Muddiest Points 2

Muddiest Points – Session 2

Frage 1: Datensatz einlesen - zusätzliche Spalte

Bei mir gibt es beim Einlesen des Datensatzes immer noch eine zusätzliche Spalte, welche nochmal durchnummeriert ist (also wie eine ID, aber ohne Namen). Wie lese ich den Datensatz richtig ein, damit ich diese Spalte nicht immer noch zusätzlich löschen muss?

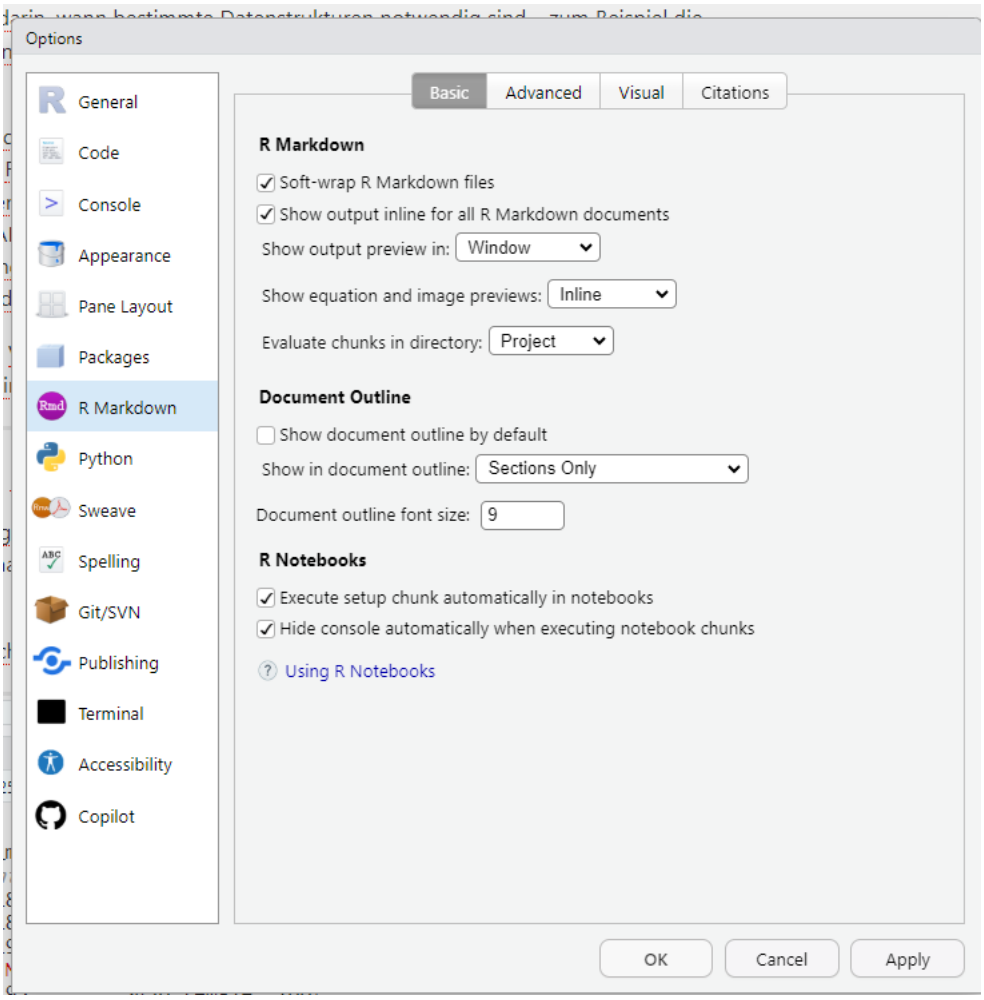
Antwort:
Wahrscheinlich hast du beim **Speichern** der Daten mit `write.csv()` das Argument `row.names` nicht gesetzt. Wenn du `row.names = FALSE` angibst, entsteht diese zusätzliche Spalte nicht. Genauere Erklärung findest du hier. Das Problem liegt also vermutlich nicht am Einlesen der Daten, sondern daran wie du die Daten abgespeichert hast. Wenn dem nicht so ist - bitte komm noch mal auf uns zu!

Frage 2: Relative Pfade

Manchmal sind mir die Pfade, die ich beim Einlesen und Abspeichern einer Datei angeben muss, nicht klar. Die kürzere Version funktioniert oft nicht, deshalb muss ich immer den ganzen Pfad eingeben.

