

Willkommen zur Ruby-Schulung im Linuxhotel 12.10.2026

Vorstellung

Jan Krutisch

- Freier Software-Entwickler und [Gründer](#)
- Hamburg
- Ruby/ Rails seit 2005
- Projekte in allen Formen und Farben

Jetzt Ihr

- Name
- Vorkenntnisse (Auch: Andere Programmiersprachen)
- Kursziele, ggf. Anwendungsfeld

Was uns erwartet

- 5 volle Tage
- Viel Praxis, viel Pausen
- Viele Fragen, bitte!

Das wichtigste

- Editor der Wahl
- Das Pad

Themen (Grob)

1. Tag: Einführung, Installation, Grundlagen
2. Tag: Fortsetzung, der Kern von Ruby
3. Tag: Objektorientierung
4. Tag: Die Standard-Library, Rubygems, das Web
5. Tag: This page intentionally left blank

Try Ruby & Registrierung

- <https://try.ruby-lang.org/>
- Registrierung für lab.linuxhotel.de



Ruby

Ruby

**Entwickelt von Yukihiro "Matz" Matsumoto
ab 1995**

Bekannt in Europa und USA seit ca. 2002

**Große Popularität im Westen dank Ruby on
Rails ab 2004/ 2005**

Design-Goals:

- Developer Happiness
- Von Beginn an Objektorientiert
- Ideen aus Perl, Lisp und Smalltalk

Offizielle Implementierung:

MRI ("Matz' Ruby-Interpreter")

Diverse andere Implementierungen:

JRuby (JVM)

TruffleRuby

DragonRuby

mruby

...

Ruby ist...

...objektorientiert

...elegant

...lesbar

...schön?

Beispiele

```
3.times do  
  print "Welcome"  
end
```

```
20.days.from_now  
# (Nicht wirklich ruby)
```

Installation

Viele Wege führen nach Rom

Windows

<https://rubyinstaller.org/>

Mac

homebrew

Linux

Paketmanager

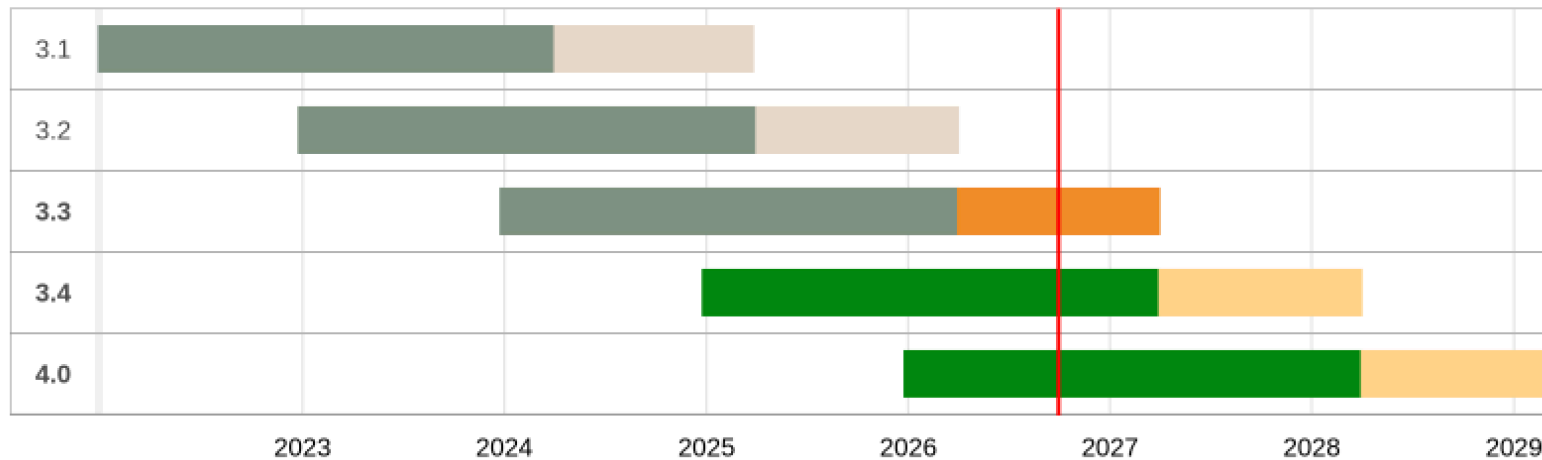
Aber!

Paket ruby

- **resolute (26.04 LTS)** (interpreters): Interpreter of object-oriented scripting language Ruby (default version)
1:3.3build1: amd64 arm64 armhf i386 ppc64el riscv64 s390x

VS.

