2. Fortgeschrittenes Arbeiten mit Excel

Professioneller Import und Export mit readxl und openxlsx2

Dr. Paul Schmidt

Packages laden

```
for (pkg in c("glue", "openxlsx2", "readxl", "tidyverse")) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg, dependencies = TRUE)
  }
  library(pkg, character.only = TRUE)
}
```

```
Lade nötiges Paket: glue
```

```
Lade nötiges Paket: openxlsx2
```

```
Lade nötiges Paket: readxl
```

```

Attache Paket: 'readxl'
```

```
Das folgende Objekt ist maskiert 'package:openxlsx2':

    read_xlsx
```

```
Lade nötiges Paket: tidyverse
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✓ dplyr     1.1.4     ✓ readr     2.1.5
✓ forcats   1.0.0     ✓ stringr   1.5.2
✓ ggplot2   4.0.2     ✓ tibble    3.3.0
✓ lubridate 1.9.4     ✓ tidyr     1.3.1
✓ purrr     1.1.0
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter()     masks stats::filter()
✖ dplyr::lag()        masks stats::lag()
✖ readxl::read_xlsx() masks openxlsx2::read_xlsx()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts
to become errors
```

```
# Output-Verzeichnis erstellen falls nicht vorhanden
if (!dir.exists("output")) {
  dir.create("output")
}
```

Beispiel-Excel-Datei vorbereiten

Für die Import-Beispiele verwenden wir eine Excel-Datei, die mit dem `openxlsx2`-Paket mitgeliefert wird. Diese enthält zwei Sheets und verschiedene Datentypen, was sie ideal für unsere Demonstrationen macht. Wir kopieren sie in unser Output-Verzeichnis, damit wir im gesamten Kapitel damit arbeiten können:

```
# Beispiel-Datei aus openxlsx2-Paket lokalisieren
example_file <- system.file("extdata", "openxlsx2_example.xlsx", package =
"openxlsx2")

# In output-Ordner kopieren
file.copy(
  from = example_file,
  to = "output/example.xlsx",
  overwrite = TRUE
)
```

```
[1] TRUE
```

1. Import: Beyond the Basics

In früheren Kapiteln haben wir `readxl::read_excel()` kennengelernt, um einzelne Excel-Dateien einzulesen. In der Praxis begegnen uns jedoch oft komplexere Situationen: Dateien mit mehreren Tabellenblättern, unordentliche Strukturen mit Kopfzeilen und Fußnoten, oder spezielle Zellbereiche, die wir gezielt auslesen möchten. Hier erweitern wir unsere Kenntnisse um diese häufigen Spezialfälle.

1.1 Multiple Sheets importieren

Excel-Dateien enthalten häufig mehrere Tabellenblätter, die thematisch zusammengehören – beispielsweise verschiedene Messzeitpunkte eines Experiments oder unterschiedliche Datensätze einer Studie. Anstatt jedes Sheet einzeln und manuell zu laden, können wir mit `excel_sheets()` alle vorhandenen Sheet-Namen auslesen und diese dann systematisch in einer Schleife importieren. Das Ergebnis speichern wir in einer named list, sodass wir über die Sheet-Namen direkt auf die jeweiligen Datensätze zugreifen können.

```
# Unsere Beispiel-Datei
file_path <- "output/example.xlsx"

# Alle Sheet-Namen auflisten
sheet_names <- excel_sheets(file_path)
sheet_names
```

```
[1] "Sheet1" "Sheet2"
```

```
# Alle Sheets in eine named list importieren
all_data <- map(sheet_names, \(sheet) read_excel(file_path, sheet = sheet))
```

```
New names:
New names:
• `` -> `...3`
```

```
names(all_data) <- sheet_names

# Zugriff auf einzelnes Sheet
all_data$Sheet1
```

```
# A tibble: 10 × 9
   Var1   Var2 ...3    Var3 Var4  Var5                Var6          Var7
   <lgl> <dbl> <lgl>  <dbl> <chr> <dttm>              <chr>        <dbl>
 1 TRUE      1 NA       1    a     2023-05-29 00:00:00 3209324 This  NA
 2 TRUE     NA NA      NA    b     2023-05-23 00:00:00 <NA>           0
 3 TRUE      2 NA       1.34 c     2023-02-01 00:00:00 <NA>          NA
 4 FALSE     2 NA      NA    <NA>  NA                  <NA>           2
 5 FALSE     3 NA       1.56 e     NA                  <NA>          NA
```

```
 6 FALSE     1 NA      1.7  f     2023-03-02 00:00:00 <NA>           2.7
 7 NA       NA NA     NA    <NA>  NA                  <NA>          NA
 8 FALSE     2 NA     23    h     2023-12-24 00:00:00 <NA>          25
 9 FALSE     3 NA     67.3  i     2023-12-25 00:00:00 <NA>           3
10 NA        1 NA    123    <NA>  2023-07-31 00:00:00 <NA>         122
# i 1 more variable: Var8 <dttm>
```