Antwort:
Wenn relative Pfade nicht funktionieren, überprüfe zuerst, ob du dein Projekt korrekt geöffnet hast. Falls das Problem bleibt, kannst du mit `getwd()` dein aktuelles Arbeitsverzeichnis ausgeben lassen. Dieses sollte auf deinen Projektordner zeigen.

Wenn das nicht funktionieren sollte prüfe ob du diese Einstellung unter "Tools - Global Options" vorgenommen hast: Evaluate Chunks in directory: Projekt



Weitere Möglichkeiten:

Wenn du dich zum Beispiel im Ordner `grinschgl2020/code/` befindest, aber eine Datei aus einem übergeordneten Ordner einlesen möchtest (z. B. `grinschgl2020/data/`), kannst du beim Einlesen `..` verwenden. Damit geht R eine Ebene nach oben, und relative Pfade funktionieren wie erwartet.

Beispiel:

```
# Wir befinden uns in: grinschgl2020/code/

# Eine Ebene nach oben gehen (..) und in den data-Ordner wechseln
daten <- read.csv("../data/raw/daten_roh.csv")
```

Frage 3: Verschachtelung und Across

Einige Funktionen wie `across()` werden in anderen Funktionen verschachtelt verwendet und brauchen zusätzliche Argumente. Die Logik der Verschachtelung ist mir noch nicht klar.

Antwort:
Funktionen wie `across()` werden verschachtelt, weil sie nur innerhalb von dplyr-Funktionen wie `mutate()` oder `summarise()` arbeiten. Die äussere Funktion bestimmt, was passieren soll (z. B. Spalten verändern). `across()` legt dann fest, **welche Spalten** betroffen sind und **welche Transformation** angewendet wird. Deshalb braucht `across()` Argumente wie `.cols` und `.fns`. Die äussere Funktion gibt den Rahmen vor, `across()` steuert die konkrete Operation.

Beispiel:

`across()` hat zwei wichtige Argumente:

- **.cols** → *Welche Spalten sollen ausgewählt werden?*
  Im Beispiel: alle Spalten, die auf **"_r"** enden. Across geht wert für wert durch diese Spalten
- **.fns** → *Welche Operation soll auf jeden einzelnen Wert dieser Spalten angewendet werden?*
  Im Beispiel: Für jeden Wert wird **6 − Wert** berechnet (Umkehrkodierung).
- Das `.x` steht für den aktuellen Wert durch den across durchgeht. Manchmal wird auch nur ein Punkt ausgeschrieben da beide Schreibweisen funktionieren.

```
data_reversed <- data_numeric |>
  mutate(
    across(ends_with("_r"), ~ 6 - .x) #~ 6 - . --> Für jede wert der col -6
  )
```

Frage 4: Dataframe oder Vektor

Ich habe manchmal ein Data Frame an eine Funktion übergeben, obwohl diese einen Vektor erwartet, oder umgekehrt. Mir ist noch nicht klar, welche Funktionen Vektoren und welche Data Frames erwarten.

Gibt es irgendeine "Merkhilfe" um zu wissen bei welchen Packages/ Funktionen ich die Variablen auf welche Art auflisten muss? –> Also ob mit "Variable" oder c(Variable)?

Antwort:
Es gibt keine feste Regel, aber die Hilfeseite (`?funktion`) zeigt immer, welcher Datentyp erwartet wird. Grundsätzlich gilt: Funktionen, die Spaltenweise etwas berechnen (z. B. `mean`, `sum`, `skew`), erwarten Vektoren – oft extrahiert man diese mit `$` aus einen Dataframe.
Funktionen, die Daten transformieren (`summarise`, `mutate`, `filter`, `select`), erwarten dagegen Data Frames, da sie auf mehreren Variablen gleichzeitig arbeiten.

Das **c** bei Vektoren steht für *concatenate* und wird verwendet, wenn wir mit Vektoren arbeiten. Wenn nur ein Element im Vektor vorhanden ist, brauchen wir kein `c()` (Skalar). Sobald mehrere Elemente enthalten sind, benötigen wir `c()`.

Frage 5: Datenstrukturen

Ich bin mir noch unsicher darin, wann bestimmte Datenstrukturen notwendig sind – zum Beispiel die Umwandlung von Variablen in Faktoren.

