DATAVIZ (/TAGS/DATAVIZ) TUTORIAL (/TAGS/TUTORIAL) R (/TAGS/R) TIDYVERSE (/TAGS/TIDYVERSE) GGPLOT2 (/TAGS/GGPLOT2)

A GGPLOT2 TUTORIAL FOR BEAUTIFUL PLOTTING IN R

POSTED BY CÉDRIC ON MONDAY, AUGUST 5, 2019

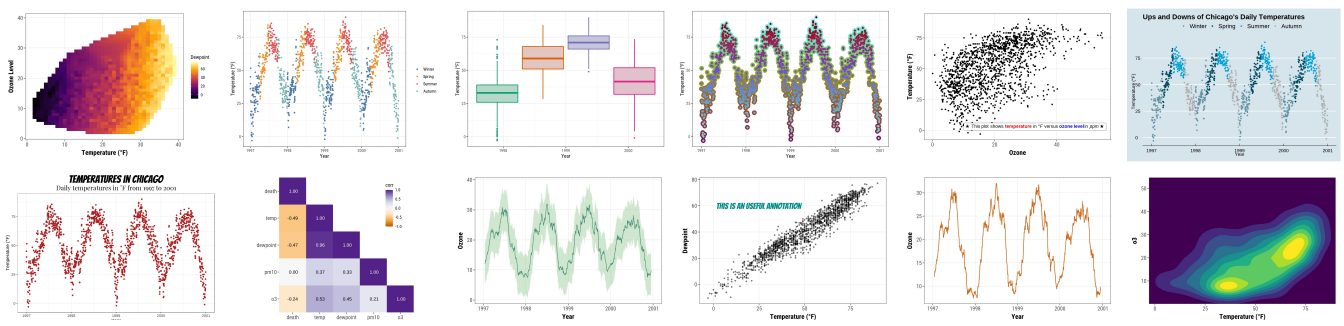
Last update: 2021-02-09

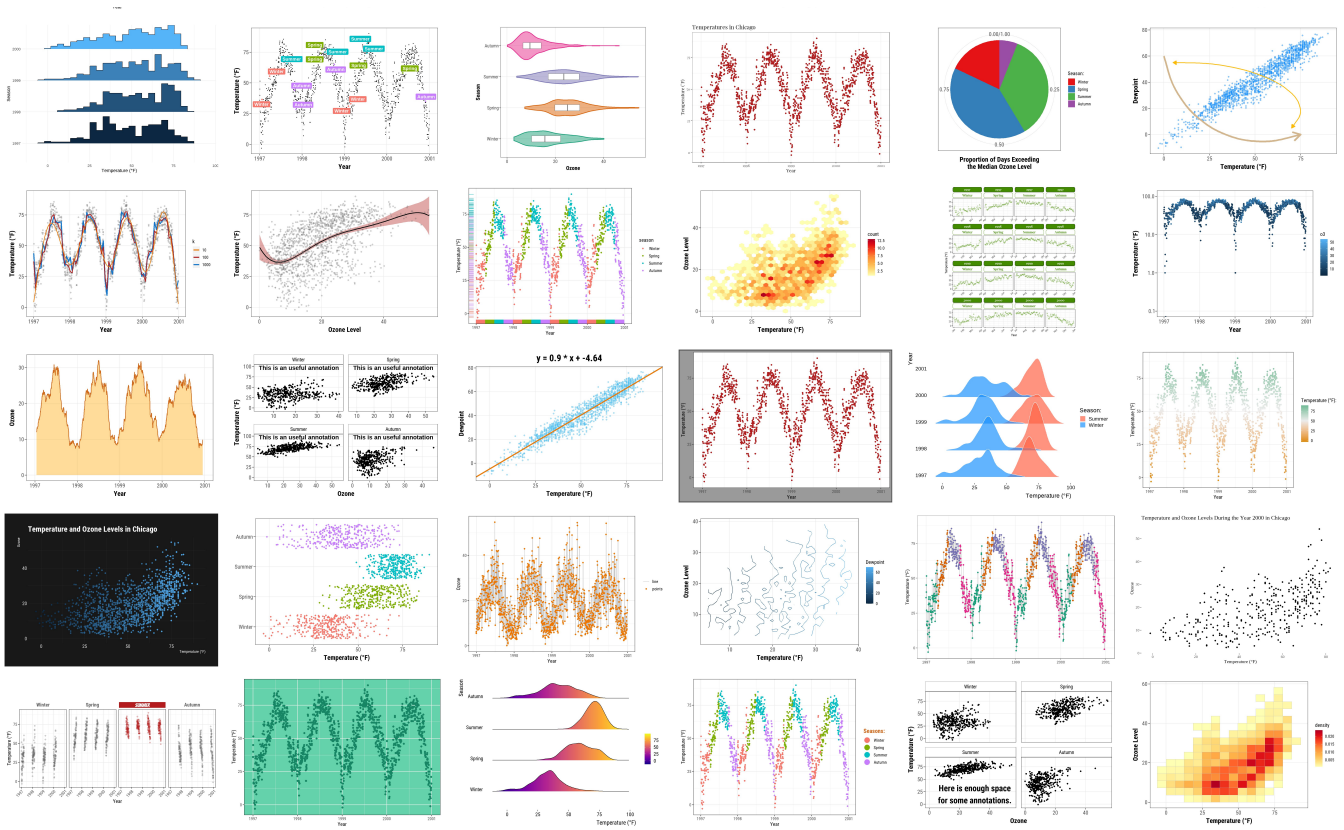
INTRODUCTORY WORDS

I don't care, just show me the content!

Back in 2016, I had to prepare my PhD introductory talk and I started using `{ggplot2}` to visualize my data. I never liked the syntax and style of base plots in R, so I was quickly in love with ggplot. Especially useful was its faceting utility. But because I was short on time, I plotted these figures by trial and error and with the help of lots of googling. The resource I came always back to was a blog entry called **Beautiful plotting in R: A ggplot2 cheatsheet** (http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/) by Zev Ross (https://twitter.com/zevross), updated last in January 2016. After giving the talk which contained some decent plots thanks to the blog post, I decided to go through this tutorial step-by-step. I learned so much from it and directly started modifying the codes and over the time I added additional code snippets, chart types and resources.

