Mastering ggplot2: From Novice to Data Viz Pro

Unleash Your Inner Data Scientist

Tuhin Rana

2023-03-17

Table of Contents

# Before We Start

In December 2017, after completing my first year of statistics, I delved into learning R. Having a background in Java, C, and C++ coding since high school, I enjoyed R but found its plots not so appealing and the code a bit tricky. On a quest for something beautiful and user-friendly, I stumbled upon a blog titled [**Beautiful plotting in R: A ggplot2 cheatsheet**](http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/) by [Zev Ross](https://twitter.com/zevross), last updated in January 2016. Intrigued, I decided to follow the tutorial step by step, learning a great deal. As time passed, I tweaked and expanded the codes, adding new chart types and resources.

Realizing that Zev Ross’s blog hadn’t been updated for years, I took the initiative to create my own version, incorporating updates like the amazing {patchwork}, {ggtext}, and {ggforce} packages. I also shared insights on custom fonts, colors, and introduced a variety of R packages for interactive charts. The journey led to a unique tutorial, and now, I’ve decided to make it public, adding even more updates, such as Maps! because who doesn’t love maps!!

I incorporated the following enhancements into my tutorial:

* Following the R style guide (e.g., by [Hadley Wickham](https://style.tidyverse.org), [Google](https://google.github.io/styleguide/Rguide.xml), or the [Coding Club](https://ourcodingclub.github.io/2017/04/25/etiquette.html#syntax) style guides).
* Implementing changes to the style and aesthetics of plots, including axis titles, legends, and color schemes for all plots.
* Ensuring that the tutorial remains up-to-date with changes in {ggplot2} (current version: 3.4.0).
* Modifying data import methods to utilize GitHub as a data source.
* Offering additional tips on various topics such as chart selection, color palettes, title modifications, adding lines, adjusting legends, annotations with labels, arrows and boxes, multi-panel plots, Geospatial Visualizations and interactive visualizations. …

## What you’ll discover

* [Tie Your Seatbelt](#prep): Setting the stage for your journey into advanced plotting techniques.
* [The Dataset](#data): Understanding the importance of data in crafting compelling visualizations.
* [The {ggplot2} Package](#ggplot): Unleashing the power of {ggplot2} for elegant and customizable plots.
* [A Default ggplot](#default): Exploring the basics with a default {ggplot} and understanding its components.
* [Working with Axes](#axes): Mastering the art of manipulating axes to convey meaningful insights.
* [Working with Titles](#titles): Crafting informative and visually appealing titles to captivate your audience.
* [Working with Legends](#legends): Enhancing clarity and interpretation by effectively managing legends.
* [Working with Backgrounds & Grid Lines](#style): Elevating aesthetics with stylish backgrounds and grid lines.
* [Working with Margins](#margins): Fine-tuning margins to optimize plot presentation.
* [Working with Multi-Panel Plots](#panels): Diving into the world of multi-panel plots for comprehensive data representation.
* [Working with Colors](#colors): Harnessing the power of color to convey information and evoke emotions.
* [Working with Themes](#themes): Creating cohesive visual narratives with carefully curated themes.
* [Working with Lines](#lines): Adding emphasis and clarity through strategic use of lines.
* [Working with Text](#text): Utilizing text annotations to provide context and highlight key findings.
* [Working with Coordinates](#coords): Manipulating coordinates to achieve desired plot layouts and perspectives.
* [Working with Chart Types](#charts): Expanding your repertoire with diverse and impactful chart types.
* [Working with Ribbons (AUC, CI, etc.)](#ribbons): Enhancing visualizations with ribbons for confidence intervals and more.
* [Working with Smoothings](#smooths): Incorporating smoothings to reveal underlying trends and patterns.
* [Working with Interactive Plots](#interactive): Engaging your audience with interactive visualizations for dynamic exploration.
* [Remarks, Tipps & Resources](#tipps): Leveraging insights, tips, and resources to further refine your plotting skills.

# 1. Tie Your Seatbelt

To fully execute the tutorial, you’ll need to install the following packages:

* {ggplot2}, a part of the {tidyverse} package collection
* {tidyverse} package collection, including:
  + {dplyr} for data wrangling
  + {tibble} for modern data frames
  + {tidyr} for data cleaning
  + {forcats} for handling factors
* {corrr} for calculating correlation matrices
* {cowplot} for composing ggplots
* {ggforce} for creating sina plots and other advanced visualizations
* {ggrepel} for enhancing text labeling in plots
* {ggridges} for creating ridge plots
* {ggsci} for accessing nice color palettes
* {ggtext} for advanced text rendering in plots
* {ggthemes} for additional plot themes
* {grid} for creating graphical objects
* {gridExtra} for additional functions for “grid” graphics
* {patchwork} for generating multi-panel plots
* {prismatic} for manipulating colors
* {rcartocolor} for accessing great color palettes
* {scico} for perceptional uniform color palettes
* {showtext} for utilizing custom fonts
* {shiny} for developing interactive apps
* Several packages for interactive visualizations, including:
  + {charter}
  + {echarts4r}
  + {ggiraph}
  + {highcharter}
  + {plotly}

## 1.1 Installing Packages

To install the necessary packages, run the following code:

# install CRAN packages  
install.packages(  
 c("ggplot2", "tibble", "tidyr", "forcats", "purrr", "prismatic", "corrr",   
 "cowplot", "ggforce", "ggrepel", "ggridges", "ggsci", "ggtext", "ggthemes",   
 "grid", "gridExtra", "patchwork", "rcartocolor", "scico", "showtext",   
 "shiny", "plotly", "highcharter", "echarts4r")  
)  
  
# install from GitHub since not on CRAN  
install.packages(devtools)  
devtools::install\_github("JohnCoene/charter")

For instructional purposes, and to ensure smooth transitions for learners navigating directly to specific plots, I’ll load the necessary package beside {ggplot2} in the corresponding section.

# 2. The Dataset

We are utilizing data from the *National Morbidity and Mortality Air Pollution Study* (NMMAPS), focusing specifically on data pertaining to Chicago and the years 1997 to 2000 to ensure manageability of the plots. For a more comprehensive understanding of this dataset, readers can refer to Roger Peng’s book [Statistical Methods in Environmental Epidemiology with R](http://www.springer.com/de/book/9780387781662).

To import the data into our R session, we can employ read\_csv() from the {readr} package. Subsequently, we’ll store the data in a variable named chic using the *assignment arrow* <-. Just Copy and Paste the following code.

chic <- readr::read\_csv("https://raw.githubusercontent.com/rana2hin/ggplot\_guide/master/chicago\_data.csv")

Rows: 1461 Columns: 10  
── Column specification ────────────────────────────────────────────────────────  
Delimiter: ","  
chr (3): city, season, month  
dbl (6): temp, o3, dewpoint, pm10, yday, year  
date (1): date  
  
ℹ Use `spec()` to retrieve the full column specification for this data.  
ℹ Specify the column types or set `show\_col\_types = FALSE` to quiet this message.

|  |
| --- |
| Using namespace Directly |
| The :: symbolizes *namespace* and enables accessing a function without loading the entire package. Alternatively, you could load the readr package first using library(readr) and then execute chic <- read\_csv(...) subsequently. |

tibble::glimpse(chic)

Rows: 1,461  
Columns: 10  
$ city <chr> "chic", "chic", "chic", "chic", "chic", "chic", "chic", "chic…  
$ date <date> 1997-01-01, 1997-01-02, 1997-01-03, 1997-01-04, 1997-01-05, …  
$ temp <dbl> 36.0, 45.0, 40.0, 51.5, 27.0, 17.0, 16.0, 19.0, 26.0, 16.0, 1…  
$ o3 <dbl> 5.659256, 5.525417, 6.288548, 7.537758, 20.760798, 14.940874,…  
$ dewpoint <dbl> 37.500, 47.250, 38.000, 45.500, 11.250, 5.750, 7.000, 17.750,…  
$ pm10 <dbl> 13.052268, 41.948600, 27.041751, 25.072573, 15.343121, 9.3646…  
$ season <chr> "Winter", "Winter", "Winter", "Winter", "Winter", "Winter", "…  
$ yday <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18…  
$ month <chr> "Jan", "Jan", "Jan", "Jan", "Jan", "Jan", "Jan", "Jan", "Jan"…  
$ year <dbl> 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1…

library(gt)  
head(chic, 10) %>% gt()

| city | date | temp | o3 | dewpoint | pm10 | season | yday | month | year |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| chic | 1997-01-01 | 36.0 | 5.659256 | 37.500 | 13.052268 | Winter | 1 | Jan | 1997 |
| chic | 1997-01-02 | 45.0 | 5.525417 | 47.250 | 41.948600 | Winter | 2 | Jan | 1997 |
| chic | 1997-01-03 | 40.0 | 6.288548 | 38.000 | 27.041751 | Winter | 3 | Jan | 1997 |
| chic | 1997-01-04 | 51.5 | 7.537758 | 45.500 | 25.072573 | Winter | 4 | Jan | 1997 |
| chic | 1997-01-05 | 27.0 | 20.760798 | 11.250 | 15.343121 | Winter | 5 | Jan | 1997 |
| chic | 1997-01-06 | 17.0 | 14.940874 | 5.750 | 9.364655 | Winter | 6 | Jan | 1997 |
| chic | 1997-01-07 | 16.0 | 11.920985 | 7.000 | 20.228428 | Winter | 7 | Jan | 1997 |
| chic | 1997-01-08 | 19.0 | 8.678477 | 17.750 | 33.134819 | Winter | 8 | Jan | 1997 |
| chic | 1997-01-09 | 26.0 | 13.355892 | 24.000 | 12.118381 | Winter | 9 | Jan | 1997 |
| chic | 1997-01-10 | 16.0 | 10.448264 | 5.375 | 24.761534 | Winter | 10 | Jan | 1997 |

# 3. The ggplot2 Package

ggplot2 is a graphics system that facilitates the declarative creation of visualizations, founded on principles outlined in [The Grammar of Graphics](https://link.springer.com/chapter/10.1007/978-3-642-21551-3_13). With ggplot2, you furnish the data, specify how variables should be mapped to aesthetics, define graphical parameters to employ, and the system handles the rest.

A ggplot is made up several key elements:

1. **Data**: Your raw dataset that you want to visualize.
2. **Geometries** geom\_: These are the shapes that represent your data, like points, lines, or bars.
3. **Aesthetics** aes(): This controls how your data is visually represented, including aspects like color, size, and shape.
4. **Scales** scale\_: These map the data onto the aesthetic dimensions, like converting data values to plot dimensions or factor values to colors.
5. **Statistical transformations** stat\_: These are statistical summaries of your data, such as calculating quantiles or fitting curves.
6. **Coordinate system** coord\_: This defines how your data coordinates are mapped onto the plot’s coordinate system.
7. **Facets** facet\_: This organizes your data into a grid of plots based on specified variables.
8. **Visual themes** theme(): These set the overall appearance of your plot, covering things like background, grids, axes, default fonts, sizes, and colors.

|  |  |
| --- | --- |
|  | **The number of elements may vary depending on the situation you’re working on.** |

## 3.1 A Default ggplot

Before diving into the capabilities of {ggplot2}, we need to load the package. Alternatively, we can load it through the [tidyverse package collection](https://www.tidyverse.org/):

library(ggplot2)  
# Or,  
library(tidyverse)

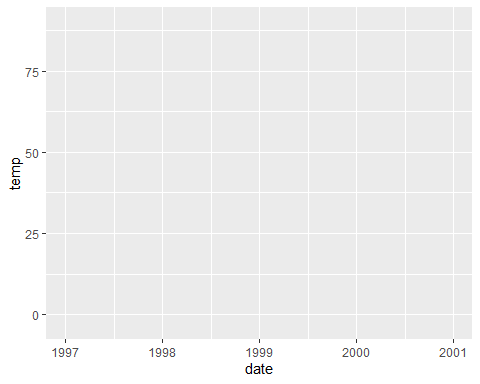
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──  
✔ dplyr 1.1.4 ✔ readr 2.1.5  
✔ forcats 1.0.0 ✔ stringr 1.5.1  
✔ lubridate 1.9.3 ✔ tibble 3.2.1  
✔ purrr 1.0.2 ✔ tidyr 1.3.1  
── Conflicts ────────────────────────────────────────── tidyverse\_conflicts() ──  
✖ dplyr::filter() masks stats::filter()  
✖ dplyr::lag() masks stats::lag()  
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors

The syntax of {ggplot2} differs from base R. Following the basic elements, a default ggplot requires specifying three things: the *data*, *aesthetics*, and a *geometry*. To begin defining a plotting object, we call ggplot(data = df), indicating that we’ll work with that dataset. Typically, we aim to plot two variables—one on the x-axis and one on the y-axis. These are *positional aesthetics*, so we add aes(x = var1, y = var2) to the ggplot() call (where aes() denotes aesthetics). However, there are cases where one may need to specify one, three, or more variables, which we’ll address later.

|  |
| --- |
| Pay Attention! |
| We indicate the data *outside* of aes() and include the variables that ggplot maps to the aesthetics *inside* of aes(). |

In this instance, we assign the variable date to the x-position and the variable temp to the y-position. Subsequently, we’ll also map variables to various other aesthetics such as color, size, and shape.

(g <- ggplot(chic, aes(x = date, y = temp)))



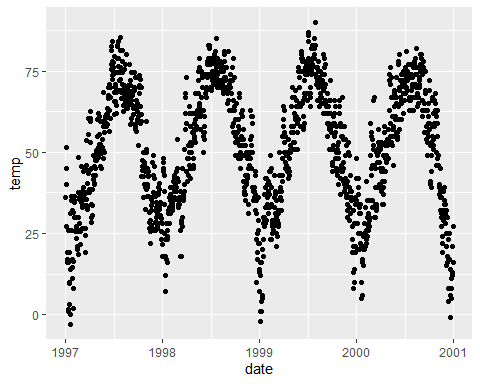
Ah, the reason only a panel is generated when executing this code is because {ggplot2} lacks information on how we want to visualize the data. We still need to specify a geometry!

In {ggplot2}, you can store the current ggobject in a variable of your choosing, such as g. This allows you to extend the ggobject later by adding additional layers, either all at once or by assigning it to the same or another variable.

|  |
| --- |
| A Quick Tip! |
| By using parentheses when assigning an object, the object will be printed immediately. Instead of writing g <- ggplot(...) followed by g, we can simply write (g <- ggplot(...)). |

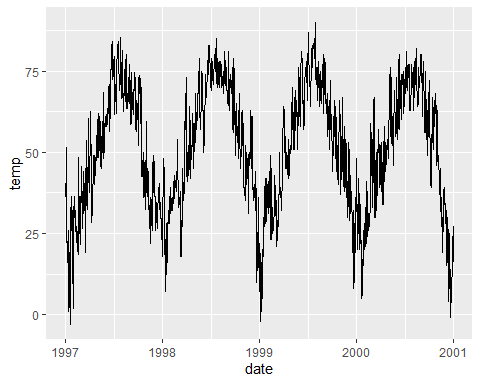
There’s a wide array of geometries in {ggplot2}, often referred to as *geoms* because their function names typically start with geom\_. You can find the full list of default geoms [here](https://ggplot2.tidyverse.org/reference/), and there are even more options available through extension packages, which you can explore [here](https://exts.ggplot2.tidyverse.org/). To instruct {ggplot2} on the style we want to use, we can, for example, add geom\_point() to create a scatter plot:

g + geom\_point()



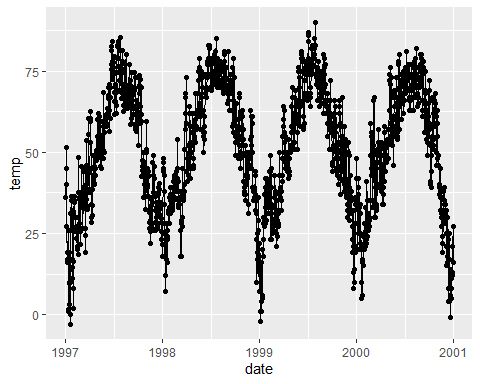
Great! However, this data could also be represented as a line plot (although it might not be the optimal choice, but it’s a common practice). So, we can simply replace geom\_point() with geom\_line() and boom!

g + geom\_line()



Indeed, one can combine multiple geometric layers, and this is where the magic and fun truly begin!

g + geom\_line() + geom\_point()

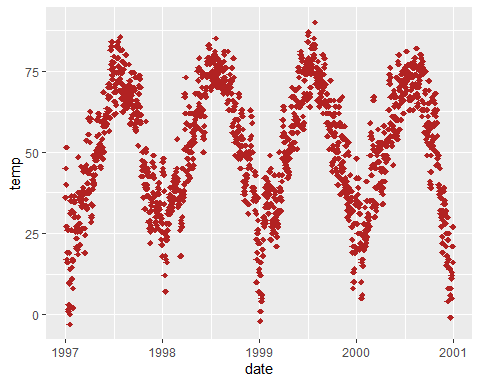


That’s enough discussion on geometries for now. Don’t worry, we’ll dive into various plot types at a later point, as outlined [here](#charts).

### 3.1.1 Change Properties of Geometries

Within the geom\_\* command, you can already manipulate visual aesthetics such as the color, shape, and size of your points. Let’s transform all points into large fire-red diamonds!

g + geom\_point(color = "firebrick", shape = "diamond", size = 2)

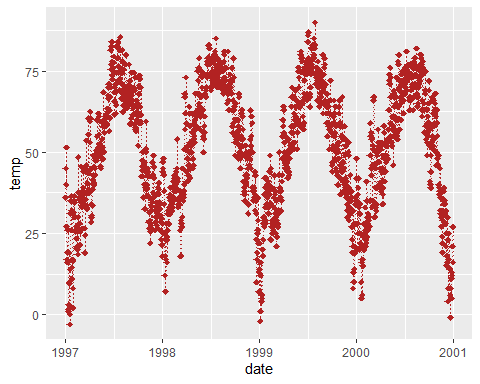


|  |
| --- |
| Color or Colour? |
| {ggplot2} understands both color and colour as well as the short version col. |

|  |
| --- |
| Color Presets 🎨 |
| You can utilize preset colors (a full list can be found [here](http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf)) or [hex color codes](https://www.techopedia.com/definition/29788/color-hex-code), both enclosed in quotes. Additionally, you can specify RGB/RGBA colors using the rgb() function. Click to expand:  g + geom\_point(color = "#b22222", shape = "diamond", size = 2) g + geom\_point(color = rgb(178, 34, 34, maxColorValue = 255), shape = "diamond", size = 2) |

Each geom has its unique properties, referred to as *arguments*, and the same argument might produce different effects depending on the geom you’re employing.

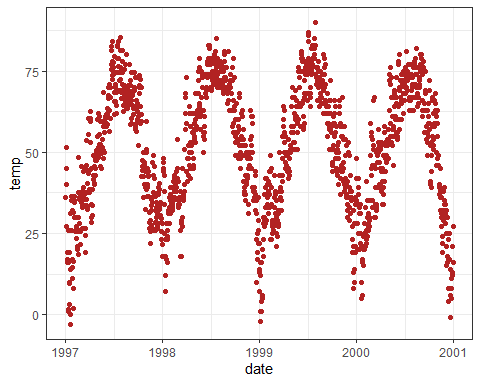
g + geom\_point(color = "firebrick", shape = "diamond", size = 2) +  
 geom\_line(color = "firebrick", linetype = "dotted", lwd = .3)



### 3.1.2 Replace the default ggplot2 theme

To further demonstrate ggplot’s versatility, let’s enhance the appearance by removing the default grayish {ggplot2} style and setting a different built-in theme, such as theme\_bw(). By using theme\_set(), all subsequent plots will adopt the same black-and-white theme. This adjustment will notably enhance the appearance of the red points!

theme\_set(theme\_bw())  
  
g + geom\_point(color = "firebrick")



For further details on using built-in themes and customizing themes, refer to the section [“Working with Themes”](#themes). Starting from the next chapter, we’ll also utilize the theme() function to customize specific elements of the theme.

|  |
| --- |
| Remember! |
| theme() is a crucial command for manually adjusting various theme elements such as texts, rectangles, and lines. |

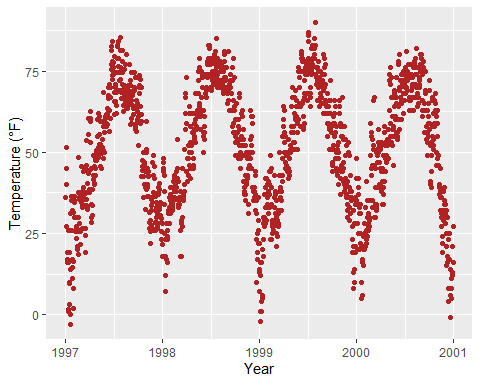
To explore the numerous details of a ggplot theme that can be modified, refer to the extensive list available [here](https://ggplot2.tidyverse.org/reference/theme.html). Take your time, as it’s a comprehensive list!

# 4. Working with Axes

## 4.1 Change Axis Titles

To add clear and descriptive labels to the axes, we can utilize the labs() function. This function allows us to provide a character string for each label we wish to modify, such as x and y:

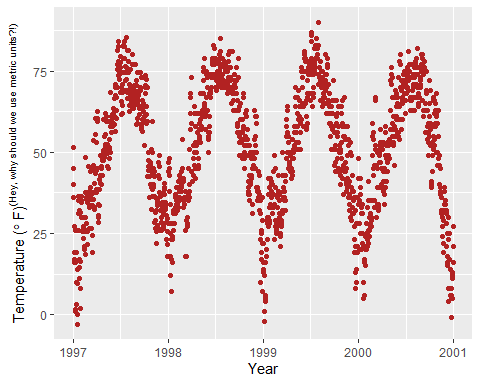
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)")



|  |
| --- |
| xlab() and ylab() |
| You also can add axis titles by using xlab() and ylab(). Click to see example.  ggplot(chic, aes(x = date, y = temp)) +  geom\_point(color = "firebrick") +  xlab("Year") +  ylab("Temperature (°F)") |

Typically, you can specify symbols by directly adding the symbol itself (e.g., “°”). However, the code below also enables the addition of not only symbols but also features like superscripts:

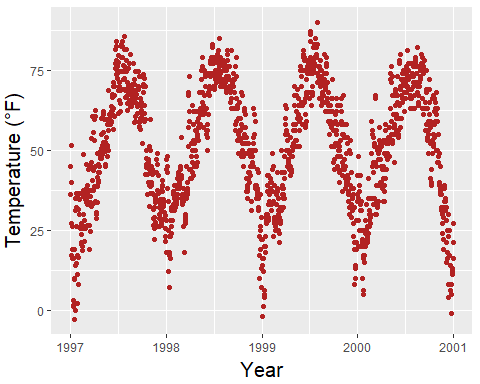
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = expression(paste("Temperature (", degree ~ F, ")"^"(Hey, why should we use metric units?!)")))



## 4.2 Increase Space between Axis and Axis Titles

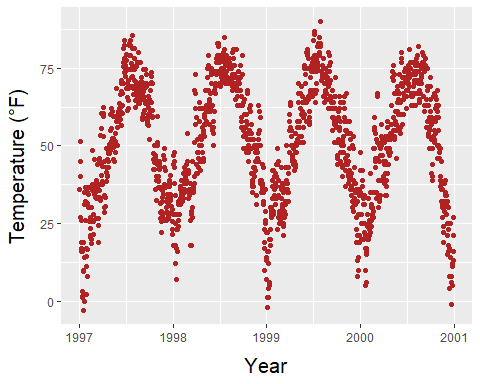
theme() is a crucial command for adjusting specific theme elements such as texts, titles, boxes, symbols, backgrounds, and more. We’ll be utilizing this command extensively! Initially, we’ll focus on modifying text elements. We can customize the properties of all text elements or specific ones, such as axis titles, by overriding the default element\_text() within the theme() call:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.title.x = element\_text(vjust = 0, size = 15),  
 axis.title.y = element\_text(vjust = 2, size = 15))



The vjust parameter controls vertical alignment and typically ranges between 0 and 1, but you can also specify values outside that range. It’s worth noting that even when adjusting the position of the axis title along the y-axis horizontally, we still need to specify vjust (which is correct from the perspective of the label’s alignment). Additionally, you can modify the distance by specifying the margin for both text elements:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.title.x = element\_text(margin = margin(t = 10), size = 15),  
 axis.title.y = element\_text(margin = margin(r = 10), size = 15))



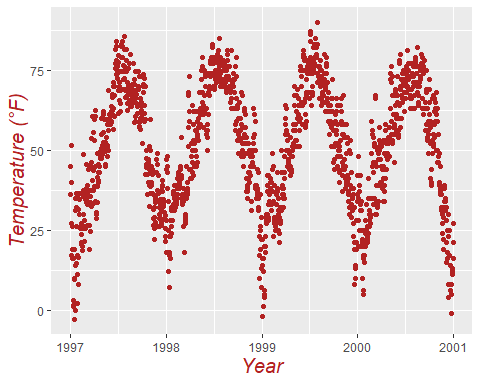
The labels t and r within the margin() object correspond to *top* and *right*, respectively. Alternatively, you can specify all four margins using margin(t, r, b, l). It’s important to note that we need to adjust the right margin to modify the space on the y-axis, not the bottom margin.

|  |
| --- |
| Having trouble with Margins? |
| A helpful mnemonic for remembering the order of the margin sides is “*t*-*r*-ou-*b*-*l*-e”. |

## 4.3 Change Aesthetics of Axis Titles

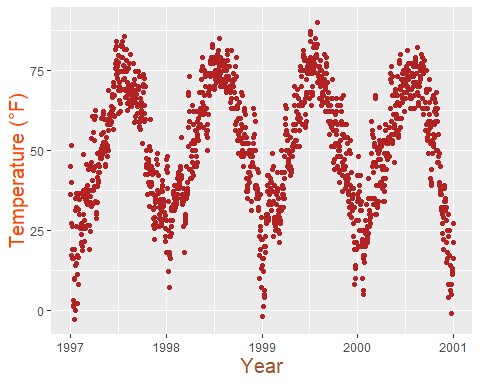
Once more, we utilize the theme() function to modify the axis.title element and/or its subordinated elements, axis.title.x and axis.title.y. Within the element\_text() function, we can override defaults for properties such as size, color, and face:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 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.

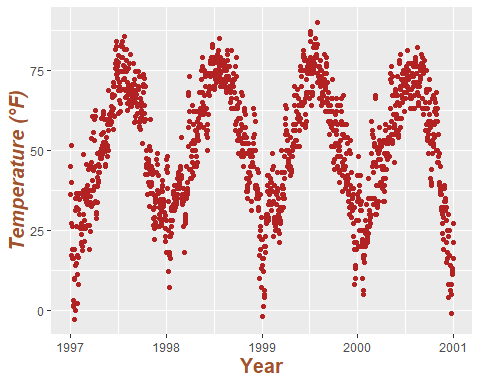
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.title.x = element\_text(color = "sienna", size = 15),  
 axis.title.y = element\_text(color = "orangered", size = 15))



|  |
| --- |
| Customising Invidual Axis |
| You could also employ a combination of axis.title and axis.title.y, as axis.title.x inherits values from axis.title. Expand to See the example below:  ggplot(chic, aes(x = date, y = temp)) +  geom\_point(color = "firebrick") +  labs(x = "Year", y = "Temperature (°F)") +  theme(axis.title = element\_text(color = "sienna", size = 15),  axis.title.y = element\_text(color = "orangered", size = 15)) |

You can adjust some properties for both axis titles simultaneously, while modifying others exclusively for one axis or individual properties for each axis title:

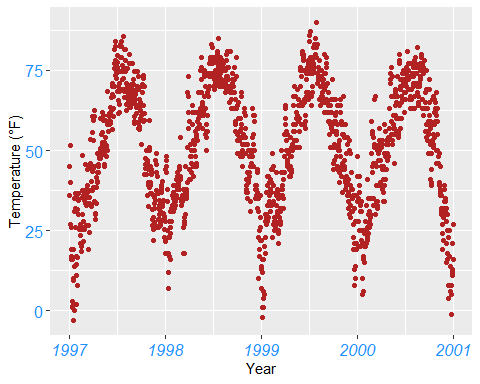
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.title = element\_text(color = "sienna", size = 15, face = "bold"),  
 axis.title.y = element\_text(face = "bold.italic"))



## 4.4 Change Aesthetics of Axis Text

Likewise, you can alter the appearance of the axis text (i.e., the numbers) by utilizing axis.text and/or its subordinated elements, axis.text.x and axis.text.y:

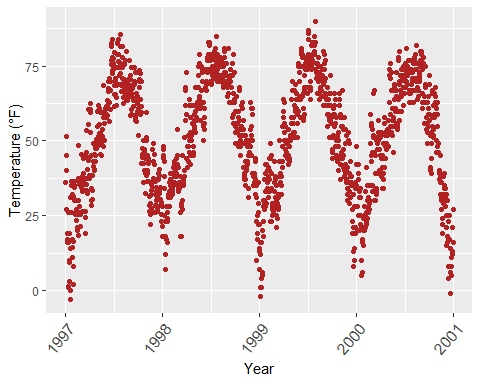
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.text = element\_text(color = "dodgerblue", size = 12),  
 axis.text.x = element\_text(face = "italic"))



## 4.5 Rotate Axis Text

You can rotate any text elements by specifying an angle. Subsequently, you can adjust the position of the text horizontally (0 = left, 1 = right) and vertically (0 = top, 1 = bottom) using hjust and vjust:

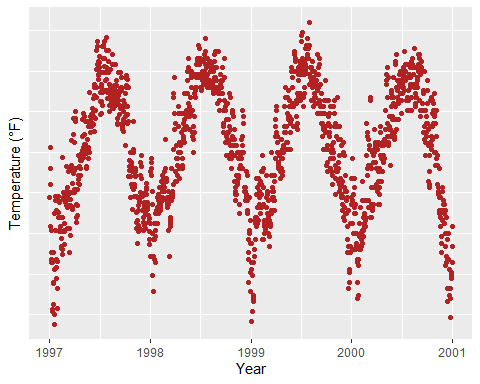
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.text.x = element\_text(angle = 50, vjust = 1, hjust = 1, size = 12))



## 4.6 Removing Axis Text & Ticks

There might be rare occasions where you need to remove axis text and ticks. Here’s how you can achieve it:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.ticks.y = element\_blank(),  
 axis.text.y = element\_blank())



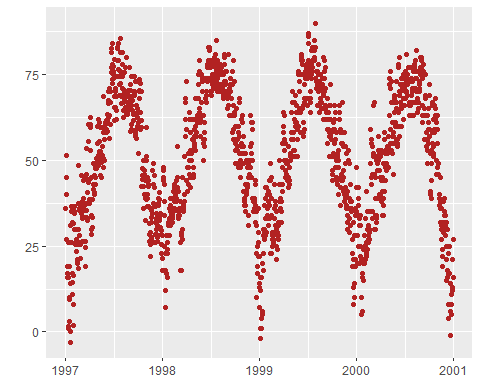
I’ve introduced three theme elements—text, lines, and rectangles—but there’s actually one more: element\_blank(), which removes the element entirely. However, it’s not considered an official element like the others.

|  |
| --- |
| Removing Theme Element |
| If you wish to remove a theme element entirely, you can always use element\_blank(). |

## 4.7 Removing Axis Titles

We can use theme\_blank(), but it’s much simpler to just omit the label in the labs() (or xlab()) call:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = NULL, y = "")



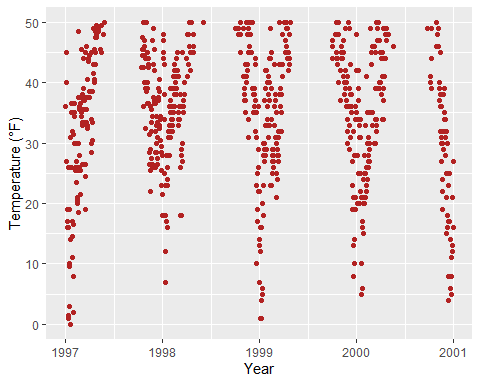
|  |
| --- |
| RAnother Tip! |
| Note that NULL removes the element (similarly to element\_blank()), while empty quotes "" will keep the spacing for the axis title but print nothing. |

## 4.8 Limiting Axis Range

Occasionally, you may want to focus on a specific range of your data without altering the dataset itself. You can accomplish this with ease:

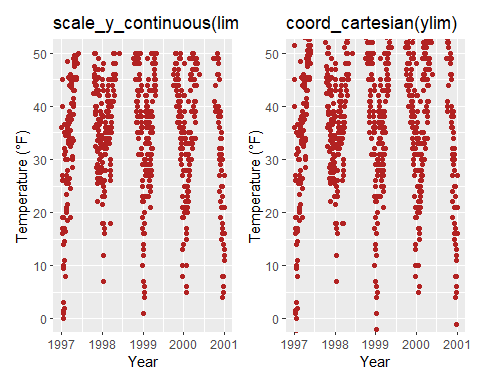
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 ylim(c(0, 50))

Warning: Removed 777 rows containing missing values or values outside the scale range  
(`geom\_point()`).



