*Cédric Scherer*

# *Graphic Design with ggplot2*

***Create Beautiful and Engaging Data Visualizations in R***

To my family,thank you for all your support and love.

# *Contents*

# *List of Figures*

# *List of Tables*

# *Preface*

Back in 2016, I had to prepare my PhD introductory talk to inform about my plans for the next three years and to showcase my first preliminary results. I planned to create a visualization using small multiples to show various outcomes of the scenarios I ran with my simulation model. I was already using the R programming language for years and quickly came across the graphics library **ggplot2** which comes with the functionality to easily create small multiples. I never liked the syntax and style of plots created in base R or with the **lattice** package, so I immediately fell in love with the idea and implementation of **ggplot2**'s *Grammar of Graphics* that allows to combine and modify plot components in a very sophisticated manner. But because I was short on time, I plotted these figures by trial and error and with the help of lots of googling. The resource I came always back to was a blog entry called "Beautiful plotting in R: A **ggplot2** cheatsheet" by Zev Ross[1]. After giving the talk which contained some decent plots thanks to the blog post, I decided to go through this tutorial step-by-step. I learned so much from it and directly started modifying the codes and adding additional code snippets, chart types, and resources.

Fast forward to 2019. I successfully finished my PhD and started participating in a weekly data visualization challenge called #TidyTuesday[2]. Every week, a raw data set is shared with the aim to explore and visualize the data with **ggplot2**. Thanks to my experience with the **tidyverse** and especially **ggplot2** during my PhD and the open-source approach of the challenge that made it possible to learn from other participants, my visualizations quickly became more advanced and complex.

A few months later, I had built a portfolio of various charts and maps and decided to start working as an independent data visualization specialist. I am now using **ggplot2** every day: for my academic work, design requests, reproducible reports, educational purposes, and personal data visualization projects. What I especially love about my current job specification: It challenges and satisfies my creativity on different levels. Besides the creativity one can express in terms of chart choice and design, there is also creativity needed to come up with solutions and tricks to bring the most venturous ideas to life. At the same time, there is the gratification when your code works and *magically* translates code snippets to visuals.

The blog entry by Zev Ross was not updated since January 2016, so I decided to add more examples and tricks to my version, which was now hosted on my personal blog[3]. Step by step, my version became a unique tutorial that now contains for example also the fantastic **patchwork**, **ggtext** and **ggforce** packages, a section on custom fonts and color palettes, a collection of R packages tailored to create interactive charts, and several new chart types. The updated version now contains ~3.000 lines of code and 188 plots and received a lot of interest from **ggplot2** users from many different professional fields.

The extensive tutorial served as the starting point for the book you hold in your hands. Plans changed a lot over the time and the book became something unique, that is loosely

[1] http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/
[2] https://github.com/rfordatascience/tidytuesday/blob/master/README.md
[3] https://www.cedricscherer.com/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/

based on the tutorial and a 2-day workshop I was honored to teach at the rstudio::conf(2022) in Washington DC. In addition, I decided to include more advanced tips and insights in the process of creating custom, engaging data visualizations with **ggplot2**. I hope you enjoy it as much as I enjoy learning and sharing ggplot wizardry!

## Why Read This Book

Often, people that use common graphic design and charting tools or have basic experience with **ggplot2** cannot believe what one can achieve with this graphics library—and I want to show you how one can create a publication-ready graphic that goes beyond the traditional scientific scatter or box plot.

**ggplot2** is already used by a large and diverse group of graduates, researchers, and analysts and the current rise of R and the tidyverse will likely lead to an even increasing interest in this great plotting library. While there are many tutorials on **ggplot2** tips and tricks provided by the R community, to my knowledge there is no book that specifically addresses the complete design of specific details up to building an ambitious multipanel graphic with **ggplot2**. As a blend of strong grounding in academic foundations of data visualization and hands-on, practical codes, and implementation material, the book can be used as introductory material as well as a reference for more experienced **ggplot2** practitioners.

The book is intended for students and professionals that are interested in learning **ggplot2** and/or taking their default ggplots to the next level. Thus, the book is potentially interesting for **ggplot2** novices and beginners, but hopefully also helpful and educational for proficient users.

Among other things, the book covers the following:

- Look-up resource for every-day and more specific ggplot adjustments and design options
- Practical hands-on introduction to **ggplot2** to quickly build appealing visualization
- Discussion of best practices in data visualization (e.g. color choice, direct labeling, chart type selection) along the way
- Coverage of useful **ggplot2** extension packages
- Ready-to-start code examples
- Reference implementations illustrating code solutions and design choices

## How to Read This Book

This book can either serve as a textbook or as a reference. Depending on your skill level, some codes and tricks may already be known or not helpful at the moment. In case you want to directly jump to the chapters you find most promising or helpful, here are some suggestions:

- What is the idea of **ggplot2** and how does it actually work? → Chapter 2
- How do I get started with the code? → Chapter 3
- Can I get a quick walk-through of the power of **ggplot2**? → Chapter 4
- How can I modify colors and pick good color palettes? → Chapter 8

- Which ggplot tricks and design decisions are used to create custom graphics? → Chapter 11

## Prerequisites

To run any of the materials locally on your own machine, you will need the following:

- A recent version of R (download from here[4])
- Preferably an *Integrated Development Environment* (IDE) to store scripts and run code, e.g. RStudio (download from here[5]) or Visual Studio Code (download from here[6])
- The following R packages installed:
  - **ggplot2**[7]
  - A set of other **tidyverse** packages:**readr**[8], **tibble**[9], **dplyr**[10], **tidyr**[11], **forcats**[12], **stringr**[13], **lubridate**[14]
  - Some **ggplot2** extension packages:**ggforce**[15], **ggrepel**[16], **ggtext**[17], **patchwork**[18]
  - Packages providing color palettes and functionality:**colorspace**[19], **scico**[20], **rcartocolor**[21]
  - A few other packages providing useful functionality:**magick**[22], **ragg**[23] (includes **systemfonts**[24]), **sf**[25]
- The following typefaces installed on your system:
  - Asap Condensed[26]
  - Spline Sans[27]
  - Spline Sans Mono[28]

To install all packages in one go, run the following code in the R console:

[4] https://cloud.r-project.org/
[5] https://posit.co/download/rstudio-desktop/
[6] https://code.visualstudio.com/download
[7] https://ggplot2.tidyverse.org/
[8] https://readr.tidyverse.org/
[9] https://tibble.tidyverse.org/
[10] https://dplyr.tidyverse.org/
[11] https://tidyr.tidyverse.org/
[12] https://forcats.tidyverse.org/
[13] https://stringr.tidyverse.org/
[14] https://lubriate.tidyverse.org/
[15] https://ggforce.data-imaginist.com/
[16] https://ggrepel.slowkow.com/
[17] https://wilkelab.org/ggtext/
[18] https://patchwork.data-imaginist.com/
[19] https://colorspace.r-forge.r-project.org/
[20] https://github.com/thomasp85/scico
[21] https://jakubnowosad.com/rcartocolor/
[22] https://docs.ropensci.org/magick/
[23] https://ragg.r-lib.org/
[24] https://systemfonts.r-lib.org/
[25] https://r-spatial.github.io/sf/
[26] https://fonts.google.com/specimen/Asap+Condensed
[27] https://fonts.google.com/specimen/Spline+Sans
[28] https://fonts.google.com/specimen/Spline+Sans+Mono

```
install.packages(c(
  "ggplot2", "readr", "tibble", "dplyr", "tidyr", "forcats",
  "stringr", "lubridate", "ggforce", "ggtext", "colorspace", "scico",
  "rcartocolor", "patchwork", "magick", "ragg", "systemfonts", "sf"
))
```

To get a bundle of all typefaces, download the zip file via ADD LINK[29].

## Software Information and Conventions

Package names are in **bold text** and wrapped into curly brackets, e.g. **ggplot2**. Inline code and file names are formatted in a `monospaced typewriter font`. Function names are followed by parentheses as in `ggplot2::ggplot()`.

The book was written with the **knitr** package (Xie, 2015) and the **bookdown** package (Xie, 2023) with the following setup:

```
## R version 4.2.3 (2023-03-15)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Big Sur ... 10.16
##
## Locale: en_US.UTF-8 / en_US.UTF-8 / en_US.UTF-8 / C / en_US.UTF-8 / en_US.UTF-8
##
## Package version:
##   base64enc_0.1.3   bookdown_0.34
##   bslib_0.5.0       cachem_1.0.7
##   cli_3.6.0         compiler_4.2.3
##   cpp11_0.4.3       digest_0.6.31
##   ellipsis_0.3.2    evaluate_0.20
##   fastmap_1.1.1     fs_1.6.1
##   glue_1.6.2        graphics_4.2.3
##   grDevices_4.2.3   highr_0.10
##   htmltools_0.5.4   jquerylib_0.1.4
##   jsonlite_1.8.4    knitr_1.42
##   lifecycle_1.0.3   magrittr_2.0.3
##   memoise_2.0.1     methods_4.2.3
##   mime_0.12         R6_2.5.1
##   ragg_1.2.5        rappdirs_0.3.3
##   rlang_1.1.1       rmarkdown_2.20
##   rstudioapi_0.14   sass_0.4.5
##   stats_4.2.3       stringi_1.7.12
##   stringr_1.5.0     systemfonts_1.0.4
##   textshaping_0.3.6 tinytex_0.44
##   tools_4.2.3       utils_4.2.3
##   vctrs_0.6.2       xfun_0.39
##   yaml_2.3.7
```

29

## Acknowledgements

Thanks to David Grubbs, Alberto Cairo, Emily Riederer, Oscar Baruffa, Malcolm Barrett, and Tanya Shapiro for all your constructive feedback and your patience and willingness to discuss the tiny details with me.

Cédric Scherer
Berlin, Germany

# *About the Author*

Dr Cédric Scherer[30] is a data visualization designer, consultant, and instructor helping clients and workshop participants to create engaging and effective graphics. As a graduated ecologist, he has acquired an extensive hypothesis–driven research experience and problem–solving expertise in data wrangling, statistical analysis, and model development. As an independent data visualization designer, Cédric later combined his expertise in analyzing large data sets with his passion for design, colors and typefaces.

Cédric has designed graphics across all disciplines, purposes, and styles applying a code–first approach and regularly talks about data visualization design and ggplot2 techniques. Due to participation in social data challenges such as #TidyTuesday, he is now well known for complex and visually appealing figures, entirely made with **ggplot2**, that look as if they have been created with a vector design tool. He also uses R and the **tidyverse** packages to automate data analyses and plot generation, following the code-first philosophy of a reproducible workflow.

[30] https://cedricscherer.com

# 1

# *Introduction*

## 1.1 Communicating Data

Communicating data is critical for many of us, no matter if scientists, journalists, or analysts. How we present data affects the engagement of and interpretation by the audience. Showing data in an honest, meaningful—and maybe sometimes even playful or artistic—way is the art of ***data visualization*** or ***information visualization***. Data visualization can be described as the transformation of numbers into visual quantities, encoded by forms, positions, and colors. The transformation allows us to see patterns and trends in data and identify relationships between different variables. In the best case, a well-designed data visualization fulfills one or more of the following: amplify cognition, facilitate insight, discover patterns, spark curiosity, explain trends, and make decisions.

Data visualizations, or broadly speaking ***information graphics***, are often classified as being either exploratory or explanatory. ***Exploratory graphics*** are generated to understand the data and search for the relevant information. ***Explanatory graphics*** aim to communicate the derived information between people (Koponen and Hildén, 2019). In contrast to exploratory graphics, the creation of engaging explanatory graphics involves not only the display of data but also requires many choices with regard to the storytelling and design.

When designing visualizations myself or looking at the work of others, the most important question to me is the ***purpose*** of the graphic. Without a clear understanding of the purpose, it is impossible to design an effective and engaging visualization. The same applies when evaluating a visualization: without the consideration of the purpose—the audience, the message, the mood—the designer had in mind when creating the visualization, the critique of design choices often becomes obsolete. A common assumption is that the single aim of data visualizations is to guide decisions. This might be true for business or scientific applications that aim for precision and accuracy by creating *pragmatic visualizations* (Kosara, 2007).

At the same time, it is ignorant to assume that efficiency and functionality are the main purpose of every visualization. Many of the great visualizations we have seen and that stick to our mind go beyond the precise, informative display of data[1]. They experiment with new approaches, use clever, unusual ways to tell stories or were designed simply to transport joy, curiosity or concern. In some cases, the design and visual novelty may even be the main focus with the aim to create a novel, artistic experience for the viewer. Such artworks are not necessarily created to maximize discovery or communication but to elicit emotions and can be termed ***affective graphics***[2].

As a *creator*, clearly defining the purpose of a visualization helps to make decisions about

[1] However, I am not saying that these are the only ones that are great—there are definitely several magnificent pragmatic visualizations that come to my mind!

[2] Credit to the term 'affective graphics' goes to Alberto Cairo, thank you for sharing your thoughts with me.

the data, the chart type, and the design. As a *reader* or *viewer*, identifying the purpose helps rating the quality of the presentation. Some people like to think that there is a single best approach to visualize data: the one that has survived the test of time and is the most efficient to quantify information. Some believe that a chart has to be designed in a 'neutral' way. I strongly disagree with both opinions, for multiple reasons. The most important: Every time we present the data, we make decisions; and it is not about *if* we make decisions but *which*. Also, the tool being used comes with default decisions that may impact the final look—because you don't want, care or simply can't change some settings. Chart types are not inherently 'right' or 'wrong' but might be more or less suitable for the purpose. Colors are associated with some emotional value—how could we pick one that has a 'neutral' meaning, association or emotion for every person that might look at our visualization?

Even if we agree on the 'right' decisions—the best chart type and a neutral color encoding, likely some shades of grey—we still can't ensure that all people interpret it in the same way. People will always find their own message in graphs and the interpretation will likely differ based on individual differences through culture, attitude and mood.

A quote from Alberto Cairo that is close to my heart sums it up brilliantly:

---

> Visualizations can be designed and experienced in various ways, by people of various backgrounds, and in various circumstances. That's why reflecting on the purpose of a visualization is paramount before we design it—or before we critique it. (Cairo, 2021)

---

In the optimal case, the decisions made by the creator are based on some thoughtful consideration of the following:

- *data* — which information is meaningful and robust?
- *audience* — what do readers already know?
- *context* — how will the reader encounter the visualization?
- *story* — what is the main message of the visualization?
- *goal* — which chart type is suitable to transport the story?
- *design* — how can I facilitate engagement and understanding?

While some decisions might (and should) be made before crafting the visualization, the creation of purposeful, well-designed graphics is an iterative process. Rarely[3] the first draft is what ends up being printed on physical material or being displayed on your computer or smartphone screen. Nowadays, computational approaches ease the cyclic process of prototyping, exploring, testing, and designing the 'best' visual encoding of information for a given purpose.

[WIP] Include basic business graphic versus artsy + catchy viz -> maybe astronauts?

[3] I was very tempted to write "never" but I don't have data to support this claim…

## 1.2 Coding Visualizations

As data visualizations involve the quantitative representation of variables, a setup that allows to handle, wrangle, analyse, ***and*** visualize data in the same environment is beneficial. Classical design software is great to create vector-based graphics of all kinds but must often be paired with a 'visualization tool' if the data and/or the chart type becomes more complex. While there are many tools that allow to quickly create specific, predefined chart types (e.g. DataWrapper, Flourish, RAWgraphs), often also with beautiful and very sensible defaults, such chart builders usually do not provide full flexibility. In addition, updating the data can become a time-consuming and painful task as one has to repeat the same *point-and-click* procedure over and over again.

By using a computational, code-driven approach we can combine all steps related to data visualization in the same environment: from the data import and cleaning to the precise and flexible encoding of quantitative information with custom designs. Programming languages such as JavaScript, Python, or R have a much steeper learning curve than chart builders but at the same time allow users to create almost any visualization one can think of. Also, thanks to the script-based approach, updating the underlying data is a simple task and can even be automated. Furthermore, they come with several *extension libraries* (e.g. D3.js, echarts, Vega, Matplotlib, ggplot2) that provide additional approaches or add more opportunities to existing code.

Data visualizations that are generated with code have several other benefits. The *reproducibility* of code makes the process more efficient by being able to update the data or to use the code as a template for future projects. The *transparency* of coded (and well-documented) data workflows increases trust. The *scalability* of code allows to produce the graphics for multiple data sets and use cases.

Of course, the visualization does not need to be created by code alone. Switching from a code-based approach to a vector-graphics tool makes a lot of sense in use cases where reproducibility does not matter or graphics are stand-alone artworks. Honestly, in terms of efficiency and freedom, a combined approach is likely the best approach in such a case.

With that in mind, knowing how to code visualizations is likely beneficial in any data-related field.

## 1.3 Why R and ggplot2

As a computational ecologist, I've learned and used a range of different tools and programming languages for various purposes such as data wrangling, statistical analyses, and model building. The open-source language R was and is the programming language most widely used by ecologists to handle and analyze ecological data (Sciaini et al., 2018). Consequently, I was *of course* using R in my daily life as a scientific researcher.

Nowadays, R plays a crucial part in many data-related workflows, no matter if for scientific, educational, or industrial use cases. Thanks to the ever growing R community and the rich collection of libraries that add additional functionality and simplify workflows, R is an attractive programming language that has outgrown of its original purpose: statistical

analyses. Today, R can serve as tool to generate automated reports, develop stand-alone web apps, and draft presentation slides, books, and web pages. And to design high-level, publication–ready visualizations.

Even though R—similar to most programming languages—has a steep learning curve, the level of functionality, flexibility, automation, and reproducibility offered can be a major benefit also in a design context:

- The layered approach of **ggplot2** opens the possibility to build any type of visualization. Even though other tools using similar approaches are becoming popular, **ggplot2** is still the most mature and flexible out there.
- Learning a programming language can be tough. R often feels "easy to learn" for non-programers.
- Various extension packages add missing functionality.
- Script-based workflows instead of *point–and–click* approaches allow for reproducibility—which means you can run the code again after receiving new data or create thousands of visualizations for various data sets in no time.
- Sharing code is becoming the golden standard in many fields and thus facilitates transparency and credibility as well as reuse and creative advancement.
- A helpful community and many free resources simplify learning experiences and the search for solutions.
- The visualizations created in R can be exported as vector files and thus allow for post–processing with vector graphics software like Adobe Illustrator, Inkscape or Figma.

The following collection of graphics illustrate the power and versatility of **ggplot2**—and a range of extension packages—to create customized, partly uncommon or complex charts. All these charts are the outcome of 100% R code, making use of multiple layers and customized colors, fonts, and themes.

These are just a few examples of the many types of charts that you can create using **ggplot2**. With a little creativity and experimentation, you can come up with your own unique and informative visualizations or artful pieces.

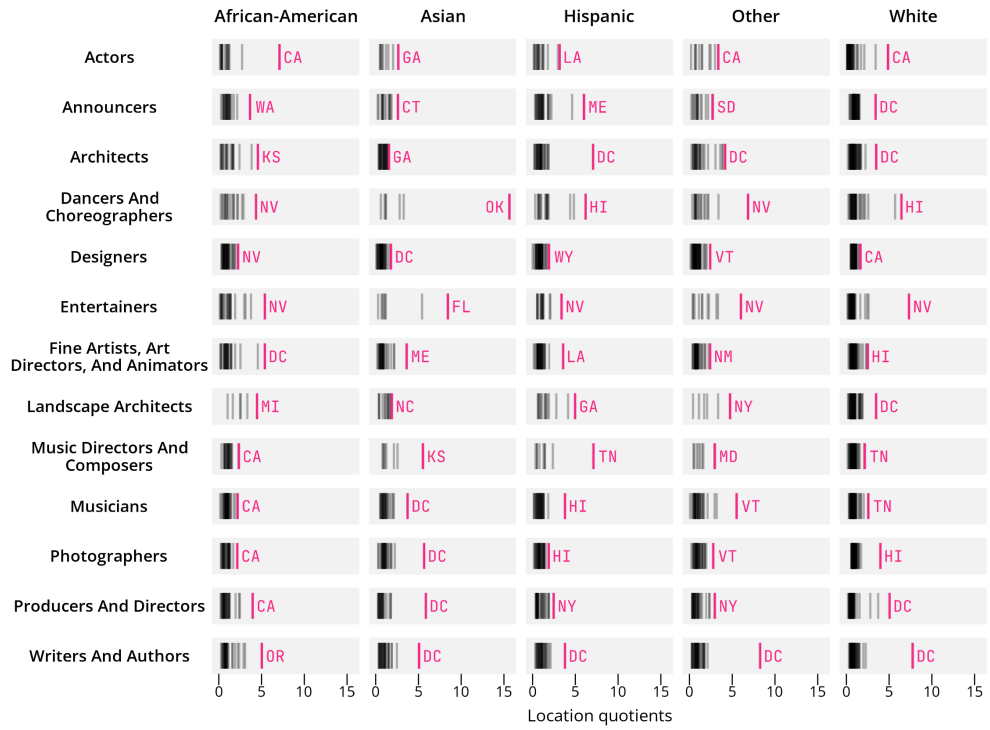



**FIGURE 1.1:** "Artists in the USA" by Lee Olney

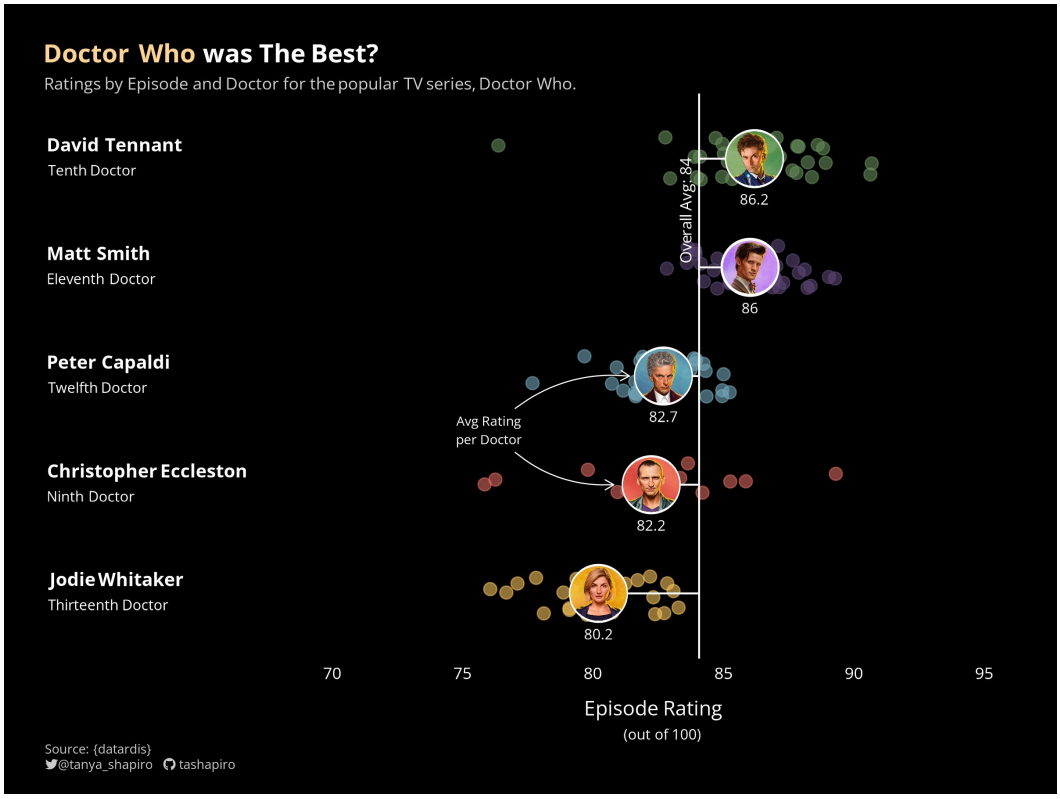



**FIGURE 1.2:** "Doctor Who was the Best?" by Tanya Shapiro

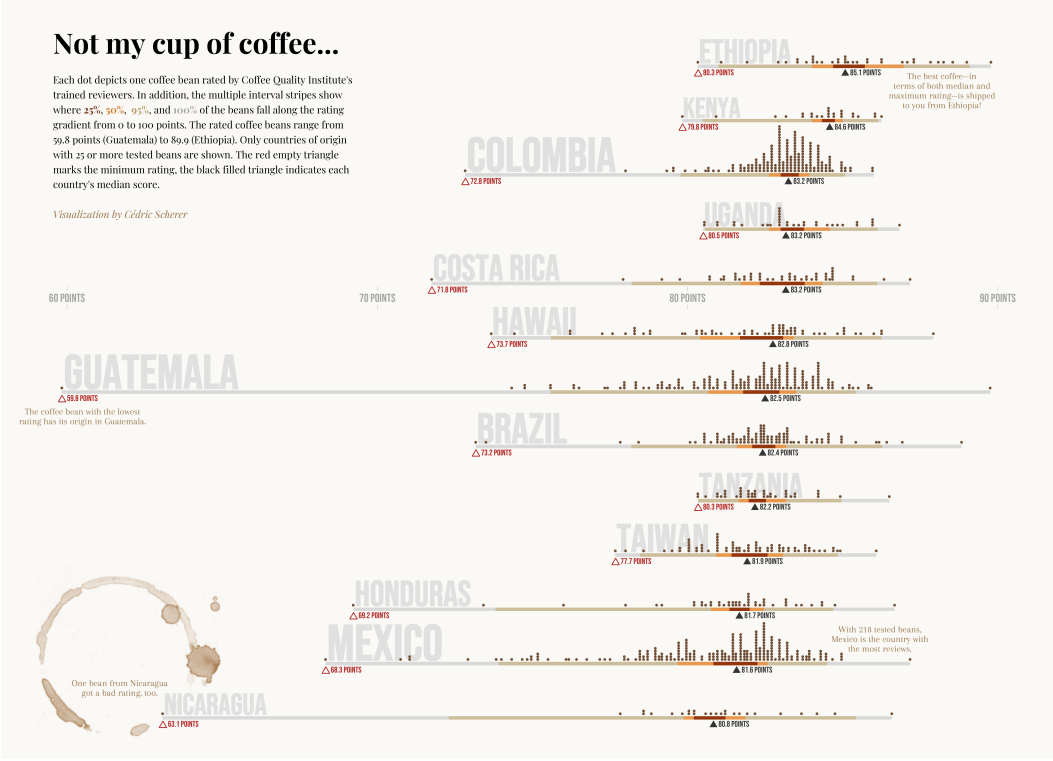



**FIGURE 1.3:** "Not My Cup of Coffee" by myself

# Part I

# General Overview

# 2

# *The Layered Grammar of Graphics*

"The Grammar of Graphics" is a general concept for data visualization, proposed by Leland Wilkinson in 2005 (Wilkinson, 2005). Based on the earlier work "Semiology of Graphics" by Jacques Bertin (Bertin, 1983), Wilkinson's "grammar" provides a structured vocabulary for designing and interpreting graphics. Graphics are treated as a language, comprising a set of fundamental components that can be combined and modified to convey insights clearly and coherently.

The grammar of graphics is founded on a layered approach, that define data visualizations as semantic components rather than as predefined chart types. In his view, a graphic is constructed through discrete layers of visual elements, allowing practitioners to create a wide range of visualizations while maintaining a coherent and principled approach. These layers include data, aesthetic mappings, geometric objects, statistical transformations, and scales.

Each layer contributes to the final visualization, enabling the separation of concerns and promoting modularity. *Data* forms the foundation, representing the information to be visualized, while *aesthetic mappings* define how data attributes map to visual properties like size, color, and shape. *Geometric objects* encompass the graphical shapes employed, ranging from points and lines to bars and polygons; *statistical transformations* offer the ability to summarize or transform data, enhancing the analytical power of visualizations. Finally, *scales* help translate data values into perceptually meaningful visual attributes.

## 2.1 The ggplot2 Package

In 2005 Hadley Wickham implemented Leland Wilkinson's "The Grammar of Graphics" as an R package called **ggplot2** (Wickham, 2016). As in the theoretical concept, **ggplot2** treats data visualizations as a layered combination of components rather than as predefined chart types.

Wickham created the package "as a response to the shortcomings of the current plotting systems" in R (Wickham (2006)), with the idea to allow building up a graphic from multiple layers of data while handling (shared) axes, colors and other attributes consistently and automatically. The ability to control and combine multiple components makes it a powerful approach to compose complex graphs, iterate quickly over different visual data representations, and modify existing plots.

The package was initially released on June 10, 2007 and has since then become one of the most popular R packages and the standard for producing custom, high-quality graphics in R.

The predecessor, the original **ggplot** package[1], was released in 2006 and is made available out of historical interest.

