Robustheit

Max Brede und Johannes Andres

2021-11-08

Contents

Chapter 1

Vorwort

Dieses mit `bookdown` erstellte Dokument ist das Skript zum Seminar "psyM9-1: Psychologische Forschungsmethoden. Projektseminar I" und "PSY_B_20_e-1: Forschungsorientierte Vertiefung: Forschungsmethoden" der CAU zu Kiel.

Chapter 2

Lehrplan

2.1 Semesterplan

Sitzung	Datum	Sitzungstitel	Lernziele
			Die Studierenden…
1	2021-10-25	Rste Codeschnipsel	können Funktionen in R definieren, deren Bestandteile nennen und diese benutzen.
2	2021-11-01	Iterationen und Zufallszahlen	wissen, was if-Statements sind und können diese in R einsetzen wissen, was for- und while-Schleifen sind und können diese in R einsetzen können Werte von Zufallszahlen in R erzeugen
3	2021-11-08	Listen und Matrizen	wissen, wie eine „list“ in R aufgebaut ist und können diese benutzen können Matrizen in R benutzen
4	2021-11-15	Welche Iterationen?	kennen verschiedene Arten von Rs iterativen „functionals“ und können diese benutzen können basales microbenchmarking einsetzen um Funktionsperformance zu evaluieren
5	2021-11-22	Beispiele	haben erste Erfahrungen mit der Durchführung von Simulationsstudien in R
6	2021-11-29	Beispiele	
7	2021-12-06	Psychopy I	können einfache Reaktionszeit- und Rating-Experimente in Psychopy erstellen
8	2021-12-13	Psychopy II	können Ergebnisse von mit Psychopy und Pavlovia durchgeführten Studien in R einlesen und aufbereiten
9	2021-12-20	Beispiele	haben erste Erfahrungen mit der Durchführung von Experimenten in Psychopy und Pavlovia
10	2022-01-10	Puffer	
11	2022-01-17	Puffer	
12	2022-01-24	Puffer	
13	2022-01-31	Puffer	

Part I

Programmieren in R

Chapter 3

Funktionen

Wir kennen Funktionen ja schon aus den EDV-Veranstaltungen.

Zum Beispiel macht die `sum`-Funktion mit einem Vektor das, was der Name sagt:

```
sum(c(1,2,3))
```

```
## [1] 6
```

```
1 + 2 + 3
```

```
## [1] 6
```

Funktionen können wir auch selbst definieren. Eine Funktion, die uns begrüßt könnte zum Beispiel wie folgt aussehen:

```
greet_me <- function(){
  return('Hello! Nice to see you!')
}
```

Wenn wir jetzt die `greet_me`-Funktion aufrufen, sehen wir:

```
greet_me()
```

```
## [1] "Hello! Nice to see you!"
```

In der Funktionsdefinition können wir ein paar Teile wiederkennen. Zum Einen ist da der `greet_me <-`-Teil, den wir ja schon als Objektzuweisung kennen. Wir weisen also dem Ergebnis eines Ausdrucks den Namen `greet_me` zu.

Funktions-*Objekte* werden also genauso wie Datensätze und Vektoren als eine Kombination von Namen und zugehörigem Inhalt definiert.

Dabei erstellt die Funktion `function()` einen Objektinhalt, der aus *body* und *formals* besteht und in einem *environment* definiert ist.

Der *body* ist der Teil der Funktion, der definiert, was passieren soll und wird in R in geschweiften Klammern hinter der `function`-Funktion definiert In unserem Beispiel besteht der body aus dem Aufruf, einen Text zurückzugeben:

```
body(greet_me)
```

```
## {
##     return("Hello! Nice to see you!")
## }
```

Die *formals* sind die Argumente, die bei der Ausführung des *body*s genutzt werden sollen.

In unserem Beispiel haben wir noch keine Argumente berücksichtigt:

```
formals(greet_me)
```

```
## NULL
```

Wir könnten die Funktion aber neu definieren, so dass sie einen gegebenen Namen begrüßt. Dafür geben wir der `function`-Funktion ein Argument, das der Name des erwarteten Arguments sein soll:

```
greet_someone <- function(name){
  return(paste0('Hello ',name,'! Nice to see you!'))
}

greet_someone('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!"
```

Wenn wir uns jetzt nochmal die `formals` ausgeben lassen, sehen wir, dass ein Argument vorgesehen ist:

```
formals(greet_someone)
```

```
## $name
```

Der letzte Teil einer Funktionsdefinition ist das *environment*. Damit ist der Namensraum gemeint, auf den die Funktion zugreifen kann:

```
environment(greet_someone)
```

```
## <environment: R_GlobalEnv>
```

In unserem Beispiel wurde die Funktion in der laufenden R-Session definiert (wie die allermeisten Funktionen, die wir dieses Semester nutzen werden). Der Output `R_GlobalEnv` heißt also, dass die Funktion auf Objekte in der Haupt-R-Session zurückgreifen kann.

Praktisch heißt das, dass wir zum Beispiel Konstanten einmal außerhalb einer Funktion definieren können, die diese dann nutzen kann:

```
n <- 10

greet_someone_n_times <- function(name) {
  return(rep(paste0('Hello ',
                    name,
                    '! Nice to see you!'),
             n))
}

greet_someone_n_times('Marvin')
```

```
##  [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [3] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [5] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [7] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
##  [9] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
```

Was das aber nicht heißt ist, dass die Funktion eine externe Variable ändern kann:

```
greet_someone_n_times <- function(name) {

  n <- 3

  return(rep(paste0('Hello ',
                    name,
                    '! Nice to see you!'),
             n))
}

greet_someone_n_times('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

```
n
```

```
## [1] 10
```

Das liegt daran, dass die Funktion beim Aufrufen eine Kopie des globalen Environments als eigene Variablenumgebung zugewiesen bekommt und diese Kopie auch verändern kann. Die Kopie wird aber bei Beenden der Funktion verworfen, so dass Änderungen in der Funktion nicht erhalten bleiben.

Dabei werden die Argumente der Funktion dem Environment der Funktion hinzugefügt, wobei im Zweifelsfall gleichnamige Objekte für die Funktion überschrieben werden:

```
x <- 1:10
y <- 11:20

my_function <- function(x,y){
  return(c(x,y))
}

my_function(21,22)
```

```
## [1] 21 22
```

3.0.1 Aufgabe

1. Erstelle eine Funktion mit dem Namen `my_mean`, die den Mittelwert eines gegebenen Vektors berechnet.
2. Was ergibt der folgende Aufruf:

```
dummy_function <- function(a,b){
  a * b
}

c <- dummy_function(1,2)

formals(dummy_function)
```

3. Was ist am Ende des folgenden Aufrufs in `number` abgelegt?

```
number <- 'a number'

important_calculation <- function(number){
  number <- 42 / number * number
  return(number)
}

important_calculation(5)
```

```
## [1] 42
```

Antworten

1.

```
my_mean <- function(x){
  return(sum(x)/length(x))
}
```

2.

```
## $a
```

```
##
##
## $b
```

3.

```
'a number'
```

3.1 Geschachtelte Funktionen

Um den Code übersichtlicher zu halten, können Teile von Funktionen in andere Funktionen ausgegliedert werden. Dabei nutzen wir, dass unsere Funktionen auch auf das global Environment zugreifen und Funktionen ja auch nur Arten von Objekten sind.

Wenn wir unsere Funktionen `greet_someone` und `greet_someone_n_times` von vorhin nochmal angucken sehen wir, dass ein Teil des Codes in beiden auftaucht:

```
print(greet_someone)
```

```
## function(name){
##   return(paste0('Hello ',name,'! Nice to see you!'))
## }
```

```
print(greet_someone_n_times)
```

```
## function(name) {
##
##   n <- 3
##
##   return(rep(paste0('Hello ',
##                     name,
##                     '! Nice to see you!'),
##              n))
## }
```

Wir können `greet_someone_n_times` jetzt so umschreiben, dass sie `greet_someone` benutzt:

```
greet_someone_n_times <- function(name){
  n <- 3

  return(rep(greet_someone(name),
             n))
}


greet_someone_n_times('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

3.1.1 Aufgabe

1. Erstelle eine `my_var` Funktion, die die unkorrigierte Varianz eines Vektors ($S^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - M_X)^2$) berechnet. Benutze für den Mittelwert deine Mittelwerts-Funktion aus dem letzten Aufgaben-Block.

Antworten

1.

```
my_var <- function(x){
  m_x <- my_mean(x)

  x <- (x - m_x)^2

  return(1/length(x) * sum(x))
}
```

3.2 Optionale Argumente

Unsere `greet_someone_n_times`-Funktion sieht ja im Moment wie folgt aus:

```
print(greet_someone_n_times)
```

```
## function(name){
##   n <- 3
##
##   return(rep(greet_someone(name),
##              n))
## }
```

Ungünstig daran ist noch, dass das `n` bei jedem Aufruf als Konstante neu definiert ist.

