

6 - Loops und Funktionen

author: Benedict Witzenberger date: 18. April 2019 autosize: true

Recap

Es gibt wieder eine Übung zu allem, was wir gestern gelernt haben:

<https://benedict-witzenberger.de/ifp-r-bootcamp19/Test2.html>

Was wir heute vorhaben

R

Loops in R

eigene Funktionen in R

Wo bekommen wir Hilfe?

Ausblick: Visualisierung mit Base R

Projekte

Wie machen wir Datenjournalismus

Unser eigenes Projekt

Loops (Schleifen) in R

Warum brauchen wir Loops?

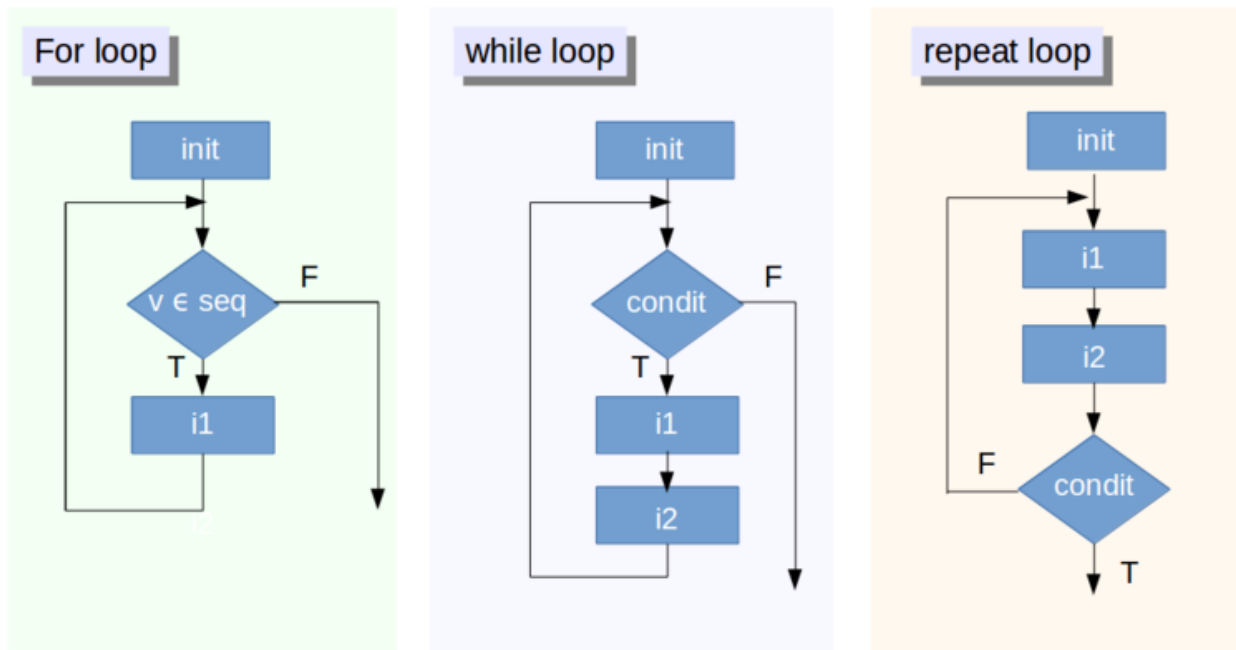
Wir wiederholen die gleichen Arbeitsschritte immer und immer wieder...

... bis ein bestimmter Zustand eingetreten ist:

- for-Loops: “bis” ein Zustand eingetreten ist
- while-Loops: “während” ein Zustand eingetreten ist (ähnlich zum repeat-Loop)

Diese Wiederholungen werden im Entwicklerslang: “Iteration” genannt

Wie Loops aussehen



For-Loop I

```
iterator <- c(1:10)

for (i in iterator) {
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Was ist i?

For-Loop II

```
result <- vector()

input_vector <- c(1:10)
```

```
for(i in seq_along(input_vector)) {
  result[i] <- input_vector[i] * input_vector[i]
  print(paste("i:", i, "; Result:", result[i]))
}
```

```
[1] "i: 1 ; Result: 1"
[1] "i: 2 ; Result: 4"
[1] "i: 3 ; Result: 9"
[1] "i: 4 ; Result: 16"
[1] "i: 5 ; Result: 25"
[1] "i: 6 ; Result: 36"
[1] "i: 7 ; Result: 49"
[1] "i: 8 ; Result: 64"
[1] "i: 9 ; Result: 81"
[1] "i: 10 ; Result: 100"
```

Naming Convention in For-Loops

Die

```
for (i in seq_along(vector1)) {
  for (j in seq_along(vector2)) {
    for (k in seq_along(vector3)) {
      do_something()
    }
  }
}
```

Denkaufgabe: For-Loops für Matrix

Erstellt mit einem Loop eine Matrix, in der der Inhalt jedes Feldes das Produkt seiner Indizes ist.