Since the blog entry by Zev Ross was not updated for some years and step by step this became a unique version of a tutorial, I decided to host the updated version on my GitHub. Now it finds its proper place on this homepage! (Plus I added a ton of other updates—just to name a few: The fantastic `{patchwork}`, `{ggtext}` and `{ggforce}` packages. How to deal with custom fonts and colors. A collection of R packages tailored to create interactive charts. And several other chart types including pie charts because everyone looooves pie charts!)





Some exemplary plots included in this tutorial.

Major changes I've made:

- to follow the R style guide (e.g. by Hadley Wickham (http://adv-r.had.co.nz/Style.html), Google (https://google.github.io/styleguide/Rguide.xml) or the Coding Club (https://ourcodingclub.github.io/2017/04/25/etiquette.html#syntax) style guides),
- to change style and aesthetics of plots (e.g. axis titles, legends and nice colors for all plots not only some),
- to have a updated version which keeps track of changes in `{ggplot2}` (current version: 3.3.2),
- to modify data import (GitHub source),
- to add additional tips on a vast range of topics, including for example chart choice, color palettes, modifying titles, adding lines, modifying legends, annotations with labels, arrows and boxes, multi-panel plots, interactive visualizations, ...

TABLE OF CONTENT

- Working with Titles
- Working with Legends
- Working with Backgrounds & Grid Lines
- Working with Margins
- Working with Multi-Panel Plots
- Working with Colors
- Working with Themes
- Working with Lines
- Working with Text
- Working with Coordinates
- Working with Chart Types
- Working with Ribbons (AUC, CI, etc.)
- Working with Smoothings
- Working with Interactive Plots
- Remarks, Tipps & Resources

PREPARATION

- You can find the Rmarkdown script with the code executed in this blogpost here (https://github.com/Z3tt/Z3tt/blob/master/content/post/2019-08-05_ggplot2-tutorial.Rmd).
- You can also download the R script containing only the code here (https://cedricscherer.netlify.app/codes/ggplot-tutorial-cedric-raw.R).
- You need to install the following packages to execute the full tutorial:
  - `{ggplot2}`, part of the `{tidyverse}` package collection
  - `{tidyverse}` package collection, namely
    - `{dplyr}` for data wrangling
    - `{tibble}` for modern data frames
    - `{tidyr}` for data cleaning
    - `{forcats}` for handling factors
  - `{colorspace}` for manipulating colors
  - `{corrr}` for calculating correlation matrices
  - `{cowplot}` for composing ggplots
  - `{ggdark}` for themes and inverting colors
  - `{ggforce}` for sina plots and other cool stuff
  - `{ggrepel}` for nice text labeling
  - `{ggridges}` for ridge plots
  - `{ggsci}` for nice color palettes
  - `{ggtext}` for advanced text rendering
  - `{ggthemes}` for additional themes

- `{grid}` for creating graphical objects
- `{gridExtra}` for additional functions for "grid" graphics
- `{patchwork}` for multi-panel plots
- `{rcartocolor}` for great color palettes
- `{scico}` for perceptional uniform palettes
- `{showtext}` for custom fonts
- `{shiny}` for interactive apps
- a number of packages for interactive visualizations
  - `{charter}`
  - `{echarts4r}`
  - `{ggiraph}`
  - `{highcharter}`
  - `{plotly}`

```
# install CRAN packages
install.packages(c("tidyverse", "colorspace", "corrr",  "cowplot",
                   "ggdark", "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 teaching reasons and if people jump to any plot, I load the package needed beside `{ggplot2}` in the respective section.)

THE DATASET

We are using data from the *National Morbidity and Mortality Air Pollution Study* (NMMAPS). To make the plots manageable we are limiting the data to Chicago and 1997–2000. For more detail on this data set, consult Roger Peng's book Statistical Methods in Environmental Epidemiology with R (http://www.springer.com/de/book/9780387781662). You can download the data we are using during this tutorial here (https://github.com/Z3tt/R-Tutorials/blob/master/ggplot2/chicago-nmmaps.csv) (but you don't have to).

We can import the data into our R session for example with `read_csv()` from the `{readr}` package. To access the data later, we are storing it in a variable called `chic` by using the *assignment arrow* `<-`.

```
chic <- readr::read_csv("https://raw.githubusercontent.com/Z3tt/R-Tutorials/master/ggplot2,
```

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

```
tibble::glimpse(chic)
```

```
## Rows: 1,461
## Columns: 10
## $ city     <chr> "chic", "chic", "chic", "chic", "chic", "chic", "chic", "chic", "chi
## $ date     <date> 1997-01-01, 1997-01-02, 1997-01-03, 1997-01-04, 1997-01-05, 1997-01
## $ death    <dbl> 137, 123, 127, 146, 102, 127, 116, 118, 148, 121, 110, 127, 129, 151
## $ temp     <dbl> 36.0, 45.0, 40.0, 51.5, 27.0, 17.0, 16.0, 19.0, 26.0, 16.0, 1.5, 1.0
## $ dewpoint <dbl> 37.50000, 47.25000, 38.00000, 45.50000, 11.25000, 5.75000, 7.00000,
## $ pm10     <dbl> 13.052268, 41.948600, 27.041751, 25.072573, 15.343121, 9.364655, 20.
## $ o3       <dbl> 5.659256, 5.525417, 6.288548, 7.537758, 20.760798, 14.940874, 11.920
## $ time     <dbl> 3654, 3655, 3656, 3657, 3658, 3659, 3660, 3661, 3662, 3663, 3664, 36
## $ season   <chr> "Winter", "Winter", "Winter", "Winter", "Winter", "Winter", "Winter"
## $ year     <dbl> 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 1997, 19
```

