Reguläre Ausdrücke

regex

Michael Dienert

Walther-Rathenau-Gewerbeschule Freiburg

19. März 2019

Inhalt

Regulären Ausdrücke

- Mit regulären Ausdrücken lassen sich Texte nach bestimmten *Mustern* durchsuchen.
- Der reguläre Ausdruck ist selbst eine Zeichenkette, die einer bestimmten Syntax genügen muss.
- Mit einem reguläre Ausdruck kann genau eine bestimmte oder auch eine Menge von Suchmustern beschrieben werden.
- Beispiel: **.\?aus** (oder auch **.?aus**, bei extended-regex s.u.) passt auf 'maus', 'haus', 'Haus', 'raus', aber nicht 'klaus'.
- Die meisten Editoren, Programmiersprachen und viele spezielle Dienstprogramme können reguläre Ausdrücke auswerten.
- Wichtigstes Kommando zum Absuchen von Textdateien: `grep`

Regulären Ausdrücke

- Mit regulären Ausdrücken lassen sich Texte nach bestimmten *Mustern* durchsuchen.
- Der reguläre Ausdruck ist selbst eine Zeichenkette, die einer bestimmten Syntax genügen muss.
- Mit einem reguläre Ausdruck kann genau eine bestimmte oder auch eine Menge von Suchmustern beschrieben werden.
- Beispiel: **.\?aus** (oder auch **.?aus**, bei extended-regex s.u.) passt auf 'maus', 'haus', 'Haus', 'raus', aber nicht 'klaus'.
- Die meisten Editoren, Programmiersprachen und viele spezielle Dienstprogramme können reguläre Ausdrücke auswerten.
- Wichtigstes Kommando zum Absuchen von Textdateien: `grep`

Regulären Ausdrücke

- Mit regulären Ausdrücken lassen sich Texte nach bestimmten *Mustern* durchsuchen.
- Der reguläre Ausdruck ist selbst eine Zeichenkette, die einer bestimmten Syntax genügen muss.
- Mit einem reguläre Ausdruck kann genau eine bestimmte oder auch eine Menge von Suchmustern beschrieben werden.
- Beispiel: **.\?aus** (oder auch **.?aus**, bei extended-regex s.u.) passt auf 'maus', 'haus', 'Haus', 'raus', aber nicht 'klaus'.
- Die meisten Editoren, Programmiersprachen und viele spezielle Dienstprogramme können reguläre Ausdrücke auswerten.
- Wichtigstes Kommando zum Absuchen von Textdateien: `grep`

Regulären Ausdrücke

- Mit regulären Ausdrücken lassen sich Texte nach bestimmten *Mustern* durchsuchen.
- Der reguläre Ausdruck ist selbst eine Zeichenkette, die einer bestimmten Syntax genügen muss.
- Mit einem reguläre Ausdruck kann genau eine bestimmte oder auch eine Menge von Suchmustern beschrieben werden.
- Beispiel: **.\?aus** (oder auch **.?aus**, bei extended-regex s.u.) passt auf 'maus', 'haus', 'Haus', 'raus', aber nicht 'klaus'.
- Die meisten Editoren, Programmiersprachen und viele spezielle Dienstprogramme können reguläre Ausdrücke auswerten.
- Wichtigstes Kommando zum Absuchen von Textdateien: `grep`

Regulären Ausdrücke

- Mit regulären Ausdrücken lassen sich Texte nach bestimmten *Mustern* durchsuchen.
- Der reguläre Ausdruck ist selbst eine Zeichenkette, die einer bestimmten Syntax genügen muss.
- Mit einem reguläre Ausdruck kann genau eine bestimmte oder auch eine Menge von Suchmustern beschrieben werden.
- Beispiel: **.\?aus** (oder auch **.?aus**, bei extended-regex s.u.) passt auf 'maus', 'haus', 'Haus', 'raus', aber nicht 'klaus'.
- Die meisten Editoren, Programmiersprachen und viele spezielle Dienstprogramme können reguläre Ausdrücke auswerten.
- Wichtigstes Kommando zum Absuchen von Textdateien: `grep`

