

6.2 REGULÄRE AUSDRÜCKE

Ein *regulärer Ausdruck* ist eine Art *Muster*, das mit einem String verglichen wird. Mit regulären Ausdrücken kann man

- ◆ prüfen, ob ein String einem gegebenen Muster korrespondiert („matched“)
- ◆ aus einem String alle Abschnitte extrahieren, die dem Muster partiell oder vollständig entsprechen
- ◆ den String gezielt verändern, indem man die dem Muster entsprechenden Abschnitte durch andere ersetzt.

Es gibt verschiedene Möglichkeiten, reguläre Ausdrücke zu bezeichnen. Am weitesten verbreitet ist es, einen regulären Ausdruck durch *Slashes* zu begrenzen:

`/<Ausdruck>/`

Der Ausdruck kann prinzipiell beliebige Zeichen enthalten. Allerdings haben folgende Zeichen eine besondere Bedeutung:

`. | () [] { } + \ ^ $ * ?`

Die übrigen Zeichen *matchen*/entsprechen sich selbst.

6.2 REGULÄRE AUSDRÜCKE

```
str = "Dog and Cat"
new_str = str.sub(/Cat/, "Gerbil")
puts "Let's go to the #{new_str} for a pint."
```

Let's go to the Dog and Gerbil for a pint.

```
str = "Dog and Cat"
new_str1 = str.sub(/a/, "")
new_str2 = str.gsub(/a/, "")
puts "Using sub: #{new_str1}"
puts "Using gsub: #{new_str2}"
```

*Using sub: Dog *nd Cat*

*Using gsub: Dog *nd C*t*

```
str = "now is the time"
str.sub!(/i/, "")
str.gsub!(/t/, "T")
puts str
```

*now *s The Time*

6.2 REGULÄRE AUSDRÜCKE

Erweiterte Möglichkeiten

Da in Ruby reguläre Ausdrücke (wie alles andere auch) einfach Objekte sind (Instanzen der Klasse **Regexp**), können sie u.a. Variablen zugewiesen werden und als Parameter von Methoden verwendet werden.

```
str = "dog and cat"
pattern = /nd/
pattern =~ str           # => 5
str =~ pattern           # => 5

/mm\dd/                  # => /mm\dd/
Regexp.new("mm/dd")      # => /mm\dd/
%r{mm/dd}                 # => /mm\dd/
```

Das Verhalten von regulären Ausdrücken kann durch folgende vier Optionen modifiziert werden:

- | | | | |
|----------|----------------------------------|----------|-----------------------|
| i | <ignoriert Groß/Kleinschreibung> | o | <einmalige Ersetzung> |
| m | <Mehrzeilenmodus> | x | <erweiterter Modus> |

Position der Optionsbezeichner: nach Terminator/als 2. Parameter bei [Regexp.new](#)

6.2 REGULÄRE AUSDRÜCKE

Erweiterte Möglichkeiten

Ein regulärer Ausdruck kann auf einen String durch die `Regexp#match(string)` oder die Operatoren `=~` und `!~` (negativer Match) angewendet werden. Bei der Verwendung der Operatoren ist die Reihenfolge der Argumente irrelevant.

```
name = "Fats Waller"
name =~ /a/           # => 1
name =~ /z/           # => nil
/a/ =~ name           # => 1
/a/.match(name)       # => #<MatchData "a">
Regexp.new("all").match(name) # => #<MatchData "all">
```

Nach einem erfolgreichen Match setzt Ruby eine Reihe von Variablen: `~$&` enthält den Abschnitt des Strings, der mit dem regulären Ausdruck entspricht, `$`` den Abschnitt des Strings vor dem Match und `$'` und `!~` den Abschnitt des Strings nach dem Match. Bei `MatchData`-Objekten kann man stattdessen den Index `[0]` und die Methoden `pre_match` bzw. `post_match` verwenden, um die entsprechenden Abschnitte des Strings anzusprechen.

6.2 REGULÄRE AUSDRÜCKE

Erweiterte Möglichkeiten

```
def show_regexp(string, pattern)
  match = pattern.match(string)
  if match
    "#{match.pre_match}->#{match[0]}<-#{match.post_match}"
  else
    "no match"
  end
end

show_regexp('very interesting', /t/)    # => very in->t<-eresting
show_regexp('Fats Waller', /a/)        # => F->a<-ts Waller
show_regexp('Fats Waller', /lle/)      # => Fats Wa->lle<-r
show_regexp('Fats Waller', /z/)        # => no match
```

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Anker

Normalerweise wird ein String von links nach rechts durchsucht und der erste Teilstring gefunden, der dem regulären Ausdruck genügt.