```
head(chic, 10)
```

```
## # A tibble: 10 x 10
##    city  date       death  temp dewpoint  pm10    o3  time season  year
##    <chr> <date>     <dbl> <dbl>    <dbl> <dbl> <dbl> <dbl> <chr>  <dbl>
##  1 chic  1997-01-01   137  36      37.5  13.1   5.66  3654 Winter  1997
##  2 chic  1997-01-02   123  45      47.2  41.9   5.53  3655 Winter  1997
##  3 chic  1997-01-03   127  40      38    27.0   6.29  3656 Winter  1997
##  4 chic  1997-01-04   146  51.5    45.5  25.1   7.54  3657 Winter  1997
##  5 chic  1997-01-05   102  27      11.2  15.3  20.8   3658 Winter  1997
##  6 chic  1997-01-06   127  17       5.75  9.36 14.9   3659 Winter  1997
##  7 chic  1997-01-07   116  16       7    20.2  11.9   3660 Winter  1997
##  8 chic  1997-01-08   118  19      17.8  33.1   8.68  3661 Winter  1997
##  9 chic  1997-01-09   148  26      24    12.1  13.4   3662 Winter  1997
## 10 chic  1997-01-10   121  16       5.38 24.8  10.4   3663 Winter  1997
```

THE `{ggplot2}` PACKAGE

> *`ggplot2` is a system for declaratively creating graphics, based on The Grammar of Graphics (https://www.amazon.com/Grammar-Graphics-Statistics-Computing/dp/0387245448/ref=as_li_ss_tl?ie=UTF8&qid=1477928463&sr=8-1&keywords=the+grammar+of+graphics&linkCode=sl1&tag=ggplot2-20&linkId=f0130e557161b83fbe97ba0e9175c431). You provide the data, tell `ggplot2` how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.*

A ggplot is built up from a few basic elements:

1. **Data**: The raw data that you want to plot.

2. **Geometries** `geom_` : The geometric shapes that will represent the data.
3. **Aesthetics** `aes()` : Aesthetics of the geometric and statistical objects, such as position, color, size, shape, and transparency
4. **Scales** `scale_` : Maps between the data and the aesthetic dimensions, such as data range to plot width or factor values to colors.
5. **Statistical transformations** `stat_` : Statistical summaries of the data, such as quantiles, fitted curves, and sums.
6. **Coordinate system** `coord_` : The transformation used for mapping data coordinates into the plane of the data rectangle.
