Reguläre Ausdrücke

Reguläre Ausdrücke

- sind ein typisches Leistungsmerkmal vieler Skriptsprachen
- bestehen aus Mustern (engl. "*patterns*") zum Suchen und Ersetzen in Zeichenketten ("*strings*")
- zerlegen Strings in verschiedene Abschnitte
- wirken im Normalfall auf Textzeilen, können aber auch auf mehrzeilige Strings angewendet werden.
- existieren schon seit Jahrzehnten in Unix und konzeptionell bereits seit den 1940er Jahren
- wurden durch Perl seit Anfang der 1990er Jahre populär
- beeinflussen eine Reihe von globalen Variablen
- wurden deshalb von Ruby objekt-orientiert gekapselt
- besitzen Optionen (!)
- werden von Ruby sowohl in traditioneller als auch in objekt-orientierter Weise angeboten

- ... sind einfach unverzichtbar!

Reguläre Ausdrücke

- Der *match*-Operator **=~**
  - Ein regulärer Ausdruck wird mittels des "*match*"-Operators **=~** zu einem String in Beziehung gesetzt (Bem,: Reihenfolge…):

```
str =~ re
```

```
re =~ str
```

  - Falls String und regulärer Ausdruck zueinander passen ("*match*"), liefert **str =~ re** den Index des ersten passenden Zeichens innerhalb **str** zurück, beginnend mit 0:

```
"0123456789" =~ /2/ # 2
"ABCDEFGHIJ" =~ /CD/# 2
```

  - Falls der Vergleich scheitert, ergibt der Ausdruck **nil**:

```
"0123456789" =~ /A/ # nil
```

  - Es ist zwar möglich, mit **=~** zwei Strings zu vergleichen; der zweite String wird dann implizit in einen regulären Ausdruck gewandelt:

```
"0123456789" =~ '2' # 2
```

    Dieses Vorgehen ist aber nicht erwünscht und bewirkt eine Warnung!

Achtung: **=~** bewirkt Seiteneffekte auf eine Reihe globaler Variablen. Siehe unten!

Reguläre Ausdrücke

- Reguläre Ausdrücke in Ruby
  - Ruby verwaltet reguläre Ausdrücke als Objekte einer eigenen Klasse **`Regexp`**

```
re = /2/                  # Traditionelle Schreibweise
"0123456789" =~ re        # 2
re.class                  # Regexp
```

```
re = Regexp.new('2')      # alternativ, objekt-orientiert
"0123456789" =~ re        # 2
re.class                  # Regexp
```

```
re = %r(2); ...           # Erinnerung: Generalis. String
```

- Optionen regulärer Ausdrücke (Verwendung s.u.)
  - i *case insensitive*
  - o *substitute once* (bezogen auf Ersetzungsmuster #{}...)
  - m *multiline mode* ('.' passt dann auch auf \n, s.u.)
  - x *extended mode* (Aktivierung erweiterter Möglichkeiten)
  - Multibyte-Optionen n, e, s, u (z.B. für UTF-8), hier ignoriert.

Muster in regulären Ausdrücken

- Normale Zeichen
  - Alle "normalen" Zeichen (Sonderzeichen s.u.) suchen sich selbst:

```
"Dies ist ein Text" =~ /ein/      # 9 (Offset of match)
"Dies ist ein Text" =~ /kein/     # nil (no match)
```

  - Wirkung des Match-Operators **=~** hier wie Substring-Suche

- Sonderzeichen:

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

- Sonderzeichen \ (b*ackslash*)
  - Ähnlich wie bei Stringeingaben dient \ hier als *Escape*-Zeichen: Sonderzeichen werden durch *Escaping* (mit Präfix "\") zu normal suchbaren Zeichen:

```
"Rechne: 3 ** 4 = 81" =~ /\*\*/   # 10 (offset of '*')
```

```
winpath = 'C:\TEMP\sample.txt'
winpath =~ /\\Temp/i              # 2; mit Option i!
```

Muster in regulären Ausdrücken

- Sonderzeichen **.** (Punkt)
  - Der Punkt passt zu jedem beliebigen einzelnen Zeichen (*wildcard*).

```
"Beleg" =~ /el.g/    # 1
"Belag" =~ /el.g/    # 1
"bel*g" =~ /el.g/    # 1
```