Alternativ können wir auch mit einer klassischen for-Schleife arbeiten, falls wir `purrr` nicht verwenden möchten oder die Logik expliziter gestalten wollen:

```
all_data <- list()
for (sheet in sheet_names) {
  all_data[[sheet]] <- read_excel(file_path, sheet = sheet)
}
```

```
New names:
New names:
• `` -> `...3`
```

1.2 Präzises Lesen: Ranges & Skip

In der realen Arbeitswelt sind Excel-Dateien selten so aufgeräumt wie in Lehrbüchern. Oft finden wir Beschreibungstexte oberhalb der eigentlichen Daten, Fußnoten unterhalb, leere Zeilen als Trenner, oder die eigentlichen Daten beginnen erst in Zeile 10 und Spalte C. Für solche Situationen bietet `readxl` mehrere nützliche Optionen, mit denen wir präzise steuern können, welche Teile der Datei wir einlesen möchten.

Mit der `range`-Option können wir einen exakten Zellbereich in Excel-Notation angeben (z.B. `"B5:G20"`), um nur diesen Bereich einzulesen. Die `skip`-Option überspringt eine bestimmte Anzahl von Zeilen am Anfang der Datei – praktisch, wenn die Daten erst nach mehreren Kopfzeilen beginnen. Falls die Spaltennamen selbst chaotisch oder unbrauchbar sind, können wir mit `col_names = FALSE` das automatische Einlesen der Header deaktivieren. Und schließlich können wir mit `na` festlegen, welche Zeichenketten als fehlende Werte interpretiert werden sollen – denn nicht jeder verwendet “NA” für Missing Values.

```
# Nur einen bestimmten Zellbereich einlesen (Beispiel mit unserer Datei)
df <- read_excel("output/example.xlsx", range = "B3:G10")
```

```
New names:
• `1` -> `1...2`
• `` -> `...3`
• `1` -> `1...4`
```

```
df
```

```
# A tibble: 7 × 6
  `TRUE` `1...2` ...3  `1...4` a     `45075`
  <lgl>    <dbl> <lgl>   <dbl> <chr> <dttm>
1 TRUE        NA NA       NA   b     2023-05-23 00:00:00
2 TRUE         2 NA        1.34 c     2023-02-01 00:00:00
3 FALSE        2 NA       NA    <NA>  NA
4 FALSE        3 NA        1.56 e     NA
5 FALSE        1 NA        1.7  f     2023-03-02 00:00:00
6 NA          NA NA       NA    <NA>  NA
7 FALSE        2 NA       23    h     2023-12-24 00:00:00
```

```
# Erste 2 Zeilen überspringen
df <- read_excel("output/example.xlsx", skip = 2)
```

```
New names:
• `1` -> `1...2`
• `` -> `...3`
• `1` -> `1...4`
• `` -> `...8`
```

```
df
```

```
# A tibble: 9 × 9
  `TRUE` `1...2` ...3  `1...4` a     `45075`             `3209324 This`  ...8
  <lgl>    <dbl> <lgl>   <dbl> <chr> <dttm>              <lgl>          <dbl>
1 TRUE        NA NA       NA   b     2023-05-23 00:00:00 NA               0
2 TRUE         2 NA        1.34 c     2023-02-01 00:00:00 NA              NA
3 FALSE        2 NA       NA    <NA>  NA                  NA               2
4 FALSE        3 NA        1.56 e     NA                  NA              NA
5 FALSE        1 NA        1.7  f     2023-03-02 00:00:00 NA               2.7
6 NA          NA NA       NA    <NA>  NA                  NA              NA
7 FALSE        2 NA       23    h     2023-12-24 00:00:00 NA              25
8 FALSE        3 NA       67.3  i     2023-12-25 00:00:00 NA               3
9 NA           1 NA      123    <NA>  2023-07-31 00:00:00 NA             122
# i 1 more variable: `6.0590277777777778E-2` <dttm>
```

```
# Keine automatischen Spaltennamen (wenn Header chaotisch ist)
df <- read_excel("output/example.xlsx", col_names = FALSE)
```

```
New names:
• `` -> `...1`
• `` -> `...2`
• `` -> `...3`
• `` -> `...4`
• `` -> `...5`
• `` -> `...6`
• `` -> `...7`
• `` -> `...8`
• `` -> `...9`
```

```
df
```

```
# A tibble: 11 × 9
   ...1  ...2  ...3  ...4  ...5  ...6  ...7         ...8  ...9
   <chr> <chr> <lgl> <chr> <chr> <chr> <chr>        <chr> <chr>
 1 Var1  Var2  NA    Var3  Var4  Var5  Var6         Var7  Var8
 2 TRUE  1     NA    1     a     45075 3209324 This <NA>  6.0590277777777778E-2
 3 TRUE  <NA>  NA    <NA>  b     45069 <NA>         0     0.58538194444444447
 4 TRUE  2     NA    1.34  c     44958 <NA>         <NA>  0.959050925925926
 5 FALSE 2     NA    <NA>  <NA>  <NA>  <NA>         2     0.72561342592592604
 6 FALSE 3     NA    1.56  e     <NA>  <NA>         <NA>  <NA>
 7 FALSE 1     NA    1.7   f     44987 <NA>         2.7   0.36525462962962968
 8 <NA>  <NA>  NA    <NA>  <NA>  <NA>  <NA>         <NA>  <NA>
 9 FALSE 2     NA    23    h     45284 <NA>         25    <NA>
10 FALSE 3     NA    67.3  i     45285 <NA>         3     <NA>
11 <NA>  1     NA    123   <NA>  45138 <NA>         122   <NA>
```

```
# Custom NA-Werte definieren
df <- read_excel(
  "output/example.xlsx",
  na = c("", "NA", "#NUM!", "#DIV/0!")
)
```

```
New names:
• `` -> `...3`
```

```
df
```

```
# A tibble: 10 × 9
   Var1   Var2 ...3    Var3 Var4  Var5                Var6          Var7
   <lgl> <dbl> <lgl>  <dbl> <chr> <dttm>              <chr>        <dbl>
 1 TRUE      1 NA       1    a     2023-05-29 00:00:00 3209324 This  NA
 2 TRUE     NA NA      NA    b     2023-05-23 00:00:00 <NA>           0
 3 TRUE      2 NA       1.34 c     2023-02-01 00:00:00 <NA>          NA
 4 FALSE     2 NA      NA    <NA>  NA                  <NA>           2
 5 FALSE     3 NA       1.56 e     NA                  <NA>          NA
 6 FALSE     1 NA       1.7  f     2023-03-02 00:00:00 <NA>           2.7
 7 NA       NA NA      NA    <NA>  NA                  <NA>          NA
 8 FALSE     2 NA      23    h     2023-12-24 00:00:00 <NA>          25
 9 FALSE     3 NA      67.3  i     2023-12-25 00:00:00 <NA>           3
10 NA        1 NA     123    <NA>  2023-07-31 00:00:00 <NA>         122
# i 1 more variable: Var8 <dttm>
```

```r
# Kombination mehrerer Optionen
df <- read_excel(
  "output/example.xlsx",
  sheet = "Sheet1",
  range = "B2:F8",
  col_names = TRUE
)
```

```
New names:
• `` -> `...3`
```

```r
df
```

```
# A tibble: 6 × 5
  Var1   Var2 ...3   Var3 Var4
  <lgl> <dbl> <lgl> <dbl> <chr>
1 TRUE      1 NA      1    a
2 TRUE     NA NA    NA    b
3 TRUE      2 NA      1.34 c
4 FALSE     2 NA    NA    <NA>
5 FALSE     3 NA      1.56 e
6 FALSE     1 NA      1.7  f
```