7. **Facets** `facet_` : The arrangement of the data into a grid of plots.
8. **Visual themes** `theme()` : The overall visual defaults of a plot, such as background, grids, axes, default typeface, sizes and colors.

💡 **The number of elements may vary depending on how you group them and whom you ask.**

A DEFAULT GGPLOT

First, to be able to use the functionality of `{ggplot2}` we have to load the package (which we can also load via the tidyverse package collection (https://www.tidyverse.org/)):

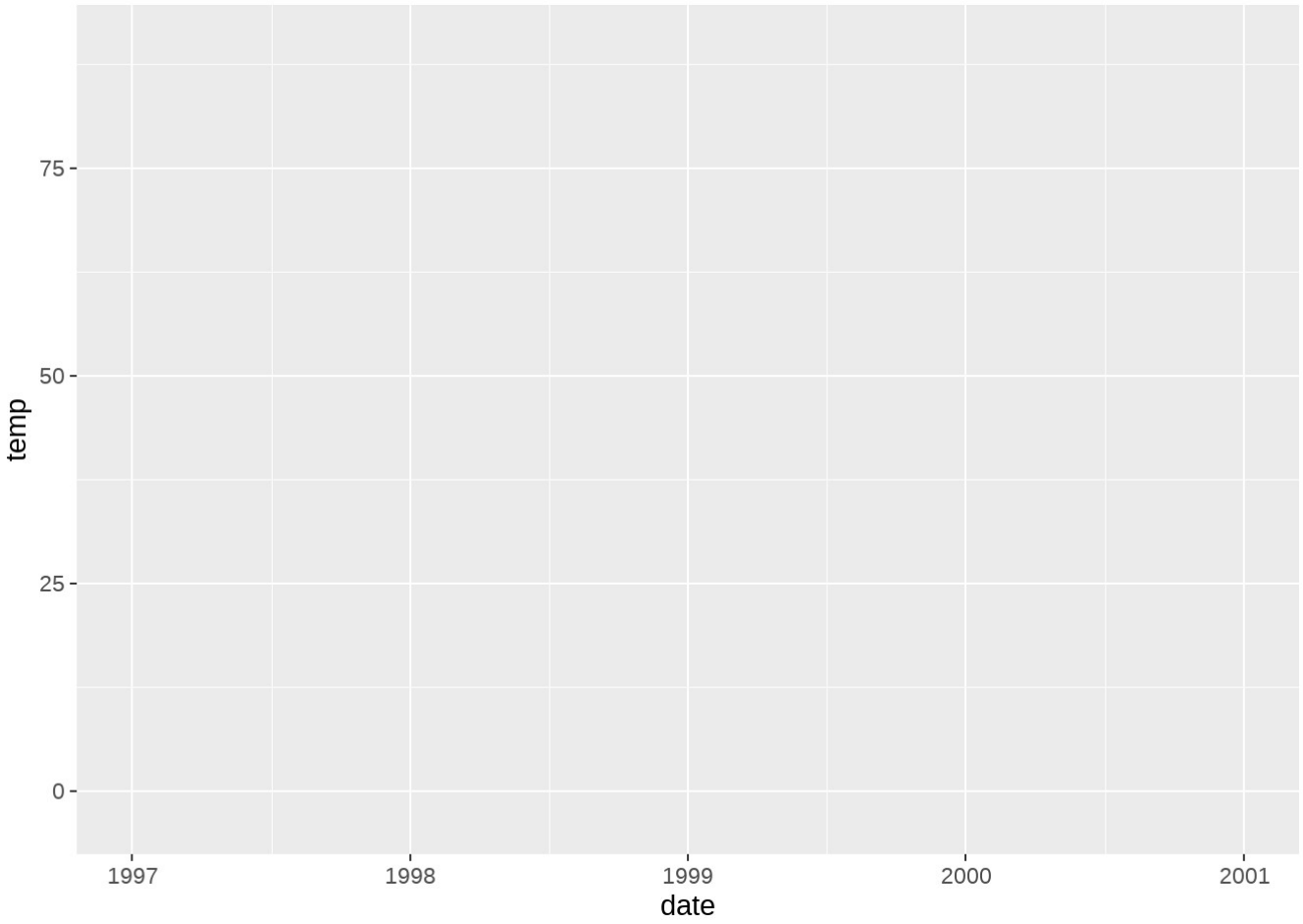
```
#library(ggplot2)
library(tidyverse)
```

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

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

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

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



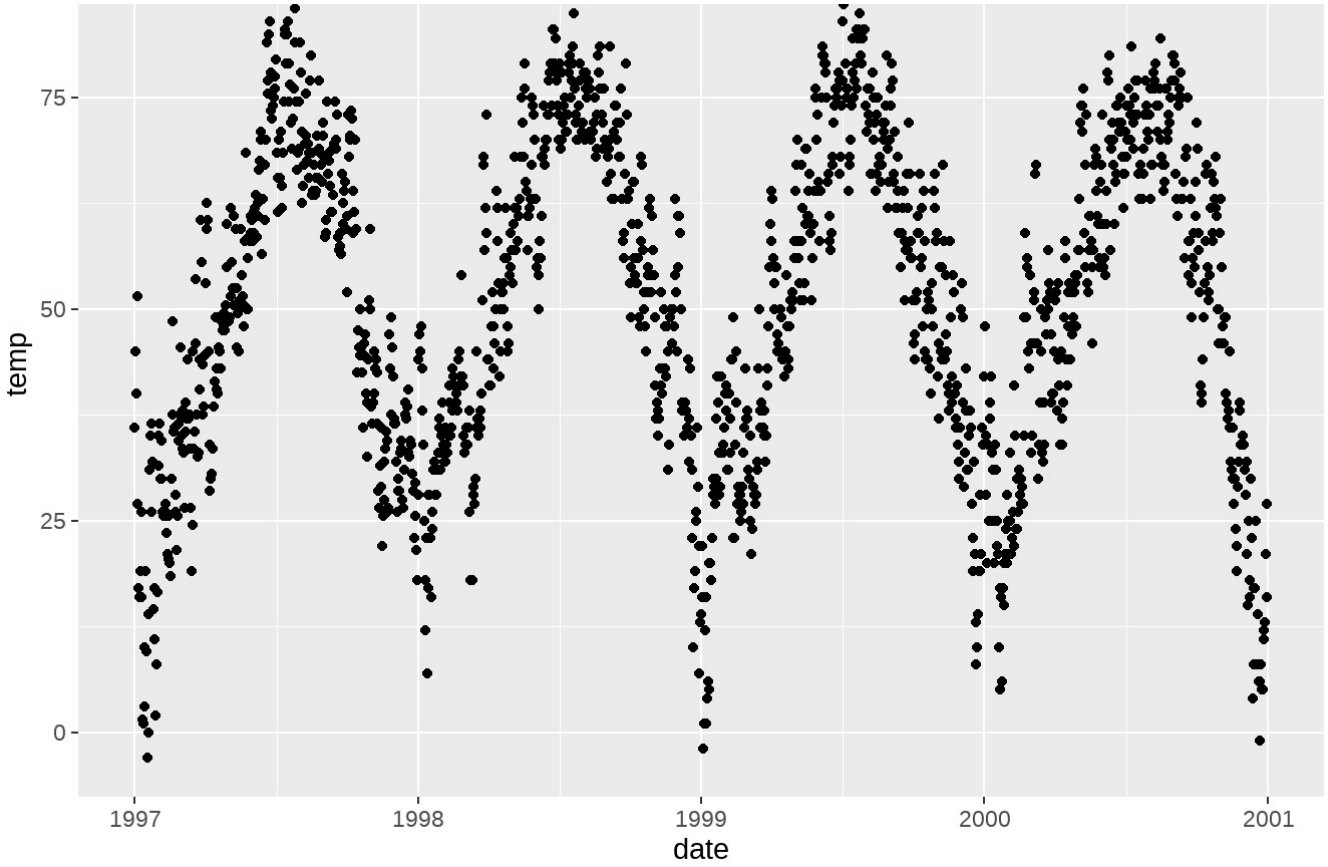
Hm, only a panel is created when running this. Why? This is because `{ggplot2}` does not know *how* we want to plot that data—we still need to provide a geometry!

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

💡 **By using parentheses while assigning an object, the object will be printed immediately (instead of writing `g <- ggplot(...)` and then `g` we simply write `(g <- ggplot(...))`).**

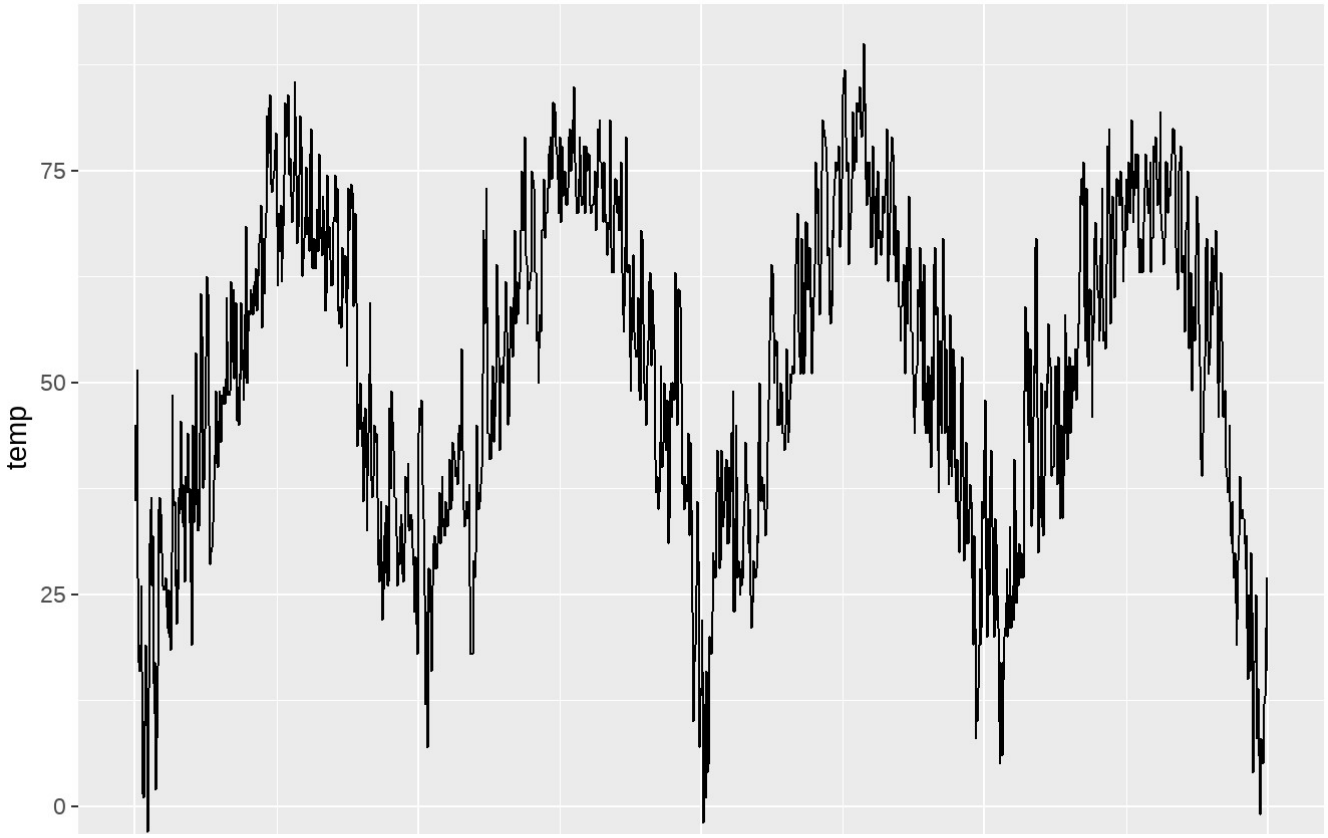
There are many, many different geometries (called *geoms* because each function usually starts with `geom_`) one can add to a ggplot by default (see here (https://ggplot2.tidyverse.org/reference/) for a full list) and even more provided by extension packages (see here (https://exts.ggplot2.tidyverse.org/) for a collection of extension packages). Let's tell `{ggplot2}` which style we want to use, for example by adding `geom_point()` to create a scatter plot:

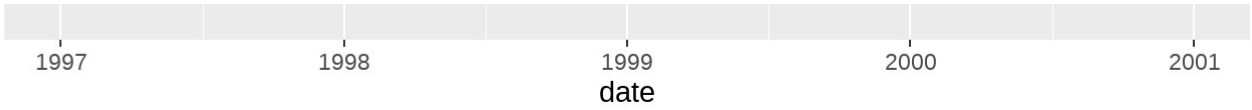
```
g + geom_point()
```



Nice! But this data could be also visualized as a line plot (not optimal, but people do things like this all the time). So we simply add `geom_line()` instead and voilá:

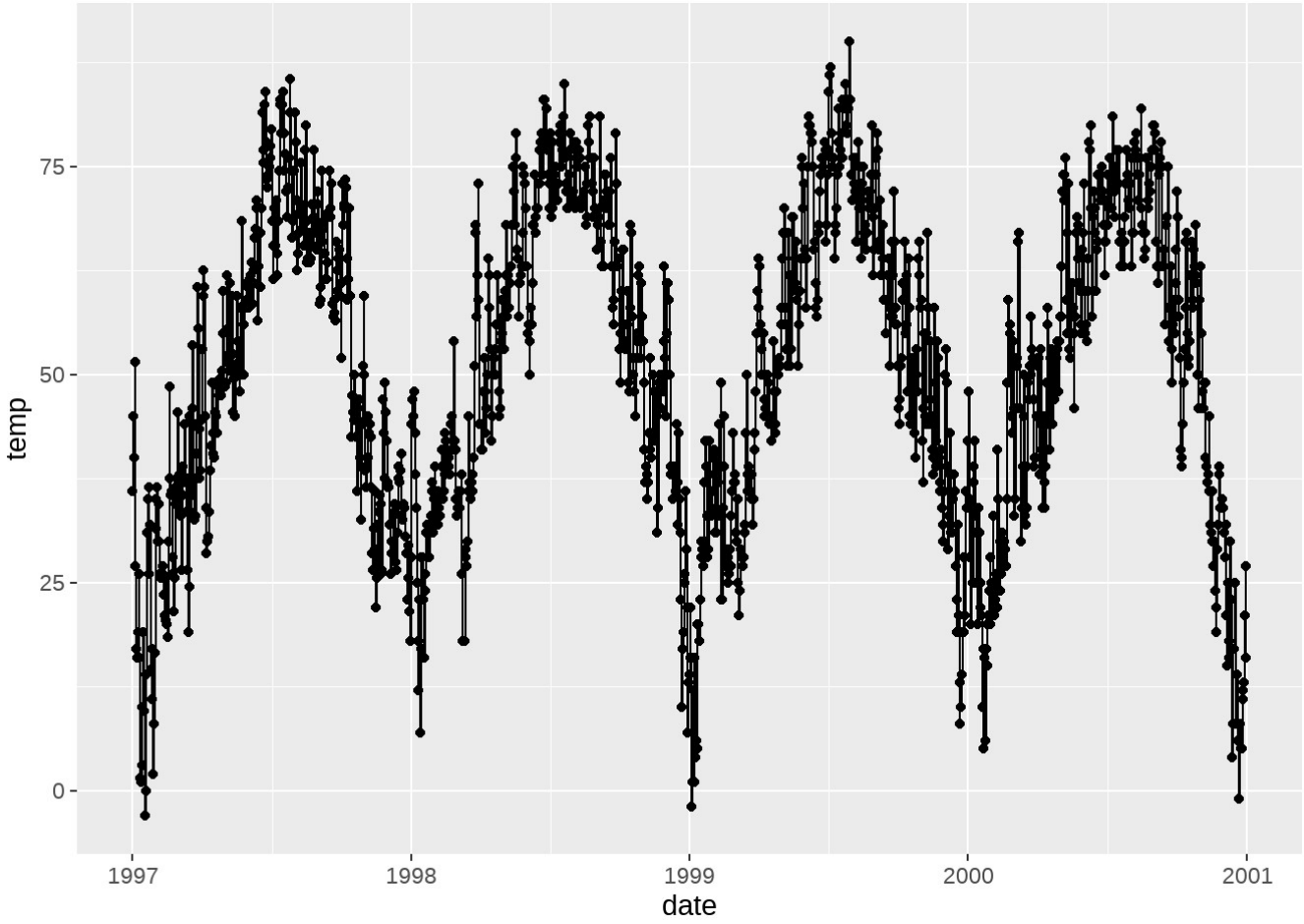
```
g + geom_line()
```





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