  - Verliert Sonderbedeutung innerhalb eckiger Klammern [ ], s.u.
  - Sonderfall Zeilenende-Zeichen (\n):

```
a = "Zeile1\nZeile2\nZeile3\n"  # Multiline-Beispiel
```

```
a =~ /eile./          # 1
a =~ /eile../         # nil  (. passt nicht zu \n!)
a =~ /eile../m        # 1    (. passt nun zu \n)
```

```
re = Regexp.new('eile..', Regexp::MULTILINE)
a =~ re               # 1    (analog)
```

Muster in regulären Ausdrücken

- Sonderzeichen **^** (Zeilenanfang); **A** (String-Anfang)

```
"ein Fall"  =~ /ei/          # 0
"kein Fall" =~ /ei/          # 1
```

```
"ein Fall"  =~ /^ei/         # 0
"kein Fall" =~ /^ei/         # nil
```

```
"ein Fall"  =~ /\Aei/        # 0
"kein Fall" =~ /\Aei/        # nil
```

  - ^ und \A sind nur im Fall einzeiliger Strings gleichwertig:

```
a = "eins\nzwei\ndrei\n"
a =~ /^zwei/                 # 5
a =~ /\Azwei/                # nil
```

Muster in regulären Ausdrücken

- Sonderzeichen **$** (Zeilenende); **\Z**, **\z** (String-Ende)

```
"Fall"  =~ /ll/                 # 2
"Falle" =~ /ll/                 # 2
```

```
"Fall"  =~ /ll$/                # 2
"Falle" =~ /ll$/                # nil
```

```
"Fall"  =~ /ll\Z/               # 2
"Falle" =~ /ll\Z/               # nil
```

  - $ und \Z sind nur im Fall einzeiliger Strings gleichwertig:

```
a = "eins\nzwei\ndrei\n"
a =~ /zwei$/                    # 5
a =~ /zwei\Z/                   # nil
```

  - \Z ignoriert \n am String-Ende, \z nicht:

```
a =~ /drei$/                    # 10
a =~ /drei\Z/                   # 10
```

```
a =~ /drei\z/                   # nil
a =~ /drei.\z/                  # nil
a =~ /drei.\z/m                 # 10
```

Muster in regulären Ausdrücken

- **Wiederholungsangaben** mit Sonderzeichen **{**
  - In geschweiften Klammern gibt man die minimale und maximale Wiederholungszahl für das davor stehende Muster an:

```
re = /abc{2,4}/       # 'ab' gefolgt von 2..4 mal 'c'
"abc" =~ re           # nil
"abcc" =~ re          # 0
"abccc" =~ re         # 0
"abcccc" =~ re        # 0
"abccccc" =~ re       # 0 (letztes 'c' stört nicht!)
```

  - Kurzformen: {n} = {n,n}, {n,} = {n, ∞}
- Traditionelle Kurzformen: Sonderzeichen **?**, **+**, *
  - **?** = {0,1}, **+** = {1, }, * = {0, }

```
"ab" =~ /bc*/         # 1
"abc" =~ /bc?d/       # nil
"abd" =~ /bc?d/       # 1
"abcd" =~ /bc+d/      # 1
"abccd" =~ /bc+d/     # 1
"abccd" =~ /bc?d/     # nil
```

Muster in regulären Ausdrücken

- **Wiederholungsangaben**: *Greedy* vs. *non-greedy*
  - Manchmal ist es entscheidend, zwischen der maximal und der minimal ausführbaren Wiederholungszahl zu unterscheiden:

```
# Suche nach Zeichen zwischen whitespace
line = "Dies ist ein kurzer Satz."
# "gierige" Variante (Normalfall)
line =~ /\s.*\s/     # Trefferbereich: "ist ein kurzer"
# "nicht gierige" Variante
line =~ /\s.*?\s/    # Trefferbereich: "ist"
# "nicht gierige" Variante, verallgemeinert
line =~ /\s.{5,}?\s/# Trefferbereich: "ist ein"
```

  - Unterscheide:

`*`	und	`*?`
`+`	und	`+?`
`{n,}`	und	`{n,}?`

  - **Test**: Was ergibt (mit "line" wie oben definiert)

```
line =~ /A*/
```

Antwort: **0** (Jeder String enthält mind. 0-mal 'A'!)