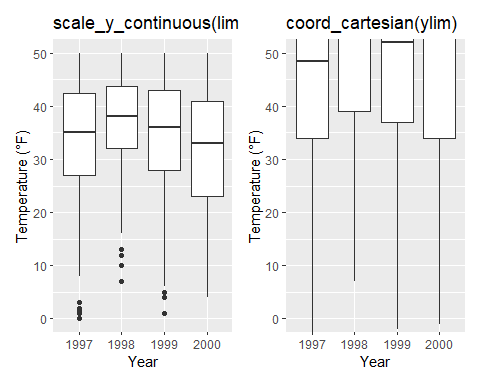
Alternatively, you can utilize scale\_y\_continuous(limits = c(0, 50)) or coord\_cartesian(ylim = c(0, 50)). The former removes all data points outside the specified range, while the latter adjusts the visible area, similar to ylim(c(0, 50)). At first glance, it may seem that both approaches yield the same result. However, there is an important distinction—compare the following two plots:

Warning: Removed 777 rows containing missing values or values outside the scale range  
(`geom\_point()`).



You may have noticed 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 effectively illustrates the concept of subsetting (left) versus zooming (right). To demonstrate why this distinction is significant, let’s examine a different chart type: a box plot.

Warning: Removed 777 rows containing non-finite outside the scale range  
(`stat\_boxplot()`).

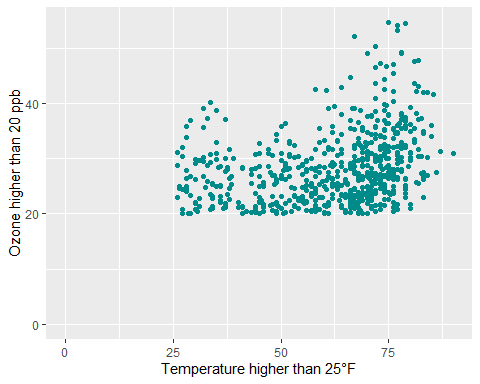


Indeed, because scale\_x|y\_continuous() subsets the data first, we obtain completely different (and potentially incorrect, especially if this was not our intention) estimates for the box plots! This realization highlights the importance of ensuring data integrity throughout the plotting process. It’s crucial to avoid inadvertently manipulating the data while plotting, as it could lead to inaccurate summary statistics reported in your report, paper, or thesis.

## 4.9 Forcing Plot to Start at Origin

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

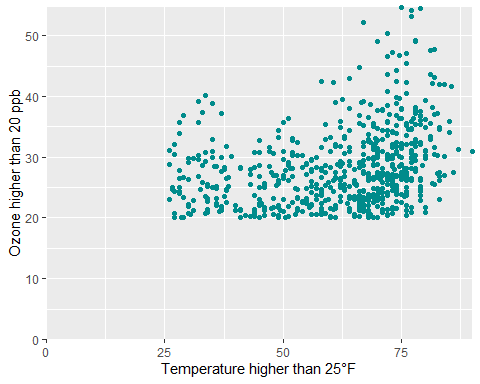
chic\_high <- dplyr::filter(chic, temp > 25, o3 > 20)  
  
ggplot(chic\_high, aes(x = temp, y = o3)) +  
 geom\_point(color = "darkcyan") +  
 labs(x = "Temperature higher than 25°F",  
 y = "Ozone higher than 20 ppb") +  
 expand\_limits(x = 0, y = 0)



|  |
| --- |
| Another Way using coord\_cartesian(xlim = c(0, NA), ylim = c(0, NA)) |
| Using coord\_cartesian(xlim = c(0, NA), ylim = c(0, NA)) will produce the same result. CLICK to See the example below:  chic\_high <- dplyr::filter(chic, temp > 25, o3 > 20)  ggplot(chic\_high, aes(x = temp, y = o3)) +  geom\_point(color = "darkcyan") +  labs(x = "Temperature higher than 25°F",  y = "Ozone higher than 20 ppb") +  coord\_cartesian(xlim = c(0, NA), ylim = c(0, NA)) |

But we can also ensure that it *truly* starts at the origin!

ggplot(chic\_high, aes(x = temp, y = o3)) +  
 geom\_point(color = "darkcyan") +  
 labs(x = "Temperature higher than 25°F",  
 y = "Ozone higher than 20 ppb") +  
 expand\_limits(x = 0, y = 0) +  
 coord\_cartesian(expand = FALSE, clip = "off")



|  |
| --- |
| Tip |
| The clip = "off" argument in any coordinate system, always starting with coord\_\*, enables drawing outside of the panel area. |

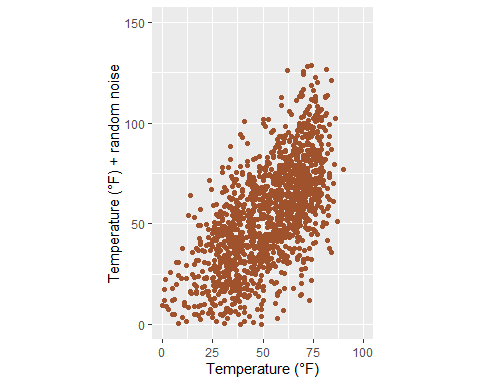
Here, I invoke it to ensure that the tick marks at c(0, 0) remain intact and are not truncated. For further insights, refer to the [Twitter thread by Claus Wilke](https://twitter.com/clauswilke/status/991542952802619392?lang=en).

## 4.10 Axes with Same Scaling

For demonstration purposes, let’s plot temperature against temperature with some random noise. The coord\_equal() function provides a coordinate system with a specified ratio, representing the number of units on the y-axis equivalent to one unit on the x-axis. By default, ratio = 1 ensures that one unit on the x-axis is the same length as one unit on the y-axis:

ggplot(chic, aes(x = temp, y = temp + rnorm(nrow(chic), sd = 20))) +  
 geom\_point(color = "sienna") +  
 labs(x = "Temperature (°F)", y = "Temperature (°F) + random noise") +  
 xlim(c(0, 100)) + ylim(c(0, 150)) +  
 coord\_fixed()

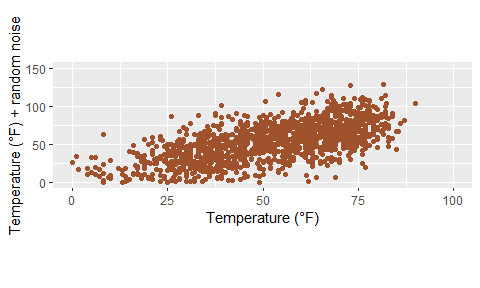
Warning: Removed 55 rows containing missing values or values outside the scale range  
(`geom\_point()`).



Ratios higher than one result in units on the y-axis being longer than units on the x-axis, while ratios lower than one have the opposite effect:

ggplot(chic, aes(x = temp, y = temp + rnorm(nrow(chic), sd = 20))) +  
 geom\_point(color = "sienna") +  
 labs(x = "Temperature (°F)", y = "Temperature (°F) + random noise") +  
 xlim(c(0, 100)) + ylim(c(0, 150)) +  
 coord\_fixed(ratio = 1/5)

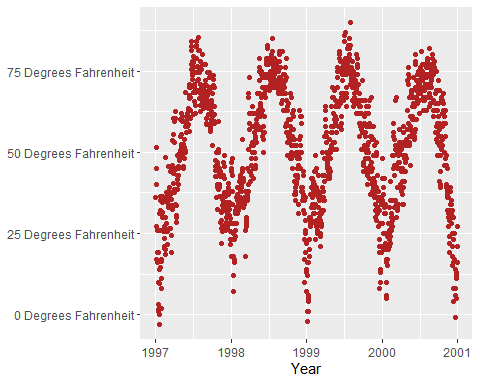
Warning: Removed 50 rows containing missing values or values outside the scale range  
(`geom\_point()`).



## 4.11 Using a Function to Alter Labels

Occasionally, it’s useful to slightly modify your labels, such as adding units or percent signs, without altering your underlying data. You can achieve this using a function:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = NULL) +  
 scale\_y\_continuous(label = function(x) {return(paste(x, "Degrees Fahrenheit"))})

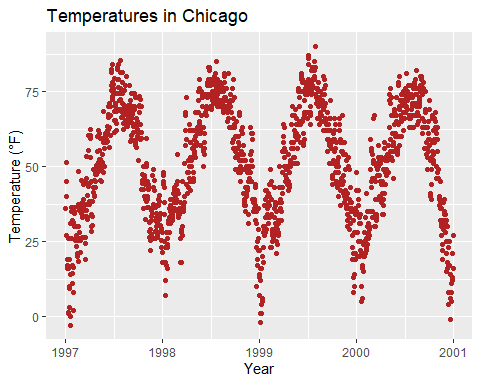


# 5. Working with Titles

## 5.1 Add a Title

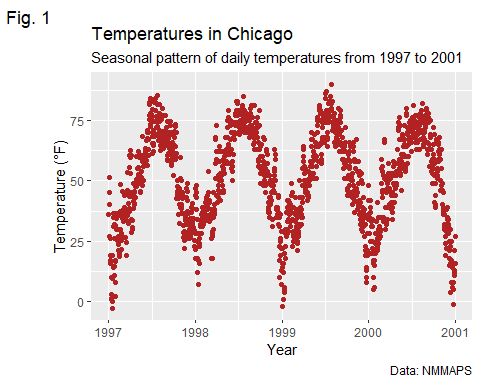
We can add a title by using the ggtitle() function:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 ggtitle("Temperatures in Chicago")



Alternatively, you can utilize labs(). Here, you can include multiple arguments, such as a subtitle, a caption, and a tag, in addition to axis titles as demonstrated earlier:

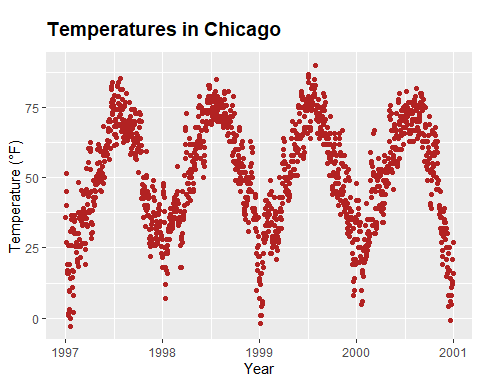
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)",  
 title = "Temperatures in Chicago",  
 subtitle = "Seasonal pattern of daily temperatures from 1997 to 2001",  
 caption = "Data: NMMAPS",  
 tag = "Fig. 1")



## 5.2 Making Title Bold & Adding a Space at the Baseline

Once again, to adjust the properties of a theme element, we employ the theme() function. Similar to modifying text elements like axis.title and axis.text, we can alter the font face and margin for the title. These modifications apply not only to the title but also to other labels such as plot.subtitle, plot.caption, plot.tag, legend.title, legend.text, axis.title, and axis.text.

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)",  
 title = "Temperatures in Chicago") +  
 theme(plot.title = element\_text(face = "bold",  
 margin = margin(10, 0, 10, 0),  
 size = 14))

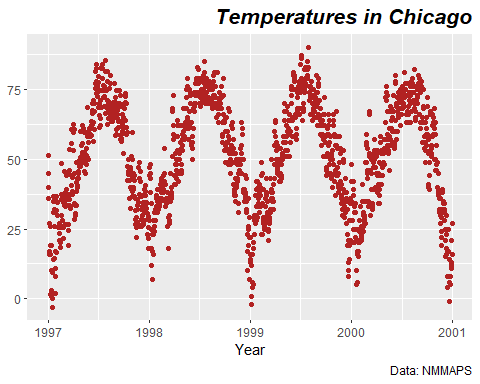


|  |
| --- |
| Having trouble with Margins? |
| A helpful mnemonic for remembering the order of the margin sides is “*t*-*r*-ou-*b*-*l*-e”. |

## 5.3 Adjusting Position of Titles

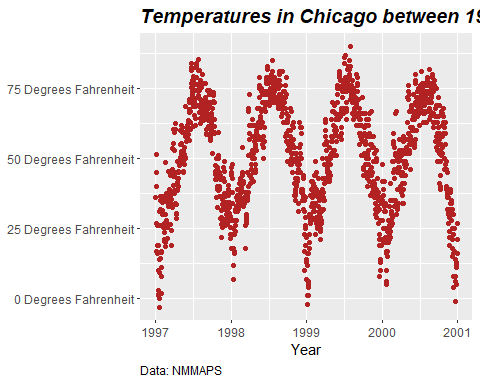
The general alignment (left, center, right) is controlled by hjust (horizontal adjustment):

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = NULL,  
 title = "Temperatures in Chicago",  
 caption = "Data: NMMAPS") +  
 theme(plot.title = element\_text(hjust = 1, size = 16, face = "bold.italic"))

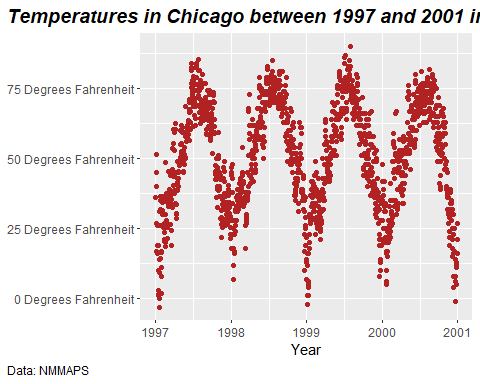


Certainly, it’s also possible to adjust the vertical alignment, which is controlled by vjust. Since 2019, users have been able to specify the alignment of the title, subtitle, and caption either based on the panel area (the default) or the plot margin via plot.title.position and plot.caption.position. The latter is often the preferred choice from a design perspective, as it yields better results in most cases. Many users have expressed satisfaction with this new feature, particularly as it addresses issues with alignment, especially when dealing with very long y-axis labels:

(g <- ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 scale\_y\_continuous(label = function(x) {return(paste(x, "Degrees Fahrenheit"))}) +  
 labs(x = "Year", y = NULL,  
 title = "Temperatures in Chicago between 1997 and 2001 in Degrees Fahrenheit",  
 caption = "Data: NMMAPS") +  
 theme(plot.title = element\_text(size = 14, face = "bold.italic"),  
 plot.caption = element\_text(hjust = 0)))



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



## 5.4 Using a Non-Traditional Font in Your Title

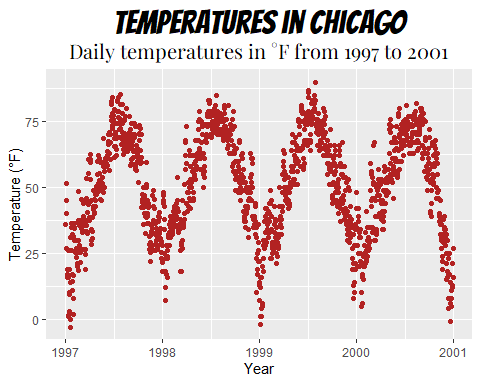
You can incorporate different fonts, not just the default one provided by ggplot (which can vary between operating systems). Several packages facilitate the usage of fonts installed on your machine, such as the [showtext package](https://github.com/yixuan/showtext), which simplifies the utilization of various font types (TrueType, OpenType, Type 1, web fonts, etc.) in R plots.

Once the package is loaded, you’ll need to import the desired font, which must also be installed on your device. I often utilize [Google fonts](https://fonts.google.com/), which can be imported using the font\_add\_google() function. However, you can add other fonts using font\_add() as well. It’s important to note that even when using Google fonts, you must install the font and restart RStudio to apply it. if you found any warnings after doing all the steps, or the fonts aren’t working. Just install extrafont package and run font\_import() function to import all the fonts in your system. and then loadfonts(device = "win", quiet = TRUE) to load the fonts. It’ll work like a charm. You can also check the available fonts in your system by running fonts().

library(showtext)  
library(extrafont)  
font\_add\_google("Playfair Display", ## name of Google font  
 "Playfair Display") ## name that will be used in R  
font\_add\_google("Bangers", "Bangers")  
loadfonts(device = "win", quiet = TRUE)

Now, we can use those font families by theme() function:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)",  
 title = "Temperatures in Chicago",  
 subtitle = "Daily temperatures in °F from 1997 to 2001") +  
 theme(plot.title = element\_text(family = "Bangers", hjust = .5, size = 25),  
 plot.subtitle = element\_text(family = "Playfair Display", hjust = .5, size = 15))



You can also establish a non-default font for all text elements of your plots. For more details, refer to the section [“Working with Themes”](#themes). In this case, I’ll use *Roboto Condensed* as the new font for all subsequent plots.

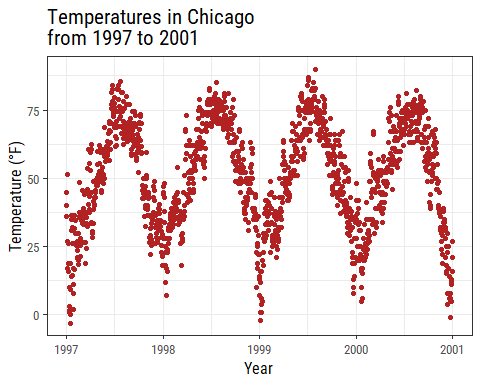
font\_add\_google("Roboto Condensed", "Roboto Condensed")  
theme\_set(theme\_bw(base\_size = 12, base\_family = "Roboto Condensed"))

(Previously, this tutorial utilized the [{extrafont} package](https://cran.r-project.org/web/packages/extrafont/README.html), which performed admirably until last year. However, suddenly I encountered issues where I couldn’t add any new fonts, and even after acquiring a new laptop, the package failed to detect any fonts altogether. As an alternative, I typically recommend the [{ragg} package](https://ragg.r-lib.org/) now. However, I encountered difficulties in making it work for my homepage. Therefore, I’m utilizing the {showtext} package, which is also excellent, albeit with a key distinction: you need to explicitly import the font you wish to use with {showtext}. Nonetheless, it appears that there are some technical challenges that are not optimally resolved by {showtext} (as mentioned in [this Twitter thread](https://twitter.com/thomasp85/status/1355083725156077571)), so you may want to consider using the package only as a last resort.)

## 5.5 Adjusting Spacing in Multi-Line Text

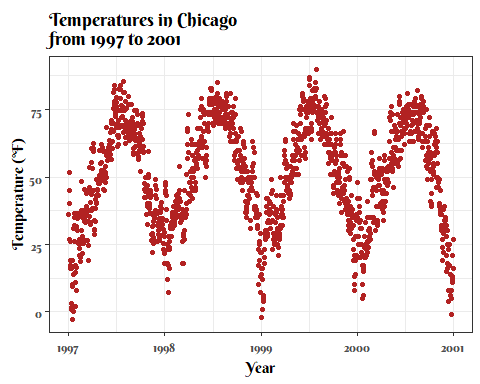
To modify the spacing between lines, you can utilize the lineheight argument. In the following example, I’ve compressed the lines together (lineheight < 1).

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 ggtitle("Temperatures in Chicago\nfrom 1997 to 2001") +  
 theme(plot.title = element\_text(lineheight = .8, size = 16))



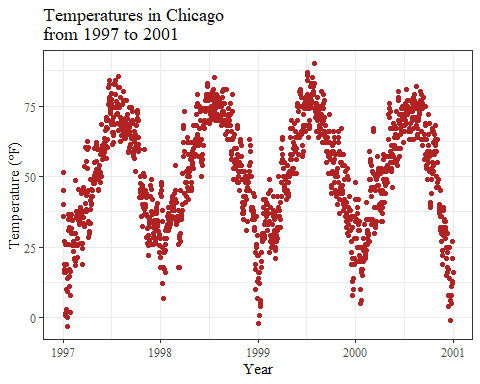
Now You can Change fonts on the fly!

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 ggtitle("Temperatures in Chicago\nfrom 1997 to 2001") +  
 theme\_bw(base\_family = "Berkshire Swash")



Or, Change it to Traditional Times New Roman:

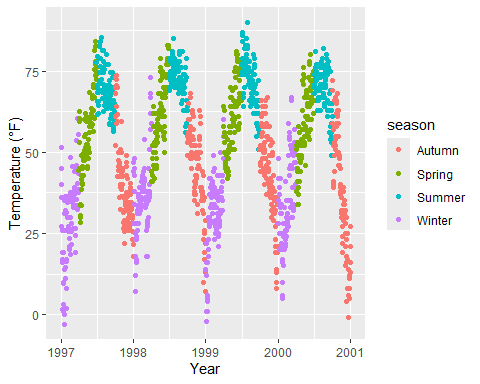
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 ggtitle("Temperatures in Chicago\nfrom 1997 to 2001") +  
 theme\_bw(base\_family = "Times New Roman")



# 6. Working with Legends

In this section, we will color code the plot based on the season. Or, to phrase it more in the style of ggplot: we’ll map the variable season to the aesthetic color. One of the advantages of {ggplot2} is that it automatically adds a legend when mapping a variable to an aesthetic. As a result, the legend title defaults to what we specified in the color argument:

ggplot(chic,  
 aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)")

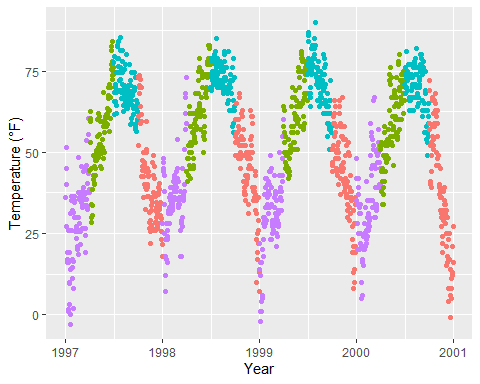


## 6.1 Disabling the Legend

One of the most common questions is: “How do I remove the legend?”

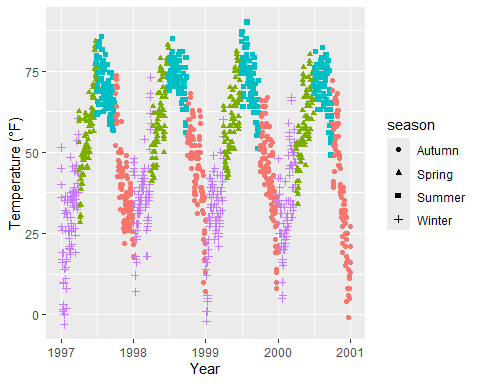
It’s quite straightforward and always effective with theme(legend.position = "none"):

ggplot(chic,  
 aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.position = "none")



You can also utilize guides(color = "none") or scale\_color\_discrete(guide = "none"), depending on the specific case. While altering the theme element removes all legends at once, you can selectively remove specific legends using the latter options while keeping others:

ggplot(chic,  
 aes(x = date, y = temp,  
 color = season, shape = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 guides(color = "none")

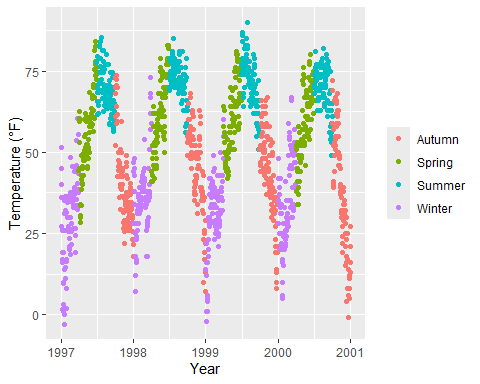


Here, for example, we retain the legend for the shapes while discarding the one for the colors.

## 6.2 Eliminating Legend Titles

As we’ve previously learned, utilize element\_blank() to render *nothing*:

ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.title = element\_blank())

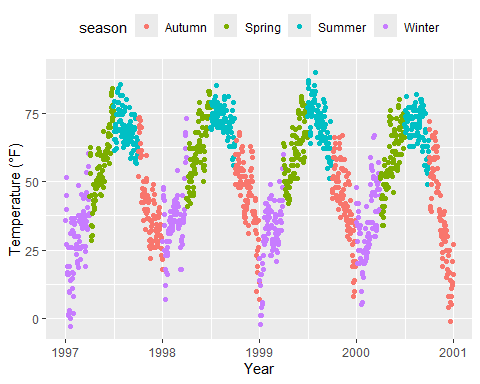


|  |
| --- |
| Other Ways to remove Legend Titles |
| You can achieve the same outcome by setting the legend name to NULL, either through scale\_color\_discrete(name = NULL) or labs(color = NULL). Expand to see examples.  ggplot(chic, aes(x = date, y = temp, color = season)) +  geom\_point() +  labs(x = "Year", y = "Temperature (°F)") +  scale\_color\_discrete(name = NULL)    ggplot(chic, aes(x = date, y = temp, color = season)) +  geom\_point() +  labs(x = "Year", y = "Temperature (°F)") +  labs(color = NULL) |

## 6.3 Adjusting Legend Position

To relocate the legend from its default position on the right side, you can use the legend.position argument within theme. Available positions include “top”, “right” (the default), “bottom”, and “left”.

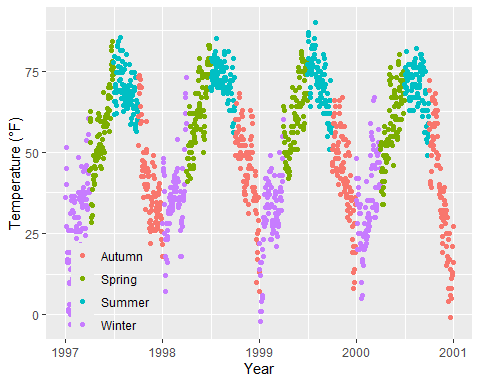
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.position = "top")



You can also position the legend inside the panel by specifying a vector with relative x and y coordinates ranging from 0 (left or bottom) to 1 (right or top):

ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)",  
 color = NULL) +  
 theme(legend.position = c(.15, .15),  
 legend.background = element\_rect(fill = "transparent"))

Warning: A numeric `legend.position` argument in `theme()` was deprecated in ggplot2  
3.5.0.  
ℹ Please use the `legend.position.inside` argument of `theme()` instead.

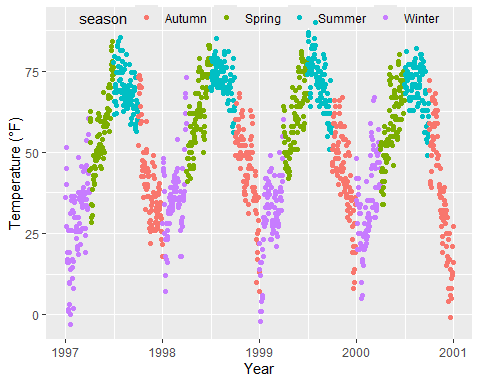


Here, I also override the default white legend background with a transparent fill to ensure the legend doesn’t obscure any data points.

## 6.4 Modifying Legend Direction

By default, the legend direction is vertical. However, when you select either the “top” or “bottom” position, it becomes horizontal. Nevertheless, you can freely switch the direction as desired:

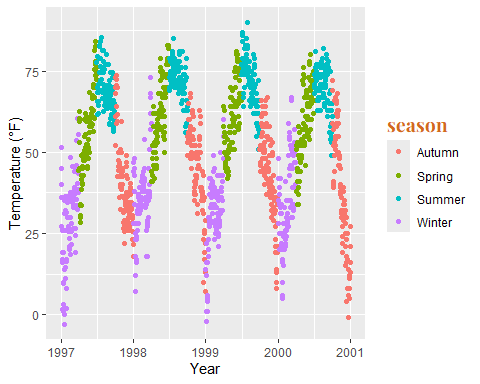
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.position = c(.5, .97),  
 legend.background = element\_rect(fill = "transparent")) +  
 guides(color = guide\_legend(direction = "horizontal"))



## 6.5 Change Style of the Legend Title

You can customize the appearance of the legend title by adjusting the theme element legend.title:

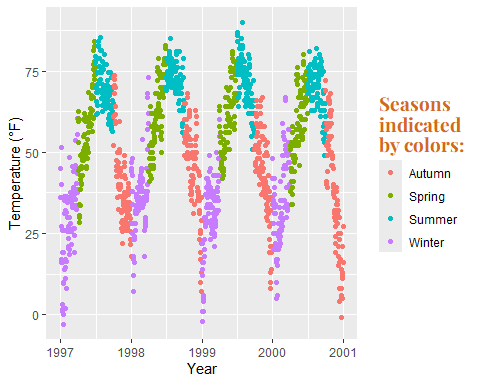
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.title = element\_text(family = "Playfair Display",  
 color = "chocolate",  
 size = 14, face = "bold"))



## 6.6 Modifying Legend Title

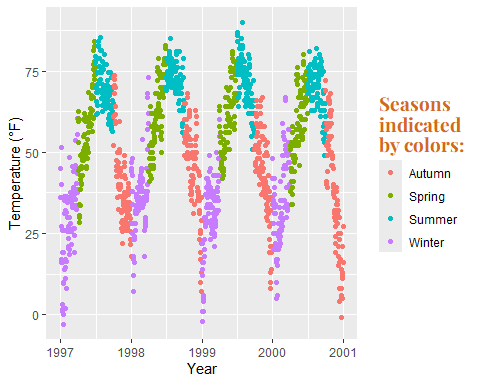
The simplest method to change the title of the legend is through the labs() layer:

ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)",  
 color = "Seasons\nindicated\nby colors:") +  
 theme(legend.title = element\_text(family = "Playfair Display",  
 color = "chocolate",  
 size = 14, face = "bold"))



You can adjust the legend details using scale\_color\_discrete(name = "title") or guides(color = guide\_legend("title")):

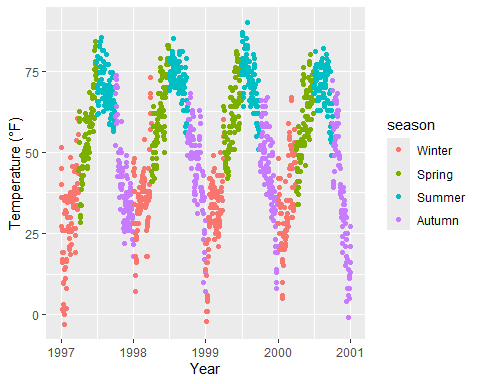
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.title = element\_text(family = "Playfair Display",  
 color = "chocolate",  
 size = 14, face = "bold")) +  
 scale\_color\_discrete(name = "Seasons\nindicated\nby colors:")



## 6.7 Rearrange Order of Legend Keys

This can be accomplished by changing the levels of season:

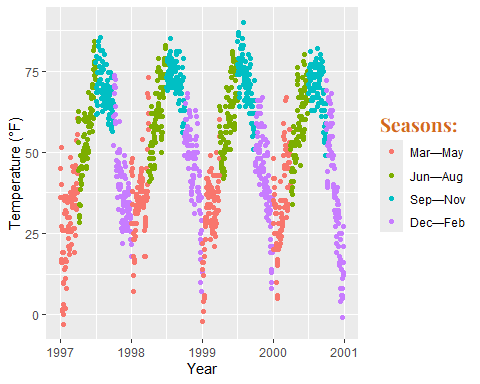
chic$season <-  
 factor(chic$season,  
 levels = c("Winter", "Spring", "Summer", "Autumn"))  
  
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)")



## 6.8 Modify Legend Labels

To replace the seasons with the months they represent, provide a vector of names in the scale\_color\_discrete() call:

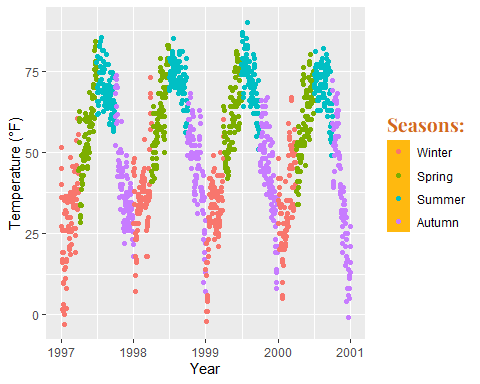
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 scale\_color\_discrete(  
 name = "Seasons:",  
 labels = c("Mar—May", "Jun—Aug", "Sep—Nov", "Dec—Feb")  
 ) +  
 theme(legend.title = element\_text(  
 family = "Playfair Display", color = "chocolate", size = 14, face = 2  
 ))



## 6.9 Adjust Background Boxes in the Legend

To alter the background color (fill) of the legend keys, we modify the setting for the theme element legend.key:

ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.key = element\_rect(fill = "darkgoldenrod1"),  
 legend.title = element\_text(family = "Playfair Display",  
 color = "chocolate",  
 size = 14, face = 2)) +  
 scale\_color\_discrete("Seasons:")

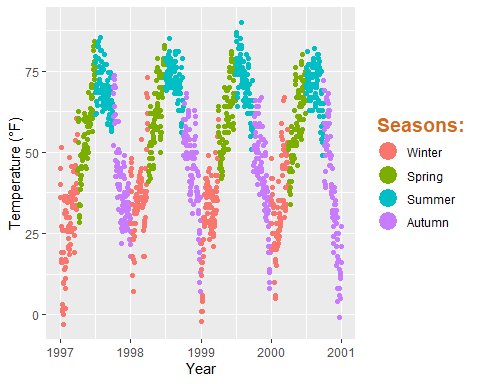


If you wish to remove them entirely, use fill = NA or fill = "transparent".

