2. Advanced Excel Workflows

Professional Import and Export with readxl and openxlsx2

Dr. Paul Schmidt

Loading Packages

```
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
```

```
# Create output directory if it doesn't exist
if (!dir.exists("output")) {
  dir.create("output")
}
```

Preparing the Example Excel File

For the import examples, we use an Excel file that comes bundled with the `openxlsx2` package. It contains two sheets and various data types, making it ideal for our demonstrations. We copy it to our output directory so we can work with it throughout the chapter:

```
# Locate example file from openxlsx2 package
example_file <- system.file("extdata", "openxlsx2_example.xlsx", package =
"openxlsx2")

# Copy to output folder
file.copy(
  from = example_file,
  to = "output/example.xlsx",
  overwrite = TRUE
)
```

```
[1] TRUE
```

1. Import: Beyond the Basics

In earlier chapters, we learned how to use `readxl::read_excel()` to read single Excel files. In practice, however, we often encounter more complex situations: files with multiple worksheets, messy structures with headers and footnotes, or specific cell ranges that we want to read selectively. Here we expand our knowledge to cover these common special cases.

1.1 Importing Multiple Sheets

Excel files often contain multiple worksheets that belong together thematically - for example, different measurement timepoints from an experiment or different datasets from a study. Instead of loading each sheet individually and manually, we can use `excel_sheets()` to read all available sheet names and then systematically import them in a loop. We store the result in a named list so we can access the respective datasets directly via the sheet names.

```
# Our example file
file_path <- "output/example.xlsx"

# List all sheet names
sheet_names <- excel_sheets(file_path)
sheet_names
```

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

```
# Import all sheets into a named list
all_data <- map(sheet_names, \(sheet) read_excel(file_path, sheet = sheet))
```

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

```
names(all_data) <- sheet_names

# Access individual 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>
```

Alternatively, we can use a classic for loop if we don't want to use `purrr` or prefer more explicit logic:

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

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

1.2 Precise Reading: Ranges & Skip

In the real working world, Excel files are rarely as tidy as in textbooks. We often find descriptive text above the actual data, footnotes below, empty rows as separators, or the actual data only starts at row 10 and column C. For such situations, `readxl` offers several useful options that let us precisely control which parts of the file we want to read.

With the `range` option, we can specify an exact cell range in Excel notation (e.g., `"B5:G20"`) to read only that range. The `skip` option skips a certain number of rows at the beginning of the file - practical when the data only starts after several header rows. If the column names themselves are chaotic or unusable, we can disable automatic header reading with `col_names = FALSE`. And finally, we can use `na` to specify which character strings should be interpreted as missing values - since not everyone uses "NA" for missing values.

```
# Read only a specific cell range (example with our file)
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
```

```
# Skip first 2 rows
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>
```

```
# No automatic column names (when header is chaotic)
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>
```

```
# Define custom NA values
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>
```

```
# Combination of multiple options
df <- read_excel(
  "output/example.xlsx",
  sheet = "Sheet1",
  range = "B2:F8",
  col_names = TRUE
)
```

```
New names:
• `` -> `...3`
```

```
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
>
> The `range` option accepts Excel notation (e.g., `"B3:F20"`) or just starting points (e.g., `"B3"` reads from B3 to the end).

---

2. Export: Professional Formatting

Exporting with `openxlsx2` goes far beyond simply writing data. While base R and many other packages merely write the bare numbers and text to an Excel file, `openxlsx2` enables us to create professionally formatted Excel files that are immediately presentation-ready. We can adjust column widths, highlight headers, apply conditional formatting, and much more - all programmatically and reproducibly.

Opening Excel Files Directly from R

After creating an Excel file, we can open it directly from R to check the result:

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

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

Creating Example Data

For all the following examples, we use a small, consistent dataset from a clinical trial. It contains patient IDs, treatment groups, timepoints, outcome values, and visit dates. This allows us to demonstrate the various formatting options using a consistent example throughout:

```
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 (Very Brief)

As a reminder: the basic creation of an Excel file always follows the same pattern. We create a workbook object, add one or more worksheets, write data to them, and save the workbook as an `.xlsx` file. This workflow forms the basis for all further formatting:

```
# Create workbook
wb <- wb_workbook()

# Add worksheet
wb <- wb |> wb_add_worksheet("Trial Data")

# Write data
wb <- wb |> wb_add_data(x = trial_data)
```