Muster in regulären Ausdrücken

- **Auswahl** mit Sonderzeichen |
  - "Oder"-Beziehungen sind einfach möglich mittels |:

```
"abc" =~ /c|d/       # 2    # c oder d
"abd" =~ /c|d/       # 2
"abe" =~ /c|d/       # nil
```

  - | bindet nur schwach:

```
"abc" =~ /bc|d/      # 1 ("b", dann "c" oder "d"?)
"abd" =~ /bc|d/      # 2   ("bc" oder "d" !!)
"abe" =~ /bc|d/      # nil
"abd" =~ /bc|bd/     # 1   ("bc" oder "bd")
```

- **Gruppenbildung** mit Sonderzeichen **()**

```
"abc" =~ /b(c|d)/    # 1 ("b", dann "c" oder "d")
"abd" =~ /b(c|d)/    # 1
"abcabdabe" =~ /(ab.)+/     # 0
"abcde" =~ /b(c.)e/         # 1
```

  - Seiteneffekt: \1 ... \9 (bzw. $1 ... $9) gesetzt (*backreferences*)

Muster in regulären Ausdrücken

- **Zeichenmengen** mit Sonderzeichen **[**
  - In eckigen Klammern listet man alle Zeichen, die an einer bestimmten Stelle zulässig sind:

```
"abdf" =~ /[cde]/    # 2  ('d' passt)
```

  - Auch Bereiche (bez. ASCII-Codierung) sind zulässig

```
/[01]/               # eine Binärziffer
/[0-7]/              # eine Oktalziffer
/[0-9a-fA-F]+/       # mind. eine Hex-Ziffer
```

  - Suche nach '-' bzw. ']' möglich, wenn Zeichen am Anfang der Liste:

```
/[-+]/               # Vorzeichensuche
/[])}>]/             # Suche nach schließenden Klammern
```

  - Sonderzeichen: In Zeichenmengen wie normale Zeichen (außer '\')

```
/[.|(){}+^$*?]/      # in [...] normal suchbar
```

  - Komplementbildung durch Voranstellen von '^'

```
/[^0-9]/             # Alles außer einer Ziffer
/[^^]/               # Alles außer '^'
```

Muster in regulären Ausdrücken

- **Spezielle Zeichen und Zeichenmengen**
  - \A, \Z, \z — schon besprochen (String-Anfang / -Ende)
  - \d, \D — [0-9], [^0-9]
  - \n, \t, ... — *newline*, *tab*, ... (vgl. Strings)
  - \000, ... \277 — Suche über Oktalwert des Zeichens
  - \x00, ... \xff — Suche über Hexadezimalwert des Zeichens
  - \s, \S — [\t\n\r\f] (*white space*), [^...] (invers)
  - \b — Backspace, wenn innerhalb [...], Wortgrenze sonst;
    \B — Treffer, wenn keine Wortgrenze

```
"einszwei zweidrei" =~ /zwei/     # 4
"einszwei zweidrei" =~ /\bzwei/   # 9
"einszwei zweidrei" =~ /\Bzwei/   # 4
```

  - \w, \W — [0-9A-Za-z_] (*word char*), [^...] (invers)
  - \u20AC\u{fc} — Suche über Unicode-Wert, hier: €ü. Ruby 1.9
  - \G — Stelle, an der der letzte Treffer endete
    (dokumentiert, aber ohne Beispiel. Noch zu klären!)

Reguläre Ausdrücke und globale Variablen

- Match-Operator bewirkt String-Zerlegung und Setzen bestimmter „globaler“ Variablen:
  - $` Teilstring vor dem Treffer
  - $& Trefferbereich
  - $' Teilstring hinter dem Trefferbereich
  - $1...$9 Falls Gruppen (...) angegeben: Teilstring jeder Gruppe
  - $~ Das erzeugte MatchData-Objekt (siehe Teil III)
- Beispiel zur Nutzung (aus dem *pickaxe*-Buch):

```
def showRE(a, re)    # Trefferbereich hervorheben
   a =~ re ? "#{$`}<<#{$&}>>#{$'}" : "no match"
end
```

```
showRE("yes | no", /\|/)    # "yes <<|>> no"
```

```
a = "Eins zwei drei"
showRE(a, /ei/)             # "Eins zw<<ei>> drei"
showRE(a, /ei/i)            # "<<Ei>>ns zwei drei"
```

Reguläre Ausdrücke und globale Variablen

- Reichweite der „globalen“ Variablen:
  - *Thread*-lokal
  - Methoden-lokal:
    Aufrufende Methoden sind nicht betroffen

- Änderbarkeit dieser Variablen?
  - Basis ist stets **`$~`** - **`$``**, **`$&`** und **`$´`** etc. werden davon abgeleitet
  - Es ist möglich, **`$~`** mit einem eigenen MatchData-Objekt zu überschreiben. Die abgeleiteten Variablen ändern sich dann entsprechend.
  - **`$``**, **`$&`** und **`$´`** sowie **`$1 … $9`** lassen sich nicht direkt überschreiben (*read-only*)

Reguläre Ausdrücke und *backreferences*

- Wirkung von Gruppenbildung mit runden Klammern (...)