When looking into the package description of the **ggplot2** package, it states the following:

---

> **ggplot2** is a system for declaratively creating graphics, based on The Grammar of Graphics. You provide the data, tell **ggplot2** how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

---

The most important insight from this technical description is that

1. *we map variables to aesthetics*, i.e. defining the visual channels used to represent the variables (e.g. position, color, shape)
2. *we use graphical primitives*, i.e. defining one or multiple forms to represent the variables (e.g. lines, points, rectangles)

Both are important when writing **ggplot2** code and together with the provided data they are the key components of a ggplot. Additional components allow to control the visual appearance, the layout, and the coordinate system.

## 2.2 The Components of a ggplot

In general, a ggplot is built up from the following components:

1. ***Data***:
   The raw data that you want to plot.
2. ***Aesthetics***:
   The mapping of variables to visual properties, such as position, color, size, shape, and transparency.
3. ***Layers:***
   The representation of the data on the plot panel which is a combination of the *geometric shapes* representing the data and the *statistical transformation* of the data, such as fitted curves, counts, and data summaries.
4. ***Scales***:
   The control of the mapping between the data and the aesthetic dimensions, such as data range to positional aesthetics or qualitative or quantitative values to colors.
5. ***Coordinate system***:
   The transformation used for mapping data coordinates into the plane of the graphic.
6. ***Facets***:

[1] https://github.com/hadley/ggplot1

The arrangement of the data into a grid of plots (also known as *trellis* or *lattice plot*, or simply *small multiples*).

7. ***Visual themes***:
   The overall visual (non-data) details of a plot, such as background, grid lines, axes, typefaces, sizes, and colors.

The number of elements may vary depending on how you group them and whom you ask. This list is based on the list provided in the "ggplot2" book by Hadley Wickham (Wickham, 2016).

A basic ggplot needs three key components that you have to specify: the *data*, *aesthetics*, and a *layer*. All other additional components can be further modified to customize your graphic.

You can think of a ggplot as a receipt for a dish: it can be based on a few or a diversity of ingredients. Also, you are free to add additional ingredients to spice-up your creation (literally and visually).

Similarly, you can build rather basic charts such as scatter plots, histograms, box-and-whisker plots, or area charts with only a few lines code. But **ggplot2** also allows to create rather complex charts that combine multiple geometries, statistical transformations and maybe even data sets. On top, it is up to you how much effort you take to polish the plot.

You can rely on the defaults used for data-related aesthetics and non-data aspects (as in Fig. 2.1). Or you decide to modify the data-related aesthetics such as axes and color palettes and/or customize the theme elements of your graphic to your needs (as in Fig. 2.2). Or you might even choose a storytelling approach and add additional annotations such as direct labels, shapes, and other highlights to provide context and make a graphic more informative and easier to read (as in Fig. 2.3).

## 2.3 Key Components

### 2.3.1 Data

Without data, there is no data visualization. Luckily, there are many sources of data available to us: statistics, surveys, experiments, and observations. The data may be collected by governments, researcher labs and organisations, companies—or yourself. However, it is important to consider the quality and context of the data you choose in order to gain accurate and valuable insights.

The ***quality*** of the data we use will have a direct impact on the validity and usefulness of the insights we gain from our data visualization. Poor quality data can lead to incorrect conclusions, while high quality data can provide valuable insights and help us make informed decisions.

In addition to the quality of the data, it is also important to consider the ***context*** in which the data was collected. Different data sources may have different biases or limitations, and it is important to consider these when interpreting and visualizing the data.

Usually, data visualization should be based on real data. At the same time, it is of course possible to create visualizations using hypothetical or simulated data to train yourself or experiment with new chart types. However, you should always keep in mind the origin of

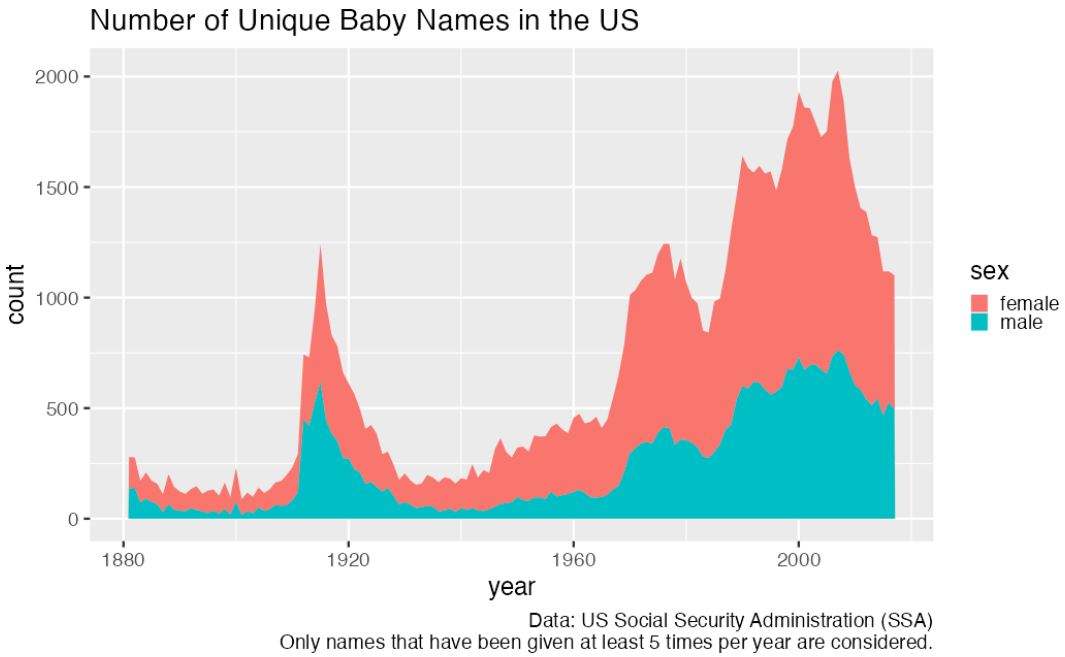



**FIGURE 2.1:** A stacked area graph of the number of unique babynames per year and sex between 1881 and 2017 as an example for a basic ggplot, relying on the defaults for all components.

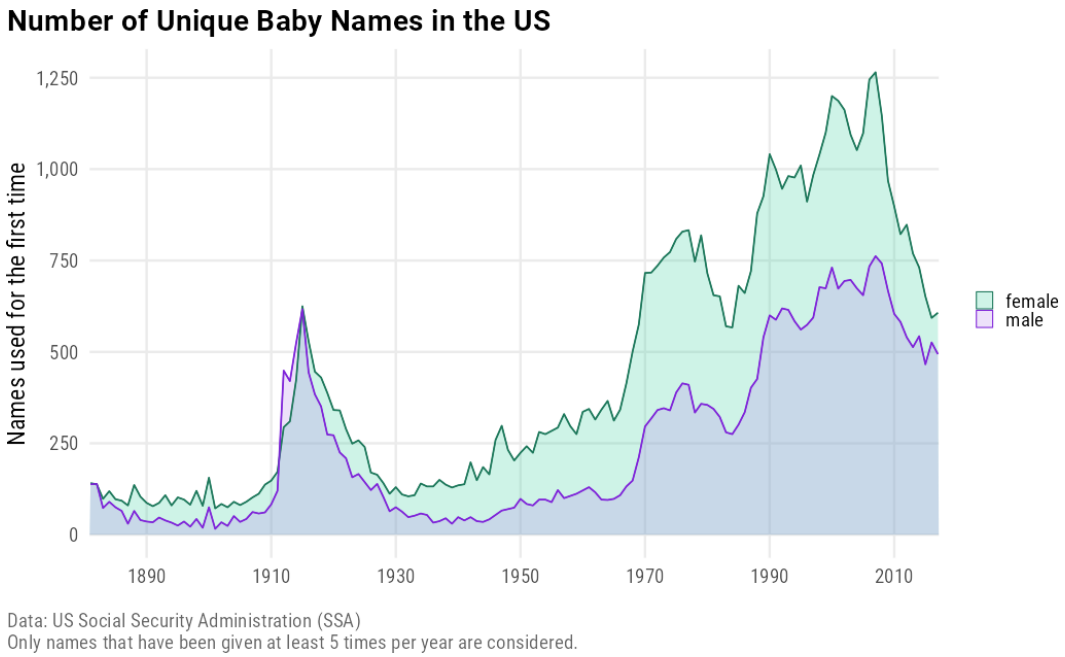



**FIGURE 2.2:** In contrast to Fig. 2.1, the visual theming, scales (axes styling and color palette), and the behavior of the layer have been customized. Besides the different "feel" of the graphic, the chart now focusses on comparing the two sexes due to the common baseline, and not on the overall numbers as in Fig. 2.1.

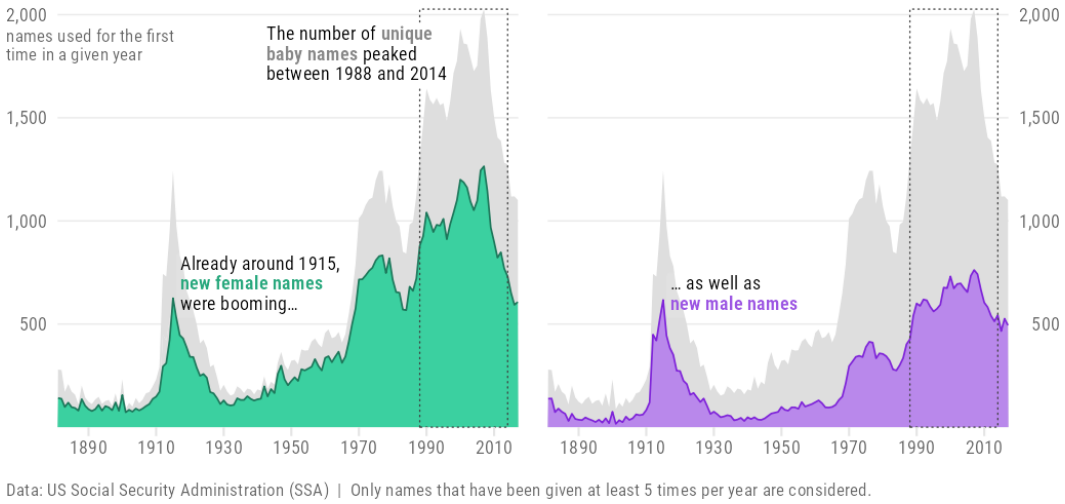



**FIGURE 2.3:** A facetted version of the area charts in Fig. 2.1 and 2.2 using a storytelling-approach by adding annotations (e.g. direct labels, boxes, color hightlights, and overall counts) to provide context and guide the viewer.

the data and communicate the fact clearly to your audience to avoid misleading insights. In order to truly understand and learn from data, we need to work with real, accurate, and reliable data.

Depending on how you want to display your data in **ggplot2**, you have to prepare the data in different formats. The general recommendation is to use a "long format" or "tidy format". In a tidy-form data set, each variable is stored in a column while rows form single observations (2.4). With such a data set, we can display each variable using a different visual channel, the *aesthetics*, such as position, color and shape (see 2.5 A). Consequently, data in true "long format" (i.e. the variable is specified in a dedicated row) is only useful in case you want to display the variables using the same visual channel (see 2.5 B). A wide format, as you might often find it in case of governmental data, often needs some reshaping except the goal is the representation of a single combination.

If you need to reshape your data, the `pivot_*()` functions from the **tidyr** package are handy. Use `pivot_longer()` to convert a wide data set into the long or tidy format. To go the other direction, use the `pivot_wider()`. You can find an example in chapter 3.2.5.2.

### 2.3.2 Aesthetics

To visualize certain variables in your data set with **ggplot2**, values are mapped to visual channels called *aesthetics*. Aesthetic attributes include positional information such as x and y but also colors, fills, point shapes, line types, sizes, and levels of transparency.

Sticking to our small data set, we could use our tidy-form data (Fig. 2.4, top right) to create a scatter plot of the two numeric metrics with the categorical group mapped to color and

**Long format:**

– rows are single measurements
– all measurements are stored in a single column

| group | year | metric | value |
|---|---|---|---|
| A | 2022 | x | 46 |
| B | 2022 | x | 2 |
| C | 2022 | x | 21 |
| A | 2023 | x | 32 |
| B | 2023 | x | 16 |
| C | 2023 | x | 7 |
| A | 2022 | y | 12 |
| B | 2022 | y | 35 |
| C | 2022 | y | 24 |
| A | 2023 | y | 1 |
| B | 2023 | y | 42 |
| C | 2023 | y | 27 |

**Tidy format:**

– rows are observations, columns are variables
– all values are cell inputs

| group | year | metric_x | metric_y |
|---|---|---|---|
| A | 2022 | 10 | 32 |
| B | 2022 | 2 | 35 |
| C | 2022 | 13 | 10 |
| A | 2023 | 12 | 43 |
| B | 2023 | 16 | 42 |
| C | 2023 | 7 | 27 |

**Wide format:**

– variables and/or observations are spread across multiple columns
– some values are encoded in the column names

| group | metric_x_22 | metric_y_22 | metric_x_23 | metric_y_23 |
|---|---|---|---|---|
| A | 46 | 12 | 32 | 1 |
| B | 2 | 35 | 16 | 42 |
| C | 21 | 24 | 7 | 27 |

**FIGURE 2.4:** A comparison of data arranged in a long (left) versus wide formats (right). The two different metrics by color. Groups are additionally encoded by shaded rows.

year mapped to shape (Fig. 2.5 A). However, we could also use the long-form data (Fig. 2.4, top left) to show the metrics as a group wise dot plot and encode the metrics by color (Fig. 2.5 B).

The optimal shape of your data set relates to the plot you have in mind. Stick to the rule that any variable that you want to use for an aesthetic should have a dedicated column.

### 2.3.3 Layers

Layers in **ggplot2** define the the statistical transformation, geometrical representation, and positional adjustment of the mapped values. In the example above, we have used a layer with geometry "point" and without any statistical transformation or positional adjustment. This specification simply means "draw the raw values as points by using the specified aesthetics of position, color, and shape".

Statistical transformation are responsible for calculating summaries such as counts and averages. You can also perform more advanced transformations of the data such as calculating

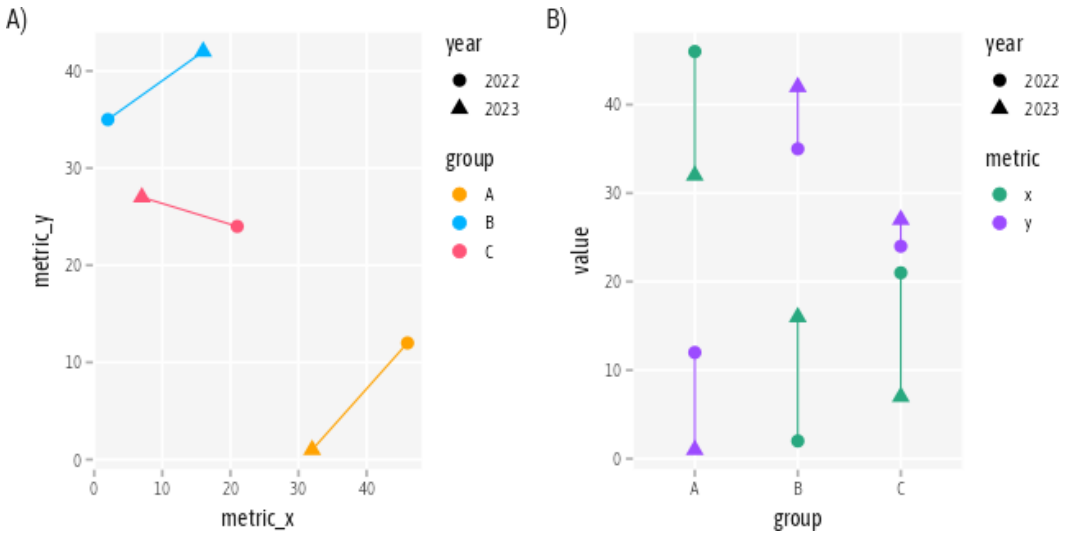



**FIGURE 2.5:** Basic ggplot outputs mapping four different variables (columns of the data set) to aesthetics, using the long-format data (`data_long`, left plot) and wide-format data (`data_wide`, right plot). Lines connecting the groups were added to make the difference between both plots more obvious.

smoothings and densities. If there is any statistical transformation specified, the calculation is applied to the data before they are plotted.

The raw or transformed data are then used to draw the specified geometrical object(s), representing the parsed data e.g. as points, bars, or lines. On top, you can also adjust the position of the geometries. Examples for positional adjustments are the grouping or stacking of bars or the jittering of points.

For example, you might use a statistical transformation to calculate linear regression lines to fit them to your variables mapped to x and y and display them as banded lines (geometrical object). Or you might decide to show the raw data as points without any statistical transformation (Fig. 2.6).

In general, each layer in a ggplot is created using a separate function call. For each layer, you can specify the visual appearance, such as color, and size, independently by *setting properties* (e.g. turn all points green) or *mapping aesthetics* (e.g. base the color on the group variable). This allows you to build up a complex plot by adding and customizing individual layers, giving you fine-grained control over the appearance of your plots.

The layered approach allows to create a wide range of charts and graphics. One can also create more complex, potentially unusual graphics as it allows you to combine layers in a traditional but also nontraditional way as you like.

## 2.4 Additional Components

The following components are set by default but can be tweaked to:

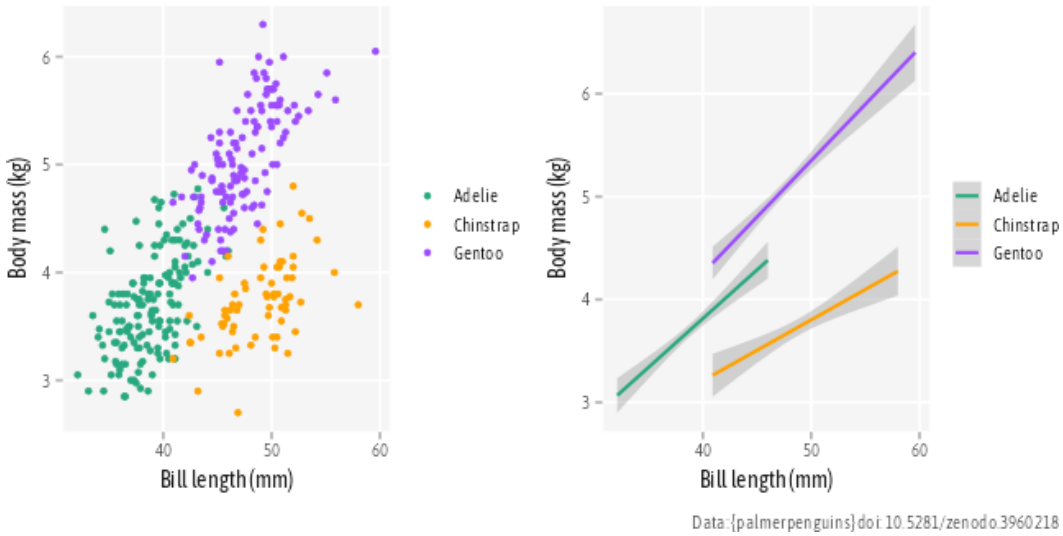



**FIGURE 2.6:** The same data, visualized as a scatter plot showing the raw data without any statistical transformation (left) and after a statistical tranformation has been applied to calculate linear fittings for each Penguin species (right). The visualizations use the Palmer Archipelago penguin data by A.M. Horst, A.P. Hill & K.B. Gorman (2020).

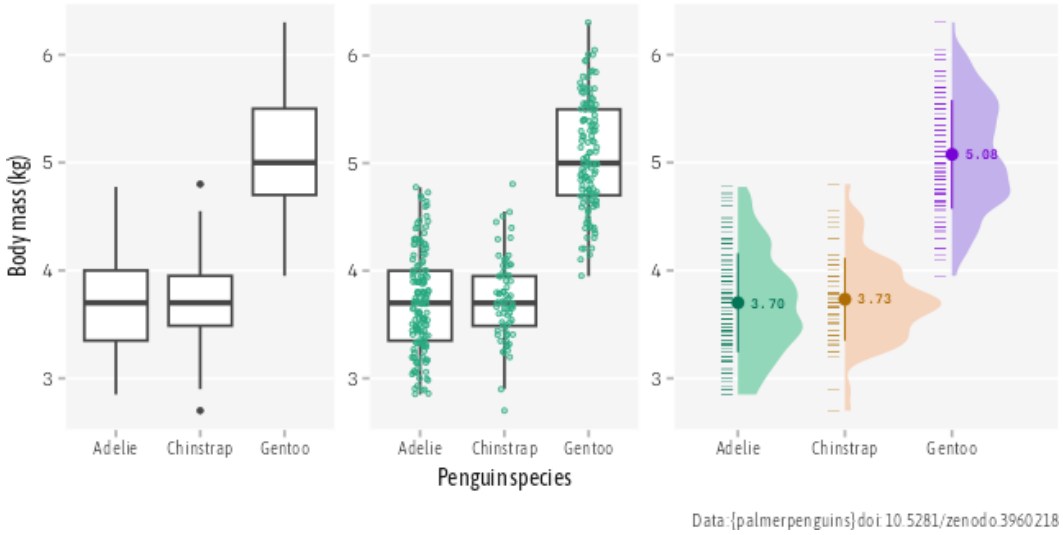



**FIGURE 2.7:** Three different visualizations showing the distribution of body mass across three penguin species. A) A box-and-whiskers plot using a single layer. B) Adding a second layer to plot A allows to show the raw data as jittered points. C) By combining multiple layers, one can build more complex visualizations like this variant of a raincloud plot. Four layers are used here: one for the density curve, one for the pointrange, another one for the barcode strip and finally one for the annotation with the mean values. The visualizations use the Palmer Archipelago penguin data by A.M. Horst, A.P. Hill & K.B. Gorman (2020).

- adjust the properties of the aesthetics (*scales*)
- control the mapping of positional aesthetics (*coordinate systems*)
- create small multiples of the specified chart (*facets*)
- modify non-data related elements (*themes*)

### 2.4.1 Scales

Scales translate between the value range of our data, mapped to aesthetics, and the perceptual property range. Every time you map a column to an aesthetic, a respective suitable scale component is added to your plot.

To modify the default settings you can specify your own scale, for example tweak the number of axis ticks or customize the colors used to encode groups. Furthermore, scales are also used to transform the data before it is processed by the layer. An example of transforming the values with a scale component is the display of the data in discrete bins.

### 2.4.2 Coordinate Systems

Coordinate systems interpret the position aesthetics. By default, a Cartesian coordinate system is used to encode the x and y aesthetics. As this is likely the most common type in data visualization, you might modify coordinate systems only in two cases: creating circular plots such as pie charts and circular barplots or when projecting spatial data.

### 2.4.3 Facets

Facets split variables to multiple panels, allowing to create the same visualization for several combinations. Such small multiples can be a powerful tool to show high-dimensional data, to explore big data sets, and to compare patterns across variables and groups.

A special case of facets are geo-referenced small multiples: a set of visualizations is laid out in a grid that represents the original topography. Such graphics can be easily created in **ggplot2** with the help of the **geofacet** extension package.

### 2.4.4 Themes

Themes encode all non-data elements of your plots such as the typeface, background colors, and titles and captions. **ggplot2** comes with a set of complete themes that can be added to your plot and further modified with high flexibility.

The modified themes can easily be turned into your own custom theme function which can be used as a component in the same way. Also, several extension packages such as **ggthemes**, **hrbrthemes** or **tvthemes** provide even more complete themes to change the look of your visualization.

The ability to use pre-coded themes is a great feature as it allows for a consistent and corporate style while saving time to write the same code over and over again.

# 3

# *Get Started*

## 3.1 The Data Set

We are using historical data for bike sharing in London in 2015 and 2016, provided by *TfL (Transport for London)*[1]. The data was collected from the TfL data base and is 'Powered by TfL Open Data'. The processed data set contains hourly information on the number of rented bikes and was combined with weather data acquired from freemeteo.com. The data was contributed to the Kaggle online community[2] by Hristo Mavrodiev.

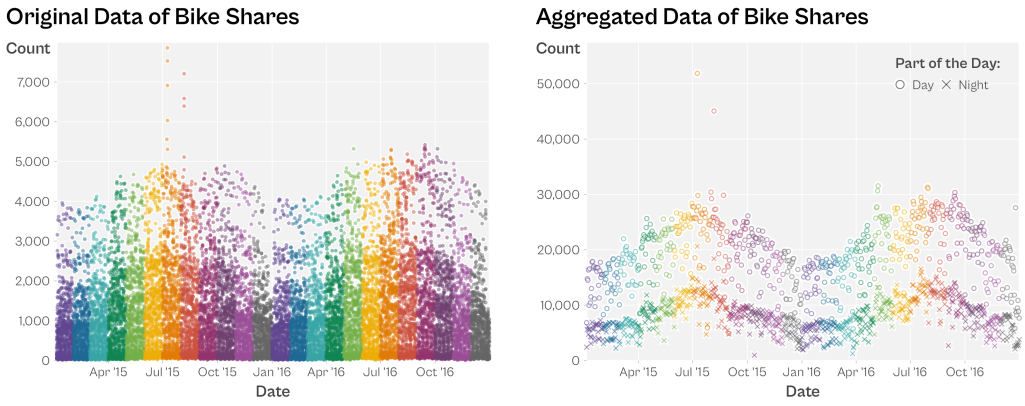



**FIGURE 3.1:** The original and aggregated data sets in direct comparison: counts of bike shares registered by TfL over time with month encoded by colour. The left panel shows counts for every hour of the day, while in the right panel the hourly data was aggregated into two periods of the day (day and night).

To make the visualizations manageable and patterns more insightful, we are using a modified data set with all variables aggregated for day (6:00am–5:59pm) and night (6:00pm–5:59am) (3.1). The bike counts were summarized while all weather-related variables where averaged. Finally, for the weather type, the most common was used and, in case of a tie, one of the most common types was randomly chosen. The modified data sets contains 14 variables (columns) with 1,454 observations (rows). To give you a better idea what the data set contains, a visual overview of the variables is provided in table 3.1 and figure 3.2.

[1] https://tfl.gov.uk/modes/cycling/santander-cycles
[2] https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset

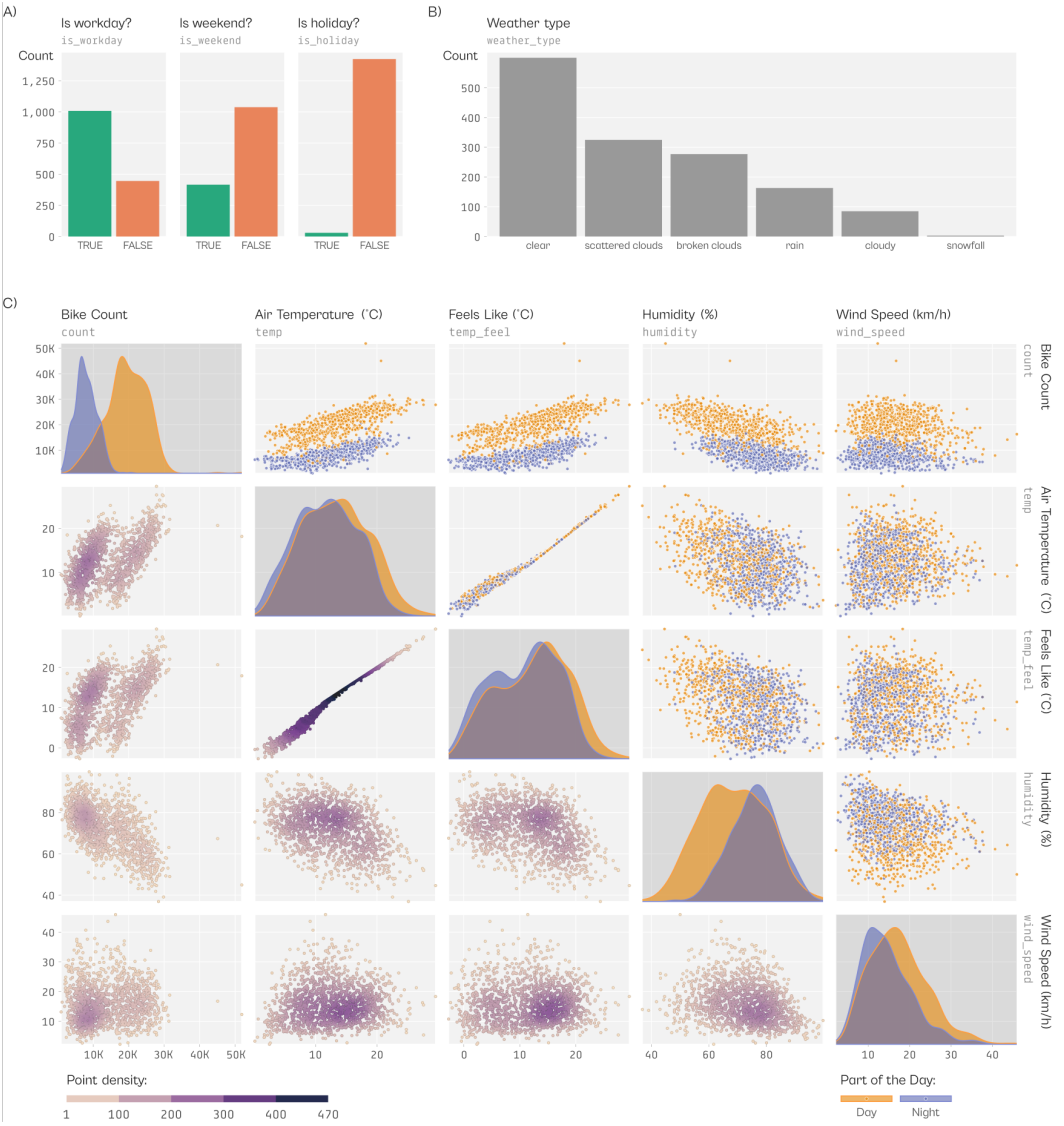



**FIGURE 3.2:** Overview of the distribution of the boolean variables `is_workday`, `is_weekend`, and `is_holiday` (A), the categorical variable `weather_type` (B), and the continuous variables `count`, `temp`, `temp_feel`, `humidity`, and `wind_speed` (C) of the cleaned and aggregated bike sharing data set. In panel C, the correlation between the variables is shown as scatterplot encoded by `timeperiod` (upper triangle) and encoded by point density (lower triangle), highlighting the level of overlap of data points.

