Cédric Scherer

Graphic Design with ggplot2

Create Beautiful Data Visualizations in R

To my son,

without whom I should have finished this book two years earlier

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 base plots in R, so I immediately fell in love with the idea and implementation of **ggplot2**'s *Grammar of Graphics*. 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 of custom fonts and colors, 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.

Today, on a sunny day in July 2021, this tutorial serves as the starting point for the book you hold in your hands. I hope you enjoy it as much as I enjoyed learning and sharing **ggplot2** wizardry!

[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/

Why read this book

Often, people that use common graphic design and charting tools or have basic experience 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:

- How do I get started with the code? → Chapter 2
- I have no idea how **ggplot2** actually works and need a quick introduction → Chapter 3

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])
- A set of R packages installed:
  - **tidyverse**[7] that includes **ggplot2**[8]
  - **ggforce**[9]
  - **ggrepel**[10]
  - **ggtext**[11]
  - **magick**[12]
  - **patchwork**[13]
  - **ragg**[14]
  - **rnaturalearth**[15]
  - **scico**[16]
  - **sf**[17]

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

```
install.packages(c(
  "tidyverse", "ggforce", "ggtext", "magick", "patchwork",
  "ragg", "rnaturalearth", "scico", "sf"
))
```

Software information and conventions

The book was written with the **knitr** package (Xie, 2015) and the **bookdown** package (Xie, 2021) with the following setup:

```
## R version 4.1.0 (2021-05-18)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19043)
##
## Locale:
##   LC_COLLATE=German_Germany.1252
##   LC_CTYPE=German_Germany.1252
##   LC_MONETARY=German_Germany.1252
##   LC_NUMERIC=C
```

[4] https://cloud.r-project.org/
[5] https://rstudio.com/products/rstudio/download/#download
[6] https://code.visualstudio.com/download
[7] https://www.tidyverse.org/
[8] https://ggplot2.tidyverse.org/
[9] https://ggforce.data-imaginist.com/
[10] https://ggrepel.slowkow.com/
[11] https://wilkelab.org/ggtext/
[12] https://docs.ropensci.org/magick/
[13] https://patchwork.data-imaginist.com/
[14] https://ragg.r-lib.org/
[15] https://docs.ropensci.org/rnaturalearth/
[16] https://github.com/thomasp85/scico
[17] https://r-spatial.github.io/sf/

```
##   LC_TIME=German_Germany.1252
## system code page: 65001
##
## Package version:
##   base64enc_0.1.3   bookdown_0.24
##   compiler_4.1.0    cpp11_0.4.2
##   digest_0.6.29     evaluate_0.14
##   fastmap_1.1.0     glue_1.4.2
##   graphics_4.1.0    grDevices_4.1.0
##   highr_0.9         htmltools_0.5.2
##   jquerylib_0.1.4   jsonlite_1.7.2
##   knitr_1.36        magrittr_2.0.1
##   methods_4.1.0     ragg_1.1.3
##   rlang_0.4.12      rmarkdown_2.11
##   rstudioapi_0.13   stats_4.1.0
##   stringi_1.7.5     stringr_1.4.0
##   systemfonts_1.0.3 textshaping_0.3.6
##   tinytex_0.35      tools_4.1.0
##   utils_4.1.0       xfun_0.27
##   yaml_2.2.1
```

Package names are in bold text (e.g. **ggplot2**), and inline code and file names are formatted in a monospaced typewriter font (e.g. `read_csv("data.csv")`). Function names are followed by parentheses (e.g. `ggplot2::ggplot()`). Notes are formatted as coloured, italic text.

Acknowledgments

Thanks to David Grubbs, Alberto Cairo, Emily Riederer, Oscar Baruffa, and Malcolm Barrett for all your constructive feedback.

Cédric Scherer
Berlin, Germany

About the Author

Dr Cédric Scherer is a graduated computational ecologist with a passion for good design. In 2020, he combined his expertise in analyzing and visualizing large data sets in R with his interest in design and his perfectionism to become an independent data designer and data visualization consultant.

Cédric has created visualizations across all disciplines, purposes, and styles. Due to regular participation to social data challenges, he is now well known for complex and visually appealing figures, entirely made with ggplot2, that look as they have been created with a vector design tool.

As a data visualization specialist, he does rarely create dashboards but acts as a consultant and designer improving chart and design choices and workshop coach teaching data visualization principles and courses on R and **ggplot2**. He also uses R and the **tidyverse** packages to automate data analyses and plot generation, following the philosophy of a reproducible workflow.

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. In the best case, a well-designed data visualization helps to amplify cognition, facilitate insights, discover, spark curiosity, explain, 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 the data, the chart type, and the design. As a *reader*, identifying the purpose helps rating the quality of the presentation. Some people like to think that there is a single best approach

[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.

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*. 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 your 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.

1.2 Coding Visualizations

As data visualizations involve the quantitative representation of variables, an environment that allows to handle, wrangle and quantify data is preferential. Classical design software

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

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 chart types (e.g. DataWrapper, Flourish, RAWgraphs), often also with beautiful and very sensible defaults, such chart builder do not provide full flexibility. Furthermore, the combination of a suite of tools might be …

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 but at the same time allow users to create almost any visualization one can think of. 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 an 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.
- Various extension packages add missing functionality.
- Script-based workflows instead of *point–and–click* approaches allow for reproducibility—

which means you can simply (in theory) 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 modification 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 graphic design tools.

2

Get Started

2.1 The Data

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.

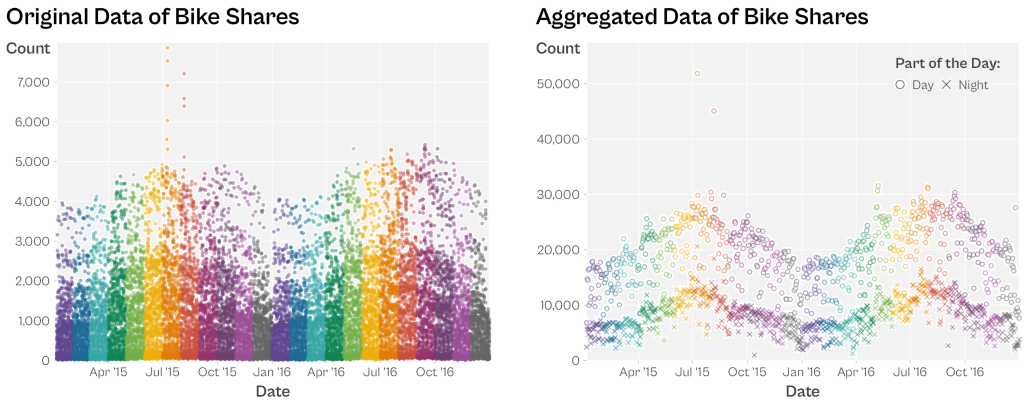


FIGURE 2.1: The original and aggregated data sets in direct comparison: counts of bike shares registered by TfL over time with month encoded by colour. The left panel shows counts for every hour of the day, while in the right panel the hourly data was aggregated into two periods of the day (day and night).

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) (2.1. 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 2.1 and figure 2.2.

COMMENT: Decide on a version to provide and overview of the variables as table or list.

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

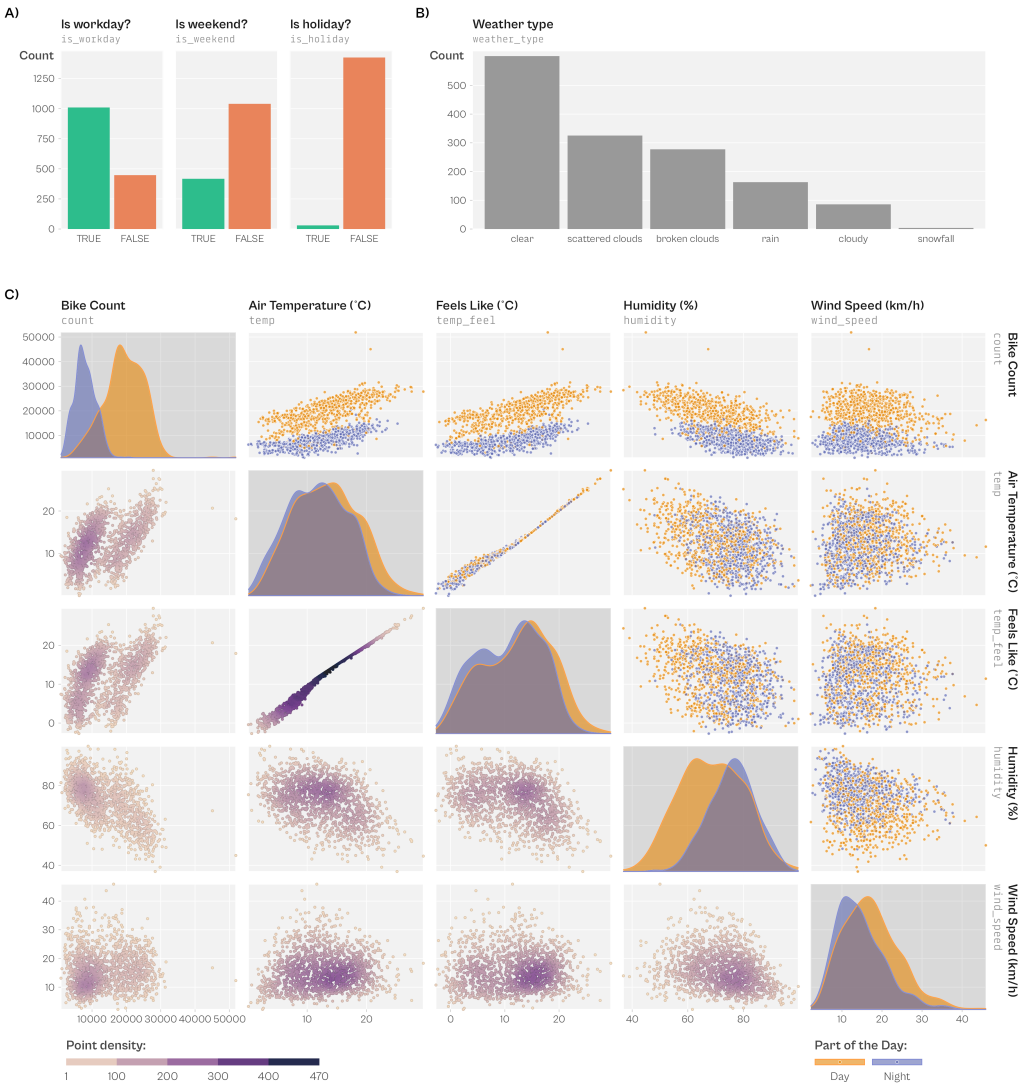


FIGURE 2.2: Overview of the distribution of the boolean variables `is_workday`, `is_weekend`, and `is_holiday` (A), the categorical variable `weather_type` (B), and the continuous variables `count`, `temp`, `temp_feel`, `humidity`, and `wind_speed` (C) of the cleaned and aggregated bike sharing data 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 2.1: 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

2.2 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].