> **i** Range-Syntax
>
> Die `range`-Option akzeptiert Excel-Notation (z.B. `"B3:F20"`) oder auch nur Startpunkte (z.B. `"B3"` liest ab B3 bis zum Ende).

---

2. Export: Professional Formatting

Der Export mit `openxlsx2` geht weit über das einfache Schreiben von Daten hinaus. Während base R und viele andere Packages lediglich die nackten Zahlen und Texte in eine Excel-Datei schreiben, ermöglicht uns `openxlsx2` die Erstellung von professionell formatierten Excel-Dateien, die direkt präsentationsreif sind. Wir können Spaltenbreiten anpassen, Header hervorheben, bedingte Formatierungen anwenden und vieles mehr – alles programmatisch und reproduzierbar.

Excel-Dateien direkt aus R öffnen

Nach dem Erstellen einer Excel-Datei können wir diese direkt aus R heraus öffnen, um das Ergebnis zu überprüfen:

```
# Windows
shell.exec("output/trial_table.xlsx")

# macOS/Linux
system2("open", "output/trial_table.xlsx")  # macOS
system2("xdg-open", "output/trial_table.xlsx")  # Linux
```

Beispieldaten erstellen

Für alle folgenden Beispiele verwenden wir einen kleinen, konsistenten Datensatz aus einer klinischen Studie. Dieser enthält Patienten-IDs, Behandlungsgruppen, Messzeitpunkte, Outcome-Werte und Besuchsdaten. So können wir die verschiedenen Formatierungsmöglichkeiten an einem durchgängigen Beispiel demonstrieren:

```
set.seed(42)
trial_data <- tibble(
  patient_id = glue("P{str_pad(1:12, width = 3, pad = '0')}"),
  treatment = rep(c("Drug A", "Drug B", "Control"), each = 4),
  timepoint = rep(c("Baseline", "Week 4", "Week 8"), times = 4),
  outcome = round(rnorm(12, mean = 50, sd = 10), 1),
  visit_date = seq.Date(from = as.Date("2024-01-15"), by = "week", length.out = 12)
)

trial_data
```

```
# A tibble: 12 × 5
   patient_id treatment timepoint outcome visit_date
   <glue>     <chr>     <chr>       <dbl> <date>
 1 P001       Drug A    Baseline     63.7 2024-01-15
 2 P002       Drug A    Week 4       44.4 2024-01-22
 3 P003       Drug A    Week 8       53.6 2024-01-29
 4 P004       Drug A    Baseline     56.3 2024-02-05
 5 P005       Drug B    Week 4       54   2024-02-12
 6 P006       Drug B    Week 8       48.9 2024-02-19
 7 P007       Drug B    Baseline     65.1 2024-02-26
 8 P008       Drug B    Week 4       49.1 2024-03-04
 9 P009       Control   Week 8       70.2 2024-03-11
10 P010       Control   Baseline     49.4 2024-03-18
11 P011       Control   Week 4       63   2024-03-25
12 P012       Control   Week 8       72.9 2024-04-01
```

