Raku Quiz

Herbert „Lichtkind“ Breunung

Wer bin ich ?

- Programmierer, Student, Autor
- Softwaretechnik, Musik, Graphik
- auch Sprachen (Perl)
  - Software (Projekte)
  - Module (CPAN)
  - Artikel (heise, perlmag), Dokumentation
  - Sprecher auf Konferenzen (lichtkind.de/vortrag)

Lichtkind

LICHTKIND
Herbert Breunung

lichtkind@cpan.org
kephra.sourceforge.net

ACTIVITY

24 month

All Releases

TOOLS

CPANTesters Reports
CPANTS
Module permissions
MetaCPAN Explorer

The Perl Toolchain Summit 2025 Needs You: You can help 🙏 Learn more

11 distributions uploaded by Herbert Breunung (LICHTKIND)

Search distributions by Herbert Breunung

All Releases by Herbert Breunung

Release		Uploaded
App-GUI-Juliagraph-0.7	drawing Mandelbrot and Julia fractals with compl…	4 days ago
App-GUI-Cellgraph-0.81	draw pattern by cellular automata	Apr 15, 2025
App-GUI-Harmonograph-1.0	drawing by lateral and rotary pendula	Apr 08, 2025
Kephra-0.406	compact, effective and inventive coding editor	Dec 11, 2023
Graphics-Toolkit-Color-1.71	color palette constructor	Sep 21, 2023
Chart-v2.403.9	a series of charting modules	Nov 28, 2022
Alien-Font-Uni-0.3	access to Unifont truetype file	Nov 18, 2022
Tie-Wx-Widget-1.01	implicit access value of a Wx widget	Nov 18, 2022

Warum ich ?

- befasse mich schon mit Raku seit 2006
- schrieb eine ehem. Doku (Perl 6 Tablets)
- … einige Artikel (offizielles Release)
- Autor von Math::Matrix (die pr (pp) Lösung)

Dieser Vortrag - Plan

- Grundlagen
  - Philosophie / Syntax / Semantik
  - Beispiele fürs Verständnis
- Neuerungen
  - Stärken: Grammatiken, parallel, OO
  - hÅrc Stack (HTMX, Air, Red, Cro), LLM

Dieser Vortrag - Realität

- wenig Wissen voraus setzbar
- wenige Neuerungen
- interessante Feature brauchen Vorwissen
- zu viel Theorie ist langweilig
- Was machen ? ? ?

Dieser Vortrag - Plan

- spiele einen der frisch umsteigt Perl → Raku
- anhand von Beispielen (Grundlagen + Stärken)
  - Philosophie / Syntax / Semantik nebenbei
- Neugier wecken, Gefühl entwickeln
  - schwierigen Fälle → Humor
  - zur Frustreduktion der Raku - Beginner

Ein paar Vokabeln

- Raku ist die Sprache
  - Definition der Syntax und Semantik
  - 70-90% Kind von Larry Wall
  - entwickelt als Perl 6
  - Rakuda ist Kamel auf japanisch

Ein paar Vokabeln

- Raku ist die Sprache (wie Perl, jap. Kamel)
- Rakudo der Interpreter/Compiler
  - der Wichtigste
  - Hauptautor: Jonathan Worthington
  - Raku(da) – Do , Weg des Kamels: 駱駝道
  - basiert auf MoarVM und (Not Quite Perl)

Core Rakudo

- Raku ist die Sprache (wie Perl, jap. Kamel)
- Rakudo der Interpreter (wie perl, Weg des ..)
- Raku ecosystem
  - Rakudo + Moar + Kernmodule

zef hosted by AMAZON

- Raku ist die Sprache (wie Perl, jap. Kamel)
- Rakudo der Interpreter (wie perl, Weg des ..)
- Raku ecosystem (core distro)
- zef ecosystem (zef ~~ cpanm, CPAN)
  - ID: name, author, version, API version
  - (AWS lambda, S3, cloudfront) upload per fez

Rakuland sucht in 3 Archiven

- Raku ist die Sprache (wie Perl, jap. Kamel)
- Rakudo der Interpreter (wie perl, Weg des ..)
- Raku ecosystem (core distro)
- zef ecosystem (CPAN + cpanm)
- https://raku.land (MetaCPAN -Seite)
  - zef search - Archive: zef / CPAN / p6c

Längere Dateiendungen

- Raku ist die Sprache (wie Perl, jap. Kamel)
- Rakudo der Interpreter (wie perl, Weg des ..)
- Raku ecosystem (core distro)
- zef ecosystem (CPAN + cpanm)
- https://raku.land (MetaCPAN -Seite)
- .pm → .rakumod, pl → raku, t → rakutest

Ein paar mehr Vokabeln

- Zur Programmiersprache ….