```
mymat <- matrix(nrow=30, ncol=30)
```

```
# For each row and for each column, assign values based on position: product of two indexes
for(i in 1:dim(mymat)[1]) {
  for(j in 1:dim(mymat)[2]) {
    mymat[i,j] =
  }
}
```

Tipp: Ihr müsst in der Funktion drei Sachen eintragen

Lösung: For-Loops für Matrix

```
mymat <- matrix(nrow=30, ncol=30)

for(i in 1:dim(mymat)[1]) {
  for(j in 1:dim(mymat)[2]) {
    mymat[i,j] = i*j
  }
}
```

```
}
}
```

```
mymat[1:10, 1:10]
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]
[1,]	1	2	3	4	5	6	7	8	9	10
[2,]	2	4	6	8	10	12	14	16	18	20
[3,]	3	6	9	12	15	18	21	24	27	30
[4,]	4	8	12	16	20	24	28	32	36	40
[5,]	5	10	15	20	25	30	35	40	45	50
[6,]	6	12	18	24	30	36	42	48	54	60
[7,]	7	14	21	28	35	42	49	56	63	70
[8,]	8	16	24	32	40	48	56	64	72	80
[9,]	9	18	27	36	45	54	63	72	81	90
[10,]	10	20	30	40	50	60	70	80	90	100

Best Practises bei For-Loops

Füllt eine Ergebnisvariable nicht IM Loop. Sie muss dann von Loop-Durchgang zu Loop-Durchgang kopiert werden, das macht den Code langsam.

Besser: Erstellt eine Ergebnisvariable (zum Beispiel Liste, Vektor oder Dataframe) VOR dem Loop. Und befüllt mit [i] nur die entsprechenden Bereiche im Loop.

Nicht:

```
for (i in seq_along(vector)) {
  loop_result <- compute_something()
  result_variable <- c(result_variable, loop_result)
}
```

Besser:

```
result_variable <- vector(length = length(input_vector))

for (i in seq_along(input_vector)) {
  loop_result <- compute_something()
  result_variable[i] <- loop_result
}
```

Bei Dataframes

```
result_list <- list()

for (i in seq_along(input_vector)) {
  loop_result_df <- compute_something()
  result_list[[i]] <- loop_result_df
}
```

```
dplyr::bind_rows(result_list)
```

dplyr lernt ihr im nächsten Blockkurs genauer kennen.

For-Loops gibt es in vielen Sprachen

Java:

```
// Prints the numbers 0 to 99 (and not 100), each followed by a space.
for (int i=0; i<100; i++)
{
    System.out.print(i);
    System.out.print(' ');
}
System.out.println();
```

Python:

```
for item in some_iterable_object:
    do_something()
```

Wann brauchen wir For-Loops in R?

- Viele Daten einlesen und bearbeiten
- Viele Webseiten scrapen

Eher nicht bei:

- Mehrere Variablen eines Dataframes verändern
- Mehrere Rechnungen mit Vektoren anstellen

Vorsicht bei For-Loops I

```
a <- c(1:10)
b <- c(1:10)

res <- numeric(length = length(a))
for (i in seq_along(a)) {
    res[i] <- a[i] + b[i]
}
res
```

```
[1]  2  4  6  8 10 12 14 16 18 20
```

Was macht die For-Schleife?

