Skriptsprachen
* Ruby *

06 - Standardtypen und Reguläre Ausdrücke

Naumann
Wintersemester 2016

6.1 Standardtypen

Zahlen

Ruby unterstützt die üblichen Zahlentypen; d.h. ganze Zahlen, Fließkommazahlen, rationale und komplexe Zahlen. Ganze Zahlen, die innerhalb eines bestimmten, vordefinierten Bereichs liegen, werden intern in binärer Form kodiert und bilden den Typ bzw. die Klasse **Fixnum**; die anderen ganzen Zahlen bilden die Klasse **Bignum**. Ruby konvertiert automatisch nach Bedarf zwischen beiden Typen / Klassen.

```
num = 81
6.times do
  puts "#{num.class}: #{num}"
  num *= num
end

Fixnum: 81
Fixnum: 6561
Fixnum: 43046721
Bignum: 1853020188851841
Bignum: 3433683820292512484657849089281
```

6.1 Standardtypen

Zahlen

Einer ganzen Zahl kann ein Vorzeichen und ein Ausdruck, der das verwendete Zahlensystem bezeichnet, vorangestellt werden. Unterstrich-Zeichen (‚_‘) werden ignoriert:

```
123456                  => 123456                # Fixnum
0d123456                => 123456                # Fixnum
123_456                 => 123456                # Fixnum - underscore ignored
-543                    => -543                  # Fixnum - negative Zahl
0xaabb                  => 43707                 # Fixnum - hexadezimal
0377                    => 255                   # Fixnum - oktal
-0b10_1010              => -42                   # Fixnum - binär (negativ)
123_456_789_123_456_789 => 123456789123456789    # Bignum
```

Bei Fließkommazahlen muss links und rechts vom Dezimalpunkt eine Zahl stehen; ein Ausdruck wie `12.e4` wird nicht als Dezimalzahl, sondern als Aufruf der Methode `e4` auf die Zahl `12` interpretiert.

6.1 Standardtypen

Zahlen

Wenn bei numerischen Operationen Zahlen unterschiedlichen Typs verwendet werden, ist das Resultat (in der Regel) eine Zahl des allgemeineren Typs:

```
1 + 2                    # => 3
1 + 2.0                  # => 3.0
1.0 + 2                  # => 3.0
1.0 + Complex(1,2)       # => (2.0+2i)
1 + Rational(2,3)        # => (5/3)
1.0 + Rational(2,3)      # => 1.6666666666666667


1.0 / 2                  # => 0.5
1 / 2.0                  # => 0.5
1 / 2                    # => 0
```

6.1 Standardtypen

Zahlen

Wie für viele andere Klassen gibt es auch für Fixnum und Bignum eine Reihe nützlicher Iteratoren: times, upto, downto, ...

```
3.times             { print "X " }
1.upto(5)           {|i| print i, " " }
99.downto(95)       {|i| print i, " " }
50.step(80, 5)      {|i| print i, " " }
```

X X X 1 2 3 4 5 99 98 97 96 95 50 55 60 65 70 75 80

```
10.downto(7).with_index {|num, index| puts "#{index}: #{num}"}
```

0: 10
1: 9
2: 8
3: 7

6.1 Standardtypen

Zahlen

!!! Warnung für Perl-Programmierer !!!

Strings, die ausschließlich aus Zahlzeichen bestehen, werden nicht automatisch in Zahlen konvertiert wenn numerische Operationen auf ihnen ausgeführt werden.

```
datei_x.each do |line|
    v1, v2 = line.split          # Leerzeichen wird als Trenner interpretiert
    print v1 + v2, " "
end
```

34 56 78

```
datei_x.each do |line|
    v1, v2 = line.split          # Leerzeichen wird als Trenner interpretiert
    print Integer(v1) + Integer(v2), " "
end
```

7 11 15

```
3   4
5   6
7   8
```

datei_x

6.1 Standardtypen

Strings

In Ruby sind Strings nichts anderes als Folgen von Zeichen. Dabei muss es sich nicht unbedingt um darstellbare Zeichen handeln: Strings können durchaus auch binäre Daten enthalten. Als Escape-Zeichen wird der Backslash ‚\' verwendet:

```
'escape using "\\"' # => escape using "\"
'That\'s right' # => That's right
```

In mit doppelten Anführungszeichen begrenzten Strings kann beliebiger Kode eingebettet werden (#{...}). Wenn es sich nur um eine Variable handelt, können die Klammern weggelassen werden:

```
"Seconds/day: #{24*60*60}" # => Seconds/day: 86400
   "#{'Ho! '*3}Merry Christmas!" # => Ho! Ho! Ho! Merry Christmas!

   puts "now is #{ def the(a)
                       'the ' + a
                   end
                   the('time')
                } for all good coders..."
# => now is the time for all good coders...
```