Antwort:
Viele Transformationen sind notwendig, weil statistische Funktionen bestimmte Datentypen voraussetzen. Die ANOVA etwa benötigt Faktoren, wenn Gruppen verglichen werden sollen, damit R die Gruppen-Variable als kategorial erkennt. Solche Transformationen sind also Teil der korrekten Vorbereitung der Daten. Wenn du z.B. eine ANOVA durchführst und vergisst die Gruppen-Variable als Faktor umzuwandeln, bekommst du eine Warnung. Durch diese erkennst du dann, dass ein Faktor benötigt wird - es ist also nicht allzu schlimm, wenn du die Umwandlung zunächst verpasst hast.

Es ist eine gute Faustregel, Variablen in Faktoren umzuwandeln, wenn sie klar kategoriale Gruppen darstellen – zum Beispiel eine Gruppenzugehörigkeit wie “above”, “below” oder “control”.

Frage 6: Allgemeine Fehler

Ich wäre froh, wenn wir allgemeine Fehler beim Codieren durchgehen könnten – also häufige Syntaxfehler und worauf man achten sollte.

Antwort:
(Dieser Punkt ist als Wunsch notiert.)

Frage 7: Einlesen speichern und Organisieren

Daten korrekt einlesen, abspeichern, und im Projektkontext richtig organisieren. / Abspeichern der Daten, gute Ordnerstruktur? Das ist nur ein "kleiner" Punkt, aber ich habe immer ein bisschen ein Durcheinander, wie ich Skripts und Projekte am besten abspeichere, damit auch immer alles funktioniert bei der Analyse. Oft habe ich bspw. zu lange Dateipfade

Antwort:
Grundsätzlich gilt: Alle **Rohdaten** werden im **raw/**-Ordner gespeichert (und **niemals überschrieben**). Diese Daten werden dann im *processing*-Skript bereinigt und verarbeitet. Die verarbeiteten Datensaetze werden anschliessend mit `write.csv()` im **processed/**-Ordner gespeichert.

Im *analysis*-Skript werden **keine Bereinigungen** mehr vorgenommen, sondern nur noch die eigentlichen Analyseschritte durchgefuehrt (mit Ausnahme kleiner Anpassungen wie das Setzen von Faktoren, da diese Aenderungen nicht gespeichert werden).

Diese Struktur orientiert sich am **PsychDS-Standard**.

Hausübungen und Hands on: Da wir mit vielen verschiedenen Datensätzen und Skripten arbeiten, ist es sinnvoll, eine klare Ordnerstruktur zu verwenden, zum Beispiel einen Ordner für Skripte und einen für Daten. Man kann sich dabei an der PsychDS-Struktur orientieren und diese übernehmen, auch wenn wir nicht mit den *dat_full*-Daten arbeiten.

Zu den Pfaden: Achte darauf, relative Pfade zu verwenden. Lange Pfade sind grundsätzlich kein Problem, solange sie relativ, nachvollziehbar und konsistent aufgebaut sind.

Frage 8: Wide - Long

Wide to long: Weshalb ist das notwendig, weshalb kann man nicht beim einen Format bleiben?

Antwort:
Viele Funktionen setzen ein bestimmtes Datenformat voraus. Repeated-Measures-Funktionen sind das beste Beispiel – sie benötigen Long-Format, weil jede Zeile eine Beobachtung darstellt. Das wird in den nächsten Wochen noch verständlicher - wenn wir dann Berechnungen mit dem Wide als auch Long Format durchführen.

Frage 9: Class, Attributes, und Table

Was ist der Unterschied zwischen `class()`, `attributes()` und `table()`?

Antwort:

- **class()** zeigt den Typ eines Objekts

```
  class(penguins)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
  class(penguins$species)
```

```
[1] "factor"
```

- **table()** zeigt Häufigkeiten

```
table(penguins$island)
```

```
   Biscoe     Dream Torgersen
      168       124        52
```

- **attributes()** zeigt die Attribute eines Objekts
  - bei Data Frames: Variablennamen, rownames, Datentypen
  - bei Faktoren: Levels und Klasse

```
attributes(penguins)
```

```
$class
[1] "tbl_df"     "tbl"        "data.frame"

$row.names
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
```

```
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
[163] 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
[181] 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198
[199] 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216
[217] 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234
[235] 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252
[253] 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270
[271] 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288
[289] 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306
[307] 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324
[325] 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342
[343] 343 344

$names
[1] "species"           "island"            "bill_length_mm"
[4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
[7] "sex"               "year"
```

```
attributes(penguins$species)
```

```
$levels
[1] "Adelie"    "Chinstrap" "Gentoo"

$class
[1] "factor"
```

Frage 10: Nachhaltige Anpassungen Datensatz

Wie kann ich nachhaltige Anpassungen an Variablen im Datensatz vornehmen?