## 6.10 Adjust Size of Legend Symbols

The default size of points in the legend may cause them to appear too small, especially without boxes. To modify this, you can again use the guides layer as follows:

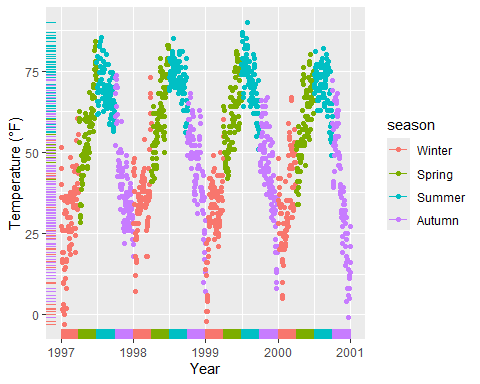
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.key = element\_rect(fill = NA),  
 legend.title = element\_text(color = "chocolate",  
 size = 14, face = 2)) +  
 scale\_color\_discrete("Seasons:") +  
 guides(color = guide\_legend(override.aes = list(size = 6)))



## 6.11 Exclude a Layer from the Legend

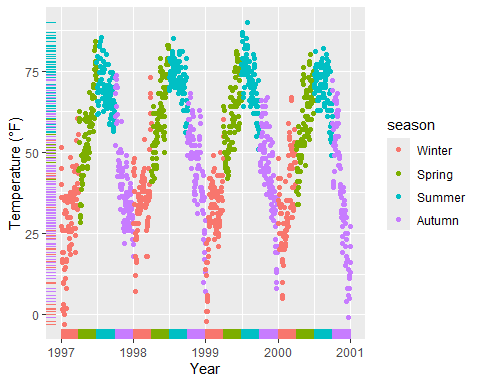
Suppose you have two different geometric layers mapped to the same variable, such as color being used as an aesthetic for both a point layer and a rug layer of the same data. By default, both the points and the “line” end up in the legend like this:

ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 geom\_rug()



You can utilize show.legend = FALSE to exclude a layer from the legend:

ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 geom\_rug(show.legend = FALSE)

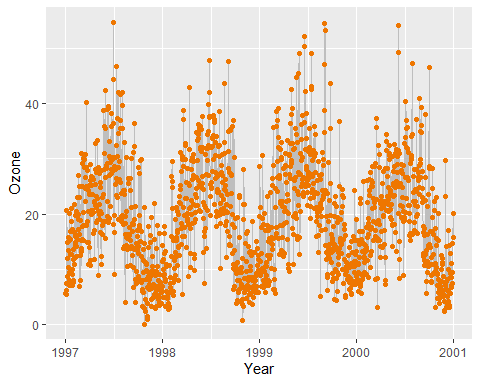


## 6.12 Manually Adding Legend Items

By default, {ggplot2} won’t add a legend unless you map aesthetics (color, size, etc.) to a variable. However, there are occasions where you may want to include a legend for clarity.

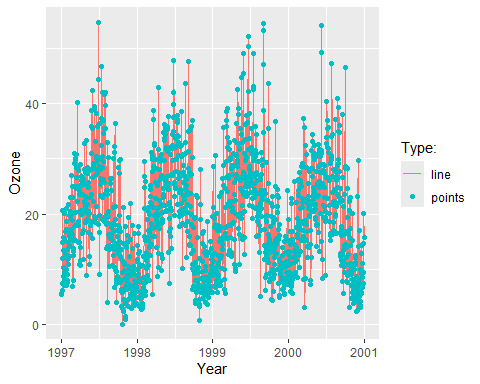
Here’s the default behavior:

ggplot(chic, aes(x = date, y = o3)) +  
 geom\_line(color = "gray") +  
 geom\_point(color = "darkorange2") +  
 labs(x = "Year", y = "Ozone")



To force a legend, we can map a guide to a *variable*. Here, we’re mapping the lines and the points using aes(), but we’re not mapping to a variable in our dataset. Instead, we’re using a single string for each, ensuring we get just one color for each.

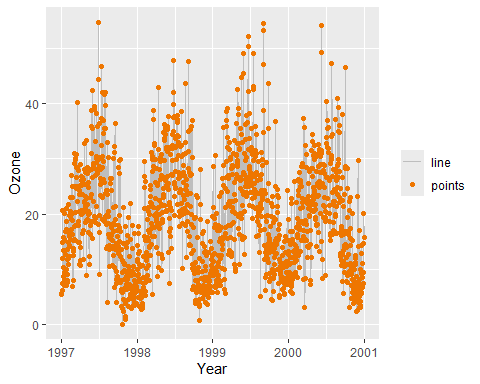
ggplot(chic, aes(x = date, y = o3)) +  
 geom\_line(aes(color = "line")) +  
 geom\_point(aes(color = "points")) +  
 labs(x = "Year", y = "Ozone") +  
 scale\_color\_discrete("Type:")



We’re getting close, but this is not what we want. We desire gray lines and red points. To change the colors, we use scale\_color\_manual(). Additionally, we override the legend aesthetics using the guide() function.

Now, we have a plot with gray lines and red points, as well as a single gray line and a single red point as legend symbols.

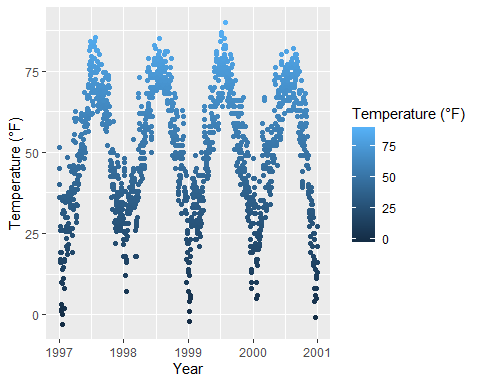
ggplot(chic, aes(x = date, y = o3)) +  
 geom\_line(aes(color = "line")) +  
 geom\_point(aes(color = "points")) +  
 labs(x = "Year", y = "Ozone") +  
 scale\_color\_manual(name = NULL,  
 guide = "legend",  
 values = c("points" = "darkorange2",  
 "line" = "gray")) +  
 guides(color = guide\_legend(override.aes = list(linetype = c(1, 0),  
 shape = c(NA, 16))))



## 6.13 Use Other Legend Styles

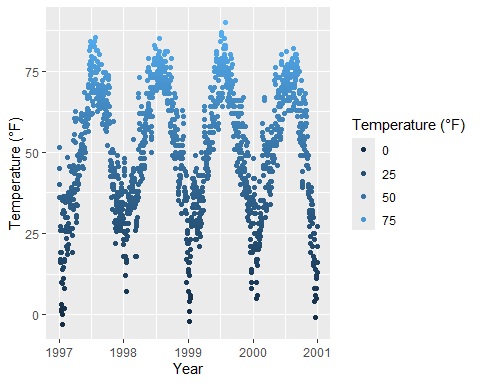
The default legend for categorical variables such as season is a guide\_legend(), as you have seen in several previous examples. However, if you map a continuous variable to an aesthetic, {ggplot2} will by default not use guide\_legend() but guide\_colorbar() (or guide\_colourbar()).

ggplot(chic,  
 aes(x = date, y = temp, color = temp)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F)")



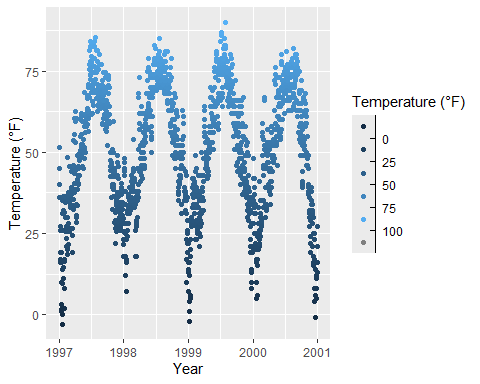
However, by using guide\_legend(), you can force the legend to display discrete colors for a given number of breaks as in the case of a categorical variable:

ggplot(chic,  
 aes(x = date, y = temp, color = temp)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F)") +  
 guides(color = guide\_legend())



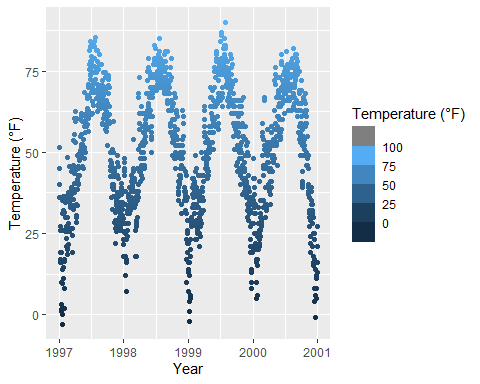
You can also utilize *binned scales*:

ggplot(chic,  
 aes(x = date, y = temp, color = temp)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F)") +  
 guides(color = guide\_bins())



… or binned scales as *discrete colorbars*:

ggplot(chic,  
 aes(x = date, y = temp, color = temp)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F)") +  
 guides(color = guide\_colorsteps())



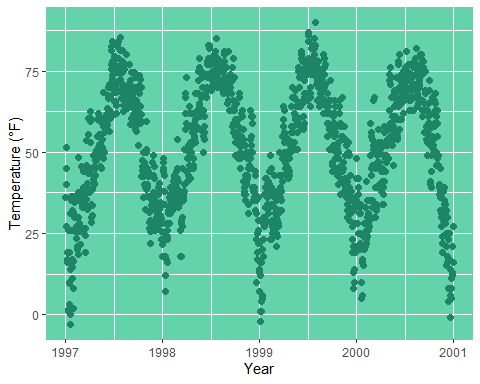
# 7. Working with Backgrounds & Grid Lines

To modify the overall appearance of your plot, you can use various functions. While altering the entire theme of your plot is one option (covered in detail in the [“Working with Themes”](#themes) section below), you can also make specific changes to individual elements such as backgrounds and grid lines.

## 7.1 Change the Panel Background Color

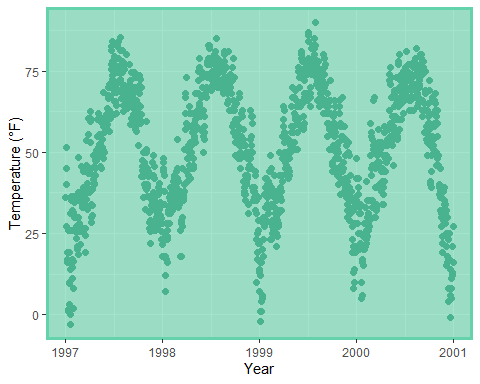
You can adjust the background color (fill) of the panel area (where the data is plotted) by modifying the theme element panel.background:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "#1D8565", size = 2) +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(panel.background = element\_rect(  
 fill = "#64D2AA", color = "#64D2AA", linewidth = 2)  
 )



Keep in mind that the true color — the outline of the panel background — didn’t change despite our specification. This is because there’s a layer on top of panel.background, namely panel.border. However, it’s important to use a transparent fill here; otherwise, your data will be hidden behind this layer. In the following example, I illustrate this by using a semitransparent hex color for the fill argument in element\_rect:

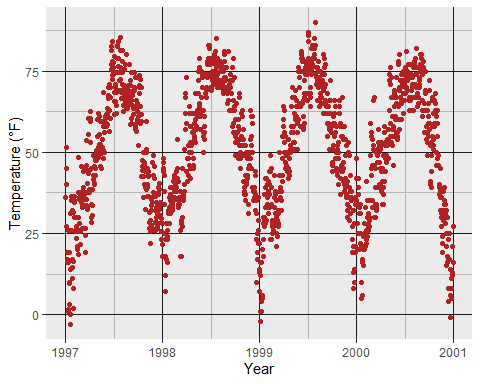
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "#1D8565", size = 2) +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(panel.border = element\_rect(  
 fill = "#64D2AA99", color = "#64D2AA", linewidth = 2)  
 )



## 7.2 Change Grid Lines

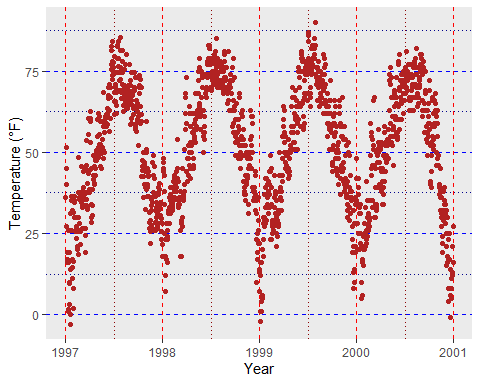
There are two types of grid lines: major grid lines indicating the ticks and minor grid lines between the major ones. You can customize both by overwriting the defaults for panel.grid or for each set of gridlines separately, panel.grid.major and panel.grid.minor.

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(panel.grid.major = element\_line(color = "gray10", linewidth = .5),  
 panel.grid.minor = element\_line(color = "gray70", linewidth = .25))



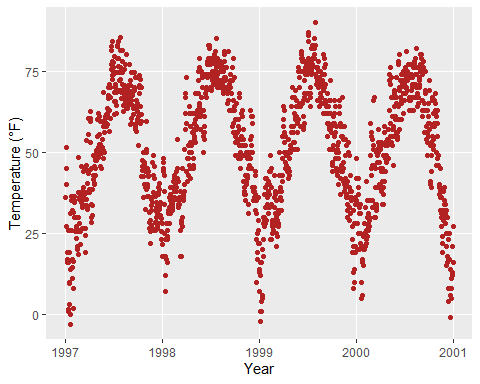
You can even specify settings for all four different levels of grid lines: major horizontal, major vertical, minor horizontal, and minor vertical.

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(panel.grid.major = element\_line(linewidth = .5, linetype = "dashed"),  
 panel.grid.minor = element\_line(linewidth = .25, linetype = "dotted"),  
 panel.grid.major.x = element\_line(color = "red1"),  
 panel.grid.major.y = element\_line(color = "blue1"),  
 panel.grid.minor.x = element\_line(color = "red4"),  
 panel.grid.minor.y = element\_line(color = "blue4"))

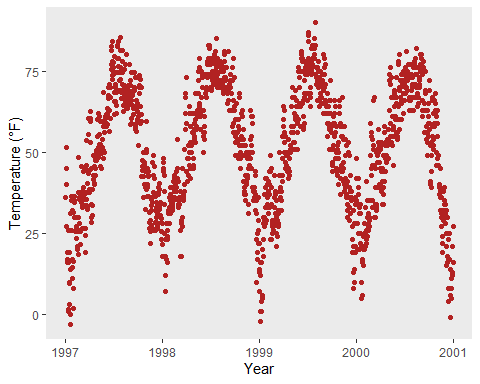


And, of course, you can remove some or all grid lines if you like. For instance, to remove all grid lines, you can set panel.grid = element\_blank(). Alternatively, you can remove only major or minor grid lines by specifying panel.grid.major or panel.grid.minor accordingly and setting them to element\_blank().

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(panel.grid.minor = element\_blank())



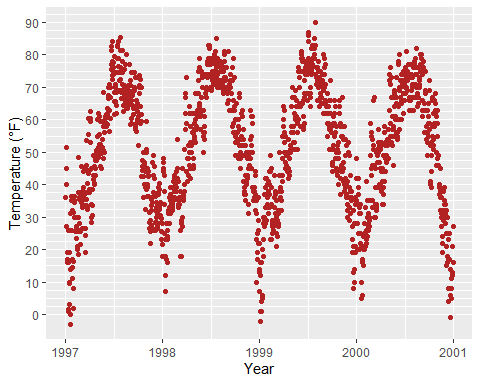
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(panel.grid = element\_blank())



## 7.3 Change Spacing of Gridlines

Furthermore, you can also define the breaks between both major and minor grid lines by specifying the breaks argument.

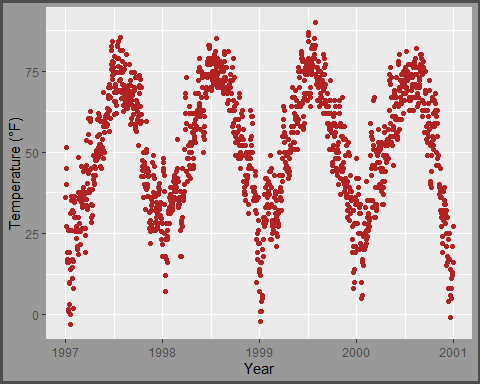
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 scale\_y\_continuous(breaks = seq(0, 100, 10),  
 minor\_breaks = seq(0, 100, 2.5))



## 7.4 Change the Plot Background Color

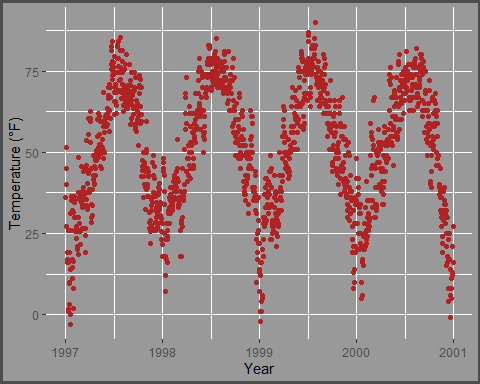
Similarly, to change the background color (fill) of the plot area, you can modify the theme element plot.background using the theme() function. This allows you to customize the appearance of the entire plot area according to your preferences.

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(plot.background = element\_rect(fill = "gray60",  
 color = "gray30", linewidth = 2))



You can achieve a unique background color by either setting the same colors in both panel.background and plot.background or by setting the background filling of the panel to "transparent" or NA. This customization can help you create visually appealing plots that match your design preferences.

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(panel.background = element\_rect(fill = NA),  
 plot.background = element\_rect(fill = "gray60",  
 color = "gray30", linewidth = 2))

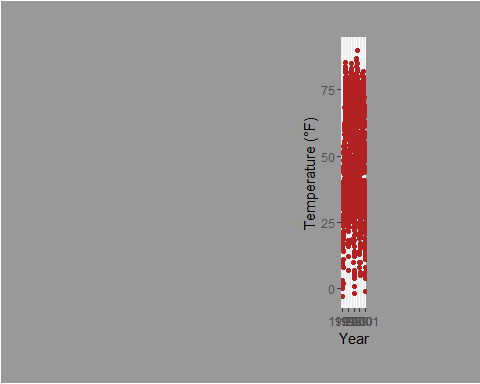


# 8. Working with Margins

Sometimes it is useful to add a little space to the plot margin. Similar to the previous examples, we can use an argument to the theme() function. In this case, the argument is plot.margin. As illustrated in the previous example where we changed the background color using plot.background, we can now add extra space to both the left and right.

The plot.margin argument can handle a variety of different units (cm, inches, etc.), but it requires the use of the unit function from the package grid to specify the units. You can either provide the same value for all sides (easiest via rep(x, 4)) or particular distances for each. Here, I am using a 1cm margin on the top and bottom, 3 cm margin on the right, and an 8 cm margin on the left.

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(plot.background = element\_rect(fill = "gray60"),  
 plot.margin = margin(t = 1, r = 3, b = 1, l = 8, unit = "cm"))



|  |
| --- |
| Having trouble with Margins? |
| A helpful mnemonic for remembering the order of the margin sides is “*t*-*r*-ou-*b*-*l*-e”. |

|  |
| --- |
| unit() instead of margin() |
| You can also use unit() instead of margin().  ggplot(chic, aes(x = date, y = temp)) +  geom\_point(color = "firebrick") +  labs(x = "Year", y = "Temperature (°F)") +  theme(plot.background = element\_rect(fill = "gray60"),  plot.margin = unit(c(1, 3, 1, 8), "cm")) |

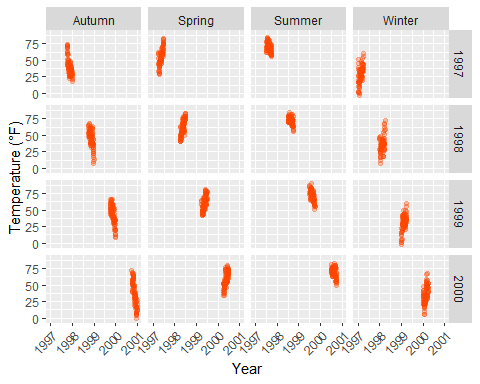
# 9. Working with Multi-Panel Plots

The {ggplot2} package offers two handy functions for creating multi-panel plots, called *facets*. They are related but have slight differences: facet\_wrap creates a ribbon of plots based on a single variable, while facet\_grid spans a grid of plots based on two variables.

## 9.1 Create a Grid of Small Multiples Based on Two Variables

When dealing with two variables, facet\_grid is the appropriate choice. In this function, the order of the variables determines the number of rows and columns in the grid:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "orangered", alpha = .3) +  
 theme(axis.text.x = element\_text(angle = 45, vjust = 1, hjust = 1)) +  
 labs(x = "Year", y = "Temperature (°F)") +  
 facet\_grid(year ~ season)

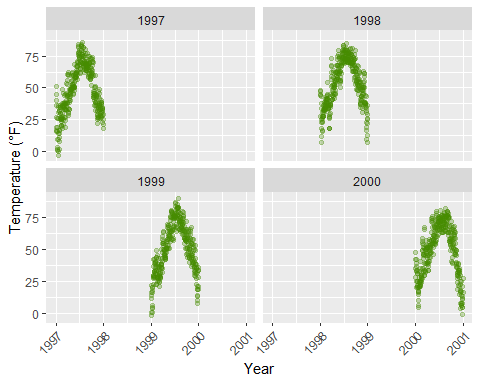


To switch from a row-based arrangement to a column-based one, you can modify facet\_grid(year ~ season) to facet\_grid(season ~ year).

## 9.2 Create Small Multiples Based on One Variable

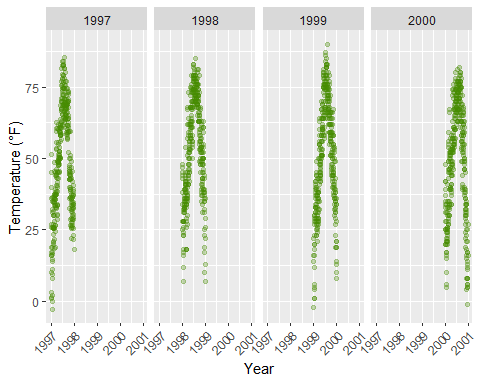
facet\_wrap creates a facet of a single variable, specified with a tilde in front: facet\_wrap(~ variable). The appearance of these subplots is determined by the arguments ncol and nrow:

g <-  
 ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "chartreuse4", alpha = .3) +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(axis.text.x = element\_text(angle = 45, vjust = 1, hjust = 1))  
  
g + facet\_wrap(~ year)



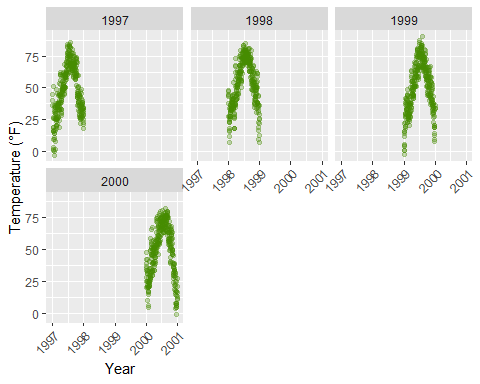
Accordingly, you can arrange the plots as you like, instead as a matrix in one row…

g + facet\_wrap(~ year, nrow = 1)



… or even as a asymmetric grid of plots:

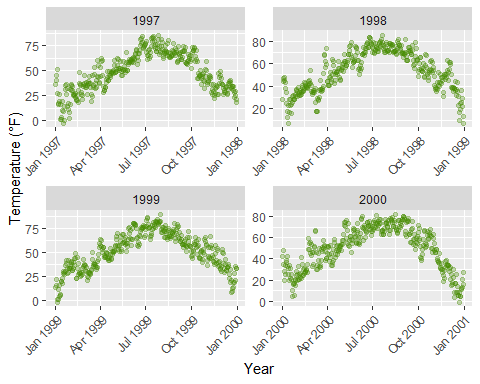
g + facet\_wrap(~ year, ncol = 3) + theme(axis.title.x = element\_text(hjust = .15))



## 9.3 Allow Axes to Roam Free

The default for multi-panel plots in {ggplot2} is to use equivalent scales in each panel. But sometimes you want to allow a panels own data to determine the scale. This is often not a good idea since it may give your user the wrong impression about the data. But sometimes it is indeed useful and to do this you can set scales = "free":

g + facet\_wrap(~ year, nrow = 2, scales = "free")

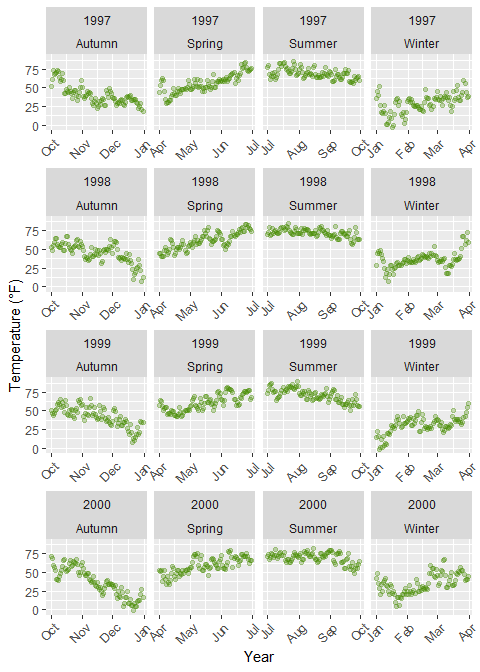


Note that both, x and y axes differ in their range!

#### 9.3.0.1 Use facet\_wrap with Two Variables

The function facet\_wrap can also take two variables:

g + facet\_wrap(year ~ season, nrow = 4, scales = "free\_x")



When using facet\_wrap you are still able to control the grid design: you can rearrange the number of plots per row and column and you can also let all axes roam free. In contrast, facet\_grid will also take a free argument but will only let it roam free per column or row:

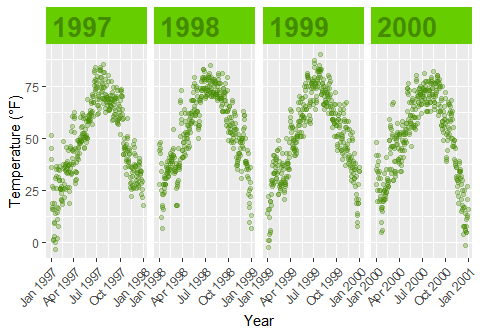
g + facet\_grid(year ~ season, scales = "free\_x")



## 9.4 Modify Style of Strip Texts

By using theme, you can modify the appearance of the strip text (i.e. the title for each facet) and the strip text boxes:

g + facet\_wrap(~ year, nrow = 1, scales = "free\_x") +  
 theme(strip.text = element\_text(face = "bold", color = "chartreuse4",  
 hjust = 0, size = 20),  
 strip.background = element\_rect(fill = "chartreuse3", linetype = "dotted"))

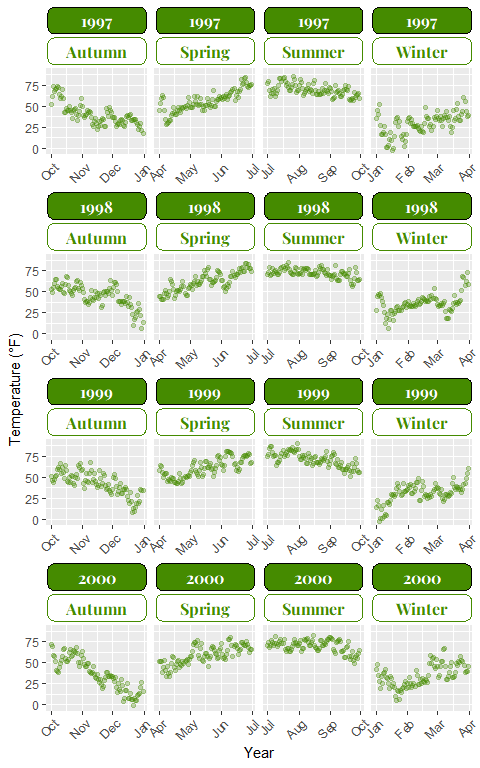


The following [two functions adapted from this answer by Claus Wilke](https://stackoverflow.com/questions/60332202/conditionally-fill-ggtext-text-boxes-in-facet-wrap), the author of the [{ggtext} package](https://wilkelab.org/ggtext/), allow to highlight specific labels in combination with element\_textbox() that is provided by {ggtext}.

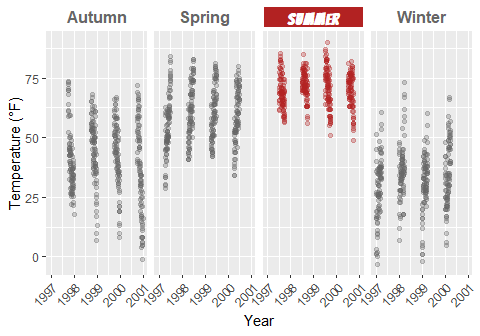
library(ggtext)  
library(purrr) ## for %||%  
  
element\_textbox\_highlight <- function(..., hi.labels = NULL, hi.fill = NULL,  
 hi.col = NULL, hi.box.col = NULL, hi.family = NULL) {  
 structure(  
 c(element\_textbox(...),  
 list(hi.labels = hi.labels, hi.fill = hi.fill, hi.col = hi.col, hi.box.col = hi.box.col, hi.family = hi.family)  
 ),  
 class = c("element\_textbox\_highlight", "element\_textbox", "element\_text", "element")  
 )  
}  
  
element\_grob.element\_textbox\_highlight <- function(element, label = "", ...) {  
 if (label %in% element$hi.labels) {  
 element$fill <- element$hi.fill %||% element$fill  
 element$colour <- element$hi.col %||% element$colour  
 element$box.colour <- element$hi.box.col %||% element$box.colour  
 element$family <- element$hi.family %||% element$family  
 }  
 NextMethod()  
}

Now you can use it and specify for example all striptexts:

g + facet\_wrap(year ~ season, nrow = 4, scales = "free\_x") +  
 theme(  
 strip.background = element\_blank(),  
 strip.text = element\_textbox\_highlight(  
 family = "Playfair Display", size = 12, face = "bold",  
 fill = "white", box.color = "chartreuse4", color = "chartreuse4",  
 halign = .5, linetype = 1, r = unit(5, "pt"), width = unit(1, "npc"),  
 padding = margin(5, 0, 3, 0), margin = margin(0, 1, 3, 1),  
 hi.labels = c("1997", "1998", "1999", "2000"),  
 hi.fill = "chartreuse4", hi.box.col = "black", hi.col = "white"  
 )  
 )



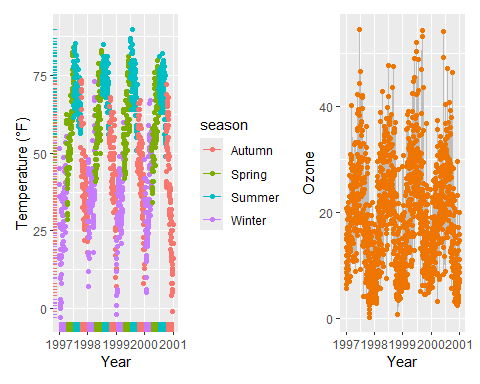
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(aes(color = season == "Summer"), alpha = .3) +  
 labs(x = "Year", y = "Temperature (°F)") +  
 facet\_wrap(~ season, nrow = 1) +  
 scale\_color\_manual(values = c("gray40", "firebrick"), guide = "none") +  
 theme(  
 axis.text.x = element\_text(angle = 45, vjust = 1, hjust = 1),  
 strip.background = element\_blank(),  
 strip.text = element\_textbox\_highlight(  
 size = 12, face = "bold",  
 fill = "white", box.color = "white", color = "gray40",  
 halign = .5, linetype = 1, r = unit(0, "pt"), width = unit(1, "npc"),  
 padding = margin(2, 0, 1, 0), margin = margin(0, 1, 3, 1),  
 hi.labels = "Summer", hi.family = "Bangers",  
 hi.fill = "firebrick", hi.box.col = "firebrick", hi.col = "white"  
 )  
 )



## 9.5 Create a Panel of Different Plots

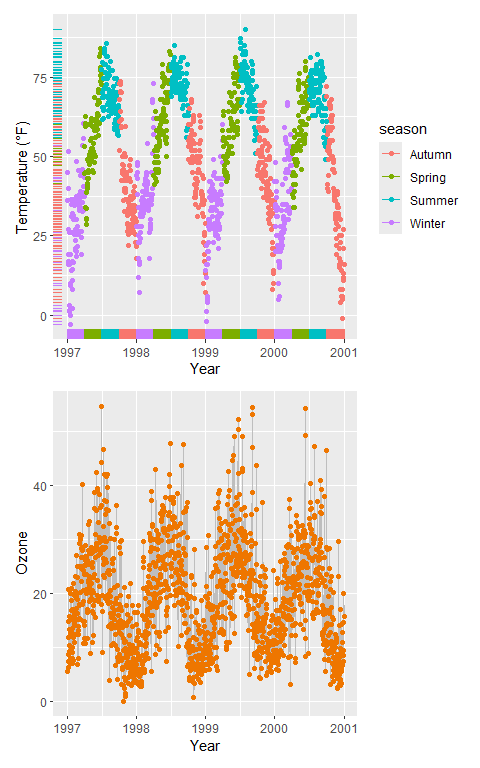
There are several ways how plots can be combined. The easiest approach in my opinion is the [{patchwork} package](https://github.com/thomasp85/patchwork) by Thomas Lin Pedersen:

p1 <- ggplot(chic, aes(x = date, y = temp,  
 color = season)) +  
 geom\_point() +  
 geom\_rug() +  
 labs(x = "Year", y = "Temperature (°F)")  
  
p2 <- ggplot(chic, aes(x = date, y = o3)) +  
 geom\_line(color = "gray") +  
 geom\_point(color = "darkorange2") +  
 labs(x = "Year", y = "Ozone")  
  
library(patchwork)  
p1 + p2



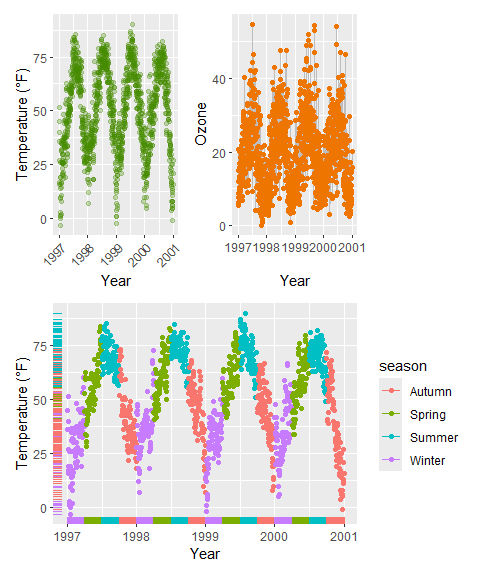
We can change the order by “dividing” both plots (and note the alignment even though one has a legend and one doesn’t!):

p1 / p2



And also nested plots are possible!