Um dieses n anpassen zu können, können wir es als zweites Argument übergeben, das beim Aufruf festgelegt werden kann:

```
greet_someone_n_times <- function(name, n){
  return(rep(greet_someone(name),
             n))
}

greet_someone_n_times('Marvin',3)
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

Da wir uns aber das Tippen sparen wollen und nicht jedes Mal unseren Standard-Fall (n=3) explizit machen wollen, können wir dem Argument einen Standardwert zuweisen.

Dadurch machen wir aus dem `n` ein *optionales Argument*, wie wir es ja auch schon aus anderen Situationen kennen:

```
greet_someone_n_times <- function(name, n=3){
  return(rep(greet_someone(name),
             n))
}

greet_someone_n_times('Marvin')
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!"
```

```
greet_someone_n_times('Marvin', 5)
```

```
## [1] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [3] "Hello Marvin! Nice to see you!" "Hello Marvin! Nice to see you!"
## [5] "Hello Marvin! Nice to see you!"
```

Chapter 4

`if`-Statements

Als letzten Schritt wollen wir besonders höflich zu speziellen Nutzern sein. So soll der Name 'Justus'[1] ausschweifender gegrüßt werden, als alle anderen Namen.

Um das umsetzten zu können, müssen wir irgendwie einen Test einfügen, ob der gegebene Name gleich 'Justus' ist.

Dafür können wir ein so genanntes `if...else`-Statement nutzen. Die erste Hälfte, das `if`-Statement ermöglicht es uns, besonderes Verhalten auszulösen, wenn eine logische Bedingung erfüllt ist. Das `else` danach können wir nutzen um jeden anderen Fall zu definieren:

```
greet_someone <- function(name){
  if(name == 'Justus'){
    return(paste0('Hello ',name,'! How extraordinarily nice to see you! I hope you are doing well
  }else{
    return(paste0('Hello ',name,'! Nice to see you!'))
  }
}
```

Das funktioniert zwar:

```
greet_someone('Justus')
```

```
## [1] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
```

```
greet_someone('Jonas')
```

```
## [1] "Hello Jonas! Nice to see you!"
```

Und bemerkensweiterweise auch in der geschachtelten Form:

[1] Nicht, dass er uns verklagt.

```
greet_someone_n_times('Justus')
```

```
## [1] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
## [2] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
## [3] "Hello Justus! How extraordinarily nice to see you! I hope you are doing well!"
```

Aber wirklich gut lesbar ist das nicht unbedingt. Ein gängiger Ansatz, um die Lesbarkeit einer solchen Funktion mit verschiedenen Outputs zu verbessern, ist, nur ein `return`-statement ans Ende der Funktion zu stellen und die Teil-Änderung durch das `if`-statement in einem Objekt abzulegen, das im gemeinsamen `return` genutzt wird. So könnten wir den letzten Teil des Grußes zum Beispiel in einem Objekt namens `text` ablegen und je nach Nutzer anpassen:

```
greet_someone <- function(name){
  if(name == 'Justus'){
    text <- '! How extraordinarily nice to see you! I hope you are doing well!'
  }else{
    text <- '! Nice to see you!'
  }
  return(paste0('Hello ',name,text))
}
```

Ein letzter Trick, um diese Funktion leichter lesbar zu gestalten, ist den Regelfall *vor* das `if`-statement zu stellen und sich so das `else` zu sparen. Da die Anweisung im `if`-statement nur evaluiert wird, wenn der Test positiv aufgeht, ist das Ergebnis der folgenden Funktion äquivalent:

```
greet_someone <- function(name){
  text <- '! Nice to see you!'

  if(name == 'Justus'){
    text <- '! How extraordinarily nice to see you! I hope you are doing well!'
  }

  return(paste0('Hello ',name,text))
}
```

4.0.1 Aufgabe

1. Erweitere die `my_var`-Funktion um ein optionales `corrected` Argument, das standardmäßig auf `TRUE` gesetzt ist. Die Funktion soll, wenn dieses Argument auf `TRUE` gesetzt ist, die korrigierte($s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - M_X)^2$); wenn es auf `FALSE` gesetzt wird die unkorrigierte ($S^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - M_X)^2$) Stichprobenvarianz ausgeben.

Antworten

1.

```r
my_var <- function(x, corrected=TRUE){
  m_x <- my_mean(x)

  x <- (x - m_x)^2

  factor <- 1 / (length(x) - 1)

  if(!corrected){
    factor <- 1 / (length(x))
  }

  return(factor * sum(x))
}
```

Chapter 5

Zufallswerte

Mit R kann man sehr einfach quasi-zufällige[1] Wertreihen erstellen. Aus der Veranstaltung letztes Semester kennen wir ja schon die Funktion `sample`, die aus einer vorgegebenen Urne ziehen kann.

Für unsere Simulations-Probleme gibt es noch eine andere Reihe an Funktionen, die für eine feste Verteilungsklasse zufällige Wertfolgen erstellen können.

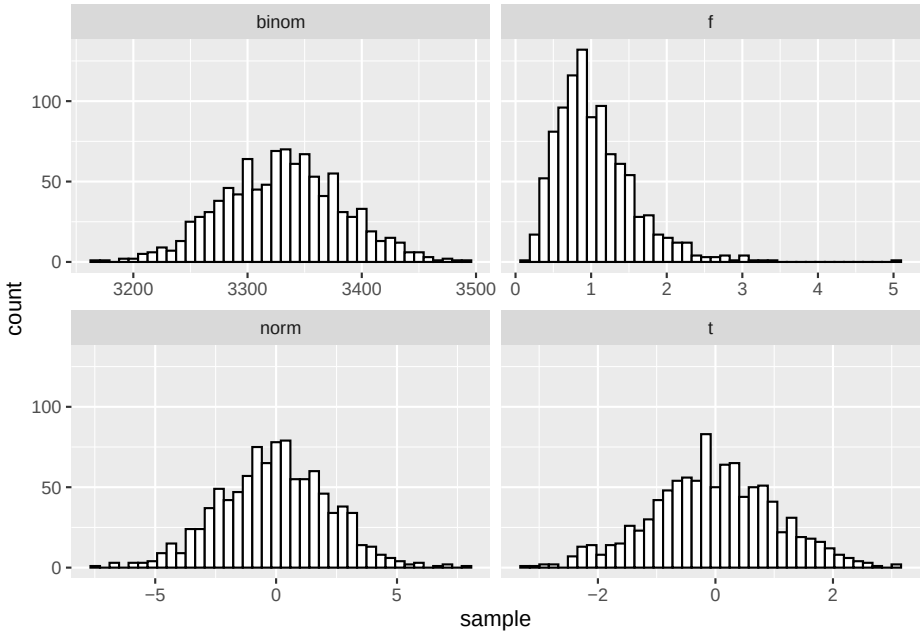
Diese Funktionen werden mit einem `r` für *random* und der Verteilungsklasse aufgerufen. Eine Übersicht der implementierten Verteilungen kann man mit `?Distributions` aufrufen.

Mit diesen Funktionen können wir uns zum Beispiel jeweils 1000 zufällige Werte aus einer $N(0,5)-$, einer $B(20000,1/6)-$, einer $t(50)$- und einer $F(12,36)-$Verteilung ziehen und in Histogrammen darstellen:

```
library(tidyverse)
set.seed(42)
tibble(norm = rnorm(1000,mean = 0,sd = sqrt(5)),
       binom = rbinom(1000, size = 20000,prob = 1/6),
       t = rt(1000,df = 50),
       f = rf(1000,df1 = 12,df2 = 36)) %>%
  pivot_longer(everything(),
               names_to = 'dist',
               values_to = 'sample') %>%
  ggplot(aes(x = sample)) +
  geom_histogram(fill = 'white',
```

[1] Es ist für einen Rechner sehr schwierig, wahren Zufall zu erzeugen. Auf der Hilfeseite `?Random` kann man sich eine Liste der Algorithmen anschauen, die R zur Generation quasi-zufälliger Wertfolgen nutzt. Diese nicht-wirklich-zufällige Natur der Wertfolgen hat aber auch den Vorteil, dass wir "zufällige" Ergebnisse reproduzierbar machen können. mit `set.seed()` können wir einen Startwert für die Zufalls-Generation festlegen, den andere R-Nutzer dann auch wählen können.