Regulären Ausdrücke

- Mit regulären Ausdrücken lassen sich Texte nach bestimmten *Mustern* durchsuchen.
- Der reguläre Ausdruck ist selbst eine Zeichenkette, die einer bestimmten Syntax genügen muss.
- Mit einem reguläre Ausdruck kann genau eine bestimmte oder auch eine Menge von Suchmustern beschrieben werden.
- Beispiel: **.\?aus** (oder auch **.?aus**, bei extended-regex s.u.) passt auf 'maus', 'haus', 'Haus', 'raus', aber nicht 'klaus'.
- Die meisten Editoren, Programmiersprachen und viele spezielle Dienstprogramme können reguläre Ausdrücke auswerten.
- Wichtigstes Kommando zum Absuchen von Textdateien: `grep`

Regulären Ausdrücke

- Mit regulären Ausdrücken lassen sich Texte nach bestimmten *Mustern* durchsuchen.
- Der reguläre Ausdruck ist selbst eine Zeichenkette, die einer bestimmten Syntax genügen muss.
- Mit einem reguläre Ausdruck kann genau eine bestimmte oder auch eine Menge von Suchmustern beschrieben werden.
- Beispiel: **.\?aus** (oder auch **.?aus**, bei extended-regex s.u.) passt auf 'maus', 'haus', 'Haus', 'raus', aber nicht 'klaus'.
- Die meisten Editoren, Programmiersprachen und viele spezielle Dienstprogramme können reguläre Ausdrücke auswerten.
- Wichtigstes Kommando zum Absuchen von Textdateien: `grep`

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Syntax von regulären Ausdrücken

- Es gibt mehrere Standards für die Syntax von regulären Ausdrücken. Die beiden wichtigsten sind der **Perl**- und der **POSIX**-Standard.
- 1992 hat man den POSIX.2-Standard für reguläre Ausdrücke festgelegt.
- Der POSIX.2-Standard zerfällt wiederum in zwei Teile:
  - BRE: Basic Regular Expression
  - ERE: Extended Regular Expression
- zusammen mit der Programmiersprache `Perl` hat sich seit 1980 der Perl-Regex-Standard entwickelt. Perl-Regexes sind mächtiger und leichter zu lesen als die POSIX-Ausdrücke.
- Perl-Regexes werden als *PCRE - Perl Compatible* Regular Expressions bezeichnet.
- Java verwendet die PCRE (mit ein paar Ausnahmeregelungen).

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Literale und Metazeichen

- Ein regulärer Ausdruck ist, wie schon erwähnt, eine Zeichenkette (String), die *Pattern* genannt wird.
- Patterns bestehen aus zwei Arten von Zeichen:
  - Normale Textzeichen (Literale).
  - Metazeichen (Metazeichen). Metazeichen haben eine spezielle Bedeutung, die auch vom verwendeten Standard abhängt.
- Mit vorangestellten \-Zeichen kann man die Sonderbedeutung eines Metazeichens *ausschalten* und die Metazeichen in Literale umwandeln.
- Z.B. sind die runden Klammern () Metazeichens. Mit \(kann man in einer Zeichenkette nach einer (-Klammer suchen.
- Da das \-Zeichen in Java selbst ein Sonderzeichen ist, muss man ihm in einem Java-Regex-Pattern selbst ein \-Zeichen voranstellen. Alle \-Zeichen im Regex werden in Java **verdoppelt**!

Inhalt

Zeichengruppen

[abc]	genau *eines* der angegebenen Zeichen.
[a-z]	genau *eines* der angegebenen Zeichen, diesmal mit *Bereichsausdruck*.
[a-fu-z]	genau *eines* der angegebenen Zeichen
[^abc]	genau *ein* Zeichen, das *nicht* 'a', 'b' oder 'c' ist
.	ein beliebiges Zeichen ausser *Newline*.