(g + p2) / p1



(Note the alignment of the plots even though only one plot includes a legend.)

Alternatively, the [{cowplot} package](https://wilkelab.org/cowplot/articles/introduction.html) by Claus Wilke provides the functionality to combine multiple plots (and lots of other good utilities):

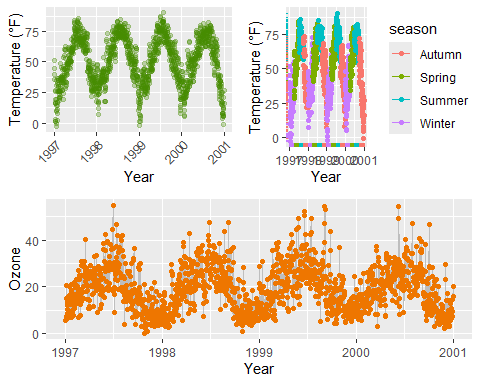
library(cowplot)

Attaching package: 'cowplot'

The following object is masked from 'package:patchwork':  
  
 align\_plots

The following object is masked from 'package:lubridate':  
  
 stamp

plot\_grid(plot\_grid(g, p1), p2, ncol = 1)



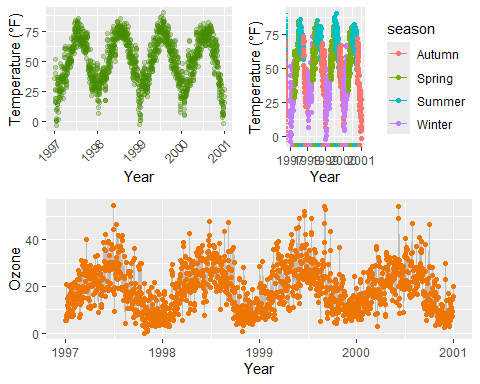
… and so does the [{gridExtra} package](https://cran.r-project.org/web/packages/gridExtra/vignettes/arrangeGrob.html) as well:

library(gridExtra)

Attaching package: 'gridExtra'

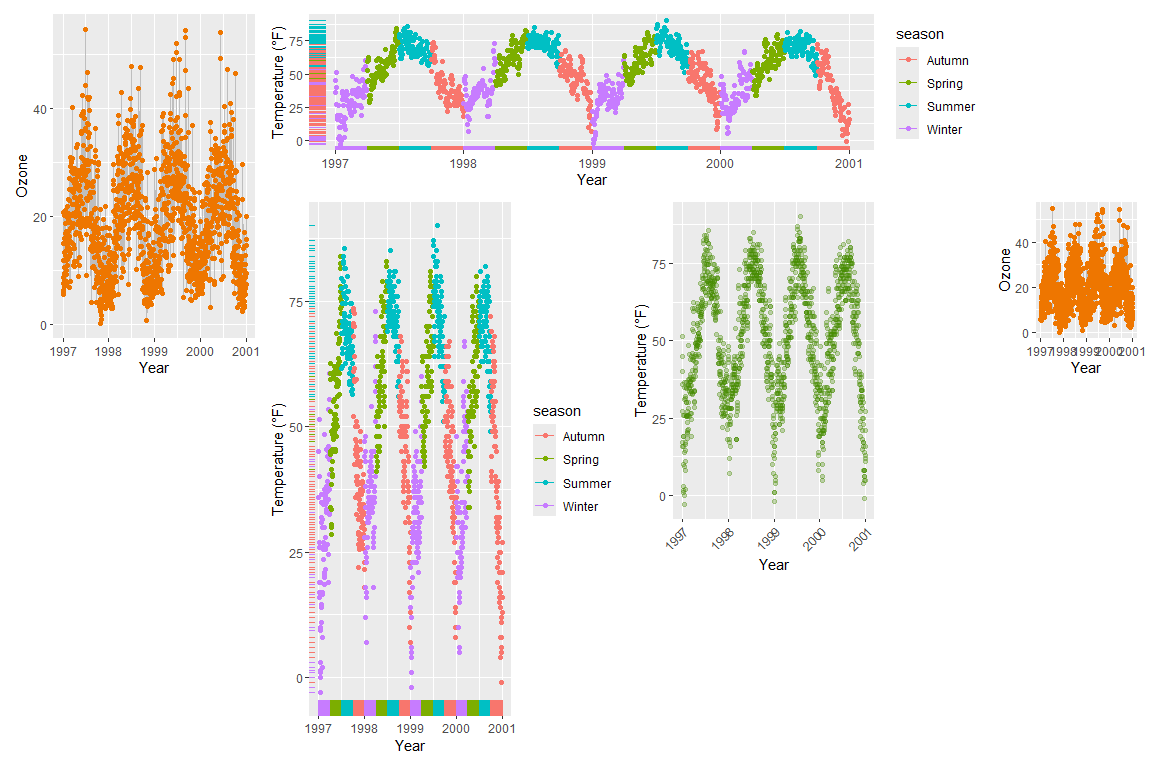
The following object is masked from 'package:dplyr':  
  
 combine

grid.arrange(g, p1, p2,  
 layout\_matrix = rbind(c(1, 2), c(3, 3)))



The same idea of defining a layout can be used with {patchwork} which allows creating complex compositions:

layout <- "  
AABBBB#  
AACCDDE  
##CCDD#  
##CC###  
"  
  
p2 + p1 + p1 + g + p2 +  
 plot\_layout(design = layout)



# 10. Working with Colors

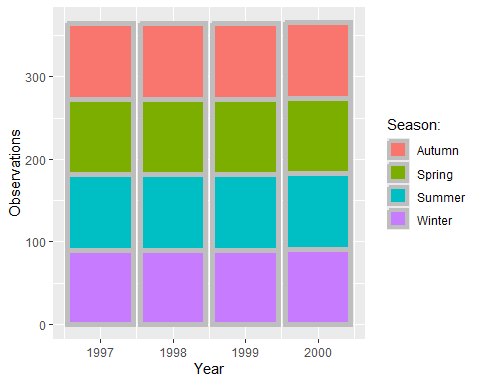
For simple applications working with colors is straightforward in {ggplot2}. For a more advanced treatment of the topic you should probably get your hands on [Hadley’s book](http://www.springer.com/de/book/9780387981413#otherversion=9780387981406) which has nice coverage. Other good sources are the [R Cookbook](http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/) and the [`color section in the R Graph Gallery](https://www.r-graph-gallery.com/ggplot2-color.html) by Yan Holtz.

There are two main differences when it comes to colors in {ggplot2}. Both arguments, color and fill, can be

1. specified as single color or
2. assigned to variables.

As you have already seen in the beginning of this tutorial, variables that are *inside* the aesthetics are encoded by variables and those that are *outside* are properties that are unrelated to the variables. This complete nonsense plot showing the number of records per year and season illustrates that fact:

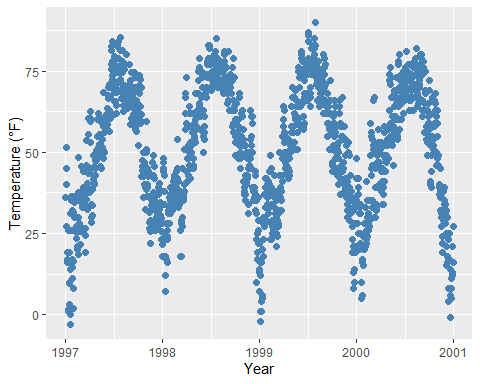
ggplot(chic, aes(year)) +  
 geom\_bar(aes(fill = season), color = "grey", linewidth = 2) +  
 labs(x = "Year", y = "Observations", fill = "Season:")



## 10.1 Specify Single Colors

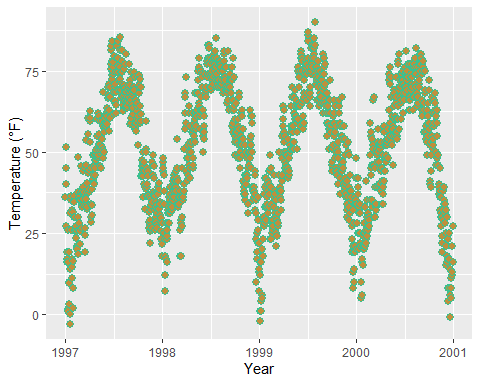
Static, single colors are simple to use. We can specify a single color for a geom:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "steelblue", size = 2) +  
 labs(x = "Year", y = "Temperature (°F)")



… and in case it provides both, a color (outline color) and a fill (filling color):

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(shape = 21, size = 2, stroke = 1,  
 color = "#3cc08f", fill = "#c08f3c") +  
 labs(x = "Year", y = "Temperature (°F)")



Tian Zheng at Columbia has created a useful [PDF of R colors](http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf). Of course, you can also specify hex color codes (simply as strings as in the example above) as well as RGB or RGBA values (via the rgb() function: rgb(red, green, blue, alpha)).

## 10.2 Assign Colors to Variables

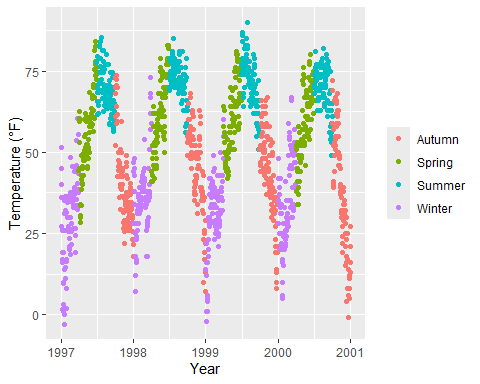
In {ggplot2}, colors that are assigned to variables are modified via the scale\_color\_\* and the scale\_fill\_\* functions. In order to use color with your data, most importantly you need to know if you are dealing with a categorical or continuous variable. The color palette should be chosen depending on type of the variable, with sequential or diverging color palettes being used for continuous variables and qualitative color palettes for categorical variables:

## 10.3 Qualitative Variables

Qualitative or categorical variables represent types of data which can be divided into groups (*categories*). The variable can be further specified as nominal, ordinal, and binary (dichotomous). Examples of qualitative/categorical variables are:

The default categorical color palette looks like this:

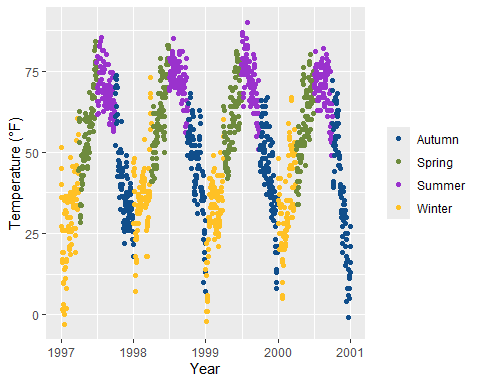
(ga <- ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)", color = NULL))



### 10.3.1 Manually Select Qualitative Colors

You can pick your own set of colors and assign them to a categorical variables via the function scale\_\*\_manual() (the \* can be either color, colour, or fill). The number of specified colors has to match the number of categories:

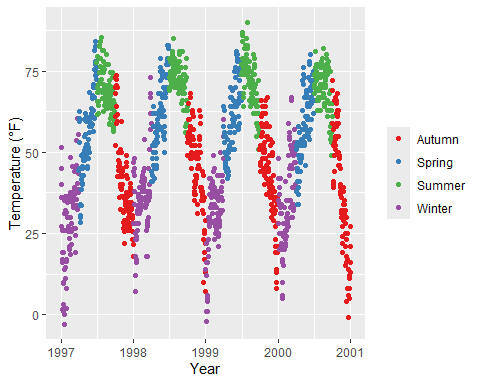
ga + scale\_color\_manual(values = c("dodgerblue4",  
 "darkolivegreen4",  
 "darkorchid3",  
 "goldenrod1"))



### 10.3.2 Use Built-In Qualitative Color Palettes

The [ColorBrewer palettes](http://colorbrewer2.org/) is a popular online tool for selecting color schemes for maps. The different sets of colors have been designed to produce attractive color schemes of similar appearance ranging from three to twelve. Those palettes are available as built-in functions in the {ggplot2} package and can be applied by calling scale\_\*\_brewer():

ga + scale\_color\_brewer(palette = "Set1")



|  |
| --- |
| Note |
| You can explore all schemes available via RColorBrewer::display.brewer.all(). |

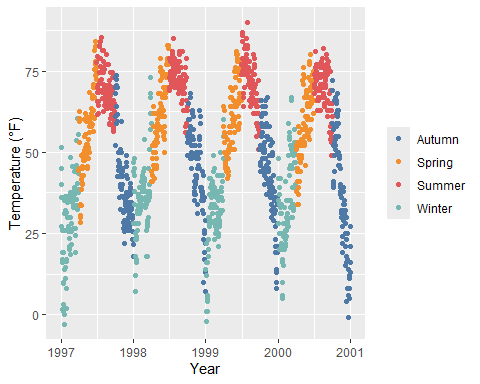
### 10.3.3 Use Qualitative Color Palettes from Extension Packages

There are many extension packages that provide additional color palettes. Their use differs depending on the way the package is designed. For an extensive overview of color palettes available in R, check the [collection provided by Emil Hvitfeldt](https://github.com/EmilHvitfeldt/r-color-palettes/blob/master/README.md#comprehensive-list-of-color-palettes-in-r). One can also use his [{paletteer} package](https://github.com/EmilHvitfeldt/paletteer), a comprehensive collection of color palettes in R that uses a consistent syntax.

**Examples:**

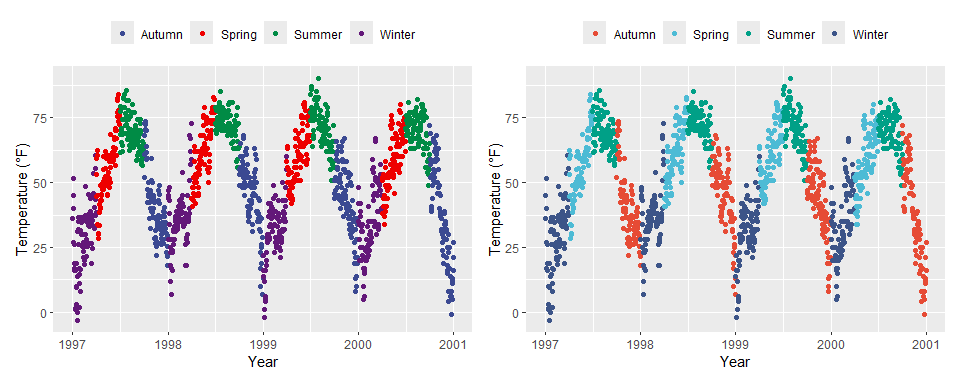
The [{ggthemes} package](https://jrnold.github.io/ggthemes/) for example lets R users access the Tableau colors. Tableau is a famous visualiztion software with a [well-known color palette](http://www.tableau.com/de-de/about/blog/2016/7/colors-upgrade-tableau-10-56782).

library(ggthemes)  
ga + scale\_color\_tableau()



The [{ggsci} package](https://nanx.me/ggsci/articles/ggsci.html) provides scientific journal and sci-fi themed color palettes. Want to have a plot with colors that look like being published in *Science* or *Nature*? Here you go!

library(ggsci)  
g1 <- ga + scale\_color\_aaas()  
g2 <- ga + scale\_color\_npg()  
  
library(patchwork)  
(g1 + g2) \* theme(legend.position = "top")



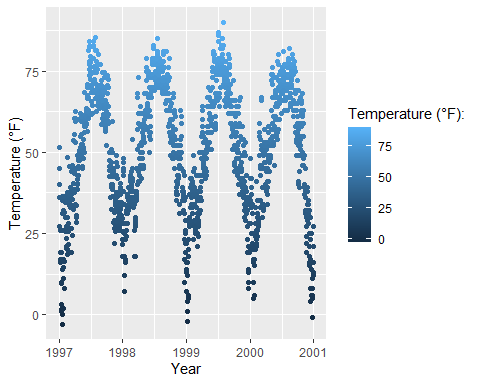
## 10.4 Quantitative Variables

Quantitative variables represent a measurable quantity and are thus numerical. Quantitative data can be further classified as being either continuous (floating numbers possible) or discrete (integers only):

In our example we will change the variable we want to color to ozone, a continuous variable that is strongly related to temperature (higher temperature = higher ozone). The function scale\_\*\_gradient() is a sequential gradient while scale\_\*\_gradient2() is diverging.

Here is the default {ggplot2} sequential color scheme for continuous variables:

gb <- ggplot(chic, aes(x = date, y = temp, color = temp)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F):")  
  
gb + scale\_color\_continuous()

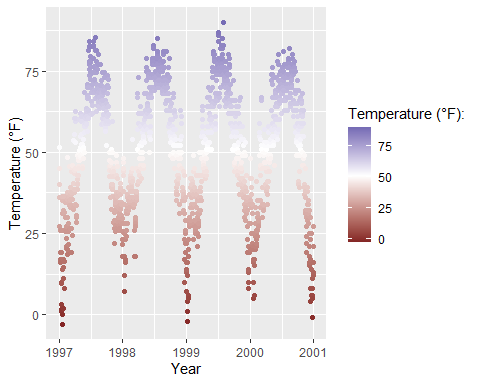


This code produces the same plot:

gb + scale\_color\_gradient()

And here is the diverging default color scheme:

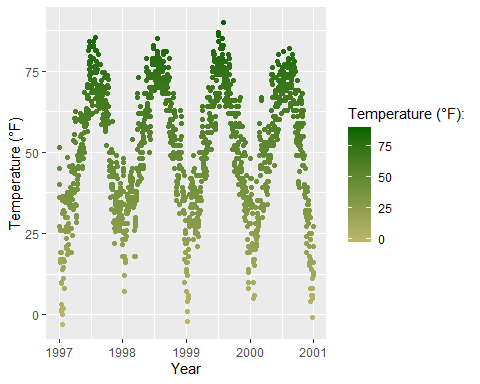
mid <- mean(chic$temp) ## midpoint  
  
gb + scale\_color\_gradient2(midpoint = mid)



### 10.4.1 Manually Set a Sequential Color Scheme

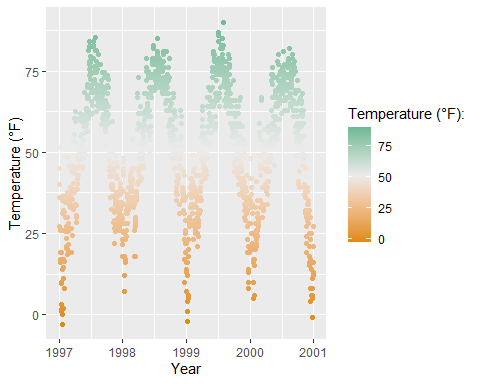
You can manually set gradually changing color palettes for continuous variables via scale\_\*\_gradient():

gb + scale\_color\_gradient(low = "darkkhaki",  
 high = "darkgreen")



Temperature data is normally distributed so how about a diverging color scheme (rather than sequential)… For diverging color you can use the scale\_\*\_gradient2() function:

gb + scale\_color\_gradient2(midpoint = mid, low = "#dd8a0b",  
 mid = "grey92", high = "#32a676")

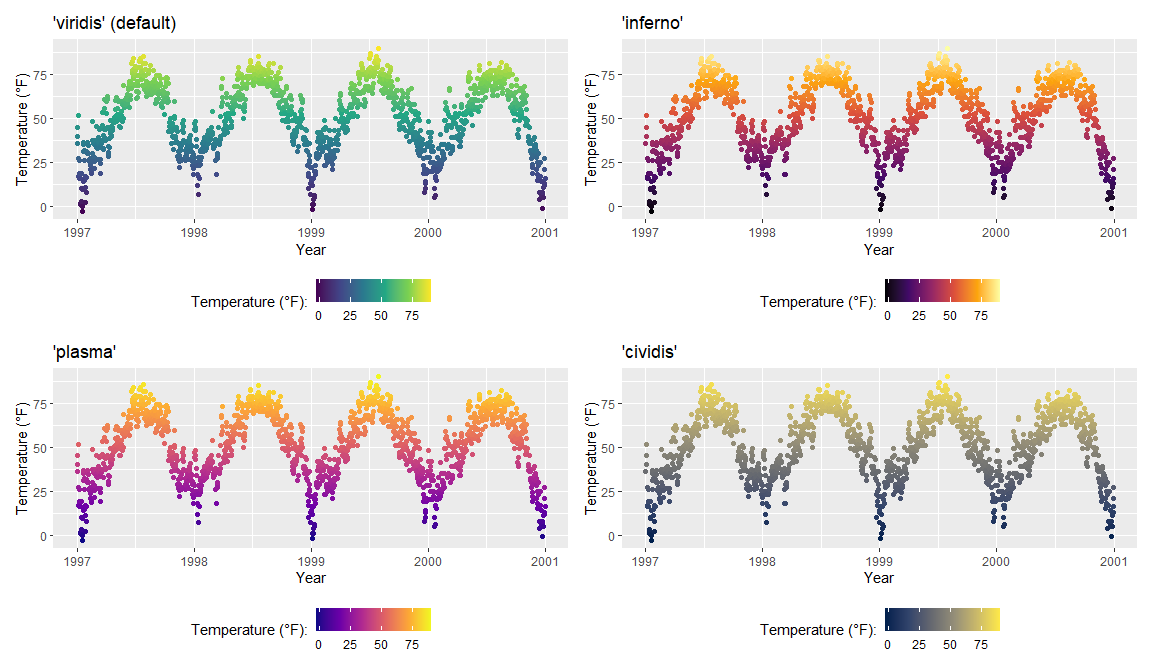


### 10.4.2 The Beautiful Viridis Color Palette

The [**viridis** color palettes](https://sjmgarnier.github.io/viridis/articles/intro-to-viridis.html) do not only make your plots look pretty and good to perceive but also easier to read by those with colorblindness and print well in gray scale. You can test how your plots might appear under various form of colorblindness using [{dichromat}](https://cran.r-project.org/web/packages/dichromat/index.html) package.

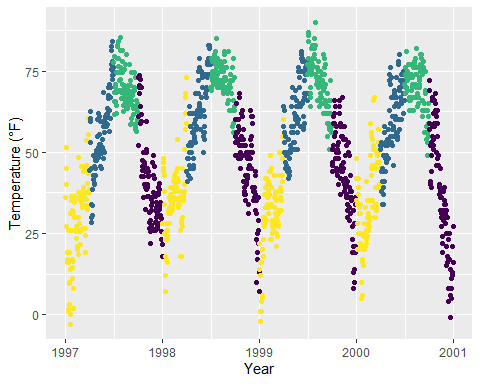
And they also come now shipped with {ggplot2}! The following multi-panel plot illustrates three out of the four viridis palettes:

p1 <- gb + scale\_color\_viridis\_c() + ggtitle("'viridis' (default)")  
p2 <- gb + scale\_color\_viridis\_c(option = "inferno") + ggtitle("'inferno'")  
p3 <- gb + scale\_color\_viridis\_c(option = "plasma") + ggtitle("'plasma'")  
p4 <- gb + scale\_color\_viridis\_c(option = "cividis") + ggtitle("'cividis'")  
  
library(patchwork)  
(p1 + p2 + p3 + p4) \* theme(legend.position = "bottom")



It is also possible to use the viridis color palettes for discrete variables:

ga + scale\_color\_viridis\_d(guide = "none")



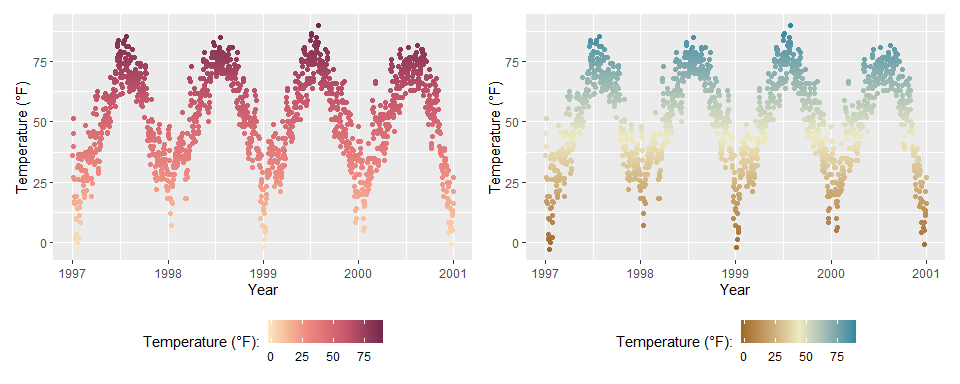
### 10.4.3 Use Quantitative Color Palettes from Extension Packages

The many extension packages provide not only additional categorical color palettes but also sequential, diverging and even cyclical palettes. Again, I point you to the great [collection provided by Emil Hvitfeldt](https://github.com/EmilHvitfeldt/r-color-palettes/blob/master/README.md#comprehensive-list-of-color-palettes-in-r) for an overview.

**Examples:**

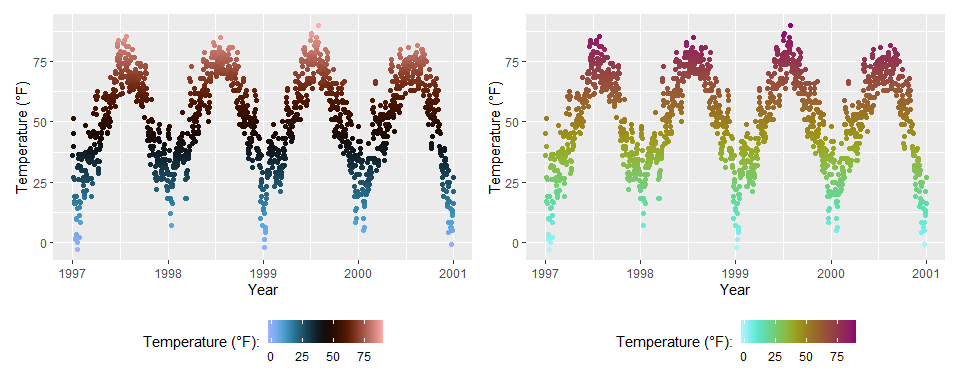
The [{rcartocolors} packages](https://github.com/Nowosad/rcartocolor) ports the beautiful [CARTOcolors](https://www.google.com/search?client=firefox-b-d&q=carto+oclors) to {ggplot2} and contains several of my most-used palettes:

library(rcartocolor)  
g1 <- gb + scale\_color\_carto\_c(palette = "BurgYl")  
g2 <- gb + scale\_color\_carto\_c(palette = "Earth")  
  
(g1 + g2) \* theme(legend.position = "bottom")



The [{scico} package](https://github.com/thomasp85/scico) provides access to the [color palettes developed by Fabio Crameri](http://www.fabiocrameri.ch/colourmaps.php). These color palettes are not only beautiful and often unusual but also a good choice since they have been developed to be perceptually uniform and ordered. In addition, they work for people with color vision deficiency and in grayscale:

library(scico)  
g1 <- gb + scale\_color\_scico(palette = "berlin")  
g2 <- gb + scale\_color\_scico(palette = "hawaii", direction = -1)  
  
(g1 + g2) \* theme(legend.position = "bottom")



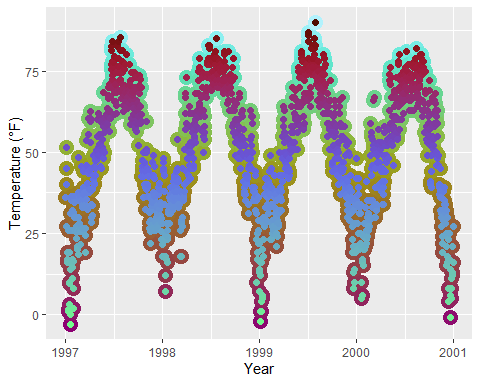
#### 10.4.3.1 Modify Color Palettes Afterwards

Since the release of ggplot2 3.0.0, one can modify layer aesthetics after they have been mapped to the data. Or as the {ggplot2} phrases it: “Use after\_scale() to flag evaluation of mapping for after data has been scaled.”

So why not use the modified colors in the first place? Since {ggplot2} can only handle one color and one fill scale, this is an interesting functionality. Look closer at the following example where we use clr\_negate() from the [{prismatic} package](https://emilhvitfeldt.github.io/prismatic/):

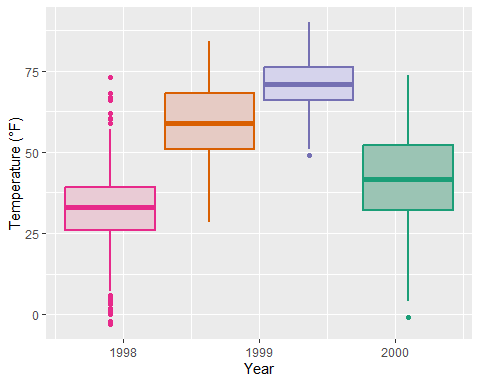
library(prismatic)  
  
ggplot(chic, aes(date, temp, color = temp)) +  
 geom\_point(size = 5) +  
 geom\_point(aes(color = temp,  
 color = after\_scale(clr\_negate(color))),  
 size = 2) +  
 scale\_color\_scico(palette = "hawaii", guide = "none") +  
 labs(x = "Year", y = "Temperature (°F)")

Warning: Duplicated aesthetics after name standardisation: colour



Changing the color scheme afterwards is especially fun with functions from the {prismatic} packages, namely clr\_negate(), clr\_lighten(), clr\_darken() and clr\_desaturate(). You can even combine those functions. Here, we plot a box plot that has both arguments, color and fill:

library(prismatic)  
  
ggplot(chic, aes(date, temp)) +  
 geom\_boxplot(  
 aes(color = season,  
 fill = after\_scale(clr\_desaturate(clr\_lighten(color, .6), .6))),  
 linewidth = 1  
 ) +  
 scale\_color\_brewer(palette = "Dark2", guide = "none") +  
 labs(x = "Year", y = "Temperature (°F)")



Note that you need to specify the color and/or fill in the aes() of the respective geom\_\*() or stat\_\*() to make after\_scale() work.

|  |
| --- |
| Important |
| This seems a bit complicated for now—one could simply use the color and fill scales for both. Yes, that is true but think about use cases where you need several color and/or fill scales. In such a case, it would be senseless to occupy the fill scale with a slightly darker version of the palette used for color. |

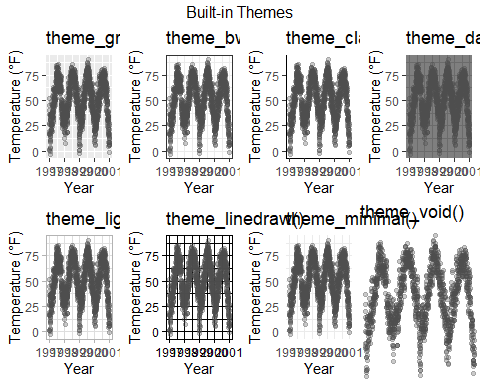
# 11. Working with Themes

## 11.1 Change the Overall Plotting Style

You can change the entire look of the plots by using themes. {ggplot2} comes with eight built-in themes:

Attaching package: 'gridExtra'

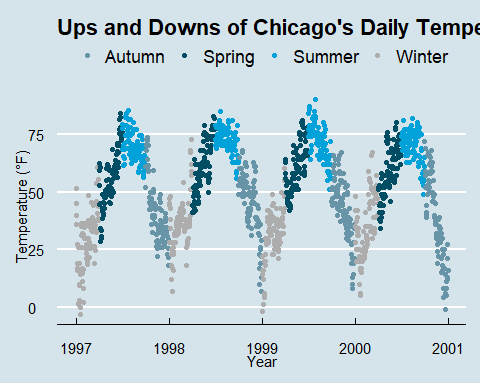
The following object is masked from 'package:dplyr':  
  
 combine



There are several packages that provide additional themes, some even with different default color palettes. As an example, Jeffrey Arnold has put together the library {ggthemes} with several custom themes imitating popular designs. For a list you can visit the [{ggthemes} package site](https://github.com/jrnold/ggthemes). Without any coding you can just adapt several styles, some of them well known for their style and aesthetics.