2.2.1 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 traditional tabular data as .txt or .csv one can use the **readr** package.

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 it in variables called `url_data` and `bikes` by using the *assignment arrow* `<-`. The `col_types` argument allows to specify the column types, e.g. `i` are integer values, `f` encodes factors, and `l` turns a column into logical, boolean variable that is either `TRUE` or `FALSE`.

[3]Note that the **ggplot2** package itself belongs to the **tidyverse** as well.

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

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 = "Dcfffilllddddfc")
```

2.2.2 Inspect 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
## # ... with 1,444 more rows, and 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 data frame (class `data.frame`). On the top of the output, you can directly see that our data set consists of 14 variables (columns) frame with 1454 observations (rows). Also, it will show you the first ten rows. Alternatively you can inspect the data with the help of `str()` or `tibble::glimpse()` to print a transposed version.

If you have looked carefully, you may have noticed that a tibble prints also the class of each column, e.g. `<chr>`. We have specified the class of the columns manually when importing the data; if not specified, `readr::read_csv()` as most other import functions guess the class. Thus, it is always worth to check the classes of the columns. 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 (i.e. class integer, numeric, date) or qualitative (i.e. class character, factor, boolean). By using the `$` symbol one can access single columns of a data frame, e.g. `bikes$date`.

```
class(bikes)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"
## [4] "data.frame"
```

```
class(bikes$date)
```

```
## [1] "Date"
```

```
class(bikes$count)
```

```
## [1] "integer"
```

```
class(bikes$temp)
```

```
## [1] "numeric"
```

```
class(bikes$day_night)
```

```
## [1] "character"
```

```
class(bikes$weather_type)
```

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

```
class(bikes$is_holiday)
```

```
## [1] "logical"
```

To explore individual variables, the following functions are useful:

- `min()`, `max()`, `range()` to extract extreme values
- `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$weather_type)
```

```
## [1] broken clouds    clear            cloudy
## [4] scattered clouds rain             snowfall
## 6 Levels: broken clouds clear ... snowfall
```

```
length(unique(bikes$year))
```

```
## [1] 2
```

2.2.3 Data types

2.2.4 Data wrangling

2.3 Working with Rmarkdown

3

The ggplot2 Package

ADD SHORT HISTORY OF GGPLOT2

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[1]. You provide the data, tell **ggplot2** how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

A ggplot is built up from a few components:

1. **Data**:
   The raw data that you want to plot.
2. **Aesthetics** `aes()`:
   Aesthetics that variables are mapped to such as position, color, size, shape, and transparency
3. **Layers:**
   The geometric shapes (`geom_`) that will represent the data or statistical transformation ('stat_')of the data, such as quantiles, fitted curves, and counts.
4. **Scales** `scale_`:
   Maps between the data and the aesthetic dimensions, such as data range to positional aesthetics or qualitative or quantitative values to colors.
5. **Coordinate system** `coord_`:
   The transformation used for mapping data coordinates into the plane of the graphic.
6. **Facets** `facet_`:
   The arrangement of the data into a grid of plots (also known as *trellis* or *lattice plot*, or simply *small multiples*).
7. **Visual themes** `theme()`:
   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[2].

[1] https://link.springer.com/chapter/10.1007/978-3-642-21551-3_13
[2] https://ggplot2-book.org/introduction.html

A basic ggplot needs three things that you have to specify: the *data*, *aesthetics*, and a *geometry*. All other components can be added to customize the graphic.

3.1 A Basic ggplot

First, to be able to use the functionality of **ggplot2** we have to load the package (which we can also load via the tidyverse package collection[3]):

```
#library(tidyverse)
library(ggplot2)
```

The syntax of **ggplot2** is very different from plotting functionality of provided by base R. We always start to define a plotting object by calling `ggplot(data = df)` which just tells **ggplot2** that we are going to work with that data. In most cases, you might want to plot two variables—one on the x and one on the y axis. These are *positional aesthetics* and thus we add `aes(x = var1, y = var2)` to the `ggplot()` call (yes, the `aes()` stands for aesthetics). However, there are also cases where one has to specify only one or even three or more variables.

We specify the data outside `aes()` and add the variables that ggplot maps the aesthetics to inside `aes()`.

Here, we map the variable `date` to the x position and the variable `temp` to the y position. Later, we will also map variables to all kind of other aesthetics such as color, size, and shape.

```
ggplot(data = bikes, mapping = aes(x = date, y = count))
```

Only a panel is created when running this. Why? This is because **ggplot2** does not know *how* we want to plot that data—we still need to provide a geometry!

ggplot2 allows you to store the current `ggobject` in a variable of your choice by assigning it to a variable, in our case called `g`. You can extend this `ggobject` later by adding other layers, either all at once or by assigning it to the same or another variable.

Thanks to *implicit matching* we can rewrite the code as follows: `ggplot(bikes, aes(date, count))`. But be aware that the order matters! I suggest to use a mixture if you feel confident to do so.

```
ggplot(bikes, aes(x = date, y = count))
```

Omitting the arguments `data` and `mapping` saves you a ton of typing when creating dozens to hundreds ggplot's per day but being specific about `x` and `y` is a good idea. This is the syntax I am using throughout the book.

There are many, many different geometries (called *geoms* because each function usually starts with `geom_`) one can add to a ggplot by default (see here[4] for a full list) and even

[3] https://www.tidyverse.org/
[4] https://ggplot2.tidyverse.org/reference/

more provided by extension packages (see here[5] for a collection of extension packages). Let's tell **ggplot2** which style we want to use, for example by adding `geom_point()` to create a scatter plot:

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

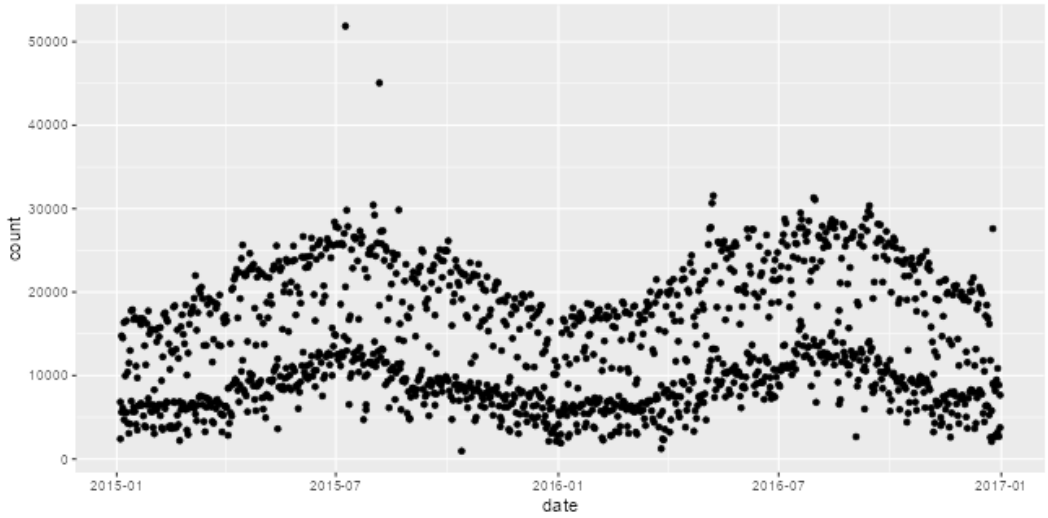


FIGURE 3.1: A default scatter plot of temperature measured in Chicago created with the **ggplot2** package.

One can also combine several geometric layers—and this is where the magic and fun starts!

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

That's it for now about geometries. No worries, we are going to learn several plot types in Chapter `charts`.

3.2 Change Properties of Geometries

Within the `geom_*` command, you already can manipulate visual aesthetics such as the color, shape, and size of your points. Let's turn all points to large fire-red diamonds!