```
g + geom_line() + geom_point()
```

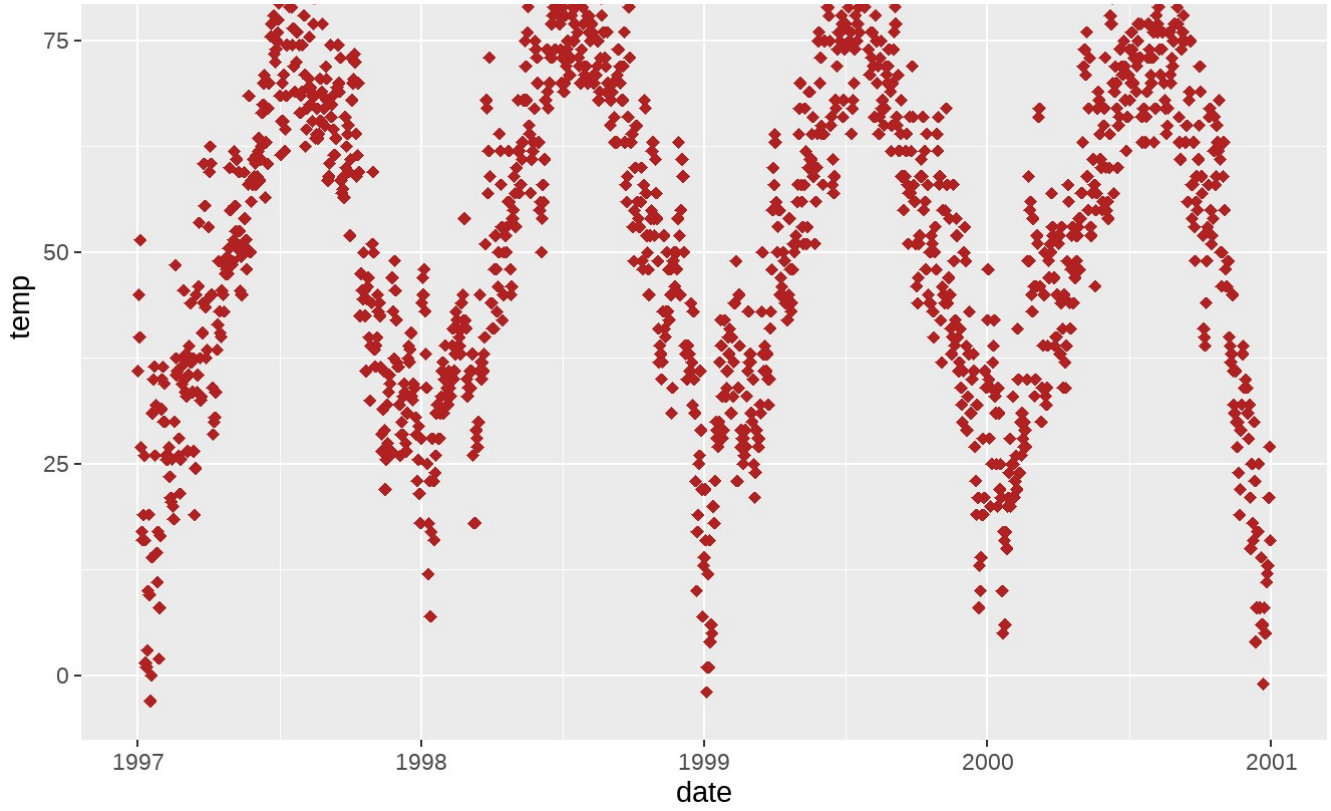


That's it for now about geometries. No worries, we are going to learn several plot types at a later point.

CHANGE PROPERTIES OF GEOMETRIES

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