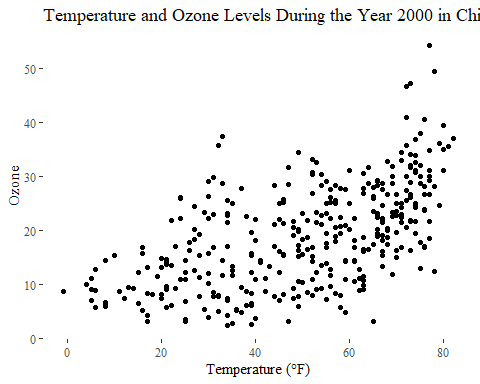
Here is an example copying the [plotting style](https://www.google.de/search?q=economist+graphic&tbm=isch) in the [The Economist](http://www.economist.com/) magazine by using theme\_economist() and scale\_color\_economist():

library(ggthemes)  
  
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 ggtitle("Ups and Downs of Chicago's Daily Temperatures") +  
 theme\_economist() +  
 scale\_color\_economist(name = NULL)



Another example is the plotting style of Tufte, a minimal ink theme based on [Edward Tufte](http://ww%20w.aiga.org/medalist-edwardtufte)’s book [The Visual Display of Quantitative Information](https://www.edwardtufte.com/tufte/books_vdqi). This is the book that popularized [Minard’s chart depicting Napoleon’s march on Russia](https://www.edwardtufte.com/tufte/minard) as one of the **best statistical drawings ever created**. Tufte’s plots became famous due to the purism in their style. But see yourself:

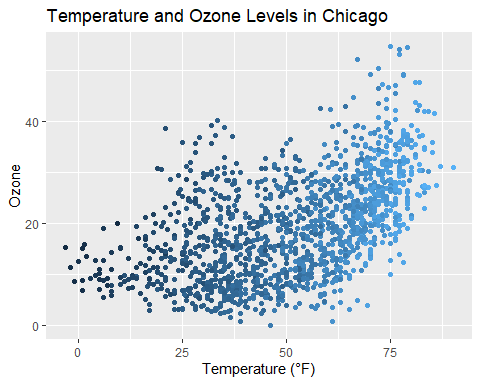
library(dplyr)  
chic\_2000 <- filter(chic, year == 2000)  
  
ggplot(chic\_2000, aes(x = temp, y = o3)) +  
 geom\_point() +  
 labs(x = "Temperature (°F)", y = "Ozone") +  
 ggtitle("Temperature and Ozone Levels During the Year 2000 in Chicago") +  
 theme\_tufte()



I reduced the number of data points here simply to fit it Tufte’s minimalism style. If you like the way of plotting have a look on [this blog entry](http://motioninsocial.com/tufte/) creating several Tufte plots in R.

Another neat packages with modern themes and a preset of non-default fonts is the [{hrbrthemes} package by Bob Rudis](https://github.com/hrbrmstr/hrbrthemes) with several light but also dark themes:

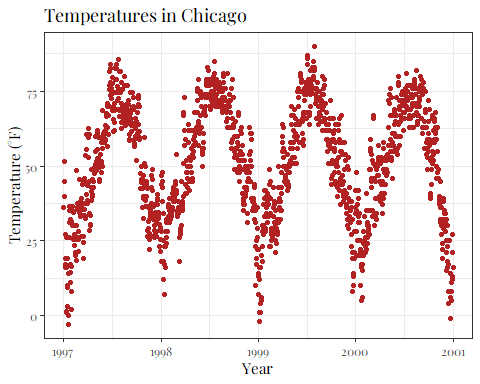
library(hrbrthemes)  
  
ggplot(chic, aes(x = temp, y = o3)) +  
 geom\_point(aes(color = dewpoint), show.legend = FALSE) +  
 labs(x = "Temperature (°F)", y = "Ozone") +  
 ggtitle("Temperature and Ozone Levels in Chicago")



## 11.2 Change the Font of All Text Elements

It is incredibly easy to change the settings of all the text elements at once. All themes come with an argument called base\_family:

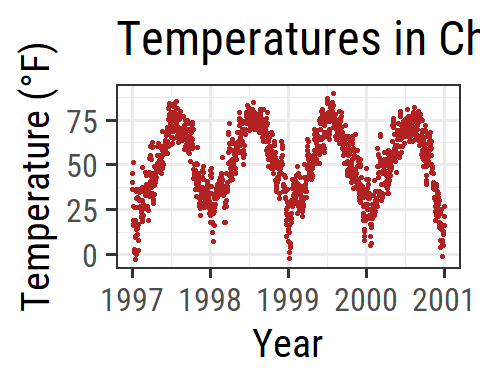
g <- ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "firebrick") +  
 labs(x = "Year", y = "Temperature (°F)",  
 title = "Temperatures in Chicago")  
  
g + theme\_bw(base\_family = "Playfair Display")



## 11.3 Change the Size of All Text Elements

The theme\_\*() functions also come with several other base\_\* arguments. If you have a closer look at the default theme (see chapter “Create and Use Your Custom Theme” below) you will notice that the sizes of all the elements are relative (rel()) to the base\_size. As a result, you can simply change the base\_size if you want to increase readability of your plots:

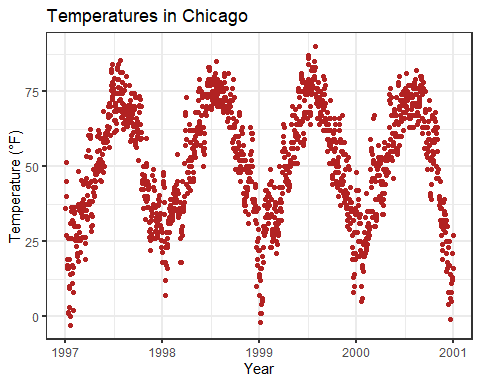
g + theme\_bw(base\_size = 30, base\_family = "Roboto Condensed")



## 11.4 Change the Size of All Line and Rect Elements

Similarly, you can change the size of all elements of type line and rect:

g + theme\_bw(base\_line\_size = 1, base\_rect\_size = 1)



## 11.5 Create Your Own Theme

If you want to change the theme for an entire session you can use theme\_set as in theme\_set(theme\_bw()). The default is called theme\_gray (or theme\_gray). If you wanted to create your own custom theme, you could extract the code directly from the gray theme and modify. Note that the rel() function change the sizes relative to the base\_size.

theme\_gray

function (base\_size = 11, base\_family = "", base\_line\_size = base\_size/22,   
 base\_rect\_size = base\_size/22)   
{  
 half\_line <- base\_size/2  
 t <- theme(line = element\_line(colour = "black", linewidth = base\_line\_size,   
 linetype = 1, lineend = "butt"), rect = element\_rect(fill = "white",   
 colour = "black", linewidth = base\_rect\_size, linetype = 1),   
 text = element\_text(family = base\_family, face = "plain",   
 colour = "black", size = base\_size, lineheight = 0.9,   
 hjust = 0.5, vjust = 0.5, angle = 0, margin = margin(),   
 debug = FALSE), axis.line = element\_blank(), axis.line.x = NULL,   
 axis.line.y = NULL, axis.text = element\_text(size = rel(0.8),   
 colour = "grey30"), axis.text.x = element\_text(margin = margin(t = 0.8 \*   
 half\_line/2), vjust = 1), axis.text.x.top = element\_text(margin = margin(b = 0.8 \*   
 half\_line/2), vjust = 0), axis.text.y = element\_text(margin = margin(r = 0.8 \*   
 half\_line/2), hjust = 1), axis.text.y.right = element\_text(margin = margin(l = 0.8 \*   
 half\_line/2), hjust = 0), axis.text.r = element\_text(margin = margin(l = 0.8 \*   
 half\_line/2, r = 0.8 \* half\_line/2), hjust = 0.5),   
 axis.ticks = element\_line(colour = "grey20"), axis.ticks.length = unit(half\_line/2,   
 "pt"), axis.ticks.length.x = NULL, axis.ticks.length.x.top = NULL,   
 axis.ticks.length.x.bottom = NULL, axis.ticks.length.y = NULL,   
 axis.ticks.length.y.left = NULL, axis.ticks.length.y.right = NULL,   
 axis.minor.ticks.length = rel(0.75), axis.title.x = element\_text(margin = margin(t = half\_line/2),   
 vjust = 1), axis.title.x.top = element\_text(margin = margin(b = half\_line/2),   
 vjust = 0), axis.title.y = element\_text(angle = 90,   
 margin = margin(r = half\_line/2), vjust = 1), axis.title.y.right = element\_text(angle = -90,   
 margin = margin(l = half\_line/2), vjust = 1), legend.background = element\_rect(colour = NA),   
 legend.spacing = unit(2 \* half\_line, "pt"), legend.spacing.x = NULL,   
 legend.spacing.y = NULL, legend.margin = margin(half\_line,   
 half\_line, half\_line, half\_line), legend.key = NULL,   
 legend.key.size = unit(1.2, "lines"), legend.key.height = NULL,   
 legend.key.width = NULL, legend.key.spacing = unit(half\_line,   
 "pt"), legend.text = element\_text(size = rel(0.8)),   
 legend.title = element\_text(hjust = 0), legend.ticks.length = rel(0.2),   
 legend.position = "right", legend.direction = NULL, legend.justification = "center",   
 legend.box = NULL, legend.box.margin = margin(0, 0, 0,   
 0, "cm"), legend.box.background = element\_blank(),   
 legend.box.spacing = unit(2 \* half\_line, "pt"), panel.background = element\_rect(fill = "grey92",   
 colour = NA), panel.border = element\_blank(), panel.grid = element\_line(colour = "white"),   
 panel.grid.minor = element\_line(linewidth = rel(0.5)),   
 panel.spacing = unit(half\_line, "pt"), panel.spacing.x = NULL,   
 panel.spacing.y = NULL, panel.ontop = FALSE, strip.background = element\_rect(fill = "grey85",   
 colour = NA), strip.clip = "inherit", strip.text = element\_text(colour = "grey10",   
 size = rel(0.8), margin = margin(0.8 \* half\_line,   
 0.8 \* half\_line, 0.8 \* half\_line, 0.8 \* half\_line)),   
 strip.text.x = NULL, strip.text.y = element\_text(angle = -90),   
 strip.text.y.left = element\_text(angle = 90), strip.placement = "inside",   
 strip.placement.x = NULL, strip.placement.y = NULL, strip.switch.pad.grid = unit(half\_line/2,   
 "pt"), strip.switch.pad.wrap = unit(half\_line/2,   
 "pt"), plot.background = element\_rect(colour = "white"),   
 plot.title = element\_text(size = rel(1.2), hjust = 0,   
 vjust = 1, margin = margin(b = half\_line)), plot.title.position = "panel",   
 plot.subtitle = element\_text(hjust = 0, vjust = 1, margin = margin(b = half\_line)),   
 plot.caption = element\_text(size = rel(0.8), hjust = 1,   
 vjust = 1, margin = margin(t = half\_line)), plot.caption.position = "panel",   
 plot.tag = element\_text(size = rel(1.2), hjust = 0.5,   
 vjust = 0.5), plot.tag.position = "topleft", plot.margin = margin(half\_line,   
 half\_line, half\_line, half\_line), complete = TRUE)  
 ggplot\_global$theme\_all\_null %+replace% t  
}  
<bytecode: 0x0000027496cbfda8>  
<environment: namespace:ggplot2>

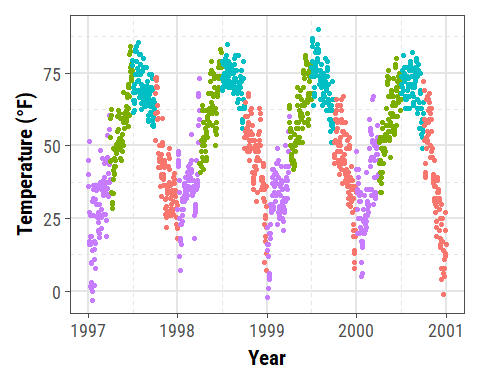
Now, let us modify the default theme function and have a look at the result:

theme\_2hin <- function (base\_size = 12, base\_family = "Roboto Condensed") {  
 half\_line <- base\_size/2  
 theme(  
 line = element\_line(color = "black", linewidth = .5,  
 linetype = 1, lineend = "butt"),  
 rect = element\_rect(fill = "white", color = "black",  
 linewidth = .5, linetype = 1),  
 text = element\_text(family = base\_family, face = "plain",  
 color = "black", size = base\_size,  
 lineheight = .9, hjust = .5, vjust = .5,  
 angle = 0, margin = margin(), debug = FALSE),  
 axis.line = element\_blank(),  
 axis.line.x = NULL,  
 axis.line.y = NULL,  
 axis.text = element\_text(size = base\_size \* 1.1, color = "gray30"),  
 axis.text.x = element\_text(margin = margin(t = .8 \* half\_line/2),  
 vjust = 1),  
 axis.text.x.top = element\_text(margin = margin(b = .8 \* half\_line/2),  
 vjust = 0),  
 axis.text.y = element\_text(margin = margin(r = .8 \* half\_line/2),  
 hjust = 1),  
 axis.text.y.right = element\_text(margin = margin(l = .8 \* half\_line/2),  
 hjust = 0),  
 axis.ticks = element\_line(color = "gray30", linewidth = .7),  
 axis.ticks.length = unit(half\_line / 1.5, "pt"),  
 axis.ticks.length.x = NULL,  
 axis.ticks.length.x.top = NULL,  
 axis.ticks.length.x.bottom = NULL,  
 axis.ticks.length.y = NULL,  
 axis.ticks.length.y.left = NULL,  
 axis.ticks.length.y.right = NULL,  
 axis.title.x = element\_text(margin = margin(t = half\_line),  
 vjust = 1, size = base\_size \* 1.3,  
 face = "bold"),  
 axis.title.x.top = element\_text(margin = margin(b = half\_line),  
 vjust = 0),  
 axis.title.y = element\_text(angle = 90, vjust = 1,  
 margin = margin(r = half\_line),  
 size = base\_size \* 1.3, face = "bold"),  
 axis.title.y.right = element\_text(angle = -90, vjust = 0,  
 margin = margin(l = half\_line)),  
 legend.background = element\_rect(color = NA),  
 legend.spacing = unit(.4, "cm"),  
 legend.spacing.x = NULL,  
 legend.spacing.y = NULL,  
 legend.margin = margin(.2, .2, .2, .2, "cm"),  
 legend.key = element\_rect(fill = "gray95", color = "white"),  
 legend.key.size = unit(1.2, "lines"),  
 legend.key.height = NULL,  
 legend.key.width = NULL,  
 legend.text = element\_text(size = rel(.8)),  
 legend.text.align = NULL,  
 legend.title = element\_text(hjust = 0),  
 legend.title.align = NULL,  
 legend.position = "right",  
 legend.direction = NULL,  
 legend.justification = "center",  
 legend.box = NULL,  
 legend.box.margin = margin(0, 0, 0, 0, "cm"),  
 legend.box.background = element\_blank(),  
 legend.box.spacing = unit(.4, "cm"),  
 panel.background = element\_rect(fill = "white", color = NA),  
 panel.border = element\_rect(color = "gray30",  
 fill = NA, linewidth = .7),  
 panel.grid.major = element\_line(color = "gray90", linewidth = 1),  
 panel.grid.minor = element\_line(color = "gray90", linewidth = .5,  
 linetype = "dashed"),  
 panel.spacing = unit(base\_size, "pt"),  
 panel.spacing.x = NULL,  
 panel.spacing.y = NULL,  
 panel.ontop = FALSE,  
 strip.background = element\_rect(fill = "white", color = "gray30"),  
 strip.text = element\_text(color = "black", size = base\_size),  
 strip.text.x = element\_text(margin = margin(t = half\_line,  
 b = half\_line)),  
 strip.text.y = element\_text(angle = -90,  
 margin = margin(l = half\_line,  
 r = half\_line)),  
 strip.text.y.left = element\_text(angle = 90),  
 strip.placement = "inside",  
 strip.placement.x = NULL,  
 strip.placement.y = NULL,  
 strip.switch.pad.grid = unit(0.1, "cm"),  
 strip.switch.pad.wrap = unit(0.1, "cm"),  
 plot.background = element\_rect(color = NA),  
 plot.title = element\_text(size = base\_size \* 1.8, hjust = .5,  
 vjust = 1, face = "bold",  
 margin = margin(b = half\_line \* 1.2)),  
 plot.title.position = "panel",  
 plot.subtitle = element\_text(size = base\_size \* 1.3,  
 hjust = .5, vjust = 1,  
 margin = margin(b = half\_line \* .9)),  
 plot.caption = element\_text(size = rel(0.9), hjust = 1, vjust = 1,  
 margin = margin(t = half\_line \* .9)),  
 plot.caption.position = "panel",  
 plot.tag = element\_text(size = rel(1.2), hjust = .5, vjust = .5),  
 plot.tag.position = "topleft",  
 plot.margin = margin(rep(base\_size, 4)),  
 complete = TRUE  
 )  
}

|  |
| --- |
| Note |
| You can only overwrite the defaults for all elements you want to change. Here I listed all. so you can see that you can change *literally* everything! |

Have a look on the modified aesthetics with its new look of panel and gridlines as well as axes ticks, texts and titles:

theme\_set(theme\_2hin())  
  
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() + labs(x = "Year", y = "Temperature (°F)") + guides(color = "none")

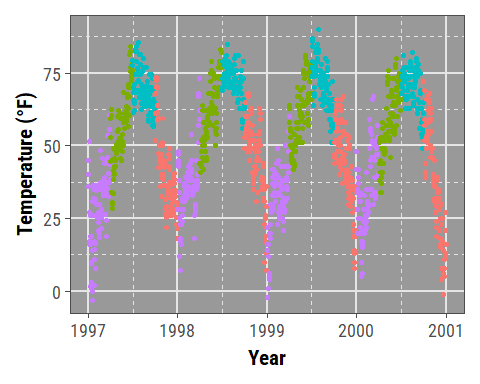


**This way of changing the plot design is highly recommended!** It allows you to quickly change any element of your plots by changing it once. You can within a few seconds plot all your results in a congruent style and adapt it to other needs (e.g. a presentation with bigger font size or journal requirements).

## 11.6 Update the Current Theme

You can also set quick changes using theme\_update():

theme\_2hin <- theme\_update(panel.background = element\_rect(fill = "gray60"))  
  
ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point() + labs(x = "Year", y = "Temperature (°F)") + guides(color = "none")



For further exercises, we are going to use our own theme with a white filling and without the minor grid lines:

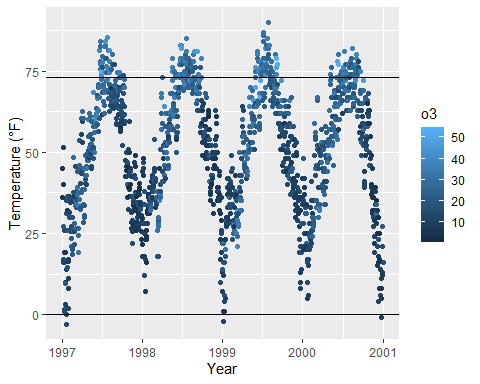
theme\_2hin <- theme\_update(  
 panel.background = element\_rect(fill = "white"),  
 panel.grid.major = element\_line(linewidth = .5),  
 panel.grid.minor = element\_blank()  
)

# 12. Working with Lines

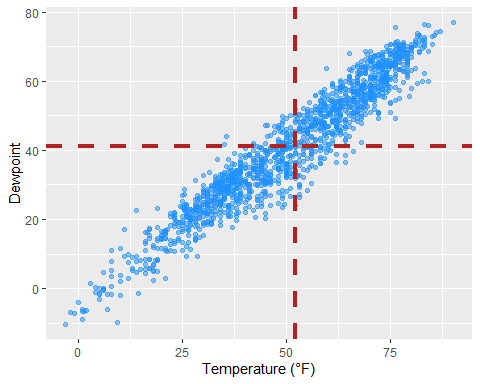
## 12.1 Add Horizonal or Vertical Lines to a Plot

You might want to highlight a given range or threshold, which can be done plotting a line at defined coordinates using geom\_hline() (for “horizontal lines”) or geom\_vline() (for “vertical lines”):

ggplot(chic, aes(x = date, y = temp, color = o3)) +  
 geom\_point() +  
 geom\_hline(yintercept = c(0, 73)) +  
 labs(x = "Year", y = "Temperature (°F)")

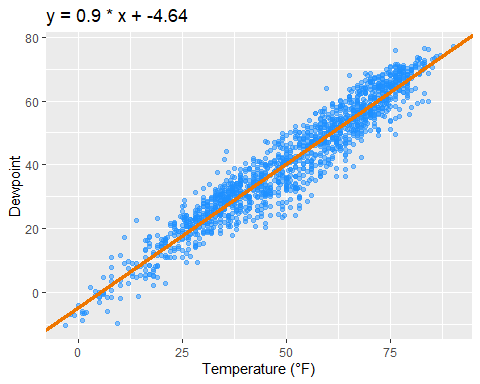


g <- ggplot(chic, aes(x = temp, y = dewpoint)) +  
 geom\_point(color = "dodgerblue", alpha = .5) +  
 labs(x = "Temperature (°F)", y = "Dewpoint")  
  
g +  
 geom\_vline(aes(xintercept = median(temp)), linewidth = 1.5,  
 color = "firebrick", linetype = "dashed") +  
 geom\_hline(aes(yintercept = median(dewpoint)), linewidth = 1.5,  
 color = "firebrick", linetype = "dashed")



If you want to add a line with a slope not being 0 or 1, respectively, you need to use geom\_abline(). This is for example the case if you want to add a regression line using the arguments intercept and slope:

reg <- lm(dewpoint ~ temp, data = chic)  
  
g +  
 geom\_abline(intercept = coefficients(reg)[1],  
 slope = coefficients(reg)[2],  
 color = "darkorange2",   
 linewidth = 1.5) +  
 labs(title = paste0("y = ", round(coefficients(reg)[2], 2),  
 " \* x + ", round(coefficients(reg)[1], 2)))

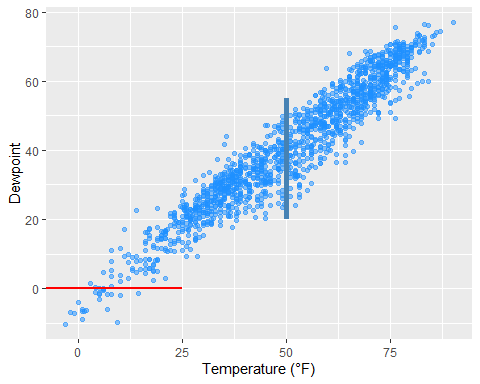


Later, we will learn how to add a linear fit with one command using stat\_smooth(method = "lm"). However, there might be other reasons to add a line with a given slope and this is how one does it 🤷

## 12.2 Add a Line within a Plot

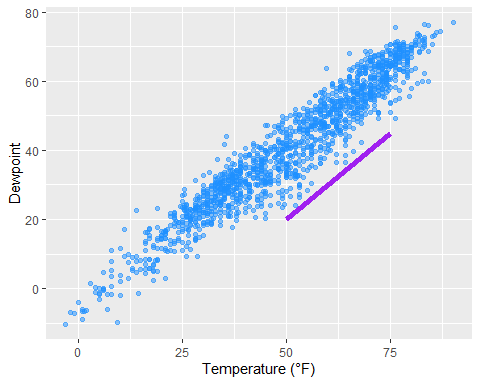
The previous approaches always covered the whole range of the plot panel, but sometimes one wants to highlight only a given area or use lines for annotations. In this case, geom\_linerange() is here to help:

g +  
 ## vertical line  
 geom\_linerange(aes(x = 50, ymin = 20, ymax = 55),  
 color = "steelblue", linewidth = 2) +  
 ## horizontal line  
 geom\_linerange(aes(xmin = -Inf, xmax = 25, y = 0),  
 color = "red", linewidth = 1)



Or you can use annotate(geom = "segment") to draw lines with a slope differing from 0 and 1:

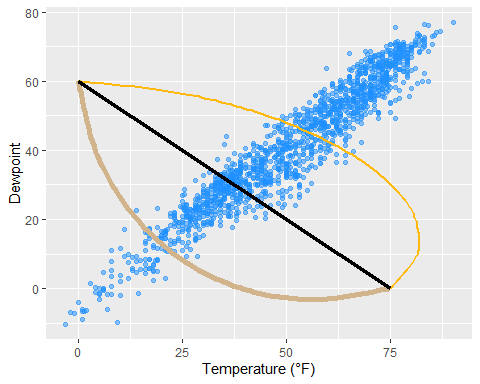
g +  
 annotate(geom = "segment",   
 x = 50, xend = 75,  
 y = 20, yend = 45,  
 color = "purple", linewidth = 2)



## 12.3 Add Curved Lines and Arrows to a Plot

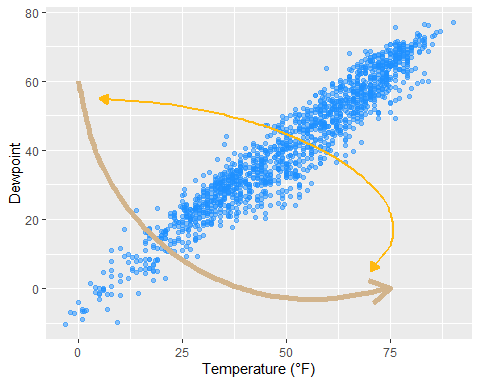
annotate(geom = "curve") adds curves. Well, and straight lines if you like:

g +  
 annotate(geom = "curve",x = 0, y = 60, xend = 75, yend = 0,  
 color = "tan", linewidth = 2) +  
 annotate(geom = "curve",   
 x = 0, y = 60, xend = 75, yend = 0,  
 curvature = -0.7, angle = 45,  
 color = "darkgoldenrod1", linewidth = 1) +  
 annotate(geom = "curve", x = 0, y = 60, xend = 75, yend = 0,  
 curvature = 0, linewidth = 1.5)



The same geom can be used to draw arrows:

g +  
 annotate(geom = "curve", x = 0, y = 60, xend = 75, yend = 0,  
 color = "tan", linewidth = 2,   
 arrow = arrow(length = unit(0.07, "npc"))) +  
 annotate(geom = "curve", x = 5, y = 55, xend = 70, yend = 5,  
 curvature = -0.7, angle = 45,  
 color = "darkgoldenrod1", linewidth = 1,  
 arrow = arrow(length = unit(0.03, "npc"),  
 type = "closed",  
 ends = "both"))

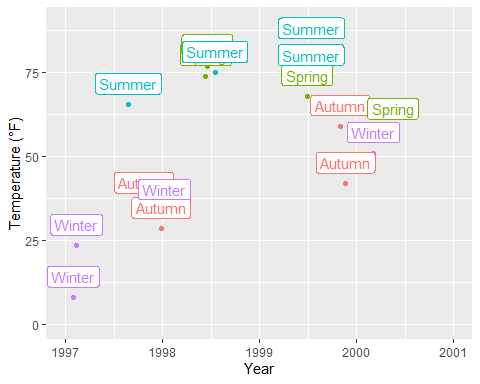


# 13. Working with Text

## 13.1 Add Labels to Your Data

Sometimes, we want to label our data points. To avoid overlaying and crowding by text labels, we use a 1% sample of the original data, equally representing the four seasons. We are using geom\_label() which comes with a new aesthetic called label:

set.seed(2020)  
  
sample <- chic |>  
 dplyr::group\_by(season) |>  
 dplyr::sample\_frac(0.01)  
  
## code without pipes:  
## sample <- sample\_frac(group\_by(chic, season), .01)  
  
ggplot(sample, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 geom\_label(aes(label = season), hjust = .5, vjust = -.5) +  
 labs(x = "Year", y = "Temperature (°F)") +  
 xlim(as.Date(c('1997-01-01', '2000-12-31'))) +  
 ylim(c(0, 90)) +  
 theme(legend.position = "none")

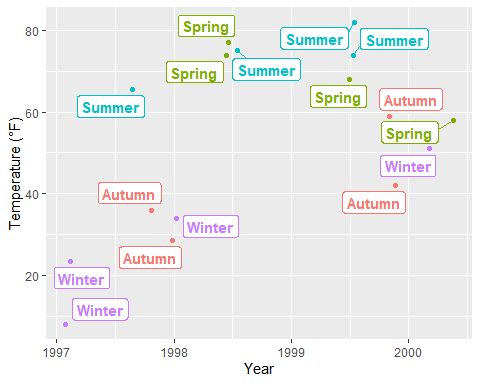


Okay, avoiding overlap of labels did not work out. But don’t worry, we are going to fix it in a minute!

|  |
| --- |
| Using geom\_text() |
| You can also use geom\_text() if you don’t like boxes around your labels. Expand to see example.  ggplot(sample, aes(x = date, y = temp, color = season)) +  geom\_point() +  geom\_text(aes(label = season), fontface = "bold",  hjust = .5, vjust = -.25) +  labs(x = "Year", y = "Temperature (°F)") +  xlim(as.Date(c('1997-01-01', '2000-12-31'))) +  ylim(c(0, 90)) +  theme(legend.position = "none") |

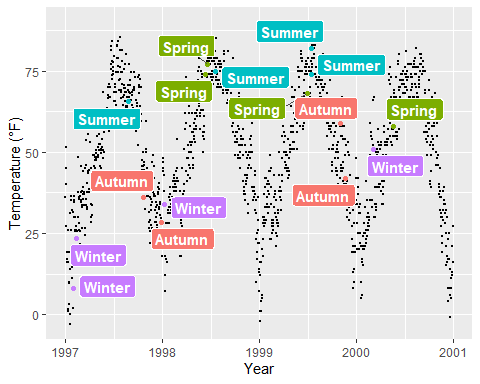
The {ggrepel} package offers some great utilities by providing geoms for {ggplot2} to repel overlapping text as in our examples above. We simply replace geom\_text() by geom\_text\_repel() and geom\_label() by geom\_label\_repel():

library(ggrepel)  
  
ggplot(sample, aes(x = date, y = temp, color = season)) +  
 geom\_point() +  
 geom\_label\_repel(aes(label = season), fontface = "bold") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.position = "none")



It may look nicer with filled boxes so we map season to fill instead to color and set a white color for the text:

ggplot(sample, aes(x = date, y = temp)) +  
 geom\_point(data = chic, size = .5) +  
 geom\_point(aes(color = season), size = 1.5) +  
 geom\_label\_repel(aes(label = season, fill = season),  
 color = "white", fontface = "bold",  
 segment.color = "grey30") +  
 labs(x = "Year", y = "Temperature (°F)") +  
 theme(legend.position = "none")

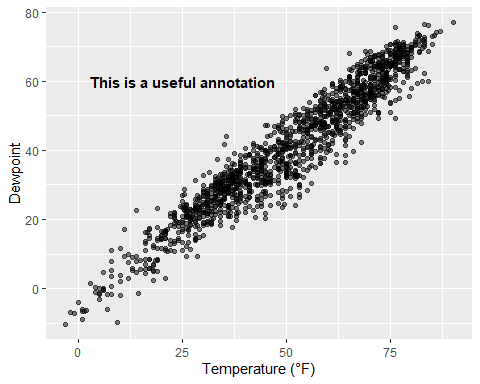


This also works for the pure text labels by using geom\_text\_repel(). Have a look at all the [usage examples](https://ggrepel.slowkow.com/articles/examples.html).