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick", shape = "diamond", size = 2)
```

[5] https://exts.ggplot2.tidyverse.org/

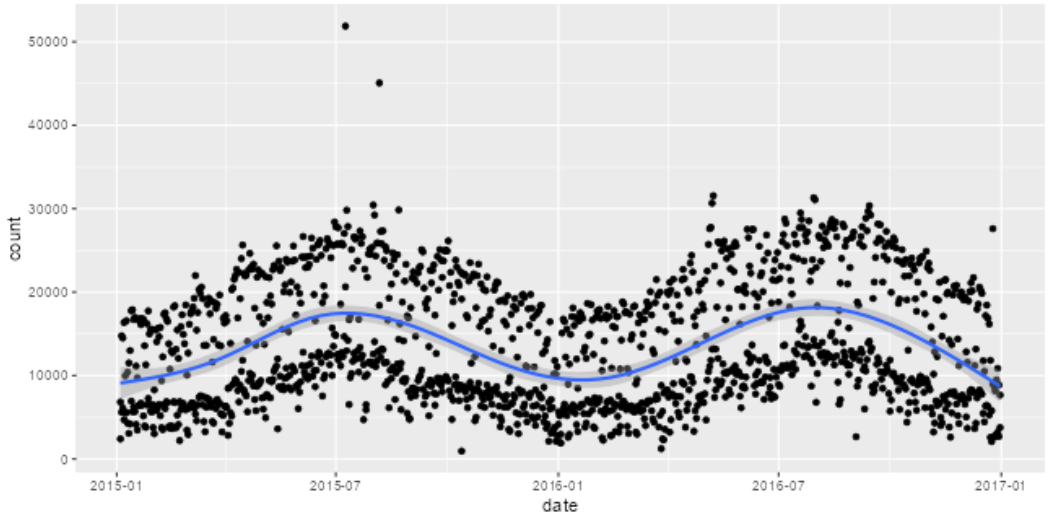


FIGURE 3.2: Again, the same data, now shown as a connected scatterplot as a combination of points and lines.

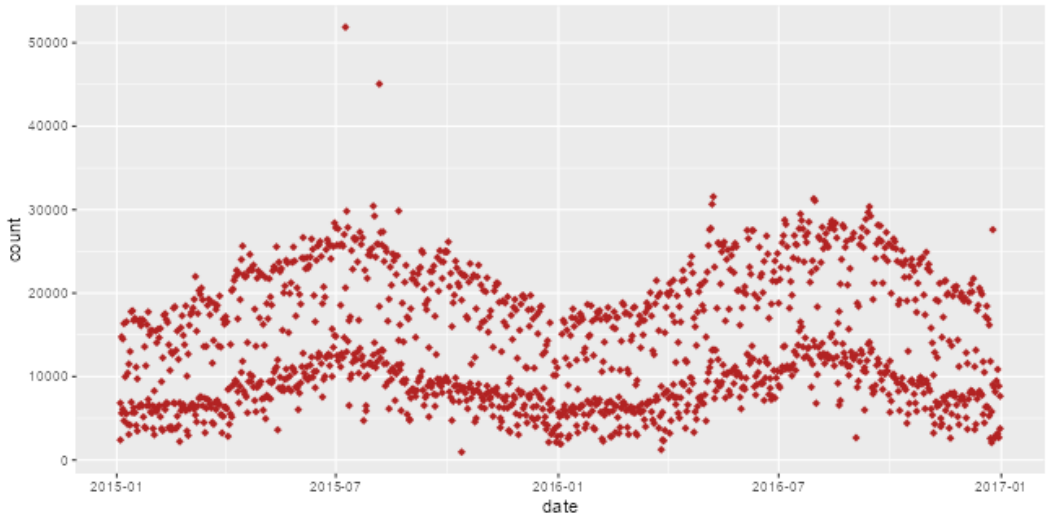


FIGURE 3.3: We can change the appearance of the geometry, here illustrated by turning the black dots in larger, red diamonds.

ggplot2 understands both `color` and `colour` as well as the short version `col`.

You can use preset colors (here is a full list[6]) or hex color codes[7], both in quotes, and even RGB/RGBA colors by using the `rgb()` function.

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "#b22222", shape = "diamond", size = 2)

ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = rgb(178, 34, 34, maxColorValue = 255), shape = "diamond", size = 2)
```

Each geom comes with its own properties (called *arguments*) and the same argument may result in a different change depending on the geom you are using.

```
ggplot(bikes, aes(x = date, y = count)) +
    geom_point(color = "firebrick", shape = "diamond", size = 2) +
    geom_smooth(formula = y  ~poly(x, 4), se = FALSE,
                color = "gray40", size = 2)
```

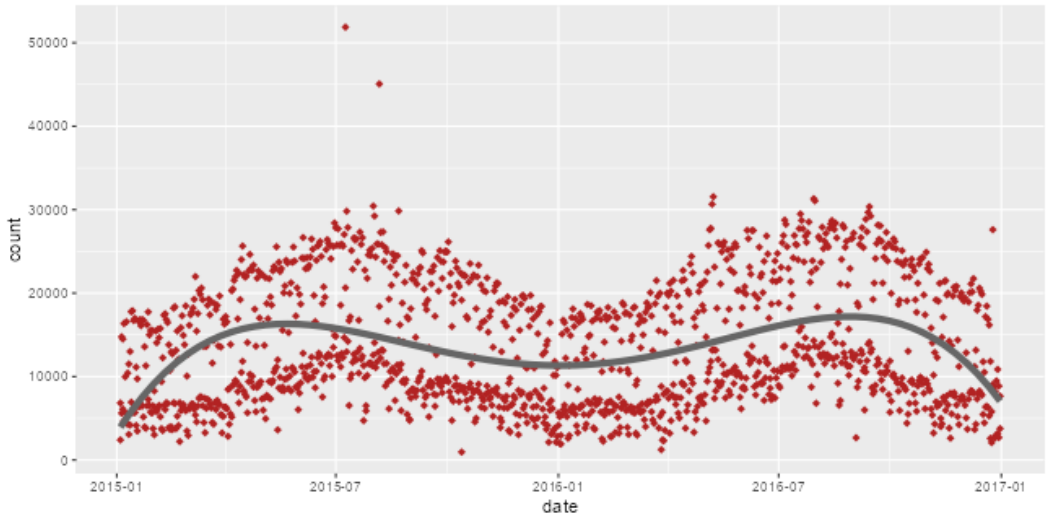


FIGURE 3.4: You can style each geometrical layer on its own. Each geometry also comes with a set of individual properties.

3.3 Mapping Data to Aesthetics

You already have seen two *positional aesthetics*, `x` and `y` that can be used in combination with the `aes()` function to map variables to the x- and y-axis, respectively. There are many more aesthetic attributes we are going to use throughout the book. Some are related to

[6] http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf
[7] https://www.techopedia.com/definition/29788/color-hex-code

positions such as `ymin` and `ymax` while others change the appearance of the layer based on the variables they are mapped to such as `color` and `shape`.

As with the `x` and `y`, the mapping needs to be wrapped into `aes()` so that ggplot knows that you are referring to columns of your data set. There are two different levels on which you can apply aesthetic mappings: either for all layers or for individual layers only.

In case we want to color our points based on the period of the day to reveal the two patterns, we add `aes(color = day_night)` to our point layer `geom_point()`:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(aes(color = day_night)) +
  geom_smooth()
```

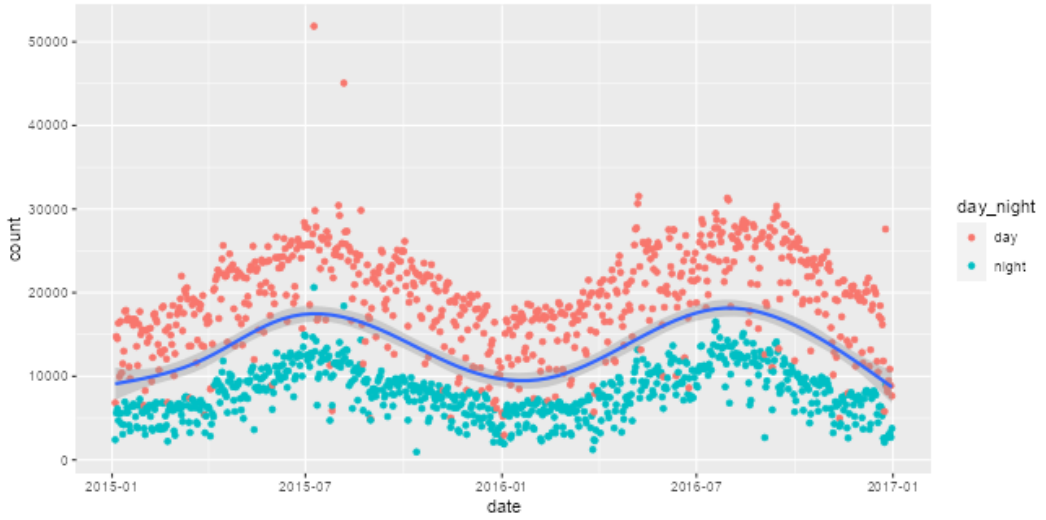


FIGURE 3.5: By applying aesthetic mapping to the `day_night` variable in the point layers, the two groups can be identified. Note that **ggplot2** automatically adds a legend to the plot.

By supplying the aesthetic mapping inside `ggplot()` all layers are applying the same mapping. Consequently, the grouping and coloring is also used for the smoothing:

```
ggplot(bikes, aes(x = date, y = count, color = day_night)) +
  geom_point() +
  geom_smooth()
```

The aesthetic mapping to both, `geom_point()` and `geom_smooth()`, is also reflected in the legend which now shows points, lines, and ribbons.

Note that you can overwrite the mapping specified in `ggplot()` in individual layers by either supplying a fixed value to that property:

```
ggplot(bikes, aes(x = date, y = count, color = day_night)) +
  geom_point(color = "black") +
```

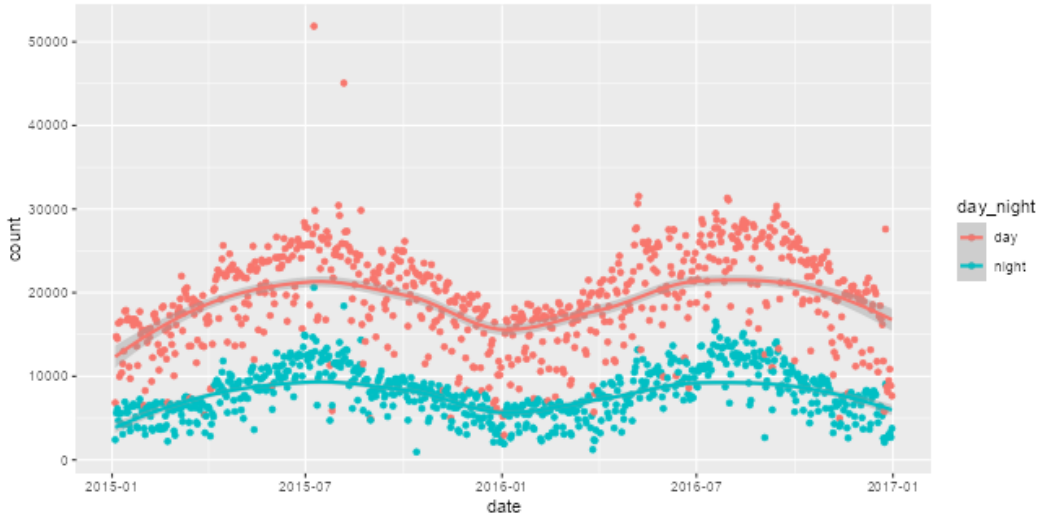


FIGURE 3.6: Both, the points and the smoothing lines are encoded by daytime. Because of that, the smoothing is hard to see and best visible in the legend.