```
                  color = 'black',
                  bins = 40) +
  facet_wrap(~dist,scales = 'free_x')
```



5.0.1 Aufgabe

Erstelle eine Funktion `gen_distributed_values`, die mit Hilfe von `if`-Statements anhand eines Arguments `distribution` und eines Arguments `n` einen Vektor an Zufallszahlen generiert. Dabei soll `distribution` angeben, welche aus drei möglichen Verteilungen genutzt wird. Denke außerdem an mögliche Verteilungseigenschaften der genutzten Funktionen und füge sie als optionale Argumente zur Funktion hinzu.

Antwort

```
gen_distributed_values <- function(distribution,
                                   n,
                                   df = 1,
                                   lambda = 25,
                                   rate = 5) {
  if(distribution == 't'){
    return(rt(n, df))
  }else if(distribution == 'pois'){
    return(rpois(n, lambda))
  }else if(distribution == 'exp'){
    return(rexp(n, rate))
```

```
  }
  print('Distribution is not implemented, returning zeros')
  return(numeric(n))
}

gen_distributed_values('pois',
                       5,
                       lambda = 3)
```

```
## [1] 4 4 3 5 2
```

Chapter 6

Schleifen

Bei so gut wie allen Simulationsproblemen stehen wir vor der Situation, dass wir eine Operation, zum Beispiel das Generieren einer gewissen Zahl an Zufallswerten und die Berechnung der zugehörigen Teststatistik, mehrere hundert Mal ausführen wollen.

Wir könnten jetzt die Operation mehrere hundert mal in unser Skript schreiben, damit würden wir aber zum Einen Fehler einladen, zum Anderen viel zu viel Lebenszeit verschwenden.

Deswegen gibt es in so gut wie jeder Programmiersprache irgendeine Form von Ausdrücken, die eine *iterative* Wiederholung eines Ausdrucks ermöglichen.

Ein Beispiel für die Anweisung einer solchen iterativen Wiederholung sind Schleifen. In R gibt es davon drei Arten:

- `repeat` - die flexibelste Schleife, die so lange wiederholt bis sie mit `break` unterbrochen wird
- `while` - eine Schleife, die zu Beginn jeder Wiederholung einen logischen Test durchführt und bei Zutreffen die Operation wiederholt
- `for` - die unflexibelste der drei Möglichkeiten. `for` iteriert einen Wert durch einen Vektor oder eine Liste, bis alle Einträge einmal dran waren. Das heißt, dass wir eine feste Laufzeit und damit den einfachsten Umgang haben, weswegen wir auch `for` in diesem Kurs verwenden werden.

Die `for`-Syntax sieht dabei wie folgt aus:

```
for(value in vector){
  do something
}
```

Um jetzt Beispielsweise alle Werte von eins bis zehn durch zu laufen und jeden Wert auszugeben können wir den folgenden Ausdruck benutzen:

```
for(i in seq_len(10)){
  print(i)
}
```

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

Wie man an diesem Beispiel schon sehen kann, können wir in der Schleife auf den `i`-Wert zugreifen. Das können wir zum Beispiel benutzen, um die ersten zehn Zahlen der Fibonacci-Reihe zu berechnen, in der jeder Wert die Summe der zwei vorhergegangenen ist:

```
fib <- numeric(10)
for(i in seq_along(fib)){
  if(i<3){ # die ersten zwei Stellen müssen Einsen sein
    fib[i] <- 1
  }else{
    fib[i] <- fib[i-1] + fib[i-2] # nimm die letzten zwei Einträge und summier sie auf
  }
}
fib
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

In diesem Beispiel ist noch ein weiteres Programmier-Prinzip sichtbar, die *Allokation* des Ergebnis-Vektors vor der Berechnung der Werte. Damit ist einfach gemeint, dass wir den leeren `fib`-Vektor erstellt haben, bevor wir mit der Schleife angefangen haben.

Der Grund für dieses Vorgehen ist, dass jedes Anlegen eines Vektors einer bestimtmen Größe ein bisschen Rechenzeit kostet. Wenn wir von vornherein festlegen, wie lang der Vektor werden soll, müssen wir nur einen Vektor anlegen. Wenn wir stattdessen wie im folgenden Beispiel in jeder Iteration den Vektor vergrößern, erstellt R implizit in jeder Iteration einen neuen Vektor.

```
fib <- 1
for(i in seq_len(10)){
  if(i<3){
    fib[i] <- 1
  }else{
```

```
    fib[i] <- fib[i-1] + fib[i-2]
  }
}
fib
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

Bei 10 Stellen ist der Unterschied noch nicht wirklich bemerkbar. Wenn wir jetzt aber das ganze in Funktionen verpacken und für längere Sequenzen laufen lassen und die Laufzeit stoppen sehen wir den Unterschied:

```
fib_alloc <- function(n){
  fib <- numeric(n)
  for(i in seq_along(fib)){
    if(i<3){
      fib[i] <- 1
    }else{
      fib[i] <- fib[i-1] + fib[i-2]
    }
  }
  return(fib)
}


fib_no_alloc <- function(n){
  fib <- 1
  for(i in seq_len(n)){
    if(i<3){
      fib[i] <- 1
    }else{
      fib[i] <- fib[i-1] + fib[i-2]
    }
  }
  return(fib)
}
start <- Sys.time()
a <- fib_alloc(100000)
runtime_alloc <- Sys.time() - start
start <- Sys.time()
b <- fib_no_alloc(100000)
runtime_no_alloc <- Sys.time() - start

runtime_alloc
```

```
## Time difference of 0.03683734 secs
```

```
runtime_no_alloc
```

```
## Time difference of 0.08184695 secs
```

Mehr als die Hälfte der Zeit geht für das Erstellen des neuen Vektors drauf!

6.0.1 Aufgabe

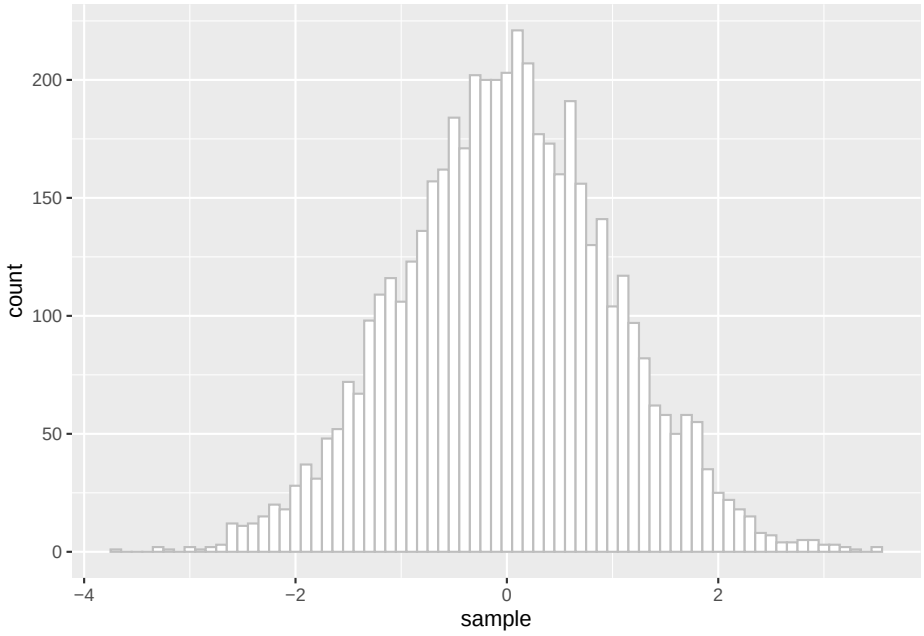
In QM-2 habt Ihr im Rahmen des zentralen Grenzwertsatzes gelernt, dass die Verteilungsfunktion der z-Transformation der n-ten Summe einer Reihe von unabhängigen Zufallsvariablen für wachsendes n schwach gegen die Standardnormalverteilung konvergiert.

Schreibe eine Funktion, die für eine gegebene Anzahl an Summen und eine gegebene Verteilungsklasse (und den entsprechenden Parametern inklusive der Stichprobengröße) einen Vektor mit den entsprechenden Summen zurückgibt. Nutze dafür deine Funktion aus der letzten Aufgabe.

Nutze diese Funktion dann um ein Histogramm mit 5000 dieser z-transformierten Summen zu erstellen.

Antwort

```
gen_central_lim_vec <- function(N,
                                distribution,
                                n,
                                df = 1,
                                lambda = 25,
                                rate = 5) {

  ret_vec <- numeric(N)
  for(i in seq_len(N)){
    ret_vec[i] <- sum(gen_distributed_values(distribution,
                                             n,
                                             df,
                                             lambda,
                                             rate))
  }
  return(ret_vec)
}

tibble(sample = scale(gen_central_lim_vec(5000, 'exp', 1000,rate = 5))) %>%
  ggplot(aes(x = sample)) +
  geom_histogram(binwidth = .1,
                 color = 'grey',
                 fill = 'white')
```


200
150
100
50
0
count
−4
−2
0
2
sample

Chapter 7

Listen

Listen sind uns aus der R-Übung schon im Rahmen der Gruppenweisen Aggregation mit `across` und als Output von Inferenzstatistischen Funktionen untergekommen. Aber was genau Listen architektonisch sind, haben wir bisher übergangen.

Listen sind in R ziemlich ähnlich zu Vektoren [1]. Sie bilden Aneinander-Kettungen von Objekten ab, bei denen wir die einzelnen Elemente benennen können. Ein Unterschied zu Vektoren ist aber, dass die Objekte nicht von einem einzelnen Typ sein müssen. Ein anderer Unterschied wird deutlich, wenn wir uns den Output von benannten Vektoren und Listen genauer anschauen.