## 13.2 Add Text Annotations

There are several ways how one can add annotations to a ggplot. We can again use annotate(geom = "text"), annotate(geom = "label"), geom\_text() or geom\_label():

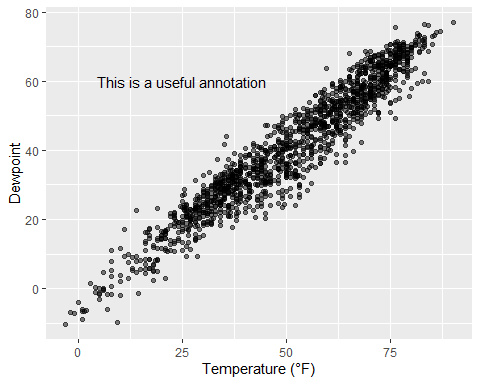
g <-  
 ggplot(chic, aes(x = temp, y = dewpoint)) +  
 geom\_point(alpha = .5) +  
 labs(x = "Temperature (°F)", y = "Dewpoint")  
  
g +   
 annotate(geom = "text", x = 25, y = 60, fontface = "bold",   
 label = "This is a useful annotation")



However, now ggplot has drawn one text label per data point—that’s 1,461 labels and you only see one! You can solve that by setting the stat argument to "unique":

g +  
 geom\_text(aes(x = 25, y = 60,  
 label = "This is a useful annotation"),  
 stat = "unique")

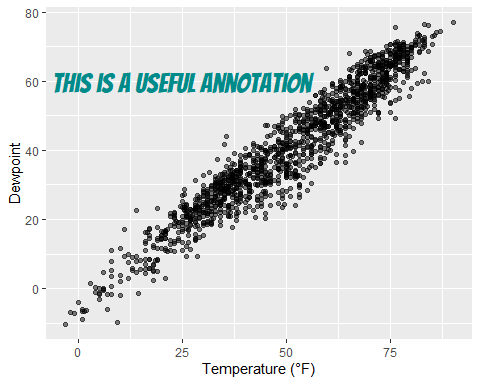
Warning in geom\_text(aes(x = 25, y = 60, label = "This is a useful annotation"), : All aesthetics have length 1, but the data has 1461 rows.  
ℹ Please consider using `annotate()` or provide this layer with data containing  
 a single row.



By the way, of course one can change the properties of the displayed text:

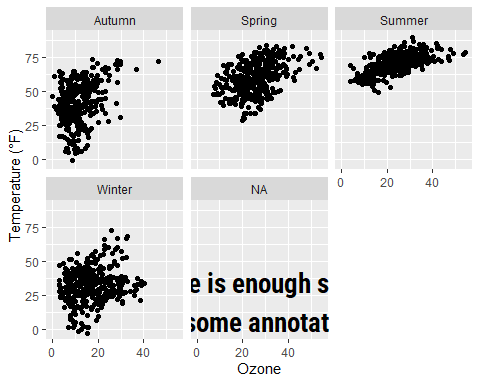
g +  
 geom\_text(aes(x = 25, y = 60,  
 label = "This is a useful annotation"),  
 stat = "unique", family = "Bangers",  
 size = 7, color = "darkcyan")

Warning in geom\_text(aes(x = 25, y = 60, label = "This is a useful annotation"), : All aesthetics have length 1, but the data has 1461 rows.  
ℹ Please consider using `annotate()` or provide this layer with data containing  
 a single row.



In case you use one of the facet functions to visualize your data you might run into trouble. One thing is that you may want to include the annotation only once:

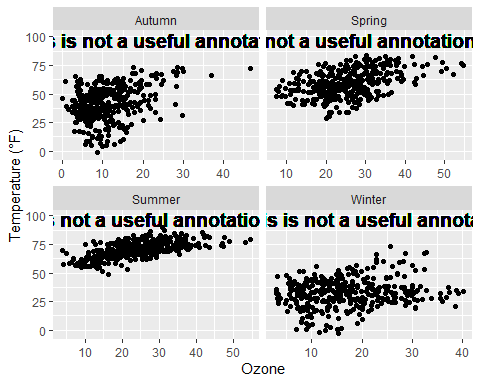
ann <- data.frame(  
 o3 = 30,  
 temp = 20,  
 season = factor("Summer", levels = levels(chic$season)),  
 label = "Here is enough space\nfor some annotations."  
)  
  
g <-  
 ggplot(chic, aes(x = o3, y = temp)) +  
 geom\_point() +  
 labs(x = "Ozone", y = "Temperature (°F)")  
  
g +  
 geom\_text(data = ann, aes(label = label),  
 size = 7, fontface = "bold",  
 family = "Roboto Condensed") +  
 facet\_wrap(~season)



Another challenge are facets in combination with free scales that might cut your text:

g +  
 geom\_text(aes(x = 23, y = 97,  
 label = "This is not a useful annotation"),  
 size = 5, fontface = "bold") +  
 scale\_y\_continuous(limits = c(NA, 100)) +  
 facet\_wrap(~season, scales = "free\_x")

Warning in geom\_text(aes(x = 23, y = 97, label = "This is not a useful annotation"), : All aesthetics have length 1, but the data has 1461 rows.  
ℹ Please consider using `annotate()` or provide this layer with data containing  
 a single row.



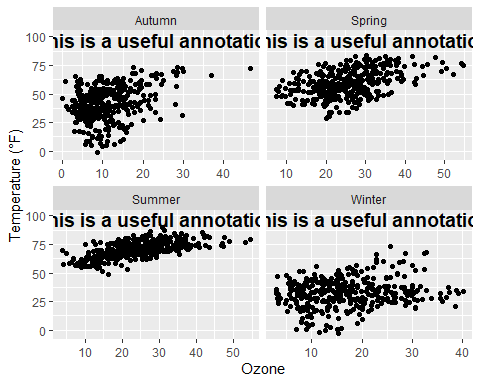
One solution is to calculate the midpoint of the axis, here x, beforehand:

ann <-  
 chic |>  
 dplyr::group\_by(season) |>  
 dplyr::summarize(  
 o3 = min(o3, na.rm = TRUE) +  
 (max(o3, na.rm = TRUE) - min(o3, na.rm = TRUE)) / 2  
)  
  
ann

# A tibble: 4 × 2  
 season o3  
 <chr> <dbl>  
1 Autumn 23.3  
2 Spring 31.0  
3 Summer 29.2  
4 Winter 21.5

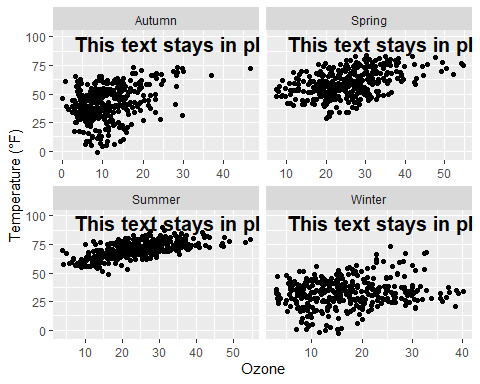
… and use the aggreated data to specify the placement of the annotation:

g +  
 geom\_text(data = ann,  
 aes(x = o3, y = 97,  
 label = "This is a useful annotation"),  
 size = 5, fontface = "bold") +  
 scale\_y\_continuous(limits = c(NA, 100)) +  
 facet\_wrap(~season, scales = "free\_x")



However, there is a simpler approach (in terms of fixing the cordinates)—but it also takes a while to know the code by heart. The {grid} package in combination with {ggplot2}’s annotation\_custom() allows you to specify the location based on scaled coordinates where 0 is low and 1 is high. grobTree() creates a grid graphical object and textGrob creates the text graphical object. The value of this is particularly evident when you have multiple plots with different scales.

library(grid)  
my\_grob <- grobTree(textGrob("This text stays in place!",  
 x = .1, y = .9, hjust = 0,  
 gp = gpar(col = "black",  
 fontsize = 15,  
 fontface = "bold")))  
  
g +  
 annotation\_custom(my\_grob) +  
 facet\_wrap(~season, scales = "free\_x") +  
 scale\_y\_continuous(limits = c(NA, 100))

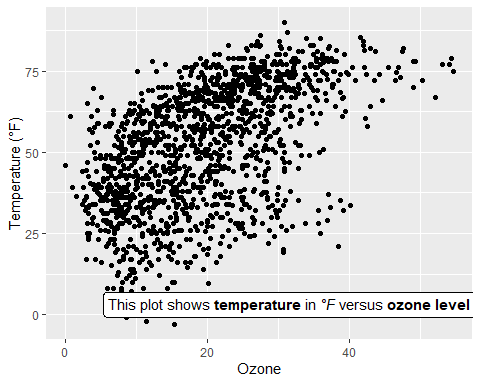


## 13.3 Use Markdown and HTML Rendering for Annotations

Again, we are using Claus Wilke’s [{ggtext} package](https://wilkelab.org/ggtext/) that is designed for improved text rendering support for {ggplot2}. The {ggtext} package defines two new theme elements, element\_markdown() and element\_textbox(). The package also provides additional geoms. geom\_richtext() is a replacement for geom\_text() and geom\_label() and renders text as markdown…

library(ggtext)  
  
lab\_md <- "This plot shows \*\*temperature\*\* in \*°F\* versus \*\*ozone level\*\* in \*ppm\*"  
  
g +  
 geom\_richtext(aes(x = 35, y = 3, label = lab\_md),  
 stat = "unique")

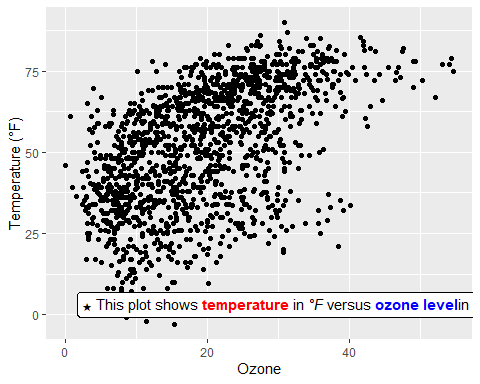
Warning in geom\_richtext(aes(x = 35, y = 3, label = lab\_md), stat = "unique"): All aesthetics have length 1, but the data has 1461 rows.  
ℹ Please consider using `annotate()` or provide this layer with data containing  
 a single row.



… or html:

lab\_html <- "&#9733; This plot shows <b style='color:red;'>temperature</b> in <i>°F</i> versus <b style='color:blue;'>ozone level</b>in <i>ppm</i> &#9733;"  
  
g +  
 geom\_richtext(aes(x = 33, y = 3, label = lab\_html),  
 stat = "unique")

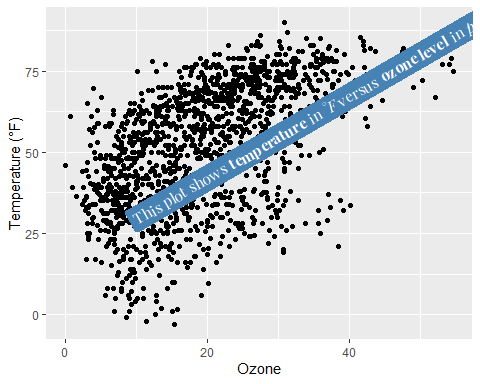
Warning in geom\_richtext(aes(x = 33, y = 3, label = lab\_html), stat = "unique"): All aesthetics have length 1, but the data has 1461 rows.  
ℹ Please consider using `annotate()` or provide this layer with data containing  
 a single row.



The geom comes with a lot of details one can modify, such as angle (which is not possible in the default geom\_text() and geom\_label()), properties of the box and properties of the text.

g +  
 geom\_richtext(aes(x = 10, y = 25, label = lab\_md),  
 stat = "unique", angle = 30,  
 color = "white", fill = "steelblue",  
 label.color = NA, hjust = 0, vjust = 0,  
 family = "Playfair Display")

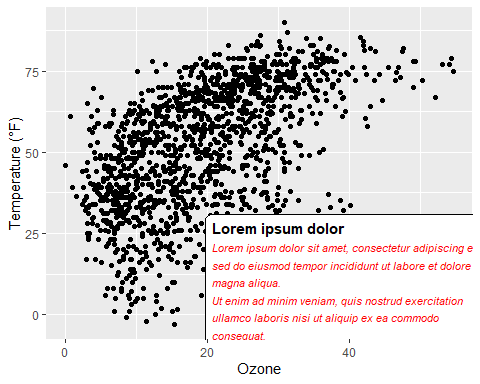
Warning in geom\_richtext(aes(x = 10, y = 25, label = lab\_md), stat = "unique", : All aesthetics have length 1, but the data has 1461 rows.  
ℹ Please consider using `annotate()` or provide this layer with data containing  
 a single row.



The other geom from the {ggtext} package is geom\_textbox(). This geom allows for dynamic wrapping of strings which is very useful for longer annotations such as info boxes and subtitles.

lab\_long <- "\*\*Lorem ipsum dolor\*\*<br><i style='font-size:8pt;color:red;'>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.<br>Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.</i>"  
  
g +  
 geom\_textbox(aes(x = 40, y = 10, label = lab\_long),  
 width = unit(15, "lines"), stat = "unique")

Warning in geom\_textbox(aes(x = 40, y = 10, label = lab\_long), width = unit(15, : All aesthetics have length 1, but the data has 1461 rows.  
ℹ Please consider using `annotate()` or provide this layer with data containing  
 a single row.



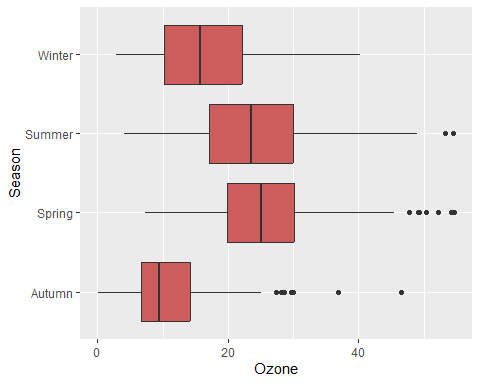
Note that it is not possible to either rotate the textbox (always horizontal) nor to change the justification of the text (always left-aligned).

# 14. Working with Coordinates

## 14.1 Flip a Plot

It is incredibly easy to flip a plot on its side. Here I have added the coord\_flip() which is all you need to flip the plot. This makes most sense when using geom’s to represent categorical data, for example bar charts or, as in the following example, box and whiskers plots:

ggplot(chic, aes(x = season, y = o3)) +  
 geom\_boxplot(fill = "indianred") +  
 labs(x = "Season", y = "Ozone") +  
 coord\_flip()

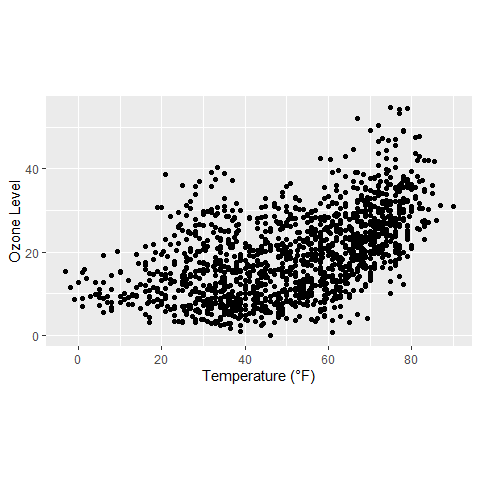


|  |
| --- |
| Using orientation = "y" |
| Since {ggplot2} version 3.0.0 it is also possible to draw geom’s horizontally via the argument orientation = "y". Expand to see example.  ggplot(chic, aes(x = o3, y = season)) +  geom\_boxplot(fill = "indianred", orientation = "y") +  labs(x = "Ozone", y = "Season") |

## 14.2 Fix an Axis

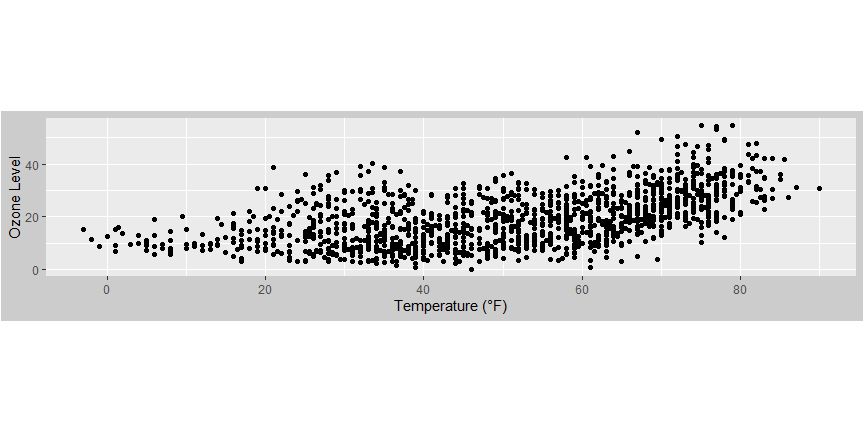
One can fix the aspect ratio of the Cartesian coordinate system and literally force a physical representation of the units along the x and y axes:

ggplot(chic, aes(x = temp, y = o3)) +  
 geom\_point() +  
 labs(x = "Temperature (°F)", y = "Ozone Level") +  
 scale\_x\_continuous(breaks = seq(0, 80, by = 20)) +  
 coord\_fixed(ratio = 1)



This way one can ensure not only a fixed step length on the axes but also that the exported plot looks as expected. However, your saved plot likely contains a lot of white space in case you do not use a suitable aspect ratio:

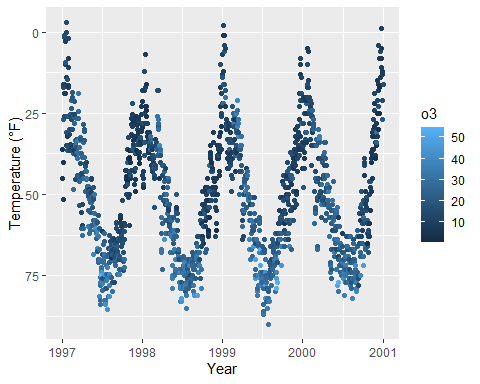
ggplot(chic, aes(x = temp, y = o3)) +  
 geom\_point() +  
 labs(x = "Temperature (°F)", y = "Ozone Level") +  
 scale\_x\_continuous(breaks = seq(0, 80, by = 20)) +  
 coord\_fixed(ratio = 1/3) +  
 theme(plot.background = element\_rect(fill = "grey80"))



## 14.3 Reverse an Axis

You can also easily reverse an axis using scale\_x\_reverse() or scale\_y\_reverse(), respectively:

ggplot(chic, aes(x = date, y = temp, color = o3)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 scale\_y\_reverse()



|  |
| --- |
| Note |
| Note that this will only work for continuous data. If you want to reverse discrete data, use the fct\_rev() function from the [{forcats} package](https://forcats.tidyverse.org/). Expand to see example.  ## the default ggplot(chic, aes(x = temp, y = season)) +  geom\_jitter(aes(color = season), show.legend = FALSE) +  labs(x = "Temperature (°F)", y = NULL)    library(forcats) set.seed(10) ggplot(chic, aes(x = temp, y = fct\_rev(season))) +  geom\_jitter(aes(color = season), show.legend = FALSE) +  labs(x = "Temperature (°F)", y = NULL) |

## 14.4 Transform an Axis

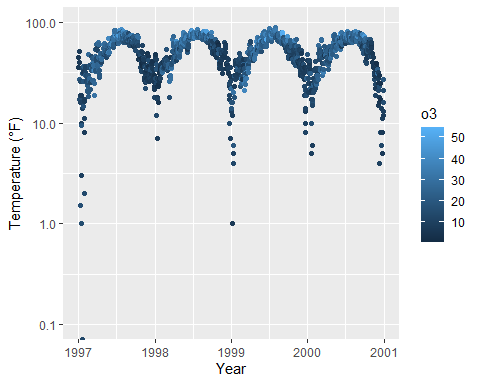
… or transform the default linear mapping by using scale\_y\_log10() or scale\_y\_sqrt(). As an example, here is a log10-transformed axis (which introduces NA’s in this case so be careful):

ggplot(chic, aes(x = date, y = temp, color = o3)) +  
 geom\_point() +  
 labs(x = "Year", y = "Temperature (°F)") +  
 scale\_y\_log10(lim = c(0.1, 100))

Warning in transformation$transform(x): NaNs produced

Warning in scale\_y\_log10(lim = c(0.1, 100)): log-10 transformation introduced  
infinite values.

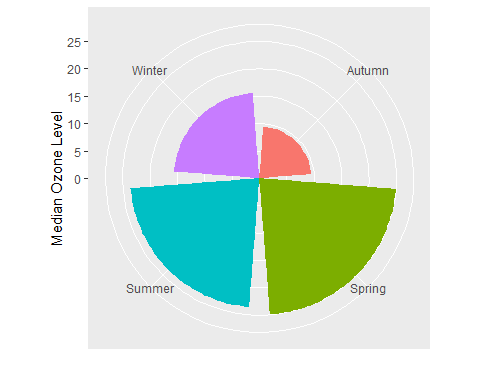
Warning: Removed 3 rows containing missing values or values outside the scale range  
(`geom\_point()`).



## 14.5 Circularize a Plot

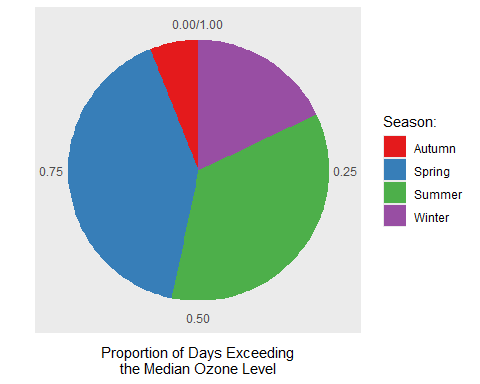
It is also possible to circularize (polarize?) the coordinate system by calling coord\_polar().

chic |>  
 dplyr::group\_by(season) |>  
 dplyr::summarize(o3 = median(o3)) |>  
 ggplot(aes(x = season, y = o3)) +  
 geom\_col(aes(fill = season), color = NA) +  
 labs(x = "", y = "Median Ozone Level") +  
 coord\_polar() +  
 guides(fill = "none")



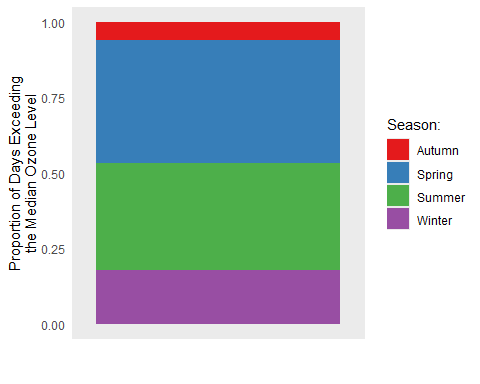
This coordinate system allows to draw pie charts as well:

chic\_sum <-  
 chic |>  
 dplyr::mutate(o3\_avg = median(o3)) |>  
 dplyr::filter(o3 > o3\_avg) |>  
 dplyr::mutate(n\_all = n()) |>  
 dplyr::group\_by(season) |>  
 dplyr::summarize(rel = n() / unique(n\_all))  
  
ggplot(chic\_sum, aes(x = "", y = rel)) +  
 geom\_col(aes(fill = season), width = 1, color = NA) +  
 labs(x = "", y = "Proportion of Days Exceeding\nthe Median Ozone Level") +  
 coord\_polar(theta = "y") +  
 scale\_fill\_brewer(palette = "Set1", name = "Season:") +  
 theme(axis.ticks = element\_blank(),  
 panel.grid = element\_blank())



I suggest to always look also at the outcome of the same code in a Cartesian coordinate system, which is the default, to understand the logic behind coord\_polar() and theta:

ggplot(chic\_sum, aes(x = "", y = rel)) +  
 geom\_col(aes(fill = season), width = 1, color = NA) +  
 labs(x = "", y = "Proportion of Days Exceeding\nthe Median Ozone Level") +  
 #coord\_polar(theta = "y") +  
 scale\_fill\_brewer(palette = "Set1", name = "Season:") +  
 theme(axis.ticks = element\_blank(),  
 panel.grid = element\_blank())



# 15. Working with Chart Types

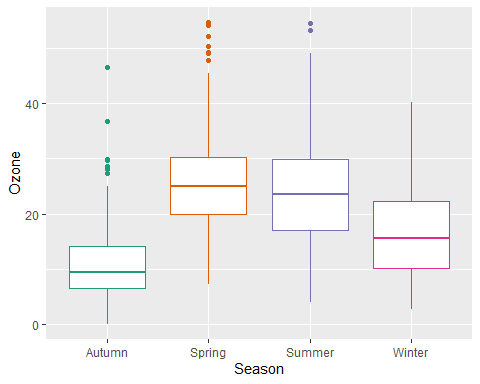
## 15.1 Alternatives to a Box Plot

Box plots are great, but they can be so incredibly boring. Also, even if you are used to looking at box plots, remember there might be plenty people looking at your plot that have never seen a box and whisker plot before.

|  |
| --- |
| Recall: Box and Whiskers Plot |
| A box-and-whisker plot (sometimes called simply a box plot) is a histogram-like method of displaying data, invented by J. Tukey. The thick **middle line** notates the median, also known as quartile Q2. The limits of the **box** are determined by the lower and upper quartiles, Q1 and Q3. The box contains thus 50% of the data and is called *“interquartile range”* (IQR). The length of the **whiskers** is determined by the most extreme values that are not considered as outliers (i.e. values that are within 3/2 times the interquartile range). Fig: Structure of a Boxplot |

There are alternatives, but first we are plotting a common box plot:

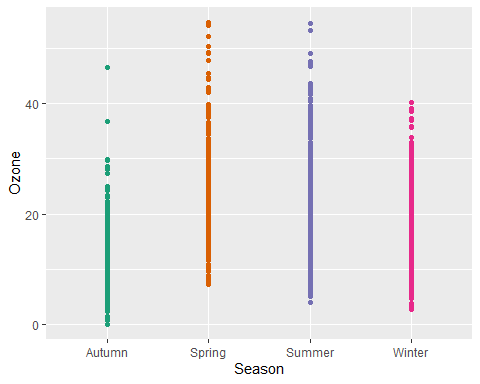
g <-  
 ggplot(chic, aes(x = season, y = o3,  
 color = season)) +  
 labs(x = "Season", y = "Ozone") +  
 scale\_color\_brewer(palette = "Dark2", guide = "none")  
  
g + geom\_boxplot()



### 15.1.1 Alternative: Plot of Points

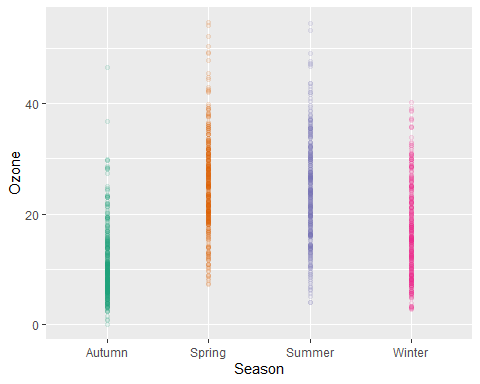
Let’s plot just each data point of the raw data:

g + geom\_point()



Not only boring but uninformative. To improve the plot, one could add transparency to deal with overplotting:

g + geom\_point(alpha = .1)

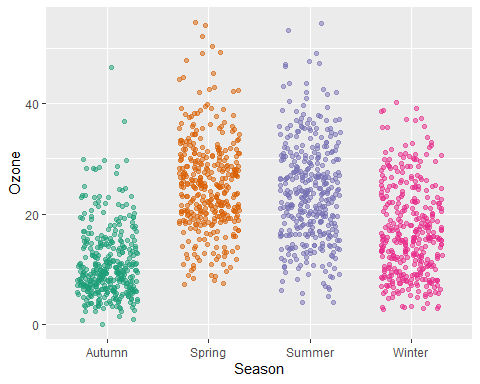


However, setting transparency is difficult here since either the overlap is still too high or the extreme values are not visible. Bad, so let’s try something else.

### 15.1.2 Alternative: Jitter the Points

Try adding a little jitter to the data. I like this for in-house visualization but be careful using jittering because you are purposely adding noise to your data and this can result in misinterpretation of your data.

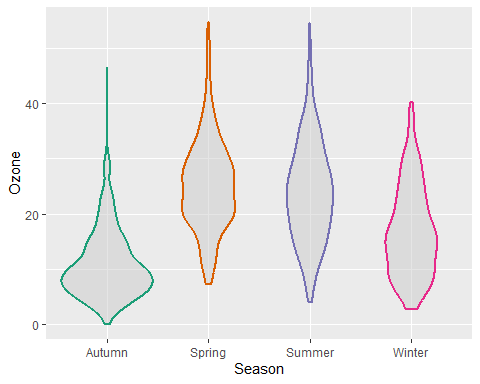
g + geom\_jitter(width = .3, alpha = .5)



### 15.1.3 Alternative: Violin Plots

Violin plots, similar to box plots except you are using a kernel density to show where you have the most data, are a useful visualization.

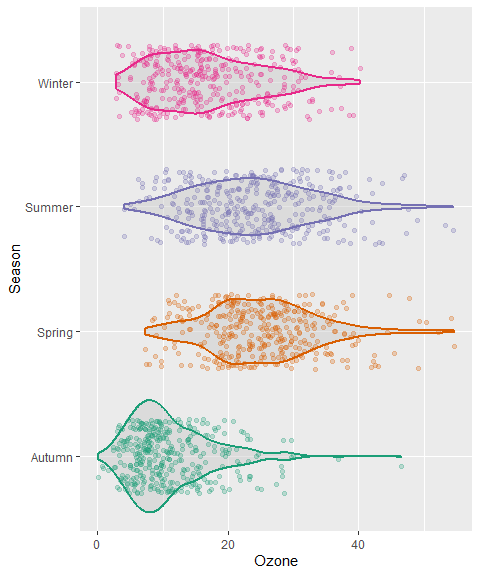
g + geom\_violin(fill = "gray80", linewidth = 1, alpha = .5)



### 15.1.4 Alternative: Combining Violin Plots with Jitter

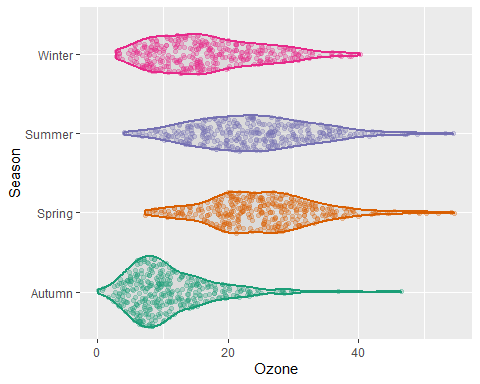
We can of course combine both, estimated densities and the raw data points:

g + geom\_violin(fill = "gray80", linewidth = 1, alpha = .5) +  
 geom\_jitter(alpha = .25, width = .3) +  
 coord\_flip()



The [{ggforce} package](https://ggforce.data-imaginist.com/) provides so-called sina functions where the width of the jitter is controlled by the density distribution of the data—that makes the jittering a bit more visually appealing:

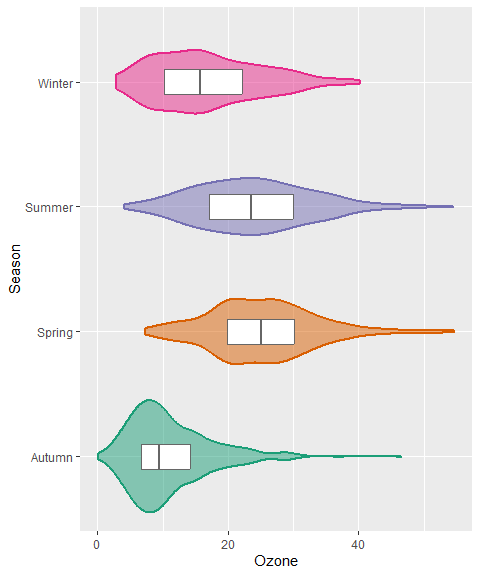
library(ggforce)  
  
g + geom\_violin(fill = "gray80", linewidth = 1, alpha = .5) +  
 geom\_sina(alpha = .25) +  
 coord\_flip()



### 15.1.5 Alternative: Combining Violin Plots with Box Plots

To allow for easy estimation of quantiles, we can also add the box of the box plot inside the violins to indicate 25%-quartile, median and 75%-quartile:

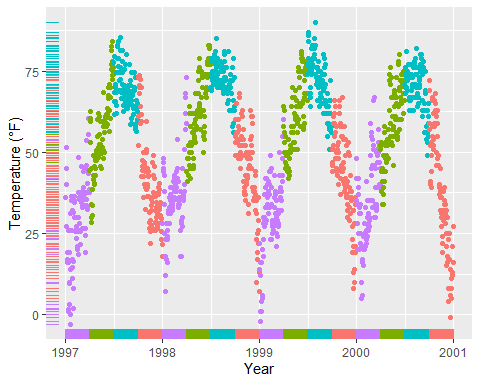
g + geom\_violin(aes(fill = season), linewidth = 1, alpha = .5) +  
 geom\_boxplot(outlier.alpha = 0, coef = 0,  
 color = "gray40", width = .2) +  
 scale\_fill\_brewer(palette = "Dark2", guide = "none") +  
 coord\_flip()



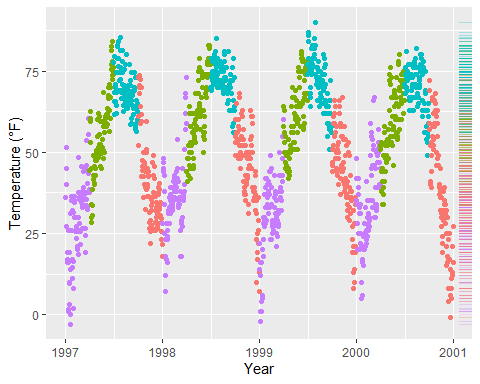
## 15.2 Create a Rug Representation to a Plot

A rug represents the data of a single quantitative variable, displayed as marks along an axis. In most cases, it is used in addition to scatter plots or heatmaps to visualize the overall distribution of one or both of the variables:

ggplot(chic, aes(x = date, y = temp,  
 color = season)) +  
 geom\_point(show.legend = FALSE) +  
 geom\_rug(show.legend = FALSE) +  
 labs(x = "Year", y = "Temperature (°F)")



ggplot(chic, aes(x = date, y = temp, color = season)) +  
 geom\_point(show.legend = FALSE) +  
 geom\_rug(sides = "r", alpha = .3, show.legend = FALSE) +  
 labs(x = "Year", y = "Temperature (°F)")



## 15.3 Create a Correlation Matrix

There are several packages that allow to create correlation matrix plots, some also using the{ggplot2} infrastructure and thus returning ggplots. I am going to show you how to do this without extension packages.

