Inhaltsverzeichnis

1 Grundlagen

1.1 Code Beispiele

Die im Text abgebildeten Code Beispiele sind in diesem PDF-Dokument angeheftet.

Diese können unter Linux mit Okular oder Acrobat Reader eingesehen und gespeichert werden. Unter Windows drücken Sie im PDF-XChange Viewer Ctrl-Shift-A, um die Anhänge sichtbar zu machen.

1.2 Hello World!

Das erste Programm soll nichts weiter als "Hello World!" auf der Konsole ausgeben:

Listing 1 `hello.rb`

```
# Dies ist ein Kommentar und bewirkt gar nichts

# Schreibt Hello World! auf die Konsole
puts "Hello World!"

# Wartet bis Return gedrueckt wurde
gets
```

1.3 Ein- und Ausgabe

Mit `puts` kann etwas ausgegeben werden, mit `gets` kann etwas gelesen werden.

Listing 2 `eingabe.rb`

```
puts "Bitte Namen eingeben"
name = gets
puts "Hallo "+name+"!"
puts "#{name} ist aber ein toller Name!"
puts "\n[Return]!"
gets
```

Wenn Sie diesen Code ausprobieren, wird er nicht ganz Ihren Erwartungen entsprechen.

1.3.1 Weniger oder mehr Zeilenumbrüche

Der Befehl `puts` macht immer eine neue Zeile. Der Befehl `print` hingegen nicht.

Der Befehl `gets` liest ebenfalls den Zeilenumbruch mit ein. Verwenden Sie `gets.chomp`, um die neue Zeile abzuschneiden.

Die Zeichenkette `"\n"` gibt einen Zeilenumbruch aus.

1.3.2 Variablen in Zeichenketten

Variablen, die Zeichenketten enthalten, können entweder mit `+` mit Zeichenketten verknüpft werden, oder auch innerhalb der Zeichenkette mit `#{variablenname}` geschrieben werden. Mit diesem Syntax können sogar komplizierte Befehle (anstelle des Variablennamens) innerhalb von Strings geschrieben werden. Der Vorteil der

letzteren Variante ist, dass sie für Variablen beliebigen Typs funktioniert.

Aufgabe 1: Input / Output Verschönern Sie das Programm `eingabe.rb`

1.4 Eingabe von Zahlen

Wird `.to_i` hinter einen **String** (Zeichenkette) gesetzt, wird diese in eine **Ganzzahl** (integer) umgewandelt. Man kann `.to_i` auch hinter einen Befehl setzen, der einen string zurückgibt.

Listing 3 `zahleingabe.rb`

```
print "Bitte eine Zahl : "
z = gets.to_i
q = z*z
puts "#{z} quadriert ergibt #{q}\n\n[RETURN]"
gets
```

Mit `.to_f` wird ein Objekt (falls möglich) in eine Dezimalzahl (mit eventuellen Nachkommastellen) umgewandelt. Dies ist z.B. nötig, wenn das Resultat einer Division Nachkommastellen haben soll. Dazu muss die Umwandlung eines Operanden aber *vor der Division* erfolgen!

Aufgabe 2: Rechnen Schreiben Sie ein Programm, das zwei Zahlen einliest und dann deren Summe, Differenz, Produkt und Quotient ausgibt. Probieren Sie Ihr Programm mehrmals aus, und korrigieren Sie es.

1.5 If-Abfrage

Studieren Sie folgendes Programm und probieren Sie es aus:

Listing 4 `if.rb`

```
print "Ihr Name : "
name = gets.chomp
print "Sind sie (m)aennlich oder (w)eiblich? "
g = gets.chomp
puts "\n\n\n"
if (g == "m")
   anrede = "Sehr geehrter Herr #{name}"
elsif (g == "w")
   anrede = "Sehr geehrte Frau #{name}"
else
   anrede = "Sehr geehrtes Tastaturgenie"
end

puts anrede
puts
puts "Freundliche Gruesse!\n\n\n\n[RETURN]"
gets
```

- Gewöhnen Sie sich an, den Code innerhalb eines `if` **immer einzurücken**.
- Vergleiche werden mit einem doppelten Gleich `==` durchgeführt. Das einfache Gleich `=` dient der Zuweisung

von Variablen. Ungleichheit wird mit `!=` überprüft.