Vorsicht bei For-Loops II

Für viele Vektoroperationen gibt es einfacherere - und schnellere Alternativen, als eine For-Schleife:

Hier, eine einfache Summe.

```
res2 <- a + b

all.equal(res, res2) # testet, ob die Variablen gleich sind
```

```
[1] TRUE
```

Apply-Familie in R

Loops, die ihr über Vektoren laufen lassen wollt, können oft mit Apply kombiniert werden. Diese Funktionen sind deutlich schneller, als For-Loops (wird daher erst bei großen Datensätzen interessant)

- Apply: `apply(X, MARGIN, FUN, ...)`, MARGIN sind die Zeilen und Spalten einer Matrix (oder eines Dataframes). Es geht auch `MARGIN = c(1,2)`. FUN kann jede beliebige (auch selbst geschriebene Funktion sein).
- Lapply: Gibt die Ergebnisse der Apply-Berechnung als Liste zurück. Jedes Ergebnis ist ein Element der Liste.
- Sapply: Funktioniert wie Lapply, versucht aber, das Ergebnis als Vektor oder Matrix auszugeben.
- Vapply: Hier gibt der Nutzer vorher ein, welche Klasse das Ergebnis haben soll, z.B. `VUN.VALUE = character(1)`

Apply bei einer Matrix I

```
matrix_x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
```

```
dimnames(matrix_x)[[1]] <- letters[1:8]
```

```
matrix_x
```

	x1	x2
a	3	4
b	3	3
c	3	2
d	3	1
e	3	2
f	3	3
g	3	4
h	3	5

```
apply(matrix_x, 2, mean, trim = .2)
```

	x1	x2
	3	3

Apply bei einer Matrix II

```
col.sums <- apply(matrix_x, 2, sum)
```

```
row.sums <- apply(matrix_x, 1, sum)
```

```
rbind(cbind(matrix_x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))
```

	x1	x2	Rtot
a	3	4	7
b	3	3	6
c	3	2	5
d	3	1	4

```
e      3  2   5
f      3  3   6
g      3  4   7
h      3  5   8
Ctot 24 24  48
```

Lapply, Sapply

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
```

```
lapply(x, quantile)
```

```
$a
  0%   25%   50%   75%  100%
1.00  3.25  5.50  7.75 10.00
```

```
$beta
      0%      25%      50%      75%      100%
0.04978707 0.25160736 1.00000000 5.05366896 20.08553692
```

```
$logic
  0%  25%  50%  75% 100%
0.0  0.0  0.5  1.0  1.0
```

```
sapply(x, quantile)
```

```
      a      beta logic
0%    1.00 0.04978707  0.0
25%    3.25 0.25160736  0.0
50%    5.50 1.00000000  0.5
75%    7.75 5.05366896  1.0
100%  10.00 20.08553692  1.0
```

Vapply

```
vapply(x, mean, numeric(1))
```

```
      a      beta      logic
5.500000 4.535125 0.500000
```

Übung: Vapply

Beim folgenden Code scheitert `sapply()`. Woran könnte das liegen?

Und: Wie könnten wir `vapply` benutzen, um zu merken, dass wir in einen Fehler laufen??

```
market_crash <- list(dow_jones_drop = 777.68,
                    date = as.POSIXct("2019-04-01"))
```

```
lapply(market_crash, class)
```

```

$dow_jones_drop
[1] "numeric"

$date
[1] "POSIXct" "POSIXt"
sapply(market_crash, class)

$dow_jones_drop
[1] "numeric"

$date
[1] "POSIXct" "POSIXt"

```

Lösung: Vapply

So geht der Test mit `vapply` - die Funktion wirft einen Fehler aus, weil die beiden Ergebnisse nicht ein einzelner `character` sind:

```

vapply(market_crash, class, FUN.VALUE = character(1))

> vapply(market_crash, class, FUN.VALUE = character(1))
Error in vapply(market_crash, class, FUN.VALUE = character(1)) :
  values must be length 1,
  but FUN(X[[2]]) result is length 2

```

So würde es funktionieren, wir wandeln die `POSIXcts` in `'characters'` um:

```

vapply(market_crash, FUN = function(x) paste(class(x), collapse = "; "),
      FUN.VALUE = character(1))

      dow_jones_drop      date
      "numeric" "POSIXct; POSIXt"

```

While und Repeat

Es gibt noch zwei weitere Wege: `While` und `Repeat`

Ein Beispiel für `While` (muss man nacheinander ausführen):

```

readinteger <- function(){
  n <- readline(prompt="Please, enter your ANSWER: ")
}

response <- as.integer(readinteger())

while (response!=42) {
  print("Sorry, the answer to whatever the question MUST be 42");
  response <- as.integer(readinteger());
}

```


Repeat ist eine Variante von While

Dabei wird der `repeat`-Block wenigstens einmal ausgeführt.

