5 - Style und Conditionals

author: Benedict Witzenberger date: 17. April 2019 autosize: true

Wie sieht guter Stil für Programmierer aus?

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> Phil Karlton, Netscape

In R gibt es gewissen Konventionen, die helfen sollen, dass auch fremde Leute unseren Code verstehen.

Denn es gibt nichts schlimmeres als Code, den eure Nachfolger komplett umschreiben müssen, weil sie mit ihm nicht arbeiten können.

Benennung von Dateien

Einfacher Tipp: Schreibt, was die Datei tut.

Gut:

```
load_data.R
make_graphic.R
```

Schlecht:

```
Projekt.R
test.R
final.R
```

Benennung von Variablen

Variablen sollten immer klein geschrieben werden und mehrere Wörter durch einen Unterstrich _ getrennt werden. Nutzt niemals ein Leerzeichen, das macht nur Ärger!

Gut:

```
tag_eins
daten_alt
```

Schlecht:

```
tag1
datenAlt
`daten eingelesen`
```

Versucht, die Namen von bestehenden Objekten und Funktionen nicht als variablenname zu nehmen. Das geht, aber irritiert enorm.

```
True <- FALSE
c <- 10
mean <- function(x) sum(x)
```

Exkurs: Verschiedene Sprachen. verschiedene Akzente

Ich habe für R mal die Benennungsform “snake_case” gelesen

R orientiert sich bei der Benennung am ehesten an der Sprache `C`: `user_name, delete_user, create_bank_account`

Andere Sprachen, zum Beispiel Javascript nutzen CamelCase: `userName, deleteUser, createBankAccount`

Es gibt aber auch PascalCase: `UserName, DeleteUser, CreateBankAccount`

Oder die COBOL-Variante mit Bindestrich: `USER-NAME, DELETE-USER, CREATE-BANK-ACCOUNT`

Fazit: Egal, was ihr nutzt: nutzt es durchgehend. Und überlegt, ob/wie andere Leute euren Code möglichst gut verstehen können.

Syntax: Lasst Platz

Vor `+ - / * = > < <-` etc. sollte immer ein Leerzeichen gelassen werden. Das macht den Code viel besser lesbar.

Gut:

```
average <- mean(feet / 12 + inches, na.rm = TRUE)
```

Schlecht:

```
average<-mean(feet/12+inches,na.rm=TRUE)
```

Einrückungen

Bei längeren Funktionsaufrufen, Dataframes oder Listen könnt ihr auch Einrückungen verwenden

```
d %>%
mutate(
  total = a + b + c,
  mean  = (a + b + c) / n
)
```

Normal sind bei R zwei Leerzeichen Einrückung. Außer bei langen Funktionsaufrufen.

RStudio macht das automatisch mit Strg/Cmd + I

Klammern

Klammern sollten ans Ende einer Zeile und nicht alleine stehen.

```
if (y == 0) {
  log(x)
} else {
  y ^ x
}
```

Schlecht:

```
if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

Kurze Statements dürfen in einer Zeile bleiben: `if (y == 0) { log(x) }`

Nutzt Kommentare

Kommentare helfen, euren Code verständlicher zu machen. Für Andere, aber auch für euch. Aber erklärt warum ihr etwas tut, nicht, was ihr tut. Das sieht man ja im Code.

Kommentare macht ihr mit #

Mehrzeilige Kommentare mit Strg/Cmd + Shift + C in

Nutzt für die Optik gerne auch - oder =

```
# Load data ---------------------------

# Delete remaining Files ==============
```

Übung: Macht den Code sauber

Findet die Fehler im Stil in den folgenden Codezeilen und korrigiert sie:

```
# setting x
x=c(1,3,6,9)