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

2.2 Column Widths

One of the first things we notice when opening a freshly exported Excel file are often columns that are too narrow or too wide. Long texts get cut off, numbers display as `###`, while other columns waste unnecessary space. With `wb_set_col_widths()` we can elegantly solve this problem: the option `widths = "auto"` automatically calculates the optimal width based on each column's content. This ensures all data is displayed completely and clearly without requiring manual adjustments in Excel.

```r
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  wb_add_data(x = trial_data) |>
  # Automatic width for all columns
  wb_set_col_widths(cols = 1:ncol(trial_data), widths = "auto")

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

> **Tip**
>
> Alternatively, we can set specific widths in Excel units, for example when we know exactly how wide certain columns should be:
>
> ```r
> wb_set_col_widths(cols = 1:3, widths = c(15, 20, 12))
> ```

2.3 Header Styling

The header row is the most important visual orientation point in a table. In professional Excel files, column names are therefore typically bold and highlighted with color - usually with a subtle gray background. This formatting makes the table immediately more readable and gives it a professional appearance. With `wb_add_font()` we make the text bold, with `wb_add_fill()` we add the background color. We apply both functions specifically to the first row (the header):

```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 bold + gray background
  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 (Filterable)

While `wb_add_data()` simply writes cell values, `wb_add_data_table()` creates a proper Excel table with built-in functionality. Excel tables automatically provide filter buttons in the header row, structured references for formulas, and a consistent design. This is especially practical when we later share the file with colleagues who want to filter or sort within it. The various `table_style` options offer predefined designs that we can apply directly:

```
wb <- wb_workbook() |>
  wb_add_worksheet("Trial Data") |>
  # wb_add_data_table() instead of 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)
```

Available Table Styles

Excel offers many predefined styles: `"TableStyleLight1"` through `"TableStyleLight21"`, `"TableStyleMedium1"` through `"TableStyleMedium28"`, etc. Best to just try different ones to find the right style!

2.5 Turning Off Gridlines

By default, Excel shows gridlines across the entire worksheet, even in empty areas. This can be distracting for smaller, focused tables. If we prefer a clean layout where only the cells with data are highlighted by borders, we can disable the gridlines with `grid_lines = FALSE` when creating the worksheet. Combined with an Excel table (which has its own borders), this gives us a very tidy, professional appearance:

```
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

Conditional formatting is one of the most powerful features in Excel and is particularly useful for highlighting patterns in data. Instead of manually scrolling through columns and comparing values, we can automatically color cells based on their value, add bars, or mark them with icons. The following examples show three common use cases.

Color Scales

With color scales, each cell is colored based on its value - low values might be red, medium yellow, high green. This provides an immediate visual overview of the value distribution. Particularly useful for outcome variables, scores, or any measurements where magnitude is relevant:

```
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") |>
  # Color Scale for outcome column (Column D = 4)
  wb_add_conditional_formatting(
    dims = "D2:D13",  # without header
    type = "colorScale",
    style = c("red", "yellow", "green"),
    rule = c(0, 50, 100)
```

```
  )

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

Data Bars

Data bars show a horizontal bar in each cell whose length corresponds to the value. This works like a mini bar chart directly in the table and makes size differences visible at a glance. The numbers remain visible so we can see both the exact value and the visual proportion:

```
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 for outcome column
  wb_add_conditional_formatting(
    dims = "D2:D13",
    type = "dataBar",
    style = c("#4472C4"),  # Blue
    params = list(showValue = TRUE, gradient = TRUE)
  )

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

Rule-based Formatting

Sometimes we don't want to color all values, just those that meet a certain criterion - for example, all outcome values above a threshold. With rule-based formatting, we define a condition (e.g., `">55"`) and a style (font color, background color) that is applied to the corresponding cells. This is ideal for highlighting critical values:

```
# Custom style for values > 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")

# Add style to workbook
wb$styles_mgr$add(high_style, "high_values")

# Apply conditional formatting
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)
```

! More Conditional Formatting Options

There are many more types like `"topN"` (the top N highest values), `"bottomN"`, `"duplicatedValues"` (mark duplicates), `"iconSet"` (traffic light icons), etc. For details, see the Conditional Formatting Vignette.

2.7 Freeze Panes

With longer tables, we quickly lose track of which column contains which data when scrolling down - because the header row disappears from view. With freeze panes, we can fix the first row (or also the first column) so that it always remains visible when scrolling. This is one of the most-used features in Excel and makes working with larger datasets considerably more comfortable:

```
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") |>
  # Fix first row
  wb_freeze_pane(first_row = TRUE)

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

Tip

We can also freeze the first column (useful for wide tables with many columns):

```
wb_freeze_pane(first_col = TRUE)
```

Or even both at the same time:

```
wb_freeze_pane(first_row = TRUE, first_col = TRUE)
```

2.8 Hyperlinks

Hyperlinks make Excel files interactive and connect different pieces of information. We can embed external URLs (e.g., to protocols or documentation) or create internal links to other sheets. This is especially practical for tables of contents or when we want to navigate between different worksheets. In `openxlsx2`, there are two different approaches: external links use `wb_add_hyperlink()`, while internal sheet links are created via `create_hyperlink()` and `wb_add_formula()`:

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

# Add Trial Data sheet
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")

# External URL as 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"
  )

# Internal link to another sheet (with 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 External vs. Internal Links

- **External URLs**: `wb_add_hyperlink()` with `target =`
- **Internal sheet links**: `create_hyperlink()` + `wb_add_formula()`

2.9 Date/Number Formats

Excel often interprets numbers and dates differently than we expect - dates appear as numbers, decimal places are missing, or currencies appear without symbols. With number formats, we can specify exactly how values should be displayed. This only changes the presentation, not the underlying value. This is especially important for reports that we share with others, so the data appears in the desired form immediately:

```
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 as number with 1 decimal place
  wb_add_numfmt(dims = "D2:D13", numfmt = "0.0") |>
  # Date as dd.mm.yyyy
  wb_add_numfmt(dims = "E2:E13", numfmt = "dd.mm.yyyy")

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

ℹ Common Number Formats

- `"0.00"` - two decimal places
- `"0.00%"` - percentage
- `"#,##0.00"` - thousands separator
- `"#,##0.00 €"` - currency
- `"dd.mm.yyyy"` - date German style
- `"yyyy-mm-dd"` - date ISO
- `"[h]:mm:ss"` - time over 24h

For custom formats with text: see openxlsx2 Styling Manual

2.10 Advanced Examples from ox2-book

The ox2-book is the comprehensive handbook for `openxlsx2` and contains numerous advanced examples and techniques. Below we show some highlights from the chapters on styling, conditional formatting, and formulas. These examples only scratch the surface of what's possible - for deeper applications, it's worth looking at the respective chapters.

Text Rotation (Ch. 5: Styling)

Rotating text by 45° is particularly useful for tables with many columns and long header texts. The rotated text saves horizontal space and makes the table more compact without compromising readability. Combined with bold text and a background color, this creates a very professional 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)
```

More styling options: Chapter 5 - Styling of worksheets

Icon Sets (Ch. 7: Conditional Formatting)

Icon sets are an elegant variant of conditional formatting: instead of coloring cells, we add small icons (e.g., traffic light symbols) that show at a glance whether values are good, medium, or poor. This is especially useful for dashboards and reports since the icons are also clearly visible when printed:

```
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") |>
```

```r
  # Icon Set: 3 traffic light colors
  wb_add_conditional_formatting(
    dims = "D2:D13",
    type = "iconSet",
    params = list(
      iconSet = "3Symbols",  # Traffic light: red/yellow/green
      showValue = TRUE
    )
  )

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

More icon sets: `"3Arrows"`, `"4Rating"`, `"5Quarters"`, etc. See Conditional Formatting Vignette.

Excel Formulas (Ch. 8: Formulas)

Excel formulas are the heart of dynamic spreadsheets. With `openxlsx2`, we can write formulas directly into cells that are then automatically calculated when the file is opened in Excel. This is practical for sums, averages, or more complex calculations. Important: the formulas are only evaluated in Excel, not in R:

```r
# Example with SUM formula
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 formula for total
  wb_add_formula(dims = "D14", x = "SUM(D2:D13)") |>
  # AVERAGE formula
  wb_add_formula(dims = "D15", x = "AVERAGE(D2:D13)") |>
  # Add labels
  wb_add_data(dims = "C14", x = "Total") |>
  wb_add_data(dims = "C15", x = "Average")

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

More formula examples: Chapter 8 - Spreadsheet formulas

Pivot Tables (Ch. 9: Brief Mention)

`openxlsx2` can also create pivot tables, although this is an advanced and complex topic. Pivot tables are powerful tools for data analysis and summarization directly in Excel. However, creating them is considerably more involved than the other features shown here. For details and complete examples, see Chapter 9 - Pivot tables.

---

3. Template Workflow: Populating Existing Excel Files

So far, we have always created Excel files from scratch. In practice, however, there is often a different use case: we have a **pre-formatted Excel template** with complex layout, corporate design, formulas, or pivot tables, and we just want to populate it with current data. Manually recreating such templates in R would be extremely time-consuming – instead, we simply load the existing file and write only the data into it.

When is the Template Workflow Useful?