```
  geom_smooth()

ggplot(bikes, aes(x = date, y = count, color = day_night)) +
  geom_point() +
  geom_smooth(color = "black")
```

FIGURE 3.7: Overwriting the color encoding in the point (left) or the smoothing (right). Note that the grouping is removed as well which changes the behaviour of the smoothing geom.

3.4 Replace the default ggplot2 theme

And to illustrate some more of ggplot's versatility, let's get rid of the grayish default **ggplot2** look by setting a different built-in theme, e.g. `theme_bw()`. One can add a theme directly to a ggplot composition or setting a theme globally—by calling `theme_set()` all following plots will have the same black'n'white theme. In the same step, we can also increase the `base_size` of the plot elements. The default base size is 11 and tends to be too small, at least for my workflow.

```
theme_set(theme_bw(base_size = 16))

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

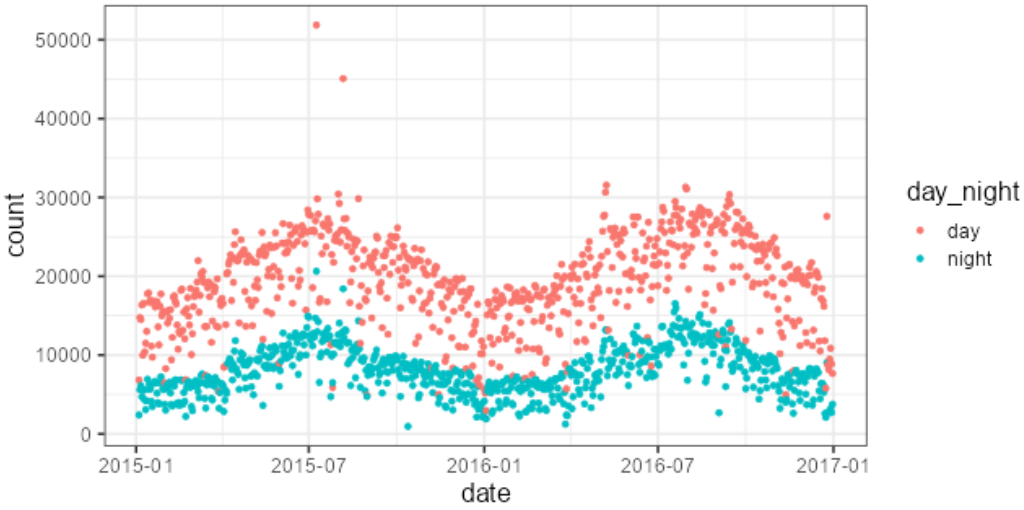


FIGURE 3.8: We can change the appearance for all following plots generated with **ggplot2** by globally setting another theme via `theme_set()`.

You can find more on how to use built-in themes and how to customize themes in the section `themes`. From the next chapter on, we will also use the `theme()` function to customize particular elements of the theme.

`theme()` is an essential command to manually modify all kinds of theme elements (texts, rectangles, and lines). To see which details of a ggplot theme can be modified have a look here[8]—and take some time, this is a looong list.

3.5 Export ggplot2 as graphic files

[8] https://ggplot2.tidyverse.org/reference/theme.html

Part I

How To Polish Your Graphics

4

Quick Steps to Improve Your Graphic

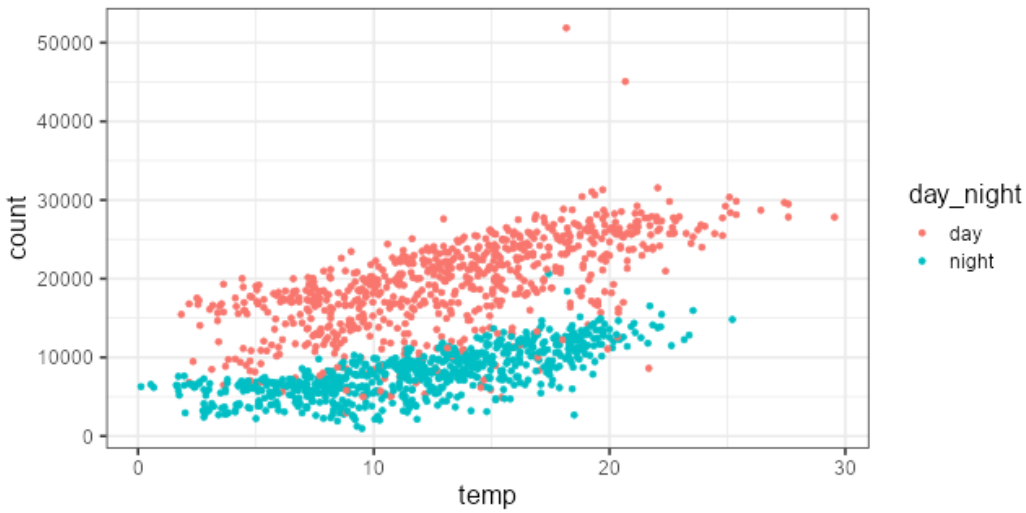
5

Working with Text

5.1 Titles

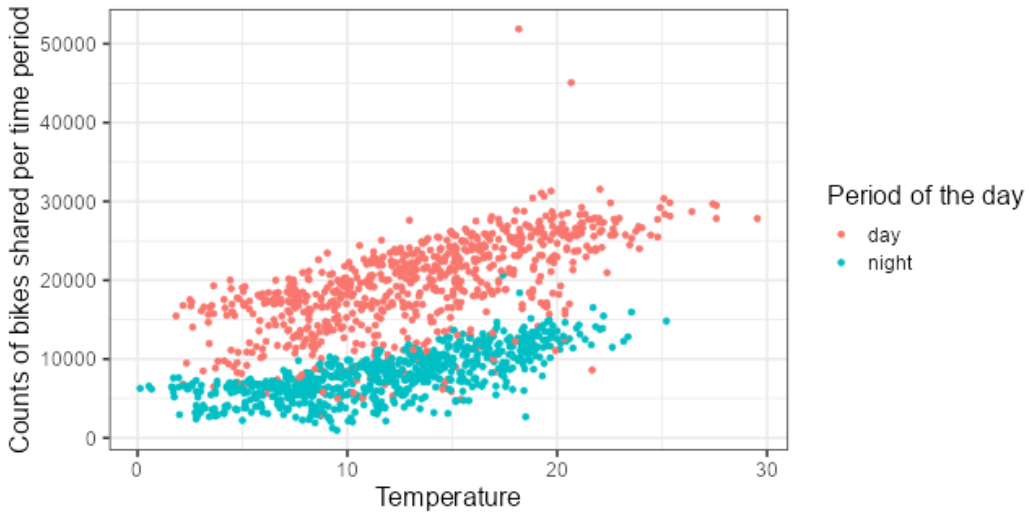
By default, **ggplot2** uses the column names specified for the aesthetic mappings as titles. That means, one could overwrite the column names in the source data. Usually the names are optimized for developing code (i.e. they contain no white spaces or unusual symbols) and it is better to leave them as they are. Also, the columns might be addressed somewhere else in the script and overwriting their names sounds like a bad idea in such a case.

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



Let's add some well-written labels to the axes and legend without changing the input object. For this, we add the `labs()` function and provide a character string for each label we want to change (here `x`, `y`, and `color`):

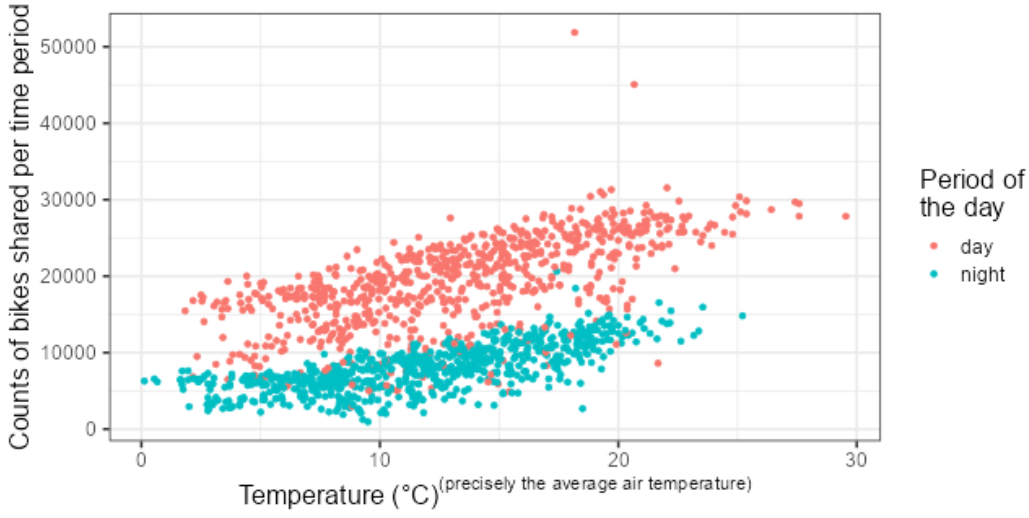
```
ggplot(bikes, aes(y = count, x = temp, color = day_night)) +
  geom_point() +
  labs(x = "Temperature",
       y = "Counts of bikes shared per time period",
       color = "Period of the day")
```



You can also add each axis title via `xlab()` and `ylab()` to specify each label individually. However, I recommend to use the `labs()` as (a) it allows you to add more labels to your plot and (b) collects all labels in one place.