Ein paar mehr Vokabeln

- Slang = Sublanguage
  - Sprache in der Sprache mit eigener Syntax
  - z.B. Regex, Quoting
  - könnte auch SQL sein
  - Module erweitern Sprache

Ein paar mehr Vokabeln

- Twigil = Tweak + Sigil, sekundäre Sigil
- steht für Namensräume :
  - globale Variablen: $*b
  - öffentliche Attribute: $.name
  - private Attribute: $!name
  - Metaobjekt-Methoden: $^name

Ein paar mehr Vokabeln

- Twigil = Tweak + Sigil, sekundäre Sigil
- steht für Namensräume :
  - dyn. Spezialvariablen: $*RAKU.version ($])
  - statische compiler hint: $?LINE (__LINE__)
    - Überlebende: $! $/ $_ @_

Ein paar mehr Vokabeln

- -$d prefix
- $a + $b infix
- $n++ postfix operator
- [“text“] circumfix
- $g[3] postcircumfix

Raku Versionen

- entstand als Perl 6 (6 fest verankert)
- 6.c (Коледа, Christmas) == 1.0 (Whn 2015)
- 6.d (Diwali) ist aktuell (Juli 2019)
- 6.e kommt nicht bald
- Rakudo, MoarVM Versionen: Jahr.Monat

Mögen die Spiele beginnen :

Raku Quiz

Testrunde: Wie in Perl 5 ?

- `use v6.c;` `# Start jedes Scripts`

Interessante Möglichkeit

- `use v6;` `# .pl vs. .raku Weiche`

Wieder wie in Perl 5!

- `use v6;` `# .pl vs. .raku Weiche`
- `my($ö) = 3.;` `# weiterhin Sigils: $ @ % &`

Fehlermeldung:

- `use v6;` `# .pl vs. .raku Weiche`
- `my($ö) = 3.;` `# ! var. ‘$ö‘ not declared`

Fehlermeldungen

- `use v6;` `# .pl vs. .raku Weiche`
- `my($ö) = 3.;` `# ! var. ‘$ö‘ not declared`

```
                    # besser:no sub my declared
                # rakudo sucht eine sub my {...}
            # da kein Leerz. zwischen my und (
            # immer Leerzeichen nach keyword!
```

Erster Fehler behoben !

```
use v6;             # .pl vs. .raku Weiche
my ($ö) = 3.;       # Leerzeichen nach keyword!
                    # jetzt gut?
```

Zweiter Fehler !

- `use v6;` `# .pl vs. .raku Weiche`
- `my ($ö) = 3.;` `# Leerzeichen nach keyword!`

  `# ! natürlich nicht`

Hängendes Komma !

```
use v6;
my ($ö) = 3.;
```

```
# .pl vs. .raku Weiche
# ! Decimal point must be
#   followed by digit
# .0, 0.0, .1, 2_000 geht
```

Zweiter Fehler behoben !

- `use v6;` `# .pl vs. .raku Weiche`
- `my ($ö) = 3;` `# geht es jetzt?`

Ö ist kön Problem

- `use v6;`
- `my ($ö) = 3;`

```
# .pl vs. .raku Weiche
# das läuft !
# UTF identifier normal
# use utf8; per default
```

Was haben wir gelernt ?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# keine hängende Punkte`

```
                    # verzichte auf () !
                    # my$ö=3; geht
                    # aber besser nicht

  # Moral: Raku verlangt mehr Formatierung !
```

Es gibt echte Datentypen !

```
use v6;                 # .pl vs. .raku Weiche
my $ö = 3;              # achte auf Formatierung
say $ö.WHAT;            # (Int) oder .^name : Int
                        # echter Datentyp
                        # alles ist ein Objekt
                        # auch die Klassen
```

Wann werden sie geprüft ?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `say $ö.WHAT;` `# (Int)`
- `$ö = "ein Gedicht"; # Typfehler ???`

Prüfung nicht immer !

```
use v6;             # .pl vs. .raku Weiche
my $ö = 3;          # achte auf Formatierung
say $ö.WHAT;        # (Int)
$ö = "ein Gedicht"; # natürlich geht das
# skalares Container - Objekt $ö
# Wertobjekt 3 ist Instanz der Klasse Int
```