```
c('a'= 1,
  'b'= 2)
```

```
## a b
## 1 2
```

```
list('a'= 1,
     'b'= 2)
```

```
## $a
## [1] 1
##
## $b
## [1] 2
```

Die Ausgabe der Liste ähnelt unter der jeweiligen Überschrift (z.B.: `$a`) dem Output, den wir bei einem unbenannten Vektor sehen:

[1] Wobei der Ausdruck "Vektoren" hier irreführend ist, in R wird eigentlich die Eltern-Klasse der Objekte `Vector` genannt, zu der `Atomic`(unsere "Vektoren") und `List` gehören. Da wir aber in den Grundlagenfächern wegen der mathematischen Analogie "Vektor" zu den `Atomic`-Objekten gesagt haben, behalten wir das hier bei.

```
c(1)
```

```
## [1] 1
```

Das liegt daran, dass in einer Liste unter dem Namen auch der ganze Vektor "abgespeichert" ist. Abgespeichert ist hier in Anführungszeichen, da in der Liste eigentlich nur ein Verweis auf einen Vektor liegt. Das kann man sich verdeutlichen, wenn man sich die Größen von Vektoren und Listen im Arbeitsspeicher anguckt.

Dazu erstellen wir einen Vektoren mit den Zahlen von 1:1000 und eine Liste, der wir dreimal diesen Vektor übergeben.

```
a <- c(1:1000)

b <- list(a = a, b = a, c = a, d = a)
```

Wenn wir uns jetzt die Größen der beiden Objekte angucken, sehen wir dass die Liste kleiner ist, als vielleicht zuerst erwartet:

```
lobstr::obj_size(a)
```

```
## 4,048 B
```

```
lobstr::obj_size(b)
```

```
## 4,544 B
```

Wenn wir die Liste mit einem vergleichbaren Vektor gegenüberstellen sehen wir, dass dieser die Werte offensichtlich direkt ablegt, wohingegen die Liste die Werte nur einmal beinhaltet (plus ein bisschen Speicher für die wiederholten Verweise und Namen):

```
d <- c(a, a, a, a)

lobstr::obj_size(a)
```

```
## 4,048 B
```

```
lobstr::obj_size(b)
```

```
## 4,544 B
```

```
lobstr::obj_size(d)
```

```
## 16,048 B
```

Was passiert nun, wenn wir einen Teil eines der vier Einträge in der Liste ändern?

Dazu können wir die schon von `data.frames` bekannte Index-Variante mit dem `$`-Operator nutzen um einen der Vektoren in der Liste zu modifizieren:

```
b$a[1] <- 5
lobstr::obj_size(b)
```

```
## 12,592 B
```

Die Liste wird größer. Dass sie nicht in Inkrementen von 4000 B größer wird, liegt daran, dass die Zahlen in `a` als Sequenz von R effizienter gespeichert werden können, als als einfache Zahlen. Wenn wir uns `a` alleine angucken und die Größe vor und nach Änderung der ersten Stelle betrachten, wird das deutlich:

```
lobstr::obj_size(a)
```

```
## 4,048 B
```

```
a[1] <- 5

lobstr::obj_size(a)
```

```
## 8,048 B
```

Für mehr Details zu diesem Speicher-Verhalten und dem zugrunde liegenden Prinzip ist das kostenlos hier zugängliche Buch "Advanced R" Wickham (2019) sehr gut, vor allem die Kapitel 2.2 und folgende und das Kapitel über Vektoren.

7.0.0.1 Aufgabe

Überlege dir, wie sich der Speicherbedarf der Liste `b` ändert, wenn Du den zweiten Platz der Liste mit dem ersten Platz der Liste überschreibst. Probiere dann aus, ob sich deine Vorhersage bewahrheitet. Überprüfe dann, was passiert, wenn du die erste Stelle des dritten Platzes der Liste durch fünf ersetzt. Was könnte hier passiert sein?

Antwort

```
a <- c(1:1000)
b <- list(a = a,b = a,c = a,d = a)
b$a[1] <- 5
lobstr::obj_size(b)
```

```
## 12,592 B
```

```
b$b <- b$a
lobstr::obj_size(b)
```

```
## 12,592 B
```

```
b$c[1] <- 5
lobstr::obj_size(b)
```

```
## 20,640 B
```

7.1 Listen und Datensätze

Das Arbeiten mit Listen ist ziemlich ähnlich zu der mit Datensätzen. Das liegt ganz einfach daran, dass Datensätze auch Listen sind, der einzige wirklich wichtige Unterschied ist, dass Datensätze im Vergleich zu Listen einheitliche Längen von den eingefügten Vektoren erwarten.

Zu sehen ist dieses Verhältnis ganz einfach, wenn man sich den `mode` eines Datensatzes anguckt:

```
a <- data.frame(1:10,
                1:10)

class(a)
```

```
## [1] "data.frame"
```

```
mode(a)
```

```
## [1] "list"
```

Der Unterschied ist in den `attributes` des Datensatzes festgelegt.

```
attributes(a)
```

```
## $names
## [1] "X1.10"   "X1.10.1"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Nur zum Spaß können wir so auch versuchen, umständlich einen Datensatz zu erstellen:

```
a <- list(1:10,1:10)
attributes(a)
```

```
## NULL
```

```
attributes(a) <- list(names = letters[1:2],
                      row.names = 1:10,
                      class = "data.frame")
a
```

```
##     a  b
## 1   1  1
## 2   2  2
## 3   3  3
## 4   4  4
```

```
## 5   5  5
## 6   6  6
## 7   7  7
## 8   8  8
## 9   9  9
## 10 10 10
```

7.1.0.1 Aufgabe

Überlege Dir, was wohl passiert, wenn du auf die gerade demonstrierte Art und Weise einen Datensatz erstellst, bei dem die eingefügten Vektoren von unterschiedlicher Länge sind. Überprüfe dann, ob die Erwartungen stimmen.

Antwort

```
a <- list(1:15,21:30,41:45)
attributes(a)
```

```
## NULL
```

```
attributes(a) <- list(names = letters[1:3],
                      row.names = 1:10,
                      class = "data.frame")
a
```

```
## Warning in format.data.frame(if (omit) x[seq_len(n0), , drop = FALSE] else x, :
## corrupt data frame: columns will be truncated or padded with NAs
```

```
##     a  b    c
## 1   1 21   41
## 2   2 22   42
## 3   3 23   43
## 4   4 24   44
## 5   5 25   45
## 6   6 26 <NA>
## 7   7 27 <NA>
## 8   8 28 <NA>
## 9   9 29 <NA>
## 10 10 30 <NA>
```

7.2 Arbeiten mit Listen

Da Datensätze eigentlich nur Listen sind, gibt es Listen-Operationen die wir schon von Datensätzen kennen, bei denen wir einfach noch nicht wussten, dass sie eigentlich aus dem Listen-Kontext stammen.

Insbesondere sind die Operationen, die wir auch schon hier im Skript genutzt haben, Listen-Operationen, die wir aus dem `data.frame`-Kontext kennen. Da

wären das Anlegen von Spalten/Listen-Einträgen mit Namen, wie wir es eben gesehen haben:

```
a <- list(a = 1:10)
b <- data.frame(a = 1:10)
```

Und das Indizieren mit dem $-Operator:

```
a$a
```

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

```
b$a
```

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

Als kleinen Zusatz können wir uns noch die numerische Indizierung angucken, die bei Listen und damit auch Datensätzen mit doppelten eckigen Klammern funktioniert (`[[]]`). Dieser Index-Operator ist hilfreich, wenn nicht jedes mal für jeden Eintrag ein Name angelegt werden soll. Das kann zum Beispiel sinnvoll bei `functional`-Iteratoren sein, bei denen man nur eine schnelle Stapelverarbeitung plant, dazu aber später mehr.

Bei Listen und Datensätzen sieht der `[[]]`-Einsatz dann so aus:

```
a[[1]]
```

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

```
b[[1]]
```

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

7.2.0.1 Aufgabe

Baue ein Skript, das eine Liste erstellt, die drei Einträge enthält. Jeder dieser Einträge soll auch wieder eine Liste sein. Das Skript soll nun mit Hilfe einer Schleife die Zahlen von 1 bis 100 durchgehen und alle durch 2 teilbaren Zahlen in den zweiten Eintrag, alle durch 3 teilbaren Zahlen in den dritten Eintrag und alle restlichen Zahlen in den ersten Eintrag einfügen. Nutze hierfür den "Modulo"-Operator `%%`, der den 'Rest' einer Ganzzahldivision ausgibt. Überlege dir, wie du mit Zahlen wie der 6 umgehst, die sowohl durch 3 als auch durch 2 teilbar sind. Kleiner Tipp: `length` kann hier sehr hilfreich sein.