1. Wiederholungsangaben für ganze Gruppeninhalte möglich

```
"aababcabcdabcde" =~ /(ab){2}/    # 1
```

2. „Globale“ Variablen $1 ... $9 sind danach mit Klammerwert gesetzt:

```
line                        # "x = -5"
key, value = $1, $2 if line=~/\s*(\w+)\s*=\s*([-+]?\d+)/
# key = "x", value = "-5"
```

3. Klammerwerte sind über \1, ..., \9 innerhalb des regulären Ausdrucks verfügbar ("*backreferences*").

– Beispiel 1: Suche nach doppelten Vokalen

```
"Woo sind hier doppelte Vokaale?" =~ /([aeiou])\1/i # 1
"Wo sind hier doppelte Vokaale?" =~ /([aeiou])\1/i # 25
```

– Beispiel 2: Suche nach doppelten Wörtern

```
"This has has been a word too much" =~ /(\w+)\s+\1/ # 5
```

– Beispiel 3: Flexible Suche nach *string delimiter*-Paaren

```
"He said 'Hello'" =~ /(["']).*?\1/ # 8, $& == %('Hello')
```

Reguläre Ausdrücke II

Arbeiten mit regulären Ausdrücken

Suchen mit regulären Ausdrücken

- Beispiel: Zeilenfilter
  - Nur auf bestimmte Zeilen reagieren

```
comment_lines, unmatched_lines, dict = 0, 0, {}
while line = gets
case line
when /^#/                       # Kommentarzeilen zählen
   comment_lines += 1
when /(\w+)\s*=\s*(\w.*)/       # key/value-Paar?
   dict[$1] = $2.strip
when other_regex                # usw.
   do_something_else
else
   unmatched_lines += 1
end
```

Suchen mit regulären Ausdrücken

- Beispiel: Intelligentes "Prompt"
  - Nur (per Regexp) klar definierte Eingaben akzeptieren

```
def my_prompt( ptext, regex )
  loop do
   print ptext; line = gets.chomp
   return $& if line =~ regex # $& = Trefferbereich
   puts "Illegal value - please retry."
  end
end
```

```
age = my_prompt("Enter age (0-129): ",
   /^\d$|^\d\d$|^1[012]\d$/)
date_regex =
   /^20\d\d-((0[1-9])|(1[0-2]))-((0[1-9])|([12]\d)|(3[01]))$/
date = my_prompt("Enter current date as 20YY-MM-DD: ",
                 date_regex)
# usw. / vgl. Demo!
puts "age = #{age}"
puts "date = #{date}"
```

Ersetzen mit regulären Ausdrücken

- Methoden **`String#sub`** & **`String#gsub`**
  - sub: Einmaliges Ersetzen (*substitution*)
  - gsub: Mehrmaliges Ersetzen (*global substitution*)
  - sub!, gsub!: Varianten, die den aktuellen String verändern.
- Einfacher Modus, mit Verwendung von *backreferences*:
  - **`str.sub( regexp, `*`replacement`*` )`**

```
"Hello, world".sub(/world/, "friends")
"Hello, friends"
"hello".sub(/([aeiou])/, '<\1>') # "h<e>llo"
"hello".gsub(/([aeiou])/, '<\1>')# "h<e>ll<o>"
```

- Block-Modus, mit Verwendung der globalen Variablen:
  - **`str.gsub( regexp ) {|match| block }`**

```
"Dies ist ein Satz".gsub(/\b\w/) {|c| c.upcase}
"Dies Ist Ein Satz"
"Dies ist ein Satz".gsub(/\b\w(.)/) {$1.upcase}
"Ies St In Atz"
```

Ersetzen mit regulären Ausdrücken

```
def unescapeHTML( string ) # Beispiel aus pickaxe-Buch
    str = string.dup       # Auch für XML ideal geeignet
    str.gsub!(/&(.*?);/n) { # Löst char refs und
      match = $1.dup       # eingebaute entity refs auf!
      case match
      when /\Aamp\z/ni      then '&'
      when /\Aquot\z/ni     then '"'
      when /\Agt\z/ni       then '>'
      when /\Alt\z/ni       then '<'
      when /\A#(\d+)\z/n    then Integer($1).chr
      when /\A#x([0-9a-f]+)\z/ni  then $1.hex.chr
      end
    }
    str
end
```

```
unescapeHTML("1&lt;2 &amp;&amp; 4&gt;3")    # 1<2 && 4>3
unescapeHTML("&quot;A&quot;=&#65;=&#x41;") # "A"=A=A
```