```
readinteger <- function(){
  n <- readline(prompt="Please, enter your ANSWER: ")
}

repeat {
  response <- as.integer(readinteger());
  if (response == 42) {
    print("Well done!");
    break
  } else print("Sorry, the answer to whatever the question MUST be 42");
}
```

Break und next

Neben `for`, `while` und `repeat` gibt es noch zwei Befehle, die die Schleifen steuern können:

- **`break`**; beendet den aktuellen Loop sofort. Zum Beispiel hilfreich, wenn ein Fehler auftritt.
- **`next`**; beendet den aktuellen Durchgang und beginnt den nächsten Durchgang vom Beginn des Loops (i wird also i+1)

Next-Beispiel

```
m=20

for (k in 1:m){
  if (!k %% 2)
    next
  print(k)
}
```

```
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
[1] 13
[1] 15
[1] 17
[1] 19
```

Fazit: Schleifen

Es gibt **drei verschiedene Varianten** Schleifen zu schreiben:

- `For`

- While
- Repeat

Und **zwei Befehle** um die Schleifen zu steuern:

- break
- next

Eigene Funktionen schreiben

Wir können mehrere Arbeitsschritte in R kombinieren und eigene Funktionen schreiben.

Zum Beispiel, wenn wir mehrmals dieselben Cleaning-Schritte für unsere Datensätze ausführen wollen.

Funktionen in R sind einfach aufgebaut, wir kennen das Muster schon:

```
function(ARGUMENTE) {BODY}
```

Eine Funktion, die keinem Namen zugewiesen wird, heißt “anonyme Funktion”. In der Regel sind diese Funktionen nur eine Zeile lang. Beachtet die Klammern um die Funktion:

```
(function(ARGUMENTE) {BODY})
```

Best Practise: Benennt eure eigenen Funktionen anders, als bereits bestehende Funktionen. Das macht nur Ärger.

Formals, Body, Environment

Beim Erstellen einer Funktion passieren drei Sachen:

- formals() (oder: Argumente) wird explizit angegeben
- body() wird explizit angegeben
- environment() wird implizit angegeben

Die Funktionselemente

```
f01 <- function(x, y) {
  x + y
}
```

```
formals(f01)
```

```
$x
```

```
$y
```

```
body(f01)
```

```
{
  x + y
}
```

```
environment(f01)
```

```
<environment: R_GlobalEnv>
```

f01

```
> f01()
```

```
Error in f01() : argument "x" is missing, with no default
```

```
f01(x = 1, 5) # y kann, muss aber nicht angegeben werden, weil die Position vorgegeben ist
```

```
[1] 6
```

Argumente

Unsere Funktion kann Argumente enthalten. Diese können, müssen aber nicht, einen Defaultwert haben.

```
function(x, y = 10)
```

x: Argument *x* ohne default

y: Argument *y* mit default 10.

Ein Hinweis: Normalerweise geben wir die Argumente in unsere Funktion. Das müssen wir aber nicht.

Wenn die Argumente in einer Liste vorliegen, können wir `do.call()` benutzen:

```
args <- list(1:10, na.rm = TRUE)
```

```
do.call(mean, args)
```

```
[1] 5.5
```

Body

Im Body wird ganz normal mit den Variablen gerechnet.

`missing()` überprüft, ob die Variablen vorhanden sind (falls kein default festgelegt wurde).

Eine gute **Vorgehensweise** für Funktionen: Löst ein Problem erst an einem konkreten Datensatz. Dann generiert daraus die Variablen, um das Problem zu abstrahieren.

Faustformel: Wenn ihr etwas dreimal im Code wiederholen müssten, schreibt eine Funktion.

Return

Viele Programmiersprachen nutzen `return x` am Ende einer Funktion. Das gibt den Wert der Variable *X* an die Funktion zurück. In R ist das nicht nötig, schafft aber mehr Übersicht und macht den Wert fürs Weiterarbeiten zugänglich.

Mit Ausgabe am Ende (*z* oder `return(z)`) macht das gleiche)

```
test_funct <- function(x = 2, y = 5) {  
  z <- x * y  
  return(z) # oder z ohne return  
}  
test_funct()
```

```
[1] 10
```

Ohne Ausgabe am Ende:

```
test_funct <- function(x = 2, y = 5) {  
  z <- x * y  
}  
test_funct()
```

Environment

Globale Umgebung:

```
> ls(environment())  
[1] "f01"
```

Globale vs. lokale Umgebung:

```
x <- 10  
f02 <- function(y) {  
  sum(y * x)  
}  
f02(4)
```

```
[1] 40
```

Warum wird die Variable x beachtet, obwohl sie nicht in der Funktion steht?

Welchen Wert nimmt y an, wenn ich sie außerhalb der Funktion ausgabe?

Globale vs. lokale Umgebung

```
> x <- 10  
> f02 <- function(y) {  
+   sum(y * x)  
+ }  
> y  
Error: object 'y' not found
```

Die Variable y ist nicht in der globalen Umgebung, nur in der lokalen Umgebung der Funktion f02 (genannt: Scope der Funktion).

Nachteil: Wir können sie nicht einfach aufrufen.

Vorteil: Die Variable funkt uns nicht in andere Funktionen dazwischen.

```
x <- 10
f02 <- function(y) {
  x <- 5
  sum(y * x)
}
```

Was passiert hier?

=====

```
x <- 10
f02 <- function(y) {
  x <- 5
  sum(y * x)
}
f02(4)
```

```
[1] 20
```

Funktion exportieren

Die einfachste Variante, eine Funktion zu exportieren, ist, sie als *.R-Datei zu speichern.

Im anderen Skript wird sie so aufgerufen und der dortigen globalen Umgebung hinzugefügt:

```
source("FUNKTION.R")
```

Hinweis: Man kann auch mehrere Funktionen in eine *.R-Datei schreiben.

Funktional vs. Objektorientiert: Programmierparadigmen

R ist eine funktionale Programmiersprache. Das bedeutet:

- die ganze Sprache ist in Funktionen aufgebaut. Auch Objekte sind Funktionen.
- der Code ist näher am Problem, weil nicht alles als Objekt modelliert werden muss.
- der Code ist meistens kürzer und damit weniger fehleranfällig

“You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.”

Hadley Wickham, R-Guru

Der Gegensatz dazu sind objektorientierte Programmiersprachen, wie Java, C++. Dort sind alle Daten und Funktionen in Objekten untergebracht. Die Objekte können Funktionen aufrufen, die ihnen zugeordnet sind.

Python kann beides.

Übung: Funktionen

Wir haben den folgenden Datensatz:

```
set.seed(42)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
df
```

```
   a b  c  d e f
1 -99 9 -99  6 1 9
2 -99 2  3  7 6 9
3  4 8  6 10 5 5
4 10 8 -99  2 10 8
5  8 6 -99 -99 5 1
6  6 8  2 -99 10 10
```

Schreibt eine Funktion, die alle -99 in NAs umwandelt. Ohne, dass ihr die Funktion für jede Spalte einzeln aufrufen müsst.

Lösung: Funktionen

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  return(x)
}

df[] <- lapply(df, fix_missing)

df
```

```
   a b  c  d e f
1 NA 9 NA  6 1 9
2 NA 2  3  7 6 9
3  4 8  6 10 5 5
4 10 8 NA  2 10 8
5  8 6 NA NA 5 1
6  6 8  2 NA 10 10
```

Weitere Übung Funktionen: Normalisierung

Ein typisches Problem: Wir wollen Werte normalisiert vergleichen. Zum Beispiel Mietpreise ab einem bestimmten Zeitpunkt.

Die Formel für Normalisierung ist einfach:

$$normalisiert = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

Stellt euch vor, wir haben einen Dataframe mit vier Spalten, die wir alle normalisieren wollen.

Wie würdet ihr das Problem angehen? Schreibt eine Funktion.

Aufgabe: Normalisierung

```
dataframe_normalize <- data.frame(  
  c1 = rnorm(50, 5, 1.5),  
  c2 = rnorm(50, 5, 1.5),  
  c3 = rnorm(50, 5, 1.5),  
  c4 = rnorm(50, 5, 1.5)  
)  
  
head(dataframe_normalize, n = 10)
```

	c1	c2	c3	c4
1	1.339300	6.381093	2.956826	5.127347
2	6.980170	6.081317	5.205884	6.343348
3	4.540042	3.435322	2.759562	4.655333
4	2.328037	4.864720	2.794346	6.254929
5	4.742124	5.935277	5.187054	2.382416
6	6.822012	3.569715	3.505041	7.534188
7	7.842790	4.185757	4.997266	6.297167
8	4.354296	5.871495	4.357612	4.773836
9	4.614096	6.152268	4.079493	2.826489
10	2.355255	5.695651	1.962983	5.964513