Zusatzaufgabe: Überlege dir, wie man dieses Skript so umbauen könnte, dass es für einen Vektor mit beliebigen Teilern funktioniert.

Antwort

```
divisions <- list(list(), list(), list())

for(i in 1:100){
  if(i%%2 == 0){
```

```
    divisions[[2]][[length(divisions[[2]]) + 1]] <- i
  }
  if(i%%3 == 0){
    divisions[[3]][[length(divisions[[3]]) + 1]] <- i
  }
  if(i%%2 != 0 & i%%3 != 0){
    divisions[[1]][[length(divisions[[1]]) + 1]] <- i
  }
}
summary(divisions)
```

```
##      Length Class  Mode
## [1,] 33     -none- list
## [2,] 50     -none- list
## [3,] 33     -none- list
```

```
## Zusatz:
divisions <- list(list()) # mit einer leeren Liste für die nicht-teilbaren initiieren
divider <- c(2,3,5,7,9)
for(i in divider){
  divisions[[i]] <- list()
}
for(i in 1:100){
  divided <-  F
  for(j in divider){
    if(i %% j == 0){
      divisions[[j]][[length(divisions[[j]]) + 1]] <- i
      divided <- T
    }
  }
  if(!divided){
    divisions[[1]][[length(divisions[[1]]) + 1]] <- i
  }
}

summary(divisions)
```

```
##       Length Class  Mode
##  [1,] 22     -none- list
##  [2,] 50     -none- list
##  [3,] 33     -none- list
##  [4,]  0     -none- NULL
##  [5,] 20     -none- list
##  [6,]  0     -none- NULL
##  [7,] 14     -none- list
##  [8,]  0     -none- NULL
```

```
##  [9,] 11     -none- list
```

```
unlist(divisions[[1]]) # Mit Teilern und ohne 1 haben wir hier die Primzahlen bis 100 g
```

```
##  [1]  1 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Chapter 8

Attribute

Bei der sehr umständlichen Erstellung des Datensatzes haben wir schon die `attributes` kennen gelernt. Diese Informationen, die in R zu einem Objekt neben dem eigentlichen Inhalt angelegt werden können, sind ein wichtiger Teil der Abbildung objektorientierter Programmierparadigmen[1] in R.

```
a <- list(1:10,1:10)
attributes(a)
```

```
## NULL
```

```
attributes(a) <- list(names = letters[1:2],
                      row.names = 1:10,
                      class = "data.frame")
a
```

```
##     a  b
## 1   1  1
## 2   2  2
## 3   3  3
## 4   4  4
## 5   5  5
## 6   6  6
## 7   7  7
```

[1] Was genau objektorientierte Programmierung ist, ist hier erstmal nicht so wichtig, im Prinzip sind aber alle Objekte mit `class`-Attribut nach objektorientierten Paradigmen erstellt. Hier wird das nur erwähnt, weil es die Geschichte von R als Nachfolger von S demonstriert, in dem ursprünglich "moderne" Paradigmen nicht Thema waren. Die Erweiterung der bestehenden Objekte durch das Setzen von Attributen ist die Lösung für dieses Problem, die in Rs `S3`-Objekten resultieren. Daneben gibt es in R die `S4`-Objekte, die formalisiertere Klassendefinitionen ermöglichen. Das führt aber alles ein bisschen weit für diese Veranstaltung, deswegen sei hier nochmal auf das sehr gute Buch von Wickham (2019) verwiesen, in den Kapiteln 13-15 geht er auf OO-Programmierung in R ein.

```
## 8   8  8
## 9   9  9
## 10 10 10
```

Neben der Klasse (`class`) haben wir hier in die `attributes` auch die Namen der Einträge und die Zeilennamen geschrieben. Wir sehen also, dass der sichtbare und bemerkbare Unterschied zwischen Listen und Datensätzen aus dem Setzen dieser Attribute entsteht. Außerdem wird an diesem Beispiel deutlich, dass die Funktionen, die wir bisher zum Erstellen von Datensätzen genutzt haben, auch eigentlich nur Attribute setzen. Attribute sind also zusätzliche Informationen, die den Umgang mit Objekten ändern können.

Ein weiteres Beispiel für solche Änderungen durch Attribute sind Matritzen. In R kann man mit der `matrix`-Funktion zweidimensionale Daten-Raster erstellen, für die zum Beispiel auch die dimensionierte Indizierung `[,]` definiert ist:

```
matrix(1:100,
       nrow = 10,
       ncol = 10)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   43   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   56   66   76   86    96
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   18   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
```

8.0.0.1 Aufgabe

Erstelle eine Matrix und gucke die genau die Attribute und Inhalte an.

Können wir so wie wir einen Datensatz aus einer Liste erstellt haben, irgendwie ohne die `matrix`-Funktion eine Matrix erstellen? Gucke Dir dabei nach jeder Änderung an, was `class` und `mode` zurückgeben.

Probiere außerdem aus, was passiert, wenn man die Attribute einer Matrix auf eine Liste überträgt.

Zusatz:

Was passiert, wenn wir der Liste drei Dimensionen geben?

Antwort

```
my_mat <- matrix(letters[1:20],
                 4)
```

```
attributes(my_mat)
```

```
## $dim
## [1] 4 5
```

```
mode(my_mat)
```

```
## [1] "character"
```

```
class(my_mat)
```

```
## [1] "matrix" "array"
```

```
my_list_mat <- list(1:5,6:10,11:15,16:20)

attributes(my_list_mat) <- list(dim = c(2,2))
my_list_mat
```

```
##      [,1]      [,2]
## [1,] integer,5 integer,5
## [2,] integer,5 integer,5
```

```
mode(my_list_mat)
```

```
## [1] "list"
```

```
class(my_list_mat)
```

```
## [1] "matrix" "array"
```

```
## Zusatz
my_list_mat <- list(1, 2,
                    3, 4,
                    5, 6,
                    7, 8)

attributes(my_list_mat) <- list(dim = c(2,4))
my_list_mat
```

```
##      [,1] [,2] [,3] [,4]
## [1,] 1    3    5    7
## [2,] 2    4    6    8
```

```
mode(my_list_mat)
```

```
## [1] "list"
```

```
class(my_list_mat)
```

```
## [1] "matrix" "array"
```

```
attributes(my_list_mat) <- list(dim = c(2,2,2))
my_list_mat
```

```
## , , 1
##
##      [,1] [,2]
## [1,] 1    3
## [2,] 2    4
##
## , , 2
##
##      [,1] [,2]
## [1,] 5    7
## [2,] 6    8
```

```
mode(my_list_mat)
```

```
## [1] "list"
```

```
class(my_list_mat)
```

```
## [1] "array"
```

Mit anderen Worten ist eine Matrix als ein Array definiert, das (egal ob Liste oder Vektor), zwei Dimensionen hat. Mit mehr als zwei Dimensionen fällt die Matrix-Klasse weg.

Die Matrix-Listen aus der Aufgabe können wir übrigens auch wie Matrizen verwenden:

```
attributes(my_list_mat) <- list(dim = c(2,4))

t(my_list_mat)
```

```
##      [,1] [,2]
## [1,] 1    2
## [2,] 3    4
## [3,] 5    6
## [4,] 7    8
```

```
my_list_mat[1,2]
```

```
## [[1]]
## [1] 3
```

Das Ergebnis ist halt nur eine Liste, was mit dem bekannten doppel-Index aber auch umgangen werden kann:

```
my_list_mat[[1,2]]
```

```
## [1] 3
```

Chapter 9

Functionals

Wie wir am Anfang des Semesters bei der Definition unserer eigenen Funktionen ja schon gemerkt haben, sind Funktionen in R von der Struktur her gar nicht so verschieden zu anderen Objekten wie zum Beispiel Datensätzen oder Vektoren.

Sowohl die einen, wie auch die anderen, werden als Namen mit entsprechenden Inhalten im Environment angelegt, wobei aber natürlich bei Datensätzen die Daten damit gemeint sind und bei Funktionen *Environment*-Verweis, *Body* und *Formals* angelegt werden.

Das heißt für R aber auch, dass so genannte *Functionals* genutzt werden. Diese Gruppe von Funktionen sind solche, die andere Funktionen als Argumente und diese anwenden können. Beispiele für solche Funktionen haben wir auch schon kennen gelernt, die vielleicht aus dem Kapitel zur Aggregation von Daten in EDV1 bekannte `across`-Funktion ist ein solcher Fall:

```
iris %>%
  group_by(Species) %>%
  summarise(across(where(is.numeric),
                   .fns = list(m = mean, s = sd),
                   .names = '{.fn}_{.col}'))
```