Einige Ruby-Methoden, die Regexp nutzen

- **Enumerable#grep**, insb.: **Array#grep**
  - Bei Übergabe eines Regulären Ausdrucks an „grep“ erhält man ein Array der „matching elements“ zurück.

```
a = %w|Jan Feb Mär Apr Mai Jun Jul Aug Sep Okt Nov Dez|
a.grep(/^J/)  # ["Jan", "Jun", "Jul"]
```

```
ENV.keys.grep /HOME/ # ["HOME", "JAVA_HOME"]
```

- **String#scan**
  - Rückgabe der „Treffer“ als Array (von Arrays im Fall von Gruppen)

```
"1.23 +4/-6".scan(/[-+]?\d+/)     # ["1","23","+4","-6"]
"Hallo ?".scan(/(.)(..)/)  # [["H","al"],["l","o "]]
```

- **String#split**
  - Zerlegung in Array von Teil-Strings, Regexp = Trenner

```
"a; 1,2; b; 3 ; c; 4,5; d; 7".split /\s*;\s*/
# ["a", "1,2", "b", "3", "c", "4,5", "d", "7"]
```

Reguläre Ausdrücke III

Das objekt-orientierte Interface

Die Klassen Regexp und MatchData

Reguläre Ausdrücke: Die Klasse Regexp

- Konstanten:

```
Regexp::IGNORECASE, Rexexp::MULTILINE, Regexp::EXTENDED
```

- Klassenmethoden:

Regexp.compile, **Regexp.new**(pattern[,options[,lang]])

```
re = Regexp.new("abc\s*abc", Regexp::IGNORECASE, 'n')
# entspricht: re = /abc\s*abc/in
```

- Exkurs: Umgang mit Multibyte-Zeichensätzen

```
# lang = n (none), e (euc), s (sjis), u (utf8)
# Beispiel mit UTF-8 Codierung
a = "f\xc3\xbcr"     # ü in UTF-8-Codierung = c3, bc
a =~ /f.r/           # nil
a =~ /f.r/u          # 0
```

Reguläre Ausdrücke: Die Klasse Regexp

- Klassenmethoden (Forts.):

```
Regexp.last_match -> aMatchData
# entspricht $~

Regexp.quote( aString )  -> aNewString,
Regexp.escape( aString ) -> aNewString

str = "ak2'§$'$%&$()"           # Irgendwas!
str =~ %r(#{Regexp.escape(str)}) # Immer true !!
Regexp.escape('\\*?{}.')        # \\\\\*\?\{\}\.
```

Verwenden Sie `Regexp.escape` bzw. `Regexp.quote`, wenn Sie nach Teilstrings suchen wollen, die Sonderzeichen enthalten könnten. Die Methode nimmt Ihnen die Sonderzeichenbehandlung einfach ab.

Reguläre Ausdrücke: Die Klasse Regexp

- „Reguläre“ Methoden:

==

Vergleicht zwei Regexp-Objekte. Gleichheit gegeben bei Übereinstimmung in Suchmuster, Optionen und Zeichensatz:

```
/abc/ == /abc/                          # true
%r(abc) == Regexp.new("abc")            # true
/abc/ == /abc/i                         # false
/abc/u == /abc/n                        # false
```

===

Wird vom `case`-Statement verwendet. Wirkung dort wie `=~`

=~

Vergleicht mit einem String. Siehe auch `String#=~`

```
/bc/i =~ "ABCD"      # 1
```

~

Vergleicht mit `$_` – VERALTET; NICHT MEHR VERWENDEN!

```
# $_ = "Ein Test"
~ /Te/         # 4
```

Reguläre Ausdrücke: Die Klasse Regexp

- Reguläre Methoden:

casefold? Liefert Status der Option `IGNORECASE`

```
/abc/.casefold?      # false
/abc/i.casefold?     # true
```

kcode Liefert aktuellen Character set code

```
/abc/.kcode          # nil
/abc/n.kcode         # "none"
/abc/u.kcode         # "utf8"
```

match Liefert ein `Matchdata`-Objekt & setzt `$~` oder ergibt `nil`

```
/..(\w)/.match( "abcd" )    # ergibt 0; $1 = "c"
"abcd" =~ /..(w)/           # analoge Wirkung
```

Vorteil von match: Jede Benutzung ergibt ein neues `MatchData`-Objekt, das das Ergebnis festhält. Siehe unten (MatchData).