```
g + geom_point(color = "firebrick", shape = "diamond", size = 2)
```

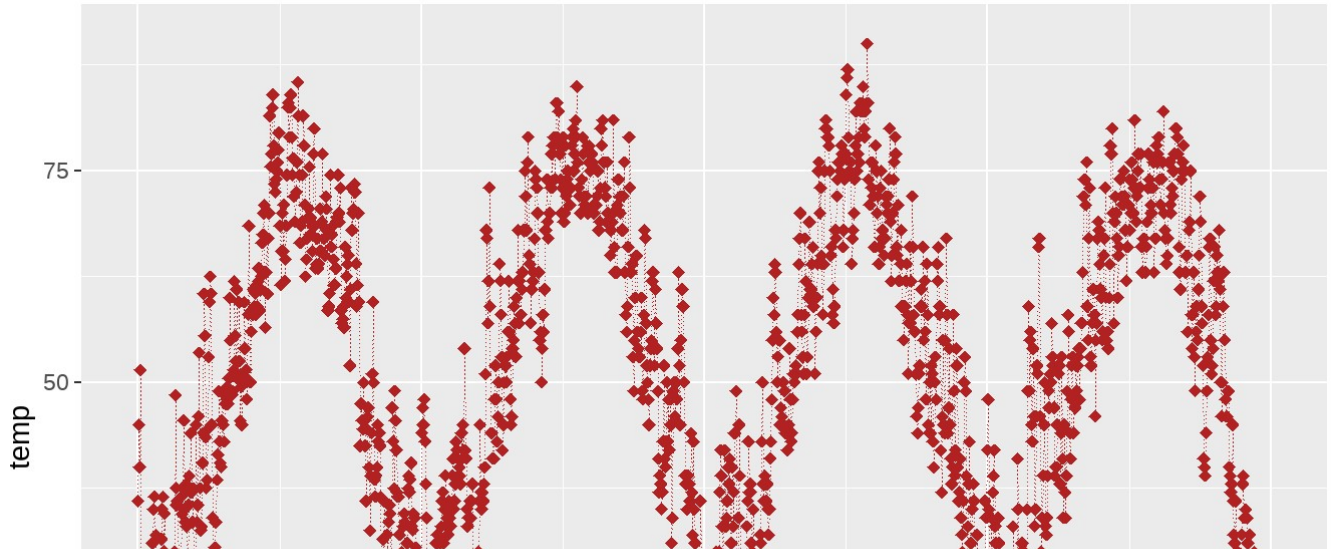


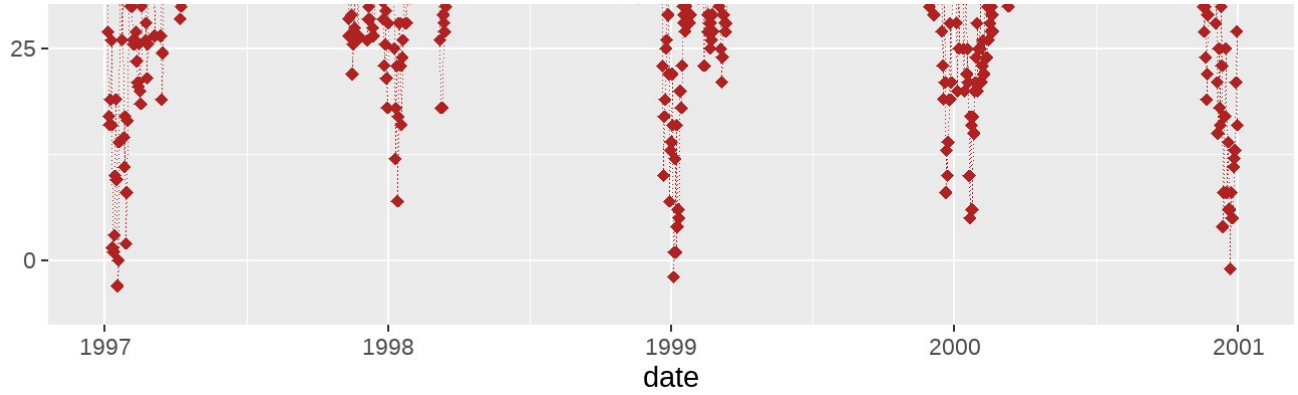
💡 **`{ggplot2}` understands both `color` and `colour` as well as the short version `col`.**

🤷 *You can use preset colors (here is a full list (http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf)) or hex color codes (https://www.techopedia.com/definition/29788/color-hex-code), both in quotes, and even RGB/RGBA colors by using the `rgb()` function. Expand to see example.*

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