# calculating mean
mean = sum(x)/length(x)
```

Konsistente Daten

Nachdem wir die Daten in den Dataframes korrekt bereitgestellt haben, ist die Frage: Wie überprüfen wir, dass die Daten auch sauber sind?

Fehlende Werte

`NA`s sind immer eine Warnung: Aber was bedeutet “not available”?

In Umfragen: keine Angabe, Antwort verweigert, unbekannt?

In Rechnungen: unerlaubte Rechenoperationen, die nicht 0 ergeben dürfen

Lösungsstrategien:

- Die meisten Funktionen in R haben ein Argument gegen NAs: `na.rm = TRUE`

- `complete.cases()` gibt nur vollständige Zeilen eines Dataframes als logischen Vektor zurück.
- `na.omit()` schmeisst unvollständige Zeilen aus einem Dataframe

Spezialwerte

Kommen vor allem bei mathematischen Funktionen vor. Genz selten wollen wir die wirklich haben

```
is.finite(c(1, 2, Inf, NaN, NaN))
```

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

Ausreißer

Ist das, was wir als Journalisten suchen. Deswegen wollen wir sie nicht entfernen, sondern finden.

Für normal-verteilte Daten helfen Box-and-Whisker-Plots, die nach der 1,5fachen IQR ab dem drittel/ersten Quartil die Outlier definieren. Das ist aber Definitionssache und kann verändert werden.

