetzt werden können. Dies darf nur einmal in einer Adresse (für 1 bis 7 aufeinanderfolgende Wörter) erfolgen, da dies ansonsten unbestimmt wäre. Dies führt zu einer Reihe von (eher unangenehmen) Variationen:

```
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$
```

```
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Wenn Sie nun alles zusammenstellen (alternierend), erhalten Sie:

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$|
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Stellen Sie sicher, dass Sie es in einem mehrzeiligen Modus und mit vielen Kommentaren aufschreiben, damit derjenige, der unweigerlich herauszufinden versucht, was dies bedeutet, nicht mit einem stumpfen Gegenstand hinter Ihnen her kommt.

Bestätigen Sie eine 12- und 24-Stunden-Zeichenfolge

Für ein 12-Stunden-Format können Sie Folgendes verwenden:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9]\s*[ap]m$
```

Woher

- `(?:0?[0-9]|1[0-2])` ist die Stunde
- `[-:]` ist das Trennzeichen, das an Ihre Bedürfnisse angepasst werden kann
- `[0-5][0-9]` ist die Minute
- `\s*[ap]m` folgt einer beliebigen Anzahl von Whitespace-Zeichen und `am` oder `pm`

Wenn Sie die Sekunden brauchen:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9][-:][0-5][0-9]\s*[ap]m$
```

Für ein 24-Stunden-Format:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9]$
```

Woher:

- `(?:[01][0-9]|2[0-3])` ist die Stunde
- `[-:h]` das Trennzeichen, das an Ihre Bedürfnisse angepasst werden kann
- `[0-5][0-9]` ist die Minute

Mit den Sekunden:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9][-:m][0-5][0-9]$
```

Wobei `[-:m]` ein zweites Trennzeichen ist, wobei `h` für Stunden durch ein `m` für Minuten ersetzt wird und `[0-5][0-9]` das zweite ist.

Entsprechende britische Postleitzahl

Regex für Postleitzahlen in Großbritannien

Das Format ist wie folgt, wobei A einen Buchstaben und 9 eine Ziffer bedeutet:

Format	Deckung	Beispiel
Zelle	Zelle	
AA9A 9AA	WC-PLZ-Bereich; EC1 – EC4, NW1W, SE1P, SW1	EC1A 1BB
A9A 9AA	E1W, N1C, N1P	W1A 0AX
A9 9AA, A99 9AA	B, E, G, L, M, N, S, W	M1 1AE, B33 8TH
AA9 9AA, AA99 9AA	Alle anderen Postleitzahlen	CR2 6XH, DN55 1PT

```
(GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY])))) [0-9][A-Z-[CIKMOV]]{2})
```

Wo erster Teil:

```
 (GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY]))))
```

Zweite:

```
[0-9][A-Z-[CIKMOV]]{2})
```

Nützlicher Regex Showcase online lesen: https://riptutorial.com/de/regex/topic/3605/nutzlicher-regex-showcase

Kapitel 15: Possessive Quantifizierer

Bemerkungen

NB Emulation von Possessiv-Quantifizierern

Examples

Grundlegende Verwendung von Possessive Quantifiers

Possessive Quantifizierer sind eine weitere Klasse von Quantifizierern in vielen Regex-Varianten, die es ermöglichen, das Zurückverfolgen effektiv für ein gegebenes Token zu deaktivieren. Dies kann dazu beitragen, die Leistung zu verbessern und in bestimmten Fällen Übereinstimmungen zu vermeiden.

Die Klasse der besitzergreifend quantifiers kann von faulen oder gierigen Quantifizierer durch Zugabe eines unterscheiden `+` nach dem Quantifizierer, wie unten zu sehen:

Quantor	Gierig	Faul	Besitzergreifend
Null oder mehr	`*`	`*?`	`*+`
Ein oder mehr	`+`	`+?`	`++`
Null oder eins	`?`	`??`	`?+`

Betrachten Sie zum Beispiel die beiden Muster `".*"` Und `".*+"` , Die mit der Zeichenfolge `"abc"d` In beiden Fällen stimmt das `"` am Anfang der Zeichenfolge" überein, aber danach weisen die beiden Muster unterschiedliche Verhaltensweisen und Ergebnisse auf.

Der gierige Quantifizierer schlurft dann den Rest der Zeichenkette `abc"d` . Da dies nicht zum Muster passt, wird er zurückverfolgt und das `d` , wobei der **Quantifizierer** `abc"` bleibt. Da dies immer noch nicht mit dem Muster übereinstimmt, wird der Quantifizierer das `"` löschen `"` , so dass nur noch " `abc` " enthalten ist. Dies stimmt mit dem Muster überein (da `"` das `"` mit einem Literal und nicht mit dem Quantifizierer übereinstimmt), und der Regex gibt den Erfolg an.

Der Possessiv-Quantifizierer verschluckt auch den Rest der Saite, wird jedoch im Gegensatz zum gierigen Quantifizierer nicht zurückgespult. Da der Inhalt `abc"d` den Rest des Musters der Übereinstimmung nicht zulässt, wird der reguläre Ausdruck angehalten und der Fehler wird gemeldet.

Da die Possessiv-Quantifizierer kein Backtracking durchführen, können sie bei langen oder komplexen Mustern zu einer erheblichen Leistungssteigerung führen. Sie können jedoch gefährlich sein (wie oben dargestellt), wenn man nicht genau weiß, wie Quantifizierer intern arbeiten.

Possessive Quantifizierer online lesen: https://riptutorial.com/de/regex/topic/5916/possessive-quantifizierer

Kapitel 16: Regex Fallstricke

Examples

Warum stimmt der Punkt (.) Nicht mit dem Zeilenvorschubzeichen ("\ n") überein?

`.*` in Regex bedeutet im Grunde " **alles** bis zum Ende der Eingabe fangen".

Also, für einfache Saiten, wie `hello world` , funktioniert `.*` perfekt. Wenn Sie jedoch eine Zeichenfolge haben, die z. B. Zeilen in einer Datei darstellt, werden diese Zeilen durch ein *Trennzeichen getrennt* , z. B. `\n` (Newline) auf Unix-ähnlichen Systemen und `\r\n` (Wagenrücklauf und Newline) Windows.

In den meisten Regex-Engines ist `.` **stimmt nicht** mit Zeilenumbrüchen überein, so dass die Übereinstimmung am Ende jeder *logischen Zeile* endet. Wenn Sie wollen `.` Um **wirklich** alles, einschließlich der Zeilenumbrüche, zu finden, müssen Sie den Modus "dot-matches-all" in Ihrer `re.DOTALL` Engine `re.DOTALL` (beispielsweise `re.DOTALL` Flag in Python oder `/s` in PCRE).

Warum überspringt ein Regex einige schließende Klammern und passt sie danach an?

Betrachten Sie dieses Beispiel:

> Er ging in das Café "Dostoevski" und sagte: "Guten Abend."

Hier haben wir zwei Sätze von Zitaten. Nehmen wir an, wir wollen beide zusammenbringen, so dass unsere Regex-Matches bei `"Dostoevski"` **und** `"Good evening."` übereinstimmen `"Good evening."`

Zunächst könnten Sie versucht sein, es einfach zu halten:

```
".*"  # matches a quote, then any characters until the next quote
```

Aber es funktioniert nicht: Es stimmt mit dem ersten Zitat in `"Dostoevski"` und **bis** zum Schlusszitat in `"Good evening."` überein `"Good evening."` , einschließlich der `and said:` Teil. Regex101-Demo

Warum ist das passiert?

Dies geschieht, weil die Regex-Engine, wenn sie auf `.*` Stößt, die gesamte Eingabe bis zum Ende "frisst". Dann muss es mit dem Finale übereinstimmen `"` . Es wird also vom Ende des Matches `"` und der übereinstimmende Text losgelassen, bis das erste `"` gefunden `"` ist - und es ist natürlich das letzte `"` im Match , am Ende von `"Good evening."` Teil.

Wie kann man dies verhindern und genau auf die ersten

Zitate passen?

Verwenden Sie `[^"]*` . Es frisst nicht alle Eingaben - nur bis zum ersten `"` , genauso wie es benötigt wird. Regex101-Demo

Regex Fallstricke online lesen: https://riptutorial.com/de/regex/topic/10747/regex-fallstricke

Kapitel 17: Regex für die Passwortüberprüfung

Examples

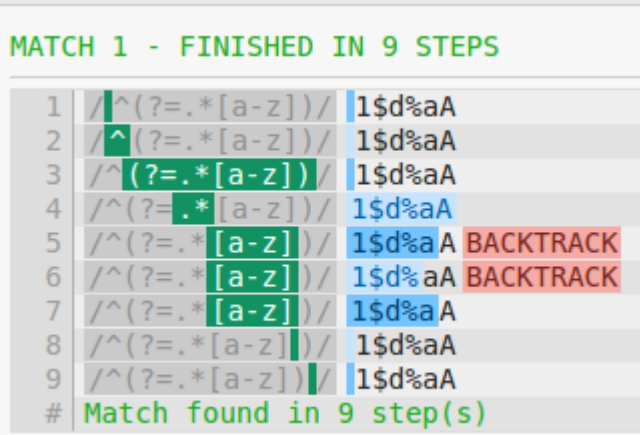
Ein Passwort, das mindestens 1 Großbuchstabe, 1 Kleinbuchstabe, 1 Ziffer, 1 Sonderzeichen enthält und eine Länge von mindestens 10 hat

Da sich die Zeichen / Ziffern an einer beliebigen Stelle in der Zeichenfolge befinden können, benötigen wir Lookaheads. Lookaheads haben `zero width` von `zero width` was bedeutet, dass sie keine Zeichenfolge verbrauchen. In einfachen Worten wird die Position der Überprüfung auf die ursprüngliche Position zurückgesetzt, nachdem jede Voraussagebedingung erfüllt ist.

Annahme : - *Nicht-Wort-Zeichen werden als besonders angesehen*

```
^(?=.{10,}$)(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*\W).*$
```

Bevor wir zur Erläuterung gehen, werfen wir einen Blick auf die Funktionsweise des Regex `^(?=.*[az])` (*Länge wird hier nicht berücksichtigt*) in Zeichenfolge `1$d%aA`



Bildnachweis : - **https://regex101.com/**

Dinge zu beachten

- Die Prüfung wird aufgrund des Ankertags `^` vom Anfang des Strings gestartet.
- Die Position der Überprüfung wird auf den Start zurückgesetzt, nachdem die Bedingung des Lookahead erfüllt ist.

Regex-Aufschlüsselung

```
^ #Starting of string
 (?=.{10,}$) #Check there is at least 10 characters in the string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[a-z]) #Check if there is at least one lowercase in string.
```

```
            #As this is lookahead the position of checking will reset to starting again
 (?=.*[A-Z]) #Check if there is at least one uppercase in string.
            #As this is lookahead the position of checking will reset to starting again
 (?=.*[0-9]) #Check if there is at least one digit in string.
            #As this is lookahead the position of checking will reset to starting again
 (?=.*\W) #Check if there is at least one special character in string.
          #As this is lookahead the position of checking will reset to starting again
.*$ #Capture the entire string if all the condition of lookahead is met. This is not required
if only validation is needed
```

Wir können auch die *nicht-gierige* Version des obigen Regex verwenden

```
^(?=.{10,}$)(?=.*?[a-z])(?=.*?[A-Z])(?=.*?[0-9])(?=.*?\W).*$
```

Ein Passwort mit mindestens 2 Großbuchstaben, 1 Kleinbuchstaben und 2 Ziffern und einer Länge von mindestens 10

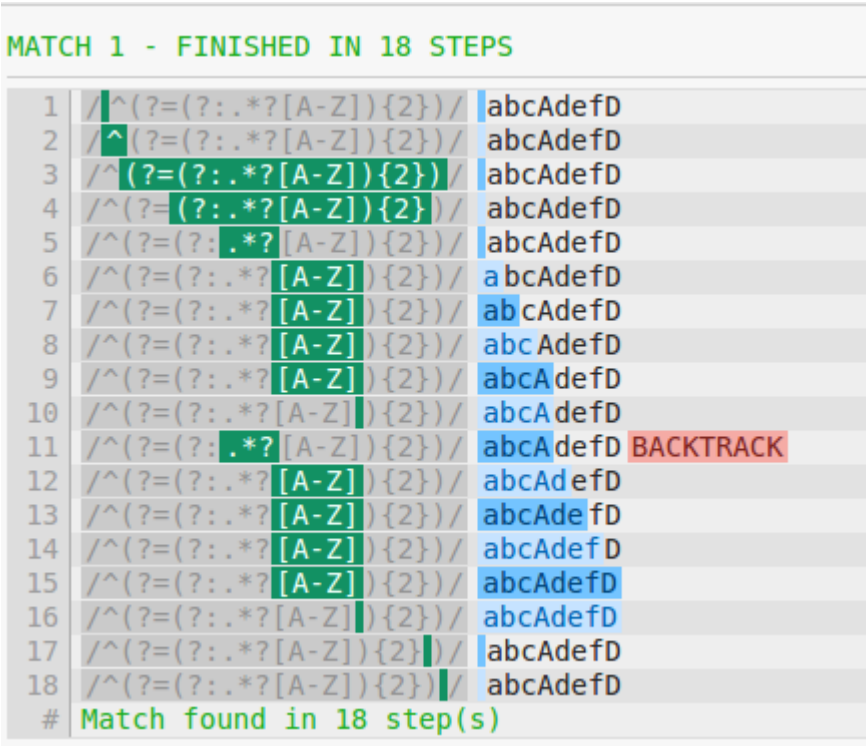
Dies kann mit ein paar Modifikationen in der obigen Regex vorgenommen werden