First step is to create the correlation matrix. Here, we use the {corrr} package that works nicely with pipes but there are also many others out there. We are using Pearson because all the variables are fairly normally distributed (but you may consider Spearman if your variables follow a different pattern). Note that since a correlation matrix has redundant information we are setting half of it to NA.

corm <-  
 chic |>  
 dplyr::select(temp, dewpoint, pm10, o3) |>  
 corrr::correlate(diagonal = 1) |>  
 corrr::shave(upper = FALSE)

Correlation computed with  
• Method: 'pearson'  
• Missing treated using: 'pairwise.complete.obs'

library(gt)  
corm %>% gt()

| term | temp | dewpoint | pm10 | o3 |
| --- | --- | --- | --- | --- |
| temp | 1 | 0.9577391 | 0.3679648 | 0.5349655 |
| dewpoint | NA | 1.0000000 | 0.3274569 | 0.4539134 |
| pm10 | NA | NA | 1.0000000 | 0.2060732 |
| o3 | NA | NA | NA | 1.0000000 |

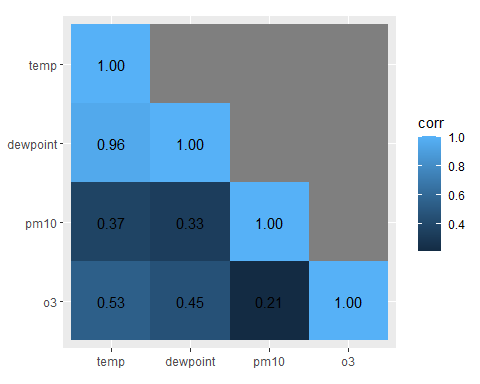
Now we put the resulting matrix in **long** format using the pivot\_longer() function from the {tidyr} package. We also directly format the labels and place empty quotes for the upper triangle. Note that I use sprintf() to ensure that the label always display two digits.

corm <- corm |>  
 tidyr::pivot\_longer(  
 cols = -term,  
 names\_to = "colname",  
 values\_to = "corr"  
 ) |>  
 dplyr::mutate(  
 rowname = forcats::fct\_inorder(term),  
 colname = forcats::fct\_inorder(colname),  
 label = dplyr::if\_else(is.na(corr), "", sprintf("%1.2f", corr))  
 )

|  | term | colname | corr | label |
| --- | --- | --- | --- | --- |
| temp | temp | temp | 1.0000000 | 1.00 |
| temp | temp | dewpoint | 0.9577391 | 0.96 |
| temp | temp | pm10 | 0.3679648 | 0.37 |
| temp | temp | o3 | 0.5349655 | 0.53 |
| dewpoint | dewpoint | temp | NA |  |
| dewpoint | dewpoint | dewpoint | 1.0000000 | 1.00 |
| dewpoint | dewpoint | pm10 | 0.3274569 | 0.33 |
| dewpoint | dewpoint | o3 | 0.4539134 | 0.45 |
| pm10 | pm10 | temp | NA |  |
| pm10 | pm10 | dewpoint | NA |  |
| pm10 | pm10 | pm10 | 1.0000000 | 1.00 |
| pm10 | pm10 | o3 | 0.2060732 | 0.21 |
| o3 | o3 | temp | NA |  |
| o3 | o3 | dewpoint | NA |  |
| o3 | o3 | pm10 | NA |  |
| o3 | o3 | o3 | 1.0000000 | 1.00 |

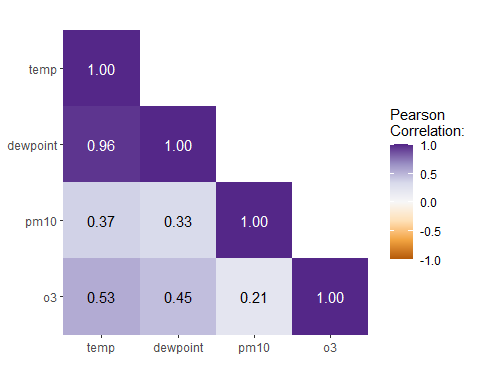
For the plot we will use geom\_tile() for the heatmap and geom\_text() for the labels:

ggplot(corm, aes(rowname, fct\_rev(colname),  
 fill = corr)) +  
 geom\_tile() +  
 geom\_text(aes(label = label)) +  
 coord\_fixed() +  
 labs(x = NULL, y = NULL)



I like to have a diverging color palette—it is important that the scale is centered at zero correlation!—with white indicating missing data. Also I like to have no grid lines and padding around the heatmap as well as labels that are colored depending on the underlying fill:

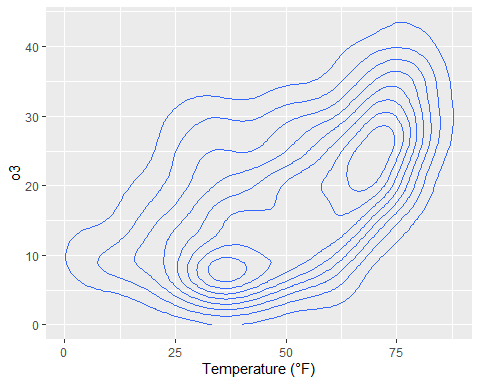
ggplot(corm, aes(rowname, fct\_rev(colname),  
 fill = corr)) +  
 geom\_tile() +  
 geom\_text(aes(  
 label = label,  
 color = abs(corr) < .75  
 )) +  
 coord\_fixed(expand = FALSE) +  
 scale\_color\_manual(  
 values = c("white", "black"),  
 guide = "none"  
 ) +  
 scale\_fill\_distiller(  
 palette = "PuOr", na.value = "white",  
 direction = 1, limits = c(-1, 1),  
 name = "Pearson\nCorrelation:"  
 ) +  
 labs(x = NULL, y = NULL) +  
 theme(panel.border = element\_rect(color = NA, fill = NA),  
 legend.position.inside = c(.85, .8))



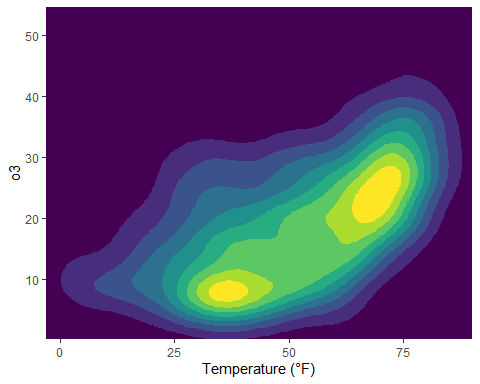
## 15.4 Create a Contour Plot

Contour plots are nice way to display eatesholds of values. One can use them to bin data, showing the density of observations:

ggplot(chic, aes(temp, o3)) +  
 geom\_density\_2d() +  
 labs(x = "Temperature (°F)", x = "Ozone Level")

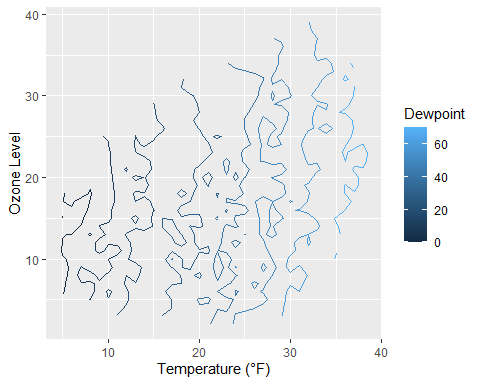


ggplot(chic, aes(temp, o3)) +  
 geom\_density\_2d\_filled(show.legend = FALSE) +  
 coord\_cartesian(expand = FALSE) +  
 labs(x = "Temperature (°F)", x = "Ozone Level")



But now, we are plotting three-dimensional data. We are going to plot the thresholds in dewpoint (i.e. [the temperature at which airborne water vapor will condense to form liquid dew](https://en.wikipedia.org/wiki/Dew_point)) related to temperature and ozone levels:

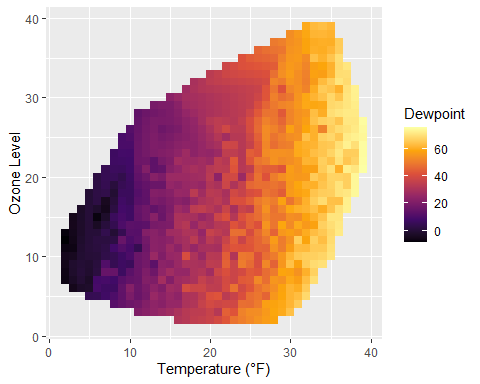
## interpolate data  
fld <- with(chic, akima::interp(x = temp, y = o3, z = dewpoint))  
  
## prepare data in long format  
df <- fld$z |>  
 tibble::as\_tibble(.name\_repair = "universal\_quiet") |>   
 dplyr::mutate(x = dplyr::row\_number()) |>  
 tidyr::pivot\_longer(  
 cols = -x,  
 names\_to = "y",  
 names\_transform = as.integer,  
 values\_to = "Dewpoint",   
 names\_prefix = "...",  
 values\_drop\_na = TRUE  
 )  
  
g <- ggplot(data = df, aes(x = x, y = y, z = Dewpoint)) +  
 labs(x = "Temperature (°F)", y = "Ozone Level",  
 color = "Dewpoint")  
  
g + stat\_contour(aes(color = after\_stat(level)))



Surprise! As it is defined, the drew point is in most cases equal to the measured temperature.

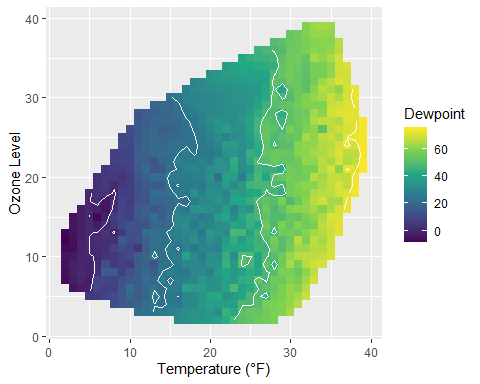
The lines are indicating different levels of drew points, but this is not a pretty plot and also hard to read due to missing borders. Let’s try a tile plot using the viridis color palette to encode the dewpoint of each combination of ozone level and temperature:

g + geom\_tile(aes(fill = Dewpoint)) +  
 scale\_fill\_viridis\_c(option = "inferno")



How does it look if we combine a contour plot and a tile plot to fill the area under the contour lines?

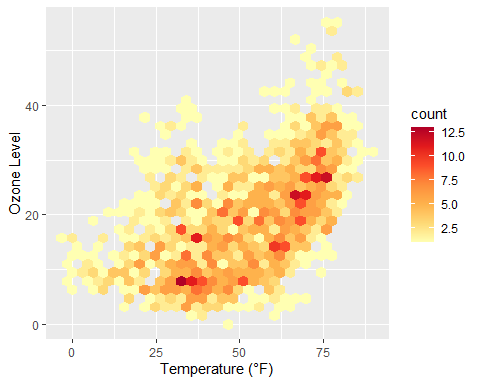
g + geom\_tile(aes(fill = Dewpoint)) +  
 stat\_contour(color = "white", linewidth = .7, bins = 5) +  
 scale\_fill\_viridis\_c()



## 15.5 Create a Heatmap of Counts

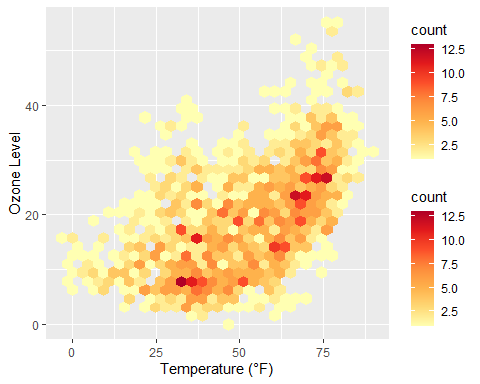
Similarly to our first contour maps, one can easily show the counts or densities of points binned to a hexagonal grid via geom\_hex():

library(hexbin)  
ggplot(chic, aes(temp, o3)) +  
 geom\_hex() +  
 scale\_fill\_distiller(palette = "YlOrRd", direction = 1) +  
 labs(x = "Temperature (°F)", y = "Ozone Level")



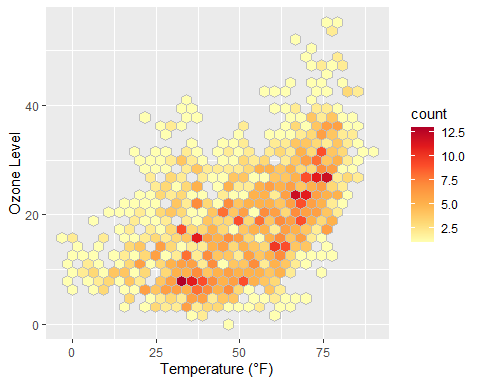
Often, white lines pop up in the resulting plot. One can fix that by mapping also color to either after\_stat(count) (the default) or after\_stat(density)…

ggplot(chic, aes(temp, o3)) +  
 geom\_hex(aes(color = after\_stat(count))) +  
 scale\_fill\_distiller(palette = "YlOrRd", direction = 1) +  
 scale\_color\_distiller(palette = "YlOrRd", direction = 1) +  
 labs(x = "Temperature (°F)", y = "Ozone Level")



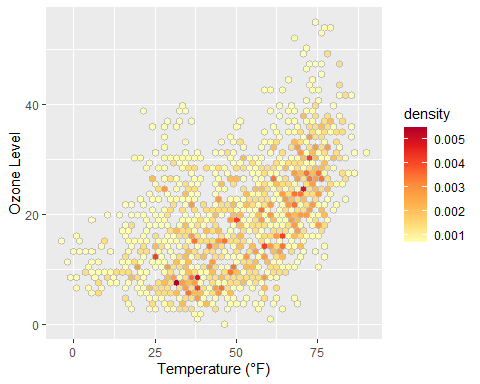
… or by setting the same color as outline for all hexagonal cells:

ggplot(chic, aes(temp, o3)) +  
 geom\_hex(color = "grey") +  
 scale\_fill\_distiller(palette = "YlOrRd", direction = 1) +  
 labs(x = "Temperature (°F)", y = "Ozone Level")



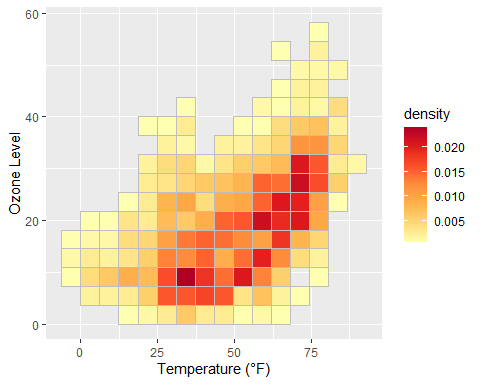
One can also change the default binning to in- or decrease the number of hexagonal cells:

ggplot(chic, aes(temp, o3, fill = after\_stat(density))) +  
 geom\_hex(bins = 50, color = "grey") +  
 scale\_fill\_distiller(palette = "YlOrRd", direction = 1) +  
 labs(x = "Temperature (°F)", y = "Ozone Level")



If you want to have a regular grid, one can also use geom\_bin2d() which summarizes the data to rectangular grid cells based on bins:

ggplot(chic, aes(temp, o3, fill = after\_stat(density))) +  
 geom\_bin2d(bins = 15, color = "grey") +  
 scale\_fill\_distiller(palette = "YlOrRd", direction = 1) +  
 labs(x = "Temperature (°F)", y = "Ozone Level")



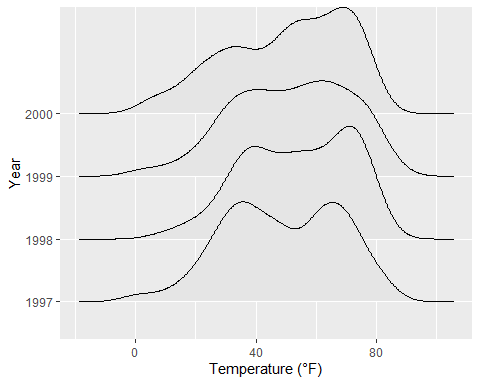
## 15.6 Create a Ridge Plot

*Ridge(line) plots* are a new type of plots which is very popular at the moment.

While you can create those plots with [basic {ggplot2} commands](https://github.com/halhen/viz-pub/blob/master/sports-time-of-day/2_gen_chart.R) the popularity lead to a package that make it easier create those plots: [{ggridges}](https://wilkelab.org/ggridges/). We are going to use this package here.

library(ggridges)  
ggplot(chic, aes(x = temp, y = factor(year))) +  
 geom\_density\_ridges(fill = "gray90") +  
 labs(x = "Temperature (°F)", y = "Year")

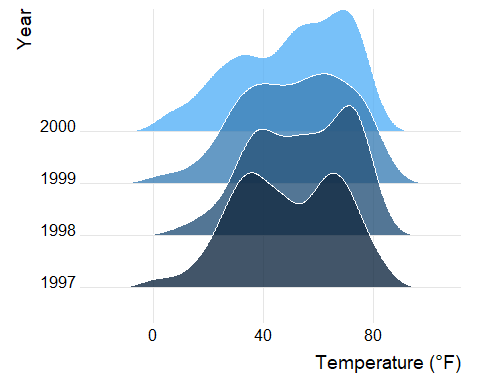
Picking joint bandwidth of 5.23



You can easily specify the overlap and the trailing tails by using the arguments rel\_min\_height and scale, respectively. The package also comes with its own theme (but I would prefer to build my own, see chapter [“Create and Use Your Custom Theme”](#themes)). Additionally, we change the colors based on year to make it more appealing.

ggplot(chic, aes(x = temp, y = factor(year), fill = year)) +  
 geom\_density\_ridges(alpha = .8, color = "white",  
 scale = 2.5, rel\_min\_height = .01) +  
 labs(x = "Temperature (°F)", y = "Year") +  
 guides(fill = "none") +  
 theme\_ridges()

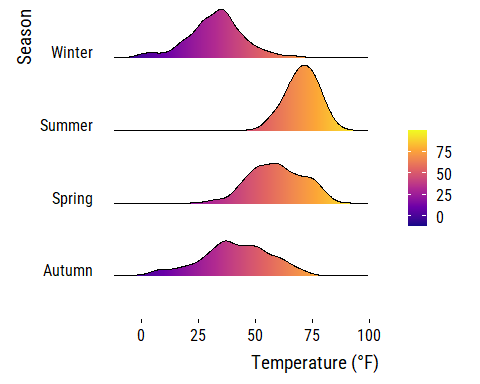
Picking joint bandwidth of 5.23



You can also get rid of the overlap using values below 1 for the scaling argument (but this somehow contradicts the idea of ridge plots…). Here is an example additionally using the viridis color gradient and the in-build theme:

ggplot(chic, aes(x = temp, y = season, fill = after\_stat(x))) +  
 geom\_density\_ridges\_gradient(scale = .9, gradient\_lwd = .5,  
 color = "black") +  
 scale\_fill\_viridis\_c(option = "plasma", name = "") +  
 labs(x = "Temperature (°F)", y = "Season") +  
 theme\_ridges(font\_family = "Roboto Condensed", grid = FALSE)

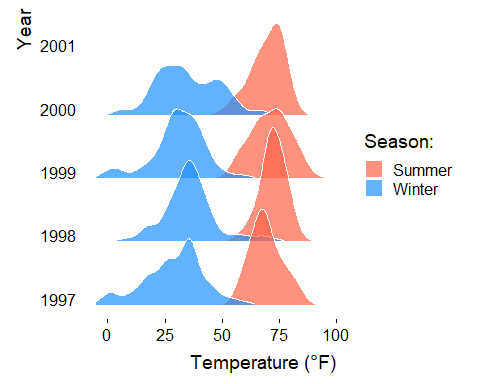
Picking joint bandwidth of 2.99



We can also compare several groups per ridgeline and coloring them according to their group. This follows the idea of [Marc Belzunces](https://twitter.com/marcbeldata/status/888697140268204032).

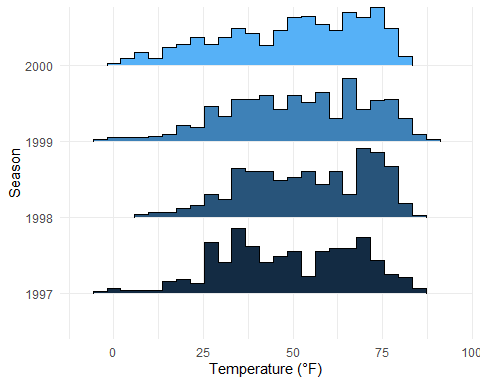
library(dplyr)  
  
## only plot extreme season using dplyr from the tidyverse  
ggplot(data = dplyr::filter(chic, season %in% c("Summer", "Winter")),  
 aes(x = temp, y = year, fill = paste(year, season))) +  
 geom\_density\_ridges(alpha = .7, rel\_min\_height = .01,  
 color = "white", from = -5, to = 95) +  
 scale\_fill\_cyclical(breaks = c("1997 Summer", "1997 Winter"),  
 labels = c(`1997 Summer` = "Summer",  
 `1997 Winter` = "Winter"),  
 values = c("tomato", "dodgerblue"),  
 name = "Season:", guide = "legend") +  
 theme\_ridges(grid = FALSE) +  
 labs(x = "Temperature (°F)", y = "Year")

Picking joint bandwidth of 3.17



The {ggridges} package is also helpful to create histograms for different groups using stat = "binline" in the geom\_density\_ridges() command:

ggplot(chic, aes(x = temp, y = factor(year), fill = year)) +  
 geom\_density\_ridges(stat = "binline", bins = 25, scale = .9,  
 draw\_baseline = FALSE, show.legend = FALSE) +  
 theme\_minimal() +  
 labs(x = "Temperature (°F)", y = "Season")

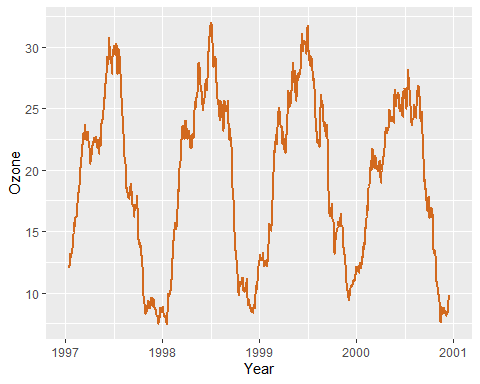


# 16. Working with Ribbons (AUC, CI, etc.)

This is not a perfect dataset for demonstrating this, but using ribbon can be useful. In this example we will create a 30-day running average using the filter() function so that our ribbon is not too noisy.

chic$o3run <- as.numeric(stats::filter(chic$o3, rep(1/30, 30), sides = 2))  
  
ggplot(chic, aes(x = date, y = o3run)) +  
 geom\_line(color = "chocolate", lwd = .8) +  
 labs(x = "Year", y = "Ozone")

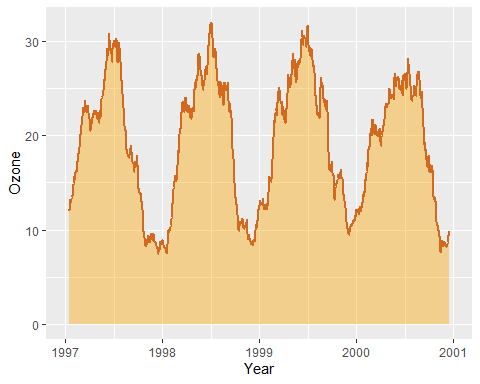
Warning: Removed 29 rows containing missing values or values outside the scale range  
(`geom\_line()`).



How does it look if we fill in the area below the curve using the geom\_ribbon() function?

ggplot(chic, aes(x = date, y = o3run)) +  
 geom\_ribbon(aes(ymin = 0, ymax = o3run),  
 fill = "orange", alpha = .4) +  
 geom\_line(color = "chocolate", lwd = .8) +  
 labs(x = "Year", y = "Ozone")

Warning: Removed 29 rows containing missing values or values outside the scale range  
(`geom\_line()`).



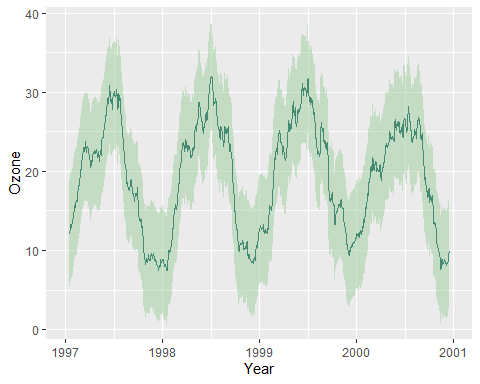
Nice to indicate the [area under the curve (AUC)](https://en.wikipedia.org/wiki/Area_under_the_curve_(pharmacokinetics)) but this is not the conventional way to use geom\_ribbon().

|  |
| --- |
| Using geom\_area() |
| Actually a nicer way to achieve the same is geom\_area().  ggplot(chic, aes(x = date, y = o3run)) +  geom\_area(color = "chocolate", lwd = .8,  fill = "orange", alpha = .4) +  labs(x = "Year", y = "Ozone")  Warning: Removed 29 rows containing non-finite outside the scale range (`stat\_align()`). |

Instead, we draw a ribbon that gives us one standard deviation above and below our data:

chic$mino3 <- chic$o3run - sd(chic$o3run, na.rm = TRUE)  
chic$maxo3 <- chic$o3run + sd(chic$o3run, na.rm = TRUE)  
  
ggplot(chic, aes(x = date, y = o3run)) +  
 geom\_ribbon(aes(ymin = mino3, ymax = maxo3), alpha = .5,  
 fill = "darkseagreen3", color = "transparent") +  
 geom\_line(color = "aquamarine4", lwd = .7) +  
 labs(x = "Year", y = "Ozone")

Warning: Removed 29 rows containing missing values or values outside the scale range  
(`geom\_line()`).



# 17. Working with Smoothings

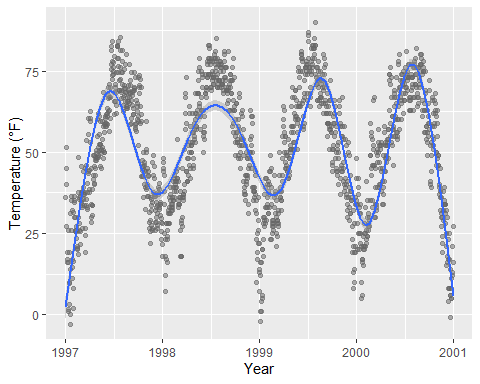
It is amazingly easy to add smoothing to your data using {ggplot2}.

## 17.1 Default: Adding a LOESS or GAM Smoothing

You can simply use stat\_smooth()—not even a formula is required. This adds a LOESS (locally weighted scatter plot smoothing, method = "loess") if you have fewer than 1000 points or a GAM (generalized additive model, method = "gam") otherwise. Since we have more than 1000 points, the smoothing is based on a GAM:

ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "gray40", alpha = .5) +  
 stat\_smooth() +  
 labs(x = "Year", y = "Temperature (°F)")

`geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'



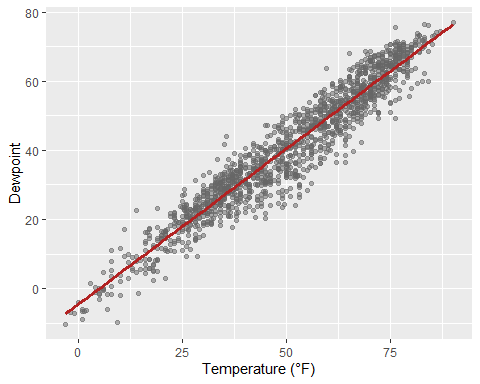
|  |
| --- |
| Note |
| In most cases one wants the points to be on top of the ribbon so make sure you always call the smoothing before you add the points. |

## 17.2 Adding a Linear Fit

Though the default is a LOESS or GAM smoothing, it is also easy to add a standard linear fit:

ggplot(chic, aes(x = temp, y = dewpoint)) +  
 geom\_point(color = "gray40", alpha = .5) +  
 stat\_smooth(method = "lm", se = FALSE,  
 color = "firebrick", linewidth = 1.3) +  
 labs(x = "Temperature (°F)", y = "Dewpoint")

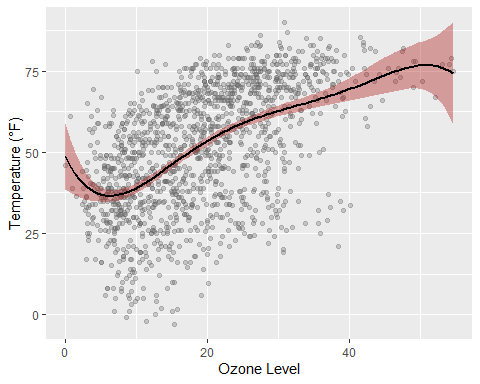
`geom\_smooth()` using formula = 'y ~ x'



## 17.3 Specifying the Formula for Smoothing

{ggplot2} allows you to specify the model you want it to use. Maybe you want to use a [polynomial regression](https://en.wikipedia.org/wiki/Polynomial_regression)?

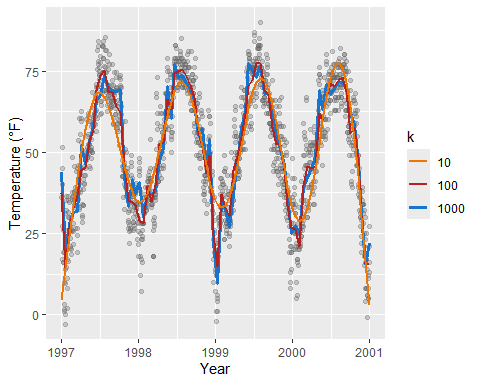
ggplot(chic, aes(x = o3, y = temp)) +  
 geom\_point(color = "gray40", alpha = .3) +  
 geom\_smooth(  
 method = "lm",  
 formula = y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5),  
 color = "black",  
 fill = "firebrick"  
 ) +  
 labs(x = "Ozone Level", y = "Temperature (°F)")



|  |
| --- |
| Difference between geom and stat |
| Huh, geom\_smooth()? There is an important difference between geom and stat but here it really doesn’t matter which one you use. Expand to compare both.  ggplot(chic, aes(x = o3, y = temp)) +  geom\_point(color = "gray40", alpha = .3) +  geom\_smooth(stat = "smooth") + ## the default  labs(x = "Ozone Level", y = "Temperature (°F)")  `geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'    ggplot(chic, aes(x = o3, y = temp)) +  geom\_point(color = "gray40", alpha = .3) +  stat\_smooth(geom = "smooth") + ## the default  labs(x = "Ozone Level", y = "Temperature (°F)")  `geom\_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")' |

Or lets say you want to increase the GAM dimension (add some additional wiggles to the smooth):

cols <- c("darkorange2", "firebrick", "dodgerblue3")  
  
ggplot(chic, aes(x = date, y = temp)) +  
 geom\_point(color = "gray40", alpha = .3) +  
 stat\_smooth(aes(col = "1000"),  
 method = "gam",  
 formula = y ~ s(x, k = 1000),  
 se = FALSE, linewidth = 1.3) +  
 stat\_smooth(aes(col = "100"),  
 method = "gam",  
 formula = y ~ s(x, k = 100),  
 se = FALSE, linewidth = 1) +  
 stat\_smooth(aes(col = "10"),  
 method = "gam",  
 formula = y ~ s(x, k = 10),  
 se = FALSE, linewidth = .8) +  
 scale\_color\_manual(name = "k", values = cols) +  
 labs(x = "Year", y = "Temperature (°F)")



# 18. Working with Interactive Plots

The following collection lists libraries that can be used in combination with {ggplot2} or on their own to create interactive visualizations in R (often making use of existing JavaScript libraries).