By adding `\n` to a string you force a line break. Usually you can also specify symbols by simply adding the symbol itself (here "°"). In case you want to add more complex expressions you can use the `expression()` function. Here we use it in combination with `paste()` to concatenate multiple strings. With this approach we can for example add superscripts:

```
ggplot(bikes, aes(y = count, x = temp, color = day_night)) +
  geom_point() +
  labs(x = expression(paste("Temperature (°C)"^"(precisely the average air temperature)")),
       y = "Counts of bikes shared per time period",
       color = "Period of\nthe day")
```



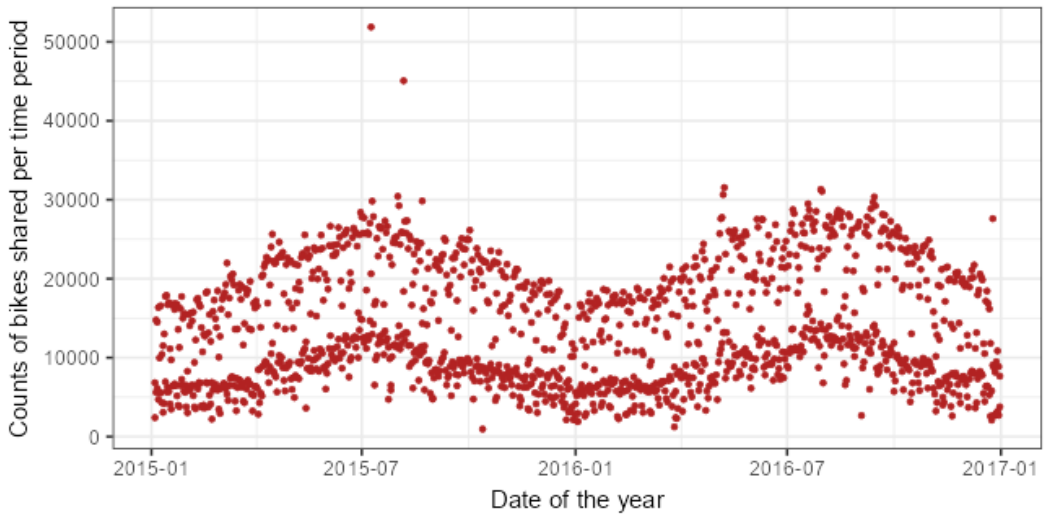
6

Working with Axes

6.0.0.0.1 Increase Space between Axis and Axis Titles

`theme()` is an essential command to modify particular theme elements (texts and titles, boxes, symbols, backgrounds, …). We are going to use them a lot! For now, we are going to modify text elements. We can change the properties of all or particular text elements (here axis titles) by overwriting the default `element_text()` within the `theme()` call:

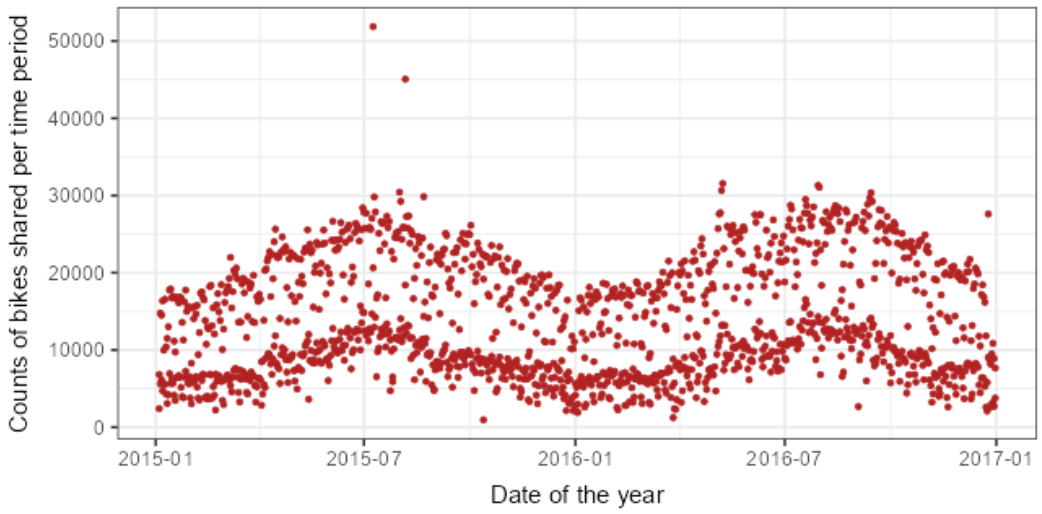
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.title.x = element_text(vjust = 0, size = 15),
        axis.title.y = element_text(vjust = 2, size = 15))
```



`vjust` refers to the vertical alignment, which usually ranges between 0 and 1 but you can also specify values outside that range. Note that even though we move the axis title on the y axis horizontally, we need to specify `vjust` (which is correct form the label's perspective). You can also change the distance by specifying the margin of both text elements:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
```

```
  theme(axis.title.x = element_text(margin = margin(t = 10), size = 15),
        axis.title.y = element_text(margin = margin(r = 10), size = 15))
```



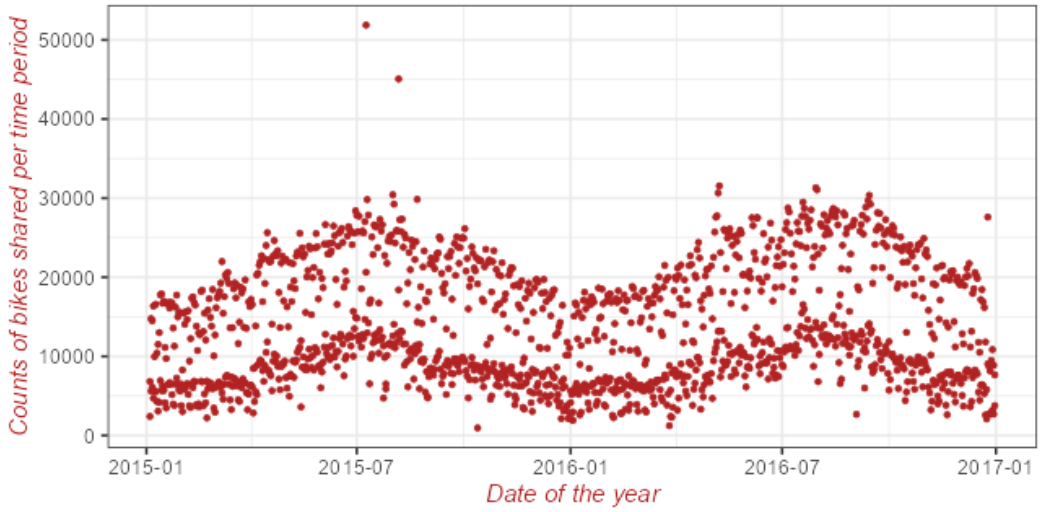
The labels `t` and `r` within the `margin()` object refer to *top* and *right*, respectively. You can also specify the four margins as `margin(t, r, b, l)`. Note that we now have to change the right margin to modify the space on the y axis, not the bottom margin.

A good way to remember the order of the margin sides is "*t*-*r*-ou-*b*-*l*-e".

6.0.0.0.2 Change Aesthetics of Axis Titles

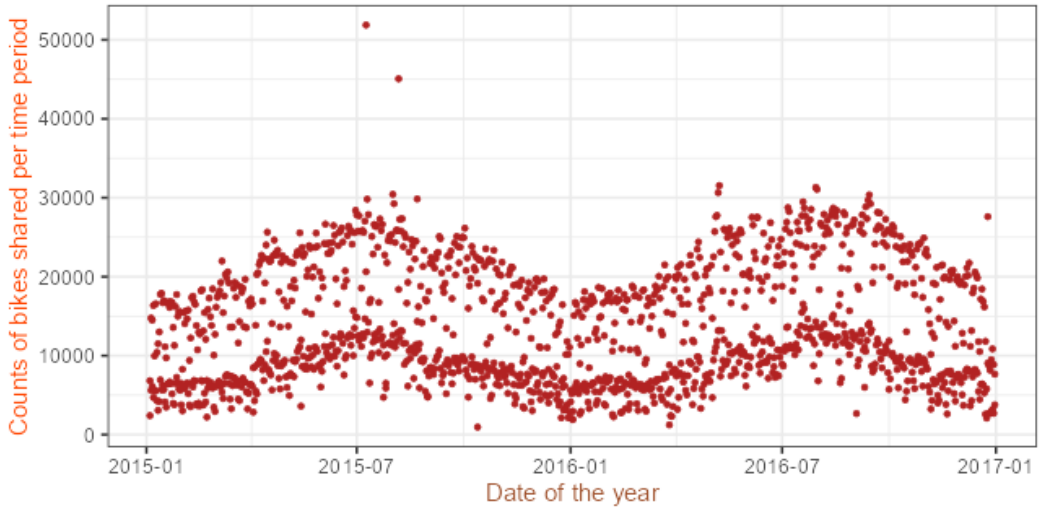
Again, we use the `theme()` function and modify the element `axis.title` and/or the subordinated elements `axis.title.x` and `axis.title.y`. Within the `element_text()` we can for example overwrite the defaults for `size`, `color`, and `face`:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.title = element_text(size = 15, color = "firebrick",
                                  face = "italic"))
```



The `face` argument can be used to make the font `bold` or `italic` or even `bold.italic`.

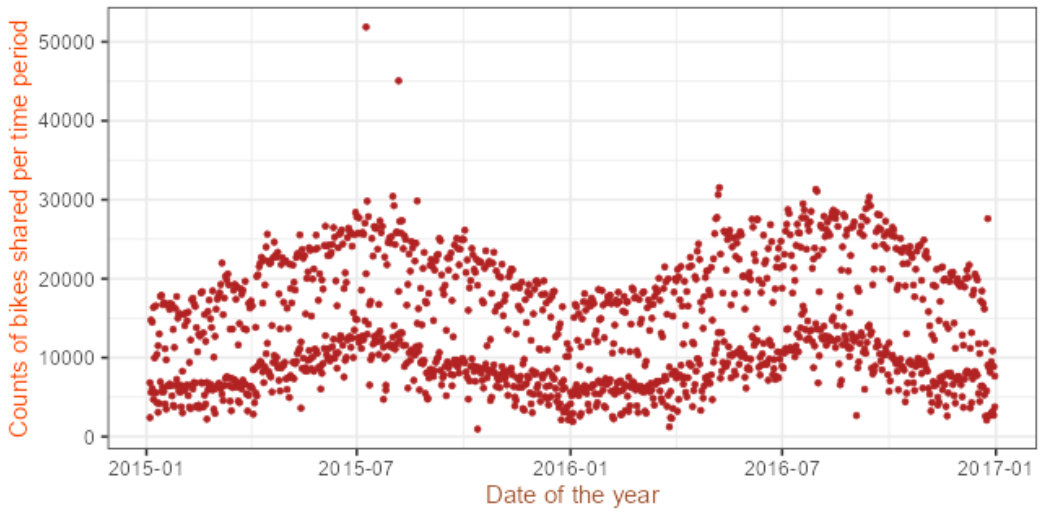
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.title.x = element_text(color = "sienna", size = 15),
        axis.title.y = element_text(color = "orangered", size = 15))
```



You could also use a combination of `axis.title` and `axis.title.y`, since `axis.title.x` inherits the values from `axis.title`. Expand to see example.