Zeichengruppen

[abc]	genau *eines* der angegebenen Zeichen.
[a-z]	genau *eines* der angegebenen Zeichen, diesmal mit *Bereichsausdruck*.
[a-fu-z]	genau *eines* der angegebenen Zeichen
[^abc]	genau *ein* Zeichen, das *nicht* 'a', 'b' oder 'c' ist
.	ein beliebiges Zeichen ausser *Newline*.

Zeichengruppen

[abc]	genau *eines* der angegebenen Zeichen.
[a-z]	genau *eines* der angegebenen Zeichen, diesmal mit *Bereichsausdruck*.
[a-fu-z]	genau *eines* der angegebenen Zeichen
[^abc]	genau *ein* Zeichen, das *nicht* 'a', 'b' oder 'c' ist
.	ein beliebiges Zeichen ausser *Newline*.

Zeichengruppen

[abc]	genau *eines* der angegebenen Zeichen.
[a-z]	genau *eines* der angegebenen Zeichen, diesmal mit *Bereichsausdruck*.
[a-fu-z]	genau *eines* der angegebenen Zeichen
[^abc]	genau *ein* Zeichen, das *nicht* 'a', 'b' oder 'c' ist
.	ein beliebiges Zeichen ausser *Newline*.

Zeichengruppen

[abc]	genau *eines* der angegebenen Zeichen.
[a-z]	genau *eines* der angegebenen Zeichen, diesmal mit *Bereichsausdruck*.
[a-fu-z]	genau *eines* der angegebenen Zeichen
[^abc]	genau *ein* Zeichen, das *nicht* 'a', 'b' oder 'c' ist
.	ein beliebiges Zeichen ausser *Newline*.

Zeichengruppen

[abc]	genau *eines* der angegebenen Zeichen.
[a-z]	genau *eines* der angegebenen Zeichen, diesmal mit *Bereichsausdruck*.
[a-fu-z]	genau *eines* der angegebenen Zeichen
[^abc]	genau *ein* Zeichen, das *nicht* 'a', 'b' oder 'c' ist
.	ein beliebiges Zeichen ausser *Newline*.

Positionsanker

^ passt auf die leere Zeichenkette am Anfang einer Zeile. Das ^-Zeichen ist nur ein Metazeichen, wenn es am Anfang eines Regex steht.

$ passt auf die leere Zeichenkette am Ende einer Zeile. Das $-Zeichen ist nur ein Metazeichen, wenn es am Ende eines Regex steht.

\< passt auf die leere Zeichenkette am Anfang eines Worts.

\> passt auf die leere Zeichenkette am Ende eines Worts.

Beispiel: alle Wörter *aus* oder *Aus* suchen: \<[a|A]us\>

Alternativen: \b passt auf Anfang oder Ende; \s passt auf *whitespace*

Positionsanker

^ passt auf die leere Zeichenkette am Anfang einer Zeile. Das ^-Zeichen ist nur ein Metazeichen, wenn es am Anfang eines Regex steht.

$ passt auf die leere Zeichenkette am Ende einer Zeile. Das $-Zeichen ist nur ein Metazeichen, wenn es am Ende eines Regex steht.

\< passt auf die leere Zeichenkette am Anfang eines Worts.

\> passt auf die leere Zeichenkette am Ende eines Worts.

Beispiel: alle Wörter *aus* oder *Aus* suchen: \<[a|A]us\>

Alternativen: \b passt auf Anfang oder Ende; \s passt auf *whitespace*

Positionsanker

^ passt auf die leere Zeichenkette am Anfang einer Zeile. Das ^-Zeichen ist nur ein Metazeichen, wenn es am Anfang eines Regex steht.

$ passt auf die leere Zeichenkette am Ende einer Zeile. Das $-Zeichen ist nur ein Metazeichen, wenn es am Ende eines Regex steht.