The template workflow is particularly useful when:

- The Excel file has a **complex, fixed layout** (e.g., report templates with logos, borders, multiple areas)
- The **corporate design** is already implemented in the template
- The file contains **Excel formulas** that should reference the inserted data
- **Recurring reports** are created regularly (e.g., monthly evaluations)
- Multiple people use the same template and only the data varies

Basic Principle

The workflow consists of three steps:

1. **Copy template** – The original template remains unchanged
2. **Load copy** – We open the copy with `wb_load()`
3. **Insert data** – We write to the correct positions with `wb_add_data()`

```
# 1. Copy template (original remains intact)
file.copy(
  from = "templates/Monthly_Report_Template.xlsx",
  to = "output/Monthly_Report_January.xlsx",
  overwrite = TRUE
)

# 2. Load copy
wb <- wb_load("output/Monthly_Report_January.xlsx")

# 3. Write data to the correct positions
wb <- wb |>
  wb_add_data(sheet = "Data", x = my_data, start_row = 5, start_col = 2)

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

Practical Example: Populating an Analysis Table

Imagine we have an Excel template with three worksheets for different analyses. The template already contains headers, formatting, and sum formulas – we just need to insert the data.

```
# Prepare example data
set.seed(123)
result_1 <- tibble(
  Category = c("A", "B", "C"),
  Count = c(45, 32, 28),
  Proportion = c(0.43, 0.30, 0.27)
)

result_2 <- tibble(
  Region = c("North", "South", "East", "West"),
  Revenue = c(12500, 18300, 9800, 15200)
)

# For this example, we create a "template"
# (in practice, this would be an existing file)
template_wb <- wb_workbook() |>
  wb_add_worksheet("Overview") |>
  wb_add_data(x = "Monthly Report", dims = "A1") |>
  wb_add_font(dims = "A1", bold = TRUE, size = 16) |>
```

```
  wb_add_worksheet("Categories") |>
  wb_add_data(x = tibble(Category = character(), Count = numeric(), Proportion =
numeric())) |>
  wb_add_font(dims = "A1:C1", bold = TRUE) |>
  wb_add_fill(dims = "A1:C1", color = wb_color(hex = "FFD3D3D3")) |>
  wb_add_worksheet("Regions") |>
  wb_add_data(x = tibble(Region = character(), Revenue = 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/template.xlsx", overwrite = TRUE)

# --- TEMPLATE WORKFLOW ---

# 1. Copy template
file.copy(
  from = "output/template.xlsx",
  to = "output/report_current.xlsx",
  overwrite = TRUE
)
```

```
[1] TRUE
```

```
# 2. Load copy
wb <- wb_load("output/report_current.xlsx")

# 3. Insert data (WITHOUT headers, as they're already in template)
wb <- wb |>
  wb_add_data(
    sheet = "Categories",
    x = result_1,
    start_row = 2,        # Row 1 is header
    col_names = FALSE     # Don't write column names
  ) |>
  wb_add_data(
    sheet = "Regions",
    x = result_2,
    start_row = 2,
    col_names = FALSE
  )

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

Important Arguments for wb_add_data()

When populating templates, the following arguments are particularly relevant:

Argument	Description	Typical Value
`sheet`	Name or index of the worksheet	`"Data"` or `1`
`x`	The data to insert (data.frame/tibble)	`my_data`
`start_row`	Starting row for the data	`2` (if row 1 = header)
`start_col`	Starting column for the data	`1` or `"B"`
`col_names`	Write column names?	`FALSE` (header in template)

Argument	Description	Typical Value
`na.strings`	How should NA values be displayed?	`""` (empty cell)

> **Tip: Document Positions in the Template**
>
> If the template is complex, it's helpful to document the insertion positions:
>
> ```
> # Positions in template "Report_Template.xlsx":
> # - Sheet "Frequencies": Data from row 2, column A
> # - Sheet "CrossTab": Data from row 5, column B
> # - Sheet "Summary": Data from row 3, column A
> ```

> **Warning: Existing Data Will Be Overwritten**
>
> `wb_add_data()` overwrites the target range without warning. If the template already contains data (e.g., example values), they will be replaced. Formulas that reference these cells will automatically calculate with the new values.

Summary

In this chapter, we learned how to create professional, presentation-ready Excel files with R. We saw how to precisely handle multiple sheets and messy data during import, and during export we learned a variety of formatting options that transform our Excel files from simple data dumps into appealing, user-friendly reports.

Import: - Systematically read multiple sheets with `excel_sheets()` and `map()` - Precise ranges and custom NA values for messy files

Export: - Automatic column width for optimal display - Professional header styling with bold text and background color - Filterable Excel tables instead of simple cell ranges - Conditional formatting (color scales, data bars, rules) for visual emphasis - Freeze panes for better navigation in large tables - Hyperlinks for connections to URLs and other sheets - Custom date/number formats for correct display - Advanced features from the ox2-book for special requirements

Template Workflow: - Load existing Excel templates with `wb_load()` instead of creating from scratch - Write data to specific positions with `start_row`, `start_col`, `col_names = FALSE` - Ideal for recurring reports with fixed layout and corporate design

Further Resources:

- openxlsx2 Documentation
- ox2-book - The openxlsx2 book
- readxl Documentation

Date: 2026-02-08

Bibliography