```
 ^(?=.{10,}$)(?=(?:.*?[A-Z]){2})(?=.*?[a-z])(?=(?:.*?[0-9]){2}).*$
```

oder

```
 ^(?=.{10,}$)(?=(?:.*[A-Z]){2})(?=.*[a-z])(?=(?:.*[0-9]){2}).*
```

Mal sehen, wie ein einfacher regex `^(?=(?:.*?[AZ]){2})` auf dem String `abcAdefD`



Bildnachweis : - **https://regex101.com/**

Regex für die Passwortüberprüfung online lesen: https://riptutorial.com/de/regex/topic/5340/regex-

fur-die-passwortuberprufung

Kapitel 18: Regex-Modifikatoren (Flags)

Einführung

Muster für reguläre Ausdrücke werden häufig mit *Modifizierern* (auch *Flags genannt*) verwendet, die das Regex-Verhalten neu definieren. Regex-Modifikatoren können *regulär* (z. B. `/abc/i`) und *inline* (oder *eingebettet*) (z. B. `(?i)abc`) sein. Die gebräuchlichsten Modifizierer sind globale, Groß- und Kleinschreibung, Multiline- und Dotall-Modifizierer. Regex-Flavors unterscheiden sich jedoch in der Anzahl der unterstützten Regex-Modifikatoren und ihren Typen.

Bemerkungen

PCRE-Modifikatoren

Modifikator	In der Reihe	Beschreibung
PCRE_CASELESS	(?ich)	Groß- und Kleinschreibung
PCRE_MULTILINE	(? m)	Mehrzeilige Übereinstimmung
PCRE_DOTALL	(? s)	. passt zu neuen Zeilen
PCRE_ANCHORED	(?EIN)	Das Meta-Zeichen ^ stimmt nur beim Start überein
PCRE_EXTENDED	(? x)	Leerzeichen werden ignoriert
PCRE_DOLLAR_ENDONLY	n / a	Metazeichen $ stimmt nur am Ende überein
PCRE_EXTRA	(? X)	Strikte Fluchtanalyse
PCRE_UTF8		Verarbeitet `UTF-8` Zeichen
PCRE_UTF16		Verarbeitet `UTF-16` Zeichen
PCRE_UTF32		Verarbeitet `UTF-32` Zeichen
PCRE_UNGREEDY	(? U)	Setzt die Engine auf Lazy Matching
PCRE_NO_AUTO_CAPTURE	(? :)	Deaktiviert die automatische Erfassung von Gruppen

Java-Modifikatoren

Modifikator (`Pattern.###`)	Wert	Beschreibung
UNIX_LINES	1	Aktiviert den Unix- Zeilenmodus.
CASE_INSENSITIVE	2	Aktiviert die Groß- und Kleinschreibung ohne Übereinstimmung.
BEMERKUNGEN	4	Erlaubt Leerzeichen und Kommentare in einem Muster.
MULTILINE	8	Aktiviert den mehrzeiligen Modus.
LITERAL	16	Ermöglicht das literale Parsen des Musters.
DOTALL	32	Aktiviert den Dotall-Modus.
UNICODE_CASE	64	Aktiviert die Unicode-Erkennung von Groß- und Kleinschreibung.
CANON_EQ	128	Ermöglicht kanonische Äquivalenz.
UNICODE_CHARACTER_CLASS	256	Aktiviert die Unicode-Version von Vordefinierten Zeichenklassen und POSIX-Zeichenklassen.

Examples

DOTALL-Modifikator

Ein Regex-Muster, bei dem ein DOTALL-Modifikator (in den meisten Regex-Varianten mit `s` ausgedrückt) das Verhalten von ändert `.` So kann es mit einem Newline-Symbol (LF) übereinstimmen:

```
/cat (.*?) dog/s
```

Dieser Regex im Perl-Stil passt zu einer Zeichenfolge wie `"cat fled from\na dog"` , die `"fled from\na"` gefangen `"fled from\na"` in Gruppe 1 `"fled from\na"` .

Eine Inline-Version: `(?s)` (zB `(?s)cat (.*?) dog`)

Hinweis : In Ruby ist der Modifikator DOTALL der Modifikator `m` , `Regexp::MULTILINE` (z. B. `/a.*b/m`).

Hinweis : JavaScript bietet keinen Modifikator DOTALL, also a `.` darf niemals mit einem Zeilenvorschubzeichen übereinstimmen. Um den gleichen Effekt zu erzielen, ist ein Workaround erforderlich, z `.` s mit einer Catch-All-Zeichenklasse wie `[\S\s]` oder einer *Nicht-Nichts-*

Zeichenklasse `[^]` (dieses Konstrukt wird jedoch von allen anderen Engines als Fehler behandelt und ist daher nicht portierbar).

MULTILINE-Modifikator

Ein anderes Beispiel ist ein MULTILINE-Modifikator (normalerweise mit `m` Flag ausgedrückt (nicht in Oniguruma (z. B. Ruby), der `m` , um einen DOTALL-Modifikator zu bezeichnen)), der die `^` und `$` -Anker mit dem Anfang / Ende einer *Zeile* und nicht mit dem Anfang / Ende übereinstimmt der ganzen Saite.

```
/^My Line \d+$/gm
```

findet alle *Zeilen* , die mit `My Line` , und enthalten dann ein Leerzeichen und 1+ Ziffern bis zum Zeilenende.

Eine Inline-Version: `(?m)` (zB `(?m)^My Line \d+$`)

HINWEIS : In Oniguruma (z. B. in Ruby) und auch in fast allen Texteditoren, die Regexps unterstützen, geben die Ankreuzzeichen `^` und `$` *standardmäßig die* Anfangs- und Endpositionen *von Zeilen* an. Sie müssen `\A` , um den gesamten Anfang des Dokuments / der Zeichenfolge zu definieren, und `\z` , um das Ende des Dokuments / der Zeichenkette anzugeben. Der Unterschied zwischen `\Z` und `\z` besteht darin, dass der Erstgenannte vor dem letzten Zeilenvorschub (LF) am Ende der Zeichenfolge `/\Astring\Z/` (z. B. `/\Astring\Z/` findet eine Übereinstimmung in `"string\n"`) (außer Python, wobei `\Z` Verhalten gleich `\z` und `\z` Anker nicht unterstützt wird).

IGNORE CASE-Modifizierer

Der häufigste Modifikator, der den Fall ignoriert, ist `i` :

```
/fog/i
```

`foG` `Fog` , `foG` usw.

Die Inline-Version des Modifikators sieht wie `(?i)` .

Anmerkungen:

In Java wird *bei der Übereinstimmung zwischen Groß- und Kleinschreibung* standardmäßig *davon ausgegangen, dass nur Zeichen im US-ASCII-Zeichensatz abgeglichen werden. Unicode-* `UNICODE_CASE` *Übereinstimmungen ohne* `UNICODE_CASE` *Groß- und Kleinschreibung können aktiviert werden, indem das Flag* `UNICODE_CASE` *in Verbindung mit diesem Flag (* `CASE_INSENSITIVE` *) angegeben wird .* (zB `Pattern p = Pattern.compile("YOUR_REGEX", Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);`). Weitere Informationen hierzu finden Sie unter *Case-Insensitive Matching in Java RegEx* . Außerdem kann `UNICODE_CHARACTER_CLASS` verwendet werden, um übereinstimmende Unicode- `UNICODE_CHARACTER_CLASS` .

VERBOSE / COMMENT / IgnorePatternWhitespace-Modifizierer

Der Modifikator, mit dem Leerzeichen in einigen Teilen des Musters verwendet werden können, um es für eine bessere Lesbarkeit zu formatieren und Kommentare zuzulassen, die mit `#` :

```
/(?x)^          # start of string
  (?=\D*\d)     # the string should contain at least 1 digit
  (?!\d+$)      # the string cannot consist of digits only
  \#            # the string starts with a hash symbol
  [a-zA-Z0-9]+ # the string should have 1 or more alphanumeric symbols
  $             # end of string
/
```

Beispiel für einen String: `#word1here` . Beachten Sie, dass das `#` -Symbol mit einem Escapezeichen versehen ist, um ein Literal `#` zu kennzeichnen, das Teil eines Musters ist.

Nicht maskierte Leerräume im Muster für reguläre Ausdrücke werden ignoriert. Umgehen Sie es, um es zu einem Teil des Musters zu machen.

Normalerweise wird der Whitespace innerhalb von Zeichenklassen (`[...]`) mit Ausnahme von Java als Literal Whitespace behandelt.

Erwähnenswert ist auch, dass in PCRE, .NET, Python, Ruby Oniguruma, Intensivstation, Boost-Regex-Flavours `(?#:...)` Kommentare im Regex-Muster verwendet werden können.

Expliziter Erfassungsmodifikator

Dies ist ein .NET-regulärer Modifikator, der mit `n` ausgedrückt wird. Bei Verwendung werden nicht benannte Gruppen (wie `(\d+)`) nicht erfasst. Nur gültige Captures werden explizit als Gruppen bezeichnet (z. B. `(?<name> subexpression)`).