- Grösser und kleiner wird mit `>` und `<` geschrieben.
- Grösser gleich und kleiner gleich wird mit `>=` und `<=` geschrieben.
- Nach `elsif` kommt eine weitere Bedingung.
- Der `else`-Block wird ausgeführt, wenn alle obigen Bedingungen **nicht** erfüllt sind.
- Der `elsif`- und der `else`-Block können auch weggelassen werden.
- Jedes `if` muss schlussendlich mit einem `end` abgeschlossen werden.

Es ist möglich, mehrere Bedingungen auf einmal zu testen, indem man sie mit `and` oder `or` verknüpft. Z.B. `if (a<0 or a>n)`. Die Klammern sind hier eigentlich nicht nötig, in Sprachen wie C, Perl und PHP aber schon.
Eine Bedingung kann auf Falschheit geprüft werden. Entweder mit `not`. Z.B. `if not (a>= and a<=n)`. Oder mit `unless` an Stelle von `if`.

Aufgabe 3: Teilbar? Schreiben Sie ein Programm, das zwei Zahlen einliest und entscheidet, ob die erste durch die zweite teilbar ist oder nicht.
Hinweis: `%` ist der Modulooperator, der den Rest der Ganzzahldivision berechnet. Bsp: `12 % 5` ist gleich 2.

1.6 For- und Whileloops

Sollen Codeteile wiederholt werden, bieten sich Loops (Schleifen) an. In Ruby gibt es drei Typen: `for`, `while` und `until` Loops.

1.6.1 For-Loop

Listing 5 `loop1.rb`

```
print "Quadratzahlen bis : "
z = gets.to_i
for i in 1..z
   puts "#{i}*#{i} = #{i*i}"
end
puts "\n\n\n[RETURN]"
gets
```

Der Ausdruck `1..z` ist ein **Range** (Bereich) und enthält alle ganzen Zahlen von der unteren Grenze bis zur oberen Grenze *inklusive*. Der Ausdruck `a...b` stellt die ganzen Zahlen von `a` bis und mit `b-1` dar.

Aufgabe 4: Grenzen setzen Bauen Sie eine Abfrage ins obige Programm, so dass `z` auf 100 begrenzt wird und eine Meldung ausgegeben wird, sollte ein grösseres `z` eingeben werden.

Aufgabe 5: Teiler Schreiben Sie ein Programm, das eine Zahl einliest und alle Teiler ausgibt.

Aufgabe 6: Primzahl Schreiben Sie ein Programm, das eine Zahl einliest und entscheidet, ob es eine Primzahl ist oder nicht.

1.6.2 while-end, begin-end-while und begin-end-until

Der Befehl `while` wird mit "so lange als" übersetzt. D.h. es wird so lange wiederholt, wie die Bedingung wahr ist.

Listing 6 `loop2.rb`

```
antwort = "y"
while (antwort == "y")
  print "Nochmals? [y/n] : "
  antwort = gets.chomp
end
puts "Auf Wiedersehen!\n\n\n[RETURN]"
gets
```

Etwas intuitiver ist der `begin end until` Loop. Es wird wiederholt, bis etwas wahr ist.

Listing 7 `while-until.rb`

```
a = 6
begin
  puts a*a
  a=a+1
end while a<10
puts "Und jetzt Wurzeln"
begin
  puts Math.sqrt(a)
  a=a-1
end until a==5
```

1.6.3 Der `.step` Loop

Diese Art von Loop erlaubt es zusätzlich die Schrittweite der Laufvariable festzulegen:

```
startwert.step(oberegrenze, schrittweite)
```

Die einstelligen ungeraden Zahlen können wie folgt ausgegeben werden:

```
1.step(9,2) { |i| puts i }
```

Die Zahlen zwischen 0 und 1 in Zehntelschritten:

```
0.step(1,0.1) { |i| puts i }
```

1.6.4 Der `.times` Loop und Zufallszahlen

Eine Zufallszahl kann mit dem Befehl `rand` erzeugt werden. Ohne Argument erhält man eine Gleitkommazahl zwischen 0 und 1. Mit einem Argument, das als Ganzzahl interpretiert wird, erhält man zufällige Ganzzahlen von 0 bis zum Argument -1.

Soll eine oder mehrere Anweisungungen mehrmals wiederholt werden, bietet sich der `.times`-Loop an.