Prüfung ? Hier schon !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `say $ö.WHAT;` `# (Int)`
- `$ö = "ein Gedicht"; # natürlich geht das`
- `my Int @a = (1,‘Haus‘); # Fail`

  `# .WHAT ==> (Array[Int])`

Gibt es Typobjekte ?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß' = Int->new(); # nimmt er das ?`

Fehlermeldung:

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß' = Int->new(); # !two terms in a row`

‘ und – nicht am Ende | Anfang

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß’s = Int->new(); # - auch erlaubt`

Jetzt gut?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int->new(); # ??`

Fehlermeldung:

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int->new(); # ! expect. postfix`

Grundlegene Syntaxfalle

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int.new(); # -> ==> .`

  `# Methodenaufruf mit .`

  `# Stolperfalle für P5 !`

Jetzt korrekt ?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int.new();`

Unüblich, aber akzeptabel!

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = Int.new(3);` `# gute Formatierung`
- `my $ß’s = Int.new();` `# $ß’s enthält 0`

```
# (default von Int)
# Argument default
# $ö.WHAT eq $ß.WHAT
```

Kein Typobjekt !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß's = Int.new();` `# $ß's enthält 0`
- `say $ß's.DEFINITE` `# True`

Any ist das neue undef

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß's = Int.new();` `# $ß's enthält 0`
- `say $ß's.DEFINITE` `# True`
- `my Int $z;` `# Any - kein Inhalt`
- `say defined $z` `# False, da Any`

$z muss definiert sein !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß's = Int.new();` `# $ß's enthält 0`
- `say $ß's.DEFINITE` `# True`
- `my Int:D $z;` `# Fail !`

`var $z muss definiert sein, weil :D`

Müsste doch gehen?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß's = Int.new();` `# $ß's enthält 0`
- `my Int $z = Inf;` `# ??`

Müsste doch gehen?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß’s = Int.new();` `# $ß’s enthält 0`
- `my Int $z = Inf;` `# !`

Int werden zu bigint bei Bedarf

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß’s = Int.new();` `# $ß’s enthält 0`
- `my Int $z = Inf;` `#`

```
    # … assign a literal of type Num (Inf)
    # to a variable of type Int
```

7 ist doch Methodenargument ?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push 7;` `# geht das?`

Ja, aber Klammern fehlen !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push 7;` `# ! Two terms in a row`

Diesmal sind Klammern Pflicht !

```
use v6;                 # .pl vs. .raku Weiche
my $ö = 3;              # gute Formatierung
my $ß's = Int.new(); # $ß's enthält 0
@a.push(7);             # push @a, 7 geht auch
                        # da push dort keyword
```

Oder Doppelpunkt !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push: 7;` `# : saugt Methoden Args`

Mehr als List::Util

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`

```
@a.push: 7;          # Containerobjet Array
                     # List::Util inklusive
                # .min, .max, .uniq und mehr
```

Sieht grenzwertig aus …

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# gute Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n<1;`

Warum nicht ?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n<1;` `# !expecting any of postfix`

Zweideutiger Syntax

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n<1;` `# fehlt Leerzeichen nach $n`

`# könnte Hash slice $n<1> sein`

Syntaxfalle Behoben

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`

  `# auch 1<3 --> 1 <3`

Dann müßte doch … ?

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { ... }`

April, April !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { ... } # Alles Gut! () optional`

Weniger Klammern in Raku

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { ... } # das ist nativ Raku`

Klammern optional !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß’s = Int.new(); # $ß’s enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if ($a < 0) { }` `# $a ist keine Spez.Var.`

Vermeide Klammern !

- `use v6;` `# .pl vs. .raku Weiche`
- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`

Immer noch Ternärer Op ?

- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ? 2 : 3; # gute Fehlermeldung!`

Alles Neu !

- `my $ö = 3;` `# achte auf Formatierung`
- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`

Die Logik für die Änderung

```
# Konsitent mit der Sprache, denn:
# ? ist bool ja und ! für bool nein
# konsistent mit && || (Kurzschluß)
# & | sind bitweise Op
# besser sichtbar als ? :
```

- `say $ß == 1 ?? 2 !! 3;`

Selten Notwendig: C-Style-Loop

- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;  # saugt Methoden Args`
- `$n < 1;  # erzeugt echten Bool`
- `if $a < 0 { }  # $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `for (my$i=0;$i <5;$i++) { say $i } # ??`

Sehr gute Fehlermeldung

- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;  # saugt Methoden Args`
- `$n < 1;  # erzeugt echten Bool`
- `if $a < 0 { }  # $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `for (my$i=0;$i <5;$i++) { say $i } # !`

C-Style- >Loop<

- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;  # saugt Methoden Args`
- `$n < 1;  # erzeugt echten Bool`
- `if $a < 0 { }  # $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my$i=0;$i <5;$i++) { say $i } # geht`

Verschiedener Kontext !