```
x <- c(1:10, 20, 30)
boxplot.stats(x)$out
```

```
[1] 20 30
```

```
boxplot(x)
```

Inkonsistenzen

Ein paar Beispiele:

- Menschen können kein negatives Alter haben
- Erwachsene wiegen normalerweise mehr als 20 Kilogramm
- Autos fahren schneller als 18 km/h (vielleicht 180?)
- Ehepaare können nicht länger verheiratet sein, als sie alt sind (bzw. erwachsen sind)

Conditionals

Ein gutes Skript muss immer wieder Entscheidungen treffen, nach Regeln, die wir ihm vorgegeben haben.

Dafür gibt es - in vielen Programmiersprachen ähnliche - Begriffe:

if

if . . . else

switch()

Alle diese Begriffe sind “reserved” words - dürfen also nicht als Variablennamen benutzt werden.

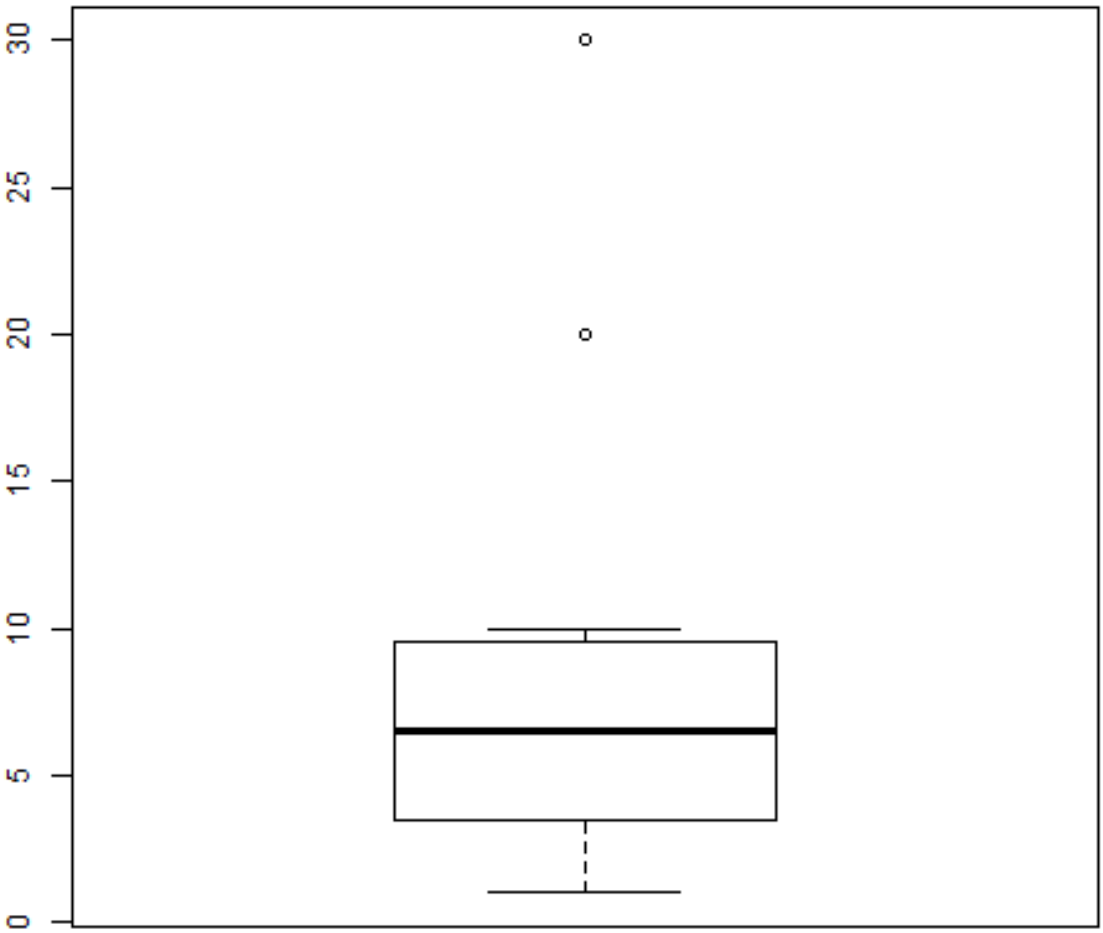
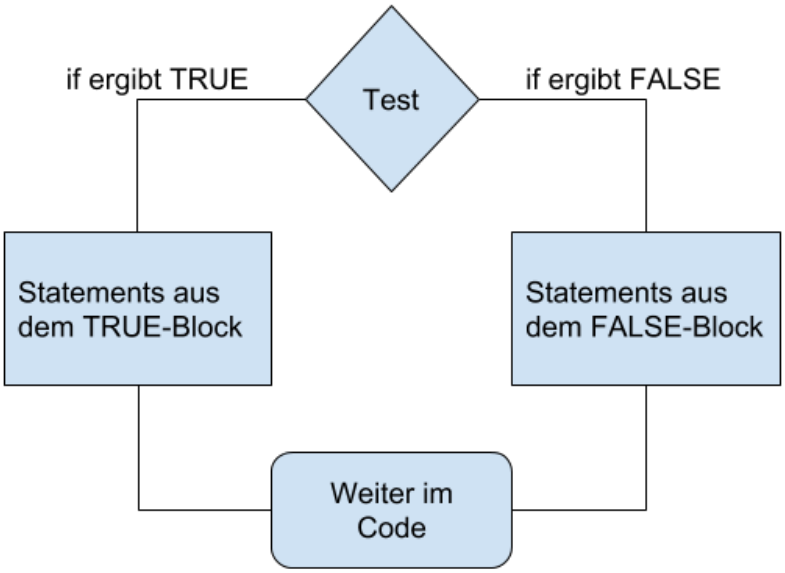


Figure 1: plot of chunk unnamed-chunk-2

if-Statements



```
if (condition) {
  Code für TRUE
}
```

Übung: if-Statements

Schreibt einen Code, der euch ausgibt "Heute ist Mittwoch", wenn das heutige Datum Mittwoch ist. Dafür könnt ihr die Funktion `weekdays()` benutzen. Sie erwartet ein POSIXct als Input.

Lösung: if-Statements

```
if (weekdays(Sys.time()) == "Mittwoch") {
  print("Heute ist Mittwoch")
}
```

Die Locale

Achtet bei Tests mit ausgeschriebenen Wochentagen auf die Locale eures Rechners. Sie bestimmt, ob ein Datum als "Wednesday" oder "Mittwoch" ausgegeben wird.

Ihr bekommt einen Überblick mit `Sys.getlocale(category = "LC_ALL")`

Steht da nichts mit German, dann könnt ihr die einzelnen Teile umstellen: `Sys.setlocale("LC_TIME", "de_DE.UTF-8")`

if. . . else-Statements

If-Statements sind gut, wenn wir eine einzige Abfrage haben, die etwas auslösen soll. Aber was ist, wenn wir etwas anderes auslösen wollen, wenn if FALSE ist?

Dafür gibt es if. . . else-Statements

```
if (condition) {
    Code für TRUE
} else {
    Code für FALSE
}
```

else if

ist nichts wirklich neues, sondern nur eine Kombination aus den beiden vorherigen Tests:

Mit "else if" können wir Varianten der Entscheidung definieren.