**TABLE 3.1:** Overview of the 15 variables contained in the cleaned and aggregated bike sharing data set.

| Variable | Description |
|---|---|
| 'date' | Date encoded as 'YYYY-MM-DD' |
| 'day_night' | 'day' (6:00am–5:59pm) or 'night' (6:00pm–5:59am) |
| 'year' | '2015' or '2016' |
| 'month' | '1' (January) to '12' (December) |
| 'season' | '0' (spring), '1' (summer), '2' (autumn) or '3' (winter) |
| 'count' | Sum of bikes rented |
| 'is_workday' | 'TRUE' being Monday to Friday and no official holiday |
| 'is_weekend' | 'TRUE' being Saturday or Sunday |
| 'is_holiday' | 'TRUE' being an official holiday in the UK |
| 'temp' | Average air temperature (°C) |
| 'temp_feel' | Average feels like temperature (°C) |
| 'humidity' | Average air humidity (%) |
| 'wind_speed' | Average wind speed (km/h) |
| 'weather_type' | Most common weather typed |

## 3.2 Working in R

**ggplot2** can be used even if you know little about the R programming language. However, the knowledge of certain basic principles is at least helpful and probably indispensable for advanced plots. This section will give you a short overview of workflows and the very basics needed. The overview makes use of the **tidyverse**, a package collection designed for data science in R. However, multiple other options exist to import, inspect, and wrangle your data if you prefer not to work with the **tidyverse** for these steps[3].

### 3.2.1 Import Data

You need to import data to be able to work with it in the current session. The data can be imported from a local directory or directly from a web source. Nowadays, all common and some less common data formats can easily be imported. For common tabular data formats such as .txt or .csv one can use the **readr** package[4] (Wickham et al., 2022b) from the **tidyverse**.

We use the `read_csv()` function to load the TfL data as .csv file directly from a web URL. To access the URL and data later, we are storing the link as `url_data` and the data set as `bikes` by using the *assignment arrow* `<-`. The `col_types` argument within the `read_csv()` function allows to specify the column types. For example, `i` represents integer values, `f` encodes factors, and `l` turns a column into a logical, boolean variable that only can have two states, `TRUE` or `FALSE`. You'll find more on the different *data types* later in this chapter.

[3] Note that the **ggplot2** package itself belongs to the **tidyverse** as well.
[4] https://readr.tidyverse.org/

```
url_data <- "https://cedricscherer.com/data/london-bikes.csv"
bikes <- readr::read_csv(file = url_data, col_types = "Dcfffillddddfc")
```

The `::` is called "namespace" and can be used to access a function without loading the package. Here, you could also run `library(readr)` first and `bikes <- read_csv(url_data)` afterwards.

If you want to load data that is stored locally, you specify the path to the file instead:

```
path_data <- "C://path/to/my/data/london-bikes.csv" ## mocked-up name for Win users
bikes <- readr::read_csv(file = path_data, col_types = "Dcfffillddddfc")
```

Note that the syntax of the path differs between operating systems. While in Windows subdirectories are separated with backslashes, Mac and Linux uses slashes. Also, absolute paths differ in their syntax: Windows machines specify the drive letter, here`C://`, or the server name; on Mac/Linux machines absolute paths start with `/users/`.

Instead of relying on the absolute path, which likely differs between users and operating systems, you can also use relative paths. Those are specified with a leading dot, e.g. as `./data/london-bikes.csv`, and look for the file relative from the working directory. You can retrieve and change your working directory with `getwd()` and `setwd()`. However, relying on the default working directory or setting a custom one will likely cause problems the moment someone else wants to run your code[5].

### 3.2.2 Project-Oriented Workflows

Preferably, we do not want to rely on absolute paths and the default or manually set working directories. So-called project-oriented workflows aim to organize each piece of work in a self-contained, bundled directory. This directory includes all relevant files needed for the project, such as scripts and images. By ensuring that paths are set relative and in a way that they are understood by any operating system, we guarantee that the project will run on any machine and can easily moved around.

If you are working in Rstudio, *Rstudio projects* provide such a project-oriented workflow. When opening the project via the associated `.Rproj` file, it ensures that the working directory is correctly set and points to the project's top-level directory (i.e. the folder that contains the `.Rproj` file).

When using Rstudio projects, a helpful package to navigate to files of interest is the **here** package[7]. The function `here()` will create paths relative to the top-level directory:

```
## point to the csv inside the "data" subdirectory of the project directory
path_data <- here::here("data", "london-bikes.csv")
```

[5] A detailed reasoning why you should not use `setwd()` is provided in Chapter 2 of "What they forgot you to teach about R"[6].

[7] https://here.r-lib.org/

### 3.2.3 Inspect the Data

After importing the data, it is advisable to have a look at the data. Does the object stored in R match the dimensions of your original data file? Are the variables displayed correctly? You can print the data by simply running the name of the object, here `bikes`.

```
bikes
```

```
## # A tibble: 1,454 x 14
##    date       day_night year  month season count
##    <date>     <chr>     <fct> <fct> <fct>  <int>
##  1 2015-01-04 day       2015  1     3       6830
##  2 2015-01-04 night     2015  1     3       2404
##  3 2015-01-05 day       2015  1     3      14763
##  4 2015-01-05 night     2015  1     3       5609
##  5 2015-01-06 day       2015  1     3      14501
##  6 2015-01-06 night     2015  1     3       6112
##  7 2015-01-07 day       2015  1     3      16358
##  8 2015-01-07 night     2015  1     3       4706
##  9 2015-01-08 day       2015  1     3       9971
## 10 2015-01-08 night     2015  1     3       5630
## # i 1,444 more rows
## # i 8 more variables: is_workday <lgl>,
## #   is_weekend <lgl>, is_holiday <lgl>, temp <dbl>,
## #   temp_feel <dbl>, humidity <dbl>, wind_speed <dbl>,
## #   weather_type <fct>
```

As we have used the **readr** package, our data is stored as a *tibble* (class `tbl_df` and related) which is the **tidyverse** subclass of a traditional data frame (class `data.frame`). There are other data structures in R such as lists and matrices; however, in this book we are going to use only data frames, more precisely tibbles (besides spatial data formats in Chapter XYZ).

On the top of the output, you can directly see that our data set consists of `format(length(bikes), big.mark = ",")` variables (columns) frame with 1,454 observations (rows). Also, the tibble output shows you the first ten rows. Alternatively you can inspect the data with the help of `str()` or `tibble::glimpse()` to print a transposed version.

### 3.2.4 Data Types

If you have looked carefully, you may have noticed that a tibble prints also the data type of each column, e.g. `<chr>`. In our case, we have specified the types of the columns manually when importing the data; if not specified, `readr::read_csv()` as most other import functions will guess the data type for each column based on the first x observations.

The data encoding is especially important when exploring chart options and writing ggplot code. You should be familiar if the data is encoded as quantitative or qualitative, if it contains missing or unusual values. Thus, it is always worth to check the classes and the values of the columns.

In R there are multiple low-level data types, and some of the **ggplot2** behavior will depend on the type of the variable(s) used for plotting. The six basic data types are:

- character

- factor
- numeric
- integer
- logical
- complex

To examine the data type of an object, use the `class()` function. The `$` symbol allows you to access single columns of a data frame, e.g. `bikes$day_night`:

```
class(bikes$day_night)
## [1] "character"

class(bikes$weather_type)
## [1] "factor"

class(bikes$temp)
## [1] "numeric"

class(bikes$count)
## [1] "integer"

class(bikes$is_weekend)
## [1] "logical"
```

Variables of type `character`, `factor`, and `logical` will be handled as **categorical, qualitative data** while `numeric`, `integer`, and `complex`[8] are treated as **numerical, quantitative data**.

The most important difference between the categorical types is their sorting: while `character` and also `logical` values are sorted alphabetically, `factor` variables have a specified order, defined by the so-called *levels*. Note that numbers can be treated as categorical as well and thus we can change their intrinsic order, too. We can inspect the order with the `levels()`:

```
levels(bikes$weather_type) ## an example of a factor with strings
## [1] "broken clouds"    "clear"
## [3] "cloudy"           "scattered clouds"
## [5] "rain"             "snowfall"

levels(bikes$season) ## an example of a factor with numbers
## [1] "3" "0" "1" "2"
```

Also, we can manually change the order of a factor by supplying a vector of level names to the `factor()` function:

```
bikes$season_mix <- factor(bikes$season, levels = c("2", "0", "3", "1"))
levels(bikes$season_mix)
## [1] "2" "0" "3" "1"
```

[8] As variables of type `complex` are rarely used in a data visualization context, we are going to ignore this data type.

The **forcats** package[9] (Wickham, 2022) provides a set of useful functions to reorder factor levels. Important functions to quickly change to order of variables include `fct_rev()`, `fct_reorder()`, `fct_infreq()`, `fct_inorder()`, and `fct_lump()`:

```
bikes$weather_type_rev <- forcats::fct_rev(bikes$weather_type)
levels(bikes$weather_type_rev)
## [1] "snowfall"        "rain"
## [3] "scattered clouds" "cloudy"
## [5] "clear"           "broken clouds"

bikes$weather_type_rank <- forcats::fct_infreq(bikes$weather_type)
levels(bikes$weather_type_rank)
## [1] "clear"           "scattered clouds"
## [3] "broken clouds"   "rain"
## [5] "cloudy"          "snowfall"
```

The single difference between the two numerical types `numeric` and `integer` is simply-speaking the existence of floating point numbers: while `numeric` variables store decimals, `integer` variables are stored as whole numbers. If you want to force integer values, you can specify them as `1L`.

```
head(bikes$temp) ## numeric
## [1] 2.167 2.792 8.958 7.125 9.000 6.708

as.integer(head(bikes$temp)) ## integer
## [1] 2 2 8 7 9 6
```

As shown in the last command, we can also convert data types. Be careful though, as R has internal rules how to *coerce* one data type into another. The same also happens if you specify a vector of multiple data types. In the following example, the `integer`, `numeric`, and `logical` values are coerced to `character` values. Afterwards, we explore the coercion behavior by converting the vector to other data types:

```
my_vector <- c(1L, 0.2, "ggplot", FALSE)
my_vector
## [1] "1"      "0.2"    "ggplot" "FALSE"

class(my_vector)
## [1] "character"

as.numeric(my_vector)
## [1] 1.0 0.2  NA  NA

as.integer(my_vector)
## [1]  1  0 NA NA

as.factor(my_vector)
```

[9] https://forcats.tidyverse.org/

```
## [1] 1      0.2    ggplot FALSE
## Levels: 0.2 1 FALSE ggplot

as.logical(my_vector)
## [1]    NA    NA    NA FALSE
```

When changing the data type of our `character` vector, some values are successfully converted (e.g. `1L` and `0.2` as `numeric` or `FALSE` as `logical`) while some are "wrongly" interpreted (e.g. `0.2` is converted into `0L` as `integer` or `FALSE` into `"FALSE"` as `character`). Some others are stored as `NA` representing *unknown, not available values* (e.g. `"ggplot"` as `numeric` or `1` as `logical`). Conversion between different data types is very useful but be careful and always inspect the output of the conversion for potential coercion mistakes.

Another important data type is the `Date` class. Dates are either represented as the number of days since a specified origin or converted `character` type with a structure of YYYY-MM-DD:

```
class(bikes$date)
## [1] "Date"

as.Date(1, origin = "1970-01-01")
## [1] "1970-01-02"

as.Date("2007-06-10")
## [1] "2007-06-10"

as.Date("2022-09-16") - as.Date("2007-06-10")
## Time difference of 5577 days
```

Furthermore, `POSIXct` and `POSIXlt` are able to represent full time stamps including a date and time. The two POSIX date/time classes only differ in the way that the values are stored internally.

```
as.POSIXct("2007-06-10 12:34:56")
## [1] "2007-06-10 12:34:56 CEST"

as.POSIXlt("2007-06-10 12:34:56")
## [1] "2007-06-10 12:34:56 CEST"
```

### 3.2.5 Data Preparation

Often, the data imported might not be in the right format to plot it with **ggplot2**. The preparation of the data—e.g. converting data types and setting factor levels, aggregating values, estimating data summaries, reshaping the data set or combining it with another source—is called *data wrangling*, *data munging*, or **data manipulation**.

To explore and retrieve summary estimates of individual variables, the following functions are useful:

- `min()`, `max()`, `range()` to extract extremes of numerical data

- `quantile()` to get an idea of the distribution numerical data
- `unique()` to get all unique entries, helpful for categorical data
- `length(unique())` to count all unique entries

```
min(bikes$temp) ## add na.rm = TRUE in case it returns `NA`
## [1] 0.125

range(bikes$date)
## [1] "2015-01-04" "2016-12-31"

quantile(bikes$count)
##    0%   25%   50%   75%  100%
##   953  7508 11965 19412 51870

unique(bikes$day_night)
## [1] "day"   "night"

length(unique(bikes$weather_type))
## [1] 6
```

The **dplyr**[10] (Wickham et al., 2022a) and other **tidyverse** packages provide a simple and intuitive syntax to wrangle data. There are a ton of unique functions, but knowing a handful is enough to empower you to bring your data in the right format.

#### 3.2.5.1 Data Wrangling with the dplyr Package

There are five main functions of **dplyr**, called *verbs*:

- `filter()`: Pick rows with matching criteria
- `select()`: Pick columns with matching criteria
- `arrange()`: Reorder rows
- `mutate()`: Create new variables
- `summarize()` or `summarise()`: Sum up variables

All of these functions follow a consistent syntax: `verb(data, condition)`. Note that we specify the data and columns individually and that within the tidyverse, column names are referred to without quotation marks:

```
## load the package functions
library(dplyr)

## only keep more than 1000 shares during the night
filter(bikes, day_night == "night", count > 1000)

## only keep 4 columns and reorder those
select(bikes, date, count, weather_type, temp)

## order by day_night and decreasing bike shares
arrange(bikes, day_night, -count)
```

[10] https://dplyr.tidyverse.org/

```
## add a column with temperature encoded as °F
mutate(bikes, temp_fahrenheit = temp * 1.8 + 32)

## calculate the mean count across all observations
summarize(bikes, count_avg = mean(count))
```

Another important and very powerful function from the **dplyr** package is `group_by()`: when a data set is grouped into subsets, we can apply any operation for each group *within a single data frame*. While you could nest those functions, the common approach within the tidyverse is the use of *pipes*. Pipes take the output of one function and send it directly to the next which avoids nested operations and allows for a more logical order of functions and their arguments. The **tidyverse** pipe is encoded as `|>`; a base R pipe is available as `|>` as well. Have a look at the following silly example:

```
## nested functions, read inside out + disconnected function inputs
go_to_work(
  breakfast(
    wake_up(
      Cedric, alarm = "06:30")
    ),
    c("coffee", "croissant")
  ),
  mode = "bus", delay = 10
)

## piped version with "Cedric" passed to the next function call
Cedric |>
  wake_up(alarm = "06:30")) |>
  breakfast(c("coffee", "croissant") |>
  go_to_work(mode = "bus", delay = 10)
```

Let's create the workflow for some serious data preparation: In a first step, we are going to estimate the average temperature for night rents per year and season:

```
bikes_summarized <-
  bikes |>
  ## only keep night observations
  filter(day_night == "night") |>
  ## create 8 subsets (4 seasons x 2 years)
  group_by(season, year) |>
  ## calculate total counts and mean temperature per subgroup
  summarize(
    count = sum(count),
    temp_avg = mean(temp)
  )

bikes_summarized
```

```
## # A tibble: 8 x 4
```

```
## # Groups:   season [4]
##   season year    count temp_avg
##   <fct>  <fct>   <int>    <dbl>
## 1 3      2015   459469     7.65
## 2 3      2016   489136     6.91
## 3 0      2015   693703     9.93
## 4 0      2016   676427     9.43
## 5 1      2015  1020524    17.2
## 6 1      2016  1044592    17.7
## 7 2      2015   685534    12.4
## 8 2      2016   744827    12.3
```

After filtering out "day" cases, the grouping by `season` and `year` allows us to summarize counts and average temperatures for each of the eight groups. Note that the resulting tibble is still grouped by `season` (as shown in the output with the `[4]` explicitly stating the number of current subsets).

In a next step, we regroup our data to add a column with shares of nightly bike rents per season on a yearly basis. Afterwards, we remove all subsets by calling `ungroup()` and clean our data set by reordering and removing columns and sorting rows by year and season:

```
bikes_summarized |>
  ## add column with relative shares per year
  group_by(year) |>
  mutate(share_year = count / sum(count)) |>
  ## remove grouping
  ungroup() |>
  ## reorder columns and remove days column
  select(year, season, count, share_year, temp_avg) |>
  ## sort by year and season
  arrange(year, season)
```

```
## # A tibble: 8 x 5
##   year  season   count share_year temp_avg
##   <fct> <fct>    <int>      <dbl>    <dbl>
## 1 2015  3       459469      0.161     7.65
## 2 2015  0       693703      0.243     9.93
## 3 2015  1      1020524      0.357    17.2
## 4 2015  2       685534      0.240    12.4
## 5 2016  3       489136      0.166     6.91
## 6 2016  0       676427      0.229     9.43
## 7 2016  1      1044592      0.354    17.7
## 8 2016  2       744827      0.252    12.3
```

#### 3.2.5.2 Reshaping Data with the tidyr Package

In case you need to reshape a data set to make it work with **ggplot2**, you can use the functions `pivot_longer()` and `pivot_wider()` from the **tidyr** package[11] (Wickham and Girlich, 2022) to move from one format to the other. Let's illustrate their behavior using one of the toy data sets used in section

[11] https://tidyr.tidyverse.org/

```
## create long-format data as showcased in section 2
data_long <- tibble::tibble(
  group = rep(c("A", "B", "C"), 4),
  year = rep(rep(c("2022", "2023"), each = 3), 2),
  metric = c(rep("x", 6), rep("y", 6)),
  value = c(46, 2, 21, 32, 16, 7, 12, 35, 24, 1, 42, 27)
)

## print long-format data
head(data_long, 5)
```

```
## # A tibble: 5 x 4
##   group year  metric value
##   <chr> <chr> <chr>  <dbl>
## 1 A     2022  x         46
## 2 B     2022  x          2
## 3 C     2022  x         21
## 4 A     2023  x         32
## 5 B     2023  x         16
```

```
## turn long-format data into wide-format data
data_wide <- tidyr::pivot_wider(
  data = data_long,        ## the long data set
  names_from = metric,     ## new column names
  values_from = value,     ## values to be filled in
  names_prefix = "metric_" ## adjust new column names
)

## print wide-format data
data_wide
```

```
## # A tibble: 6 x 4
##   group year  metric_x metric_y
##   <chr> <chr>    <dbl>    <dbl>
## 1 A     2022        46       12
## 2 B     2022         2       35
## 3 C     2022        21       24
## 4 A     2023        32        1
## 5 B     2023        16       42
## 6 C     2023         7       27
```

```
## turn wide-format data back to long-format data
data_long_again <- tidyr::pivot_longer(
  data = data_wide,             ## the wide data set
  cols = c(metric_x, metric_y), ## columns to pivot
  names_to = "metric",          ## column to hold variable
  values_to = "value",          ## column to hold values
  names_prefix = "metric_"      ## adjust new variable names
)
```

[WIP] lubridate, stringr, ...

# 4

# *A Walk-Through Example*

To illustrate the utility and flexibility of **ggplot2** to create complex, well-designed visualizations with a handful of components, we are going to create a set of four plots (so-called small multiples). For each combination of year and time of the day, we draw a scatter plot of bike rents including a linear fitting for each subset.

The creation of the plot itself will require only four lines of code! One to specify the data and global aesthetics, two for the scatter and fitting, and another one to create small multiples.

With a few more lines of code, we will adjust the labels, add annotations, and apply a custom color palette. Finally, we apply a personalized theme using custom typefaces.

## 4.1 Prerequisites

Before we can create a ggplot, we have to load the package by running `library(ggplot2)`. Also, we need import the data set we want to visualize, here our bike share data introduced in Section 3.1 which we store in an object called `bikes`.

```
## load the ggplot2 package
library(ggplot2)

## import bikes data set
url_data <- "https://cedricscherer.com/data/london-bikes.csv"
bikes <- readr::read_csv(file = url_data, col_types = "Dcfffilllddddfc")
```

## 4.2 Create a Basic ggplot

Once the **ggplot2** package is loaded, we can create a basic ggplot by specifying the *data*, *aesthetics*—the positional encoding of the variables—, and a *geometry*.

Here, we map the variable `temp_feel` of our `bikes` data object to the `x` position and the variable `count` to the `y` position (Fig. 4.1). We will also map variables to all kind of other aesthetics throughout the book. Some are related to positions such as `xmin` or `yend` while others change the appearance of the layer based on the variables they are mapped to such as `color` and `shape`.

There are many, many different geometries (often called geoms because each function starts

with `geom_`) one can add to a ggplot by default and even more provided by extension packages. By adding `geom_point()` we create a scatter plot:

```
ggplot(data = bikes) +              ## initial call + data
  aes(x = temp_feel, y = count) +   ## aesthetics
  geom_point()                      ## geometric layer
```

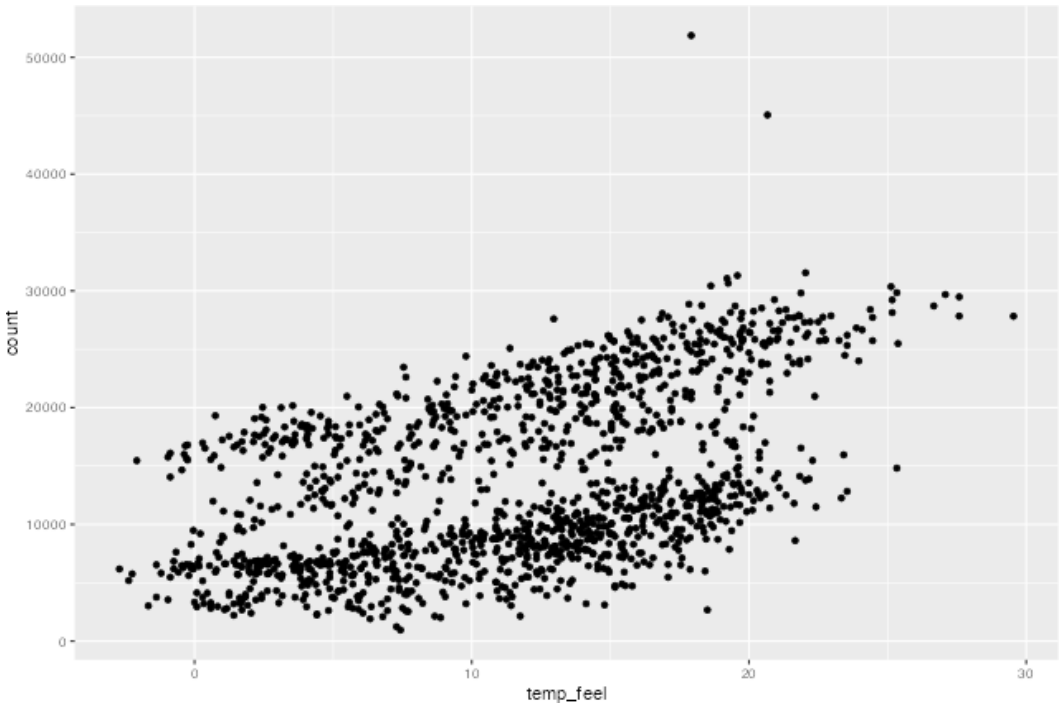



**FIGURE 4.1:** A basic scatter plot of feels-like temperature and reported TfL bike rents, created with the ggplot2 package.

In most cases, you will find ggplot code in which the aesthetics are supplied inside the `ggplot()` call; however, both versions are valid.

```
ggplot(data = bikes, mapping = aes(x = temp_feel, y = count)) +
  geom_point()
```

Due to so-called *implicit matching*, we can rewrite the first part as `ggplot(bikes, aes(date, count))`, omitting the argument names `data` and `mapping` in the `ggplot()` call and `x` and `y` in the `aes()` function. This works as long as we respect the defined order.

Omitting the arguments `data` and `mapping` saves you a ton of typing when creating dozens to hundreds ggplots per day. In my opinion, it is good practice to refer to aesthetics explicitly, and I will follow this convention throughout the book.

## 4.3 Combine Multiple Layers

One can also add several layers, specified as either geometric shapes starting with `geom_*()` or statistical transformations starting with `stat_*()` (Fig. 4.2)—and this is where the magic and fun starts!

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point() +
  ## add a GAM smoothing
  stat_smooth()
```

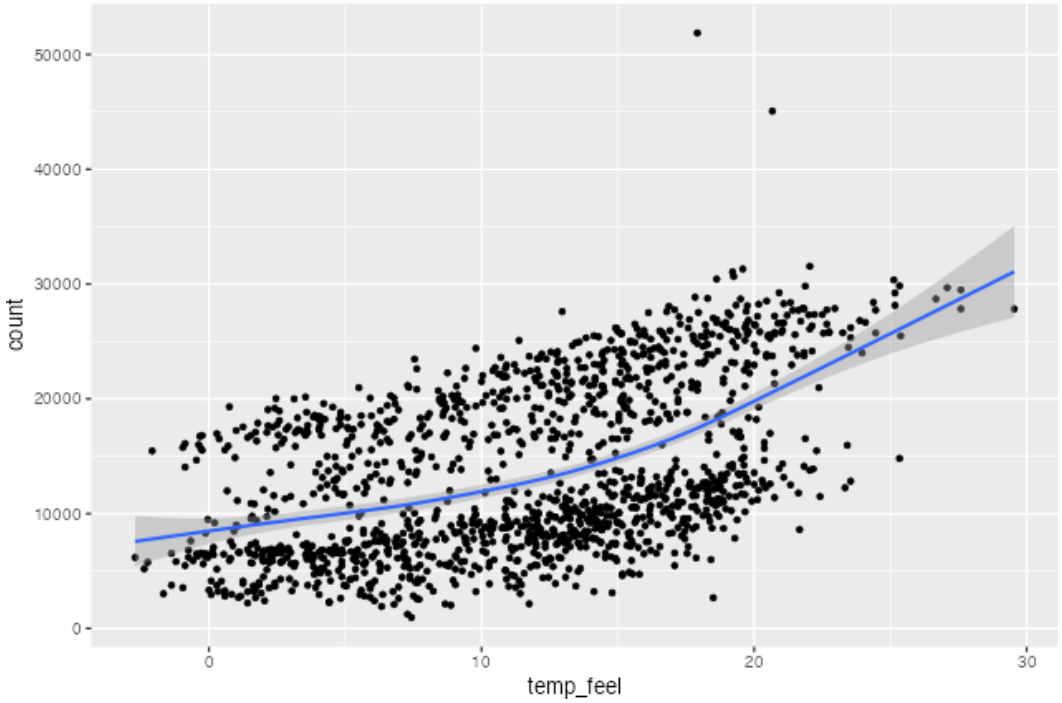



**FIGURE 4.2:** The same scatter plot, now with an additional GAM smoothing.

Both `geom_*()` and `stat_*()` internally make use of the same function `layer()` and pass default inputs for the `geom`, `stat`, and `position` arguments. Basically the use of geoms versus stats is only a personal preference: when specifying `geom_*()` we focus on the shape representing the variables; using `stat_*()` highlights the transformation applied to the variables. More information on the different geometries is provided in Chapter XYZ and a deep-dive into the power of statistical transformation is Chapter XYZ.

## 4.4 Mapping Aesthetics

By looking at these scatter plots, we can actually identify two different trends for day and night. We can highlight both groups by mapping the `day_night` variable to `color`. Note that the mapping is applied to both, `geom_point()` and `geom_smooth()` and consequently the latter layer creates two separate smoothings (Fig. 4.3).