- `while 1 { say $_ } # forciert Bool Kontext`
- `for 1, 2 { say $_ } # List Kontext`
- `loop { say $_ } # kein Kontext`

  `# runde Klammern notwendig`
- `loop (my$i=0;$i <5;$i++) { say $i } # geht`

Native Endlosschleife

- `while (1) { say $i } # Endlosschleife`
- `while 1 { say $i }   # auch gutes Raku`
- `loop { say $i }      # besser !`

- `loop (my$i=0;$i <5;$i++) { say $i } # geht`

Vermeide Gedränge !

- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;  # saugt Methoden Args`
- `$n < 1;  # erzeugt echten Bool`
- `if $a < 0 { }  # $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my$i=0;$i <5;$i++) { say $i } # Z.3`

So ist gut !

- `my $ß's = Int.new(); # $ß's enthält 0`
- `@a.push: 7;  # saugt Methoden Args`
- `$n < 1;  # erzeugt echten Bool`
- `if $a < 0 { }  # $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`

Natives Perl !

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. 4 { say }` `# ???`

Es gibt kein „say“ mehr ?

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. 4 { say }` `# ! bare word say`

Keine Autozuweisung von $_ an say

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. 4 { say $_ }` `# Kontext !`

Methode auf Objekt $_

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. 4 { .say }` `# nativ ! ($_.say)`

Schmeiß das in den Block !

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. 4 -> $i { say $i }` `# nativer !`

Normale Ad-hoc Signatur

```
                              # "->" ist kein Spezial-
if  1 < 3 -> $a {   # Syntax für "for"
    say $a                  # ergibt: True (1 < 3)
}                            # im Bool Kontext
```

- `for 0 .. 4 -> $i { say $i } # nativer !`

Beispiel mit mehr Sinn

```
if   compute() -> $result {
     say $result
}
```

- `for 0 .. 4 -> $i { say $i } # nativer !`

Signatur mit -> und <->

```
if  1 < 3 -> $a {      # -> readonly
    say $a             # Pfeil zeigt
}                      # 1. Datenflussrichtung
                       # 2. Zugriffsrechte auf Var
```

- `for 0 .. 4 <-> $i { say $i } # readwrite`

Geht das ?

```
my &greet2 =  ->  $to-whom
              { say "Grüzi $to-whom!" }
greet2('Max');
```

- `for 0 .. 4 -> $i { say $i } # nativer !`

Klar Doch !

```
my &greet2 =  ->  $to-whom
              { say "Grüzi $to-whom!" }
greet2('Max');      # Grüzi Max!
                    # & ist Sigil der sub
```

- `for 0 .. 4 -> $i { say $i } # nativer !`

Klar Doch !

```
my &greet2 =  { say "Grüzi $^to-whom!" }
greet2('Max');      # Grüzi Max!
                    # $^a  statt  $a
              # autogenerierte Argumentnamen
              # Namen sortiert nach pos.
for 0 .. 4 -> $i { say $i } # nativer !
```

Die fast normale Raku for Schleife

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. 4 -> $i { say $i }` `# nativer !`

Hört hört : 4 = 5 - 1

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. 5 - 1 -> $i { say $i }` `# ??`

N mal == 0 .. N - 1

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. $l - 1 -> $i { say $i }` `# so oft!`

Für Schreibfaule

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. ^$l -> $i { say $i }` `# so oft!`

TIMTOWTDI

```
if 0 <=  $length and $length <= 5 { ...
```

```
if 0 <= $length <= 5 { ...
```

```
if $length ~~ 0 .. 5 { ...
```

- `for 0 .. ^$l -> $i { say $i } # so oft!`

Ohne Schranke

```
if 0 <= $length and $length < 5 { ...
if 0 <= $length < 5 { ...
if $length ~~ 0 .. ^5 { ...
        # Zahlenbereich ohne Schranke 5
```

- `for 0 .. ^$l -> $i { say $i } # so oft!`

Nie wieder -1 schreiben !

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for 0 .. ^$l -> $i { say $i }` `# so oft!`

Die normale Raku for - Schleife !