Antwort:

Um Änderungen im Datensatz festzuhalten, muss eine Zuweisung mit `<-` vorgenommen werden. Wenn ein Befehl ohne Zuweisung ausgeführt wird, wird das Resultat nicht im Environment abgespeichert. Wenn ein Objekt im Environment ist, heisst das nicht, dass dieses automatisch als z. B. CSV gespeichert wird. Dafür sollte es mit einer passenden Write-Funktion gespeichert werden. Im Processing-Skript führt ihr alle Aufbereitungsschritte (z. B. Variablen

umbenennen) durch und speichert am Ende den bereinigten Datensatz mit `write.csv()` oder einer ähnlichen Funktion ab, damit alle Änderungen nachhaltig gesichert sind.

Beispiel –> Keine Veränderung am Datensatzt weil keine Zuweisung

```
penguins |>  mutate(mean_bill_length = mean(bill_length_mm, na.rm = TRUE))
```

Beispiel mit Zuweisung: Gleicher Datensatz wird verändert

```
penguins <- penguins |>
  mutate(mean_bill_length = mean(bill_length_mm, na.rm = TRUE))
```

Frage 11: Group_by

Für mich persönlich ist die group_by() Funktion irgendwie nicht ganz klar. Die Vorstellung, wie die Daten sortiert werden, ist nicht so intuitiv.

Antwort:

Für viele ist die Funktion anfangs nicht intuitiv. Die Vorstellung, wie die Daten intern in Gruppen „aufgeteilt“ werden, ohne dass sie tatsächlich sortiert oder neu angeordnet werden, fühlt sich ungewohnt an. Das ist völlig normal – die Logik hinter `group_by()` entwickelt sich meist erst, wenn man sieht, wie sie gemeinsam mit Funktionen wie `summarise()` oder `mutate()` wirkt. Kategoriale Daten werden in ihre kategorien augeteilt, kontinuierliche Daten werden in ihre einzigartigen werte gruppiert.

Beispiel Kontinuierliche Variable: Gruppe für jedes Gewicht

```
penguins_summary <- penguins |>
  group_by(body_mass_g) |>
  summarize(mean_bill_length = mean(bill_length_mm, na.rm = TRUE))

penguins_summary[1:10, 1:2]
```

```
# A tibble: 10 x 2
   body_mass_g mean_bill_length
         <int>            <dbl>
 1        2700             46.9
 2        2850             36.4
 3        2900             37.4
```

```
 4        2925             37.9
 5        2975             37.5
 6        3000             37.2
 7        3050             35.6
 8        3075             37.7
 9        3100             36
10        3150             36.6
```

Beispiel kategoriale Variable

```
penguins_summary_2 <- penguins |>
  group_by(island) |>
  summarize(mean_bill_length = mean(bill_length_mm, na.rm = TRUE))

penguins_summary_2
```

```
# A tibble: 3 x 2
  island    mean_bill_length
  <fct>                <dbl>
1 Biscoe                45.3
2 Dream                 44.2
3 Torgersen             39.0
```

Frage 12: Read_delim, read_csv, write.csv

Ich habe auch ein bisschen ein Durcheinander, wann die Daten am besten wie abgespeichert/eingelesen werden. Also mit read_delim(), row.names = FALSE, csv2 vs csv usw.

- **Einlesen:**
  Kurz zusammengefasst: Am einfachsten ist es, zunächst die Point-and-Click-Oberfläche in RStudio zu nutzen und die Optionen so einzustellen, dass die Daten sinnvoll eingelesen werden, und anschliessend den generierten Code zu übernehmen. `read_csv()` ist im Prinzip das Gleiche wie `read_delim()`, nur dass es standardmässig ein Komma als Trennzeichen verwendet (CSV bedeutet *comma separated values*). Bei `read_delim()` muss das Trennzeichen hingegen explizit angegeben werden, zum Beispiel ein Semikolon.
- **Speichern:**
  Zum Speichern nutzt ihr am besten `write.csv()`. Von `write.csv2()` sollte man eher die Finger lassen, da es historische Spezialfälle abdeckt und meist eher zu Verwirrung führt.