\< passt auf die leere Zeichenkette am Anfang eines Worts.

\> passt auf die leere Zeichenkette am Ende eines Worts.

Beispiel: alle Wörter *aus* oder *Aus* suchen: \<[a|A]us\>

Alternativen: \b passt auf Anfang oder Ende; \s passt auf *whitespace*

Positionsanker

^ passt auf die leere Zeichenkette am Anfang einer Zeile. Das ^-Zeichen ist nur ein Metazeichen, wenn es am Anfang eines Regex steht.

$ passt auf die leere Zeichenkette am Ende einer Zeile. Das $-Zeichen ist nur ein Metazeichen, wenn es am Ende eines Regex steht.

\< passt auf die leere Zeichenkette am Anfang eines Worts.

\> passt auf die leere Zeichenkette am Ende eines Worts.

Beispiel: alle Wörter *aus* oder *Aus* suchen: \<[a|A]us\>

Alternativen: \b passt auf Anfang oder Ende; \s passt auf *whitespace*

Positionsanker

- ^ passt auf die leere Zeichenkette am Anfang einer Zeile. Das ^-Zeichen ist nur ein Metazeichen, wenn es am Anfang eines Regex steht.
- $ passt auf die leere Zeichenkette am Ende einer Zeile. Das $-Zeichen ist nur ein Metazeichen, wenn es am Ende eines Regex steht.
- \< passt auf die leere Zeichenkette am Anfang eines Worts.
- \> passt auf die leere Zeichenkette am Ende eines Worts.

Beispiel: alle Wörter *aus* oder *Aus* suchen: \<[a|A]us\>

Alternativen: \b passt auf Anfang oder Ende; \s passt auf *whitespace*

Positionsanker

^ passt auf die leere Zeichenkette am Anfang einer Zeile. Das ^-Zeichen ist nur ein Metazeichen, wenn es am Anfang eines Regex steht.

$ passt auf die leere Zeichenkette am Ende einer Zeile. Das $-Zeichen ist nur ein Metazeichen, wenn es am Ende eines Regex steht.

\< passt auf die leere Zeichenkette am Anfang eines Worts.

\> passt auf die leere Zeichenkette am Ende eines Worts.

Beispiel: alle Wörter *aus* oder *Aus* suchen: \<[a|A]us\>

Alternativen: \b passt auf Anfang oder Ende; \s passt auf *whitespace*

Positionsanker

^ passt auf die leere Zeichenkette am Anfang einer Zeile. Das ^-Zeichen ist nur ein Metazeichen, wenn es am Anfang eines Regex steht.

$ passt auf die leere Zeichenkette am Ende einer Zeile. Das $-Zeichen ist nur ein Metazeichen, wenn es am Ende eines Regex steht.

\< passt auf die leere Zeichenkette am Anfang eines Worts.

\> passt auf die leere Zeichenkette am Ende eines Worts.

Beispiel: alle Wörter *aus* oder *Aus* suchen: \<[a|A]us\>

Alternativen: \b passt auf Anfang oder Ende; \s passt auf *whitespace*

Gruppierung und Multiplizität

?	Das vorangestellte Objekt tritt 0 oder 1 mal auf.
*	Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
+	Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
{n}	Das vorangestellte Objekt tritt genau n mal auf.
{n,}	Das vorangestellte Objekt tritt mindestens n mal auf.
{n,m}	Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
|	Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Multiplizität

- `?` Das vorangestellte Objekt tritt 0 oder 1 mal auf.
- `*` Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
- `+` Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
- `{n}` Das vorangestellte Objekt tritt genau n mal auf.
- `{n,}` Das vorangestellte Objekt tritt mindestens n mal auf.
- `{n,m}` Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
- `|` Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Multiplizität