Listing 8 `loop-random.rb`

```
puts "10 üWrfelzahlen:\n\n"
10.times { puts rand(6)+1 }
```

Benötigt man zusätzlich die Laufvariable, kann das z.B. wie folgt erreicht werden:

Listing 9 `loop2-random.rb`

```
print "Die Lottozahlen:\n\n"
6.times {|i| puts "Zahl #{i+1}: #{rand(45)+1}"}
```

Man könnte obige Beispiel natürlich genau so gut mit einem For-Loop lösen.

Aufgabe 7: Zahlenraten Der Benutzer soll eine Zahl zwischen 1 und 100 erraten. Das Programm soll jeweils die Anzahl Versuche angeben und ob der Benutzer zu gross oder zu klein geraten hat.

Aufgabe 8: Rechentrainer Schreiben Sie einen Kopfrechentrainer. Es sind verschiedene Aufgabentypen und Schwierigkeitsstufen vorstellbar.

1.7 Strings (Zeichenketten)

Strings werden zwischen Anführungszeichen `"` (dann werden Dinge wie `#{12*13}` ausgewertet und in die Zeichenkette eingefügt) oder *geraden* Apostrophs `'` geschrieben (dann wird nichts interpretiert).

1.7.1 Operationen auf Strings

`+` Mit dem Pluszeichen können Strings aneinandergehängt werden. Z.B. `a = "Hello"+ " " + "Welt!"`

`.length` Anzahl Bytes im String (entspricht normalerweise der Anzahl Buchstaben). Z.B. ist `"hallo".length` gleich 5.

`[pos, anz]` Liefert den Substring der Länge `anz`, beginnend an der Position `pos`, wobei die erste Position die Position Null ist. Z.B. ist `"abcde"[1,2]` gleich `"bc"`.

`[pos]` liefert den Wert (Ganzzahl) vom Byte an der Stelle `pos`.

`.chr` Kann einer Zahl angehängt werden und liefert den Buchstaben mit dem entsprechenden ASCII-Code. Z.B. ist `65.chr` gleich `"A"`.

Listing 10 `stringbsp.rb`

```
a = "mein ganz toller String"
puts "Original: ->#{a}<-"
puts "Der String ist #{a.length} Zeichen lang"
puts "a[5,9]: (Startpos 5, Laenge 9) ->#{a[5,9]}<-"
a = a + " ist lang!"
puts "capitalize: ->#{a.capitalize}<-"
puts "upcase: ->#{a.upcase}<-"
```

Aufgabe 9: Text-Quadrat Schreiben Sie ein Programm, das einen String vom Benutzer einliest und daraus folgenden Output generiert (z.B. für den String `"Linux"`):

```
L i n u x
i n u x L
n u x L i
u x L i n
x L i n u
```

Aufgabe 10: Text-Quadrat 2 Es soll nun folgendes Quadrat geniert werden:

```
L i n u x
i L i n u
n i L i n
u n i L i
x u n i L
```

Aufgabe 11: String umdrehen Schreiben Sie ein Programm, das vom Benutzer einen String einliest und den String dann rückwärts ausgibt.

Aufgabe 12: String umdrehen in Ruby Studieren Sie Ruby-Dokumentation der Klasse `String` und lösen Sie die Aufgabe 11 in einer Zeile.

Aufgabe 13: Zahlen zählen Es soll eine Folge von Zeichenketten aus Ziffern erzeugt werden, und zwar wie folgt: Man startet mit `"1"`. Die nächste ist `"11"` (eine Eins). Darauf folgt `"21"` (zwei Einsen), dann `"1211"` (eine Zwei und eine Eins), dann `"111221"` (eine Eins, eine Zwei und zwei Einsen), etc.

a) Programmieren Sie ein Programm, das diese Folge erzeugen kann.

b) Beweisen Sie, dass wenn man mit einer Eins startet, dass nie eine '4' vorkommen kann.

c) Beweisen Sie, dass wenn man mit einer Eins startet, dass nie '333' vorkommen kann.

1.8 Funktionen

Dinge, die man oft wieder braucht, sollten in Funktionen geschrieben werden. Das hat mehrere Vorteile: Der Code wird übersichtlicher, kürzer und ist einfacher zu "debuggen".

Aufgabe 14: Bug Warum nennt man einen Computerfehler "bug"?