Ruby Lifecycle Timelines



Generic Version Managers

asdf, mise, proto, aqua

Ruby Version Managers

rbenv, rvm

New kid on the block

rv

Für heute:

```
mise install ruby@3.4.10  
mise use ruby@3.4.10
```

hello_world.rb

```
#!/usr/bin/env ruby  
puts "Hello World"
```

Anatomie eines Ruby-Skripts

- "shebang" `#!/usr/bin/env ruby`
- Einrückung: 2 Leerzeichen
- Nicht vergessen: `chmod +x`
- Kommentare starten mit # (wie bash, python etc.)

Über Ruby aufrufen

(Dann kein shebang nötig)

```
$ ruby script.rb
```

Zeichenkodierung

- UTF-8 hat gewonnen (Code ist UTF-8 per default, seit Ruby 2.0.0)
- Strings haben eigenes encoding (UTF-8 in meisten Fällen default)
- Umfangreiche Funktionen für Transcoding

Syntax

Eine Anweisung pro Zeile, ohne zwingendes Semikolon

```
puts "Eins"  
puts "Zwei"
```

Semikolons können verwendet werden, sehr unüblich

```
puts "Eins"; puts "Zwei"
```

IRB

```
$ irb
```



Erste Schritte

- Schreibe ein Programm, das auf drei getrennten Zeilen auf der Konsole eine Ausgabe macht.
- Führe das Programm testweise in der Konsole aus.
- Lade Dein Programm in der IRB mit Hilfe von `require` hinzu. Was passiert?

Variablen

Lokale Variable

```
a = 1
```

Variablen

- Muss mit kleinem Buchstaben anfangen (Ausnahmen!)
- Üblich: snake_case
- Möglich: Fast alles ( = "lie")
- Keine Deklaration
- Keine Typdeklaration

Konstanten

```
EPOCH_YEAR = 1970
```

Konstanten

- Sind nicht konstant
- Müssen mit großem Buchstaben anfangen
- Üblich: SNAKE_CASE
- Auch hier: Emojis sind möglich

Globale Variablen

```
$foo = "bar"
```

Globale Variablen

- Nein
- Nein
- Nein

Datentypen

Literale

```
nil                # Nichts
1, 300_000, 3.14, 2.4e-20 # Zahlen
'hallo', "hallo", %q(hallo) # Zeichenketten
:status, :through   # Symbole
true, false         # Wahrheitswerte
10..20, 10...20, 'a'..'z' # Bereiche
/pattern/, /\d+/     # Reguläre Ausdrücke (Regexp)
```

Zeichenketten

```
a = 'aha'
b = "soso"
b = "jaja\nnundenn"
c = "->#{b}<-"
d = %Q("#{a}")
```

Symbole

- Global einmalig
- unveränderbar
- Parameter, Konfiguration, etc. etc.

Alles ist ein Objekt

```
true.class # => TrueClass  
2.class    # => Integer  
2.2.class  # => Float  
"hello".class # => String
```

Alles ist ein Objekt

- Datentypen sind Klassen
- Werte sind Objekte dieser Klassen
- Weitgehende Introspektion (`2.class.superclass # => Numeric`)

Objekte haben Methoden

```
5.next           # 6  
'hallo'.length  # 5  
'hallo'.chop    # "hall"  
:name.is_a? Symbol # true
```

Klammern um Argumente?

```
'hallo'.length()  
:name.is_a?(Symbol)
```

Typ-Umwandlung

```
23.to_s      # "23"  
"42".to_i    # 42  
"42".to_f    # 42.0  
"3.3".to_f   # 3.3  
"abc".to_i   # 0
```

Aha!

```
23.to_s(16)    # "17"  
"abc".to_i(16) # 2748
```

"Globale" Methoden

(Dokumentation: [Kernel](#))

```
puts "Hallo" # Zeile ausgeben  
a = gets     # Zeile einlesen
```



Nachfolger

Schreibt ein Programm, das den Benutzer auffordert, eine Ganzzahl einzugeben.

Anschließend soll das Programm den Nachfolger dieser Ganzzahl ausgeben.

Operatoren

```
5 + 1           # 6
5 / 2           # 2 (Achtung!)
5.0 / 2.0       # 2.5
"Hello" + " World" # "Hello World"
true && true     # true
true and false  # false
true || false   # true
false or true   # true
!true           # false
not false       # true
```


Kurzformen

Aus Operator und Zuweisung lassen sich Kurzformen bilden

```
a = a + 1  
a += 1  
a ||= 5  
a = a || 5
```

Eigene Methoden Definieren

```
def add(a, b)  
    return a + b  
end
```

Methoden

- Benennung wie lokale Variablen
- Zusätzlich gewisse Sonderzeichen erlaubt
- Beliebige Anzahl Parameter möglich

Sonderzeichen in Methodennamen

```
full? # Wahrheitswert erwartet  
reset! # Seiteneffekte, In-Place, Achtung!
```