- ? Das vorangestellte Objekt tritt 0 oder 1 mal auf.
- * Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
- + Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
- {n} Das vorangestellte Objekt tritt genau n mal auf.
- {n,} Das vorangestellte Objekt tritt mindestens n mal auf.
- {n,m} Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
- | Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Multiplizität

- ? Das vorangestellte Objekt tritt 0 oder 1 mal auf.
- * Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
- + Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
- {n} Das vorangestellte Objekt tritt genau n mal auf.
- {n,} Das vorangestellte Objekt tritt mindestens n mal auf.
- {n,m} Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
- | Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Multiplizität

- `?` Das vorangestellte Objekt tritt 0 oder 1 mal auf.
- `*` Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
- `+` Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
- `{n}` Das vorangestellte Objekt tritt genau n mal auf.
- `{n,}` Das vorangestellte Objekt tritt mindestens n mal auf.
- `{n,m}` Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
- `|` Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Multiplizität

- ? Das vorangestellte Objekt tritt 0 oder 1 mal auf.
- * Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
- + Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
- {n} Das vorangestellte Objekt tritt genau n mal auf.
- {n,} Das vorangestellte Objekt tritt mindestens n mal auf.
- {n,m} Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
- | Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Multiplizität

?	Das vorangestellte Objekt tritt 0 oder 1 mal auf.
*	Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
+	Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
{n}	Das vorangestellte Objekt tritt genau n mal auf.
{n,}	Das vorangestellte Objekt tritt mindestens n mal auf.
{n,m}	Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
|	Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Multiplizität

?	Das vorangestellte Objekt tritt 0 oder 1 mal auf.
*	Das vorangestellte Objekt tritt 0 mal oder beliebig oft auf (0,1,2,..n).
+	Das vorangestellte Objekt tritt mindestens 1 mal auf (1,2,..n).
{n}	Das vorangestellte Objekt tritt genau n mal auf.
{n,}	Das vorangestellte Objekt tritt mindestens n mal auf.
{n,m}	Das vorangestellte Objekt tritt mindestens n und höchstens m mal auf.
|	Passt auf Regex vor oder nach dem |-Zeichen (Ex-OR).

Gruppierung und Rückreferenz

(regex) Passt auf *regex*, jedoch kann *regex* als ganzes modifiziert werden (z.B. mit den Multiplizitätsoperatoren).

\n Rückreferenz auf die vorangehende n-te Gruppe. n muss eine einzelne Ziffer sein (1 - 9).

Gruppierung und Rückreferenz

(regex) Passt auf *regex*, jedoch kann *regex* als ganzes modifiziert werden (z.B. mit den Multiplizitätsoperatoren).

\n Rückreferenz auf die vorangehende n-te Gruppe. n muss eine einzelne Ziffer sein (1 - 9).

Gruppierung und Rückreferenz

(regex) Passt auf *regex*, jedoch kann *regex* als ganzes modifiziert werden (z.B. mit den Multiplizitätsoperatoren).

\n Rückreferenz auf die vorangehende n-te Gruppe. n muss eine einzelne Ziffer sein (1 - 9).

Character-Classes

\w	Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)
\W	**keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)
\d	Ziffer (Digit)
\D	**keine** Ziffer
\s	Whitespace
\S	**kein** Whitespace
\d	Anfang **oder** Ende
\D	nicht Anfang und nicht Ende
\n	Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)

\W **keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)

\d Ziffer (Digit)

\D **keine** Ziffer

\s Whitespace

\S **kein** Whitespace

\d Anfang **oder** Ende

\D nicht Anfang und nicht Ende

\n Neue Zeile

Character-Classes

\w	Gross- und Kleinbuchstaben, Ziffern und Unterstrich (Wort)
\W	**keine** Gross- und Kleinbuchstaben, Ziffern, Unterstrich (Wort)
\d	Ziffer (Digit)
\D	**keine** Ziffer
\s	Whitespace
\S	**kein** Whitespace
\d	Anfang **oder** Ende
\D	nicht Anfang und nicht Ende
\n	Neue Zeile