```
ggplot(bikes, aes(x = temp_feel, y = count, color = day_night)) +
  geom_point() +
  stat_smooth()
```


**FIGURE 4.3:** The points and smoothing lines, grouped and colored by time of the day.

Aesthetics can also be defined for each layer and are then applied locally to the respective geometry or statistical transformation only. The `group` aesthetics allows to create subsets without changing the visual appearance (in contrast to aesthetics such as `color` or `shape`; Fig. 4.4)).

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  ## color mapping only applied to points
  geom_point(aes(color = day_night)) +
  ## invisible grouping to create two trend lines
  stat_smooth(aes(group = day_night))
```

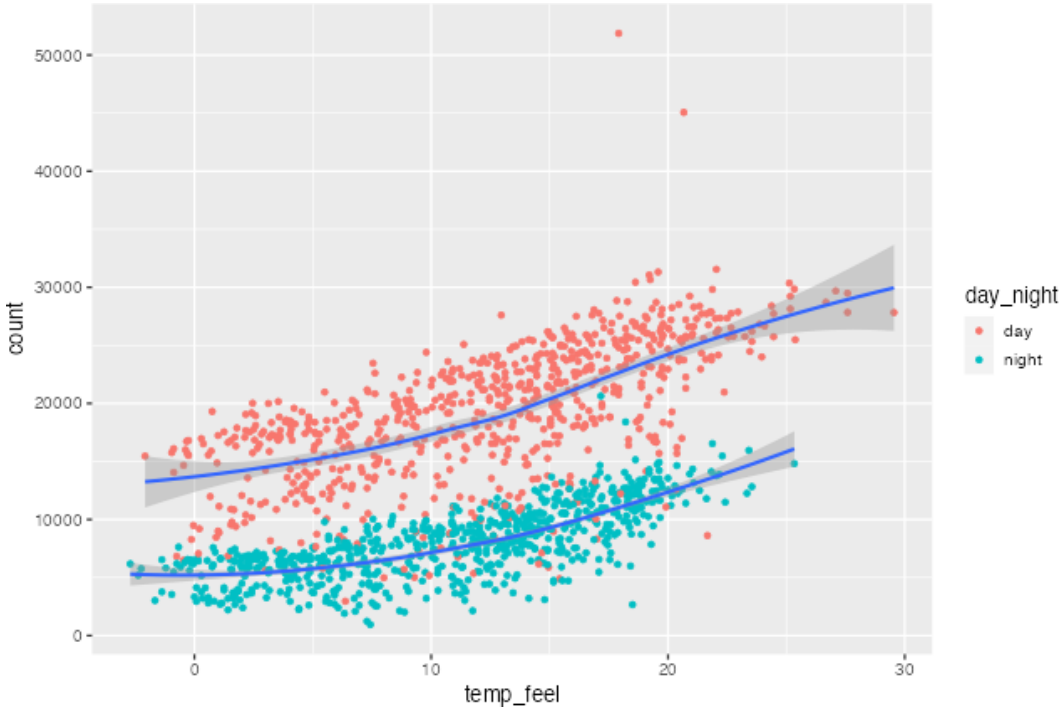



**FIGURE 4.4:** The color applied to the points only with an additional group mapping to draw smoothing lines for both groups individually.

You will learn more how to work with global and local aesthetics in Chapter XYZ and how to modify their appearance in Chapter XYZ.

## 4.5 Setting Properties

Furthermore, each layer has its own arguments to change their behavior and appearance. Let's also add some transparency to the points and turn the smoothing into a linear fitting (Fig. 4.5). As we are not mapping aesthetics but setting properties, we have to place those adjustments outside the `aes()` call.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    aes(color = day_night),
    ## setting larger points with 50% opacity
    alpha = .5, size = 2
  ) +
  stat_smooth(
    aes(group = day_night),
    ## use linear fitting + draw black smoothing lines
```

```
    method = "lm", color = "black"
  )
```

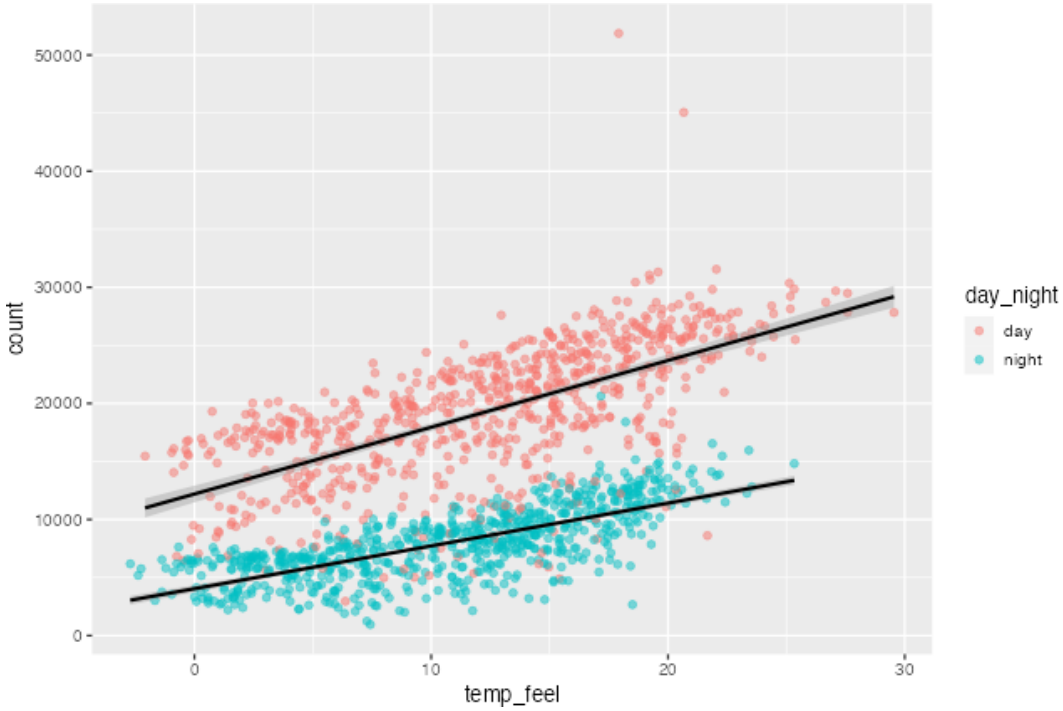



**FIGURE 4.5:** We can map aesthetics and define properties for each layer individually.

In general follow the rule to set constant properties outside `aes()` and map variables to aesthetics inside `aes()`.

The different aesthetics and the differences of global versus local mapping of aesthetics is explained in Chapter XYZ.

## 4.6 Create Small Multiples

A fantastic feature of **ggplot2** is it's ability to quickly split a single visualization in a set of so-called *small multiples*: the same visualization, no matter how complex it is, is applied to subsets contained in the same data set. In **ggplot2**, such small multiples are called *facets* wich are conditional on the variables defined in the `facet_*()` function (Fig. 4.6).

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    aes(color = day_night), alpha = .5, size = 2
  ) +
  stat_smooth(
```

```
    aes(group = day_night), method = "lm", color = "black"
  ) +
  ## small multiples
  facet_grid(
    ## time of the day (rows) vs year (columns)
    day_night ~ year
  )
```

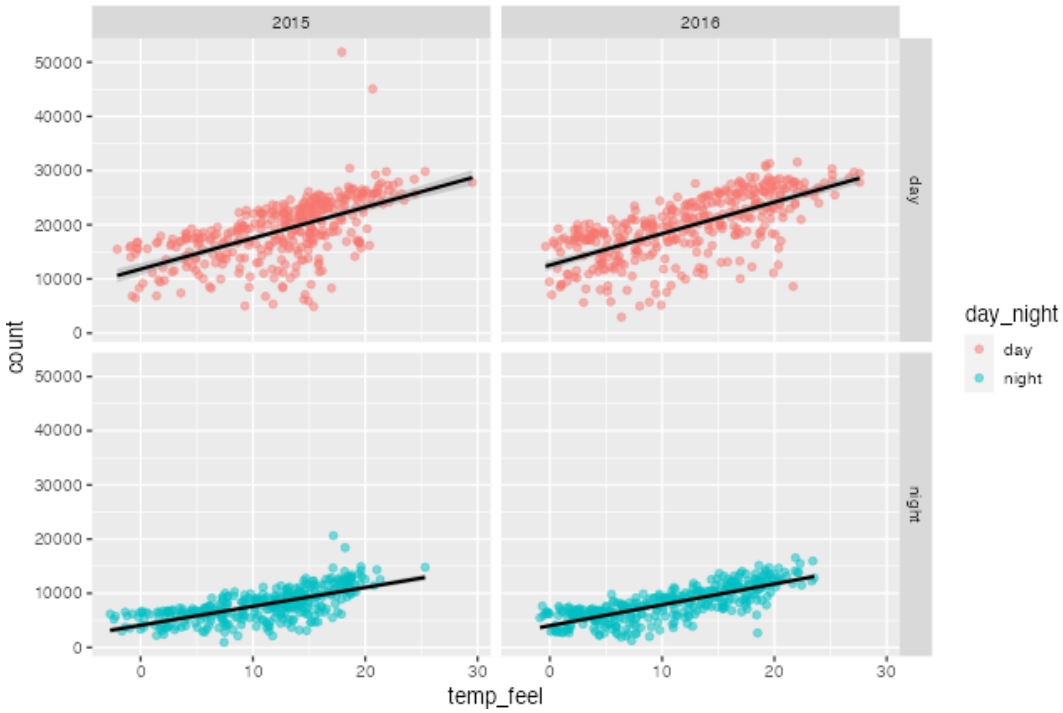



**FIGURE 4.6:** With the `facet` functions, a visualization can quickly be split into small multiples.

Consistency across axes is considered good practice as varying axis ranges are harder to compare and the potential of misleading viewers increases in case the differences keep unnoticed—or even worse are not noticeable. The implementation of facet in **ggplot2** ensures consistency across all small multiples by default, as illustrated by the empty space in the lower row as there are no values above ~21,000 reported rents during night.

The `facet` functionality also comes with the option to overwrite the default behavior by "freeing" the positional scales. Furthermore, one can ensure equal axis spacing by freeing the space as well (Fig. 4.7). This setting allows to efficiently use the available space while ensuring comparability across plots and decreasing the potential of misleading the viewer. The differences between the two ways to create small multiples, namely `facet_grid()` and `facet_wrap()`, as well as ways to adjust and annotate facets are explained in Chapter XYZ.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
```

```
    aes(color = day_night), alpha = .5, size = 2
  ) +
  stat_smooth(
    aes(group = day_night), method = "lm", color = "black"
  ) +
  facet_grid(
    day_night ~ year,
    ## free y axis range
    scales = "free_y",
    ## scale heights proportionally
    space = "free_y"
  )
```

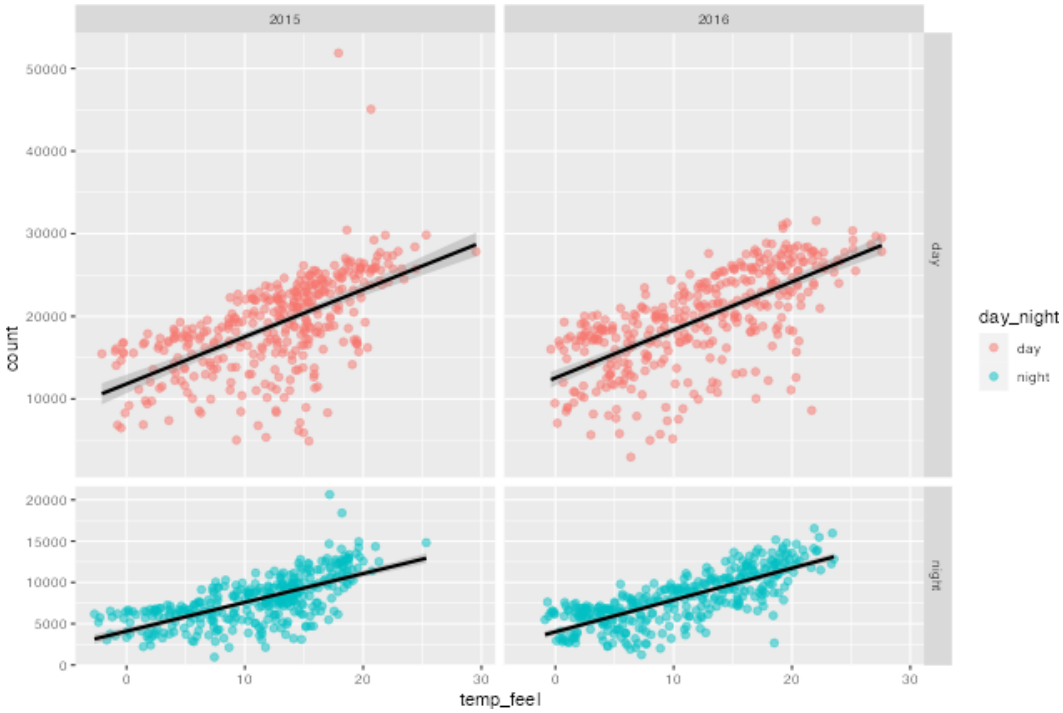



**FIGURE 4.7:** ggplot2 even allows to free the axis range—while ensuring equal axis spacing.

## 4.7 Change the Axis Scaling

For every aesthetic, **ggplot2** applies so-called *scales* that translate between variable ranges (data) and property ranges (aesthetic). For the positional axes, a set of `scale_x_*()` and `scale_y_*()` controls their behavior (Fig. 4.8). The `breaks` argument defines the placement of the axes ticks while the `labels` argument controls the labels next to those ticks. The labels can be either overwritten by a vector of the same length as the breaks (as for x in our example) or a function that returns a vector based on the breaks (as for y in our example).

There are many more arguments such as `expand` to control the padding towards the ends of the respective axis or `trans` to transform the scale. You can read more about how to control `scales` in Chapter XYZ and about label adjustment and styling in Chapter XYZ.

```
ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    aes(color = season), alpha = .5, size = 2
  ) +
  stat_smooth(
    aes(group = day_night), method = "lm", color = "black"
  ) +
  facet_grid(
    day_night ~ year, scales = "free_y", space = "free_y"
  ) +
  ## x axis
  scale_x_continuous(
    ## add °C symbol
    labels = function(x) paste0(x, "°C"),
    ## use 5°C spacing
    breaks = -1:6*5#,
    ## adjust axis padding
    #expand = c(mult = 0, add = 1)
  ) +
  ## y axis
  scale_y_continuous(
    ## add a thousand separator
    labels = scales::label_comma(),
    ## use consistent spacing across rows
    breaks = 0:5*10000#,
    ## adjust axis padding
    #expand = c(mult = .1, add = 0)
  )
```

## 4.8 Use a Custom Color Palette

The `scales` do not only control the behavior of the axes but also all other aesthetics such as `color`, `shape` or `alpha`. In case of categorical colors, `scale_color_manual()` enables us to overwrite the default color set by passing a vector of colors.

The colors will be applied in order, i.e. the levels of the factor or alphabetical in case of character variables. To ensure correct mapping, one can also use a named vector.

```
g1 <- ggplot(bikes, aes(x = temp_feel, y = count)) +
  geom_point(
    aes(color = season), alpha = .5, size = 2
  ) +
  stat_smooth(
```

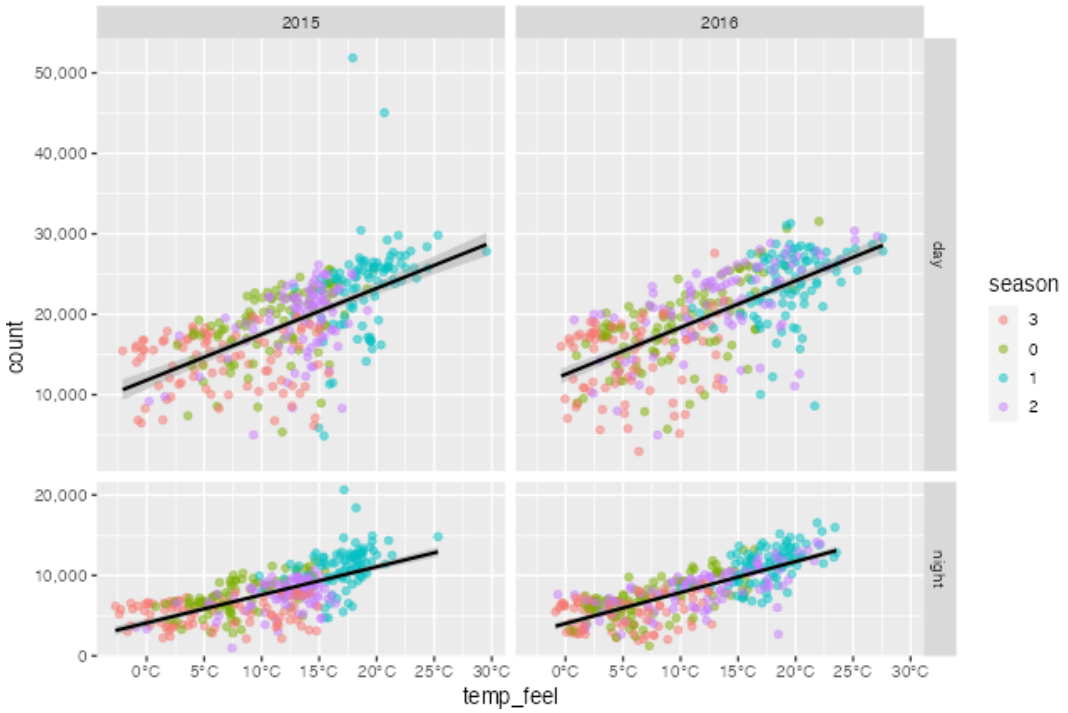



**FIGURE 4.8:** To adjust the formatting of axis labels, the respective axis needs to be addressed via the `scale_*()` functions.

```
    aes(group = day_night), method = "lm", color = "black"
  ) +
  facet_grid(
    day_night ~ year, scales = "free_y", space = "free_y"
  ) +
  scale_x_continuous(
    breaks = -1:6*5, labels = function(x) paste0(x, "°C")#, expand = c(mult = 0, add = 1)
  ) +
  scale_y_continuous(
    breaks = 0:5*10000, labels = scales::label_comma()#, expand = c(mult = .1, add = 0)
  ) +
  ## use a custom color palette for season colors
  scale_color_manual(
    values = c("#3c89d9", "#1ec99b", "#F7B01B", "#a26e7c")
  )

g1
```

You will learn more about the different color and fill scales and how to manipulate colors in Chapter XYZ. The chapter also features an example how to create a corporate color or fill scale to ensure consistent color use.

Furthermore, we are storing the current plot in an object named `g1`. This `ggplot` object can be extended afterwards. Storing intermediate ggplot outputs is especially helpful if the code

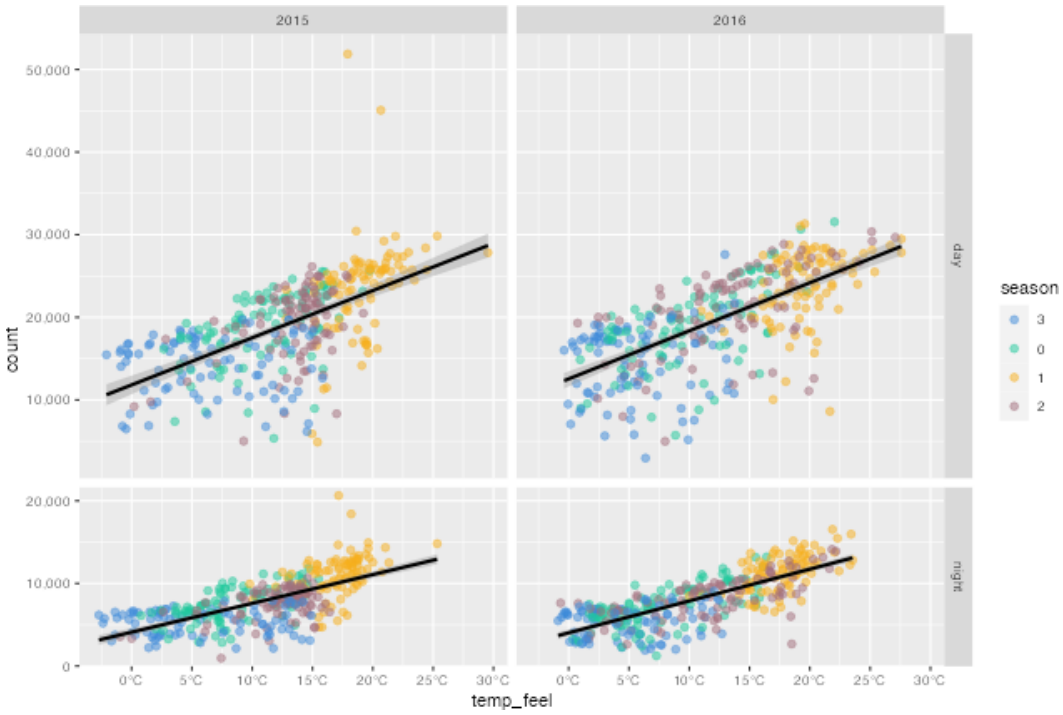



**FIGURE 4.9:** Categorical colors can be customized with the function `scale_color_manual()`.

becomes rather long and in case you have multiple variants of the same chart, for example when building up a chart stepwise. Another common use case is the creation of graphics in multiple languages.

## 4.9 Adjust Labels and Titles

For now, we have ignored that **ggplot2** uses the variable names as specified in our `bikes` object for data-related labels. Resist the temptation to change the variable names in your original data set! We can easily overwrite those by referring to the respective aesthetic (`x`, `y`, and `color`) in the `labs()` function which we add to our plot. Here, we can also specify a title, subtitle, caption, and tag (subtitle and tag are not shown in the following example; Fig. 4.10).

```
g2 <- g1 +
  labs(
    ## overwrite axis and legend titles
    x = "Average Feels-Like Temperature", y = NULL, color = NULL,
    ## add plot title and caption
    title = "Trends of reported bike rents versus feels-like temperature in London",
    caption = "Data: TfL (Transport for London), Jan 2015-Dec 2016"
  )
```

```
g2
```

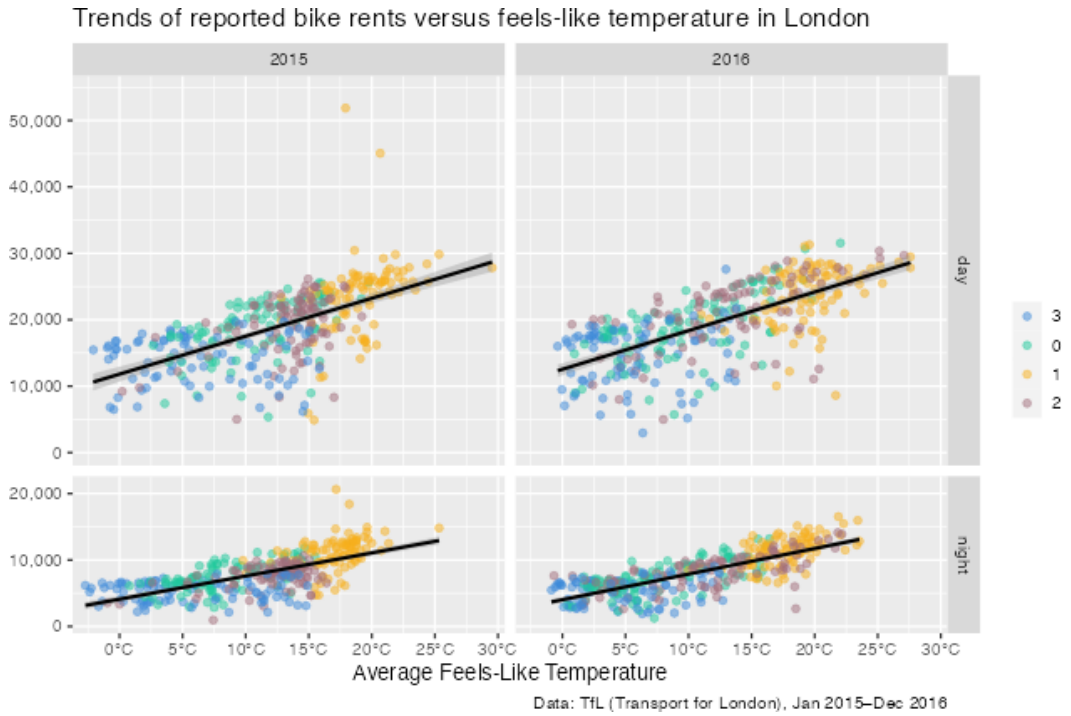



**FIGURE 4.10:** We can overwrite the default labels with the `labs()` which also allows to add a title, subtitle, caption, and tag to the graphic.

Let's also overwrite the cryptic numeric encoding of the seasons. Similar to the positional scales, we can pass a vector or function to the `labels` argument in the `scale_color_*()` component:

```
g3 <- g2 +
  scale_color_manual(
    values = c("#3c89d9", "#1ec99b", "#F7B01B", "#a26e7c"),
    labels = c("Winter", "Spring", "Summer", "Autumn")
  )

g3
```

Note that the code is returning a message informing you that the present color scale has been replaced. As you can only apply one scale per aesthetic, we also have to pass the color vector again to use the custom colors from before.

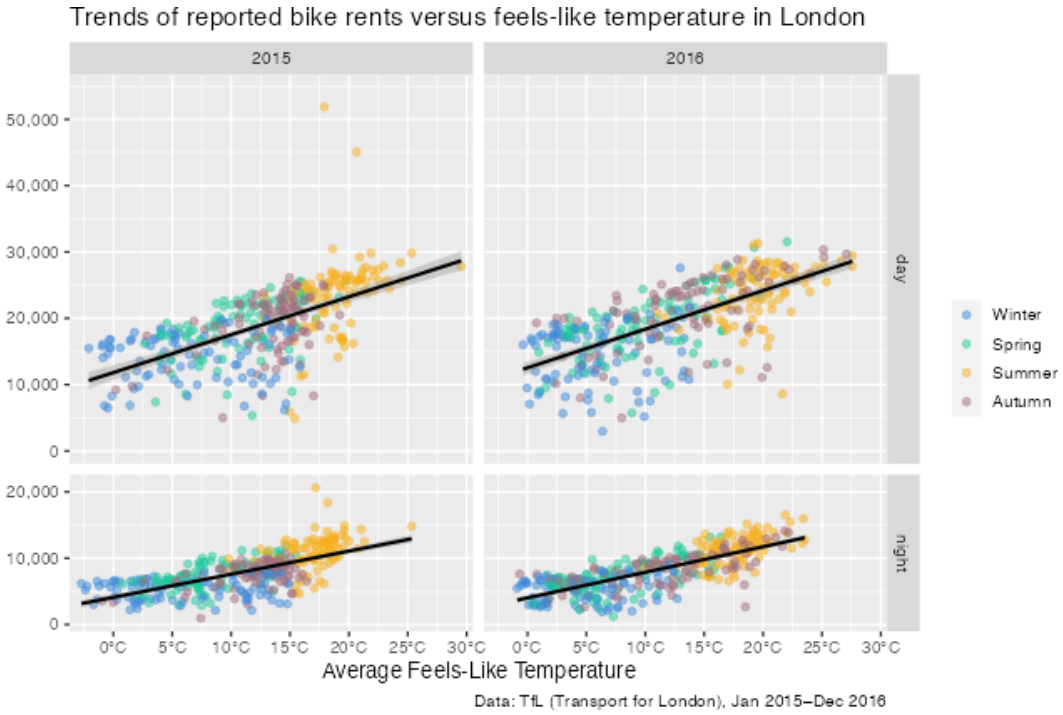



**FIGURE 4.11:** We can overwrite the legend text with a set of custom labels by passing a character vector to the `labels` argument of the `scale_color_*()` function.

## 4.10 Apply a Complete Theme

**ggplot2** comes with a set of so-called *complete themes*. We can add one of these to overwrite the default `theme_grey()` by one of the others (run `?theme_grey` for a full list of available complete themes).

When adding a theme to a ggplot, we can already overwrite the `base_size` of the theme elements, which translates to the size of text labels and line widths. Furthermore, we can set a non-default typeface by supplying a locally installed typeface in the `base_family` argument (Fig. 4.12).