## 18.1 Combination of {ggplot2} and {shiny}

{shiny} is a package from [RStudio](https://rstudio.com/) that makes it incredibly easy to build interactive web applications with R. For an introduction and live examples, visit the [Shiny homepage](http://shiny.rstudio.com/).

To look at the potential use, you can check out the Hello Shiny examples. This is the first one:

library(shiny)  
runExample("01\_hello")

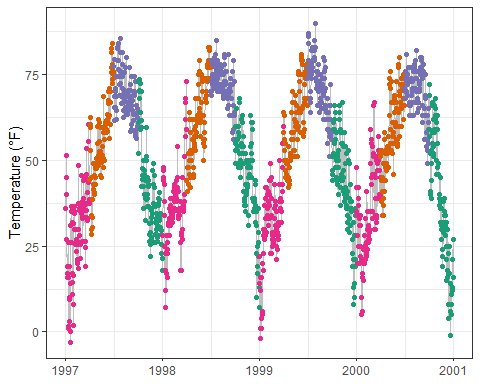
Of course, one can use ggplots in these apps. This example demonstrates the possibility to add some interactive user experience:

runExample("04\_mpg")

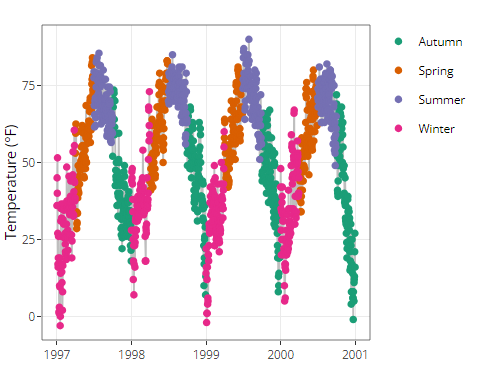
## 18.2 Plot.ly via {plotly} and {ggplot2}

[Plot.ly](https://chart-studio.plotly.com/feed/#/) is a tool for creating online, interactive graphics and web apps. The [{plotly} package](https://plot.ly/r/getting-started/) enables you to create those directly from your {ggplot2} plots and the workflow is surprisingly easy and [can be done from within R](https://plotly-r.com/). However, some of your theme settings might be changed and need to be modified manually afterwards. Also, and unfortunately, it is not straightforward to create facets or true multi-panel plots that scale nicely.

g <- ggplot(chic, aes(date, temp)) +  
 geom\_line(color = "grey") +  
 geom\_point(aes(color = season)) +  
 scale\_color\_brewer(palette = "Dark2", guide = "none") +  
 labs(x = NULL, y = "Temperature (°F)") +  
 theme\_bw()  
  
g



library(plotly)  
  
ggplotly(g)

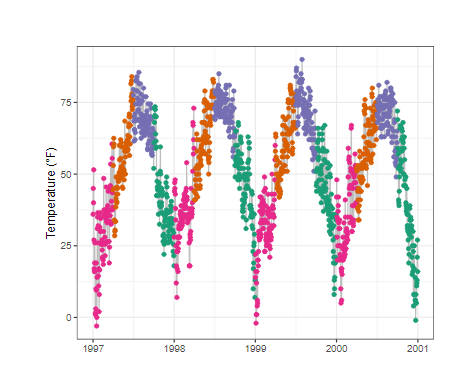


Here, for example, it keeps the overall theme setting but adds the legend again.

## 18.3 ggiraph and ggplot2

[{ggiraph}](https://davidgohel.github.io/ggiraph/index.html) is an R package that allows you to create dynamic {ggplot2} graphs. This allows you to add tooltips, animations and JavaScript actions to the graphics. The package also allows the selection of graphical elements when used in Shiny applications.

library(ggiraph)  
  
g <- ggplot(chic, aes(date, temp)) +  
 geom\_line(color = "grey") +  
 geom\_point\_interactive(  
 aes(color = season, tooltip = season, data\_id = season)  
 ) +  
 scale\_color\_brewer(palette = "Dark2", guide = "none") +  
 labs(x = NULL, y = "Temperature (°F)") +  
 theme\_bw()  
  
girafe(ggobj = g)



## 18.4 Highcharts via {highcharter}

[Highcharts](https://www.highcharts.com/), a software library for interactive charting, is another visualization library written in pure JavaScript that has been ported to R. The package [{highcharter}](https://jkunst.com/highcharter/) makes it possible to use them—but be aware that Highcharts is only free in case of non-commercial use.

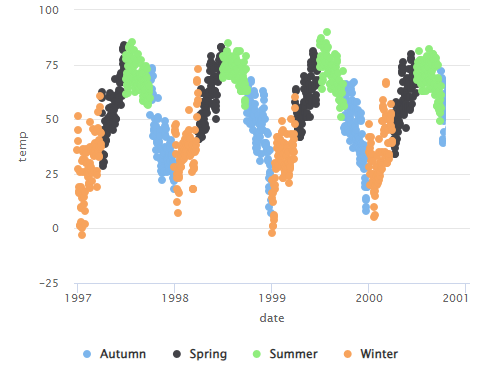
library(highcharter)

Registered S3 method overwritten by 'quantmod':  
 method from  
 as.zoo.data.frame zoo

Highcharts (www.highcharts.com) is a Highsoft software product which is

not free for commercial and Governmental use

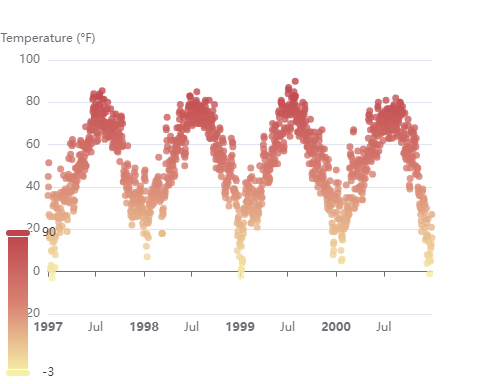
hchart(chic, "scatter", hcaes(x = date, y = temp, group = season))



## 18.5 Echarts via {echarts4r}

[Apache ECharts](https://echarts.apache.org/en/index.html) is a free, powerful charting and visualization library offering an easy way of building intuitive, interactive, and highly customizable charts. Even though it is written in pure JavaScript, one can use it in R via the [{echarts4r} library](https://echarts4r.john-coene.com/) thanks to [John Coene](https://john-coene.com/). Check out the impressive [example gallery](https://echarts4r.john-coene.com/articles/chart_types.html) or [this app](https://johncoene.shinyapps.io/fopi-contest/) made by the package developer John Coene.

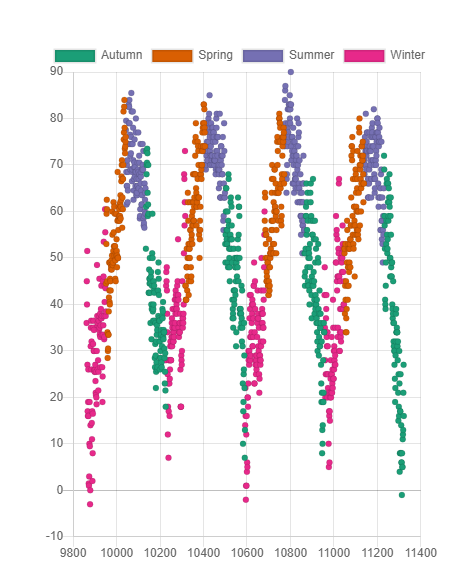
library(echarts4r)  
  
chic |>  
 e\_charts(date) |>  
 e\_scatter(temp, symbol\_size = 7) |>  
 e\_visual\_map(temp) |>  
 e\_y\_axis(name = "Temperature (°F)") |>  
 e\_legend(FALSE)



## 18.6 Chart.js via {charter}

[charter](https://github.com/JohnCoene/charter) is another package developed by John Coene that enables the use of a JavaScript visualization library in R. The package allows you to build interactive plots with the help of the [Charts.js framework](https://www.chartjs.org/).

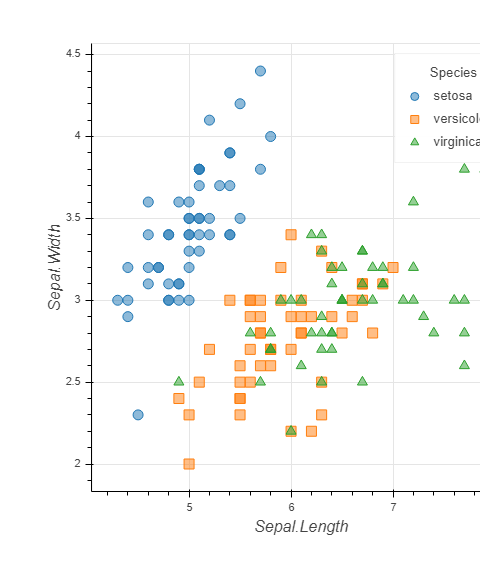
library(charter)  
  
chic$date\_num <- as.numeric(chic$date)  
## doesn't work with class date  
  
chart(data = chic, caes(date\_num, temp)) |>  
 c\_scatter(caes(color = season, group = season)) |>  
 c\_colors(RColorBrewer::brewer.pal(4, name = "Dark2"))



## 18.7 Bokeh via {rbokeh}

[{rbokeh}](https://hafen.github.io/rbokeh/) is an R package that allows you to create interactive visualizations using the [Bokeh](https://bokeh.org/) library. It is a powerful tool for creating interactive plots and adding interactivity to your visualizations. The following example demonstrates how to create an interactive scatter plot using {rbokeh}. You can find more examples and documentation on the [rbokeh website](https://hafen.github.io/rbokeh/).

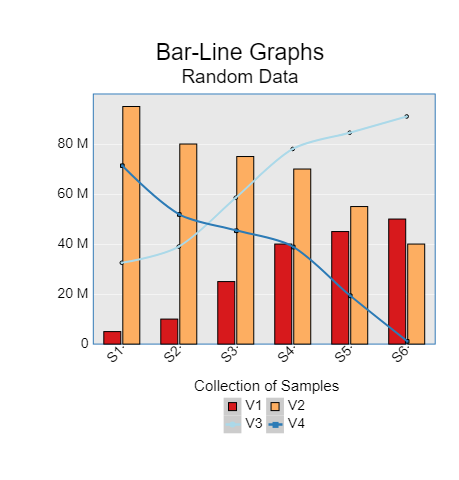
library(rbokeh)  
p <- figure() %>%  
 ly\_points(Sepal.Length, Sepal.Width, data = iris,  
 color = Species, glyph = Species,  
 hover = list(Sepal.Length, Sepal.Width))  
p



## 18.8 Advanced Interactive plots using CanvasExpress

[CanvasXpress](https://www.canvasxpress.org/) is a JavaScript library that allows you to create interactive visualizations. The package {canvasXpress} for R enables the creation of interactive plots directly from R. It is a powerful tool for creating visualizations and adding interactivity to your plots. The following example demonstrates how to create a bar-line graph using CanvasXpress. You can find more examples and documentation on the [CanvasXpress website](https://www.canvasxpress.org/).

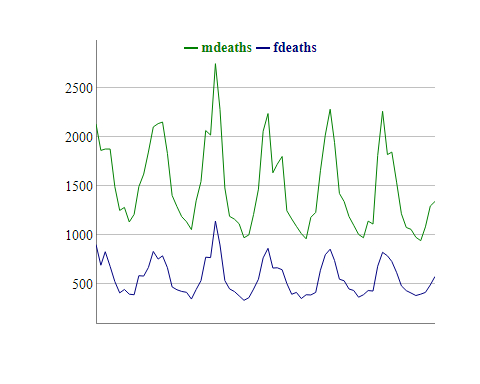
library(canvasXpress)  
y=read.table("https://www.canvasxpress.org/data/cX-generic-dat.txt", header=TRUE, sep="\t", quote="", row.names=1, fill=TRUE, check.names=FALSE, stringsAsFactors=FALSE)  
x=read.table("https://www.canvasxpress.org/data/cX-generic-smp.txt", header=TRUE, sep="\t", quote="", row.names=1, fill=TRUE, check.names=FALSE, stringsAsFactors=FALSE)  
z=read.table("https://www.canvasxpress.org/data/cX-generic-var.txt", header=TRUE, sep="\t", quote="", row.names=1, fill=TRUE, check.names=FALSE, stringsAsFactors=FALSE)  
canvasXpress(  
 data=y,  
 smpAnnot=x,  
 varAnnot=z,  
 graphOrientation="vertical",  
 graphType="BarLine",  
 legendColumns=2,  
 legendPosition="bottom",  
 lineThickness=2,  
 lineType="spline",  
 showTransition=FALSE,  
 smpLabelRotate=45,  
 smpTitle="Collection of Samples",  
 subtitle="Random Data",  
 theme="CanvasXpress",  
 title="Bar-Line Graphs",  
 xAxis=list("V1", "V2"),  
 xAxis2=list("V3", "V4"),  
 xAxis2TickFormat="%.0f T",  
 xAxisTickFormat="%.0f M"  
)



## 18.9 Dygraphs via {dygraphs}

[{dygraphs}](https://rstudio.github.io/dygraphs/) is an R package that allows you to create interactive time series plots. It is based on the JavaScript library [Dygraphs](https://dygraphs.com/).

library(dygraphs)  
lungDeaths <- cbind(mdeaths, fdeaths)  
dygraph(lungDeaths)



And there are many more options to create interactive plots in R. The choice of the right library depends on the specific requirements of your project and the desired level of interactivity. The examples above should give you a good starting point to explore the possibilities of interactive plots in R. We’ll add more examples in the future.

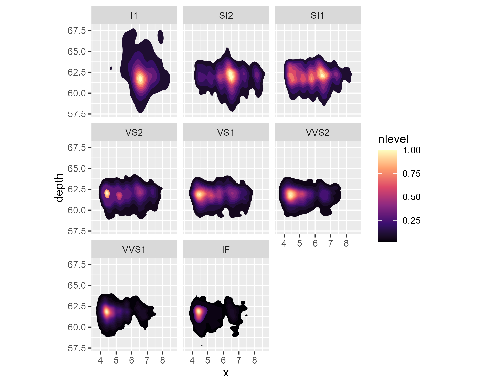
# 19. 3D Plots Using {rayshader} package

rayshader is an open source package for producing 2D and 3D data visualizations in R. rayshader uses elevation data in a base R matrix and a combination of raytracing, hillshading algorithms, and overlays to generate stunning 2D and 3D maps. In addition to maps, rayshader also allows the user to translate ggplot2 objects into beautiful 3D data visualizations.

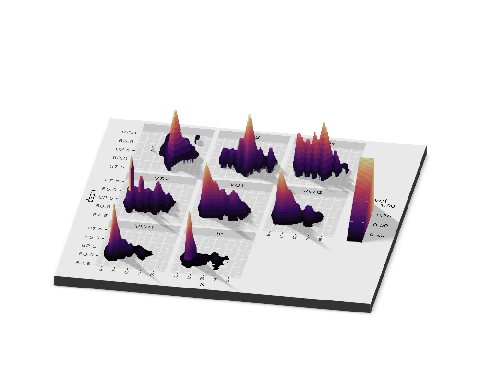
The models can be rotated and examined interactively or the camera movement can be scripted to create animations. Scenes can also be rendered using a high-quality pathtracer, rayrender. The user can also create a cinematic depth of field post-processing effect to direct the user’s focus to important regions in the figure. The 3D models can also be exported to a 3D-printable format with a built-in STL export function, and can be exported to an OBJ file for use in other 3D modeling software. rayshader is a powerful tool for creating 3D visualizations of data, and can be used to create stunning visualizations for scientific research, data analysis, and art. You can find more information about rayshader at https://www.rayshader.com/.

Let’s see some 3D plots using rayshader.

library(ggplot2)  
library(rayshader)  
  
ggdiamonds = ggplot(diamonds) +  
 stat\_density\_2d(aes(x = x, y = depth, fill = after\_stat(nlevel)),   
 geom = "polygon", n = 200, bins = 50,contour = TRUE) +  
 facet\_wrap(clarity~.) +  
 scale\_fill\_viridis\_c(option = "A")  
  
plot\_gg(ggdiamonds, width = 5, height = 5, raytrace = FALSE, preview = TRUE)



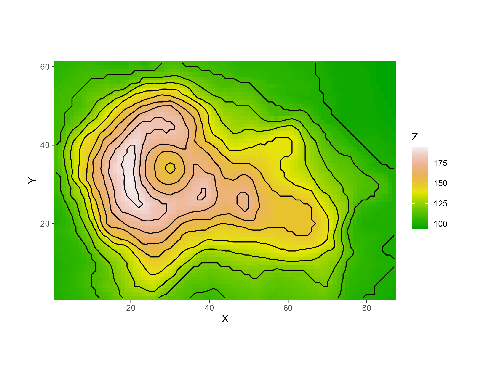
plot\_gg(ggdiamonds, width = 5, height = 5, multicore = TRUE, scale = 250,   
 zoom = 0.7, theta = 10, phi = 30, windowsize = c(800, 800))  
Sys.sleep(0.2)  
render\_snapshot(clear = TRUE)



Rayshader will automatically ignore lines and other elements that should not be mapped to 3D. Here’s a contour plot of the volcano dataset.

library(reshape2)  
#Contours and other lines will automatically be ignored. Here is the volcano dataset:  
  
ggvolcano = volcano %>%   
 melt() %>%  
 ggplot() +  
 geom\_tile(aes(x = Var1, y = Var2, fill = value)) +  
 geom\_contour(aes(x = Var1, y = Var2, z = value), color = "black") +  
 scale\_x\_continuous("X", expand = c(0, 0)) +  
 scale\_y\_continuous("Y", expand = c(0, 0)) +  
 scale\_fill\_gradientn("Z", colours = terrain.colors(10)) +  
 coord\_fixed()  
  
par(mfrow = c(1, 2))  
plot\_gg(ggvolcano, width = 7, height = 4, raytrace = FALSE, preview = TRUE)

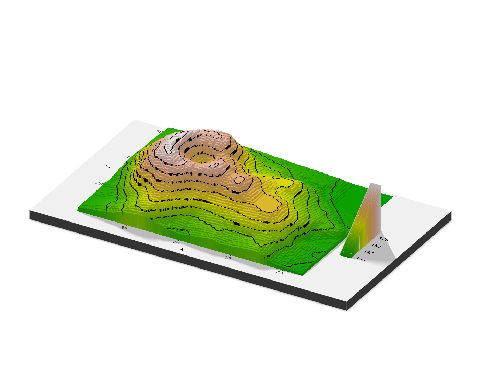
Warning: Removed 1861 rows containing missing values or values outside the scale range  
(`geom\_contour()`).



plot\_gg(ggvolcano, multicore = TRUE, raytrace = TRUE, width = 7, height = 4,   
 scale = 300, windowsize = c(1400, 866), zoom = 0.6, phi = 30, theta = 30)

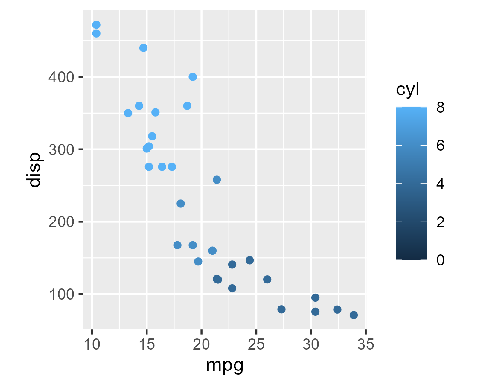
Warning: Removed 1861 rows containing missing values or values outside the scale range  
(`geom\_contour()`).

Sys.sleep(0.2)  
render\_snapshot(clear = TRUE)

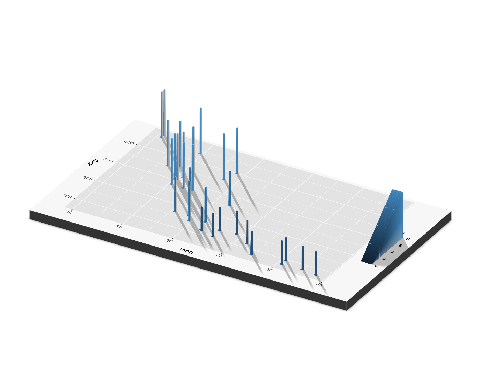


Rayshader also detects when the user passes the color aesthetic, and maps those values to 3D. If both color and fill are passed, however, rayshader will default to fill.

mtplot = ggplot(mtcars) +   
 geom\_point(aes(x = mpg, y = disp, color = cyl)) +   
 scale\_color\_continuous(limits = c(0, 8))  
  
par(mfrow = c(1, 2))  
plot\_gg(mtplot, width = 3.5, raytrace = FALSE, preview = TRUE)



plot\_gg(mtplot, multicore = TRUE, raytrace = TRUE, width = 7, height = 4,   
 scale = 300, windowsize = c(1400, 866), zoom = 0.6, phi = 30, theta = 30)  
Sys.sleep(0.2)  
render\_snapshot(clear = TRUE)

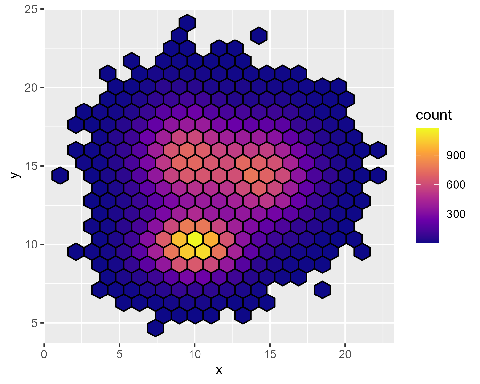


Utilize combinations of line color and fill to create different effects. Here is a terraced hexbin plot, created by mapping the line colors of the hexagons to black.

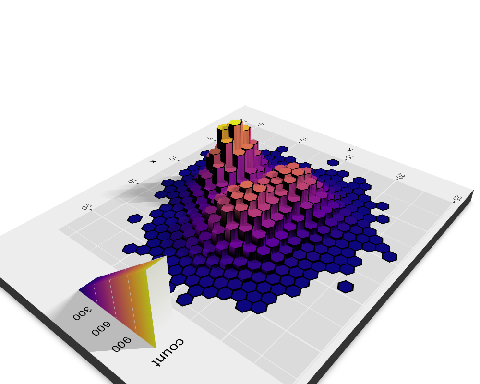
a = data.frame(x = rnorm(20000, 10, 1.9), y = rnorm(20000, 10, 1.2))  
b = data.frame(x = rnorm(20000, 14.5, 1.9), y = rnorm(20000, 14.5, 1.9))  
c = data.frame(x = rnorm(20000, 9.5, 1.9), y = rnorm(20000, 15.5, 1.9))  
data = rbind(a, b, c)  
  
#Lines  
library(hexbin)  
pp = ggplot(data, aes(x = x, y = y)) +  
 geom\_hex(bins = 20, size = 0.5, color = "black") +  
 scale\_fill\_viridis\_c(option = "C")

Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.  
ℹ Please use `linewidth` instead.

par(mfrow = c(1, 2))  
plot\_gg(pp, width = 5, height = 4, scale = 300, raytrace = FALSE, preview = TRUE)



plot\_gg(pp, width = 5, height = 4, scale = 300, multicore = TRUE, windowsize = c(1000, 800))  
render\_camera(fov = 70, zoom = 0.5, theta = 130, phi = 35)  
Sys.sleep(0.2)  
render\_snapshot(clear = TRUE)



Pretty cool, right? You can create stunning 3D visualizations using rayshader. You can find more information about rayshader at https://www.rayshader.com/.

We can also Use rayshader to create 3D maps. That will be whole another book, we’ll publish that soon. Stay tuned!

# 20. Geographical Data Analysis using {sf} and

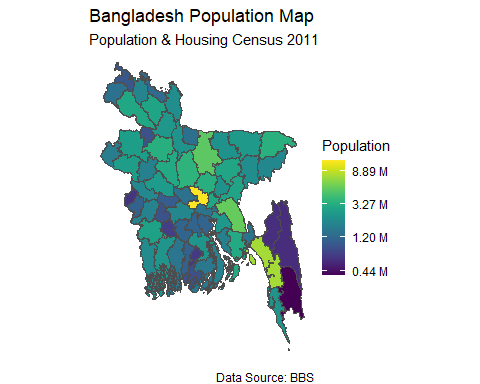
R has well-supported classes for storing spatial data (sp) and interfacing to the above mentioned environments (rgdal, rgeos), but has so far lacked a complete implementation of simple features, making conversions at times convoluted, inefficient or incomplete. The package sf tries to fill this gap, and aims at succeeding sp in the long term. However This is a Huge topic to cover that we need a separate book for this.

We’ll just give you a brief overview of how to plot maps using sf and ggplot2.

# Load necessary libraries  
library(bangladesh)  
library(ggplot2)  
library(tidyverse)

── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──  
✔ dplyr 1.1.4 ✔ readr 2.1.5  
✔ forcats 1.0.0 ✔ stringr 1.5.1  
✔ lubridate 1.9.3 ✔ tibble 3.2.1  
✔ purrr 1.0.2 ✔ tidyr 1.3.1  
── Conflicts ────────────────────────────────────────── tidyverse\_conflicts() ──  
✖ dplyr::filter() masks stats::filter()  
✖ dplyr::lag() masks stats::lag()  
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors

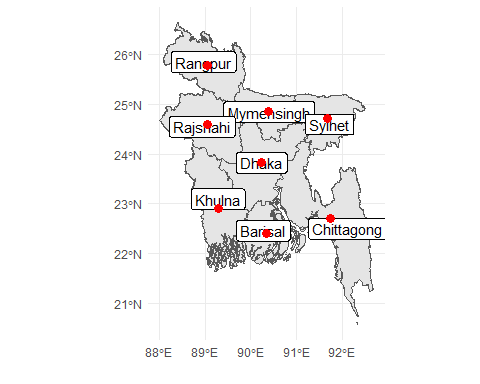
# Get map data, join with population data, and plot in a single pipeline  
data <- get\_map("district") %>%  
 left\_join(bangladesh::pop\_district\_2011[, c("district", "population")], by = c("District" = "district"))  
  
pp <- data %>%   
 ggplot() +  
 geom\_sf(aes(fill = population), col = "grey30") +  
 theme\_void() +  
 viridis::scale\_fill\_viridis(trans = "log", name="Population", labels = scales::unit\_format(unit = "M", scale = 1e-6)) +  
 labs(  
 title = "Bangladesh Population Map",  
 subtitle = "Population & Housing Census 2011",  
 caption = "Data Source: BBS"  
 )  
  
pp



Here is another example:

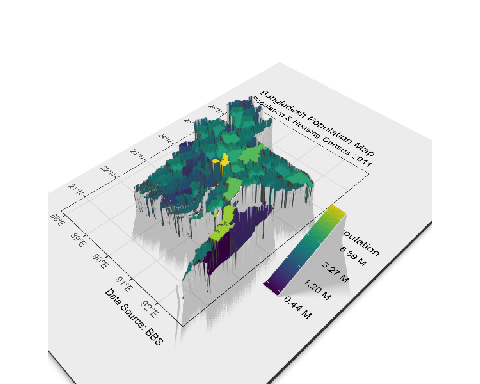
division\_data <- get\_map("division")  
division\_centroids <- bangladesh::get\_coordinates(level = "division")  
ggplot(data = division\_data) +  
 geom\_sf() +  
 geom\_sf\_label(aes(label = Division)) +  
 geom\_point(data = division\_centroids, x = division\_centroids$lon, y = division\_centroids$lat, col = "red", size = 3) +  
 xlab("")+ ylab("")+  
 theme\_minimal()

Warning in st\_point\_on\_surface.sfc(sf::st\_zm(x)): st\_point\_on\_surface may not  
give correct results for longitude/latitude data



You can also make 3D map using rayshader sf and ggplot2 package. Here is an example:

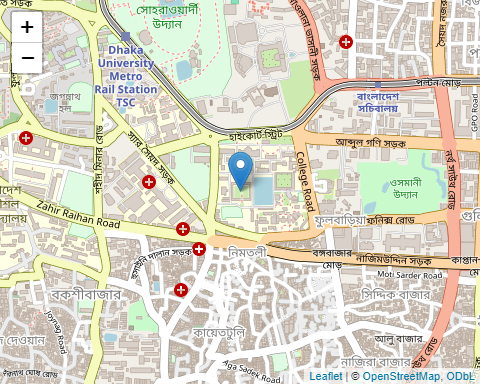
library(rayshader)  
  
plot\_gg(pp+theme\_bw(), multicore = TRUE, width = 4 ,height=6, fov = 70, zoom = 0.5)  
Sys.sleep(0.2)  
render\_snapshot(clear = TRUE)



If You Want to make maps Interactive like this 👉 [Interactive Maps](https://rana2hin.quarto.pub/mapsinbooks/), here is an example:

library(leaflet)  
  
leaflet() %>%  
 addTiles() %>%   
 addMarkers(lng=90.40155705289271, lat=23.725810762885487, popup="Shahidullah Hall")

PhantomJS not found. You can install it with webshot::install\_phantomjs(). If it is installed, please make sure the phantomjs executable can be found via the PATH variable.



The field of geographical data analysis is a vast and ever-evolving domain, offering an array of techniques and tools to explore and understand spatial data. In this book, we have provided a glimpse into the fascinating world of Data Visualization using ggplot2 and introducing you to the fundamental concepts and methods that form the foundation of spatial analysis.

While we have covered a range of topics, it is important to recognize that this is merely the tip of the iceberg. Geographical data analysis encompasses a multitude of specialized areas, each with its own unique challenges and solutions. As we continue our journey through this field, we are working on a dedicated book that will delve deeper into the intricacies of geographic data analysis.

In this forthcoming work, we will explore in greater depth the realm of shapefiles, a crucial data format for representing geographical features. Additionally, we will dive into the powerful capabilities of the sf package, which provides a comprehensive set of tools for working with spatial vector data in R.

Furthermore, we will introduce you to leaflet, a cutting-edge library that enables the creation of interactive web maps, allowing you to visualize and analyze spatial data in a dynamic and engaging manner. The rayshader package will also be explored, offering techniques for generating stunning 3D visualizations of geographical data, providing new perspectives and insights.

Another exciting area we will cover is the tmap package, a versatile tool for creating thematic maps and visualizing spatial patterns. With its rich set of features and extensive customization options, tmap empowers you to communicate your spatial data in a clear and compelling way.

Beyond these specific packages and techniques, we are actively engaged in the development of new and innovative tools for geographical data analysis. Our goal is to push the boundaries of what is possible, providing researchers, analysts, and practitioners with cutting-edge solutions to tackle complex spatial problems.

As we continue to explore the depths of this fascinating field, we invite you to stay tuned for our upcoming work. Together, we will embark on a journey of discovery, unlocking the full potential of geographical data analysis and shaping the future of spatial research and applications.

# Remarks, Tipps & Resources

## Using ggplot2 in Loops and Functions

The grid-based graphics functions in lattice and ggplot2 create a graph object. When you use these functions interactively at the command line, the result is automatically printed. However, when using source() or inside your own functions, you will need an explicit print() statement, i.e., print(g) in most of our examples. For more information, see also the [R FAQ page](https://cran.r-project.org/doc/FAQ/R-FAQ.html#Why-do-lattice_002ftrellis-graphics-not-work_003f).

## Additional Resources

* [“ggplot2: Elegant Graphics for Data Analysis”](https://ggplot2-book.org/) by Hadley Wickham, available via open-access!
* [“Fundamentals of Data Visualization”](http://serialmentor.com/dataviz/) by Claus O. Wilke about data visualization in general but using {ggplot2}. (You can find the codes on [his GitHub profile](https://github.com/clauswilke/dataviz).)
* [“Cookbook for R”](http://www.cookbook-r.com/Graphs/) by Winston Chang with recipes to produce R plots
* Gallery of the [Top 50 ggplot2 visualizations](http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html)
* Gallery of [{ggplot2} extension packages](https://exts.ggplot2.tidyverse.org/gallery/)
* [How to extend {ggplot2}](https://cran.r-project.org/web/packages/ggplot2/vignettes/extending-ggplot2.html) by Hadley Wickham
* The fantastic [R4DS Online Learning Community](https://www.rfordatasci.com/) that offers help and mentoring for all things related to the content of the [“R for Data Science” book](r4ds.had.co.nz/) by Hadley Wickham
* [#TidyTuesday](https://github.com/rfordatascience/tidytuesday), a weekly social data project focusing on ggplots—check also [#TidyTuesday on Twitter](https://twitter.com/hashtag/TidyTuesday?lang=en) and [this collection of contributions by Neil Grantham](https://nsgrantham.com/tidytuesdayrocks/)
* A two-part, 4.5-hours tutorial series by Thomas Linn Pedersen ([Part 1](https://www.youtube.com/watch?v=h29g21z0a68) | [Part 2](https://www.youtube.com/watch?v=0m4yywqNPVY))