Um nur Treffer am Anfang/Ende einer Zeile/eines Strings zu erhalten kann man folgende Ankerzeichen verwenden:

- `^` - Match am Zeilenanfang
- `$` - Match am Zeilenende
- `\A` - Match am Stringanfang
- `\z \Z` - Match am Stringende (exkl. `\n`)
- `\b \B` - Match mit Wortgrenzen/Non-Wortgrenzen.

<code>str = "this is\nthe time"</code>	
<code>show_regexp(str, /^the/)</code>	<code># => this is\n->the<- time</code>
<code>show_regexp(str, /is\$/)</code>	<code># => this ->is<-\nthe time</code>
<code>show_regexp(str, /\Athis/)</code>	<code># => ->this<- is\nthe time</code>
<code>show_regexp(str, /\Athe/)</code>	<code># => no match</code>
<code>show_regexp("this is\nthe time", /\bis/)</code>	<code># => this ->is<-\nthe time</code>
<code>show_regexp("this is\nthe time", /\Bis/)</code>	<code># => th->is<- is\nthe time</code>

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Zeichenklassen

In regulären Ausdrücken wird eine Zeichenklassen durch in eckige Klammern gesetzte Zeichen bezeichnet: $[z_1 \dots z_n]$.

Zeichen, die ansonsten in regulären Ausdrücken eine besondere Bedeutung besitzen (wie z.B. `.`) verlieren sie in diesem Kontext. Normale Stringsubstitutionen sind aber weiter möglich: So bezeichnet `\b` das Backspace-Zeichen und `\n` das Newline-Zeichen. Durch Angaben der Form $[z_1-z_n]$ Zeichenbereiche ausgewählt werden.

<code>show_regexp('Price \$12.', /[aeiou]/)</code>	<code># => Pr->i<-ce \$12.</code>
<code>show_regexp('Price \$12.', /[\\s]/)</code>	<code># => Price-> <-\$12.</code>
<code>show_regexp('Price \$12.', /[\\\$.]/)</code>	<code># => Price ->\$<-12.</code>
<code>a = 'see [The PickAxe-page 123]'</code>	
<code>show_regexp(a, /[A-F]/)</code>	<code># => see [The Pick->A<-xe-page 123]</code>
<code>show_regexp(a, /[A-Fa-f]/)</code>	<code># => s->e<-e [The PickAxe-page 123]</code>
<code>show_regexp(a, /[0-9]/)</code>	<code># => see [The PickAxe-page ->1<-23]</code>
<code>show_regexp(a, /[0-9][0-9]/)</code>	<code># => see [The PickAxe-page ->12<-3]</code>

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Zeichenklassen

Eine Zeichenklassen kann *negiert* werden, indem man direkt hinter die öffnende Klammer das Zeichen `^` setzt.

Außerdem kann man auch die POSIX-Zeichenklassenbezeichner verwenden:

<code>show_regexp('Price \$12.', /[^A-Z]/)</code>	<code># => P->r<-ice \$12.</code>
<code>show_regexp('Price \$12.', /[^\w]/)</code>	<code># => Price-> <-\$12.</code>
<code>show_regexp('Price \$12.', /[a-z][^a-z]/)</code>	<code># => Pric->e <-\$12.</code>
<code>show_regexp('Price \$12.', /[aeiou]/)</code>	<code># => Pr->i<-ce \$12.</code>
<code>show_regexp('Price \$12.', /[[:digit:]]/)</code>	<code># => Price \$->1<-2.</code>
<code>show_regexp('Price \$12.', /[[:space:]]/)</code>	<code># => Price-> <-\$12.</code>
<code>show_regexp('Price \$12.', /[[:^alpha:]]/)</code>	<code># => Price-> <-\$12.</code>
<code>show_regexp('Price \$12.', /[[:punct:]]aeiou/)</code>	<code># => Pr->i<-ce \$12.</code>

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Zeichenklassen

POSIX Character Classes (Unicode)