```
g3 +
  ## add theme with a custom font + larger element sizes
  theme_light(
    base_size = 15, base_family = "Spline Sans"
  )
```

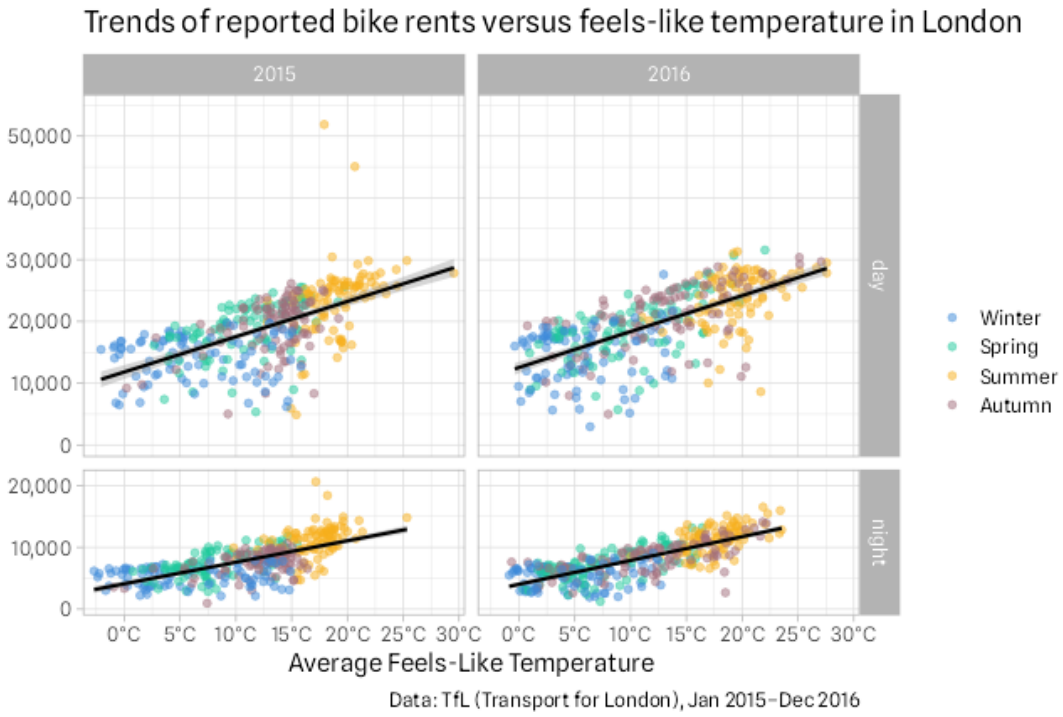



**FIGURE 4.12:** The graphic now comes with a new look and a non-default font used for all text labels.

## 4.11 Customize the Theme

The complete themes follow a set of rules that overwrite the choices made for the default `theme_grey()`. By using a `theme()` call after adding a complete theme, one can adjust the appearance of single elements (e.g. `plot.title` or `panel.grid.major.x`) as well as set some predefined options such as the positions of titles, captions, and legends (`plot.title.position`, `plot.caption.position` and `legend.position`).

In general, theme elements can be of class text (e.g. `plot.title`), line (e.g. `panel.grid.major.x`), or rectangle (e.g. `panel.background`). In addition, we can use `element_blank()` to remove an element entirely as set for `panel.grid.minor` in our example (Fig. 4.13). Other elements expect the input to be presets (e.g. `"plot"` or `"panel"` in `plot.title.position`), unit objects (e.g. `panel.spacing`), or margin specifications (e.g. `plot.margin`).

```
g3 +
  theme_light(
    base_size = 15, base_family = "Spline Sans"
  ) +
  ## theme adjustments
  theme(
    ## left-align title + right-align caption
```

```
  plot.title.position = "plot",
  plot.caption.position = "plot",
  ## larger, bold title
  plot.title = element_text(face = "bold", size = rel(1.3)),
  ## monospaced font for axes
  axis.text = element_text(family = "Spline Sans Mono"),
  ## left-aligned, grey x axis label
  axis.title.x = element_text(
    hjust = 0, color = "grey20", margin = margin(t = 12)
  ),
  ## larger, bold facet labels
  strip.text = element_text(face = "bold", size = rel(1.15)),
  ## no vertical major and no minor grid lines
  panel.grid.major.x = element_blank(),
  panel.grid.minor = element_blank(),
  ## increase white space between panels
  panel.spacing = unit(12, "pt"),
  ## place legend above plot
  legend.position = "top",
  ## larger legend labels
  legend.text = element_text(size = rel(1)),
  ## adjust margin
  plot.margin = margin(t = 2, r = 0, b = 0, l = 0)
)
```

The book covers themes in full detail in Chapter XYZ. Here you learn how to leverage complete themes, which other complete themes are provided by extension packages, and how to create your own custom, corporate theme to apply a cohesive and easily reproducible style to all of your graphics.

[WIP]

RECAP

LEAD OVER TO NEXT CHAPTER

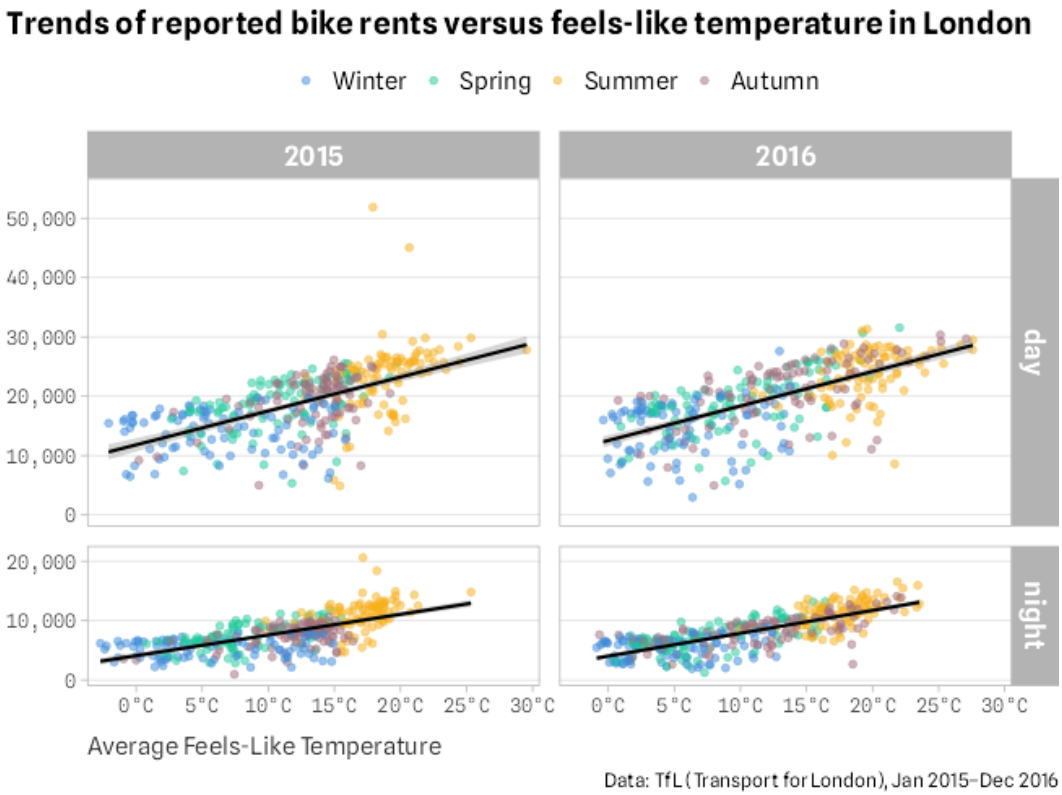



**FIGURE 4.13:** After applying one of the complete themes, the overall appearance can be further modified via the `theme()` function.

# 5

# *Tips to Improve Your ggplot Design*

In the following I want to share some tips how you quickly customize the appearance of your graphics with a few additional lines of code. First, let's create a plot to work with: we will display the counts of reported bike rents during the night only as a time series. To focus on the tips, we assign the ggplot output to `g` and extend this object in the following.

```
library(ggplot2)
library(dplyr)

bikes_nightly <- filter(bikes, day_night == "night")

g <-
  ggplot(bikes_nightly, aes(x = date, y = count)) +
  geom_point(aes(color = temp), alpha = .5) +
  ## style y axis, labels and colors
  scale_y_continuous(limits = c(0, NA), labels = scales::label_comma()) +
  scale_color_gradient(
    low = "#00B3FF", high = "#FF5477",
    labels = scales::label_number(suffix = "°C")
  ) +
  labs(
    x = NULL, y = "Number of bike rents per night", color = "Average Temperature:",
    title = "TfL bike rents peak during warm summer nights",
    subtitle = "A time series of reported bike rents between 6 pm and 6 am in London",
    caption = "Data: TfL (Transport for London), Jan 2015-Dec 2016"
  )

g
```

## 5.1 Use a Different Theme

The simplest step to give your plot a different appearance is using one of the complete themes provided by **ggplot2**. We already have seen `theme_light()` in the previous section. A full list of themes:

- `theme_gray()`, the default theme with a grey panel, white grid lines, and grey boxes for facets
- `theme_minimal()` without a panel border and facet boxes * `theme_bw()` with a white panel, grey grid lines, a black panel border, and grey boxes for facets

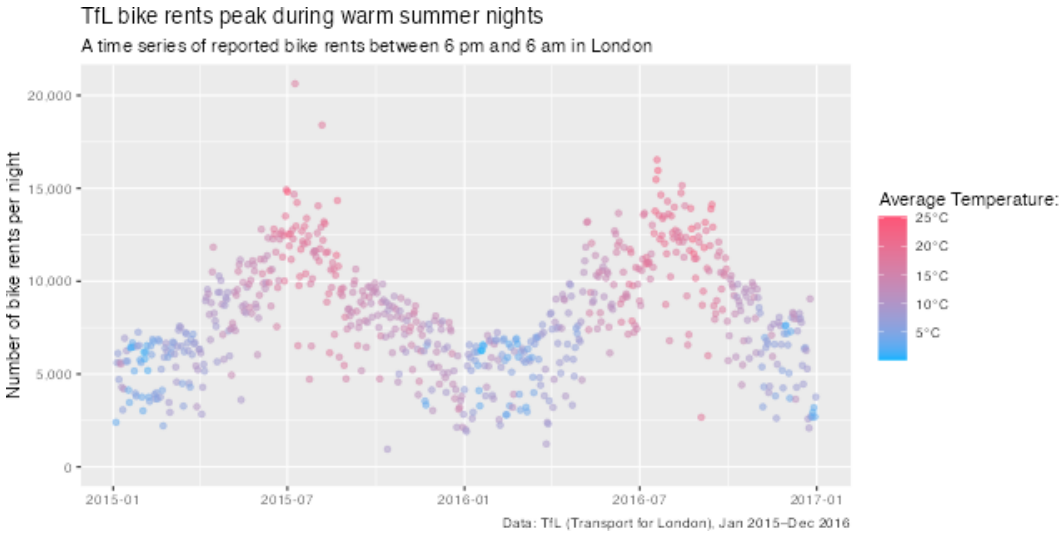



**FIGURE 5.1:** A time series of reported bike rents during the night (6 pm to 6 am) with points being encoded by the average temperature using the default **ggplot2** theme.

- `theme_linedraw()`, similar to `theme_bw()` but featuring black axis text and facet boxes
- `theme_light()`, similar to `theme_bw()` but with a grey panel border and all-grey facet boxes
- `theme_minimal()` without a panel border and facet boxes
- `theme_classic()` with axis lines but no grid lines and white facet boxes with a black outline
- `theme_dark()` with a dark-grey panel and even darker facet boxes
- `theme_void()`, a completely empty theme

The completely empty theme `theme_void()` is often helpful when working with maps. Also, you can use it to build your theme element by element like drawing on a blank canvas.

Let's use the popular `theme_minimal()` for now:

```
g + theme_minimal()
```

## 5.2 Use a Custom Font

There are multiple extension packages that aim to provide access to font files, **extrafont**, **sysfont**, **showtext**, and **systemfonts** to name a few. To my knowledge, the **systemfonts** package provides the best solution to work with typefaces and resolve font specifications. After loading the package, all locally installed font files are available in your current R session.

The default font family of ggplot themes is the default sans serif font and thus it is specific to the system you and potential collaborators are working on. You can overwrite the default by passing a font family as argument to `base_family` inside the `theme_*()` call.

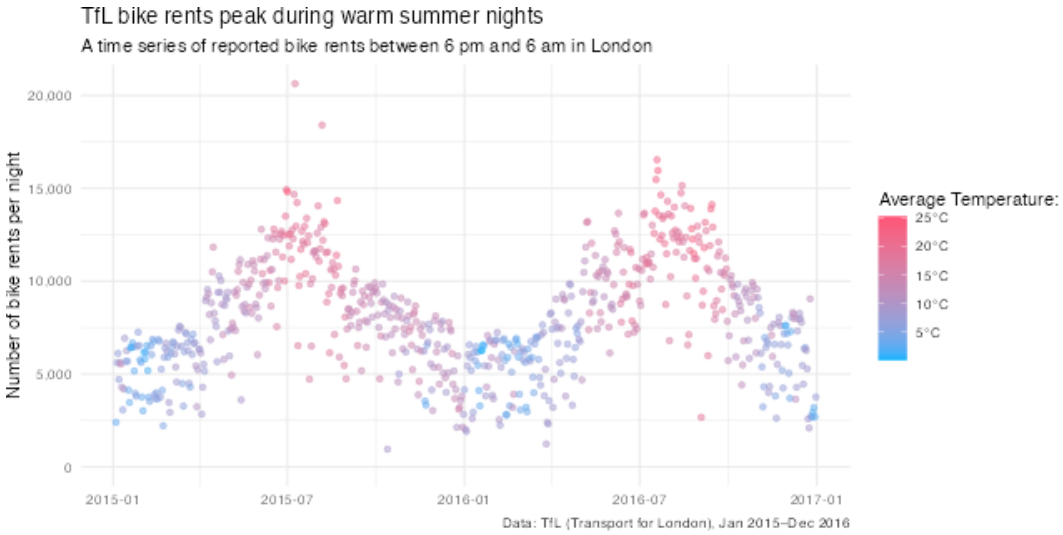



**FIGURE 5.2:** The same plot as before using the minimal theme.

```
g + theme_minimal(base_family = "Asap Condensed")
```

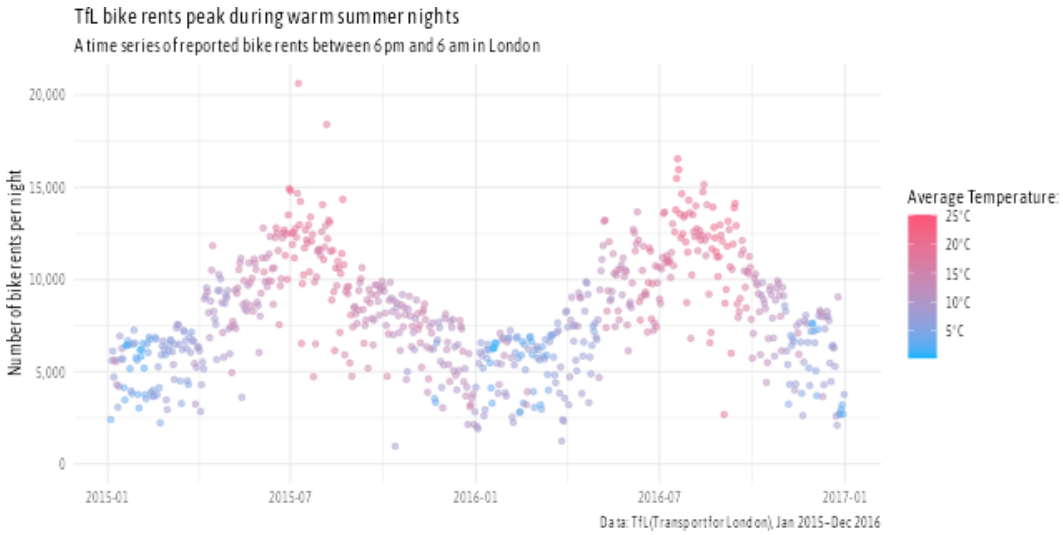



**FIGURE 5.3:** All text labels now use a custom font by changing the `base_family` of the theme.

If you are searching for a specific font, this is my script to return family names that can be used as inputs for text elements in **ggplot2**:

```
systemfonts::system_fonts() |>
  ## filter for a pattern
  dplyr::filter(stringr::str_detect(name, "Asap")) |>
  ## only keep family column
```

```
  dplyr::pull(family) |>
  ## only keep distinct families and sort them by name
  unique() |>
  sort()
```

```
## [1] "Asap"               "Asap Condensed"
## [3] "Asap Expanded"      "Asap SemiCondensed"
## [5] "Asap SemiExpanded"
```

## 5.3 Increase the Font Sizes

The base size in any ggplot theme is set to 11. This number determines the individual text sizes and line widths of your theme elements. Depending on the width and height specified when saving your plot, you may want to adjust the overall size by changing the `base_size` in the `theme_*()` call.

```
(g <- g + theme_minimal(base_family = "Asap Condensed", base_size = 14))
```

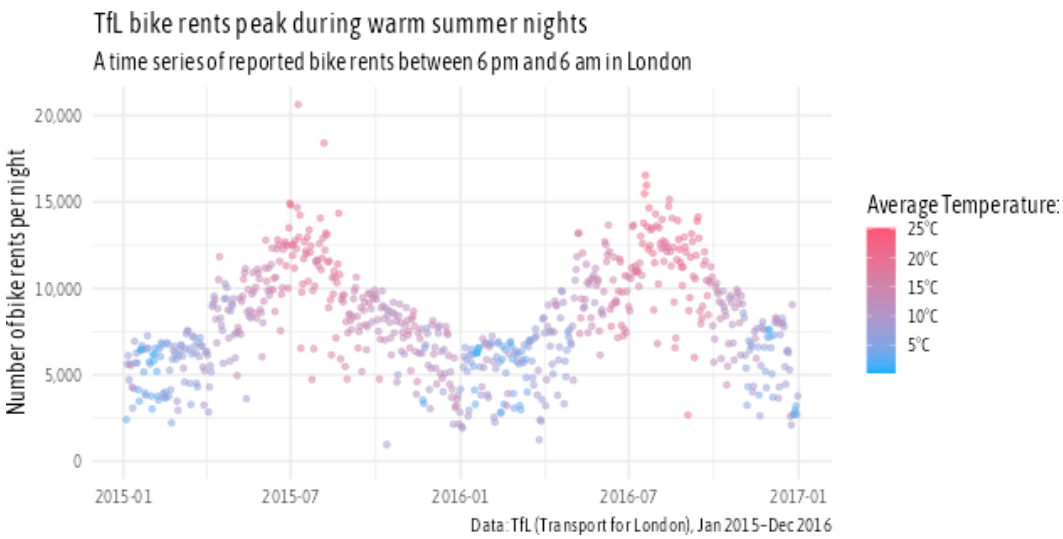



**FIGURE 5.4:** If the text elements are too small for the desired out width and height, you can adjust them by overwriting the default `base_size`.

## 5.4 Create Text Hierarchy

By default, the title is always set in regular weight. In many cases I find myself using a font that is bold and larger than by default. You can update details of the theme by directly addressing the respective theme element, here `plot.title`. As the title is a text element, we

pass the new settings inside the element_text() function. The `face` argument controls the font style and can take `"regular"`, `"bold"`, `"italic"`, and `"bold.italic"` as inputs. The size can either be set in absolute numbers or relative to the `base_size` via the `rel()` function.

```
(g <- g + theme(plot.title = element_text(face = "bold", size = rel(1.4))))
```

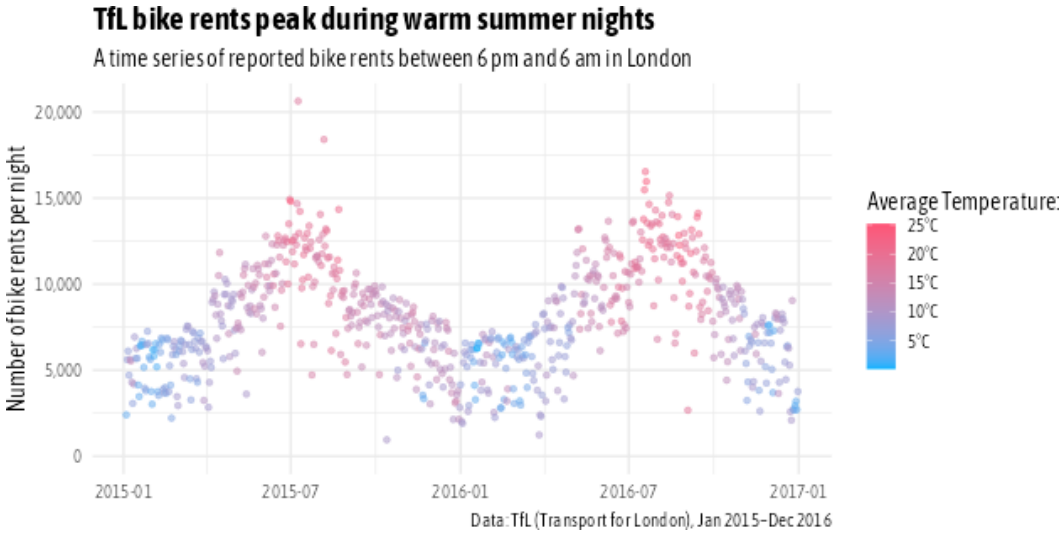
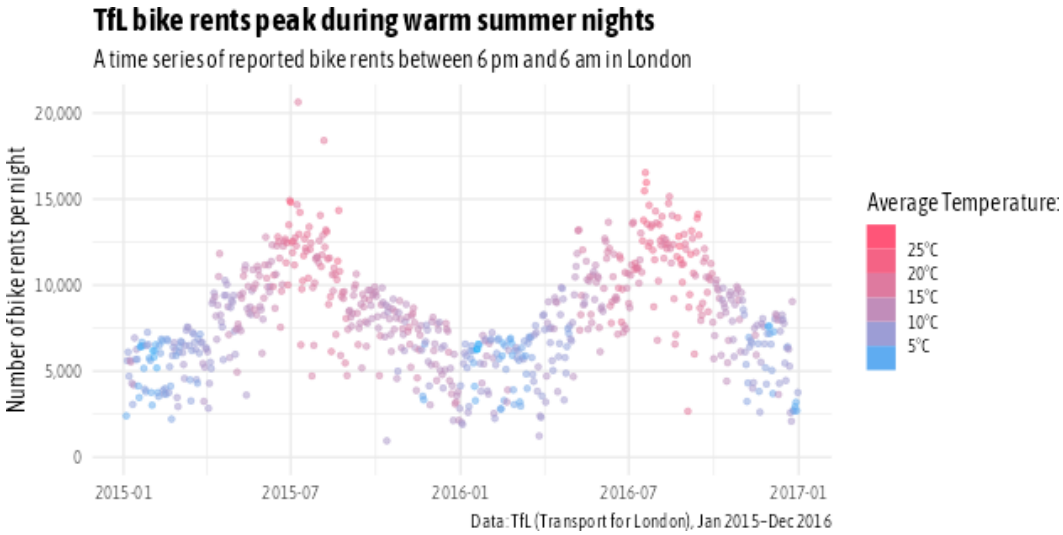



**FIGURE 5.5:** One can style single theme elements with the `theme()` function. Here, we create a more pronounced tet hierarchy by setting the title in a bigger, bold typeface.

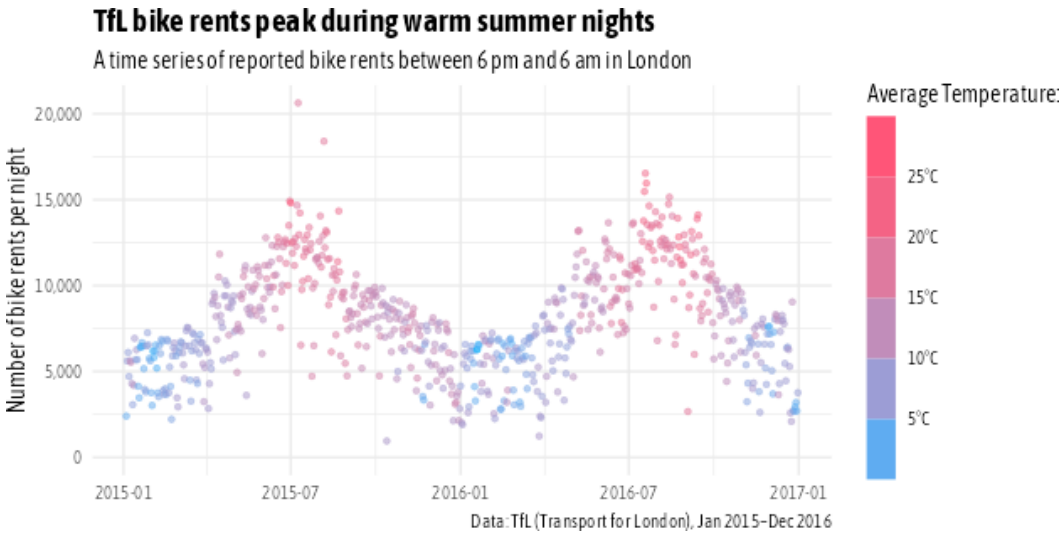
## 5.5 Modify the Legend

[WIP]

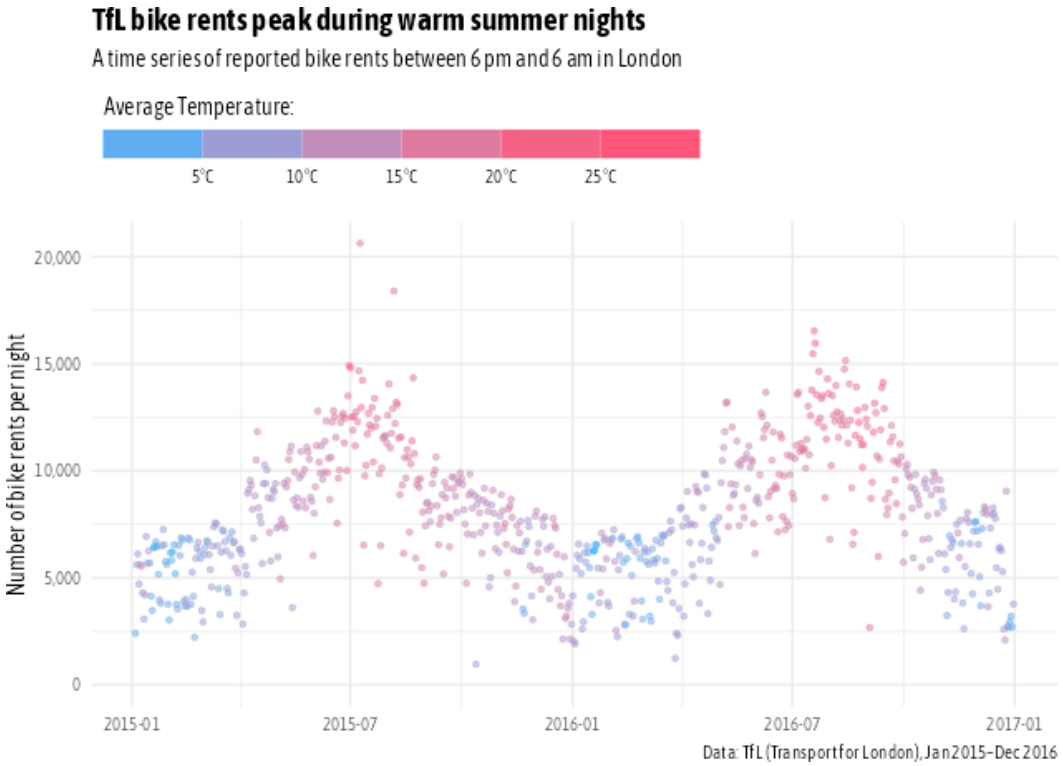
```
g + guides(color = guide_colorsteps())
```