source Liefert den Test-String des Ausdrucks.

```
re = Regexp.new("abc", Regexp::IGNORECASE, "u")
# ...
re.source            # "abc"
```

Reguläre Ausdrücke: Die Klasse MatchData

- Anmerkungen
  - Die Klasse MatchData kapselt die Ergebnisse eines *match*-Operators, die gewöhnlich in globalen Variablen vorgehalten werden, und stellt Methoden zum einfachen Zugriff auf diese Daten bereit.
  - Durch Speicherung der *match*-Ergebnisse in separaten Objekten ist konfliktfreier Zugriff auf mehrere zurückliegende Mustervergleiche möglich, die Kollisionsgefahr bez. $1, …, $9 etc. ist gebannt.
- Konstanten, Klassenmethoden:
  - Keine
- Typische Nutzung:

```
regex = Regexp.new('\s*(\w+)\s*=\s*([-+]?\d+)')
if (md = regex.match( "  x = -5 cm" ))
  x = md[1]; y = md[2]   # md[i] wie $i, i=1..9
  units = md.post_match  # wie $'
end
```

Reguläre Ausdrücke: Die Klasse MatchData

- Reguläre Methoden:

[] Für Array-artigen Zugriff auf `$0; $1..$9`

```
md[0]       # $&          (ganzer Trefferbereich)
md[1]       # $1
md[2,3]     # [$2, $3, $4] (n, m: Start, Länge)
md[1..3]    # [$1, $2, $3] (n..m: Indexbereich)
```

begin Offset des ersten Zeichens von Gruppe `i ($i)`

end Offset+1 des letzten Zeichens von Gruppe `i ($i)`

```
md.begin(0) # 0 (Offset von "  x = ...")
md.begin(1) # 2 (Offset von "x")
md.begin(2) # 6 (Offset von "-5")
md.end(0)   # 8 (Offset von "5" +1; vgl. $&)
md.end(1)   # 3 (Offset von "x" +1)
```

length, size Analog zu Klasse `Array`

```
md.length   # 3 (für $0, $1, $2)
```

offset Array mit `begin-` und `end-`Werten

```
md.offset(0)# [0, 8], [md.begin(0), md.end(0)]
md.offset(1)# [2, 3]
```

Reguläre Ausdrücke: Die Klasse MatchData

- Reguläre Methoden:

`post_match`	`$'` (String hinter dem Treffer)
`md.post_match`	`# ' cm'`
`pre_match`	`$`` (String vor dem Treffer)
`md.pre_match`	`# '' (hier leer)`
`string`	(konstanter) "match"-String
`md.string`	`# "  x = -5 cm"`
`to_a`	Wandelt in echtes Array um
`md.to_a`	`# ["  x = -5", "x", "-5"]`
`to_s`	`$&` (String der Trefferregion)
`md.to_s`	`# "  x = -5"`
`captures`	Array der Treffergruppen (neu in V 1.8)
`md.captures`	`# ["x", "-5"]`
`select`	Array der Treffergruppen mit Bedingung
`values_at`	Array der Treffergruppen mit Indexauswahl

Reguläre Ausdrücke IV

extended regular expresssions

nach POSIX 1003.2

Extensions in regulären Ausdrücken

- **Grundmuster:**

```
(?...)
```

  - Eingeleitet mit runder Klammer (und Fragezeichen ?
  - Steuerzeichen direkt hinter dem Fragezeichen!
  - Abgeschlossen mit runder Klammer)
  - Nicht mit einfacher Gruppe (...) verwechseln!
  - Gruppenwirkung besteht (z.B. für |), aber:
  - Globale Variablen $1, ..., sowie *backreferences* \1, ... werden nicht gesetzt!

- **`(?# Kommentar)`**
  - Ermöglicht das Einfügen von Kommentartexten in reguläre Ausdrücke. Nützlich bei komplexen Fällen!