```
g + geom_point(color = "firebrick", shape = "diamond", size = 2) +
    geom_line(color = "firebrick", linetype = "dotted", size = .3)
```

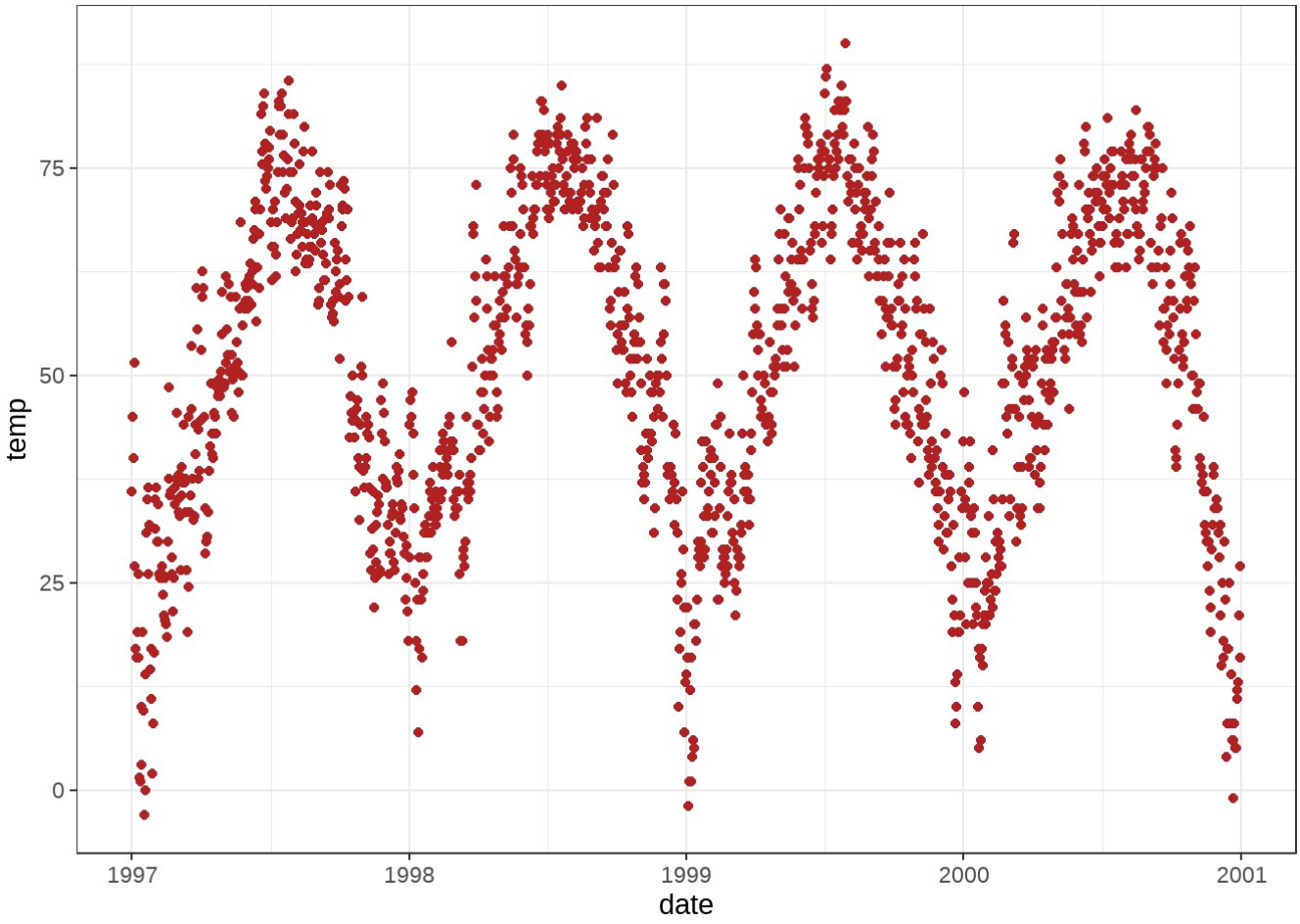




REPLACE THE DEFAULT `ggplot2` THEME

And to illustrate some more of ggplot's versatility, let's get rid of the grayish default `{ggplot2}` look by setting a different built-in theme, e.g. `theme_bw()` —by calling `theme_set()` all following plots will have the same black'n'white theme. The red points look way better now!