2.1 Basics Review (sehr kurz)

Zur Erinnerung: Das grundlegende Erstellen einer Excel-Datei folgt immer demselben Muster. Wir erstellen ein Workbook-Objekt, fügen ein oder mehrere Worksheets hinzu, schreiben Daten hinein und speichern das Workbook als `.xlsx`-Datei. Dieser Workflow bildet die Basis für alle weiteren Formatierungen:

```
# Workbook erstellen
wb <- wb_workbook()

# Worksheet hinzufügen
wb <- wb |> wb_add_worksheet("Trial Data")

# Daten schreiben
```

```r
wb <- wb |> wb_add_data(x = trial_data)

# Speichern
wb_save(wb, "output/trial_basic.xlsx", overwrite = TRUE)
```

2.2 Column Widths

Eine der ersten Dinge, die uns beim Öffnen einer frisch exportierten Excel-Datei auffällt, sind oft zu schmale oder zu breite Spalten. Lange Texte werden abgeschnitten, Zahlen als `###` angezeigt, während andere Spalten unnötig viel Platz verschwenden. Mit `wb_set_col_widths()` können wir dieses Problem elegant lösen: Die Option `widths = "auto"` berechnet automatisch die optimale Breite basierend auf dem Inhalt jeder Spalte. So werden alle Daten vollständig und übersichtlich dargestellt, ohne dass wir manuell in Excel nacharbeiten müssen.

```r
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data(x = trial_data) |>
  # Automatische Breite für alle Spalten
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto")

wb_save(wb, "output/trial_colwidths.xlsx", overwrite = TRUE)
```

> **Tipp**
>
> Alternativ können wir auch spezifische Breiten in Excel-Einheiten setzen, beispielsweise wenn wir genau wissen, wie breit bestimmte Spalten sein sollen:
>
> ```r
> wb_set_col_widths(cols = 1:3, widths = c(15, 20, 12))
> ```

2.3 Header Styling

Die Header-Zeile ist der wichtigste visuelle Orientierungspunkt in einer Tabelle. In professionellen Excel-Dateien sind die Spaltennamen daher typischerweise fettgedruckt und farblich hervorgehoben – meist mit einem dezenten grauen Hintergrund. Diese Formatierung macht die Tabelle sofort lesbarer und verleiht ihr ein professionelles Erscheinungsbild. Mit `wb_add_font()` machen wir den Text fett, mit `wb_add_fill()` fügen wir die Hintergrundfarbe hinzu. Beide Funktionen wenden wir gezielt auf die erste Zeile (den Header) an:

```r
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto") |>
  # Header fett + grauer Hintergrund
  wb_add_font(dims = "A1:E1", bold = TRUE, size = 11) |>
  wb_add_fill(dims = "A1:E1", color = wb_color(hex = "FFD3D3D3"))

wb_save(wb, "output/trial_header.xlsx", overwrite = TRUE)
```

2.4 Excel Tables (filterbar)

Während `wb_add_data()` einfach nur Zellwerte schreibt, erstellt `wb_add_data_table()` eine richtige Excel-Tabelle mit eingebauter Funktionalität. Excel-Tabellen bieten automatisch Filter-Buttons in der Header-Zeile, strukturierte Referenzen für Formeln und ein einheitliches

Design. Das ist besonders praktisch, wenn wir die Datei später an Kollegen weitergeben, die darin filtern oder sortieren möchten. Die verschiedenen `table_style`-Optionen bieten vorgefertigte Designs, die wir direkt anwenden können:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  # wb_add_data_table() statt wb_add_data()
  wb_add_data_table(
    x = trial_data,
    table_style = "TableStyleMedium2"
  ) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto")

wb_save(wb, "output/trial_table.xlsx", overwrite = TRUE)
```

> **i Verfügbare Table Styles**
>
> Excel bietet viele vorgefertigte Styles: `"TableStyleLight1"` bis `"TableStyleLight21"`, `"TableStyleMedium1"` bis `"TableStyleMedium28"`, etc. Am besten einfach verschiedene ausprobieren, um den passenden Stil zu finden!

2.5 Gridlines ausschalten

Standardmäßig zeigt Excel Gitterlinien auf dem gesamten Worksheet, selbst in leeren Bereichen. Das kann bei kleineren, fokussierten Tabellen ablenkend wirken. Wenn wir ein cleanes Layout bevorzugen, bei dem nur die Zellen mit Daten durch Rahmen hervorgehoben sind, können wir die Gitterlinien mit `grid_lines = FALSE` beim Erstellen des Worksheets deaktivieren. In Kombination mit einer Excel-Tabelle (die eigene Rahmen mitbringt) erhalten wir so ein sehr aufgeräumtes, professionelles Erscheinungsbild:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data", grid_lines = FALSE) |>
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto")

wb_save(wb, "output/trial_nogrid.xlsx", overwrite = TRUE)
```

2.6 Conditional Formatting

Bedingte Formatierung ist eines der mächtigsten Features in Excel und besonders nützlich, um Muster in Daten hervorzuheben. Statt manuell durch Spalten zu scrollen und Werte zu vergleichen, können wir Zellen automatisch basierend auf ihrem Wert einfärben, mit Balken versehen oder durch Icons markieren lassen. Die folgenden Beispiele zeigen drei häufige Anwendungsfälle.