```
## # A tibble: 3 x 9
##   Species    m_Sepal.Length s_Sepal.Length m_Sepal.Width s_Sepal.Width
##   <fct>               <dbl>          <dbl>         <dbl>         <dbl>
## 1 setosa               5.01          0.352          3.43         0.379
## 2 versicolor           5.94          0.516          2.77         0.314
## 3 virginica            6.59          0.636          2.97         0.322
## # ... with 4 more variables: m_Petal.Length <dbl>, s_Petal.Length <dbl>,
## #   m_Petal.Width <dbl>, s_Petal.Width <dbl>
```

Hier übergeben wir dem `.fns`-Argument eine Liste mit Funktionen, die dann auf alle numerischen Spalten des Datensatzes angewandt werden.

Ein anderes, sehr für R typisches Beispiel von Functionals sind `sapply`, `lapply` und `mapply`. Alle drei Funktionen nehmen als Input eins oder mehrere Objekte und eine Funktion, die auf jeden Eintrag de(s/r) übergebenen Objekte(s) angewandt werden soll.

Exemplarisch gucken wir uns `sapply` an, die Listen[1] als erstes Argument erwarten. Als Objekt nehmen wir `iris` (was ja wie wir gelernt habe eine aufgemotzte Liste ist) und lassen uns mit `sapply` für jede Spalte sagen, ob sie numerisch ist:

```
sapply(iris, is.numeric)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

Fällt Dir was auf? Wir haben gerade eine Funktion fünf mal ausgeführt, ohne ihr explizit ein Argument zu übergeben. Den (`is.numeric(iris$Sepal.Width)`,`is.numeric(iris$Sepal.L` etc., etc., …)-Teil hat uns `sapply` abgenommen. `sapply` ist sogar so weit gegangen, uns die Ergebnisse in einem praktischen Vektor wiederzugeben, hat also irgendwie einen Output für uns erstellt.

Diesen Vektor könnten wir jetzt als Index nutzen, um uns nur die numerischen Spalten ausgeben zu lassen und uns wieder mit `sapply` die Mittelwerte ausrechnen zu lassen:

```
sapply(iris[,which(sapply(iris, is.numeric))],
       mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Die `sapply`-Funktion macht also im Prinzip nicht viel anderes, als unsere `for`-Schleifen.

9.0.1 Aufgabe

Baue die oben mit `sapply` implementierte Mittlung mit `for`-Schleifen nach. Dabei soll erst überprüft werden, ob eine Spalte numerische Inhalte hat, wenn dem so ist soll der Mittelwert dieser Spalte berechnet werden. Verpacke deine Lösung in eine Funktion, die den spaltenweisen Mittelwert zurückgibt.

Überlege dir insbesondere, ob Du den Output-Vektor allozieren kannst und was ein sinnvolles Format dafür wäre.

Antwort

```
my_col_mean <- function(df){
  out <- rep(NA, ncol(df))
  names(out) <- names(df)
```

[1] oder Vektoren

```
  for(col in names(df)){
    if(is.numeric(df[[col]])){
      out[col] <- mean(df[[col]])
    }
  }
  return(out)
}
my_col_mean(iris)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##     5.843333     3.057333     3.758000     1.199333           NA
```

9.1 purrr::map

Eine Alternative zu den *apply-Funktionen ist die map-Familie aus dem purrr-Paket. Im Prinzip sind die identisch zu den *apply-Funktionen, der große Unterschied ist, dass sie spezialisierter sind als erstere.

Diese Eigenschaft ist auch der einzige wirkliche Grund dafür, die functional-Iteratoren den Schleifen vorzuziehen. Code, der so spezifisch für ein Problem wie möglich ist, ist einfach schöner weil schneller verständlich.

Die grundlegende Funktion aus der map-Familie ist das einfache map, das lapply-Analogon. map nimmt also eine Liste oder einen Vektor und eine Funktion als Input und gibt eine Liste zurück:

```
map(1:10, sqrt)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
##
```

```
## [[7]]
## [1] 2.645751
##
## [[8]]
## [1] 2.828427
##
## [[9]]
## [1] 3
##
## [[10]]
## [1] 3.162278
```

Alternativ kann mit der sogenannten Funktionsschreibweise auch ein Ausdruck formuliert werden, der für jeden iterierten Wert ausgeführt werden soll:

```
map(1:10, ~sqrt(.))
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
##
## [[7]]
## [1] 2.645751
##
## [[8]]
## [1] 2.828427
##
## [[9]]
## [1] 3
##
## [[10]]
## [1] 3.162278
```

Der besondere Vorteil dieser Funktionen gegenüber den `*apply`-Funktionen, ist dass diese Gruppe von Funktionen ermöglicht, den erwarteten Output einer Iteration klar lesbar zu definieren.

Beispielsweise kann mit `map_dbl`, `map_lgl` und `map_chr` klar festgelegt werden, dass 1. ein `atomic`-Vektor ausgegeben wird, der 2. einen klaren Datentyp hat:

```
map_dbl(1:10, ~.^2)
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
map_lgl(1:10, ~.%%2==0)
```

```
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

```
map_chr(1:10, ~letters[.])
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Und, noch praktischer, auch für komplexere Datentypen gibt es Wrapper. So können wir zum Beispiel mit `map_dfr` und `map_dfc` `tibble`s erzeugen lassen, die dann aus zeilenweise (`dfr` für rows) und spaltenweise (`dfc` für columns) zusammengefügten Ergebnissen bestehen:

```
map_dfr(1:10, ~tibble(i = .,
                      x = sample(1:10,1)))
```

```
## # A tibble: 10 x 2
##        i     x
##    <int> <int>
##  1     1    10
##  2     2     7
##  3     3     3
##  4     4     5
##  5     5     3
##  6     6     7
##  7     7     5
##  8     8    10
##  9     9     1
## 10    10     3
```

```
map_dfc(1:10, ~c(sample(.:(.+10),1)))
```

```
## New names:
## * NA -> ...1
## * NA -> ...2
## * NA -> ...3
## * NA -> ...4
## * NA -> ...5
## * ...
```

```
## # A tibble: 1 x 10
##    ...1  ...2  ...3  ...4  ...5  ...6  ...7  ...8  ...9 ...10
##   <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
## 1     5     8     3     4    15     9    11    15    13    10
```

9.2 Exkurs: Iteratoren-Vergleich und Microbenchmarking

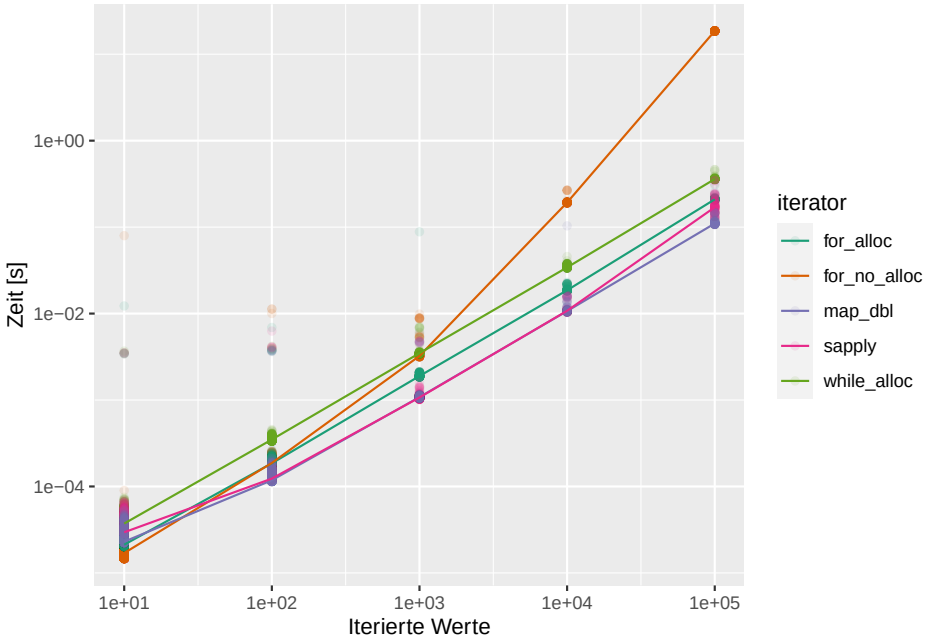
Man liest manchmal, dass loops in R wegen geringerer Geschwindigkeit in jedem Fall vermieden werden sollten. Das stimmt aber nicht wirklich, wenn man für seine `for`-Schleifen alloziert.

Um das zu zeigen vergleichen wir mit der folgenden Funktion `for` ohne Allokation, for mit Allokation, `sapply` und `map_dbl`. Um die Laufzeiten zu vergleichen, nutzen wir die `bench::mark`- und `bench::press`-Funktion, sehr akkurate Funktion um Microbenchmarking durchzuführen. Microbenchmarking heißt hier nichts anders, als dass wir die möglichen Implementationen alle unter denselben Bedingungen sehr oft ausführen und die Ausführungszeiten notieren, um eine Idee von der Effizienz dieser zu erlangen.