```
if (condition1) {
    Code für TRUE
} else if (condition2) {
    Code für TRUE
} else if (condition3) {
    Code für TRUE
} else {
    Code für FALSE
}
```

Das ist sehr hilfreich, aber wird auch schnell unübersichtlich.

switch

Hilft, wenn wir ganz viele if-Statements für verschiedene Funktionen kombinieren möchten. Macht den Ausdruck kürzer.

```r
rechenart <- "addition"
x <- c(1, 2, 3, 4, 5)

switch(rechenart,
  addition = sum(x),
  durchschnitt = mean(x),
  median = median(x))
```

```
[1] 15
```

Variante: Das Package `dplyr` (lernt ihr im nächsten Blockkurs) hat eine `case_when()`-Funktion, die für verschiedene Ergebnisse verschiedene Outputs liefert. Da könnten wir zum Beispiel auch Tests für verschiedene Zahlenwerte integrieren.

Conditions in if-Statements

- Wir kennen schon die Vergleiche mit & und |. Bei Funktionen nutzen wir aber nur && und ||: Denn die überprüfen schneller, und nur einen einzigen Wert - was wir bei if-Statements normalerweise auch wollen. (Falls wir einen logical Vector testen wollen, könnten wir `all()` oder `any()` verwenden, der bringt den Vector auf eine TRUE- oder FALSE-Aussage)
- Jede Condition muss zu TRUE oder FALSE werden können, sonst gibt es einen Fehler:

```
> if (NA) {}
Error in if (NA) { : missing value where TRUE/FALSE needed
```

- Der Test auf NA geht nicht mit `x == NA`, sondern mit `is.na()`
- Achtet auf den Codestil! Geschweifte Klammern sollten immer mit anderen Worten in einer Zeile stehen.
- Nur bei ganz kurzen Statements dürftet ihr die geschweiften Klammern weglassen:

`x <- if (y > 20) "Too low" else "Too high"` (übersichtlicher ist es aber wohl mit Klammern)

Übungen: Conditional Statements

1. Lest euch in der Hilfe den Unterschied zwischen `if` und `ifelse()` durch. Wie würdet ihr ihn beschreiben?
2. Schreibt ein if... else-Statement, für das ihr in einer Variable die Uhrzeit festlegen könnt. Je nachdem, wie viel Uhr es ist, gibt euch das Programm "Guten Morgen", "Guten Tag", "Guten Abend" oder "Gute Nacht" aus.
3. Schreibt mit `switch()` einen Test, der Addition, Substraktion, Multiplikation und Division für zwei verschiedene Variablen durchführt. Je nachdem, was in einer dritten Variable angegeben wurde.

ifelse()

Macht es möglich, dass wir ein if über Vektoren laufen lassen. Indem das Ergebnis binär wird:

```
ifelse(condition, true, false)
```

```
mein_vector <- c(1, 3, 5, 6, 8, 9)
```

```
# Geht nicht!
> if (mein_vector %% 3 == 0) {
+   print("Durch drei teilbar")
+ }
Warning message:
In if (mein_vector%%3 == 0) { :
  the condition has length > 1 and only the first element will be used
```

```r
ifelse(mein_vector %% 3 == 0, "Durch 3 teilbar", "Nicht durch drei teilbar")
```

```
[1] "Nicht durch drei teilbar" "Durch 3 teilbar"
[3] "Nicht durch drei teilbar" "Durch 3 teilbar"
[5] "Nicht durch drei teilbar" "Durch 3 teilbar"
```