Listing 11 `fertig.rb`

```ruby
# Funktion fertig (warten auf Return-Taste)
def fertig
    puts "Bitte [RETURN] druecken!"
    gets
end

# Funktion zahl einlesen und zurueckgeben
def zahl
    print "Bitte eine Zahl eingaben : "
    return gets.to_i
end

puts "Hello"
fertig
z = zahl()
puts "#{z}^2 = #{z*z}"
fertig()
```

Funktionen können auch Parameter übergeben werden.

Listing 12 `quadratgleichung.rb`

```
def loesung(a,b,c)
  d = b*b-4*a*c
  if (d<0)
    return []     # Leere Tabelle
  elsif (d==0)
    return [-b.to_f/a]    # Tabelle mit einem Eintrag
  else
    return [(-b-Math.sqrt(d))/(2*a), (-b+Math.sqrt(d))/(2*a)]
  end
end

def zahl(s)
  print "Bitte Koeffizient #{s} : "
  return gets.to_f
end

print "Quadratische Gleichung  a x^2 + b x + c = 0\n\n"
a = zahl("a")
b = zahl("b")
c = zahl("c")
l = loesung(a,b,c)
puts "Es gibt #{l.size} Loesung(en) von #{a}x^2 + #{b}x + #{c} = 0"
puts l
```

Aufgabe 15: ggT und kgV Programmieren Sie den Algorithmus von Euklid für die ggT-Berechnung, verpacken Sie diese in eine Funktion und programmieren Sie mit deren Hilfe eine Funktion für die kgV-Berechnung.

Aufgabe 16: Buchstaben vertauschen Sie kenenn sciher die Tsatache, dass man Txete ncoh zcmileih gut lseen knan, oohbwl die Beshcabtun im Inernn jeeds Wetors vlöilg zlfäluüg eersinhcen. Sichrbeen Sie enie Fktinoun, die dseie Tairasrnoomftn in eienm Txet vmromint.

Aufgabe 17: Caesar Verschlüsselung Schreiben Sie eine Funktion `caesar(text,offset)`, die die Buchstaben von `text` um `offset` Buchstaben im Alphabet verschiebt. Entschlüsseln Sie damit folgende Botschaft: `"INJLFQQNJWLWJNKJSFS"`.

1.9 Arrays

Arrays sind Tabellen, in denen beliebige Dinge (auch Arrays!) gespeichert werden können. Es gibt eine Fülle von praktischen Methoden, die auf Arrays angewandt werden können.

Listing 13 `array.rb`

```
a = [1, 3.45, "hello", [111,222,333]]
puts "a.size = #{a.size}"
puts "a[1] ist das ZWEITE Element #{a[1]}"
puts "a[3][2] = #{a[3][2]}"
puts "l=a.pop gibt das letzte Element üzurck und entfernt es."
l=a.pop
puts "l.size = #{l.size}"
puts "l.each { |e| puts e*100 }"
l.each { |e| puts e*100 }
b = Array.new(5) {|i| 100-i*i}
puts "Einfaches puts, gibt jedes Element auf einer neuen Zeile aus."
puts b
puts "Mit .inspect wird Ruby-Syntax auf einer Zeile ausgegeben."
puts b.inspect
b.sort!
puts "Jetzt ist b aufsteigend sortiert:"
puts b.inspect
c=[]            # Ein leeres Array
c=Array.new     # Auch ein leeres Array
c.push(44)      # Element hinten anhaengen
c.push("Hallo")
puts c.inspect
```

Aufgabe 18: Fibonacci-Zahlen Schreiben Sie eine Funktion, die eine Tabelle mit den ersten n Fibonacci-Zahlen zurückgibt. Dier ersten zwei Zahlen sind gleich 1, jede nachfolgende ist die Summe ihrer zwei Vorgänger.

Aufgabe 19: Pascal-Dreieck Schreiben Sie eine Funktion, die die ersten n Zeilen des Pascal-Dreiecks berechnet.

1.10 Hash-Tabellen

Hash-Tabellen funktionieren ähnlich wie Arrays, ausser dass nicht nur Zahlen sondern beliebige Ruby-Objekte als Inidizies verwendet werden können. Es wird allerdings empfohlen, nur Strings als Indizies zu verwenden (oder sich sonst einmal seriös mit den Innereien von Ruby auseinander zu setzen, weil unerwartete Effekte auftreten, wenn Schlüssel-Objekte nachträglich verändert werden).