Color Scales (Farbverläufe)

Mit Color Scales wird jede Zelle basierend auf ihrem Wert eingefärbt – niedrige Werte beispielsweise rot, mittlere gelb, hohe grün. Das ermöglicht einen sofortigen visuellen Überblick über die Verteilung der Werte. Besonders nützlich für Outcome-Variablen, Scores oder jegliche Messwerte, bei denen die Größenordnung relevant ist:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
```

```r
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto") |>
  # Color Scale für outcome-Spalte (Spalte D = 4)
  wb_add_conditional_formatting(
    dims = "D2:D13",  # ohne Header
    type = "colorScale",
    style = c("red", "yellow", "green"),
    rule = c(0, 50, 100)
  )

wb_save(wb, "output/trial_colorscale.xlsx", overwrite = TRUE)
```

Data Bars (Balken in Zellen)

Data Bars zeigen einen horizontalen Balken in jeder Zelle, dessen Länge dem Wert entspricht. Das funktioniert wie ein Mini-Balkendiagramm direkt in der Tabelle und macht Größenunterschiede auf einen Blick erkennbar. Die Zahlen bleiben dabei weiterhin sichtbar, sodass wir sowohl den exakten Wert als auch die visuelle Proportion sehen:

```r
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto") |>
  # Data Bars für outcome-Spalte
  wb_add_conditional_formatting(
    dims = "D2:D13",
    type = "dataBar",
    style = c("#4472C4"),  # Blau
    params = list(showValue = TRUE, gradient = TRUE)
  )

wb_save(wb, "output/trial_databars.xlsx", overwrite = TRUE)
```

Rule-based Formatting

Manchmal wollen wir nicht alle Werte einfärben, sondern nur die, die ein bestimmtes Kriterium erfüllen – beispielsweise alle Outcome-Werte über einem Schwellenwert. Mit rule-based Formatting definieren wir eine Bedingung (z.B. `">55"`) und einen Style (Schriftfarbe, Hintergrundfarbe), der auf die entsprechenden Zellen angewandt wird. Das ist ideal, um kritische Werte hervorzuheben:

```r
# Custom Style für Werte > 55
high_style <- create_dxfs_style(
  font_color = wb_color(hex = "FF006100"),
  bg_fill = wb_color(hex = "FFC6EFCE")
)

wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto")

# Style zum Workbook hinzufügen
wb$styles_mgr$add(high_style, "high_values")

# Conditional Formatting anwenden
wb <- wb |>
  wb_add_conditional_formatting(
    dims = "D2:D13",
    type = "expression",
    rule = ">55",
    style = "high_values"
```

```
  )

wb_save(wb, "output/trial_rules.xlsx", overwrite = TRUE)
```

> ! Weitere Conditional Formatting Optionen
>
> Es gibt viele weitere Typen wie `"topN"` (die Top-N höchsten Werte), `"bottomN"`, `"duplicatedValues"` (Duplikate markieren), `"iconSet"` (Ampel-Icons) etc. Für Details siehe das Conditional Formatting Vignette.

2.7 Freeze Panes

Bei längeren Tabellen verlieren wir beim Scrollen nach unten schnell den Überblick, welche Spalte welche Daten enthält – denn die Header-Zeile verschwindet aus dem Sichtfeld. Mit Freeze Panes können wir die erste Zeile (oder auch die erste Spalte) fixieren, sodass sie beim Scrollen immer sichtbar bleibt. Das ist eine der meistgenutzten Features in Excel und macht das Arbeiten mit größeren Datensätzen erheblich komfortabler:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto") |>
  # Erste Zeile fixieren
  wb_freeze_pane(first_row = TRUE)

wb_save(wb, "output/trial_freeze.xlsx", overwrite = TRUE)
```

> Tipp
>
> Wir können auch die erste Spalte fixieren (nützlich bei breiten Tabellen mit vielen Spalten):
>
> ```
> wb_freeze_pane(first_col = TRUE)
> ```
>
> Oder sogar beides gleichzeitig:
>
> ```
> wb_freeze_pane(first_row = TRUE, first_col = TRUE)
> ```

2.8 Hyperlinks

Hyperlinks machen Excel-Dateien interaktiv und verknüpfen verschiedene Informationen miteinander. Wir können externe URLs einbinden (z.B. zu Protokollen oder Dokumentationen) oder interne Links zu anderen Sheets erstellen. Das ist besonders praktisch für Inhaltsverzeichnisse oder wenn wir zwischen verschiedenen Tabellenblättern navigieren wollen. In `openxlsx2` gibt es zwei verschiedene Ansätze: externe Links verwenden `wb_add_hyperlink()`, während interne Sheet-Links über `create_hyperlink()` und `wb_add_formula()` erstellt werden:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Overview") |>
  wb_add_data(x = tibble(
    Description = c("Study Protocol", "Raw Data", "Analysis")
  ))
```