```
(?n)(\d+)-(\w+)-(?<id>\w+)
```

`123-1_abc-00098` mit dem gesamten `123-1_abc-00098` , aber `(\d+)` und `(\w+)` erstellen keine Gruppen im resultierenden Übereinstimmungsobjekt. Die einzige Gruppe ist `${id}` . Siehe Demo .

UNICODE-Modifizierer

Der UNICODE-Modifizierer, normalerweise ausgedrückt als `u` (PHP, Python) oder `U` (Java), veranlasst die Regex-Engine, das Muster und die Eingabezeichenfolge als Unicode-Strings und -Muster zu behandeln. Dabei werden die Musterkurzklassen wie `\w` , `\d` , `\s` usw. Unicode-fähig.

```
/\A\p{L}+\z/u
```

ist eine PHP-Regex, um Zeichenfolgen abzugleichen, die aus einem oder mehreren Unicode-Buchstaben bestehen. Siehe die Regex-Demo .

Beachten Sie, dass der Modifikator `/u` in **PHP** der PCRE-Engine ermöglicht, Zeichenfolgen als UTF8-Zeichenfolgen zu behandeln (durch `PCRE_UTF8` Verbs `PCRE_UTF8`) und die Abkürzungszeichenklassen im Unicode-Muster zu aktivieren (durch Aktivieren des `PCRE_UCP` Verbs siehe mehr auf *pcre.org*). .

Muster- und Betreff-Zeichenfolgen werden als UTF-8 behandelt. Dieser Modifikator ist ab PHP 4.1.0 oder höher unter Unix und ab PHP 4.2.3 unter Win32 verfügbar. Die UTF-8-Gültigkeit des Musters und des Subjekts wird seit PHP 4.3.5 geprüft. Bei einem ungültigen Betreff stimmt die Funktion preg_ * mit nichts überein. Ein ungültiges Muster löst einen Fehler der Stufe E_WARNING aus. UTF-8-Sequenzen mit fünf und sechs Oktetten gelten seit PHP 5.3.4 als ungültig (bzw. PCRE 7.3 2007-08-28). früher wurden diese als gültiges UTF-8 betrachtet.

In Python 2.x wirkt sich `re.UNICODE` nur auf das Muster selbst aus: *Machen Sie* `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` *und* `\S` *von der Unicode-Eigenschaftendatenbank abhängig.*

Eine Inline-Version: `(?u)` in Python, `(?U)` in Java. Zum Beispiel:

```
print(re.findall(ur"(?u)\w+", u"Dąb")) # [u'D\u0105b']
print(re.findall(r"\w+", u"Dąb"))      # [u'D', u'b']

System.out.println("Dąb".matches("(?U)\\w+")); // true
System.out.println("Dąb".matches("\\w+"));     // false
```

PCRE_DOLLAR_ENDONLY-Modifizierer

Der PCRE-kompatible *PCRE_DOLLAR_ENDONLY*- Modifizierer, mit dem der `$` anchor am *Ende der Zeichenfolge* übereinstimmt (mit Ausnahme der Position vor der letzten Zeile in der Zeichenfolge).

```
/^\d+$/D
```

entspricht

```
/^\d+\z/
```

und entspricht einer ganzen Zeichenfolge, die aus einer oder mehreren Ziffern besteht und nicht mit `"123\n"` übereinstimmt, sondern mit `"123"` .

PCRE_ANCHORED-Modifizierer

Ein weiterer PCRE-kompatibler Modifikator, der mit `/A` -Modifikator ausgedrückt wird. Wenn dieser Modifikator gesetzt ist, muss das Muster "verankert" werden, d. H., Es ist nur an den Anfang der gesuchten Zeichenfolge ("Betreff-Zeichenfolge") angepasst. Dieser Effekt kann auch durch entsprechende Konstrukte im Pattern selbst erreicht werden. Dies ist die einzige Möglichkeit, dies in Perl zu tun.

```
/man/A
```

ist das gleiche wie

```
/^man/
```

PCRE_UNGREEDY-Modifizierer

Das PCRE-kompatible PCRE_UNGREEDY-Flag, das mit `/U` ausgedrückt wird. Es wechselt die Gier innerhalb eines Musters: `/a.*?b/U` = `/a.*b/` und umgekehrt.

PCRE_INFO_JCHANGED-Modifizierer

Ein weiterer PCRE-Modifikator, der die Verwendung doppelter benannter Gruppen ermöglicht.

HINWEIS : Nur *Inline*- Version wird unterstützt - `(?J)` und muss am Anfang des Patterns platziert werden.

Wenn du benutzt

```
/(?J)\w+-(?:new-(?<val>\w+)|\d+-empty-(?<val>[^-]+)-collection)/
```

Die "val" -Gruppenwerte sind nie leer (werden immer gesetzt). Ein ähnlicher Effekt kann jedoch mit dem Zurücksetzen der Verzweigung erreicht werden.

PCRE_EXTRA-Modifizierer

Ein PCRE-Modifikator, der einen Fehler verursacht, wenn einem Backslash in einem Muster ein Buchstabe folgt, der keine besondere Bedeutung hat. Standardmäßig wird ein umgekehrter Schrägstrich gefolgt von einem Buchstaben ohne besondere Bedeutung als Literal behandelt.

Z.B

```
/big\y/
```

wird mit `bigy` , aber

```
/big\y/X
```

wird eine Ausnahme auslösen.

Inline-Version: `(?X)`

Regex-Modifikatoren (Flags) online lesen: https://riptutorial.com/de/regex/topic/5138/regex-modifikatoren--flags-

Kapitel 19: Rekursion

Bemerkungen

Rekursion ist meist in Perl-kompatiblen Geschmacksrichtungen verfügbar, z.

- Perl
- PCRE
- Oniguruma
- Boost

Examples

Rekurse das gesamte Muster

Das Konstrukt `(?R)` ist äquivalent zu `(?0)` (oder `\g<0>`) - es lässt Sie das gesamte Muster rekursieren:

```
<(?>[^<>]+|(?R))+>
```

Dadurch werden richtig ausgeglichene spitze Klammern mit einem beliebigen Text zwischen den Klammern abgeglichen, wie `<a<b>c<d>e>` .

Rekurse in ein Submuster

Sie können mit den folgenden Konstrukten (abhängig vom Flavour) in ein Submuster rekursieren, vorausgesetzt, `n` ist eine Erfassungsgruppennummer und `name` den Namen einer Erfassungsgruppe.

- `(?n)`
- `\g<n>`
- `\g'0'`
- `(?&name)`
- `\g<name>`
- `\g'name'`
- `(?P>name)`

Das folgende Muster:

```
\[(?<angle><(?&angle)*+>)*\]
```

Passt zu Text wie: `[<<><>><>]` - gut ausbalancierte spitze Klammern in eckigen Klammern. Rekursion wird häufig für das Abgleichen von ausgeglichenen Konstrukten verwendet.

Subpattern-Definitionen

Mit dem Konstrukt `(?(DEFINE) ... )` können Sie Submuster definieren, auf die Sie später durch Rekursion verweisen können. Wenn es im Muster vorkommt, wird es *nicht* abgeglichen.

Diese Gruppe sollte benannte Subpattern-Definitionen enthalten, auf die nur durch Rekursion zugegriffen werden kann. Sie können Grammatiken auf diese Weise definieren:

```
(?x) # ignore pattern whitespace
(?(DEFINE)
  (?<string> ".*?" )
  (?<number> \d+ )
  (?<value>
    \s* (?:
        (?&string)
      | (?&number)
      | (?&list)
    ) \s*
  )
  (?<list> \[ (?&value) (?: , (?&value) )* \] )
)
^(?&value)$
```

Dieses Muster überprüft den Text wie folgt:

```
[42, "abc", ["foo", "bar"], 10]
```

Beachten Sie, wie eine Liste einen oder mehrere Werte enthalten kann, und ein Wert kann selbst eine Liste sein.

Relative Gruppenreferenzen

Untermuster können mit ihrer *relativen* Gruppennummer referenziert werden:

- `(?-1)` wird in die *vorherige* Gruppe aufgenommen
- `(?+1)` kehrt in die *nächste* Gruppe zurück

Kann auch mit der `\g<N>` -Syntax verwendet werden.

Rückreferenzen in Rekursionen (PCRE)

In PCRE werden übereinstimmende Gruppen, die vor einer Rekursion für Rückreferenzen verwendet werden, in der Rekursion beibehalten. Aber nach der Rekursion haben sie sich alle auf das zurückgesetzt, was sie waren, bevor sie betreten wurden. Mit anderen Worten, übereinstimmende Gruppen in der Rekursion werden alle vergessen.

Zum Beispiel:

```
(?J)(?(DEFINE)(\g{a}(?<a>b)\g{a}))(?<a>a)\g{a}(?1)\g{a}
```

Streichhölzer

```
aaabba
```

Rekursionen sind atomar (PCRE)

In PCRE wird kein Trackback durchgeführt, nachdem der erste Treffer für eine Rekursion gefunden wurde. So

```
(?(DEFINE)(aaa|aa|a))(?1)ab
```

passt nicht zusammen

```
aab
```

denn nachdem es in der Rekursion mit `aa` übereinstimmte, versuchte es nie wieder, nur `a` .

Rekursion online lesen: https://riptutorial.com/de/regex/topic/739/rekursion

Kapitel 20: Rückverfolgung

Examples

Wodurch wird Backtracking verursacht?

Um eine Übereinstimmung zu finden, verbraucht die Regex-Engine Zeichen nacheinander. Wenn eine teilweise Übereinstimmung beginnt, merkt sich der Motor die Startposition, so dass er zurückgehen kann, falls die folgenden Zeichen die Übereinstimmung nicht vollenden.

- Wenn die Übereinstimmung abgeschlossen ist, erfolgt keine Rückverfolgung
- Wenn die Übereinstimmung nicht vollständig ist, führt die Engine die Zeichenfolge zurück (wie beim Zurückspulen eines alten Bandes), um eine vollständige Übereinstimmung zu finden.

Zum Beispiel: `\d{3}[az]{2}` gegen die Zeichenfolge `abc123def` wird wie `abc123def` durchsucht:

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does match \d (third one)
abc123def
      ^ Does match [a-z] (first one)
abc123def
       ^ Does match [a-z] (second one)
           MATCH FOUND
```

Ändern wir nun den Regex in `\d{2}[az]{2}` für dieselbe Zeichenfolge (`abc123def`):

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does not match [a-z]
abc123def
    ^ BACKTRACK to catch \d{2} => (23)
abc123def
      ^ Does match [a-z] (first one)
```

```
abc123def
       ^ Does match [a-z] (second one)
          MATCH FOUND
```

Warum kann Backtracking eine Falle sein?

Backtracking kann durch optionale Quantifizierer oder Alternativkonstrukte verursacht werden, da die Regex-Engine versucht, jeden Pfad zu erkunden. Wenn Sie den Regex `a+b` gegen `aaaaaaaaaaaaa` gibt es keine Übereinstimmung und die Engine wird es ziemlich schnell finden.

Wenn Sie den Regex jedoch in `(aa*)+b` ändern, wird die Anzahl der Kombinationen ziemlich schnell ansteigen. Die meisten (nicht optimierten) Engines versuchen alle Pfade zu erkunden und benötigen eine Ewigkeit, um eine Übereinstimmung zu finden oder eine zu werfen Timeout-Ausnahme. Dies wird als **katastrophales Backtracking bezeichnet** .

Natürlich scheint `(aa*)+b` ein Neuling-Fehler zu sein, aber es ist hier, um den Punkt zu veranschaulichen, und manchmal werden Sie mit demselben Problem enden, aber mit komplizierteren Mustern.

Ein extremer Fall von katastrophalem Backtracking tritt mit dem Regex `(x+x+)+y` (Sie haben es wahrscheinlich hier und hier gesehen), der exponentielle Zeit benötigt, um herauszufinden, dass eine Zeichenfolge, die `x` s enthält, und nichts anderes (z `xxxxxxxxxxxxxxxxxxx`) passen nicht zusammen.

Wie vermeide ich es?

Seien Sie so genau wie möglich und reduzieren Sie die möglichen Pfade so weit wie möglich. Beachten Sie, dass einige Regex-Matcher nicht anfällig für Backtracking sind, z. B. diejenigen, die in `awk` oder `grep` da sie auf Thompson NFA basieren.

Rückverfolgung online lesen: https://riptutorial.com/de/regex/topic/977/ruckverfolgung

Kapitel 21: Rückverweis

Examples

Grundlagen

Rückverweise werden verwendet, um denselben Text zu finden, der zuvor von einer Erfassungsgruppe abgeglichen wurde. Dies hilft sowohl bei der Wiederverwendung vorheriger Teile Ihres Musters als auch bei der Sicherstellung, dass zwei Teile einer Zeichenfolge übereinstimmen.

Wenn Sie beispielsweise überprüfen möchten, dass eine Zeichenfolge eine Ziffer von Null bis Neun hat, ein Trennzeichen, wie Bindestriche, Schrägstriche oder sogar Leerzeichen, ein Kleinbuchstabe, ein anderes Trennzeichen und dann eine andere Ziffer von Null bis Neun Verwenden Sie eine Regex wie folgt:

```
[0-9][-/ ][a-z][-/ ][0-9]
```

Dies würde `1-a-4` , aber *auch* `1-a/4` oder `1 a-4` . Wenn die Trennzeichen übereinstimmen sollen, können wir eine Erfassungsgruppe und eine Rückwärtsreferenz verwenden. Die Rückwärtsreferenz zeigt die Übereinstimmung der gefundenen Capture-Gruppe an und stellt sicher, dass die Position der Rückwärtsreferenz genau übereinstimmt.

Wenn wir dasselbe Beispiel verwenden, wird die Regex zu:

```
[0-9]([-/ ])[a-z]\1[0-9]
```

Das `\1` bezeichnet die erste Erfassungsgruppe im Muster. Bei dieser kleinen Änderung stimmt der Regex nun mit `1-a-4` oder `1 a 4` überein, jedoch nicht mit `1 a-4` oder `1-a/4` .

Die Nummer, die für die Rückreferenz verwendet werden soll, hängt vom Speicherort Ihrer Capture-Gruppe ab. Die Anzahl kann zwischen eins und neun liegen und kann durch Zählen der Erfassungsgruppen ermittelt werden.

```
([0-9])([-/ ])[a-z][-/ ]([0-9])
|--1--||--2--|           |--3--|
```

Verschachtelte Erfassungsgruppen ändern diese Anzahl geringfügig. Sie zählen zunächst die äußere Capture-Gruppe, dann die nächste Ebene und fahren fort, bis Sie das Nest verlassen:

```
(([0-9])([-/ ]))([a-z])
 |--2--||--3--|
|-------1------||--4--|
```

Mehrdeutige Rückmeldungen

Problem: Sie müssen Text eines bestimmten Formats zuordnen, zum Beispiel:

```
1-a-0
6/p/0
4 g 0
```

Das ist eine Ziffer, ein Trennzeichen (eins von `-` , `/` oder ein Leerzeichen), ein Buchstabe, dasselbe Trennzeichen und eine Null.

Naive Lösung: Wenn Sie den Regex aus dem Basics-Beispiel anpassen, kommen Sie zu diesem Regex:

```
[0-9]([-/ ])[a-z]\10
```

Das funktioniert aber wahrscheinlich nicht. Die meisten regulären Ausdrücke unterstützen mehr als neun Erfassungsgruppen, und nur wenige sind klug genug, um zu erkennen, dass `\10` , da es nur eine Erfassungsgruppe gibt, ein Rückverweis auf Gruppe 1 sein muss, gefolgt von einem Literal `0` . Die meisten Geschmacksrichtungen behandeln es als Rückverweis auf Gruppe 10. Einige davon lösen eine Ausnahme aus, da es keine Gruppe 10 gibt. der Rest wird einfach nicht passen.

Es gibt mehrere Möglichkeiten, dieses Problem zu vermeiden. Benutze Gruppen (und benannte Rückreferenzen):

```
[0-9](?<sep>[-/ ])[a-z]\k<sep>0
```

Wenn Ihre Regex-Sprache dies unterstützt, kann das Format `\g{n}` (wobei `n` eine Zahl ist) die Rückverweisnummer in geschweifte Klammern einschließen, um sie von den nachfolgenden Ziffern zu trennen:

```
[0-9]([-/ ])[a-z]\g{1}0
```

Eine andere Möglichkeit ist die Verwendung der erweiterten Regex-Formatierung, bei der die Elemente mit unbedeutendem Leerzeichen getrennt werden (in Java müssen Sie das Leerzeichen in den Klammern angeben):

```
(?x) [0-9] ([-/ ]) [a-z] \1 0
```

Wenn Ihr Regex-Aroma diese Funktionen nicht unterstützt, können Sie eine unnötige, aber harmlose Syntax hinzufügen, z. B. eine Gruppe ohne Capturing:

```
[0-9]([-/ ])[a-z](?:\1)0
```

... oder ein Dummy-Quantifizierer (dies ist möglicherweise der einzige Umstand, unter dem `{1}` nützlich ist):

```
[0-9]([-/ ])[a-z]\1{1}0
```

Rückverweis online lesen: https://riptutorial.com/de/regex/topic/4072/ruckverweis

Kapitel 22: Typen für reguläre Ausdrücke

Examples

NFA

Ein NFA-Motor (Nondeterministic Finite Automaton) wird *vom Muster gesteuert* .

Prinzip

Das Regex-Muster wird in einen Baum geparst.

Der *aktuelle Positionszeiger* wird auf den Anfang der Eingabezeichenfolge gesetzt, und es wird an dieser Position eine Übereinstimmung versucht. Wenn die Übereinstimmung fais ist, wird die Position auf das nächste Zeichen in der Zeichenfolge erhöht und von dieser Position aus wird eine weitere Übereinstimmung versucht. Dieser Vorgang wird wiederholt, bis eine Übereinstimmung gefunden wird oder das Ende der Eingabezeichenfolge erreicht ist.

Für jeden Spielversuch

Der Algorithmus arbeitet durch Durchlaufen des Musterbaums für eine gegebene Startposition. Beim Durchlaufen des Baums wird die *aktuelle Eingabeposition* aktualisiert, indem übereinstimmende Zeichen verwendet werden.

Wenn der Algorithmus auf einen Baumknoten trifft, der an der aktuellen Position nicht mit der Eingabezeichenfolge übereinstimmt, muss er *zurückverfolgt werden* . Dies geschieht, indem Sie zum übergeordneten Knoten in der Baumstruktur zurückkehren, die aktuelle Eingabeposition auf den Wert zurücksetzen, den sie bei der Eingabe des übergeordneten Knotens hatte, und den nächsten alternativen Zweig versuchen.

Wenn es dem Algorithmus gelingt, den Baum zu verlassen, meldet er eine erfolgreiche Übereinstimmung. Andernfalls schlägt das Match fehl, wenn alle Möglichkeiten ausprobiert wurden.

Optimierungen

Regex-Engines wenden normalerweise einige Optimierungen an, um die Leistung zu verbessern. Wenn sie zum Beispiel feststellen, dass eine Übereinstimmung mit einem bestimmten Zeichen beginnen muss, versuchen sie eine Übereinstimmung nur an den Positionen in der Eingabezeichenfolge, an denen das Zeichen erscheint.

Beispiel

`abeacab` `a(b|c)a` an den Eingabestring `abeacab` :

Der Musterbaum könnte ungefähr so aussehen:

```
CONCATENATION
    EXACT: a
    ALTERNATION
        EXACT: b
        EXACT: c
    EXACT: a
```

Der Spielvorgang läuft wie folgt ab:

```
a(b|c)a      abeacab
^            ^
```

`a` wird in der Eingabezeichenfolge gefunden, verbraucht sie und fährt mit dem nächsten Element im Musterbaum fort: der Änderung. Versuchen Sie die erste Möglichkeit: eine genaue `b` .

```
a(b|c)a      abeacab
  ^           ^
```

`b` gefunden wird, damit der Wechsel erfolgreich ist, verbrauchen Sie ihn und fahren Sie mit dem nächsten Element in der Verkettung fort: ein genaues `a` :

```
a(b|c)a      abeacab
      ^        ^
```

`a` wird *nicht* an der erwarteten Position gefunden. Kehren Sie zur Änderung zurück, setzen Sie die Eingabeposition auf den Wert zurück, den sie bei der ersten Eingabe der Änderung hatte, und versuchen Sie die *zweite* Alternative:

```
a(b|c)a      abeacab
    ^         ^
```

`c` wird an dieser Position *nicht* gefunden. Zurück zur Verkettung. An diesem Punkt gibt es keine weiteren Möglichkeiten, es gibt also keine Übereinstimmung am Anfang der Zeichenfolge.

Versuchen Sie ein zweites Spiel an der nächsten Eingabeposition:

```
a(b|c)a      abeacab
^             ^
```

`a` passt da *nicht* zusammen. Versuchen Sie ein anderes Spiel an der nächsten Position:

```
a(b|c)a      abeacab
```

```
^               ^
```

Auch kein Glück. Weiter zur nächsten Position.

```
a(b|c)a      abeacab
^               ^
```

`a` Streichholz, also verbrauchen Sie es und geben Sie die Alternative ein:

```
a(b|c)a      abeacab
  ^               ^
```

`b` passt nicht zusammen. Versuchen Sie die zweite Alternative:

```
a(b|c)a      abeacab
    ^               ^
```

`c` passt, also verbrauchen Sie es und fahren Sie mit dem nächsten Element in der Verkettung fort:

```
a(b|c)a      abeacab
      ^               ^
```

`a` stimmt überein und das Ende des Baums wurde erreicht. Bericht über ein erfolgreiches Spiel:

```
a(b|c)a      abeacab
               \_/
```

DFA

Eine DFA-Engine (Deterministic Finite Automaton) wird *von der Eingabe gesteuert* .

Prinzip

Der Algorithmus durchsucht die Eingabezeichenfolge *einmal* und speichert alle möglichen Pfade in der Regex, die übereinstimmen könnten. Wenn zum Beispiel im Muster eine Änderung auftritt, werden zwei neue Pfade erstellt und unabhängig voneinander versucht. Wenn ein bestimmter Pfad nicht übereinstimmt, wird er aus den festgelegten Möglichkeiten entfernt.

Implikationen

Die Übereinstimmungszeit ist durch die Größe des Eingabestrings begrenzt. Es gibt kein Backtracking, und die Engine kann mehrere Übereinstimmungen gleichzeitig finden, sogar überlappende Übereinstimmungen.

Der Hauptnachteil dieser Methode ist der reduzierte Funktionsumfang, der von der Engine im

Vergleich zum NFA-Engine-Typ unterstützt werden kann.

Beispiel

Match `a(b|c)a` gegen `abadaca` :