```
my_function <- function(x) x^2


results <- bench::press(n = 1:5,
                        {
                          to_do <- 1:10 ^ n
                          bench::mark(
                            min_iterations = 100,

                            for_no_alloc = {
                              res <- c()
                              for (j in to_do) {
                                res <- c(res, my_function(j))
                              }
                              res
                            },

                            for_alloc = {
                              res <- numeric(length(to_do))
                              for (j in to_do) {
                                res[j] <- my_function(j)
                              }
                              res
                            },
```

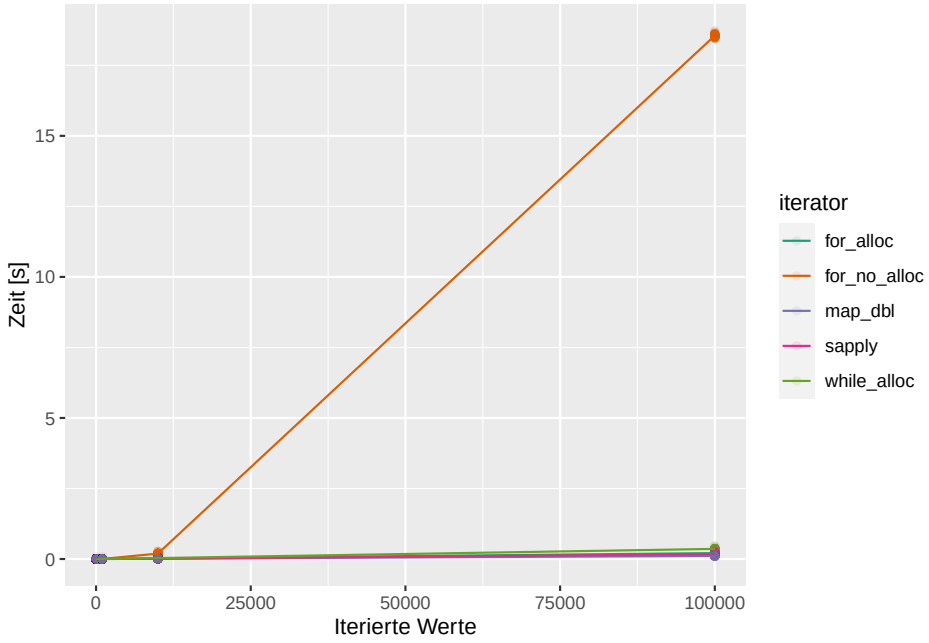
```
    while_alloc = {
      res <- numeric(length(to_do))
      j <- 1
      while(j < length(to_do)) {
        res[to_do[j]] <- my_function(to_do[j])
        j <- j + 1
      }
      res
    },

    sapply = {
      res <- sapply(to_do, my_function)
    },

    map_dbl = {
      res <- map_dbl(to_do, my_function)
    }
  )
})
```

Die Ergebnisse dieses Testlaufs sehen so aus:



Dabei stellen die Linien die Median-Verläufe der jeweiligen Aufrufe dar. Vorsicht: Die Achsen sind logarithmisch skaliert. Ohne diese Skalierung sehen die Ergebnisse so aus:

```
pmap_dfr(list(sim_data$time,
              names(sim_data$expression),
              sim_data$n),
         ~tibble(times = as.numeric(..1),
                 iterator = ..2,
                 reps = 10^..3)) %>%
  ggplot(aes(x = reps,
             y = times,
             color = iterator)) +
  geom_point(alpha = .1) +
  stat_summary(geom='line',
               fun = median) +
  labs(y = 'Zeit [s]',
       x = 'Iterierte Werte') +
  scale_color_brewer(palette = 'Dark2')
```



9.2.1 Aufgabe

Benutzt `bench::mark` um die folgenden drei Mittelwerts-Funktionen zu vergleichen:

```
my_for_mean <- function(x){
  out <- numeric(0)
  for(i in x){
    out <- out + i
```

```
  }
  return(out/length(x))
}

my_manual_mean <- function(x){
  return(sum(x)/length(x))
}

mean(x)
```

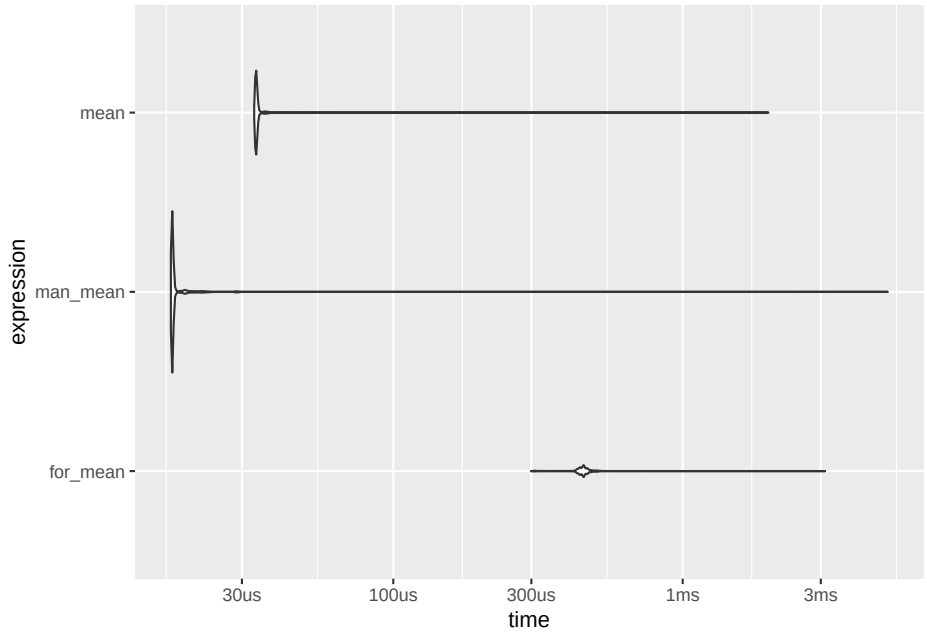
Antwort

```
x <- rnorm(10000)
bmark <- bench::mark(mean = mean(x),
              for_mean = my_for_mean(x),
              man_mean = my_manual_mean(x),
             iterations = 10000)
```

Übrigens gibt es noch eine sehr nette Funktion, die man im Kontext von `bench::mark` nochmal erwähnen sollte:

Mit `autoplot(type = 'violin')` lassen sich die Ergebnisse eines Mikrobenchmarks ganz nett darstellen:

```
library(bench)
bmark %>%
  autoplot(type = 'violin')
```



9.3 Eigene Functionals

Warum Functionals neben den Iteratoren für uns praktisch sind, wird am Simulations-Beispiel klar.

Wenn wir eine Funktion schreiben wollen, die zum Beispiel für einen t-Test das Ausschöpfen des Alpha-Niveaus bei verschiedenen Verteilungen überprüft, könnten wir wie bisher für eine Untermenge der Möglichkeiten mit if-else die richtige Funktion auswählen. Schöner wäre es aber doch, wenn wir eine Simulationsfunktion hätten, die einfach eine Funktion als Objekt nimmt und die Ergebnisse zurückgibt.

Dafür müssen wir erstmal verstehen, warum der folgende Aufruf funktioniert:

```
rnd_fct <- rnorm

rnd_fct(10)
```

```
##  [1] -0.4199126  0.2312317  0.4539680  0.2489447  1.0210670 -1.3311690
##  [7]  1.5289611  0.1749426 -0.8485907 -1.0582308
```

Wir haben in `rnd_fct` einen Verweis auf `rnorm` abgelegt und können den Namen des Verweises jetzt äquivalent zum Inhalt des Ziels, also ganz einfach wie die ursprüngliche Funktion verwenden.

Das funktioniert aus ähnlichen Gründen, aus denen unsere Listen mit Verweisen kleiner als erwartet waren. Da wir den Inhalt nicht verändert haben, spart sich R den Aufwand die Funktion zu kopieren und legt einfach einen zweiten Verweis auf die `rnorm`-Funktion an, inklusive `body` und `formals`.

