Die nächsten Schritte

R für Psychologen v1.0.2

Jan Philipp Nolte

The Health and Life Science University

Inhaltsverzeichnis

1 Wo bekomme ich Hilfe?

Das schöne an Programmiersprachen ist, dass in der Regel irgendjemand schon einmal genau dasselbe Problem gehabt hat wie man selbst. Wenn man nach einer Fehlermeldung oder spezifischen Frage im Internet sucht, ist meist der erste Vorschlag das Forum Stackoverflow. Sollte noch niemand genau das selbe Problem gehabt haben, kann man auf Stackoverflow auch selbst eine Frage stellen. Die extrem zuvorkommende Community antwortet erfahrungsgemäß innerhalb von maximal vier Stunden. Häufig hat man aber auch bereits nach rund zehn Minuten eine Antwort von einem der fünf Millionen aktiven Forenmitgliedern. Außerdem gibt es für jedes Package und jede Funktion eine Dokumentation auf CRAN. Diese kann man entweder mit `F1` bei Öffnen der Funktion, per `?funktionsName` oder über die PDF Dokumentationen im Internet aufrufen.

2 Verschiedene Datenstrukturen

Bisher bestand unser Datensatz immer aus einem sogenannten data.frame bzw. einem tibble. Es gibt allerdings noch andere Formate, in denen Daten gespeichert werden.

2.1 Vektor

Ein Vektor ist eine eindimensionale Datenstruktur, die nur einen Dateitypen enthalten darf. Also entweder Numeric, Character oder Logical. Einen Vektor kann man mit `c()` erstellen. Aber auch die einzelnen Spalten in unserem Datensatz sind Vektoren. Der Zugriff auf Vektoren erfolgt mit eckigen Klammern. Man kann ebenfalls mit `rep()` und `seq()` einen Vektor erzeugen. `rep()` wiederholt eine Zeichenfrequenz gewünscht oft und `seq()` verhält sich wie der Doppelpunkt mit dem Unterschied, dass man die Abstände zwischen den Zahlen anpassen kann. Wenn man aus einem Data.frame eine Spalte mit dem $-Operator auswählt, ist auch dieser ein Vektor.

```
vektor <- c(1, 2, 3, 4)
bigFive$age
rep(c(1, 2), 2)
seq(from = 1, to = 4, by = .5)
1:4
```

2.2 Matrix

Eine Matrix bietet gegenüber dem Vektor den Vorteil der Multidimensionalität. Sie kann mehrere Vektoren eines Datentyps enthalten. Eine Matrix kann man entweder mit `matrix()` oder mit `cbind() / rbind()` erstellen. cbind bindet Spalten und rbind bindet Zeilen zusammen. Dabei müssen alle Vektoren dieselbe Länge haben.

```r
matrix <- matrix(c(1:8), ncol = 2)
rbind(c(1:4), c(1:4))
cbind(c("a", "b", "c"), c("a", "b", "c"))
```

2.3 Data.frame / tibble

Wir haben zuvor bereits tibbles als Datenstruktur kennengelernt. Der ein oder andere, der bereits Kontakt zu R hatte, kennt möglicherweise diese Datenstruktur als `data.frame()`. Die Syntax unterscheidet sich grundsätzlich nicht. Man sollte nach Möglichkeit immer tibbles benutzen. Die genauen Unterschiede und Vorteile von tibbles können hier nachgelesen werden.

```r
data.frame(a = 1:3, b = c("a", "b", "c"))

tibble(a = 1:3, b = c("a", "b", "c"))
```

2.4 List

In einer Liste kann man alles verstauen. Dabei hat man verschiedene Listenelemente, auf die man mit doppelten eckigen Klammern zugreifen kann. Innerhalb dieser Elemente kann man Vektoren, Matrizen oder tibbles verstauen.

```r
list <- list(vektor, matrix, bigFive)
```

2.5 Zugriff auf Zeilen und Spalten

Mit eckigen Klammern kann man auf eine oder mehrere Zeilen und Spalten über die jeweilige Nummer zugreifen. Dabei steht die Zahl links vom Komma für die Zeilennummer und rechts für die Spaltennummer. Lässt man eins der beiden Felder frei, werden jeweils alle Zeilen oder Spalten ausgewählt. Wenn man mehrere Zahlen auswählen mochte, verbindet man diese mit einem Doppelpunkt. Die Funktion `c()` erlaubt es verschiedene Zahlen zu kombinieren. Ein Minus vor der Zeilen oder Spaltenzahl entfernt diese. Auf die verschiedenen Listenelemente greift man mit einer doppelten eckigen Klammer zu. Wenn sich innerhalb des Listenelements ein Vektor befindet, kann man dann mit den gewohnten einfachen eckigen Klammern auf die gewünschten Elemente zugreifen.

```r
## Zugriff auf das erste Element des Vektors
vektor[1]
## Zugriff auf die 2. Spalte von allen Beobachtungen
matrix[ ,2]
## Zugriff auf die ersten vier Spalten eines tibbles
bigFive[ ,1:4]
## Zugriff auf alle Spalte außer der vierten der ersten fünf Zeilen
```

```
bigFive[1:5, -4]
## Zugriff auf die erste, vierte und fünfte Zeile aller Spalten
bigFive[c(1, 4, 5), ]
## Zugriff auf das erste Element der Liste und zweite des Vektors
list[[1]][2]
```