Extensions in regulären Ausdrücken

- **(?:re)**
  - Gruppenbildung ohne Erzeugung von $1, \1 etc.

```
datum = "1.12.2006"
r1 = /(\d+)(-|\.)(\d+)(-|\.)(\d+)/
m1 = r1.match( datum )
m1.captures   # ["1",".","12",".","2006"]
r2 = /(\d+)(?:-|\.)(\d+)(?:-|\.)(\d+)/
m2 = r2.match( datum )
m2.captures   # ["1","12","2006"]
```

- **(?=re)**
  - Matchbedingung ohne "verbrauchende" Wirkung

```
showRE("1, 2, 3", /\d+,/)  # "<<1,>> 2, 3"
showRE("1, 2, 3", /\d+(?=,)/)     # "<<1>>, 2, 3"
"1, 2, 3".scan(/\d+(?=,)/) # ["1", "2"]
```

Extensions in regulären Ausdrücken

- `(?=re)`
  - Matchbedingung ohne "verbrauchende" Wirkung

```
showRE("1, 2, 3", /\d+,/)  # "<<1,>> 2, 3"
showRE("1, 2, 3", /\d+(?=,)/)     # "<<1>>, 2, 3"
"1, 2, 3".scan(/\d+(?=,)/) # ["1", "2"]
```

- `(?!re)`
  - Wie `(?=re)`, aber wahr, wenn *re* nicht zutrifft

```
showRE("1, 2, 3", /\d+(?!,)/)     # "1, 2, <<3>>"
```

Extensions in regulären Ausdrücken

- `(?>re)`
  - Sog. possessives Verhalten – einmal passende Zeichen unterliegen keinem *Backtracking* (s.u.) mehr
  - Ermöglicht verschachtelte reguläre Ausdrücke. Diese können die Effizienz eines Ausdrucks erheblich steigern (Laufzeitoptimierung)! Beispiel:

```
require "benchmark"
include Benchmark
# Zeichenfolge suchen: a, beliebig, b, beliebig, a
str = "a" + ("b"*5000) # Böswilliges Gegenbsp.!
bm(10) do |test|
    test.report("Normal:   "){ str =~ /a.*b.*a/ }
    test.report("Possessiv:"){ str =~ /a(?>.*b).*a/ }
    test.report("Ruby 1.9: "){ str =~ /a.*+b).*a/ } # Kurzform
end
```

  - Die Kurzform wird ab Ruby 1.9 unterstützt und bereits jetzt von anderen Sprachen, z.B. Java.
  - Ergebnis: Siehe *online*-Demo! Diskussion: Ursachen?

Extensions in regulären Ausdrücken

- **`(?>re)`** (Forts.)
  - Was ist bei "`a.*b.*a`" passiert beim Anwenden auf "`abbbbb....bb`"?
  1. "a" wird sofort gefunden
  2. ".*" ist die "gierige" Wiederholungsvariante - sie liest erst mal alles (und merkt es sich für eventuelles *backtracking)*
  3. Passt "b" ? Nein → *backtracking* um ein Zeichen
  4. *backtracking* ok? Ja: Weiter mit (3), Nein: Suche erfolglos beenden.
  5. Zweites ".*" anwenden (auch hier: Rest einlesen & merken)
  6. Passt "a" ? Ja: Erfolg, Ende.
     Nein: → *backtracking*
  7. *backtracking* (2. Block) ok? Ja: (6) Nein: (3)
- Fazit:
  - Es gibt reguläre Ausdrücke mit Laufzeiten, die exponentiell sind in Bezug auf die Länge des untersuchten Strings! (?>re) vermeidet derartige Konstellationen durch Verhindern des *backtracking*.

Reguläre Ausdrücke und deklaratives Programmieren

- Prinzip des deklarativen Programmierens
  - Sage der Maschine, was sie lösen soll.
  - Sage ihr nicht, wie!
- Beispiele für deklarative Sprachen
  - SQL
  - XSLT
  - Und: Reguläre Ausdrücke!
- Probleme in der Praxis?
  - Ressourcenverbrauch: Rechenzeit, Speicherplatz
  - Trotz aller internen „Optimierer“ entstehen manchmal Konstellationen, die doch Eingriffe in den Lösungsweg erfordern.
  - Kritisches Laufzeitverhalten mancher Kombinationen aus Reg. Ausdrücken und Strings sowie der „*workaround*“ mit „(?> …)“ sind durchaus typisch.
  - Merke: Denken ist halt doch durch nichts zu ersetzen… ☺

Extensions in regulären Ausdrücken

- `(?imx)`
- `(?-imx)`
  - Ein- bzw. Ausschalten einer der Optionen i, m, x. Bei Verwendung innerhalb einer Gruppe bleibt die Wirkung auf diese Gruppe beschränkt!
- `(?imx:re)`
- `(?-imx:re)`
  - Ein- bzw. Ausschalten einer der Optionen i, m, x nur für *re*.

```
"klein GROSS" =~ /n.gr/           # nil
"klein GROSS" =~ /n.gr/i          # 4
"klein GROSS" =~ /n.(?i:g)r/i     # 4
"klein GROSS" =~ /n.(?-i:g)r/i    # nil
"klein GROSS" =~ /n.(?i:gr)/      # 4
```