6.1 Standardtypen

Strings

Neben einfache und doppelten Anführungszeichen gibt es noch drei weitere Möglichkeiten in Ruby Stings zu erzeugen: %q, %Q und *here documents*:

```
%q/general single-quoted string/       # => general single-quoted string
%Q!general double-quoted string!       # => general double-quoted string
%Q{Seconds/day: #{24*60*60}}           # => Seconds/day: 86400
```

Wie man sieht, ist das Q optional. Das auf q/Q folgende Zeichen wird als Stringbegrenzer interpretiert: Der String wird entweder durch ein Paar von Klammern oder zwei Vorkommen eines beliebigen Zeichens begrenzt:

```
( ... )
< ...>
[ ... ]
{ ... }
x ... x
```

6.1 Standardtypen

Strings

```
string = <<END_OF_STRING
    Der String besteht aus den
    Zeilen, die auf die Startmarkierung
    (‚<<‘ + Marke) folgen. Das Ende des
    Strings wird durch eine Zeile
    markiert, die mit der Marke beginnt
END_OF_STRING
```

```
print <<-STRING1, <<-STRING2
    Concat
    STRING1
        enate
        STRING2

    Concat
        enate
```

6.1 Standardtypen

Strings

In Ruby ist jeder String mit einer Kodierung verknüpft. Als Standardkodierung wird in Ruby 1.9 US-ASCII, in Ruby 2.0 UTF-8 verwendet:

```
# Ruby 1.9
plain_string = "dog"
puts "Encoding of #{plain_string.inspect} is #{plain_string.encoding}"

Encoding of "dog" is US-ASCII

#encoding: utf-8
plain_string = "dog"
puts "Encoding of #{plain_string.inspect} is #{plain_string.encoding}"
utf_string = "ðog"
puts "Encoding of #{utf_string.inspect} is #{utf_string.encoding}"

Encoding of "dog" is UTF-8
Encoding of "ðog" is UTF-8
```

6.1 Standardtypen

Strings

```
/jazz/j00132.mp3       | 3:45 | Fats          Waller       | Ain't Misbehavin'
/jazz/j00319.mp3       | 2:58 | Louis      Armstrong       | Wonderful World
/bgrass/bg0732.mp3 | 4:09 | Strength in Numbers  | Texas Red
          :                 :          :                 :
```

```
Song = Struct.new(:title, :name, :length)
File.open("songdata") do |song_file|
    songs = []
    song_file.each do |line|
        file, length, name, title = line.chomp.split(/\s*\|\s*/)
        songs << Song.new(title, name, length)
    end
    puts songs[1]
end

#<struct Song title="Wonderful World", name="Louis    Armstrong", length="2:58">
```

6.1 Standardtypen

Strings

```
Song = Struct.new(:title, :name, :length)
File.open("songdata") do |song_file|
    songs = []
    song_file.each do |line|
        file, length, name, title = line.chomp.split(/\s*\|\s*/)
        name.squeeze!(" ")
        songs << Song.new(title, name, length)
    end
    puts songs[1]
end

#<struct Song title="Wonderful World", name="Louis Armstrong", length="2:58">
```

6.1 Standardtypen

Strings

```
Song = Struct.new(:title, :name, :length)
File.open("songdata") do |song_file|
    songs = []
    song_file.each do |line|
        file, length, name, title = line.chomp.split(/\s*\|\s*/)
        name.squeeze!(" ")
        mins, secs = length.scan(/\d+/)
        songs << Song.new(title, name, mins.to_i*60 + secs.to_i)
    end
    puts songs[1]
end

#<struct Song title="Wonderful World", name="Louis Armstrong", length=178>
```

6.1 Standardtypen

Ranges

Bereichsangaben in Ruby werden durch Angabe eines Start- und Endpunkts mit Hilfe der Operatoren .. bzw. ... gebildet. Der Unterschied zwischen beiden Operatoren liegt darin, dass im ersten Fall der Endpunkt im Bereich liegt, im zweiten Fall dagegen nicht.

```
1..10
'a'..'z'
0..."cat".length

# Bereiche können in Arrays und Enumeratoren
# konvertiert werden:

(1..10).to_a       # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
('bar'..'bat').to_a  # => ["bar", "bas", "bat"]
enum = ('bar'..'bat').to_enum
enum.next          # => "bar"
enum.next          # => "bas"
```

6.1 Standardtypen

Ranges

```
# Iteration über Bereiche

digits = 0..9
digits.include?(5)              # => true
digits.min                      # => 0
digits.max                      # => 9
digits.reject {|i| i < 5 }      # => [5, 6, 7, 8, 9]
digits.inject(:+)               # => 45
```

Um Bereichsangaben für selbstdefinierte Objekte bzw. Klassen verwenden zu können, reicht es aus, die `succ`-Methode und den `<=>`-Operator für diese Klasse zu definieren. `x <=> y` liefert den Wert `-1`, `0` oder `1` abhängig davon, ob das erste Argument kleiner, gleich oder größer als das zweite Argument ist.