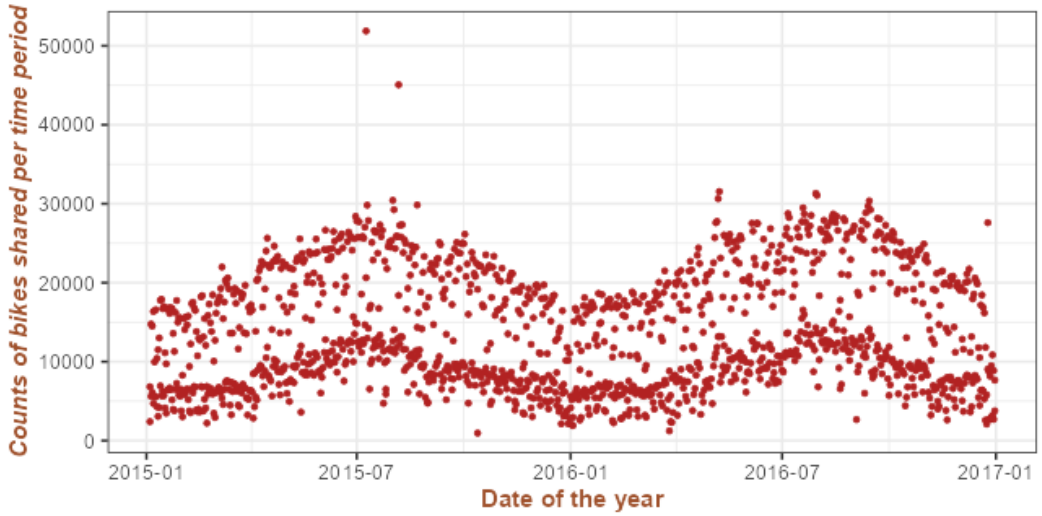
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
```

```
       y = "Counts of bikes shared per time period") +
  theme(axis.title = element_text(color = "sienna", size = 15),
        axis.title.y = element_text(color = "orangered", size = 15))
```



One can modify some properties for both axis titles and other only for one or properties for each on its own:

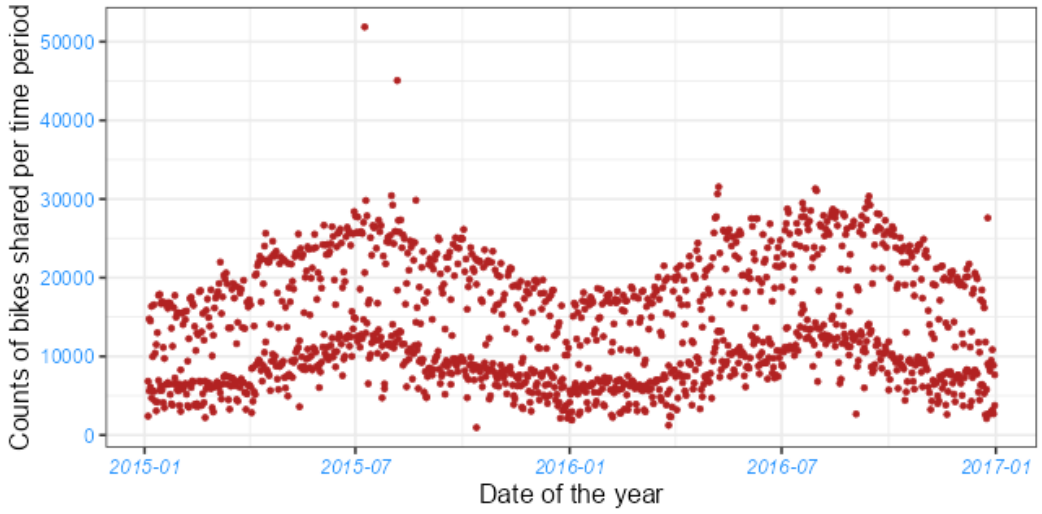
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.title = element_text(color = "sienna", size = 15, face = "bold"),
        axis.title.y = element_text(face = "bold.italic"))
```



6.0.0.0.3 Change Aesthetics of Axis Text

Similarly, you can also change the appearance of the axis text (here *the numbers*) by using `axis.text` and/or the subordinated elements `axis.text.x` and `axis.text.y`:

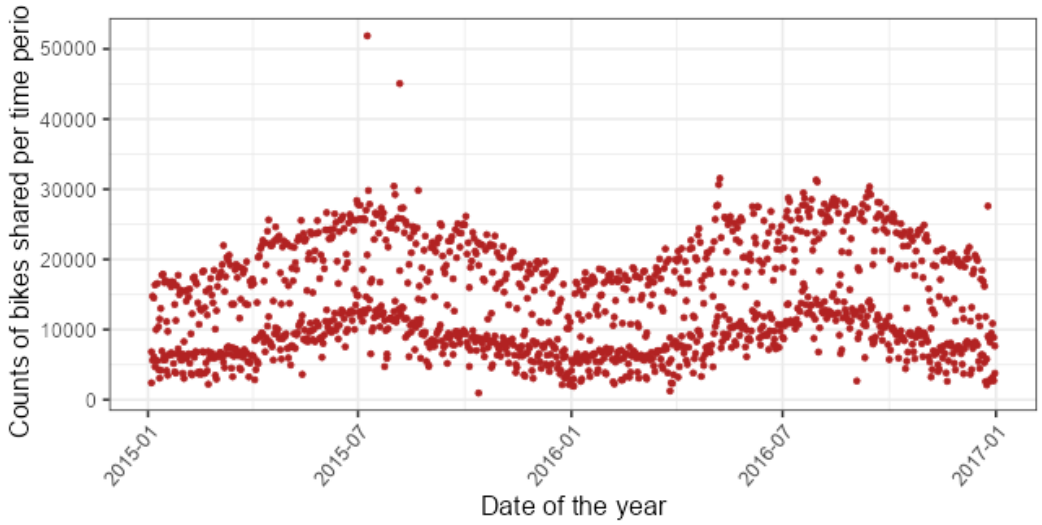
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.text = element_text(color = "dodgerblue", size = 12),
        axis.text.x = element_text(face = "italic"))
```



6.0.0.0.4 Rotate Axis Text

Specifying an `angle` allows you to rotate any text elements. With `hjust` and `vjust` you can adjust the position of the text afterwards horizontally (0 = left, 1 = right) and vertically (0 = top, 1 = bottom):

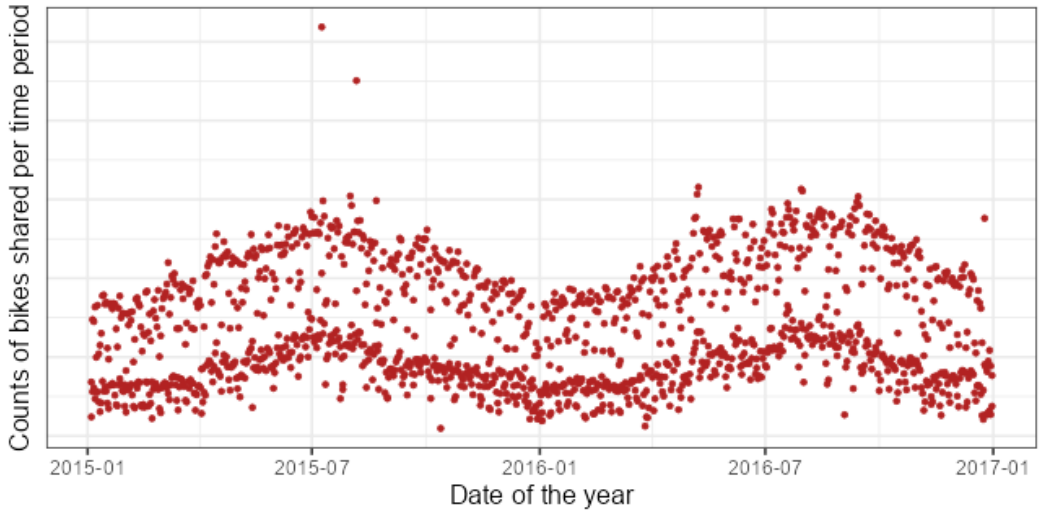
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.text.x = element_text(angle = 50, vjust = 1, hjust = 1, size = 12))
```