Zuweisungen/ "Attribute"

```
car.plate=("XUF 203")  
car.plate = "XUF 203"  
puts car.plate()  
puts car.plate
```

Operatoren

```
def +(b)  
  # ...  
end
```

```
2.+(3)  
2 + 3
```

Default-Parameter

```
def add(a, b = 1)
  return a + b
end
add(3)      # 4
add(3, 5)   # 8
```

Benannte parameter ("keyword args")

```
def format_currency(amount:, currency: 'EUR')
    formatted_amount = sprintf('%.2f', amount)
    "#{formatted_amount} #{currency}"
end
format_currency(amount: 23) # 23.00 EUR
format_currency(currency: '$', amount: 6) # 6.00 $
```

Rückgabewerte

```
def add(a, b) # "Ruby Style"  
  a + b  
end
```

```
def add(a, b) # explizit  
  return a + b  
end
```



Hello World & Quadratzahlen

Schreibt eine hello-world-Methode, die einen personalisierten Gruß ausgibt. Dazu soll ein Parameter "name" übergeben werden. Anschließend soll der Gruß "Hallo <name>" ausgegeben werden, wobei <name> durch den übergebenen Namen ersetzt wird.

Schreibt eine Methode, um Quadratzahlen zu berechnen. Als Parameter soll eine Zahl übergeben werden. Rückgabewert soll das Quadrat der Zahl sein. Schreibt jeweils einen Aufruf der Methode, der die Funktionsweise testet und das Ergebnis auf der Konsole ausgibt.

Kontrollstrukturen

Fallunterscheidungen

```
if a > 1000
  puts "zu groß"
elsif a == 23
  puts "nicht erlaubter Wert"
else
  puts "korrekte Eingabe"
end
```

Kompakte version

```
puts "OK" if a == 23
```

Ternär

```
(a == 2) ? "zwei" : a.to_s
```

Unless

```
return 20 unless status == 200
```

Unless

- Eignet sich gut für "Guards"
- Bitte nicht:
 - Mit komplexen boolschen Ausdrücken (Knoten im Kopf garantiert)
 - Mit else kombinieren

Bedingungen

Beliebige Ruby-Ausdrücke können als Bedingungen benutzt werden

- `true` ist wahr
- `false` ist unwahr
- `nil` ist unwahr
- Alles andere (auch `0`, `""` etc.) ist wahr
- Kombination mit `&&` und `||` und `!`
- (Theoretisch auch mit `and` und `or` und `not`)
- Klammern nur für Präzedenz.

Alles ist ein Ausdruck

```
a = if b == 1
    'Eins'
elsif b == 2
    'Zwei'
else
    b.to_s
end
```

Und wenn dann?!?

```
a = if false  
  'Falsch'  
end  
  
# => a = nil
```

Vergleiche

```
2 == 2      # true
3 == 3.0     # true
3.equal? 3.0 # false

2 != 3      # true
3 > 2       # true
```

Fallunterscheidungen

```
case a
when 1
  'Eins'
when 2
  'Zwei'
else
  a.to_s
end
```

Fallunterscheidungen

```
case var
when Fixnum, Float
  'Zahlwert'
when String
  'Zeichenkette'
else
  'Unbekannter Datentyp'
end
```

Drei =

```
4 === 4           # true
Integer === 4     # true
2 === 2.0         # true
4 === Integer     # false
```



Taschenrechner

Schreibt ein Taschenrechnerprogramm, das den Nutzer nacheinander auffordert eine Zahl, einen Operator und dann noch eine Zahl einzugeben.

Das Programm soll anhand des Operators die passende Rechnung mit den beiden Zahlen durchführen und das Ergebnis ausgeben.

Schleifen

```
loop do  
  puts "I WON'T STOP!"  
end
```

Mehr Schleifen

```
while true  
  puts "Do something"  
end
```

```
until false  
  puts "Do something"  
end
```

Kontrolle

```
loop do
  next if something?  # nächste iteration
  break if something? # exit
  redo if something?  # iteration wiederholen
end
```



Ratespiel

Schreibt ein Zahlratespiel, das eine zufällige Zahl von 1 bis 10 (oder 100, oder ...) auswählt und den Benutzer so lange die Zahl raten lässt, bis er korrekt geraten hat.

Bei jedem Durchlauf soll angezeigt werden, um den wievielten Versuch es sich handelt. Anschließend soll der Nutzer einen Tipp abgeben können. Das Programm gibt aus, ob der Tipp korrekt war, oder ob die gesuchte Zahl kleiner oder größer ist als der abgegebene Tipp.

Ist der Tipp korrekt, beendet sich das Programm.