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^$l -> $i { say $i }` `# seeehr nativ !`

Maximal Kondensiert !

- `@a.push: 7;` `# saugt Methoden Args`
- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i }` `# seeehr nativ !`

Iteriere über ein Hash value ?

- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i }` `# seeehr nativ !`
- `for $hash{"key"} { .say }` `# ???`

Der Hash steht doch da !

- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a ist keine Spez.Var.`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i }` `# seeehr nativ !`
- `for $hash{"key"} { .say }` `# no $hash decl.`

Invariante Sigil @ un d %

- `$n < 1;` # erzeugt echten Bool
- `if $a < 0 { }` # $a keine Spezialvariable
- `say $ß == 1 ?? 2 !! 3;` # Syntax regulär !
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i }` # seeehr nativ !
- `for %hash{"key"} { .say }` # % @ invariant!

Hashvariable forciert Kontext

```
my %hash = 'key', 'value'; # P5 kompatibel
my %hash = key => 'value'; # P5 kompatibel
my %hash = :key<value>;    # nativ Raku
  # Hash - Container
```

- `for %hash{"key"} { .say } # % @ invariant`

Hashoperator forciert Kontext

```
my $hash = %('key', 'value');
my $hash = %(key => 'value'); # kein P5=>
my $hash = %(:key<value>); # ohne quoting
  # Hash-Container-Objekt im Skalar
```

- `for $hash{"key"} { .say } # % @ invariant`

Zeig mir deinen Inhalt!

```raku
say %hash;

say "%hash";

say %hash.Str;

say %hash.gist;

say %hash.raku;
```

Unterschiedliche Formate

```
say %hash;          # ruft .gist
say "%hash";        # ruft .Str bei $
say %hash.Str;      # String Kontext
say %hash.gist;     # Kurzfassung
say %hash.raku;     # interne Darstellung
                    # reziprok zu eval
```

Formate im Detail

```
say %hash;                  { key => value }
say "%hash";                %hash
say %hash.Str;              key \t  value
say %hash.gist;             { key => value }
say %hash.raku;             { :key("value") }
say ruft .gist: IO::Handle<(Any)>(closed)
```

put ruft .Str + newline

```
put %hash;                key \t   value
put "%hash";              %hash
put %hash.Str;            key \t   value
put %hash.gist;           { key => value }
put %hash.raku;           { :key("value") }
print nur .Str
```

$hash und %hash sind 2 Variablen

- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a keine Spezialvariable`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i }` `# seeehr nativ !`
- `for %hash{"key"} { .say }` `# % @ invariant!`

<> ist das neue qw/ /

- `$n < 1;` # erzeugt echten Bool
- `if $a < 0 { }` # $a keine Spezialvariable
- `say $ß == 1 ?? 2 !! 3;` # Syntax regulär !
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i }` # seeehr nativ !
- `for %hash<key> { .say }` # nativ (! $n<1)

Iteriere über zwei Werte

- `$n < 1;` `# erzeugt echten Bool`
- `if $a < 0 { }` `# $a keine Spezialvariable`
- `say $ß == 1 ?? 2 !! 3;` `# Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i }` `# seeehr nativ !`
- `for %hash<key two> { .say }` `# qw is <>`

Eine komische Ecke in Perl 5

- `if $a < 0 { }  # $a keine Spezialvariable`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw is <>`
- `while (my ($key, $value) = each %hash) {`

Sauber in Raku ?

- `if $a < 0 { }  # $a keine Spezialvariable`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw is <>`
- `for %hash -> $key, $value { ... }`

Tut nicht was ihr denkt !

- `if $a < 0 { }  # $a keine Spezialvariable`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash -> $key, $value {...} # 2 Paar`

Zur Erinnerung :

```
my $hash = %( 'key', 'value' );
my $hash = %( key => 'value' );
my $hash = %( :key<value> );
```

- `for %hash -> $key, $value {...} # 2 Paar`

Was wurde da erzeugt ?

```
my $hash = 'key', 'value'; # List
my $hash = key => 'value'; # Pair
my $hash = :key<value>;    # Pair
                           # $hash.WHAT
# Hash in List Kontext emitiert Paare
```

- `for %hash -> $key, $value {...} # 2 Paar`

Kann man machen !