<code>[:alnum:]</code>	Alphanumeric (<i>Letter</i> <i>Mark</i> <i>Decimal_Number</i>)
<code>[:alpha:]</code>	Uppercase or lowercase letter (<i>Letter</i> <i>Mark</i>)
<code>[:ascii:]</code>	7-bit character including nonprinting
<code>[:blank:]</code>	Blank and tab (+ <i>Space_Separator</i>)
<code>[:cntrl:]</code>	Control characters—at least 0x00–0x1f, 0x7f (<i>Control</i> <i>Format</i> <i>Unassigned</i> <i>Private_Use</i> <i>Surrogate</i>)
<code>[:digit:]</code>	Digit (<i>Decimal_Number</i>)
<code>[:graph:]</code>	Printable character excluding space (Unicode also excludes <i>Control</i> , <i>Unassigned</i> , and <i>Surrogate</i>)
<code>[:lower:]</code>	Lowercase letter (<i>Lowercase_Letter</i>)
<code>[:print:]</code>	Any printable character (including space)
<code>[:punct:]</code>	Printable character excluding space and alphanumeric. Unicode: (<i>Connector_Punctuation</i> <i>Dash_Punctuation</i> <i>Close_Punctuation</i> <i>Final_Punctuation</i> <i>Initial_Punctuation</i> <i>Other_Punctuation</i> <i>Open_Punctuation</i>)
<code>[:space:]</code>	Whitespace (same as <code>\s</code>)
<code>[:upper:]</code>	Uppercase letter (<i>Uppercase_Letter</i>)
<code>[:xdigit:]</code>	Hex digit (0–9, a–f, A–F)
<code>[:word:]</code>	Alphanumeric, underscore, and multibyte (<i>Letter</i> <i>Mark</i> <i>Decimal_Number</i> <i>Connector_Punctuation</i>)

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Zeichenklassen

Sequence	As [...]	Meaning (Unicode)
\d	[0-9]	Decimal digit character (<i>Decimal_Number</i>)
\D	[^0-9]	Any character except a digit
\h	[0-9a-fA-F]	Hexadecimal digit character
\H	[^0-9a-fA-F]	Any character except a hex digit
\s	[_\t\r\n\f]	Whitespace character (+ <i>Line_Separator</i>)
\S	[^\t\r\n\f]	Any character except whitespace
\w	[A-Za-z0-9_]	Word character (+ <i>Connector_Punctuation</i> , <i>Letter</i> , <i>Mark</i> , and <i>Number</i>)
\W	[^A-Za-z0-9_]	Any character except a word character

```
a = 'see [The PickAxe-page 123]'
show_regexp(a, /\[/)           # => see [The PickAxe-page 123->]<-
show_regexp(a, /[0-9\]/)       # => see [The PickAxe-page ->1<-23]
show_regexp(a, /\d\-/)         # => see [The PickAxe->-<-page 123]

show_regexp('It costs $12.', /\s/) # => It-> <-costs $12.
show_regexp('It costs $12.', /\d/)  # => It costs $->1<-2.

str = "now is the time"
str.gsub(/[a-z&&[^aeiou]]/, '*')    # => "*o* i* **e *i*e"
```


6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Wiederholungen

Wenn r ein beliebiger regulärer Ausdruck ist, dann matcht

r^*	0 oder mehr Vorkommen von r ;
r^+	1 oder mehr Vorkommen von r ;
$r?$	0 oder 1 Vorkommen von r ;
$r\{m,n\}$	mindestens m und höchstens n Vorkommen von r ;
$r\{m,\}$	mindestens m Vorkommen von r ;
$r\{,n\}$	höchstens n Vorkommen von r und
$r\{m\}$	genau m Vorkommen von r .

```
a = "The moon is made of cheese"
show_regexp(a, /\w+/)           # => ->The<- moon is made of cheese
show_regexp(a, /\s.*\s/)        # => The-> moon is made of <-cheese
show_regexp(a, /\s.*?\s/)       # => The-> moon <-is made of cheese
show_regexp(a, /[aeiou]{2,99}/)  # => The m->oo<-n is made of cheese
show_regexp(a, /mo?o/)          # => The ->moo<-n is made of cheese
# minimaler Match durch ,?'
show_regexp(a, /mo??o/)         # => The ->mo<-on is made of cheese
```

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Alternation

Alternativen können durch das Zeichen `|` miteinander kombiniert werden:

```
a = "red ball blue sky"
show_regexp(a, /d|e/)          # => r->e<-d ball blue sky
show_regexp(a, /al|lu/)        # => red b->al<-l blue sky
show_regexp(a, /red ball|angry sky/) # => ->red ball<- blue sky
```

Gruppen

Innerhalb eines regulären Ausdrucks können Teilausdrücke durch Klammern zusammengefasst werden. Jeder Teilausdruck/Gruppe wird als eigenständiger regulärer Ausdruck behandelt.