6.0.0.0.5 Remove Axis Text & Ticks

There may be rarely a reason to do so—but this is how it works:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  theme(axis.ticks.y = element_blank(),
        axis.text.y = element_blank())
```



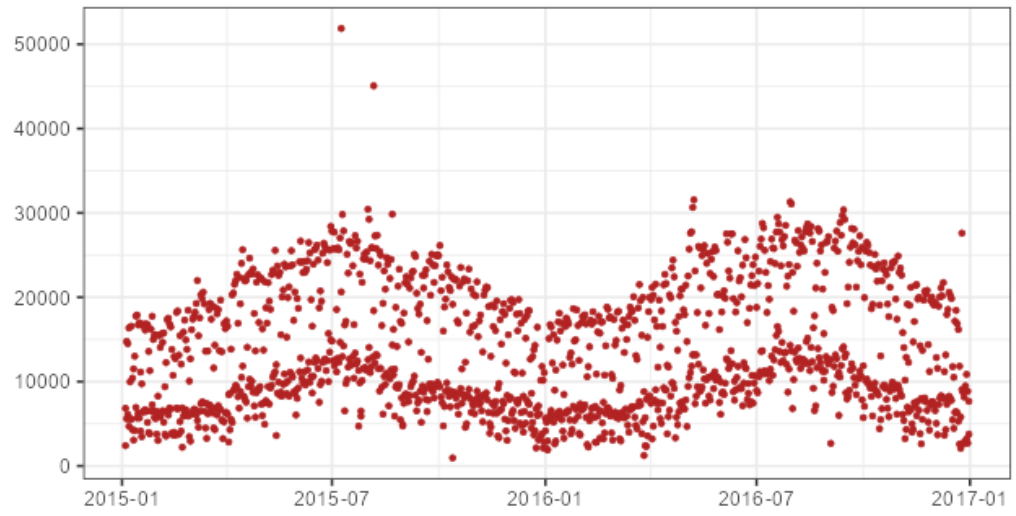
I introduced three theme elements—text, lines, and rectangles—but actually there is one more: `element_blank()` which removes the element (and thus is not considered an official element).

If you want to get rid of a theme element, the element is always `element_blank()`.

6.0.0.0.6 Remove Axis Titles

We could again use `theme_blank()` but it is way simpler to just remove the label in the `labs()` (or `xlab()`) call:

```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = NULL, y = "")
```

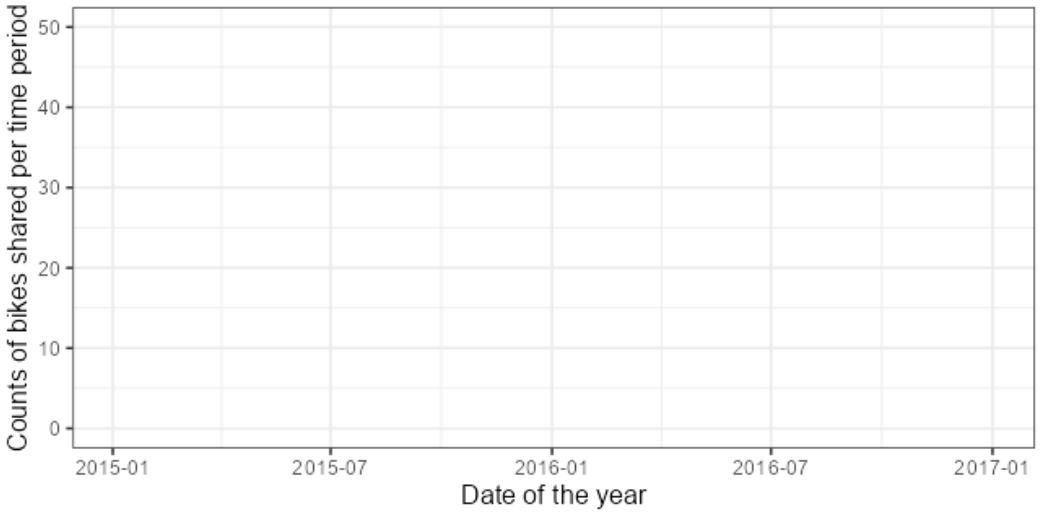


Note that `NULL` removes the element (similarly to `element_blank()`) while empty quotes "" will keep the spacing for the axis title and simply print nothing.

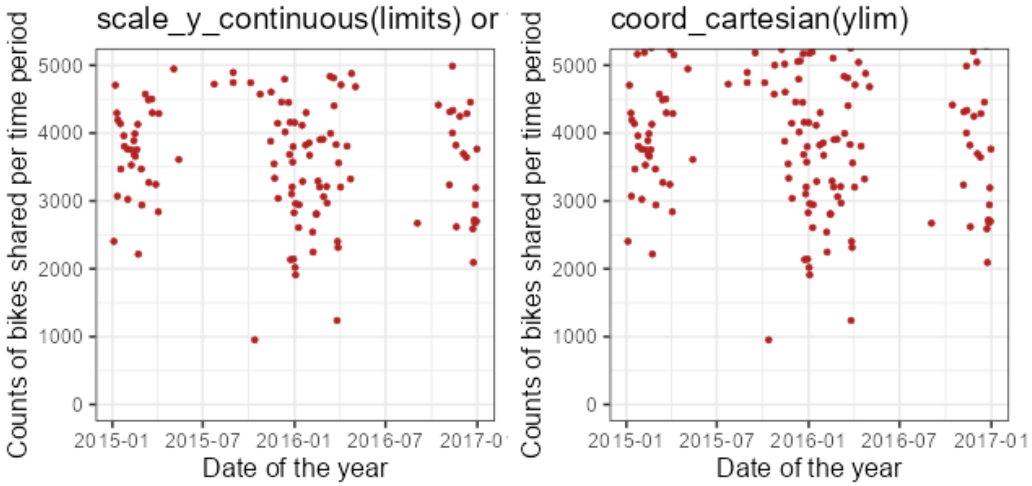
6.0.0.0.7 Limit Axis Range

Sometimes you want to zoom into take a closer look at some range of your data. You can do this without subsetting your data:

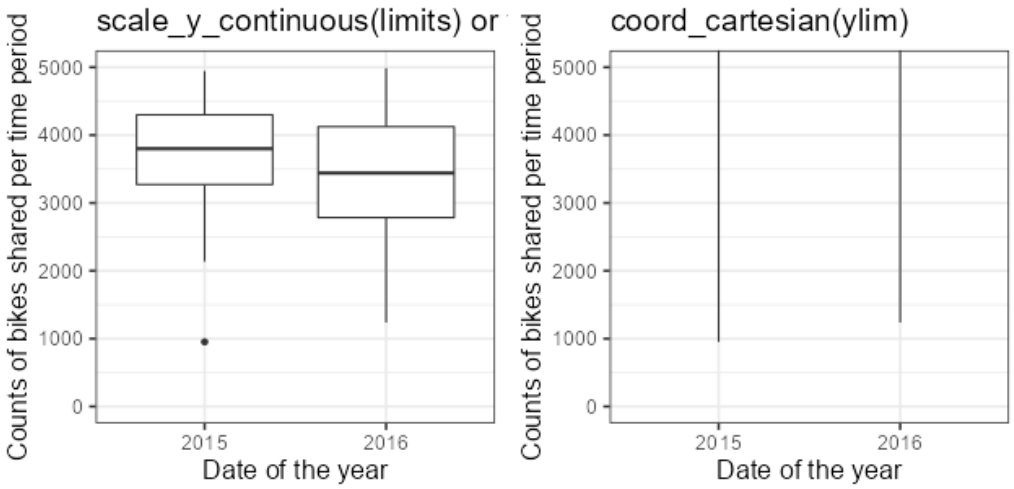
```
ggplot(bikes, aes(x = date, y = count)) +
  geom_point(color = "firebrick") +
  labs(x = "Date of the year",
       y = "Counts of bikes shared per time period") +
  ylim(c(0, 50))
```



Alternatively you can use `scale_y_continuous(limits = c(0, 50))` or `coord_cartesian(ylim = c(0, 50))`. The former removes all data points outside the range while the second adjusts the visible area and is similar to `ylim(c(0, 50))`. You may wonder: *So in the end both result in the same.* But not really, there is an important difference—compare the two following plots:



You might have spotted that on the left there is some empty buffer around your y limits while on the right points are plotted right up to the border and even beyond. This perfectly illustrates the subsetting (left) versus the zooming (right). To show why this is important let's have a look at a different chart type, a box plot:

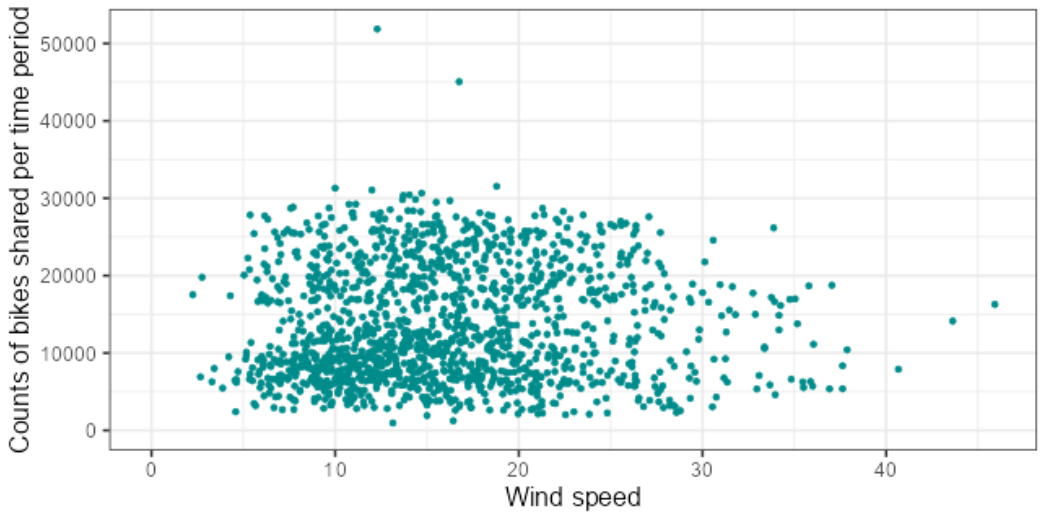


Um. Because `scale_x|y_continuous()` subsets the data first, we get completely different (and wrong, at least if in the case this was not your aim) estimates for the box plots! I hope you don't have to go back to your old scripts now and check if you *maybe* have manipulated your data while plotting and did report wrong summary stats in your report, paper or thesis...

6.0.0.0.8 Force Plot to Start at Origin

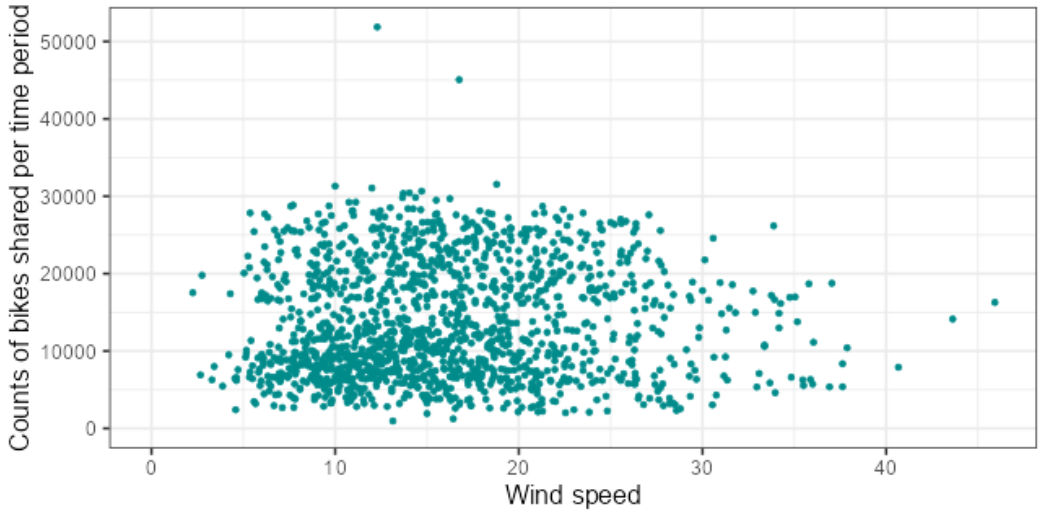
Related to that, you can force R to plot the graph starting at the origin:

```
library(tidyverse)

ggplot(bikes, aes(x = wind_speed, y = count)) +
  geom_point(color = "darkcyan") +
  labs(x = "Wind speed",
       y = "Counts of bikes shared per time period") +
  expand_limits(x = 0, y = 0)
```



Using `coord_cartesian(xlim = c(0, NA), ylim = c(0, NA))` will lead to the same result. Expand to see example.

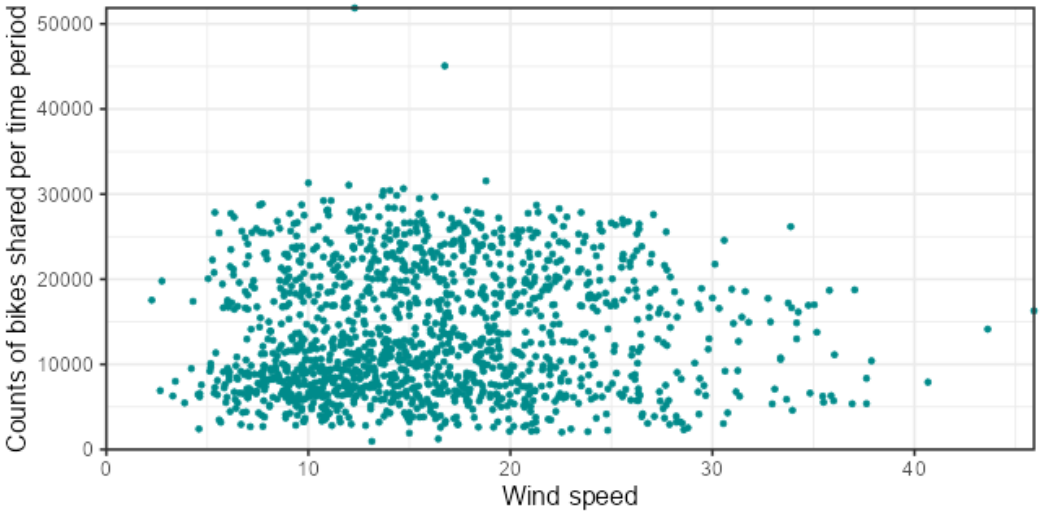
```
library(tidyverse)

ggplot(bikes, aes(x = wind_speed, y = count)) +
  geom_point(color = "darkcyan") +
  labs(x = "Wind speed",
       y = "Counts of bikes shared per time period") +
  coord_cartesian(xlim = c(0, NA), ylim = c(0, NA))
```



But we can also force it to *literally* start at the origin!

```
ggplot(bikes, aes(x = wind_speed, y = count)) +
  geom_point(color = "darkcyan") +
```

```
  labs(x = "Wind speed",
       y = "Counts of bikes shared per time period") +
  expand_limits(x = 0, y = 0) +
  coord_cartesian(expand = FALSE, clip = "off")
```



The argument `clip = "off"` in any coordinate system, always starting with `coord_*`, allows to draw outside of the panel area.

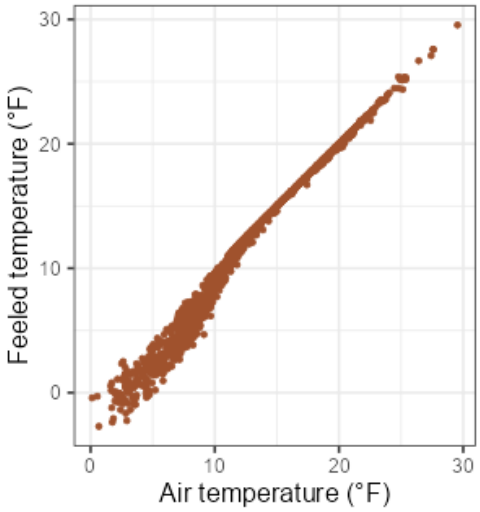
Here, I call it to make sure that the tick marks at `c(0, 0)` are not cut. See the Twitter thread by Claus Wilke[1] for more details.

6.0.0.0.9 Axes with Same Scaling

For demonstrating purposes, let's plot temperature against temperature with some random noise. The `coord_equal()` is a coordinate system with a specified ratio representing the number of units on the y-axis equivalent to one unit on the x-axis. The default, `ratio = 1`, ensures that one unit on the x-axis is the same length as one unit on the y-axis:

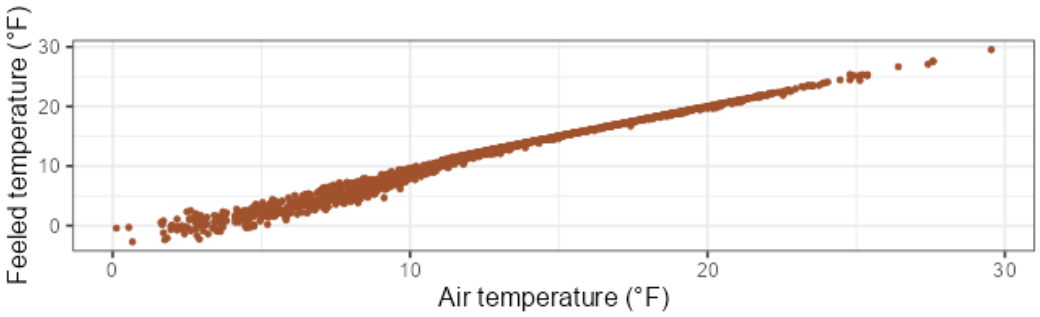
```
ggplot(bikes, aes(x = temp, y = temp_feel)) +
  geom_point(color = "sienna") +
  labs(x = "Air temperature (°F)", y = "Feeled temperature (°F)") +
  coord_fixed()
```

[1] https://twitter.com/clauswilke/status/991542952802619392?lang=en



Ratios higher than one make units on the y axis longer than units on the x-axis, and vice versa:

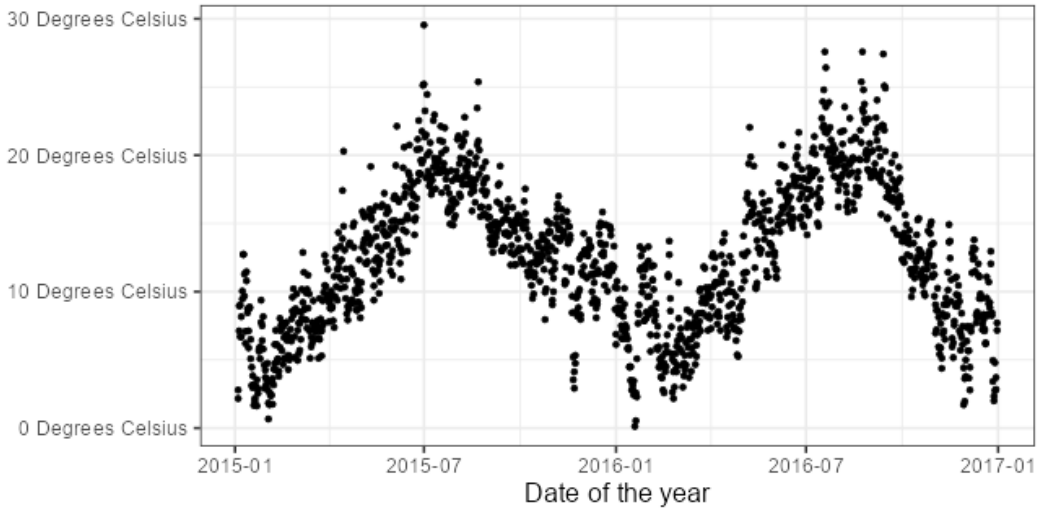
```
ggplot(bikes, aes(x = temp, y = temp_feel)) +
  geom_point(color = "sienna") +
  labs(x = "Air temperature (°F)", y = "Feeled temperature (°F)") +
  coord_fixed(ratio = 1/5)
```



6.0.0.0.10 Use a Function to Alter Labels

Sometimes it is handy to alter your labels a little, perhaps adding units or percent signs without adding them to your data. You can use a function in this case:

```
ggplot(bikes, aes(x = date, y = temp)) +
  geom_point() +
  labs(x = "Date of the year", y = NULL) +
  scale_y_continuous(label = function(x) {return(paste(x, "Degrees Celsius"))})
```



- work with scale_*_date()

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

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.

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.

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. (2021). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.24.

Index