- `if $a < 0 { }  # $a keine Spezialvariable`
- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash -> $pair { say $pair.key }`

Die native Lösung:

```
# kv Methode erzeugt flache Liste
# key1 value1 key2 value2 key3 …
# daraus entnehmen wir immer 2
```

- `for %hash.kv -> $key, $value { ... }`

Keine Spezialfälle !!!

```
# zip Operator Z
# es gab Schreibweise: zip
# (key1, value1), (key2, value2), key3 …
my %hash = @keys Z @values2;
```

- `for @l1 Z @l2 -> $el1, $el2 { ... }`

Liest sich gut ?

- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash.kv -> $key, $value { ... }`
- `for @l1 Z @l2 -> $el1, $el2 { ... }`

for does not .flat the List

- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash.kv -> $key, $value { ... }`
- `for @l1 Z @l2 -> $tupel { ... }`

Auch als Kreuzprodukt:

- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash.kv -> $key, $value { ... }`
- `for @l1 X @l2 -> $tupel { ... }`

Kreuzprodukt als Meta – Op X

- `say $ß == 1 ?? 2 !! 3; # Syntax regulär !`
- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash.kv -> $key, $value { ... }`
- `for @head X~ @body -> $animal { ... }`

benannte Args in Signaturen

- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash.kv -> $key, $value { ... }`
- `for @head X~ @body -> $animal { ... }`
- `$object.method( :key<value> );`

Wohin sind die Filetest Ops ?

- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash.kv -> $key, $value { ... }`
- `for @head X~ @body -> $animal { ... }`
- `-r 'Datei' # eigener Op im glob. Namenraum`

Da wärt ihr nie drauf gekommen !

- `loop (my $i = 0; $i < 5; $i++) { say $i }`
- `for ^5 -> $i { say $i } # seeehr nativ !`
- `for %hash<key two> { .say } # qw in <>`
- `for %hash.kv -> $key, $value { ... }`
- `for @head X~ @body -> $animal { ... }`
- `'/path/to/file'.IO ~~ :r`

Was ist hier passiert ?

```
# ~~ ist der smartmatch operator
# braucht ihr für Regex (=~ ist weg)
# er findet alles ohne KI
# übersetzt mit: passt das?
```

- `'/path/to/file'.IO ~~ :r`

Was ist hier passiert ?

```
'/path/to/file'.WHAT          # (Str)
'/path/to/file'.IO.WHAT       # (Path)
'/path/to/file'.IO.r          # True
# r ist eine Methode des IO::Path Objektes
# der Rest ist Smartmatchmagie
```

- `'/path/to/file'.IO ~~ :r`

file ops werden (Path) Methoden

```
# Methoden: r e w d (statt -r -e -w -d)
'/path/to/file'.IO.WHAT    # (Path)
if '/path/to/file'.IO.r { ...
if $pfad.IO.r { ...
```

- `'/path/to/file'.IO ~~ :r`

Kann auch Inhalt wegschlürfen !

```
# Methoden: r e w d slurp lines
'/path/to/file'.IO.WHAT    # (Path)
if '/path/to/file'.IO.r { …
my $text = 'datei'.IO.slurp;
my $text = slurp 'datei';
'datei'.IO.spurt: "I    Raku!"; # spuckt
```

lines a.k.a. split /\n/

```
# Methoden: r e w d slurp lines words
'/path/to/file'.IO.WHAT    # (Path)
for '/path/to/file'.IO.lines { ...
for lines 'datei'.IO -> $line { ...
for  slurp('datei').lines -> $line { ...
my $text = slurp 'datei';
```

PathTools inklusive

```
for words 'datei'.IO -> $word { ...

$file.basename

$file.dirname

$*CWD


say slurp 'datei' :bin; # zeigt hex Werte
```

Es gibt noch IO::Handle

```
my $fh = 'my-file.txt'.IO.open: :a;

$fh.print: "I count: ";

$fh.open / .close / .tell / .seek

$fh.get:  / $fh.put:    # String

$fh.read: / $fh.write: # binär
```

File::Find

```
for '/path'.IO.dir.grep: {.f} -> $file {
    say '== ' ~ $file;
    for $file.lines.kv -> $nr , $line {
        say "$nr : $line";
    }
}
```

Pipe Operator

```
for '/path'.IO.dir ==> grep {.f} -> $file {
    say '== ' ~ $file; # nicht ganz so schön
    for $file.lines.kv -> $nr , $line {
        say "$nr : $line";
    }
}
```

Links

- Haupseite: raku.org
- Module: raku.land
- IRC: irc.libera.chat/#raku
- Rundschau: rakudoweekly.blog/
- Folien: lichtkind.de/vortrag

Aus der Graphschaft

Danke

Aus der Graphschaft

Fragen ?