Listing 14 `hash1.rb`

```
h = {"bla"=>"wort", "hihi" => [1,2]}
puts "h ist #{h.inspect}"
puts "h hat die üSchlssel #{h.keys.inspect}"
puts "h hat die Werte #{h.values.inspect}"
puts "Zum üSchlssel bla ögehrt #{h['bla']}"
puts "Gibt es üSchlssel BLA? #{h.key?('BLA')}"
puts "Gibt es üSchlssel hihi? #{h.key?('hihi')}"
h.delete("bla")
puts "'bla' ögelscht: h=#{h.inspect}"
h["soso"]="was is?"
puts "'soso' gesetzt: h=#{h.inspect}"
h.each_pair{|schluessel,wert|
  puts "Zum üSchlssel ->#{schluessel}<- ögehrt der Wert ->#{wert}"
}
hh = Hash.new    # Leerer Hash
hhh = {}         # Auch ein leerer Hash
```

1.11 Blocks in Ruby

Blocks in Ruby werden entweder zwischen `do` und `end`, oder zwischen geschweiften Klammen geschrieben. Ein Block folgt immer auf den Aufruf einer Methode (oder Funktion). Achtung: Ein Block wird nicht zwingend dann ausgeführt, wenn er definiert wird (siehe weiter unten).

Ein Block wie eine Funktion ohne Namen, die aber auf Variablen zugreifen kann, die um den Block herum definiert sind. Übergebene Variablen werden am Anfang des Blocks zwischen `|` (pipes) aufgeführt.

Das Resultat eines Blocks ist jeweils der letzte berechnete Wert.

1.11.1 Beispiele von Methoden, die Blocks entgegennehmen

Hinweis: Der Operator `<=>` liefert -1, 0 oder 1, je nachdem, ob der linke Operand kleiner, gleich oder grösser als der rechte Operand ist.

Listing 15 `blocksbeispiel.rb`

```
# Der Block ist hier eine Funktion, die aus dem Index
# (hier in die Variable i gespeichert), den Wert des
# Array-Elements berechnet.
a = Array.new(10) { |i|  i*i % 11}
puts a.inspect
# Array aus Strings (einzelne Worte)
w = %w(null eins zwei drei vier fuenf sechs sieben acht)
# Sortieren, Standard alphabetisch
puts w.sort.inspect
# Sortieren nach Wortlaenge
# Der Block gibt an, wie zwei allgemeine Elemente des Arrays
# verglichen werden sollen.
puts (w.sort {|a,b| a.length <=> b.length}).inspect
# Sortieren nach zweitem Buchstaben
puts (w.sort {|a,b| a[1] <=> b[1] }).inspect
# Ruby beim Sortieren zuschauen
w.sort {|a,b|
  puts "Vergleiche #{a} mit #{b}"
  a <=> b  #Das Resultat des Vergleiches üzurckgeben
}
```

1.11.2 Eigene Methoden, die Blocks benutzen

"For-Schlaufe" über Quadratzahlen:

Listing 16 `squaresblock.rb`

```
class Fixnum
  def squares
    for i in 0...self
      yield i*i  # Hier wird der Block mit i*i aufgerufen
    end
  end
end
# Zehn Quadratzahlen ausgeben
10.squares {|q|  puts q }
```

Blocks werden wie Variablen übergeben und können gespeichert und zu beliebiger Zeit aufgerufen werden. Das wird z.B. in FXRuby verwendet, um anzugeben, was z.B. bei einem Mausklick zu geschehen hat.

Listing 17 `blocksave.rb`

```
class Hund
  def initialize(name, &aktion)
    @aktion = aktion
    @name = name
  end
  def bellen
    puts "#{@name}: Wau wau!"
    @aktion.call
  end
end

a = Hund.new("Bello") { puts "und wau!" }
b = Hund.new("Hasso") { a.bellen }
# Und jetzt lassen wir Hasso bellen
b.bellen
# a ueberschreiben
a = Hund.new("Schlumpi") { puts "  und w... hust keuch..."}
# Hasso soll nochmals bellen
b.bellen
# Aber wenn a kein Hund mehr ist, gibt es ein Problem
a = "Ich bin kein Hund, sondern ein String"
b.bellen
```

2 Bits und Bytes

2.1 Ganzzahlen `Fixnum` und `Bignum`

`Fixnum` und `Bignum` sind Ganzzahltypen von Ruby (beides Unterklassen von `Integer`), wobei `Fixnum` dem Integer-Typs des Systems entspricht (heute 32 oder 64 Bits) und `Bignum` eine beliebig grosse Zahl enthalten kann.