2.6 Spaltennummer herausfinden

Wenn man einen großen Datensatz hat, möchte man beim Indizieren des Datensatzes über die eckigen Klammern bestimmt nicht jede einzelne Spalte zählen. Da schafft die Funktion `grep()` Abhilfe. Mit `names()` kann man auf die Spaltennamen zugreifen. Die Spalte gender hat also die Nummer 2.

```
grep("gender", names(bigFive))
```

```
## [1] 2
```

2.7 Umwandlungen von Datenstrukturen

Datenstrukturen können grundsätzlich nur ineinander umgewandelt werden, wenn sämtliche notwendigen Eigenschaften gegeben sind. Ein tibble, welcher in 2 Spalten verschiedene Datentypen beinhaltet, kann zum Beispiel **nicht** zu einer Matrix konvertiert werden.

```
as.vector()
as.matrix()
as.data.frame()
as.tibble()
as.list()
```

3 Umgang mit fehlenden Werten

Fehlende Werte bzw. NAs können zu Fehlermeldungen führen. Viele Funktionen haben das logische Argument `na.rm`, welches man zur Berechnung schlichtweg auf `TRUE` setzen muss. Wenn man sämtliche NAs entfernen möchte, kann man dies mit `na.omit()` machen. Allerdings verwirft R dann die gesamte Zeile, in der ein NA vorkommt (je nach Vollständigkeit des Datensatzes, lässt dieser Befehl den Umfang extrem schrumpfen).

```
df_na <- tibble(
  Hallo = c(1:5, NA),
  Welt = c(NA, 1:5)
)
```

```r
mean(df_na$Hallo, na.rm = T)
na.omit(df_na)
```

4 suppressPackageStartupMessage

Da es nicht selten vorkommt, dass beim Laden eines Packages mehrere Hinweise in der Konsole angezeigt werden, kann man mithilfe von `suppressPackageStartupMessage()` diese unterdrücken. Dies ist optional und dient nur einem aufgeräumten Output.

```r
suppressPackageStartupMessages(library(tidyverse))
```

5 Publikationsreife Tabellen mit LaTeX

Wenn man beispielsweise tibbles oder data.frames in publikationsreife Tabellen in LaTeX Code umwandeln möchte, kann man das einfach mit dem Package `xtable` umsetzen.

```r
library(xtable)
xtable(tibble(a = 1:3, b = c("a", "b", "c")))
```

```
## % latex table generated in R 3.4.2 by xtable 1.8-2 package
## % Wed Jan 03 00:13:33 2018
## \begin{table}[ht]
## \centering
## \begin{tabular}{rrl}
##   \hline
##  & a & b \\
##   \hline
## 1 &   1 & a \\
##   2 &   2 & b \\
##   3 &   3 & c \\
##    \hline
## \end{tabular}
## \end{table}
```

6 Parallel Processing

Eine der wenigen Schwachstellen von R ist die Performance. Eine Möglichkeit die Geschwindigkeit von Berechnungsschritten zu beschleunigen, ist **Parallel Processing**. Damit ist die gleichzeitige Benutzung mehrerer zur Verfügung stehender CPU-(Prozessor) Kerne gemeint.

6.1 Microsoft R Open (MRO)

MRO ist eine von Microsoft entwickelte kostenlose Programmiersprache, die auf R basiert. Der große Vorteil ist die Möglichkeit, durch die automatische Nutzung der Intel Math Kernel Library in ausgewählten Szenarien sämtliche zur Verfügung stehende CPU Kerne zu benutzen. Dies erfordert allerdings einen Intel Prozessor. MRO ist mit sämtlichen Packages aus R vollständig kompatibel und kann hier heruntergeladen werden.

6.2 doParallel

Eine weitere Möglichkeit mehrere Kerne zu benutzen, ist auf Windows das `doParallel` Package. In diesem Fall werden 4 Kerne benutzt.

```
library(doParallel)

cl <- makeCluster(4)
registerDoParallel(cl)
```

7 Functional Programming

Bei komplexeren Aufgaben, stößt man relativ schnell auf Iterationsprobleme. Mal angenommen man möchte einen Graphen, der eine Normalverteilung visualisiert - einmal für 10, 20 und 30 Personen. Die einfachste, aber auch am fehleranfälligste Methode zum Lösen dieser Aufgabe wäre Copy & Paste. Man könnte den Befehl, der den Graphen erstellt drei mal kopieren und lediglich den einen Parameter, die Personenanzahl, verändern. Das mag bei drei durchgängen (Iterationen) noch einfach möglich sein. Doch manchmal kann es vorkommen, dass man über 20 unterschiedliche Szenarien durchiterieren möchte. Es gibt verschiedene Werkzeuge dies zu erreichen. Diese fallen alle unter **Functional Programming**.