```
abadaca      a(b|c)a
^            ^       Attempt 1      ==> CONTINUE

abadaca      a(b|c)a
 ^           ^       Attempt 2      ==> FAIL
               ^     Attempt 1.1    ==> CONTINUE
                 ^   Attempt 1.2    ==> FAIL

abadaca      a(b|c)a
  ^          ^       Attempt 3      ==> CONTINUE
                 ^   Attempt 1.1    ==> MATCH

abadaca      a(b|c)a
   ^         ^       Attempt 4      ==> FAIL
               ^     Attempt 3.1    ==> FAIL
                 ^   Attempt 3.2    ==> FAIL

abadaca      a(b|c)a
    ^        ^       Attempt 5      ==> CONTINUE

abadaca      a(b|c)a
     ^       ^       Attempt 6      ==> FAIL
               ^     Attempt 5.1    ==> FAIL
                 ^   Attempt 5.2    ==> CONTINUE

abadaca      a(b|c)a
      ^      ^       Attempt 7      ==> CONTINUE
                 ^   Attempt 5.2    ==> MATCH

abadaca      a(b|c)a
       ^       ^     Attempt 7.1    ==> FAIL
                 ^   Attempt 7.2    ==> FAIL
```

Typen für reguläre Ausdrücke online lesen: https://riptutorial.com/de/regex/topic/2861/typen-fur-regulare-ausdrucke

Kapitel 23: UTF-8-Matcher: Buchstaben, Marken, Interpunktion usw.

Examples

Übereinstimmende Buchstaben in verschiedenen Alphabeten

Die nachstehenden Beispiele sind in Ruby angegeben, jedoch sollten in jeder modernen Sprache dieselben Matcher verfügbar sein.

Nehmen wir an, wir haben die Zeichenfolge `"Aஃ Naïve"` , produziert von Messy Artificial Intelligence. Es besteht aus Buchstaben, aber der generische `\w` Matcher passt nicht besonders gut zusammen:

```
▶ "Aஃ Naïve"[/\w+/]
#⇒ "A"
```

Der richtige Weg, Unicode-Buchstaben mit Kombinationsmarken abzugleichen, ist die Verwendung von `\X` zur Angabe eines Graphem-Clusters. Es gibt jedoch einen Vorbehalt für Ruby. Onigmo, die Regex-Engine für Ruby, verwendet immer noch die alte Definition eines Graphem-Clusters . Es ist noch nicht auf Extended Grapheme Cluster gemäß der Definition in Unicode Standard Anhang 29 aktualisiert.

Für Ruby könnte es also zu einer Problemumgehung kommen: `\p{L}` wird fast alles gut machen, es sei denn, der kombinierte diakritische Akzent bei `i` schlägt fehl

```
▶ "Aஃ Naïve"[/\p{L}+/]
#⇒ "Aஃ Nai"
```

Durch das Hinzufügen der Markierungssymbole zum Ausdruck können wir schließlich alles abgleichen:

```
▶ "Aஃ Naïve"[/[\p{L}\p{M}]+/]
#⇒ "Aஃ Naïve"
```

UTF-8-Matcher: Buchstaben, Marken, Interpunktion usw. online lesen: https://riptutorial.com/de/regex/topic/1527/utf-8-matcher--buchstaben--marken--interpunktion-usw-

Kapitel 24: Wann sollten Sie KEINE regulären Ausdrücke verwenden?

Bemerkungen

Da reguläre Ausdrücke entweder auf eine reguläre Grammatik oder auf eine kontextfreie Grammatik beschränkt sind, kommt es häufig zu Missbrauch von regulären Ausdrücken. In diesem Thema gibt es einige Beispiele, in denen Sie *KEINE* regulären Ausdrücke verwenden sollten, sondern stattdessen Ihre bevorzugte Sprache.

> *Einige Leute denken, wenn sie mit einem Problem konfrontiert sind:*
> *"Ich weiß, ich werde reguläre Ausdrücke verwenden."*
> *Jetzt haben sie zwei Probleme.*
> - Jamie Zawinski

Examples

Passende Paare (wie Klammern, Klammern ...)

Einige reguläre Ausdrücke (z. B. .NET) können kontextfreie Ausdrücke verarbeiten und klären dies. Dies ist jedoch bei den meisten Standardmotoren nicht der Fall. Und selbst wenn dies der Fall ist, haben Sie einen komplexen, schwer lesbaren Ausdruck, während die Verwendung einer Parsing-Bibliothek die Arbeit erleichtern kann.

- Wie finde ich alle möglichen Regex-Übereinstimmungen in Python?

Einfache Stringoperationen

Da *reguläre Ausdrücke* viel *bewirken* können, ist es verlockend, sie für die einfachsten Operationen zu verwenden. Die Verwendung einer Regex-Engine verursacht jedoch Kosten in Bezug auf Arbeitsspeicher und Prozessor: Sie müssen den Ausdruck kompilieren, den Automaten im Speicher ablegen, ihn initialisieren und ihn dann mit der Zeichenfolge ausführen, um ihn auszuführen.

Und es gibt viele Fälle, in denen es nicht nötig ist, es zu benutzen! Was auch immer Ihre Sprache ist, es hat immer die grundlegenden Werkzeuge zur Bearbeitung von Strings. Wenn also in Ihrer Standardbibliothek ein Werkzeug zum Ausführen einer Aktion vorhanden ist, verwenden Sie dieses Werkzeug und nicht einen regulären Ausdruck:

- eine Zeichenfolge teilen?

Zum Beispiel funktioniert das folgende Snippet in Python, Ruby und Javascript:

```
'foo.bar'.split('.')
```

Welches ist einfacher zu lesen und zu verstehen sowie viel effizienter als der (irgendwie) äquivalente reguläre Ausdruck:

```
(\w+)\.(\w+)
```

- Leerzeichen entfernen?

Gleiches gilt für nachfolgende Leerzeichen!

```
'foobar     '.strip() # python or ruby
'foobar     '.trim() // javascript
```

Welches wäre dem folgenden Ausdruck gleichwertig:

```
([^\n]*)\s*$ # keeping \1 in the substitution
```

Analysieren von HTML (oder XML oder JSON oder C-Code oder…)

Wenn Sie etwas von einer Webseite (oder einer Repräsentations- / Programmiersprache) extrahieren möchten, ist ein regulärer Ausdruck das falsche Werkzeug für die Aufgabe. Verwenden Sie stattdessen die Bibliotheken Ihrer Sprache, um die Aufgabe zu lösen.

Wenn Sie HTML, XML oder JSON lesen möchten, verwenden Sie einfach die Bibliothek, die es richtig analysiert und als verwendbare Objekte in Ihrer bevorzugten Sprache dient! Am Ende erhalten Sie lesbaren und wartungsfreundlichen Code, und Sie werden nicht enden

- RegEx stimmen mit offenen Tags mit Ausnahme von eigenständigen XHTML-Tags überein
- HTML-Python-Analyse mit regulären Ausdrücken
- Gibt es einen Regex, um alle ganzen Zahlen für eine bestimmte Programmiersprache zu generieren

Wann sollten Sie KEINE regulären Ausdrücke verwenden? online lesen: https://riptutorial.com/de/regex/topic/4527/wann-sollten-sie-keine-regularen-ausdrucke-verwenden-

Kapitel 25: Wortgrenze

Syntax

- POSIX-Stil, Wortende: `[[:>:]]`
- POSIX-Stil, Wortanfang: `[[:<:]]`
- POSIX-Stil, Wortgrenze: `[[:<:][:>:]]`
- SVR4 / GNU, Wortende: `\>`
- SVR4 / GNU, Wortanfang: `\<`
- Perl / GNU, Wortgrenze: `\b`
- Tcl, Wortende: `\M`
- Tcl, Wortanfang: `\m`
- Tcl, Wortgrenze: `\y`
- Portable ERE, Wortanfang: `(^|[^[:alnum:]_])`
- Portable ERE, Wortende: `([^[:alnum:]_]|$)`

Bemerkungen

Zusätzliche Ressourcen

- POSIX-Kapitel zu regulären Ausdrücken
- Dokumentation des regulären Ausdrucks von Perl
- Tcl re_syntax-Handbuchseite
- GNU grep-Backslash-Ausdrücke
- BSD re_format
- Mehr lesen

Examples

Vollkommenes Wort