Starten Sie die Ruby-Shell (Windows: fxri, zu finden im Menü Start, andere Systeme: irb). Probieren Sie folgende Befehle aus und finden Sie heraus (mit den Ihnen zu Verfügung stehenden technischen Hilfsmitteln), was die Befehle genau bewirken. Experimentieren Sie auch mit negativen oder sehr grossen Zahlen (z.B. `7**77`).

```
a=21
a.class
a.size
a.to_s(2)
a.to_s(16)
31.downto(0) { |i| print a[i] }
a << 3
a & 7
a ^ 1
a | 2
~a
```

Aufgabe 20: XOR-Verschlüsselung Schreiben Sie ein Programm, das eine Zeichenkette mit Hilfe des XOR-Operators verschlüsseln und entschlüsseln kann.

Aufgabe 21: Unsicherheit der XOR-Verschlüsselung Jeder noch so komplizierte (konstante) Schlüssel kann sofort berechnet werden, wenn man den Originaltext und den verschlüsselten Text kennt. Erklären Sie wie!

Versuchen Sie eine Methode zu finden, wie Sie den Schlüssel aus vielen verschlüsselten Texten effizient erraten können.

3 Regular Expressions

Eine *regular expression* ist ein mächtiges Werkzeug, um Zeichenketten zu analysieren. Primär geht es darum, festzustellen, ob eine Zeichenkette einem bestimmten Muster entspricht.

3.1 Wichtigste Operatoren

Operator	Erklärung
.	Irgend ein Buchstabe (genau einer)
*	Null oder mehr (maximal) der Regular Expression davor
*?	Null oder mehr (minimal) der Regular Expression davor
+	Ein oder mehr (maximal) der Regular Expression davor
?	Null oder eine der Regular Expression davor
$\{n\}$	Genau n der R.E. davor
$\{n,m\}$	Zwischen n und m...
$\{n,\}$	n oder mehr...
[]	genau einer der aufgeführten Buchstaben (Bsp. [aeiou])
[^]	genau ein Buchstaben, der nicht aufgeführt ist (Bsp. [^\"])
\d	Eine Ziffer, wie [0-9]
\s	Eine Leerschlag, Tab oder Newline
\w	Ein Buchstabe, Zahl oder Underscore, wie [A-Za-z0-9_]
()	Ein Gruppe (wird auch in \1, \2 usw. gespeichert)
|	Das, was vor dem Pipe steht, oder das, was danach steht.
^	Anfang einer Linie (bzw. String, falls /m)
$	Ende einer Linie (bzw. String, falls /m)
\	Buchstaben danach schützen (z.B. den Punkt)

3.1.1 Modifikatoren

Am Ende einer Regular Expression kann noch ein Modifikator angehängt werden. Z.B. `i`, der bewirkt, dass Gross-/Kleinschreibung ignoriert wird. `m` bewirkt, dass der Punkt auch von der Newline (\n) "gematcht" wird.

3.1.2 Beispiele

Reg Exp	Matches	non-Matches
/b.a/	bla, bbbaaa, 12b3a45, abba	Bla, ba, abababa
/^0x[0-9a-f]+$/	0x12ff, 0xa3000	0x12FF, 00x12, face, 0x
/[0-9]+\.[0-9]+/	1.23, asdf3.65asdf	0., .123, 123.a123

3.2 Einen String auf einen Match überprüfen

Listing 18 `regexp-simple.rb`

```
s = "Die Werte x=12.45 und y=5 bla bla bla"
if s =~ /bla.*bla/
  puts "2x bla im String!"
end
if s =~ /x=(\d+\.?\d*)/
  # Gruppen werden in $1, $2, usw gespeichert
  puts "X-Koordinate ist #{$1}"
end
```

3.3 Gefundene Passagen ausgeben

Die einfachste Variante ist die String-Methode `scan` zu verwenden. Es wird ein Array von allen Matches ausgegeben. Werden Gruppen verwendet, wird eine Array von Arrays ausgegeben, wobei jedes Unterarry die Matches für die einzelnen Gruppen enthält.