Innerhalb eines regulären Ausdrucks kann man sich mit `\1`, `\2`, ... auf die verschiedenen Gruppen beziehen; außerhalb des Ausdrucks mit `$1`, `$2`, ...

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Gruppen

```
# This matches an 'a' followed by one or more 'n's
show_regexp('banana', /an+/)          # => b->an<-ana
# This matches the sequence 'an' one or more times
show_regexp('banana', /(an)+/)        # => b->anan<-a
a = 'red ball blue sky'
show_regexp(a, /blue|red/)             # => ->red<- ball blue sky
show_regexp(a, /(blue|red) \w+/)       # => ->red ball<- blue sky
show_regexp(a, /(red|blue) \w+/)       # => ->red ball<- blue sky
show_regexp(a, /red|blue \w+/)         # => ->red<- ball blue sky
show_regexp(a, /red (ball|angry) sky/) # => no match
a = 'the red angry sky'
show_regexp(a, /red (ball|angry) sky/) # => the ->red angry sky<-
```

```
/(\d\d):(\d\d)(..)/ =~ "12:50am"  # => 0
"Hour is #$1, minute #$2"          # => "Hour is 12, minute 50"
/((\d\d):(\d\d))(..)/ =~ "12:50am" # => 0
"Time is #$1"                       # => "Time is 12:50"
"Hour is #$2, minute #$3"           # => "Hour is 12, minute 50"
"AM/PM is #$4"                      # => "AM/PM is am"
```

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Gruppen

```
md = /(\d\d):(\d\d)(..)/.match("12:50am")
"Hour is #{md[1]}, minute #{md[2]}"          # => "Hour is 12, minute 50"
md = /((\d\d):(\d\d))(..)/.match("12:50am")
"Time is #{md[1]}"                          # => "Time is 12:50"
"Hour is #{md[2]}, minute #{md[3]}"          # => "Hour is 12, minute 50"
"AM/PM is #{md[4]}"                         # => "AM/PM is am"

# match duplicated letter
show_regexp('He said "Hello"', /(\w)\1/)    # => He said "He->ll<-o"
# match duplicated substrings
show_regexp('Mississippi', /(\w+)\1/)       # => M->ississ<-ippi
```

```
# match duplicated letter
str = 'He said "Hello"'
show_regexp(str, /(?<char>\w)\k<char>/)     # => He said "He->ll<-o"
# match duplicated adjacent substrings
str = 'Mississippi'
show_regexp(str, /(?<seq>\w+)\k<seq>/)      # => M->ississ<-ippi
```


6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Musterbasierte Substitution

```
a = "quick brown fox"
a.sub(/[aeiou]/, '*')      # => "q*ick brown fox"
a.gsub(/[aeiou]/, '*')     # => "q**ck br*wn f*x"
a.sub(/\s\S+/, "")         # => "quick fox"
a.gsub(/\s\S+/, "")        # => "quick"

a = "quick brown fox"
a.sub(/^./) {|match| match.upcase }      # => "Quick brown fox"
a.gsub(/[aeiou]/) {|vowel| vowel.upcase } # => "qUIck brOwn fOx"
```

```
def mixed_case(name)
  name.downcase.gsub(/\b\w/) {|first| first.upcase }
end

mixed_case("DAVE THOMAS")  # => "Dave Thomas"
mixed_case("dave thomas")  # => "Dave Thomas"
mixed_case("dAvE tHoMas")  # => "Dave Thomas"
```

6.2 REGULÄRE AUSDRÜCKE

Komplexere Muster Backslash-Folgen in Substitutionen

```
puts "fred:smith".sub(/(\w+):(\w+)/, '\2, \1')      # => smith, fred
puts "nercpyitno".gsub(/(.)\.)/, '\2\1')           # => encryption

puts "fred:smith".sub(/(?<first>\w+):(?<last>\w+)/, '\k<last>, \k<first>') # => smith, fred
puts "nercpyitno".gsub(/(?<c1>\.)(?<c2>\.)/, '\k<c2>\k<c1>')           # => encryption
```

```
str.gsub(/\\/, '\\\\')
```

```
str = 'a\b\c' # => "a\b\c"
```

```
str.gsub(/\\/, '\\\\\\\\\\\\\\') # => "a\\\\b\\\\c"
```

```
str = 'a\b\c' # => "a\b\c"
```

```
str.gsub(/\\/, '\&\&') # => "a\\b\\c"
```

```
str = 'a\b\c' # => "a\b\c"
```

```
str.gsub(/\\/, {'\\\\'}) # => "a\\\\b\\\\c"
```