Andernfalls wird wieder der Versuch angezeigt und um einen neuen Tipp gebeten. Usw.

Arrays

Sortierte Liste von Werten

```
a = [1, 2, 5, 6]
a[0]                # => 1
a = ["hallo", 1, 3.8, :name]
a << 3
a.push(3)
a.pop               # => 3
a.first             # "hallo"
```

Hashes

"Assoziative Arrays"

"HashMaps"

Schlüssel/Wert-Paare

Hashes

Allgemeine Syntax

```
h = {:name => "Otto Mustermann", :strasse => "Musterstraße 22", :ort => "Musterstadt"}
```

Wenn Schlüssel Symbols sind

```
h = {name: "Otto Mustermann", strasse: "Musterstraße 22", ort: "Musterstadt" }  
h[:name] \# "Otto Mustermann"
```

Code blocks

```
loop { puts "Endlos!" }
```

```
5.times do |i|  
  puts i  
end
```

Iteratoren

```
a = [1, 2, 3]
a.each { |element| puts element }
a.each_with_index do |element, i|
  puts "#{i}. #{element}"
end
```

Iteratoren

```
h = {eins: 1, zwei: 2, drei: 3}
h.each_key { |key| puts key }
h.each { |key, value| puts "#{key}: #{value}" }
```

Select & Map

```
a = [1, 2, 3, 4, 5, 6]
a.select { |e| e > 3 }      # [4, 5, 6]
a.map { |e| e * 2 }        # [2, 4, 6, 8, 10, 12]
```

API-Dokumentationen Lesen lernen

ruby-doc.org.



Praxis

API-Doku lesen

- Lest die API-Dokumentation von String, Array und/oder Hash.
- Sucht Euch eine Methode aus, die Euch interessant erscheint und lernt ihre Funktionsweise.
- Präsentiert Eure Methode.

Code blocks als objekte

- Objekte vom Typ Proc
- Erzeugen mit `Proc.new`, `proc`, `lambda` oder `->`
- `lambda` / `->`
 - parameter-checks
 - Verhalten von `return`
- `call` zum ausführen

Blöcke benutzen

```
def benchmark
  a = Time.now
  yield
  puts Time.now - a
end

benchmark do
  1000000.times { "23000".to_i }
end
```

Block?

```
def frame(content = "")
  content = yield if block_given?
  border = "-" * (content.size + 4)
  puts border
  puts "| #{content} |"
  puts border
end

frame("hallo")
frame { "zufall#{rand(1009)}" }
```

Block "durchreichen"

```
def twice(&block)
  2.times &block
end

twice { puts "hallo" }
```



Adressdatenbank

pad neu

Schreibt eine Adressdatenbank. Das Programm soll Adressen, die mindestens Name, Straße und Ort enthalten im Hauptspeicher ablegen und wieder verfügbar machen.

Dazu präsentiert das Programm dem Benutzer ein Hauptmenü mit folgenden Möglichkeiten:

- Bei der Eingabe von “N” (oder “n”) wird ein neuer Datensatz angelegt. Der Benutzer wird der Reihe nach gebeten, alle erforderlichen Daten einzugeben.
- Bei der Eingabe von “L” (oder “l”) werden alle bisher eingegebenen Datensätze in verkürzter Form ausgegeben.
- Die Darstellung soll nur den Index und den Namen enthalten.
- Bei Eingabe eines gültigen Index wird der passende Datensatz vollständig angezeigt.
- Zusätzlich soll es möglich sein, das Programm zu beenden.

I/O

- `STDIN`, `STDOUT`, `STDERR`
- `puts` ist eigentlich `STDOUT.puts`
- `gets` ist `STDIN.gets`

I/O-Operationen

- Zeilenweise: `gets`, `puts`
- Zeichenweise: `getc`, `putc`
- Mehr: `read`, `write`
- Gilt für Streams, Dateien, Netzwerk

Dateien

```
File.open("test.txt", "w") do |file|  
  file.puts "Hallo"  
end
```

“r”ead, “w”rite, “a”ppend (, “b”inary)

Datei zeilenweise verarbeiten

```
File.open("data.txt", "r") do |file|  
  file.each_line do |line|  
    puts line  
  end  
end
```

Helper

```
File.exists("/tmp/test.txt")    # false  
File.basename("/tmp/test.txt")  # "test.txt"  
File.dirname("/tmp/test.txt")   # "/tmp"  
File.join("tmp", "test.txt")  
# "tmp/test.txt"  
Dir.mkdir("/tmp/testdir")  
Dir.entries("/tmp")            # [...]  
Dir.glob("/tmp/*.txt")         # [...]
```



Adressdatenbank die zweite

Erweitert Eure Adressdatenbank, so dass Ihr die Daten dauerhaft abspeichern und auch wieder einlesen könnt.