Listing 19 `regexpscan.rb`

```
text = "Wie viele 'e' gibt es in diesem Text?"
puts text.scan(/e/).size
# ergibt 8

text = "Bello bellt doppelt"
res = text.scan(/(.)\1/)
puts "Doppelte Buchstaben: #{res.inspect}"
# ergibt [["l"], ["l"], ["p"]]

text = "Am 13.05.2003 und 18.06.2007"
res = text.scan(/(\d\d)\.(\d\d)\.(\d\d\d\d)/)
puts res.inspect
# ergibt [["13", "05", "2003"], ["18", "06", "2007"]]
```

Soll nur ein Match ausgegeben werden und man interessiert sicht nicht für die Matches der Gruppen, kann die String-Methode `match` benutzt werden die ein `MatchData`-Objekt zurückgibt. Das Element 0 enthält dann den Match. Z.B liefert `"Bello".match(/(.)\1/)[0]` den String `"ll"`.

Aufgabe 22: Bilder einer Webseite Speichern Sie den Quellcode einer Webseite und finden Sie alle Bilder darin. Hinweis: Bilder können z.B. mit `<img src="adresse">` eingebunden werden. Ziel ist es, eine Regular Expression zu erstellen, die den Job erledigt.

Aufgabe 23: News der Kanti Schreiben Sie ein Programm, das mit einer Regular Expression aus dem Quellcode von `http://www.kanti-wohlen.ch/h2.php` die News herausschält.

3.4 Ersetzungen

Mit der String-Methode `gsub` können Matches durch einen String ersetzt werden. Gruppen sind auch hier erlaubt und können für die Ersetzung gebraucht werden.

Listing 20 `regexpgsub.rb`

```
text = "Wie viele 'e' gibt es in diesem Text?"
res = text.gsub(/e/,"#")
puts res
# ergibt "Wi# vi#l# '#' gibt #s in di#s#m T#xt?"

text = "Ganz kurz normal"
res = text.gsub(/(.)/,"\\1 ")
puts res
# ergibt "G a n z   k u r z   n o r m a l "

text = "Am 13.05.2003 und 18.06.2007"
res = text.gsub(/(\d\d)\.(\d\d)\.(\d\d\d\d)/,"\\2-\\1-\\3")
puts res
# ergibt "Am 05-13-2003 und 06-18-2007"
```

Aufgabe 24: Alte Rechtschreibung Schreiben Sie eine Regular Expression (mit `gsub`), die alle Dreifachkonsonanten vor Vokalen in einem Text in doppelte verwandelt.

4 Lösungen

4.1 Lösung zu Aufgabe 1

Listing 21 `eingabe2.rb`

```
print "Ihr Name = "
name = gets.chomp
puts "\n Hallo #{name}!"
puts "#{name} ist aber ein toller Name!\n"
puts "[RETURN]"
```

4.2 Lösung zu Aufgabe 2

Listing 22 `zahleingabe2.rb`

```
print "1. Zahl : "
a = gets.to_i
print "2. Zahl : "
b = gets.to_i
puts "#{a} + #{b} = #{a+b}"
puts "#{a} - #{b} = #{a-b}"
puts "#{a} * #{b} = #{a*b}"
puts "#{a} / #{b} = #{a/b}     (Ganzzahldivision)"
puts "#{a} / #{b} = #{a.to_f/b} (Gleitkommazahldivision)"
puts "\n\n[RETURN]"
gets
```

4.3 Lösung zu Aufgabe 3

Listing 23 `teilbar.rb`

```
print "Zahl : "
z = gets.to_i
print "Teiler : "
t = gets.to_i
if (z % t == 0)
   puts "#{z} ist durch #{t} teilbar (ergibt #{z/t})"
else
   puts "#{z} ist nicht durch #{t} teilbar (ergibt den Rest #{z%t})"
end
puts "\n\n\n[RETURN]"
gets
```

4.4 Lösung zu Aufgabe 5

Listing 24 `teiler.rb`

```
print "Zahl : "
z = gets.to_i
for i in 1..z
   if (z % i == 0)
      puts "#{i} ist ein Teiler von #{z}"
   end
end
# Und jetzt noch der Ruby-Hardcore Einzeiler ;-)
puts "Teiler von #{z} : #{((1..z).select {|i| z % i == 0}).join(', ')}"
```