```r
# Trial Data Sheet hinzufügen
wb <- wb |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto")

# Externe URL als Hyperlink
wb <- wb |>
  wb_add_data(
    sheet = "Overview",
    dims = "B2",
    x = "Protocol Document"
  ) |>
  wb_add_hyperlink(
    sheet = "Overview",
    dims = "B2",
    target = "https://example.com/protocol",
    tooltip = "Link to study protocol"
  )

# Interner Link zu anderem Sheet (mit create_hyperlink + wb_add_formula)
internal_link <- create_hyperlink(
  sheet = "Trial Data",
  row = 1,
  col = 1,
  text = "Go to Trial Data"
)

wb <- wb |>
  wb_add_formula(
    sheet = "Overview",
    dims = "B3",
    x = internal_link
  )

wb_save(wb, "output/trial_hyperlinks.xlsx", overwrite = TRUE)
```

> **i** Externe vs. Interne Links
>
> - **Externe URLs**: `wb_add_hyperlink()` mit `target =`
> - **Interne Sheet-Links**: `create_hyperlink()` + `wb_add_formula()`

2.9 Date/Number Formats

Excel interpretiert Zahlen und Datumsangaben oft anders als wir es erwarten – Datumsangaben werden als Zahlen dargestellt, Dezimalstellen fehlen, oder Währungen erscheinen ohne Symbol. Mit Number Formats können wir exakt festlegen, wie Werte angezeigt werden sollen. Dabei ändern wir nur die Darstellung, nicht den zugrundeliegenden Wert. Das ist besonders wichtig für Berichte, die wir an andere weitergeben, damit die Daten sofort in der gewünschten Form erscheinen:

```r
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto") |>
  # Outcome als Zahl mit 1 Dezimalstelle
  wb_add_numfmt(dims = "D2:D13", numfmt = "0.0") |>
  # Datum als dd.mm.yyyy
  wb_add_numfmt(dims = "E2:E13", numfmt = "dd.mm.yyyy")
```

```
wb_save(wb, "output/trial_formats.xlsx", overwrite = TRUE)
```

> **Häufige Number Formats**
>
> - `"0.00"` - zwei Dezimalstellen
> - `"0.00%"` - Prozent
> - `"#,##0.00"` - Tausender-Trennzeichen
> - `"#,##0.00 €"` - Währung
> - `"dd.mm.yyyy"` - Datum deutsch
> - `"yyyy-mm-dd"` - Datum ISO
> - `"[h]:mm:ss"` - Zeit über 24h
>
> Für Custom-Formate mit Text: siehe openxlsx2 Styling Manual

2.10 Advanced Beispiele aus ox2-book

Das ox2-book ist das umfassende Handbuch zu `openxlsx2` und enthält zahlreiche fortgeschrittene Beispiele und Techniken. Im Folgenden zeigen wir einige Highlights aus den Kapiteln zu Styling, Conditional Formatting und Formeln. Diese Beispiele kratzen nur an der Oberfläche dessen, was möglich ist – für tiefergehende Anwendungen lohnt sich ein Blick in die jeweiligen Kapitel.

Text Rotation (Kap. 5: Styling)

Text um 45° zu drehen ist besonders nützlich bei Tabellen mit vielen Spalten und langen Header-Texten. Der gedrehte Text spart horizontal Platz und macht die Tabelle kompakter, ohne die Lesbarkeit zu beeinträchtigen. In Kombination mit Fettdruck und einer Hintergrundfarbe entsteht ein sehr professioneller Look:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = 12) |>
  # Text rotation + Styling
  wb_add_cell_style(
    dims = "A1:E1",
    horizontal = "center",
    text_rotation = 45
  ) |>
  wb_add_font(dims = "A1:E1", bold = TRUE, size = 10) |>
  wb_add_fill(dims = "A1:E1", color = wb_color(hex = "FF4472C4"))

wb_save(wb, "output/trial_rotation.xlsx", overwrite = TRUE)
```

Weitere Styling-Optionen: Kapitel 5 - Styling of worksheets

Icon Sets (Kap. 7: Conditional Formatting)