```
\bfoo\b
```

entspricht dem vollständigen Wort ohne alphanumerische Zeichen und _ vor oder nach ihm.

Einnahme von regularexpression.info

> Es gibt drei verschiedene Positionen, die sich als Wortgrenzen eignen:
>
> 1. Vor dem ersten Zeichen in der Zeichenfolge, wenn das erste Zeichen ein Wortzeichen ist.
> 2. Nach dem letzten Zeichen in der Zeichenfolge, wenn das letzte Zeichen ein Wortzeichen ist.
> 3. Zwischen zwei Zeichen in der Zeichenfolge, wobei eines ein Wortzeichen und

das andere kein Wortzeichen ist.

Der Begriff " *Wort"* bedeutet hier Folgendes

1. Alphabet (`[a-zA-Z]`)
2. Nummer (`[0-9]`)
3. Unterstrich _

Kurz gesagt, *Wortzeichen* = `\w` = `[a-zA-Z0-9_]`

Finden Sie Muster am Anfang oder Ende eines Wortes

Untersuchen Sie die folgenden Zeichenfolgen:

```
foobarfoo
bar
foobar
barfoo
```

- Die `bar` für reguläre Ausdrücke `bar` zu allen vier Zeichenfolgen.
- `\bbar\b` nur mit dem zweiten überein.
- `bar\b` kann die 2. und 3. Saite und
- `\bbar` der 2. und 4. Saite.

Wortgrenzen

Das `\b` Metazeichen

Um das Auffinden ganzer Wörter zu erleichtern, können wir das Metazeichen `\b` . Es markiert den **Anfang** *und* das **Ende** einer alphanumerischen Sequenz *. Da es nur dazu dient, diese Orte zu markieren, stimmt es eigentlich mit keinem Zeichen überein.

**: Es ist üblich, eine alphanumerische Folge ein Wort zu nennen, da wir die Zeichen mit einem `\w` (der Wortzeichenklasse) fangen können. Dies kann jedoch irreführend sein, da `\w` auch Zahlen und in den meisten Geschmacksrichtungen den Unterstrich enthält.*

Beispiele:

Regex	Eingang	Streichhölzer?
`\bstack\b`	`stackoverflow`	**Nein,** da gibt es keine ocurrence des **ganzen** Wortes `stack`
`\bstack\b`	`foo stack bar`	**Ja** , da gibt es weder vor noch nach dem `stack`
`\bstack\b`	`stack!overflow`	**Ja** : vor dem `stack` gibt es nichts und `!` ist kein Wortzeichen
`\bstack`	`stackoverflow`	**Ja** , da ist nichts vor dem `stack`

Regex	Eingang	Streichhölzer?
`overflow\b`	`stackoverflow`	**Ja** , da gibt es nach `overflow` nichts

Das `\B` Metazeichen

Dies ist das Gegenteil von `\b` , das mit der Position jedes Nicht-Begrenzungszeichens übereinstimmt. Wie `\b` , da er mit Orten übereinstimmt, keinem eigenen Zeichen. Es ist nützlich, um *nicht* ganze Wörter zu finden.

Beispiele:

Regex	Eingang	Streichhölzer?
`\Bb\B`	`abc`	**Ja** , da `b` nicht von Wortgrenzen umgeben ist.
`\Ba\B`	`abc`	**Nein** , `a` hat eine Wortgrenze auf der linken Seite.
`a\B`	`abc`	**Ja** , `a` hat keine Wortgrenze auf der rechten Seite.
`\B,\B`	`a,,,b`	**Ja** , es stimmt mit dem zweiten Komma überein, da `\B` auch mit dem Leerzeichen zwischen zwei Nicht-Wort-Zeichen übereinstimmt .

Text kürzer machen, aber das letzte Wort nicht brechen

Um langen Text maximal N Zeichen lang zu machen, aber das letzte Wort intakt zu lassen, verwenden Sie `.{0,N}\b`

```
^(.{0,N})\b.*
```

Wortgrenze online lesen: https://riptutorial.com/de/regex/topic/1539/wortgrenze

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit regulären Ausdrücken	0rkan, Addison, balpha, Community, Confiqure, Ibrahim, J F, JelmerS, JohnLBevan, Kendra, Laurel, Maria Deleva, Mariano, Mateus, mnoronha, Rudy M, Stephen Leppik, Tot Zam, TylerH, Wolf, Yaron, zmo
2	Ankerzeichen: Caret (^)	CPHPython, Eder, J F, JohnLBevan, Jojodmo, knut, Mateus, Mike H-R, Mr. Deathless, nhahtdh, revo, rgoliveira, Tom Lord, zb226
3	Ankerzeichen: Dollar ($)	ArtOfCode, CPHPython, hjpotter92, Kendra, rubayet.R, Tom Lord, UNagaswamy, Wiktor Stribiżew
4	Atomische Gruppierung	OnlineCop
5	Benannte Fanggruppen	Thomas Ayoub
6	Charakterklassen	Acey, CPHPython, Dmitry Bychenko, HamZa, kdhp, Lucas Trzesniewski, Maria Deleva, RamenChef, rgoliveira, rock321987, Wiktor Stribiżew
7	Einfache Muster abgleichen	balpha, GradientByte, Graham, Joe, Mariano, rgoliveira, Tot Zam, Yaron
8	Ersetzungen mit regulären Ausdrücken	Mateus
9	Flucht	CPHPython, David Knipe, Laurel
10	Gierige und faule Quantifizierer	0rkan, Confiqure, David Knipe, GradientByte, Laurel, Mario, Mark Stewart, Nathan Arthur, nhahtdh, phatfingers, sweaver2112, Thomas Ayoub, Tim Pietzcker
11	Gruppen erfassen	Addison, Alan Moore, Lucas Trzesniewski, Tomalak, Vogel612
12	Lookahead und Lookbehind	BoppreH, hwnd, Lucas Trzesniewski, Maria Deleva, Wiktor Stribiżew
13	Match zurücksetzen: \ K	nhahtdh, Wiktor Stribiżew, Will Barnwell

14	Nützlicher Regex Showcase	depperm, Devid Farinelli, Echelon, Herb, Kendra, Matas Vaitkevicius, nhahtdh, Sebastian Lenartowicz, Steve Chambers, Thomas Ayoub, Tomasz Jakub Rup, zmo
15	Possessive Quantifizierer	Mark Hurd, Sebastian Lenartowicz
16	Regex Fallstricke	BrightOne
17	Regex für die Passwortüberprüfung	rock321987
18	Regex-Modifikatoren (Flags)	Eder, Mateus, Tim Pietzcker, Wiktor Stribiżew
19	Rekursion	Keith Hall, Laurel, Lucas Trzesniewski, user23013
20	Rückverfolgung	dorukayhan, Mike, Miljen Mikic, SQB, Thomas Ayoub, Vituel
21	Rückverweis	Alan Moore, Kendra, OnlineCop
22	Typen für reguläre Ausdrücke	Lucas Trzesniewski, Markus Jarderot
23	UTF-8-Matcher: Buchstaben, Marken, Interpunktion usw.	mudasobwa
24	Wann sollten Sie KEINE regulären Ausdrücke verwenden?	dorukayhan, Kendra, zmo
25	Wortgrenze	cdm, jonathanking, kdhp, Maria Deleva, Peter G, rgoliveira, Tushar