7.1 Funktionen erstellen

Manchmal bietet es sich an, eigene Funktionen zu schreiben. Dabei schreibt man das Argument, was der Funktion weitergegeben werden soll, in die Klammern von `function()`. Erst einmal gespeichert, kann man die Funktion beliebig oft anwenden. Sie verhält sich dabei genau wie sämtliche anderen Funktionen in R, wie z.B. `mean(x)`. Wenn man unserer Funktion eine Zahl gibt, wird dieselbe Zahl hinzu addiert.

```
fun <- function(x){
  x + x
}
fun(1:5)
```

```
## [1]  2  4  6  8 10
```

7.2 Mapping mit purrr

Mit `purrr` werden eine Reihe an Funktionen geliefert, die es einem Erlauben, Funktionen wiederholt anzuwenden. Dabei seien hier 3 verschiedene Varianten der `map()` Funktion gezeigt. Die normale `map()` Funktion gibt eine Liste zurück. In unserem Beispiel ist jedes Element ein Ergebnis unserer Funktion mit den Zahlen von 1 bis 5. `map_int()` gibt einen Vektor zurück. Wenn die Ergebnisse der angewendeten Funktion jeweils data.frames sind, bindet `map_df()` diese zu einem tibble zusammen.

```
map(1:5, fun)
```

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

```
map_int(1:5, fun)
```

```
## [1]  2  4  6  8 10
```

```
map_df(1:5, as.data.frame(fun))
```

```
##   value
## 1     2
## 2     4
## 3     6
## 4     8
## 5    10
```

7.3 for-Schleife

In for-Schleifen geht man eine bestimmte Anzahl an Iterationen (Wiederholungen) der Reihe nach durch und führt für jedes `i` das aus, was in der geschweiften Klammer steht. In diesem

Beispiel addiert sich für jedes `i` von 1 bis 5 das selbe `i` hinzu. Sie macht also exakt dasselbe wie `map()` mit unserer selbst geschriebenen Funktion.

```
for (i in 1:5) {
  print(i + i)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

7.4 if. . . else Verzweigung

Manchmal möchte man bestimmte Berechnungen nur unter den gewünschten Bedingungen ausführen. Dafür hat man in R zwei Möglichiten. Der Hauptunterschied ist der, dass die `ifelse()` Funktion in bitcode kompiliert und deswegen bei vektoriellen Berechnungen teils erheblich schneller ist als die übliche if-Verzweigung in R. In der Klammer der if-Verzweigung wird geprüft, ob die Bedingung wahr ist. Wenn der Wert wahr ist, wird der Befehl innerhalb der geschweiften Klammern von if ausgeführt - wenn dieser falsch ist, der Befehl innerhalb von else.

```
if (4 < 5) {
  "wahr"
} else {
  "falsch"
}
```

```
## [1] "wahr"
```

```
ifelse(6 < 5, "wahr", "falsch")
```

```
## [1] "falsch"
```

7.5 Switch-Statement

Switch-Statements sind grundsätzlich den if. . . else Verzweigungen sehr ähnlich. Es bietet sich aber aus Performancegründen sowie aus Gründen der Übersichtlichkeit an, auf Switch-Statements immer dann zurückzugreifen, wenn man genau weiß, wie viele Möglichkeiten man abfragen möchte.

```
frucht <- "Apfel"
switch(frucht,
       "Apfel" = print("Lecker"),
```

```
       "Banane" = print("Bah")
)
```

```
## [1] "Lecker"
```

8 Transformation einer Ergebnismatrix

Häufig ist man in der Situation Daten z.B. durch SoSciSurvey erhoben zu haben, allerdings eine binäre Matrix zu benötigen, in der man ledglich die Informationen enthalten hat, ob ein Item richtig oder falsch beantwortet wurde. Mit der Funktion `transform_binary()` des `rBasics` Packages, kann man anhand eines Antwortvektors, der die richtigen Antworten enthält, die Matrix transformieren. Falls sowohl Zahlen, als auch Strings als richtige Antworten in Frage kommen, kann man der Funktion ebenfalls einen einzeiligen tibble übergeben.

```
library(rBasics)

## Identische Datentypen
soscisurvey <- tibble(
  I1 = c(3, 2, 3, 1),
  I2 = c(2, 2, 2, 1),
  I3 = c(4, 1, 4, 4)
)

antwortvec <- c(3, 2, 4)

transform_binary(soscisurvey, antwortvec)
```

```
## # A tibble: 4 x 3
##      I1    I2    I3
##   <dbl> <dbl> <dbl>
## 1     1     1     1
## 2     0     1     0
## 3     1     1     1
## 4     0     0     1
```

```
## Verschiedene Datentypen
soscisurvey <- tibble(
  I1 = c(3, 2, 3, 1),
  I2 = c("Apfel", "Banane", "Erdbeere", "Apfel"),
  I3 = c(4, 1, 4, 4)
)

antwortvec <- tibble(3, "Apfel", 4)
```

```
transform_binary(soscisurvey, antwortvec)
```

```
## # A tibble: 4 x 3
##      I1    I2    I3
##   <dbl> <dbl> <dbl>
## 1     1     1     1
## 2     0     0     0
## 3     1     0     1
## 4     0     1     1
```