Icon Sets sind eine elegante Variante der bedingten Formatierung: Statt Zellen einzufärben, fügen wir kleine Icons hinzu (z.B. Ampel-Symbole), die auf einen Blick zeigen, ob Werte gut, mittel oder schlecht sind. Das ist besonders nützlich für Dashboards und Reports, da die Icons auch beim Ausdrucken gut erkennbar bleiben:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data_table(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto") |>
  # Icon Set: 3 Ampelfarben
  wb_add_conditional_formatting(
    dims = "D2:D13",
    type = "iconSet",
    params = list(
      iconSet = "3Symbols",  # Ampel: rot/gelb/grün
      showValue = TRUE
    )
  )

wb_save(wb, "output/trial_icons.xlsx", overwrite = TRUE)
```

Weitere Icon Sets: `"3Arrows"`, `"4Rating"`, `"5Quarters"` etc. Siehe Conditional Formatting Vignette.

Excel Formulas (Kap. 8: Formulas)

Excel-Formeln sind das Herzstück von dynamischen Spreadsheets. Mit `openxlsx2` können wir Formeln direkt in Zellen schreiben, die dann beim Öffnen der Datei in Excel automatisch berechnet werden. Das ist praktisch für Summen, Durchschnitte oder komplexere Berechnungen. Wichtig: Die Formeln werden erst in Excel ausgewertet, nicht in R:

```
# Beispiel mit SUM-Formel
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data(x = trial_data) |>
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto") |>
  # SUM-Formel für Gesamtsumme
  wb_add_formula(dims = "D14", x = "SUM(D2:D13)") |>
  # AVERAGE-Formel
  wb_add_formula(dims = "D15", x = "AVERAGE(D2:D13)") |>
  # Labels hinzufügen
  wb_add_data(dims = "C14", x = "Total") |>
  wb_add_data(dims = "C15", x = "Average")

wb_save(wb, "output/trial_formulas.xlsx", overwrite = TRUE)
```

Weitere Formula-Beispiele: Kapitel 8 - Spreadsheet formulas

Pivot Tables (Kap. 9: Kurze Erwähnung)

`openxlsx2` kann auch Pivot Tables erstellen, allerdings ist dies ein fortgeschrittenes und komplexes Thema. Pivot Tables sind mächtige Werkzeuge zur Datenanalyse und -zusammenfassung direkt in Excel. Die Erstellung ist jedoch deutlich aufwändiger als die anderen hier gezeigten Features. Für Details und vollständige Beispiele siehe Kapitel 9 - Pivot tables.

---

3. Template-Workflow: Bestehende Excel-Dateien befüllen

Bisher haben wir Excel-Dateien immer von Grund auf neu erstellt. In der Praxis gibt es jedoch häufig einen anderen Anwendungsfall: Wir haben eine **vorformatierte Excel-Vorlage** (Template) mit komplexem Layout, Corporate Design, Formeln oder Pivot-Tabellen, und

möchten diese nur noch mit aktuellen Daten befüllen. Das manuelle Nachbauen solcher Templates in R wäre extrem aufwändig – stattdessen laden wir einfach die bestehende Datei und schreiben nur die Daten hinein.

Wann ist der Template-Workflow sinnvoll?

Der Template-Workflow ist besonders nützlich, wenn:

- Die Excel-Datei ein **komplexes, festes Layout** hat (z.B. Berichtsvorlagen mit Logos, Rahmen, mehreren Bereichen)
- Das **Corporate Design** bereits in der Vorlage implementiert ist
- Die Datei **Excel-Formeln** enthält, die auf die eingefügten Daten verweisen sollen
- Regelmäßig **wiederkehrende Reports** erstellt werden (z.B. monatliche Auswertungen)
- Mehrere Personen dieselbe Vorlage nutzen und nur die Daten variieren

Grundprinzip

Der Workflow besteht aus drei Schritten:

1. **Vorlage kopieren** – Die Original-Vorlage bleibt unverändert
2. **Kopie laden** – Mit `wb_load()` öffnen wir die Kopie
3. **Daten einfügen** – Mit `wb_add_data()` schreiben wir an die richtigen Positionen

```
# 1. Vorlage kopieren (Original bleibt erhalten)
file.copy(
  from = "vorlagen/Monatsbericht_Vorlage.xlsx",
  to = "output/Monatsbericht_Januar.xlsx",
  overwrite = TRUE
)

# 2. Kopie laden
wb <- wb_load("output/Monatsbericht_Januar.xlsx")

# 3. Daten an die richtigen Stellen schreiben
wb <- wb |>
  wb_add_data(sheet = "Daten", x = meine_daten, start_row = 5, start_col = 2)

# 4. Speichern
wb_save(wb, "output/Monatsbericht_Januar.xlsx", overwrite = TRUE)
```

Praxisbeispiel: Auswertungstabelle befüllen

Stellen wir uns vor, wir haben eine Excel-Vorlage mit drei Tabellenblättern für verschiedene Auswertungen. Die Vorlage enthält bereits Header, Formatierungen und Summenformeln – wir müssen nur noch die Daten einfügen.