Lösung: Normalisierung I

Die Funktion für ein einzelnes Problem sieht so aus:

```
(data_frame$c1 - min(data_frame$c1)) / (max(data_frame$c1) - min(data_frame$c1))
```

Komplett abstrahiert:

```
normalize_x <- function(x){  
  nominator <- x - min(x)  
  denominator <- max(x) - min(x)  
  normalize <- nominator / denominator  
  return(normalize)  
}
```

```
dataframe_normalize[] <- lapply(dataframe_normalize, normalize_x)
```

Lösung: Normalisierung II

```
head(dataframe_normalize, n = 10)
```

	c1	c2	c3	c4
1	0.1130506	0.5917565	0.1912412	0.51369129
2	0.8823554	0.5459506	0.6240188	0.71334006
3	0.5495695	0.1416414	0.1532826	0.43619378
4	0.2478951	0.3600542	0.1599760	0.69882289
5	0.5771295	0.5236357	0.6203953	0.06301573
6	0.8607857	0.1621768	0.2967322	0.90885773
7	1.0000000	0.2563082	0.5838752	0.70575778

```
8 0.5242374 0.5138897 0.4607890 0.45565019
9 0.5596690 0.5567920 0.4072716 0.13592572
10 0.2516071 0.4870207 0.0000000 0.65114111
```

Die R-Community

Einer der Faktoren, warum R-Lernen so einfach ist.

Es gibt:

- zig Tutorials zu fast jedem Thema
- massenhaft beantwortete Fragen auf Stack Overflow
- interaktive Lernplattformen (teilweise kostenlos)
- viele Bücher zum Lernen
- in jeder größeren Stadt Meetups
- zahlreiche Konferenzen

Anlaufstellen für Tutorials

R Tutorials

R Bloggers Auch was zum “Auf dem Laufenden bleiben” am Thema R

R Statistics

Für Journalisten:

R for Journalists Tutorial

Intro to R for Journalists Mooc

R for Journalists Blog

Stack Overflow

Entwicklercommunity für alle Programmiersprachen, auch für R: Dort wurde jede Frage schon mal gefragt, jedes Problem schon mal besprochen.

Die Suche funktioniert am besten über die Fehlermeldung + R oder ihr formuliert euer Problem auf Englisch. Achtet auf die richtigen Fachbegriffe

Tipp: Erstellt euch einen Stack Overflow-Account, irgendwann werdet ihr eine Frage dort reinschreiben müssen. Dann achtet auf folgendes: Schildert euer Problem und Ziel. Postet eure Funktionen und so viele Daten, dass man das Problem nachvollziehen - und euch helfen kann. Normalerweise geht das sehr schnell und die Leute sind sehr nett (zumindest in der R-Community).

Irgendwann könnt ihr auch anderen Leuten bei ihren Problemen helfen.

Alternative: CrossValidated

Auf dem Stand bleiben

Rstudio Webinare

R Weekly

R Studio Cheatsheets

Weiterlernen Interaktiv

Datacamp €€€

Swirl

Moocs gibt es auch für spezielle probleme: Machine Learning, Inferenz-Statistik

Data Analysis with R by Facebook vielleicht was gegen Ende des Jahres. Viele Basics, aber spannende Einblicke in Facebooks R&D-Abteilung

Weiterlernen Bücher

Grolemund, Wickham: R for Data Science (Onlineversion hier)

Wickham: Advanced R (Online hier)

Sharon Machlis: Practical R for Mass Communication and Journalism (Auf Amazon, Online nur Auszüge)

Teetor: R Cookbook (Online hier)

Meetups

München: Applied R

Berlin: R Users Group oder BerlinR

Hamburg: R Users Group

Köln: R Users Group

R Ladies: Berlin, München, Frankfurt, Freiburg

Dort gilt: Einfach vorbeikommen. Meistens freuen die sich riesig über Leute ausserhalb der Szene.

Konferenzen

Use R! internationale, Haupt-Konferenz der R Community. 2019 in Toulouse. 2020 in St. Louis

eRum das europäische R-Nutzertreffen, das nur stattfindet, wenn “Use R!” außerhalb Europas abgehalten wird.

RStudio Conference

SatRdays