```
g + guides(color = guide_colorsteps(
    barheight = unit(15, "lines")
  ))
```




```
g + guides(color = guide_colorsteps(
    barwidth = unit(25, "lines"), title.position = "top"
  )) +
  theme(legend.position = "top", legend.justification = "left")
```




## 5.6 Align Titles and Captions

By default, the titles and captions are left-aligned with the horizontal justification being set to `hjust = 0`. However, the horizontal justification does only impact the alignment of the title relative to the panel. If you want to place these text elements relative to the outer border of your graphic. you can set `plot.title.position` `plot.caption.position` to `"plot"`, respectively

```
g + theme(plot.title.position = "plot",
          plot.caption.position = "plot")
```

The importance of aligning the title either with the panel border (the default behavior) or with the plot border becomes especially important if you have long labels on the left of the plot. This is often the case when displaying categories along the y axis with a left-aligned title (Fig. 5.7. The same applies for right-aligned text in case of increased margins on the right of the panel, for example when using facets or adding legends.

## 5.7 Add Some White Space

[WIP]

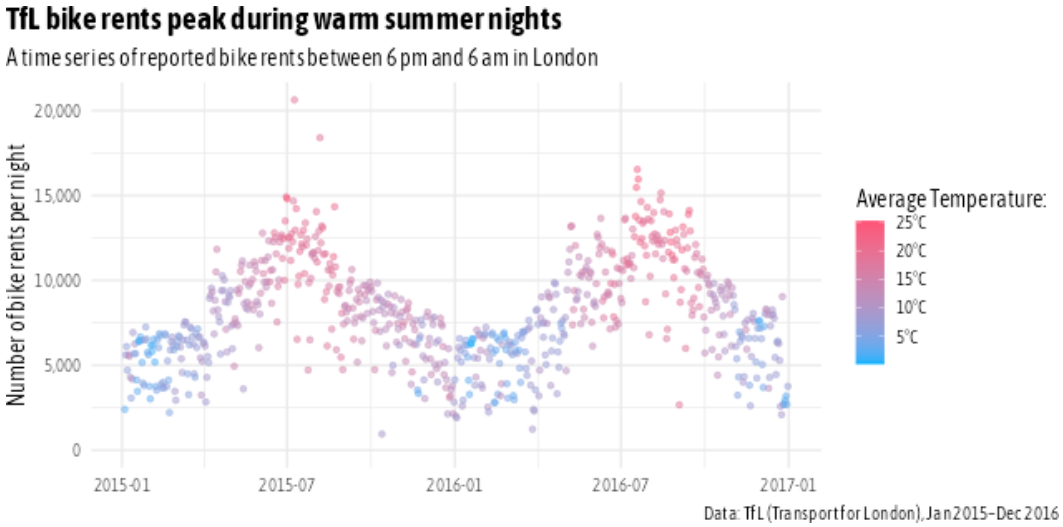



**FIGURE 5.6:** The title and sutitle can be aligned with the plot border by setting `plot.title.position` to `"plot"`. Similarly, the position of the caption can be changed by using `plot.caption.position`.

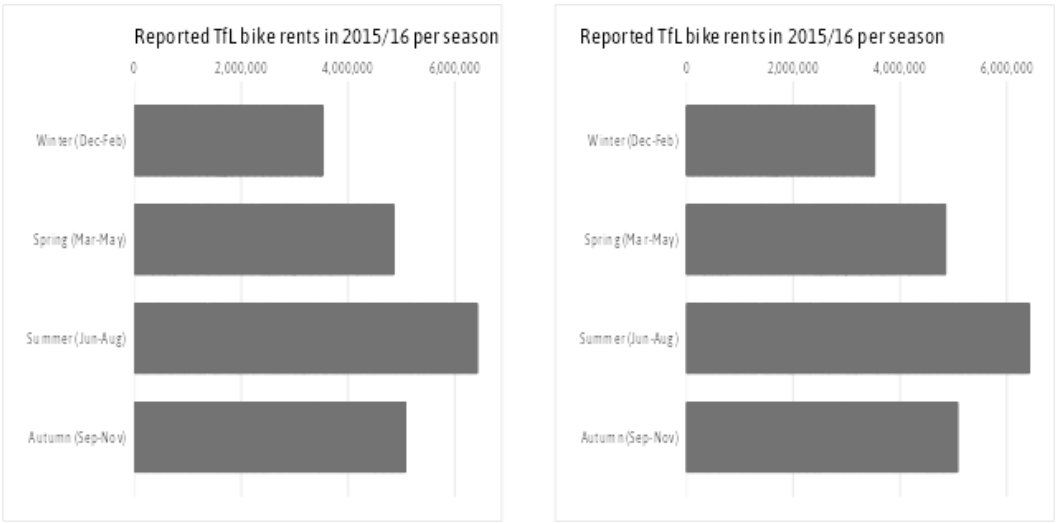
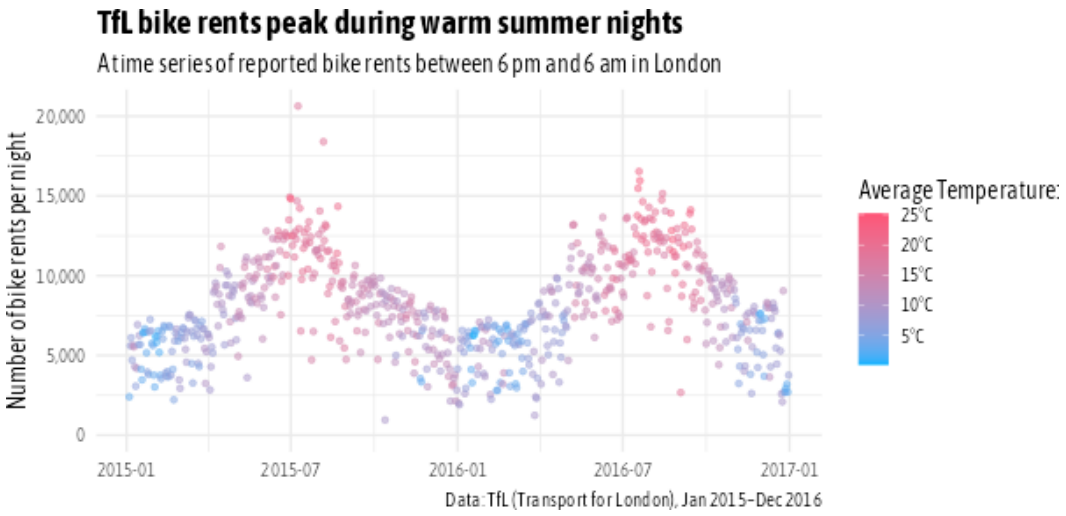
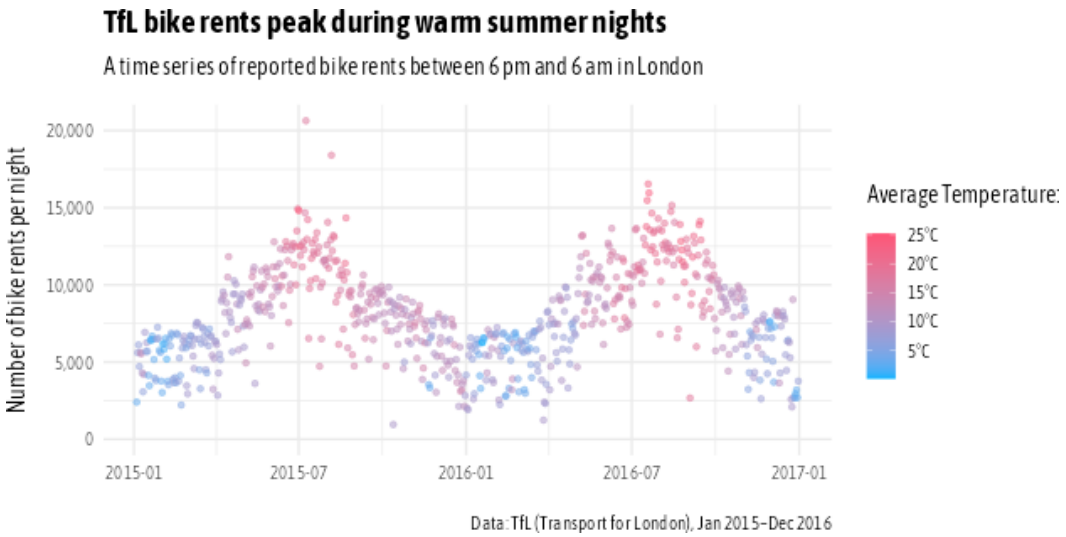



**FIGURE 5.7:** Plots with long labels on the y axis with the title being either aligned with the panel border (default behavior, left) or with the plot border (right).

```
g + theme(plot.margin = margin(rep(20, 4)))
```




```
g + theme(
  plot.title = element_text(margin = margin(b = 10)),
  plot.subtitle = element_text(margin = margin(b = 15)),
  plot.caption = element_text(margin = margin(t = 20)),
  axis.title.y = element_text(margin = margin(r = 10)),
  legend.title = element_text(margin = margin(b = 10)),
)
```



# 6

# *Tips to Improve Your ggplot Workflow*

When designing graphics with a code-based software, changes in the code need to be rendered to a graphical output to see the actual changes. The process of saving the graphic successfully in the desired format and being able to quickly inspect the adjustments are often a main source of frustration. The first tips thus cover how-to's on saving your ggplot output in the correct dimension, improving the iterative design process, and embedding non-default typefaces.

A benefit of a code-based graphics software is the ability to automate steps to avoid redundant code and allow for consistency. By adjusting the default settings, reusing code or even writing your own helper functions and themes rather than typing the same code again and again enables one to work more efficiently while preventing mistakes and inconsistencies.

## 6.1 Save ggplot Output with the "Correct" Dimensions

One of the most common question I hear is "how do I save the plot in a way that it looks as in the RStudio Plots pane?". As the Plots pane in RStudio is responsive, it will redraw the plot with new dimensions every time you resize the pane. Consequently, the relative size of geometries and theme elements changes as well. Also, the dimensions of the plot shown are not obvious so you may end up in a loop of guessing the right measurements and adjusting the sizes, margins, widths of your components.

For sure, the iteration of adjusting sizes and position is one of the most annoying and elaborate tasks when designing graphics with **ggplot2**. And most ggplot users will painfully remember the moment when a plot saved to disk looks absolutely nothing as it did in the Plots pane. So how can we avoid this time-consuming task?

***Don't rely on the preview and decide on the dimensions before polishing:*** Once you are moving from exploring the data and potential visualizations to creating your final polished graphic, save the plot to disk in the desired width and height. By default, the dimensions are in inches.

```
ggsave("plot.png", width = 12, height = 8)
ggsave("plot.png", width = 30, height = 20, unit = "cm")
```

From now on, stick to this dimension and adjust the components after inspecting the saved graphic. Of course, you can adjust the dimensions afterwards. Slight changes will have no big impact on the relative sizes; otherwise, you may have to resize some of your components.

If you are saving raster files such as .png and .jpeg, make sure to set the desired resolution by setting `dpi`. For web applications, images usually should be of low resolution to avoid

long loading times (72 dpi is the general recommendation) while graphics used for print should be of high resolution of 300 dpi or more. The default in `ggsave()` is 300 dpi.

## 6.2 Display ggplot Output with the "Correct" Dimensions

The iterative process of adjusting your graphic, saving it to disk, navigating to the folder, and looking at the plot is a tedious one. To avoid the additional workload, you can optimize your workflow by previewing your graphic in the correct dimensions without leaving RStudio.

***Work in Quarto (.qmd) or Rmarkdown (.rmd) files to inspect your plots in-line:*** In qmd and rmd formats, code is run inside individual chunks for which you can define figure dimensions by setting `fig.width` or `fig.height` as chunk options [1].

```
```{r, fig.width=12, fig.height=8}
```
```

When a chunk is executed, the in-line plot output will respect these settings. When saving the file with `ggsave()`, make sure to use the same width and height. By default, the units in both chunk settings and `ggsave()` are in inches.

**Use the camcorder package to automatically save and display your graphics:** The **camcorder** package[3] (Hughes, 2022) is an R package that automatically saves graphics generated with **ggplot2** with custom settings. The package allows you to inspect the ggplot output directly in RStudio by displaying the saved graphic in the Viewer pane. As each graphic is saved automatically, running `ggsave()` after every adjustment becomes obsolete—another time-saver!

You initiate a recoding by calling `gg_record()` with the desired path to save the plots to and image settings as you specify them in `ggsave()`:

```
install.packages("camcorder")

camcorder::gg_record(
  dir = "./plots", ## where to save the recording
  device = "png",  ## device to use to save images
  width = 12,      ## width of saved image
  height = 8,      ## height of saved image
  dpi = 300        ## dpi to use when saving image
)
```

If you want to change some settings later, use `gg_resize_film()`:

```
camcorder::gg_resize_film(width = 12.5)
```

The original purpose of **camcorder** is the generation of a GIF that showcases every step

[1] Note that the new YAML-styled `#| fig-width: 12` does not work and the width and height need to be specified in the **knitr** chunk options[2].

[3] https://thebioengineer.github.io/camcorder/

of the design process using the saved image files. If you want to create such an animation, run `gg_playback()`:

```
camcorder::gg_playback(name = "./plots/process.gif")
```

## 6.3 Make Fonts Work

***Use the systemfonts package to find and use typefaces:*** As discussed in the previous chapter, the **systemfonts** package is likely the preferred approach to work with fonts. To search for a font file on your system use `match_font()`. You can inspect all available typefaces via `system_fonts()`.

A common issue is the rendering of custom typefaces and special characters such as mathematical symbols, unusual glyphs, and emojis.

***If you are saving your graphics as PNG raster images use the agg_png device:*** The best-performing device to render your ggplot output successfully are the agg devices. In recent versions, this device is the default in `ggsave()` in case the **ragg** package is installed. It also features agg devices for other raster formats such as JPEG and TIFF.

```
library(ragg)
ggsave("plot.png", device = agg_png)
```

To make fonts work in the RStudio Plots pane, select the AGG device as default for the graphics backend by navigating to `Tools > Global Options... > General > Graphics`.

***If you are saving your graphics as PDF vector graphics use the cairo_pdf device:*** I prefer to dave my files as lossless vector graphics. The device known to work best for rendering is the Cairo device which needs to be secified as `device = cairo_pdf`. Mac users need to install XQuartz[4] which is needed to use the Cairo device.

```
ggsave("plot.pdf", device = cairo_pdf)
```

## 6.4 Set Your ggplot Theme Globally

Instead of applying and customizing a theme for each of your ggplots, you can set your theme once and it will be used for all future ggplot output by passing a complete theme to `theme_set()`.

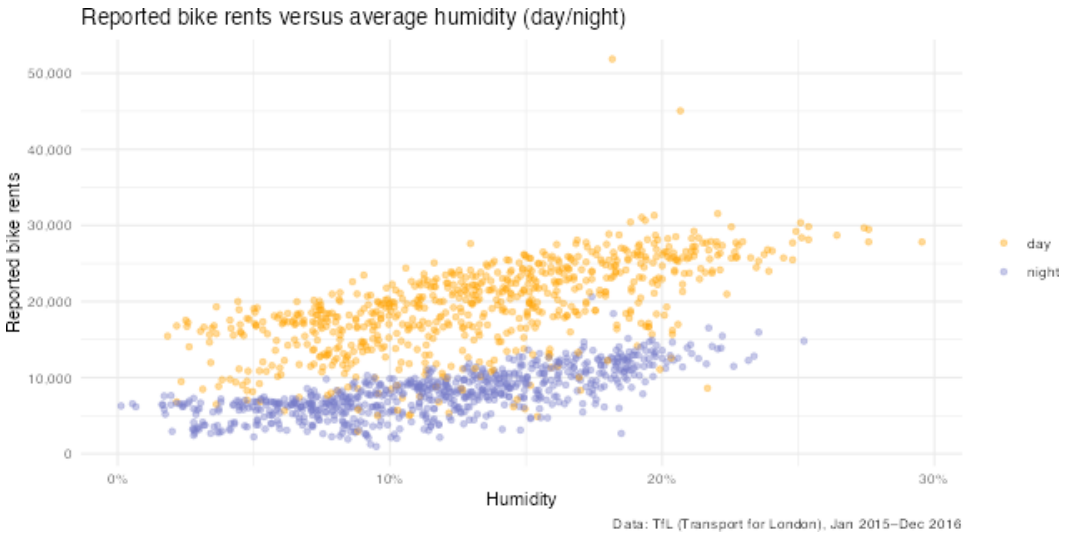
```
theme_set(theme_minimal())
```

[4] https://www.xquartz.org/

Let's see how a plot using `theme_minimal()` looks like—note that we do not apply the theme to the plot itself.

```
g <-
  ggplot(bikes, aes(x = temp, y = count)) +
  geom_point(aes(color = day_night), alpha = .4) +
  ## style labels and colors
  scale_x_continuous(labels = scales::label_percent(scale = 1)) +
  scale_y_continuous(labels = scales::label_comma()) +
  scale_color_manual(values = c("#FFA200", "#757bc7")) +
  labs(
    x = "Humidity", y = "Reported bike rents", color = NULL,
    title = "Reported bike rents versus average humidity (day/night)",
    caption = "Data: TfL (Transport for London), Jan 2015-Dec 2016"
  )

g
```




Furthermore, you can customize the new default theme supplied to `theme_set()`:

1. Set the typeface for text elements and sizes of the theme elements when calling the complete theme.
2. Adjust theme elements and components of the complete theme via `theme_update()`.

```
theme_set(
  theme_minimal(base_family = "Asap SemiCondensed", base_size = 15)
)

theme_update(
  panel.grid.minor = element_blank(),
  plot.title = element_text(face = "bold"),
```

```
  plot.title.position = "plot",
  plot.caption = element_text(color = "grey40")
)
```

[WIP]

```
g
```

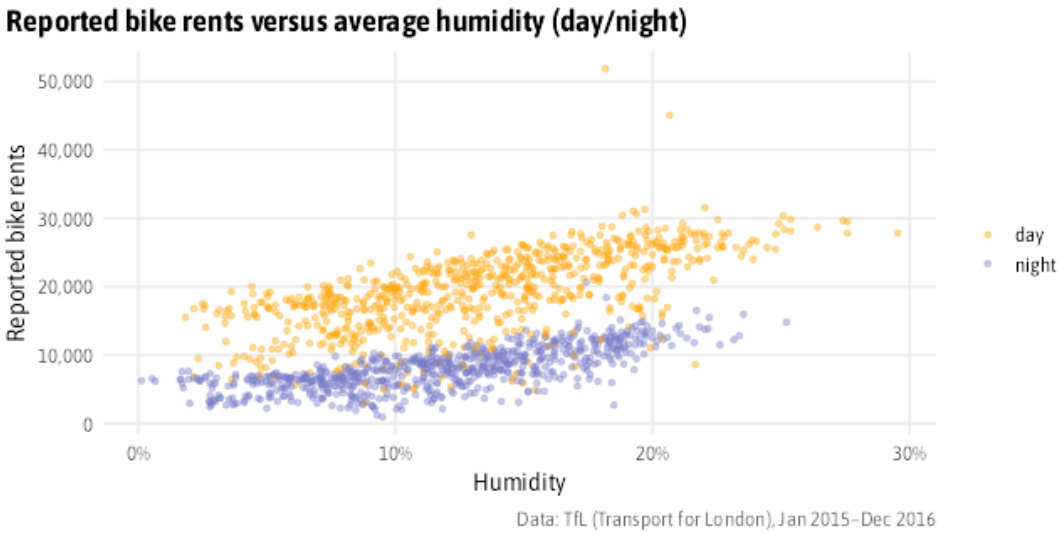



**FIGURE 6.1:** Running the same plot with updated theme defaults, globally set via `theme_update()`. This allows for a consistent style for all graphics without the need to write redundant code.

## 6.5 Automating Plots: Batch Operations

You can create the same plot for a range of subsets by iterating over a list of filter conditions. First, we need to specify a function that filters the data and plots the subset—and optionally also saves the output.

```
plot_scatter <- function(cat, save = TRUE) {
  g <-
    ggplot(data = filter(bikes, season %in% cat),
           aes(x = temp, y = count, color = day_night)) +
    geom_point() +
    ## keep axis ranges consistent
    scale_x_continuous(limits = range(bikes$temp)) +
    scale_y_continuous(limits = range(bikes$count)) +
    scale_color_manual(values = c("#FFA200", "#757bc7")) +
    labs(title = stringr::str_to_title(cat),
```

```
        x = "Temperature", y = "Bike shares", color = NULL)

  if (isTRUE(save)) {
    ggsave(filename = paste0("scatter_", cat, ".png"),
           width = 7, height = 5)
  }

  return(g)
}
```

Afterwards, we can perform the function for multiple subsets by iterating over a vector of categories (here seasons) with the help of the **{purrr}** package. Alternatively, you can use `lapply()` or a for loop to iterate over your filter conditions.

```
## print the plots as well
purrr::map(unique(bikes$season), ~plot_scatter(cat = .x))

## just run the code without printing
purrr::walk(unique(bikes$season), ~plot_scatter(cat = .x))
```

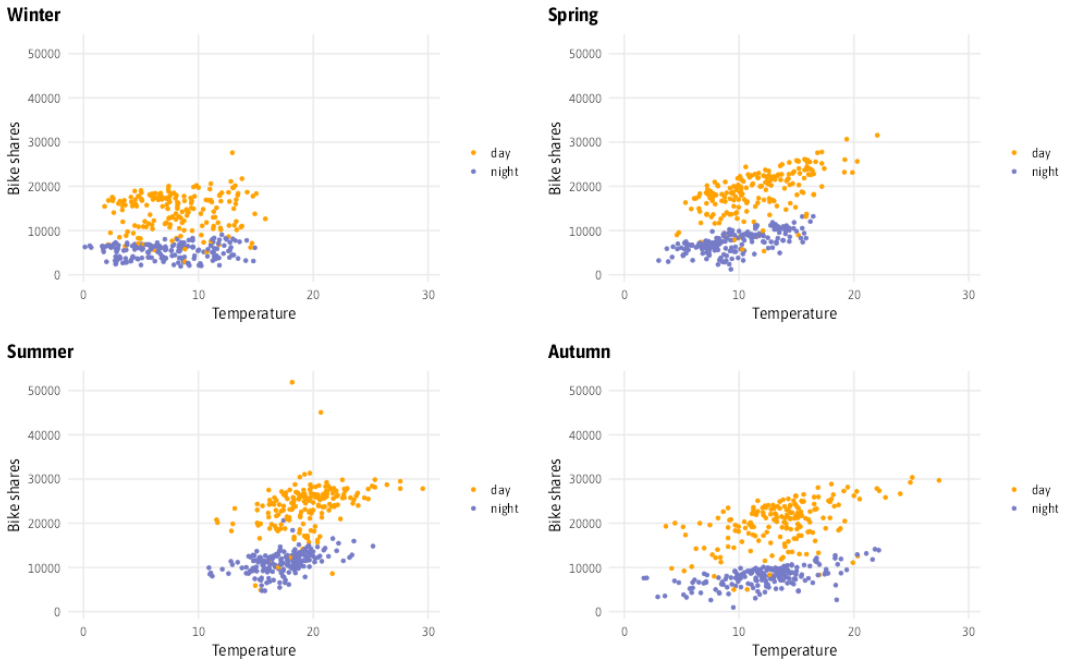



**FIGURE 6.2:** The four scatter plots of bike share versus temperature per season, produced by iterating our custom `plot_scatter()` function over the four different groups.

Note that the axis limits differ depending on the range of the positional variables `temp` and `count`. If you want to ensure consistent ranges, add the following two lines to the plot inside the function:

```
scale_x_continuous(limits = range(bikes$temp)) +
scale_y_continuous(limits = range(bikes$count))
```

You can leverage this approach to create the same chart for different data sets, to map different variables to aesthetics, and more.

Here is another, more general function to create a custom density plot for any quantitative variable of a data set of your choice. The function makes use of *nonstandard evaluation* to pass the variables and groupings as strings. The `!!sym()` notation allows to pass quoted variable names that are evaluated by **ggplot2**. `sym()` converts the string to a symbol, the so-called *bang-bang* `!!` then unquotes the symbol so that the `aes()` function can handle it as usual.

```
plot_density <- function(data, var, grp = "") {
  g <-
    ggplot(data, aes(x = !!sym(var))) +
    geom_density(aes(fill = !!sym(grp)), position = "identity",
                 color = "grey30", alpha = .3) +
    coord_cartesian(expand = FALSE, clip = "off") +
    scale_y_continuous(labels = scales::label_number()) +
    scale_fill_brewer(palette = "Dark2", name = NULL) +
    theme_minimal(base_family = "Asap Condensed") +
    theme(panel.grid.minor = element_blank(),
          panel.grid.major.x = element_blank(),
          legend.position = "top")

  return(g)
}

## apply to the `mpg` data from ggplot2
purrr::map(c("displ", "hwy", "cty"),
           ~plot_density(data = mpg, var = .x))

## apply to the `bikes` data with additional grouping
purrr::map(c("count", "temp", "humidity"),
           ~plot_density(data = bikes, var = .x, grp = "day_night"))
```

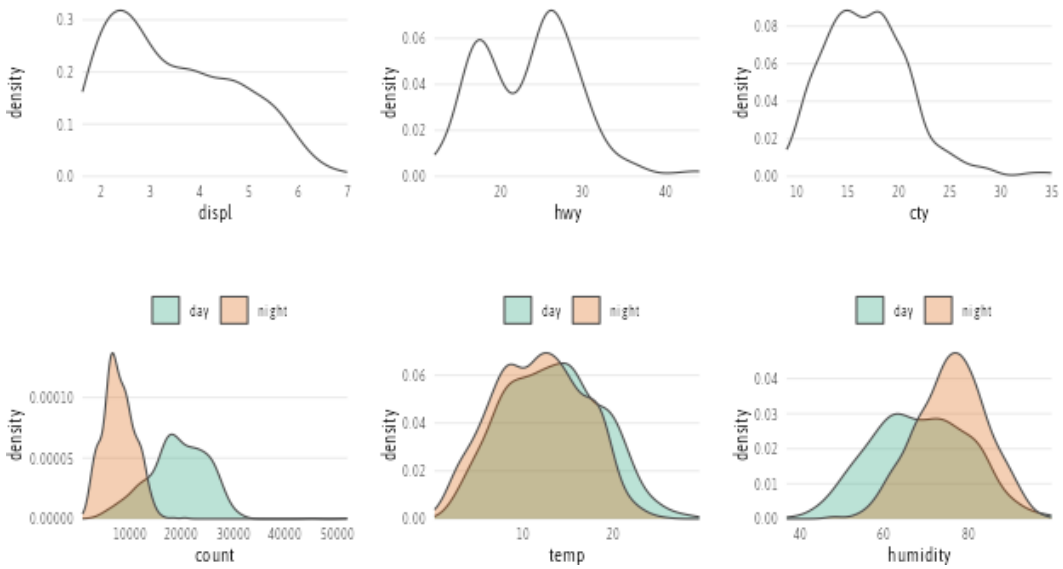



**FIGURE 6.3:** Two example use cases of the custom `plot_density()` function applied to three variables of the `mpg` data (upper row) and the `bikes` data (lower row), respectively. For the latter, `day_night` was passed as the grouping variable.

# Part II

# How To Work with Components

# 7

# *Working with Layers*

Layers represent the data in our visualizations, specifying the visual representation of variables such as points, lines, or bars. Internally all layers are created by the `layer()` function. A layer consists of

- the data set (`data`)
- a mapping of variables to aesthetics (`mapping`)
- a statistical transformation (`stat`)
- a geometric representation (`geom`)
- a positional adjustment (`position`)

```
ggplot() +
  layer(
    data = bikes,
    mapping = aes(x = temp, y = humidity),
    geom = "point",
    stat = "identity",
    position = "identity"
  )
```

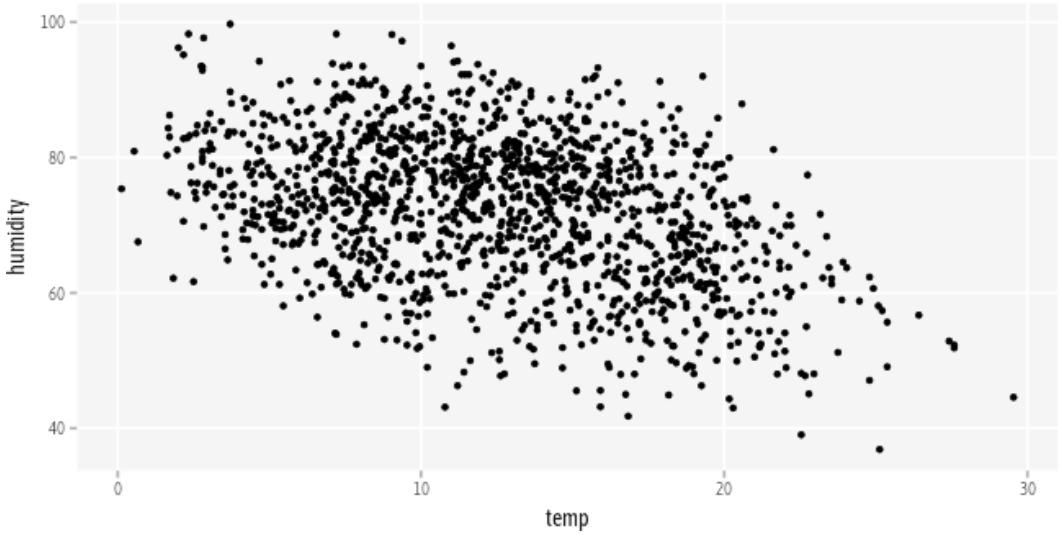



**FIGURE 7.1:** A simple scatter plot of humidty versus temperature created with the `layer()` function by passing the data, the positional aesthetic, a statistical transformation, a geomtric representatio n and the positional adjustment.

We map the variables of the specified data the positional aesthetics `x` and `y`. The data and

aesthetic mappings are often specified in the initial `ggplot()` call. These specifications are then passed to all layers.