```
body(rnorm)
```

```
## .Call(C_rnorm, n, mean, sd)
```

```
body(rnd_fct)
```

```
## .Call(C_rnorm, n, mean, sd)
```

```
formals(rnorm)
```

```
## $n
##
##
## $mean
## [1] 0
##
## $sd
## [1] 1
```

```
formals(rnd_fct)
```

```
## $n
```

```
##
##
## $mean
## [1] 0
##
## $sd
## [1] 1
```

Mit anderen Worten können wir einen Verweis auf ein Funktions-Objekt wie die eigentliche Funktion verwenden. Diesen Umstand können wir ausnutzen, wenn wir unsere Simulationsfunktion schreiben, indem wir einfach ein Argument vorsehen, dass als Funktion benutzt werden kann.

So eine Funktion könnte so aussehen:

```
generate_values <- function(rnd_fct = rnorm,
                            n = 1000){
  return(rnd_fct(n))
}

generate_values(n = 10)
```

```
##  [1]  0.2994693  0.5303306  2.3644090 -1.5828983  0.3834167 -0.8116823
##  [7] -1.1964809 -0.6136671  1.2368626  0.5172003
```

```
generate_values(runif, n = 10)
```

```
##  [1] 0.44395272 0.30926817 0.13344473 0.58773504 0.47318257 0.61197434
##  [7] 0.37293941 0.91837751 0.74896328 0.07670441
```

Und schon ist unser erster Functional fertig.

9.3.1 Aufgabe

Erstelle einen functional `summarise_for_me`, der einen Vektor als Argument x und eine Funktion als Argument `agg_fn` erwartet, und die Ihr dazu nutzen könnt, mit demselben Stichwort NA-bereinigt Summe, Mittelwert, Median und SD des Vektors `c(10, NA, 21, 25, 13)` zu berechnen.

Antwort

```
summarise_for_me <- function(x, agg_fn){
  return(agg_fn(x, na.rm = T))
}

x <- c(10, NA, 21, 25, 13)
summarise_for_me(x, mean)
```

```
## [1] 17.25
```

```
summarise_for_me(x, sum)
```

```
## [1] 69
```

```
summarise_for_me(x, median)
```

```
## [1] 17
```

```
summarise_for_me(x, sd)
```

```
## [1] 6.946222
```

9.4 Argumente durchreichen

Unsere schöne Funktion wird offensichtlich problematisch, wenn wir zum Beispiel `rf` übergeben:

```
generate_values(rf, n = 100)
```

```
## Error in rnd_fct(n): argument "df1" is missing, with no default
```

Dafür gibt es in R den Platzhalter `...`, dem Ihr auch schon an anderer Stelle begegnet sein könntet. `...` heißt nichts anderes als dass mehr Argumente möglich sind, bei denen wir noch nicht so ganz sicher sind, welche es sein werden.

Benutzen können wir das wie jedes andere Argument:

```
generate_values <- function(rnd_fct = rnorm,
                            n = 1000,
                            ...){
  return(rnd_fct(n, ...))
}

generate_values(rf, n = 100, df1 = 5, df2 = 10)
```

```
##   [1]  1.30800244  0.34386902  1.07185386  1.26556593  1.07752607  2.03497546
##   [7]  0.63611837  0.87353987  1.03485287  0.81601814  1.02639860  0.53705763
##  [13]  0.31405869  0.52439110  1.39648091  1.06064416  0.46664706  0.27070874
##  [19]  2.25204656  0.86366757  1.15693767  1.81259878  0.56410316  0.94787277
##  [25]  3.89529621  0.37557481  1.92586101  0.42849312  1.49773976  0.05813357
##  [31]  0.44438564  0.14525814  2.36696864  1.66339698  0.53827284  1.65650546
##  [37]  0.46055553  1.60033949  0.30694711  1.85127786  1.48217075  2.62585139
##  [43]  2.14504737  0.81919515  1.38374697  0.89745419  0.37185526  1.13406191
##  [49]  1.31479498  0.65300786  0.80247351  1.80981147  0.32507006  1.55078471
##  [55]  1.18369100  0.08447330  1.91984029  0.13914696  0.79177535  0.94549312
##  [61]  0.93354001  0.28173625  1.45672514  0.60835450  0.91438839  0.75916434
##  [67]  1.97458061  0.17009112  1.49389432  2.16806557  1.61570079  0.33946545
##  [73]  1.30805728  1.48765000  0.41400625  0.98439720  2.91565515  1.18407534
##  [79]  0.26514127  0.71282182  0.73094521  0.94654944  1.06797966 12.72549312
```

```
##  [85]  2.87107955  1.15050343  0.27958205  0.55015957  0.47310973  0.89918620
##  [91]  1.87930474  1.79708365  2.26899240  1.19920258  1.43975652  0.47004905
##  [97]  1.47975343  1.80849062  2.76730517  1.20793446
```

Sollten wir in dieser Liste noch für einen Sonderfall testen wollen (zum Beispiel um Sonderbehandlungen durchzuführen), können wir mit `hasArg` auf ein Argument testen:

```
generate_values <- function(rnd_fct = rnorm,
                            n = 1000,
                            ...){
  if(hasArg(df2)){
    cat('more than one df\n')
  }
  return(rnd_fct(n, ...))
}

generate_values(rf, n = 10, df1 = 5, df2 = 10)
```

```
## more than one df
```

```
##  [1] 1.7191709 0.3844587 0.3358765 0.4924094 2.3220831 0.8904054 1.5374226
##  [8] 0.3068908 1.7804266 0.1659158
```

Damit könnten wir auch eigene Argumente definieren, wenn wir die Funktionsdefinition übersichtlich halten wollen:

```
generate_values <- function(rnd_fct = rnorm,
                            n = 1000,
                            ...){
  if(hasArg(skew)){
    args <- list(...)
    skew <- args$skew
    args <- args[names(args) != 'skew']
    args$n <- n
    out <- do.call(rnd_fct, args)
    return(out + skew * out^2)
  }
  return(rnd_fct(n, ...))
}

generate_values(rf, n = 10, df1 = 5, df2 = 10, skew = 1)
```

```
##  [1] 1.4748042 2.4036888 0.3793993 1.9490523 4.1082832 1.2057001 0.5293067
##  [8] 0.8127482 0.2072046 4.8083358
```

So haben wir eine allgemeine Funktion geschrieben, die zum einen flexibel Daten mit übergebenen Funktionen generiert und zum anderen im Falle eines gesetzten Arguments `skew` dieses aus den Argumenten nimmt und das Ergebnis des

Funktionsarguments mit den restlichen ...-Argumenten mit dem Skew transformiert.

Auch nochmal zu betonen ist, dass wir den ...-Operator sowohl nutzen können, um die Argumente durchzureichen:

```
do_something <- function(.fn, ...){
  return(.fn(...))
}
do_something(mean, x = 1:10, trim = .1)
```

```
## [1] 5.5
```

als auch um eine Liste zu erstellen, die wir dann mit `do.call` als Argumentliste übergeben:

```
do_something <- function(.fn, ...){
  args <- list(...)
  return(do.call(.fn, args))
}
do_something(mean, x = 1:10, trim = .1)
```

```
## [1] 5.5
```

Letztere Methode ist dann praktisch, wenn wir entweder die Argumente ergänzen wollen, oder wie im Fall oben, nur Teile des Calls an eine Funktion weiterreichen wollen.

Da diese "impliziten" Argumente aber für `formals` und Autocomplete nicht zugänglich sind, außerdem das schreiben eines Arguments als expliziten Teil der Funktion einfacher ist, sollte außer bei sehr komplexen (z.B. grafischen) Funktionen auf diese Art der Argumente eher verzichtet werden.

Wir könnten die Verteilungs-Funktion mit Skew von oben ja auch (viel übersichtlicher und mit Standardwert) wie folgt schreiben:

```
generate_values <- function(rnd_fct = rnorm,
                            n = 1000,
                            skew = 0,
                            ...){
  out <- rnd_fct(n, ...)
  return(out + skew * out^2)
}

generate_values(rf, n = 10, df1 = 5, df2 = 10, skew = 1)
```

```
##  [1] 34.6888898  5.5226003  5.1995615 22.6796614 23.0399780  0.5764265
##  [7]  2.2664878 19.1496233  4.9516273  0.4087788
```

9.4.1 Aufgabe

Ergänze den functional `summarise_for_me` um einen `...`-Operator als Argument. Benutze diesen, um Argumente an die Funktionen weiterzugeben.

Baue außerdem eine Ausnahmebedingung in die Funktion ein, die falls das Argument `scale` auf TRUE gesetzt ist, die Daten vor der Aggregation z-transformiert.[2]

Stelle außerdem sicher, dass die Funktion unabhängig von der Eingabe des Nutzers `na.rm` auf `TRUE` setzt.

Antwort

```
summarise_for_me <- function(x, agg_fn, ...){
  args <- list(...)
  if(hasArg(scale)){
    if(args$scale){
      x <- scale(x)
      args <- args[!(names(args) == 'scale')]
    }
  }
  args <- args[!(names(args) == 'na.rm')]
  return(agg_fn(x, na.rm = T))
}

x <- c(10, NA, 21, 25, 13)
summarise_for_me(x, mean, scale = T)
```

```
## [1] 2.775558e-17
```

```
summarise_for_me(x, sum, scale = T)
```

```
## [1] 1.110223e-16
```

```
summarise_for_me(x, median, scale = T)
```

```
## [1] -0.03599079
```

```
summarise_for_me(x, sd, scale = F)
```

```
## [1] 6.946222
```

```
summarise_for_me(x, sd, scale = T)
```

```
## [1] 1
```

[2] Das funktioniert ganz einfach mit der `scale`-Funktion.

Bibliography

Wickham, H. (2019). *Advanced r*. CRC press.