```
# Beispieldaten vorbereiten
set.seed(123)
ergebnis_1 <- tibble(
  Kategorie = c("A", "B", "C"),
  Anzahl = c(45, 32, 28),
  Anteil = c(0.43, 0.30, 0.27)
)

ergebnis_2 <- tibble(
  Region = c("Nord", "Süd", "Ost", "West"),
  Umsatz = c(12500, 18300, 9800, 15200)
)
```

```
# Für dieses Beispiel erstellen wir eine "Vorlage"
# (in der Praxis wäre das eine bereits existierende Datei)
template_wb <- wb_workbook() |>
  wb_add_worksheet("Übersicht") |>
  wb_add_data(x = "Monatsbericht", dims = "A1") |>
  wb_add_font(dims = "A1", bold = TRUE, size = 16) |>
  wb_add_worksheet("Kategorien") |>
  wb_add_data(x = tibble(Kategorie = character(), Anzahl = numeric(), Anteil =
numeric())) |>
  wb_add_font(dims = "A1:C1", bold = TRUE) |>
  wb_add_fill(dims = "A1:C1", color = wb_color(hex = "FFD3D3D3")) |>
  wb_add_worksheet("Regionen") |>
  wb_add_data(x = tibble(Region = character(), Umsatz = numeric())) |>
  wb_add_font(dims = "A1:B1", bold = TRUE) |>
  wb_add_fill(dims = "A1:B1", color = wb_color(hex = "FFD3D3D3"))

wb_save(template_wb, "output/vorlage.xlsx", overwrite = TRUE)

# --- TEMPLATE-WORKFLOW ---

# 1. Vorlage kopieren
file.copy(
  from = "output/vorlage.xlsx",
  to = "output/bericht_aktuell.xlsx",
  overwrite = TRUE
)
```

```
[1] TRUE
```

```
# 2. Kopie laden
wb <- wb_load("output/bericht_aktuell.xlsx")

# 3. Daten einfügen (OHNE Header, da bereits in Vorlage)
wb <- wb |>
  wb_add_data(
    sheet = "Kategorien",
    x = ergebnis_1,
    start_row = 2,        # Zeile 1 ist Header
    col_names = FALSE     # Keine Spaltennamen schreiben
  ) |>
  wb_add_data(
    sheet = "Regionen",
    x = ergebnis_2,
    start_row = 2,
    col_names = FALSE
  )

# 4. Speichern
wb_save(wb, "output/bericht_aktuell.xlsx", overwrite = TRUE)
```

Wichtige Argumente bei wb_add_data()

Beim Befüllen von Templates sind folgende Argumente besonders relevant:

Argument	Beschreibung	Typischer Wert
`sheet`	Name oder Index des Tabellenblatts	`"Daten"` oder `1`
`x`	Die einzufügenden Daten (data.frame/tibble)	`meine_daten`
`start_row`	Startzeile für die Daten	`2` (wenn Zeile 1 = Header)

Argument	Beschreibung	Typischer Wert
`start_col`	Startspalte für die Daten	`1` oder `"B"`
`col_names`	Spaltennamen schreiben?	`FALSE` (Header in Vorlage)
`na.strings`	Wie sollen NA-Werte dargestellt werden?	`""` (leere Zelle)

Tipp: Positionen in der Vorlage dokumentieren

Wenn die Vorlage komplex ist, empfiehlt es sich, die Einfügepositionen zu dokumentieren:

```
# Positionen in der Vorlage "Tabellenband.xlsx":
# - Sheet "Häufigkeiten": Daten ab Zeile 2, Spalte A
# - Sheet "Kreuztabelle": Daten ab Zeile 5, Spalte B
# - Sheet "Zusammenfassung": Daten ab Zeile 3, Spalte A
```

Achtung: Bestehende Daten werden überschrieben

`wb_add_data()` überschreibt den Zielbereich ohne Warnung. Wenn die Vorlage bereits Daten enthält (z.B. Beispielwerte), werden diese ersetzt. Formeln, die auf diese Zellen verweisen, werden automatisch mit den neuen Werten berechnet.

Zusammenfassung

In diesem Kapitel haben wir gelernt, wie wir mit R professionelle, präsentationsreife Excel-Dateien erstellen können. Wir haben gesehen, wie wir beim Import präzise mit mehreren Sheets und unordentlichen Daten umgehen, und beim Export haben wir eine Vielzahl von Formatierungsmöglichkeiten kennengelernt, die unsere Excel-Dateien von einfachen Daten-Dumps zu ansprechenden, nutzerfreundlichen Reports machen.

Import: - Multiple Sheets systematisch einlesen mit `excel_sheets()` und `map()` - Präzise Ranges und Custom NA-Werte für chaotische Dateien

Export: - Automatische Spaltenbreite für optimale Darstellung - Professionelles Header-Styling mit Fettdruck und Hintergrundfarbe - Filterbare Excel-Tabellen statt einfacher Zellbereiche - Conditional Formatting (Color Scales, Data Bars, Rules) für visuelle Hervorhebung - Freeze Panes für bessere Navigation in großen Tabellen - Hyperlinks für Verknüpfungen zu URLs und anderen Sheets - Custom Date/Number Formats für korrekte Darstellung - Advanced Features aus dem ox2-book für spezielle Anforderungen

Template-Workflow: - Bestehende Excel-Vorlagen mit `wb_load()` laden statt neu erstellen - Daten gezielt an bestimmte Positionen schreiben mit `start_row`, `start_col`, `col_names = FALSE` - Ideal für wiederkehrende Reports mit festem Layout und Corporate Design

Weiterführende Ressourcen:

- openxlsx2 Dokumentation

- ox2-book - The openxlsx2 book
- readxl Dokumentation

Datum: 2026-02-08

Bibliography