```
ggplot(data = bikes, mapping = aes(x = temp, y = humidity)) +
  layer(geom = "point", stat = "identity", position = "identity")
```

Throughout the book, I am matching the argument names `data` and `mapping` in the initial `ggplot()` call implicitly: `ggplot(bikes, aes(x = temp, y = count))`. This is common practice and is used in many scripts, tutorials, and examples featuring **ggplot2** code.

## 7.1 Predefined Layers

To simplify your life, **ggplot2** features a long list of predefined individual layers with sensible default inputs for `geom`, `stat`, and `position`. Focusing on either the geometrical representation or the statistical transformation, these functions start with `geom_*()` or `stat_*()`.

### 7.1.1 Geometrical versus Statistical Layers

For all geometries and statistical transformation there are respective `geom_*()` and `stat_*()` functions. This means, you can usually go both routes to create the same chart. The following codes produce the same output as the `layer()` function above, a scatter plot of humidity versus temperature:

```
ggplot(bikes, aes(x = temp, y = humidity)) +
  geom_point() ## with `stat = "identity", position = "identity"`

ggplot(bikes, aes(x = temp, y = humidity)) +
  stat_identity() ## with `geom = "point", position = "identity"`
```

Similarly, `geom_bar()` and `stat_count()` produce a *bar* chart with the height encoding the *count* per variable. For both, the default positional adjustment is `"stack"`. Mapping a variable to the `fill` aesthetic thus creates stacked bar charts:

```
ggplot(bikes, aes(x = weather_type, fill = year)) +
  geom_bar() ## with `stat = "count", position = "stack"`

ggplot(bikes, aes(x = weather_type, fill = year)) +
  stat_count() ## with `geom = "bar", position = "stack"`
```

Maybe you have wondered at some point, why you can use `geom_smooth()` and `stat_smooth()` interchangeably to create a conditional smoothing? This is because both call the same underlying layer function with the default arguments: `stat = "smooth"` which computes the fitted line and `geom = "smooth"` which draws the line and ribbon to visualize the mean and standard error, respectively.

**geom_bar() or stat_count()**

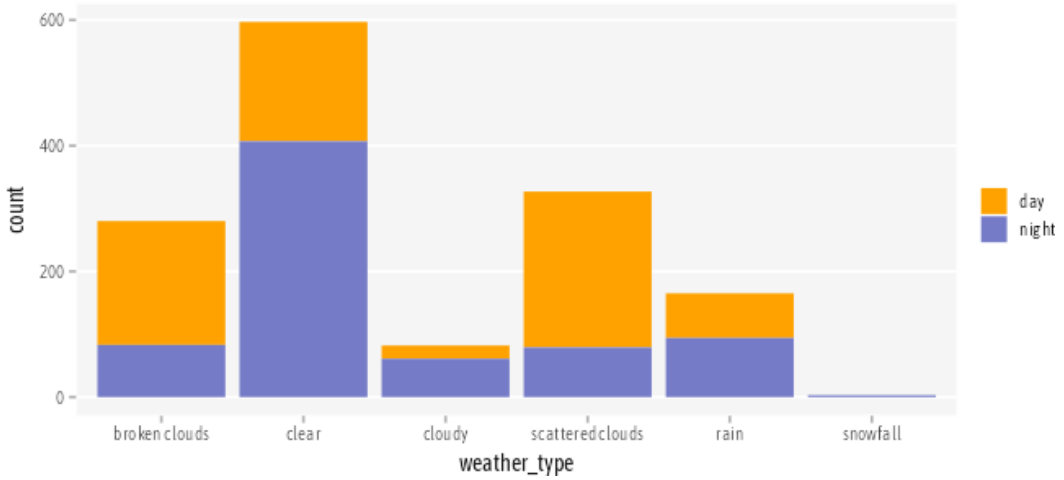



**FIGURE 7.2:** A stacked bar chart of the number of observations per weather type and time of the day, created with the predefined layers `geom_bar()` or `stat_count()`.

```
ggplot(bikes, aes(x = temp, y = humidity)) +
  geom_smooth() ## with `stat = "smooth"`

ggplot(bikes, aes(x = temp, y = humidity)) +
  stat_smooth() ## with `geom = "smooth"`
```

**geom_smooth() or stat_smooth()**

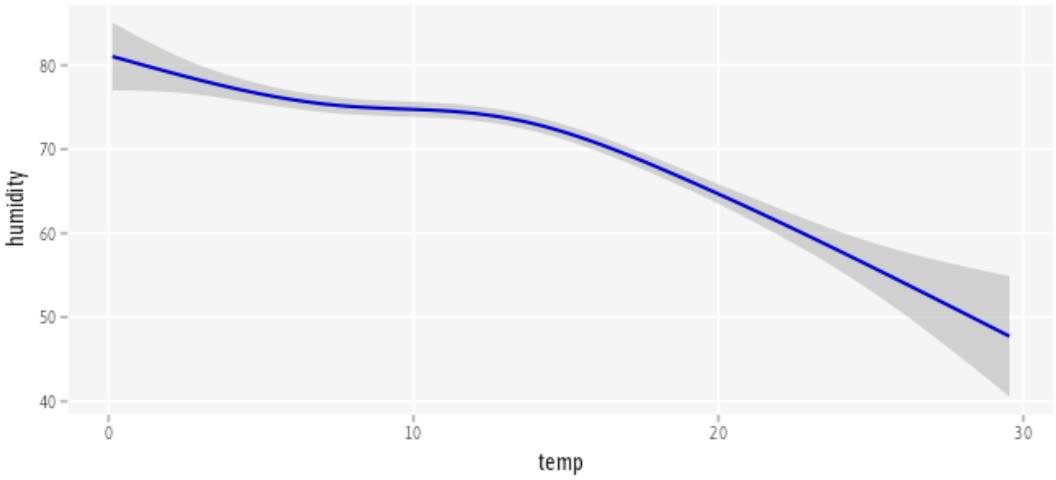



**FIGURE 7.3:** A conditional smoothing of humidity and temperature created with the predefined layers `geom_smooth()` or `stat_smooth()`.

### 7.1.2 Layer Variants

Some predefined layers just have subtle differences to accommodate different use cases. For example, you can connect points either in the order of the x variable as a *line graph* or *time series chart* (`geom_line()`) using the shortest path or as a *stairstep plot* (`geom_step()`). Alternatively, you can connect the observations in order they appear in the data object (`geom_path()`) which allows to draw *connected scatter plots* to visualize trajectories.

```
bikes_jul15 <- bikes |>
  filter(year == 2015, month == 7, day_night == "day") |>
  arrange(weather_type)

ggplot(bikes_jul15, aes(x = date, y = temp, color = date)) +
  geom_line()

ggplot(bikes_jul15, aes(x = date, y = temp, color = date)) +
  geom_step()

ggplot(bikes_jul15, aes(x = date, y = temp, color = date)) +
  geom_path()
```

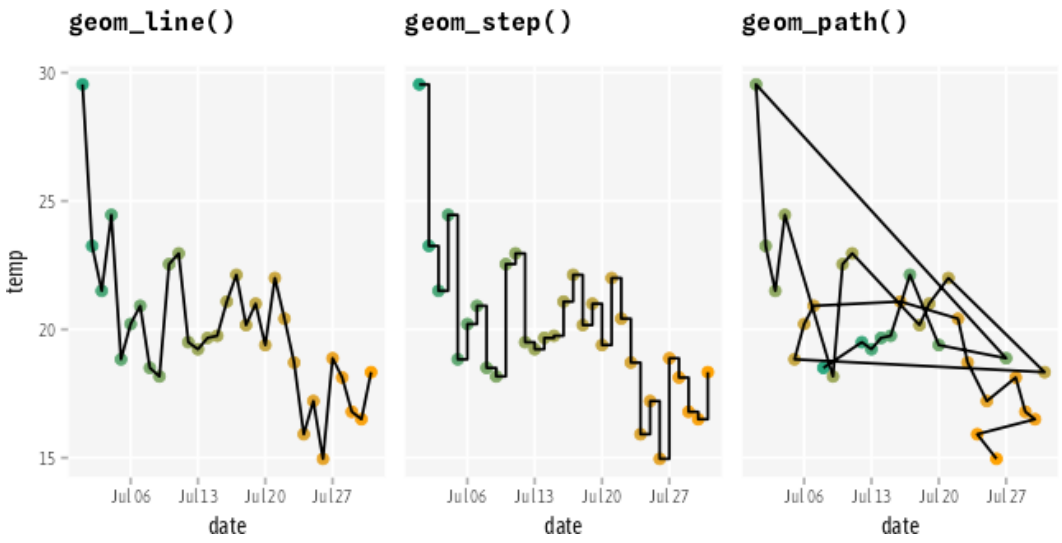



**FIGURE 7.4:** The three plots illustrate the different predefined geometrical layers that can be used to draw lines between observations. The colored dots illustrate the order in which the observations have been connected. While `geom_line()` (left) and `geom_step()` (middle) connect the points in order of the x axis, `geom_path()` (right) respects the order of the data and thus allows to move "back" on the x axis.

Other exemplary variants of geometrical layers are:

- **`geom_point()` and `geom_jitter()`** – When used with one qualitative positional variable, jittering allows to avoid overplotting by adding random noise along the qualitative axis (see next chapter).
- **`geom_bar()` and `geom_col()`** – `geom_bar()` creates bars that represent the count or fre-

quency of different categories in the data (only `x` or `y`) while `geom_col()` can be used to scale the height based on a quantitative variable (`x` and `y`).

- **`geom_density2d()` and `geom_density2d_filled()`** – Both layers compute smooth 2D density estimates, one representing the result as contour lines and the other as filled areas.

## 7.2 Changing Layer Defaults

The settings of the predefined layers are chosen with care. However, sometimes one needs to adjust the argument inputs of a layer to adjust its behavior. In the following we explore how we can make use of customizing the `geom`, `stat`, and `position`.

### 7.2.1 Positional Adjustments

While the stacked bars shown in Fig. 7.2 emphasize the total counts per weather type, a grouped bar chart simplifies comparison between groups. To change the default behavior of `geom_bar()` or `stat_count()`, which use `position = "stack"` as default, we set the `position` to `"dodge"`.

```
ggplot(bikes, aes(x = weather_type, fill = day_night)) +
  geom_bar(position = "dodge")
```

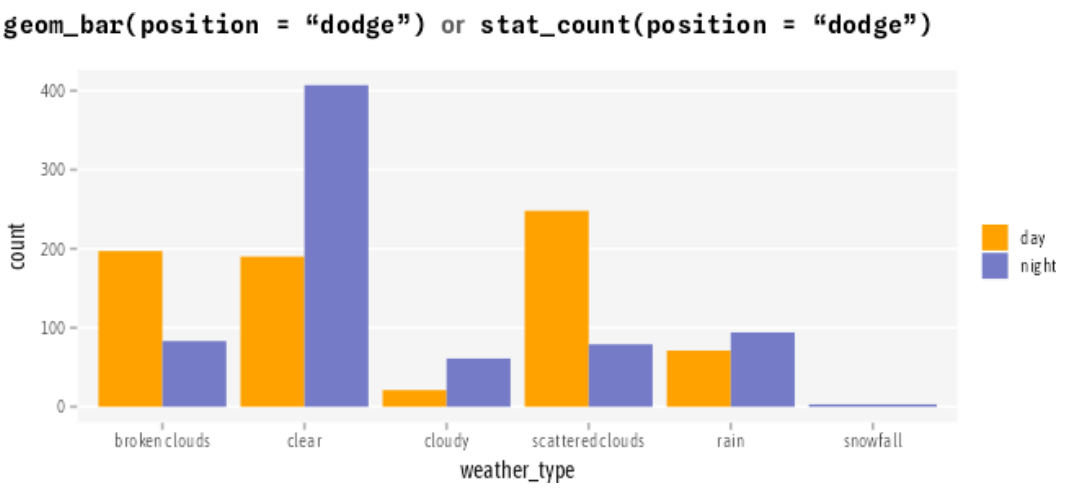
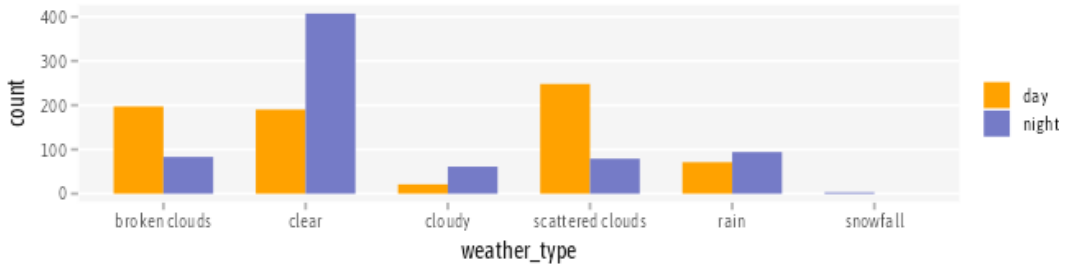



**FIGURE 7.5:** The bar chart of weather types per time of the day as dodged bars by setting the `position` to `"dodge"`, overwriting the default `"stack"`.

The quoted `"dodge"` is a shortcut to the function `position_dodge()`. If you want to adjust the default dodge behavior, you can specify arguments passed to the position function. For example, we might want to display all bars with the same width, independent of the presence of day and night (i.e. there is only one, very wide bar for snowfall as there are only day observations). We can fix this by passing `preserve = "single"` to the `position_dodge()` function. While this places the snowfall bar on the left, `position_dodge2()` centers it. At the same time, it adds some spacing between grouped bars when decreasing the bar width:

```
ggplot(bikes, aes(x = weather_type, fill = day_night)) +
  geom_bar(
    width = .7, ## more narrow bars
    position = position_dodge2(
      preserve = "single" ## consistent bar width
    )
  )
```

**position_dodge(preserve = single)**




**position_dodge2(preserve = single)**

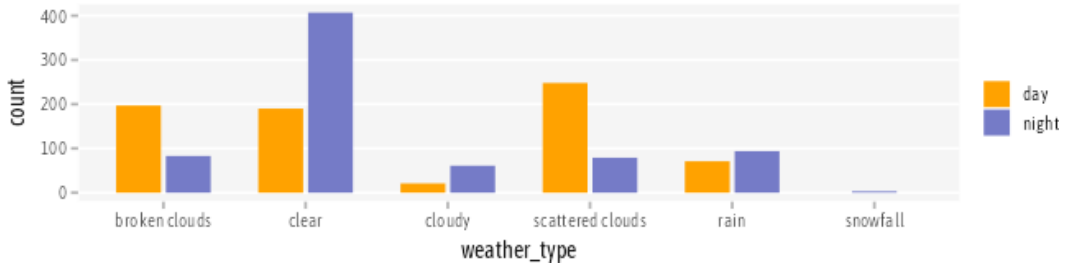



**FIGURE 7.6:** The bar chart of weather types per time of the day as dodged bars by setting the `position` to `"dodge"`, overwriting the default `"stack"`.

Another common use case of positional adjustments is the so-called *jittering* of points by adding random noise to the qualitative axis. It is a simple approach to deal with overplotting in case of dot strip plots:

```
ggplot(bikes, aes(x = season, y = count)) +
  geom_point(position = "jitter", alpha = .3)

ggplot(bikes, aes(x = season, y = count)) +
  geom_point(position = position_jitter(width = .3), alpha = .3)

ggplot(bikes, aes(x = season, y = count)) +
  geom_jitter(width = .3, alpha = .3)
```

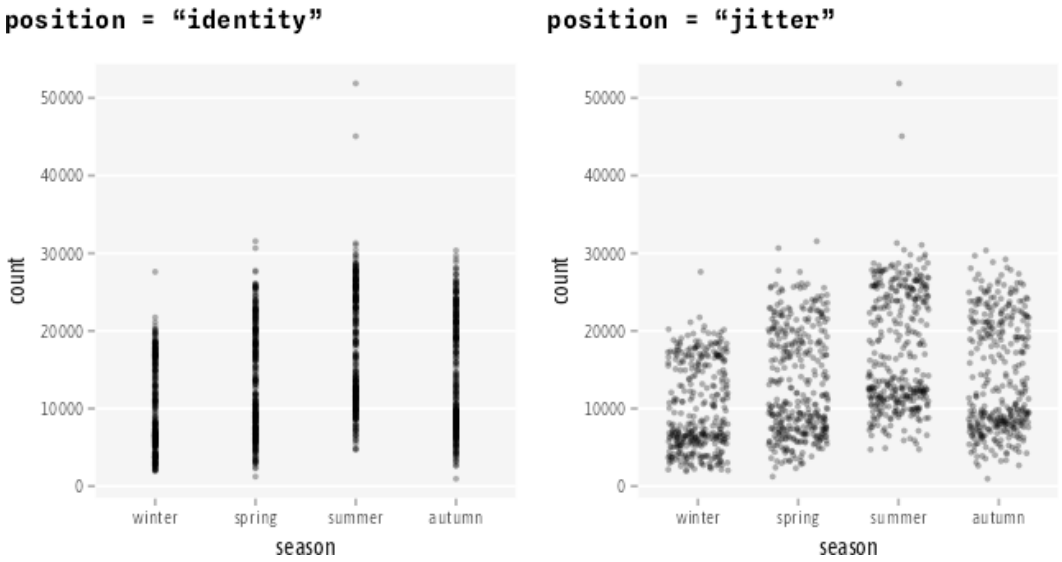



**FIGURE 7.7:** identity versus jitter

### 7.2.2 Modify Transformations and Geometries

We can also overwrite the default `stat` in geometrical layers or the default `geom` in statistical layers. For example, changing the `geom` in `stat_smooth()` to `pointrange` turns the smoothed line and ribbon in Fig. 7.3 into points with vertical lines representing the confidence intervals:

```
ggplot(bikes, aes(x = wind_speed, y = humidity)) +
  stat_smooth(geom = "pointrange")
```

Again, we could rewrite the code by using the respective geometrical layer `geom_pointrange()` and setting the default statistical transformation `"identity"` to `"smooth"`.

As you have already learned, `geom_point()` is usually used to draw scatter plots or dot strip plots showing the raw data. By replacing the default (non-)transformation `"identity"` with `"count"`, we can draw the overall count per group. In the following example, the points encode the counts per period of the day (`color = day_night`) and weather type (`x = weather_type`). The resulting plot is an alternative to the bar graph in Fig. 7.2 and Fig. 7.5 and could be as well created by overwriting the default `geom = "bar"` with `geom = "point"`. Note that in that case we also need to adjust the `position`, the counts are stacked otherwise!

```
ggplot(bikes, aes(x = weather_type, color = day_night)) +
  geom_point(stat = "count", size = 5)

## this leads to "wrong" results as counts are cumulative (stacked)
ggplot(bikes, aes(x = weather_type, color = day_night)) +
  stat_count(geom = "point", size = 5)
```

**stat_smooth(geom = “pointrange”) or geom_pointrange(stat = “smooth”)**

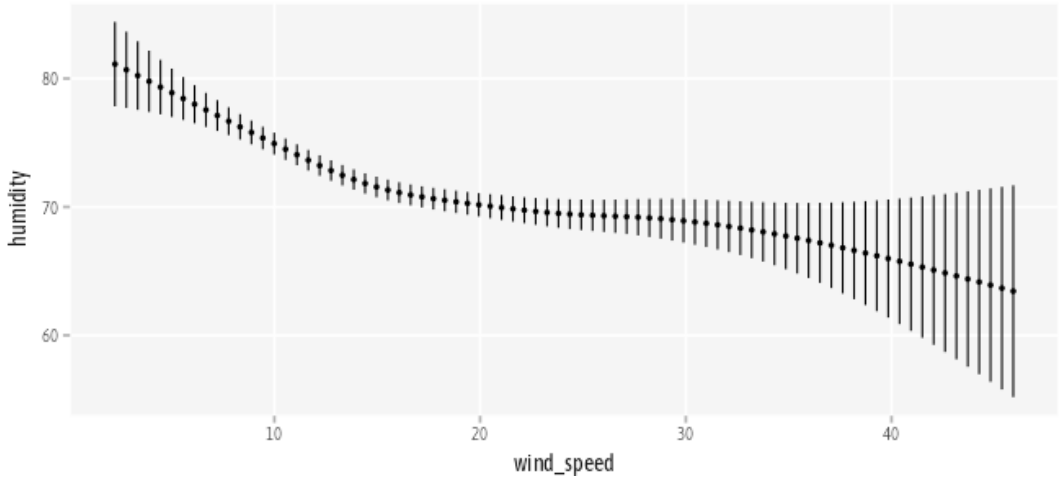



**FIGURE 7.8:** The conditional smoothing as so-called pointranges displaying the predicted mean as points and the confidence intervals as vertical lines by overwriting the default `geom` in `stat_smooth()` or using the respective geometrical layer with `stat = "smooth"`.

```
ggplot(bikes, aes(x = weather_type, color = day_night)) +
  stat_count(geom = "point", position = "identity", size = 5)
```

**geom_point(stat = “count”) or**
**stat_count(geom = “point”, position = “identity”)**

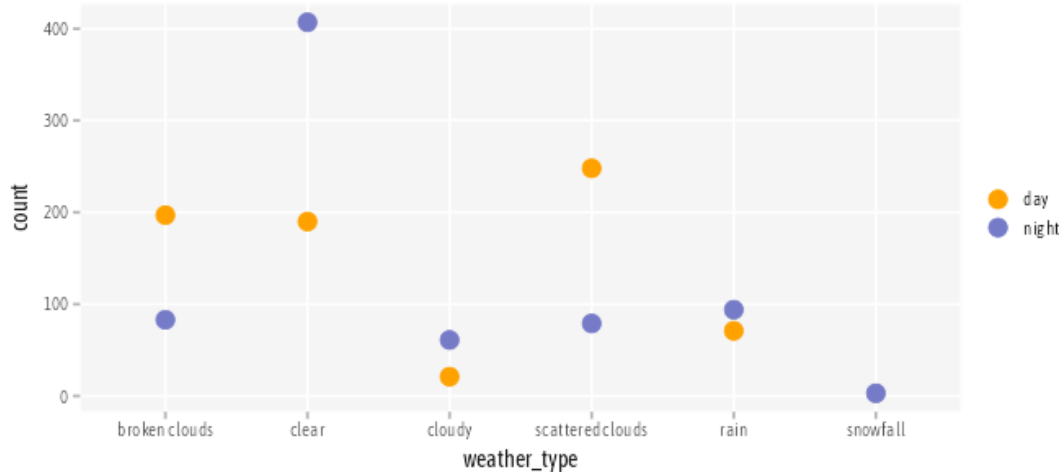



**FIGURE 7.9:** By overwriting the default stat method of `geom_point()` with `"count"`, we can alter its behaviour to calculate counts per group before plotting the data.

## 7.3 Positional Aesthetics

Many geometrical shapes work with one or two positional arguments, namely `x` and/or `y`, to represent your data. Some other geometries need more positional arguments to be drawn, such as corner points of rectangles or start and end coordinates of segments and curves. The data type of the positional variables influences the selection and behavior of geometrical layers.

### 7.3.1 Know Your Data Types

`geom_point()` is a versatile geometry that works with all combinations of quantitative and qualitative data mapped to `x` and `y`. When passing two qualitative variables, it creates a scatter plot as in Fig. 7.1. However, it can also be used to draw a dot strip plot when combining a quantitative and a qualitative variable (cf. Fig. 7.9), or to draw a heat bubble matrix when using two qualitative variables with the point size being mapped to a numeric variable.

Other geometries expect a certain combination of variables. Mapping the wrong data type to positional aesthetics can lead to unexpected behavior or may even return an error. To highlight the importance of passing the correct data types, let's consider two examples.

Box-and-whisker plots (or short "box plots") summarize the distribution of quantitative data, usually across multiple categories. By adding `geom_boxplot()`, **ggplot2** will calculate a box plot for each category if the type is `character`, `logical`, or `factor` but only a single box plot otherwise:

```
## creates four boxplots as `season` is a factor
ggplot(bikes, aes(x = season, y = count)) + geom_boxplot()

## creates two boxplots as `is_weekend` is logical
ggplot(bikes, aes(x = is_weekend, y = count)) + geom_boxplot()

## creates a single boxplot as `temp` is numeric
ggplot(bikes, aes(x = temp, y = count)) + geom_boxplot()
```

We can force **ggplot2** to group the quantitative variable into equal numerical ranges, so-called *bins*. One approach is using the `group` aesthetic. In our case, we can create bins with a width of 1°C by rounding the numbers to the next smallest integer:

```
ggplot(bikes, aes(x = temp, y = count, group = floor(temp))) +
  geom_boxplot(varwidth = TRUE) ## box width is proportional to sample size
```

We can also create groups by adjusting the variable mapped to x directly. To create bins with equal ranges, we wrap the variable inside `cut_interval()`. The corresponding `cut_number()` creates groups with equal number of observations (and potentially unequal ranges).

```
ggplot(bikes, aes(x = cut_interval(temp, 6), y = count)) +
  geom_boxplot(varwidth = TRUE)
```

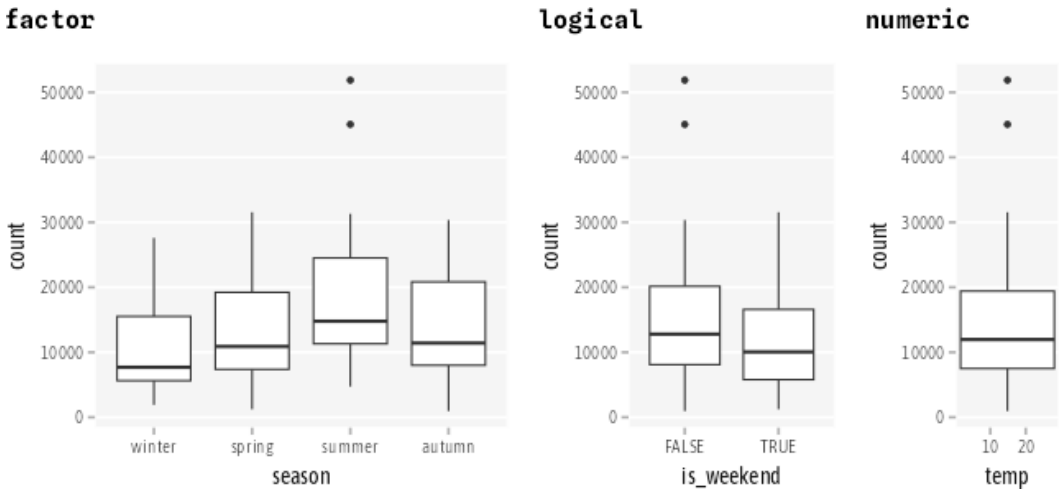



**FIGURE 7.10:** The number of indiviudal box-and-whisker plots created by `geom_boxplot()` depends on the type of variable mapped to `x`. In case of categorical variables such as `season` or `is_weekend`, the function creates as many boxplots as there are categories. For quantitative variables such as `temp`, only a single boxplot is calculated covering the complete numerical range.