Erweiterte Klammerausdrücke

- **Zeichenmengen** nach POSIX 1003.2
  - Benutzung: [:*name*:]
  - Zulässige Namen:

```
:alnum:, :alpha:, :blank:, :cntrl:,
:digit:, :graph:, :lower:, :print:,
:punct:, :space:, :upper:, :xdigit:
```

  - **Siehe auch:**

```
regex(3), regex(7), wctype(3)
```

  - **Beispiel:**

```
ShowRE("abc12defgh", /[0-9]+/)         # "abc<<12>>defgh"
ShowRE("abc12defgh", /[\d]+/)          # "abc<<12>>defgh"
ShowRE("abc12defgh", /[[:digit:]]+/)  # "abc<<12>>defgh"
ShowRE("abc12defgh", /[[:xdigit:]]+/)# "<<abc12def>>gh"
```

  - **Hinweis:** Zeichenmengen erfordern Verallgemeinerungen, wenn sie auf internationale Zeichensätze angewendet werden. Beispiel:
  - Einige XML-Standards wie XML Schema und XPath verwenden verallgemeinerte reguläre Ausdrücke im Kontext von Unicode.

Erweiterte Klammerausdrücke

Schließlich: Die folgenden POSIX-Standards sind NICHT IN RUBY VERFÜGBAR:

Gruppierungselemente wie Einzelzeichen behandeln

- Benutzung: [[.chars.]]
- Wirkung: Die in [..] eingeschlossene Zeichenfolge wird innerhalb [] wie ein Zeichen verwendet. Nur in Non-ASCII Umgebungen!
- **Beispiel** (wie es lauten könnte)**:**

```
ShowRE("chchcc", /[[.ch.]]*c/) # "<<chchc>>c"
```

Äquivalenzklassen nutzen

- Benutzung: [[=char=]]
- Wirkung: Je nach Sprachkontext (→ *locale*-Variablen) sind einige Zeichen äquivalent und können durch Angabe eines dieser Zeichen in [=…=] innerhalb eckiger Klammern gesucht werden
- **Beispiel** (wie es lauten könnte)**:**

```
ShowRE("Charité", /[[=e=]]*/) # "Charit<<é>>"
# Gesucht: „e“, gefunden: „é“ (e, é, è äquiv.)
```

Reguläre Ausdrücke: Die Zukunft

- Neu ab Ruby 1.9:
  - Unicode-Unterstützung für Strings

    … und weitere *character encodings*

  - Neue RegExp-Engine „Oniguruma“
    - Schneller!
    - Unterstützt ebendiese neuen **encodings**

Reguläre Ausdrücke: Ruby-Extras

- `/#{expr}/`
  - Wie in Strings verwenden – Ruby expandiert den Ausdruck in seine String-Darstellung, bevor daraus der RegExp gebildet wird
  - Normalfall: Späte Expansion (jedes Mal). Option „o“ („*once*“) modifiziert dies:

```
%w(a b).map{|x| /#{x}/}     # → [/a/, /b/]
%w(a b).map{|x| /#{x}/o}    # → [/a/, /a/]
```

- `[...], [^...]`
  - Zitat: „*Matches any single character (not) in brackets*“
  - (Kein Beispiel gefunden…)

Reguläre Ausdrücke: Ruby-Extras

- *Extended syntax*, x-Option
  - Eine Möglichkeit zum Aufbau lesbarer, komplexer und kommentierter regulärer Ausdrücke

```
re = / # Kommentar im Regexp, nicht von Ruby!
       R
       (uby)+
       \
     /x
```

```
"My Ruby installation" =~ re      # 3
"My Ruby" =~ re                   # nil
"No   Rubyubyuby ?" =~ re         # 5
```

  - *Whitespace* wird ignoriert, Kommentare ebenfalls
  - Hinter dem \ steht ein nicht ignorierter Blank

Reguläre Ausdrücke: Ruby-Extras

- Ruby 1.9
  - Regexp.union
  - Unicode

    `\u` Unicode-Zeichen finden
  - Gruppenbildung:

    `(?<first>\w)(?<second>\w)\k<second>\k<first>`

    `\k<`*`name`*`>` Rückbezug, Verallgemeinerung von `\1`, `\2` …

    `\g<n>`
  - Dritte Variante der Wiederholungsalgorithmen:

    `Possessive (+) beside (non-)greedy repetition`

Einzelheiten bei Bedarf bitte nachlesen – Ruby 1.9 soll hier nicht vertieft werden!