4.5 Lösung zu Aufgabe 6

Listing 25 `primzahl.rb`

```
print "Zahl : "
z = gets.to_i
prim = true
for i in 2..Math.sqrt(z)
    if (z % i == 0)
        puts "#{z} ist durch #{i} teilbar! Also keine Primzahl"
        prim = false
        break
    end
end
if (prim)
    puts "#{z} ist eine Primzahl!"
end
puts "\n[RETURN]"
gets
```

4.6 Lösung zu Aufgabe 7

Listing 26 `raten.rb`

```
puts "Erraten Sie ein Zahl zwischen 1 und 100!"
geheim = rand(100)+1
begin
  print "Raten Sie : "
  raten = gets.to_i
  if (raten < geheim)
    puts "öGrsser!"
  elsif (raten > geheim)
    puts "Kleiner!"
  end
end until (geheim == raten)
puts "Sehr gut!"
```

4.7 Lösung zu Aufgabe 9

Listing 27 `textquadrat.rb`

```
print "Ein Wort bitte : "
a = gets.chomp
a.length.times {
  puts a.split('').join(' ')
  a = a[1,a.length-1]+a[0,1]
}
```

4.8 Lösung zu Aufgabe 10

Listing 28 `textquadrat2.rb`

```
print "Ein Wort bitte : "
a = gets.chomp
a.length.times {|linie|
  a.length.times {|pos|
    print a[(linie-pos).abs,1] + " "
  }
  print "\n"
}
```

4.9 Lösung zu Aufgabe 13

Listing 29 `sly.rb`

```
kette = "1"
puts kette
12.times {
  neu = ""
  pos = 0
  wieviel = 0
  was = kette[pos,1]
  while (pos<kette.length)
    if kette[pos,1] == was
      wieviel += 1
    else
      neu += wieviel.to_s + was
      was = kette[pos,1]
      wieviel = 1
    end
    pos += 1
  end
  neu += wieviel.to_s + was
  kette = neu
  puts kette
}
```

4.10 Lösung zu Aufgabe 16

Listing 30 `vertauschen.rb`

```
def vertausch(s)
  s.split(' ').collect {|w|
    w[0,1] +
    w[1..-2].split('').sort_by { rand }.join('') +
    w[-1,1]
  }.join(' ')
end

while ((s=gets.chomp)!="")
  puts vertausch(s)
end
```

4.11 Lösung zu Aufgabe 17

Listing 31 `caesar.rb`

```
def caesar(text,offset)
  offset = offset % 26   # In positive Zahl in 0..25 umwandeln
  res = ""               # Resultat
  text.upcase.each_byte { |c|
    if c>=?A and c<=?Z
      c = ((c-?A+offset) % 26) +?A
    end
    res += c.chr
  }
  return res
end

geheim = caesar("Die Gallier greifen an", 5)
puts geheim
nachricht = caesar(geheim, -5)
puts nachricht
```

4.12 Lösung zu Aufgabe 18

Listing 32 `fibonacci.rb`

```
def fibo(n)
   if (n<3)
      return [1,1].slice(0,n)
   end
   r = [1,1]
   (n-2).times { r += [r[-1]+r[-2]] }
   return r
end

[0,1,2,3,5,10,20].each { |i|
   puts "Die ersten #{i} Fibonaccizahlen: #{fibo(i).join(',')}"
}
```

4.13 Lösung zu Aufgabe 19

Listing 33 `pascal.rb`

```
def pascal(n)
  res = [ [1] ]  # Array im Array
  for i in 1..n
    for j in 0..i
      if j==0
        res[i] = [1]  # Neue Linie anfuegen
      elsif j==i
        res[i].push(1)  # Neues Element der Linie anhaengen
      else # Summe der beiden oberen Elemente der Linie anhaengen
        res[i].push(res[i-1][j-1]+res[i-1][j])
      end
    end
  end
  return res   # Resultat zurueckgeben und Funtion beenden
end

pascal(14).each { |linie|
  puts linie.join("\t")
}
```

4.14 Lösung zu Aufgabe 20

Listing 34 `xorcrypto.rb`

```
def xorcrypto(text)
  res = ""
  b = 17
  text.each_byte { |c|
    res += (c^b).chr
    b = (b+17) & 127    # modulo 128
  }
  return res
end

geheim = xorcrypto("XOR-üVerschlsselung ist sehr unsicher!!")
puts geheim
nachricht = xorcrypto(geheim)
puts nachricht
```