6.1 Standardtypen

Ranges

```
class PowerOfTwo
    attr_reader :value
    def initialize(value)
        @value = value
    end
    def <=>(other)
        @value <=> other.value
    end
    def succ
        PowerOfTwo.new(@value + @value)
    end
    def to_s
        @value.to_s
    end
end
p1 = PowerOfTwo.new(4)
p2 = PowerOfTwo.new(32)
puts (p1..p2).to_a

4
8
16
32
```

6.1 Standardtypen

Ranges

Bereichsangaben können auch als Bedingungen verwendet werden. Sie wirken hier wie ein Schalter, der aktiviert wird, wenn der *Startpunkt* des Bereichs realisiert ist und deaktiviert wird, sobald der *Endpunkt* erscheint:

```
while line = gets
    puts line if line =~ /start/ .. line =~ /end/
end
```

Ab der ersten Zeile, die die Zeichenfolge `start` enthält, werden alle Zeilen ausgegeben, bis eine Zeile die Zeichenfolge `end` enthält.

Um zu überprüfen, ob ein Objekt innerhalb eines Bereiches liegt, kann man den `===`-Operator verwenden:

```
(1..10) === 5           # => true
(1..10) === 15          # => false
(1..10) === 3.14159     # => true
('a'..'j') === 'c'      # => true
('a'..'j') === 'z'      # => false
```

6.1 Standardtypen

Ranges

```
car_age = gets.to_f # angenommen, der Wert ist 5.2
case car_age
when 0...1
    puts "Mmm.. new car smell"
when 1...3
    puts "Nice and new"
when 3...6
    puts "Reliable but slightly dinged"
when 6...10
    puts "Can be a struggle"
when 10...30
    puts "Clunker"
else
    puts "Vintage gem"
end
```

Reliable but slightly dinged

6.1 Standardtypen

Ranges

```
car_age = gets.to_f # angenommen, der Wert ist 5.2
case car_age
when 0..0
    puts "Mmm.. new car smell"
when 1..2
    puts "Nice and new"
when 3..5
    puts "Reliable but slightly dinged"
when 6..9
    puts "Can be a struggle"
when 10..29
    puts "Clunker"
else
    puts "Vintage gem"
end
```

Vintage gem

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 Ausdrucke 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 Ausdrück 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.

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

show_regexp("this is\nthe time", /\bis/)    # => this ->is<-\nthe time
show_regexp("this is\nthe time", /\Bis/)    # => th->is<- is\nthe time
```

6.2 Reguläre Ausdrücke

Komplexere Muster
Zeichenklassen

In regulären Ausdrücken wird eine Zeichenklassen durch in eckige Klammern gesetzte Zeichen bezeichnet: [z_1 ... 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 können z_1-z_n Zeichenbereiche ausgewählt werden.

```
show_regexp('Price $12.', /[aeiou]/)        # => Pr->i<-ce $12.
show_regexp('Price $12.', /[\s]/)           # => Price-> <-$12.
show_regexp('Price $12.', /[$.]/)           # => Price ->$<-12.

a = 'see [The PickAxe-page 123]'
show_regexp(a, /[A-F]/)                     # => see [The Pick->A<-xe-page 123]
show_regexp(a, /[A-Fa-f]/)                  # => s->e<-e [The PickAxe-page 123]
show_regexp(a, /[0-9]/)                     # => see [The PickAxe-page ->1<-23]
show_regexp(a, /[0-9][0-9]/)                # => see [The PickAxe-page ->12<-3]
```

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 POSSIX-Zeichenklassenbezeichner verwenden:

```
show_regexp('Price $12.', /[^A-Z]/)          # => P->r<-ice $12.
show_regexp('Price $12.', /[^\w]/)           # => Price-> <-$12.
show_regexp('Price $12.', /[a-z][^a-z]/)     # => Pric->e <-$12.

show_regexp('Price $12.', /[aeiou]/)         # => Pr->i<-ce $12.
show_regexp('Price $12.', /[[:digit:]]/)     # => Price $->1<-2.
show_regexp('Price $12.', /[[:space:]]/)     # => Price-> <-$12.
show_regexp('Price $12.', /[[:^alpha:]]/)    # => Price-> <-$12.
show_regexp('Price $12.', /[[:punct:]aeiou]/)  # => Pr->i<-ce $12.
```

6.2 Reguläre Ausdrücke

Komplexere Muster
Zeichenklassen

POSIX Character Classes (Unicode)

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

6.2 Reguläre Ausdrücke

Komplexere Muster
Zeichenklassen

Sequence	As […]	Meaning (Unicode)
\d	[0-9]	Decimal digit character (*Decimal_Number*)
\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 (+ *Line_Separator*)
\S	[^␣\t\r\n\f]	Any character except whitespace
\w	[A-Za-z0-9_]	Word character (+ *Connector_Punctuation*, *Letter*, *Mark*, and *Number*)
\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"
```