```
ggplot(bikes, aes(x = cut_number(temp, 6), y = count)) +
  geom_boxplot(varwidth = TRUE)
```

Histograms show the distribution of numeric variables as binned bars. The corresponding `geom_histogram()` creates these *binned bars* with `stat = "bin"` and `geom = "bar"` for quantitative variable. But it fails when passing qualitative variables as calculation of bins across categories is not meaningful. In case of a qualitative x aesthetic, `geom_bar()` with the default `stat = "count"` creates the desired bar chart showing counts per category.

```
## works as `temp` is numeric
ggplot(bikes, aes(x = temp)) + geom_histogram()

## fails as `is_weekend` is categorical
ggplot(bikes, aes(x = is_weekend)) + geom_histogram()
# Error: `stat_bin()` requires a continuous x aesthetic

## create bars with discrete x aesthetics
ggplot(bikes, aes(x = is_weekend)) + geom_bar()
```

### 7.3.2 Multiple Positional Aesthetics

Other geometries need more than two positional aesthetics. `geom_linerange()` draws horizontal or vertical lines based on `xmin` and `xmax` or `ymin` and `ymax`, respectively. Let's load some stock data and show the change of opening versus closing price over time:

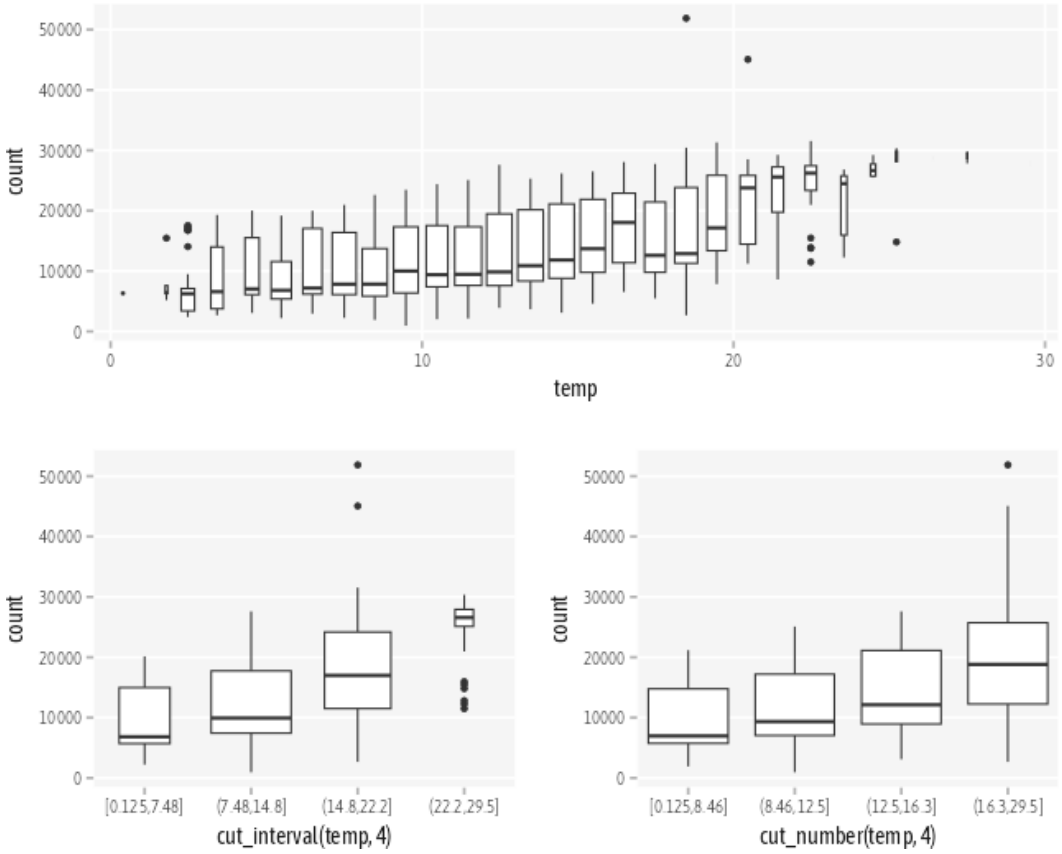



**FIGURE 7.11:** Three different ways to discretize the numeric x variable, here `temp`, to create multiple box plots. The upper row makes use of the `group` aesthetic in combination with `floor(temp)` to create bins of a width of 1°C. The lower row uses the `cut_*()` functions discretize the numeric variable into categorical. Here, we create four groups with equal range (bins of approx. 7.35°C, left) and equal number of observations (right), respectively. In all plots, the box widths are proportional to the number of observations to highlight the resulting group sizes of the different approaches.

```
coins <- readr::read_rds("./data/crypto_cleaned.rds") |>
  dplyr::filter(year == 2018, month %in% 5:6, currency == "litecoin")

ggplot(coins, aes(x = date, ymin = open, ymax = close)) +
  geom_linerange()
```

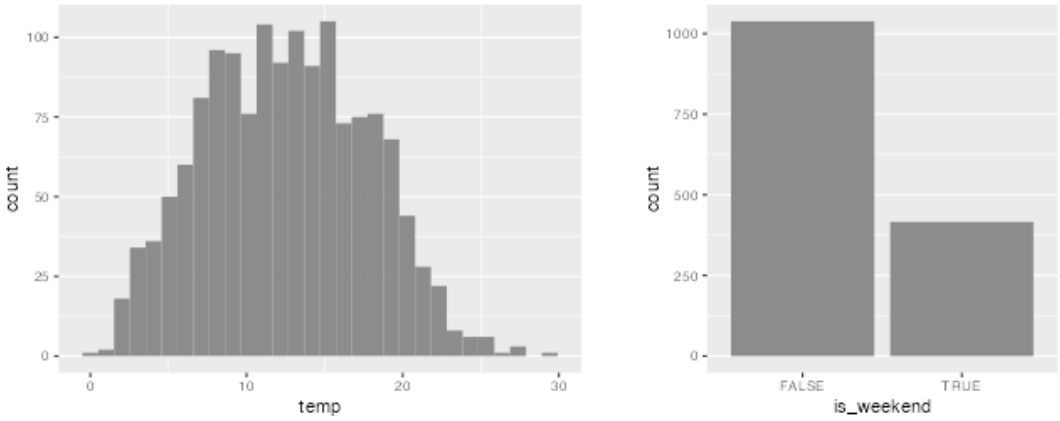
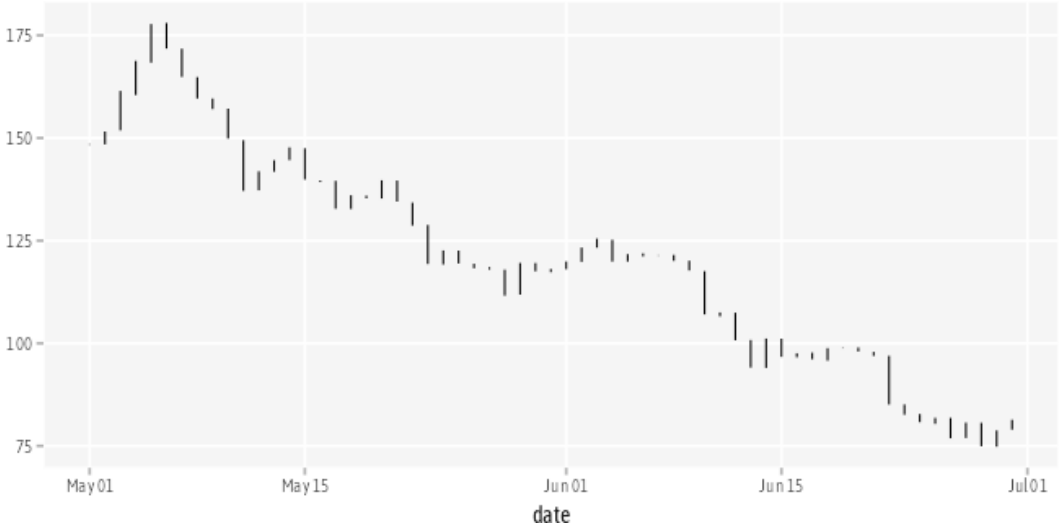
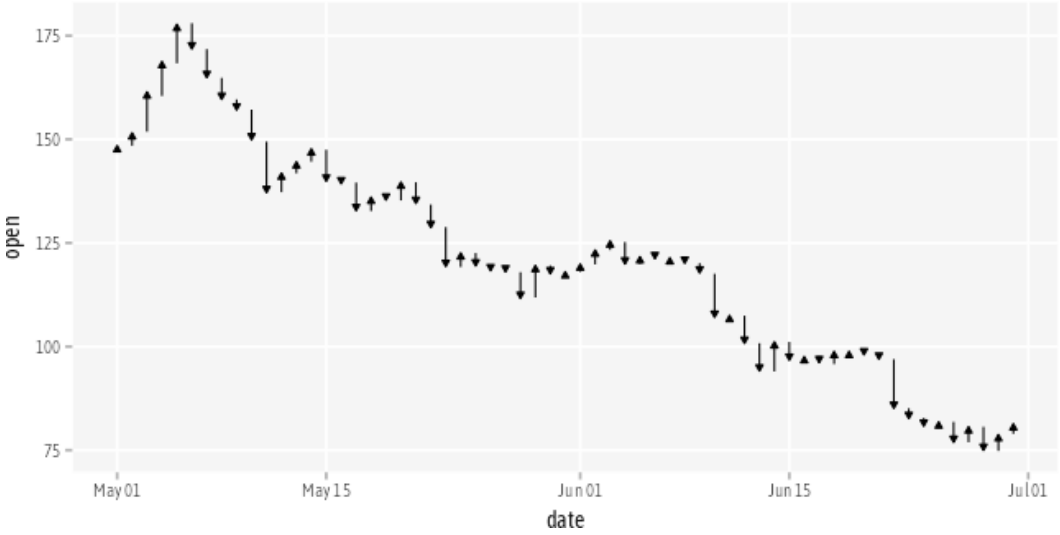



**FIGURE 7.12:** A histogram bins a quantitative variable, showing counts of obersvations per group (left). As qualitative data is already grouped, a regular bar chart is used to display counts per category (right).




That's great but in addition I would like to indicate the direction of change in daily prices. For that we can use `geom_segment()` which allows to add arrow heads to straight lines. The segments are not restricted to vertical or horizontal and thus take four positional arguments: `x` and `xend` as well as `y` and `yend`. For our use case, we map the same variable, namely `date`, to both x coordinates. We then can define the arrow to be drawn by passing the `arrow()` function to the `arrow` argument inside `geom_segement()`.

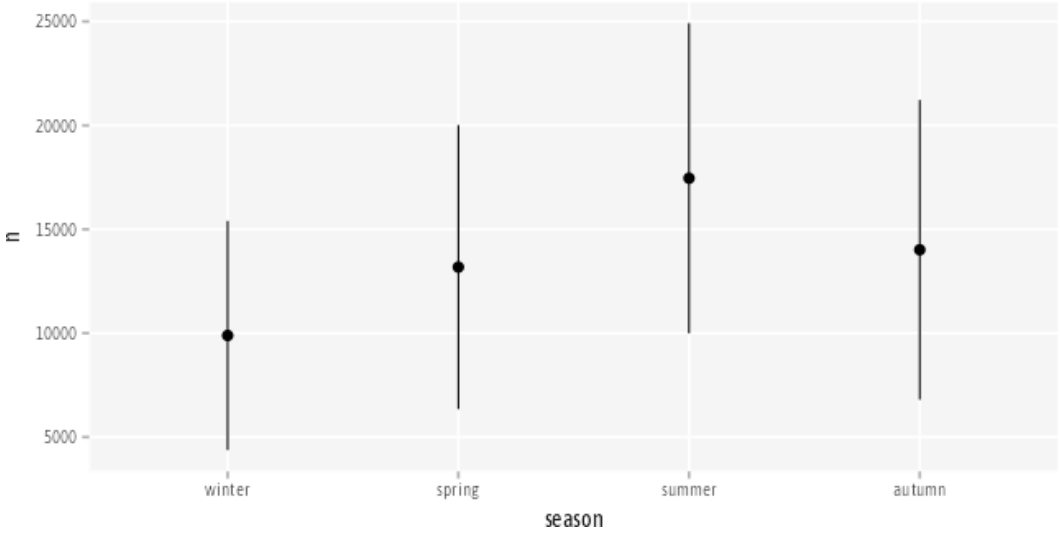
```
ggplot(coins, aes(x = date, xend = date,
                  y = open, yend = close)) +
  geom_segment(
    arrow = arrow(
      type = "closed", ## draw an triangle as arrow head
      length = unit(.3, "lines")) ## size of the arrow head
  )
```




The geom `pointrange` (that we have also used in the smoothing example) needs four positional aesthetics as well: in addition to the `x` and `y` positions, we have to specify the range as well as either `xmin` and `xmax` or `ymin` and `ymax`.

```
library(dplyr)
bikes_season <-
  bikes |>
  group_by(season) |>
  summarize(n = mean(count), sd = sd(count))

ggplot(bikes_season,
       aes(x = season, y = n, ymin = n - sd, ymax = n + sd)) +
  geom_pointrange()
```



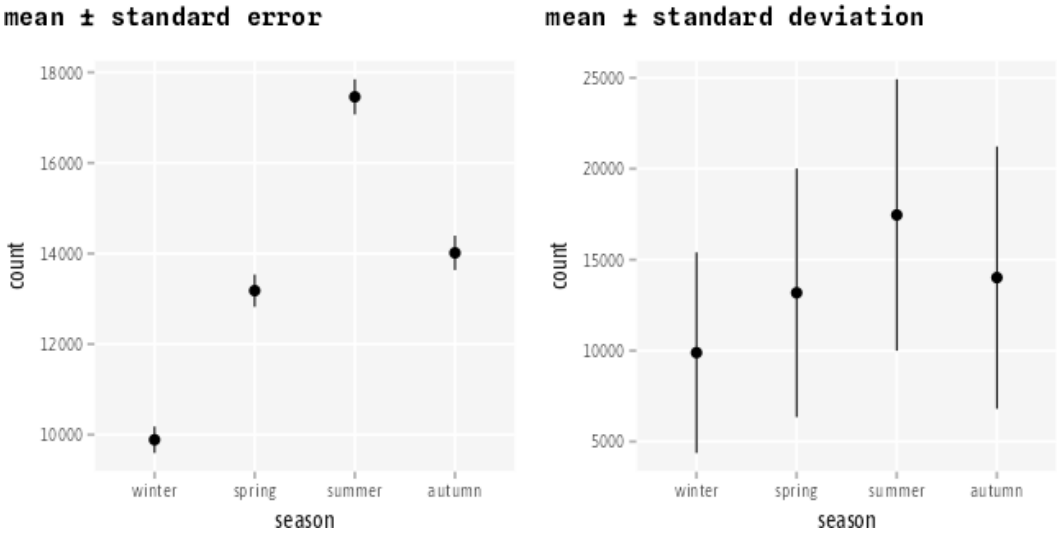
## 7.4 Statistical Summaries

You can also use `stat_summary()` and let **ggplot2** do the transformation of the raw data. By default, `stat_summary()` uses a `pointrange` geometry with the three positional aesthetics representing mean ± standard error.

```
ggplot(bikes, aes(x = season, y = count)) +
  stat_summary()
```

Passing a set of functions to the arguments `fun`, `fun.min` and `fun.max` we can modify the range:

```
ggplot(bikes, aes(x = season, y = count)) +
  stat_summary(
    fun = "mean",
    fun.min = function(y) mean(y) - sd(y),
    fun.max = function(y) mean(y) + sd(y)
  )
```




Alternatively, a function that returns the required aesthetics can be specified as input for `fun.data`. As the function `mean_sdl()` computes the mean ± 2 times the standard deviation, we have to adjust the function argument `mult` as well. Either we pass it to the `fun.args` argument or specify it as function argument directly.

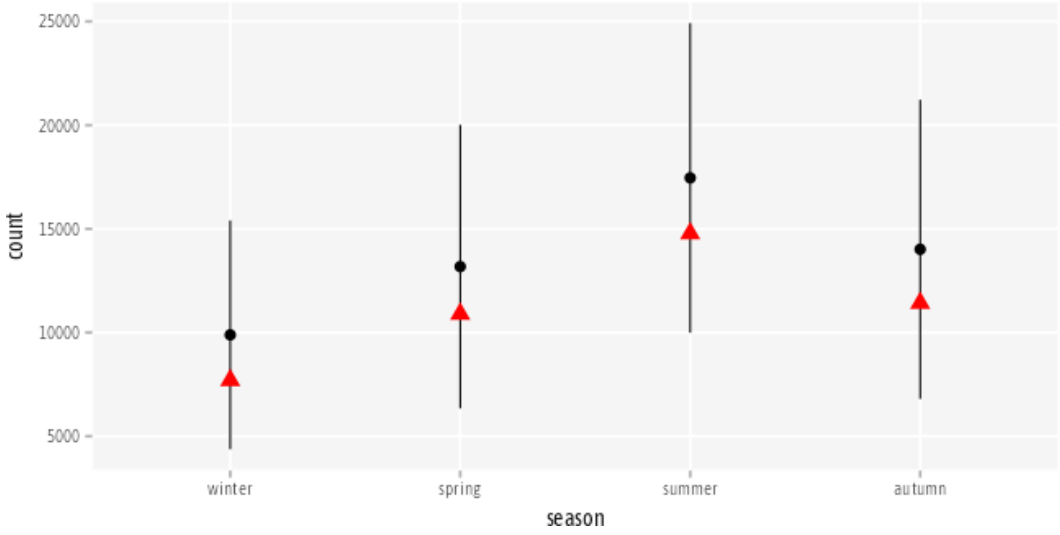
```
ggplot(bikes, aes(x = season, y = count)) +
  stat_summary(
    fun.data = "mean_sdl",
    fun.args = list(mult = 1)
  )
```

```
ggplot(bikes, aes(x = season, y = count)) +
  stat_summary(
    fun.data = function(y) mean_sdl(y, mult = 1)
  )
```

This seems to hard to remember? The nice thing is that all solutions are valid, just pick your favorite (and forget about the rest as long as there is no particular reason to use it).

Similarly, we can also overwrite the default geometry, for example to display the median count for each season by adding a red triangle:

```
ggplot(bikes, aes(x = season, y = count)) +
  stat_summary(
    fun.data = function(y) mean_sdl(y, mult = 1)
  ) +
  stat_summary(
    geom = "point",
    fun = "median",
    color = "red",
    shape = 17,
    size = 4
  )
```



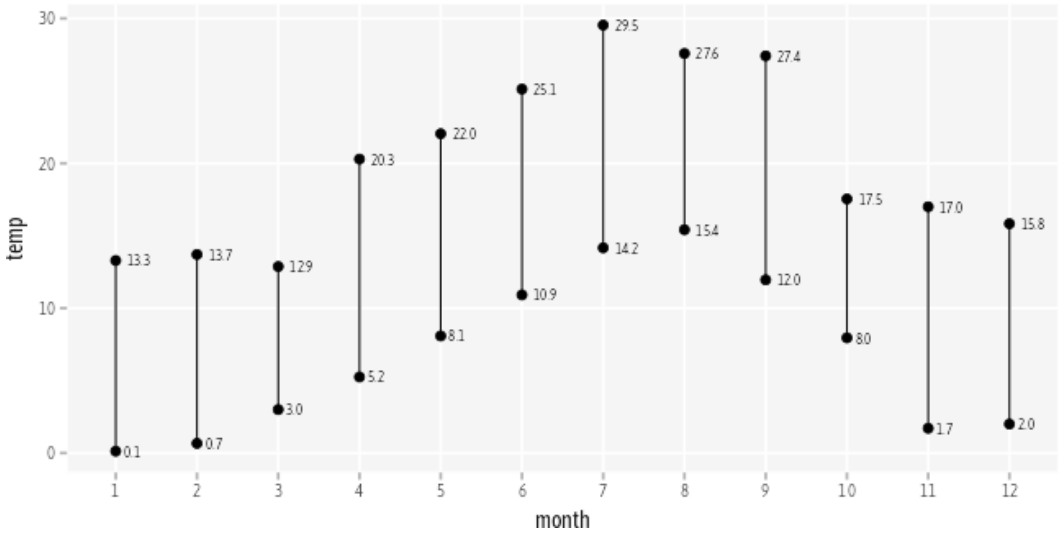

The `stat_*()` functions and especially `stat_summary` are incredible powerful and often a neat way to create all kind of chart types. Let's use a combination of summaries to draw a dumbbell chart of temperature ranges across months.

```
ggplot(bikes, aes(y = month, x = temp)) +
  stat_summary(
    geom = "linerange",
    fun.min = "min",
    fun.max = "max"
```

```
  ) +
  stat_summary(
    geom = "point",
    fun = "range",
    size = 3
  )
```

We can also use `stat_summary()` to add the actual minimum and maximum values to our plot. The geom `text` in combination with the `label` aesthetic adds data-related labels.

```
ggplot(bikes, aes(x = month, y = temp)) +
  stat_summary(
    geom = "linerange",
    fun.min = "min",
    fun.max = "max"
  ) +
  stat_summary(
    geom = "point",
    fun = "range",
    size = 3
  ) +
  stat_summary(
    geom = "text",
    fun = "range",
    aes(label = after_stat(y)),
    hjust = -.5
  )
```



# 8

## *Working with Colors*

[WIP]

# 9

## *Working with Text*

[WIP]

# 10

# *Working with Themes*

[WIP]

## 10.1 Create Your Own Theme

```
theme_custom <- function(base_size = 14, base_family = "Asap SemiCondensed",
                         base_line_size = base_size/25, base_rect_size = base_size/25,
                         color_plot = "grey96", color_panel = "white") {
    theme_minimal(
      base_size = base_size, base_family = base_family,
      base_line_size = base_line_size, base_rect_size = base_rect_size
    ) %+replace%
    theme(
      panel.grid.minor = element_blank(),
      panel.grid.major = element_line(color = color_plot),
      panel.background = element_rect(fill = color_panel, color = NA),
      plot.background = element_rect(fill = color_plot, color = NA),
      plot.title = element_text(size = rel(1.2), face = "bold", hjust = 0,
                                margin = margin(b = rel(15))),
      plot.subtitle = element_text(size = rel(1), hjust = 0),
      plot.caption = element_text(color = "grey30", size = rel(.7), hjust = 0,
                                margin = margin(t = rel(20))),
      plot.title.position = "plot",
      plot.caption.position = "plot",
      axis.text = element_text(color = "grey40", size = rel(.8)),
      complete = FALSE
  )
}
```

```
g + theme_custom()
```

```
g +
  theme_custom(
    base_size = 16, base_family = "Spline Sans",
    color_plot = "white", color_panel = "antiquewhite"
```

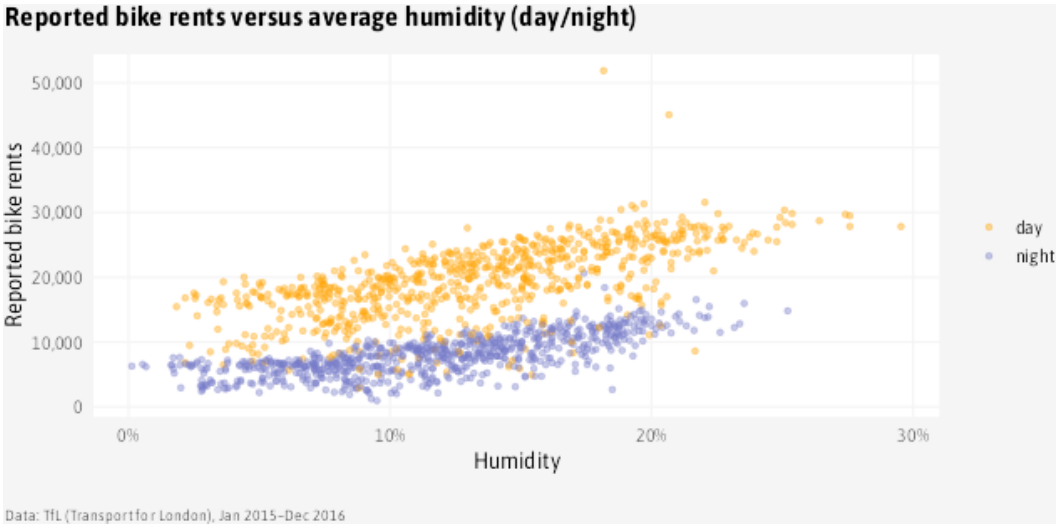



**FIGURE 10.1:** Applying our custom theme to the ggplot object from before.

```
  ) +
  theme(plot.title = element_text(color = "goldenrod"))
```

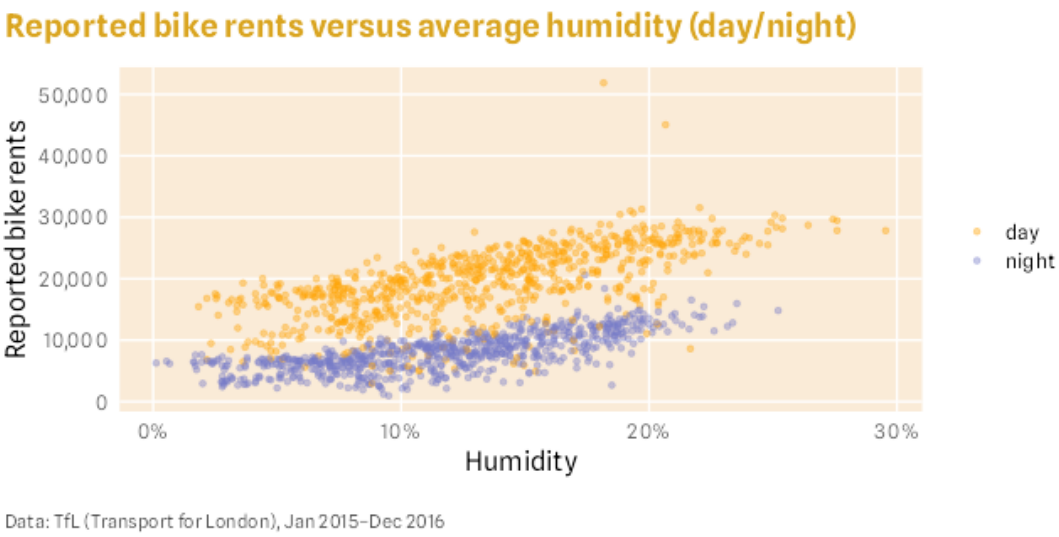



**FIGURE 10.2:** You can still overwrite the main settings and adjust single elements when using the custom theme.

# Part III

# Advanced Walk-Through Examples

# 11

*Overview*

# A

## *More to Say*

Yeah! I have finished my book, but I have more to say about some topics. Let me explain them in this appendix.

To know more about **bookdown**, see https://bookdown.org.

# Bibliography


Bertin, J. (1983). *Semiology of Graphics*. University of Wisconsin Press, Madison, Wisconsin, United States.

Cairo, A. (2021). Orthodoxy and eccentricity. In *Data Sketches by Nadieh Bremer and Shirley Wu*, pages 12–13. Chapman and Hall/CRC, Boca Raton, Florida, United States. ISBN 978-036-70-0012-7.

Hughes, E. (2022). *camcorder: Record Your Plot History*. R package version 0.1.0.

Koponen, J. and Hildén, J. (2019). *Data visualization handbook*. Aalto ARTS Books, Espoo, Finland, 1st edition. ISBN 978-952-60-7449-8.

Kosara, R. (2007). Visualization criticism - the missing link between information visualization and art. In *2007 11th International Conference Information Visualization (IV '07)*, pages 631–636.

Sciaini, M., Fritsch, M., Scherer, C., and Simpkins, C. E. (2018). Nlmr and landscapetools: An integrated environment for simulating and modifying neutral landscape models in r. *Methods in Ecology and Evolution*, 9(11):2240–2248.

Wickham, H. (2006). An introduction to ggplot: An implementation of the grammar of graphics in r. *JStatistics*, pages 1–8.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H. (2022). *forcats: Tools for Working with Categorical Variables (Factors)*. R package version 0.5.2.

Wickham, H., François, R., Henry, L., and Müller, K. (2022a). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.10.

Wickham, H. and Girlich, M. (2022). *tidyr: Tidy Messy Data*. R package version 1.2.1.

Wickham, H., Hester, J., and Bryan, J. (2022b). *readr: Read Rectangular Text Data*. R package version 2.1.3.

Wilkinson, L. (2005). *The Grammar of Graphics*. Springer Science+Business Media, Berlin/Heidelberg, Germany, 2nd edition. ISBN 978-0-387-28695-2.

Xie, Y. (2015). *Dynamic documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, United States, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2023). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.34.

# *Index*