```
theme_set(theme_bw())

g + geom_point(color = "firebrick")
```



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

customize particular elements of the theme.

💡 **`theme()` is an essential command to manually modify all kinds of theme elements (texts, rectangles, and lines).**

To see which details of a ggplot theme can be modified have a look here (https://ggplot2.tidyverse.org/reference/theme.html)—and take some time, this is a looong list.

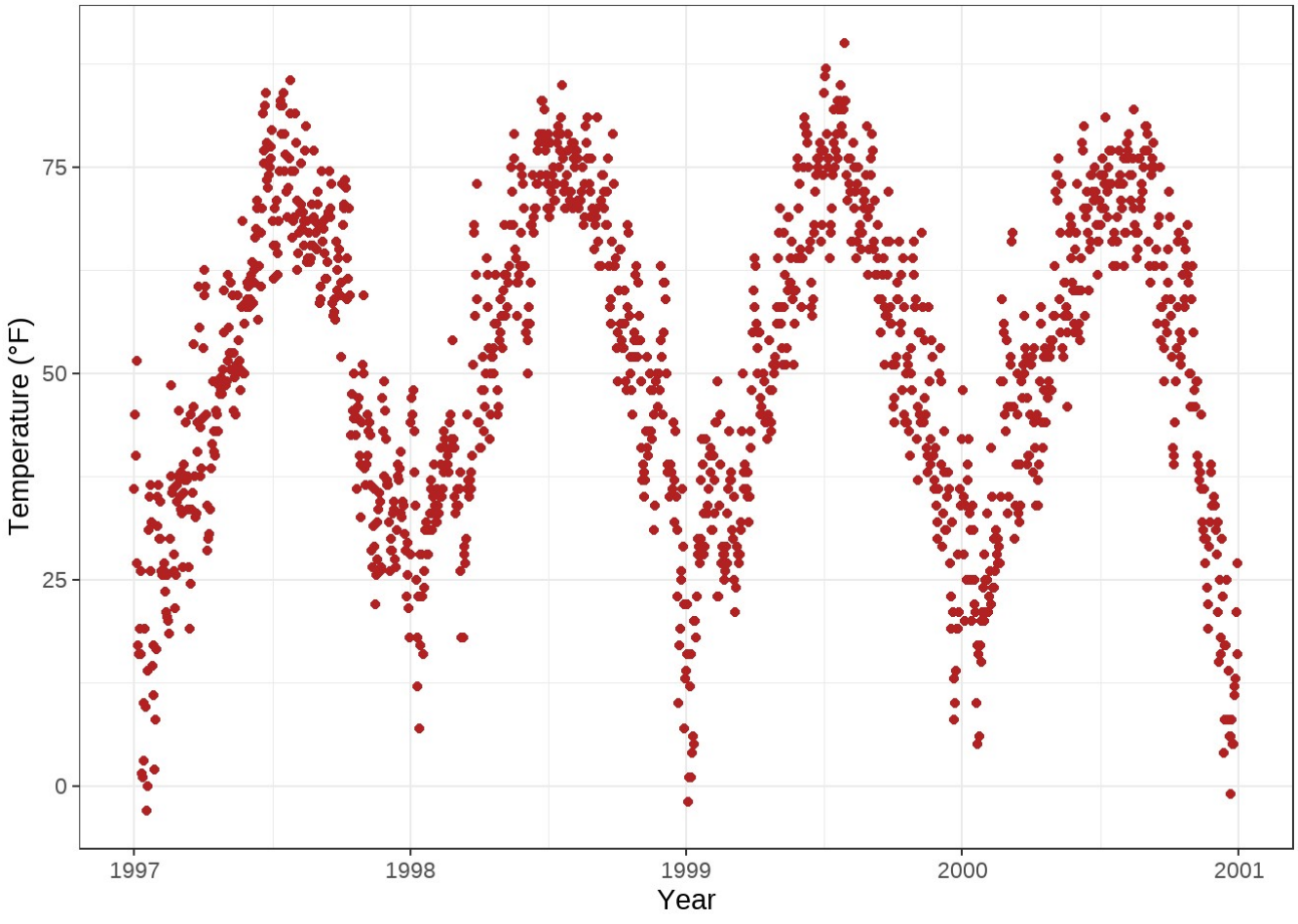
↑ Jump back to Table of Content.

WORKING WITH AXES

CHANGE AXIS TITLES

Let's add some well-written labels to the axes. For this, we add `labs()` providing a character string for each label we want to change (here `x` and `y`):

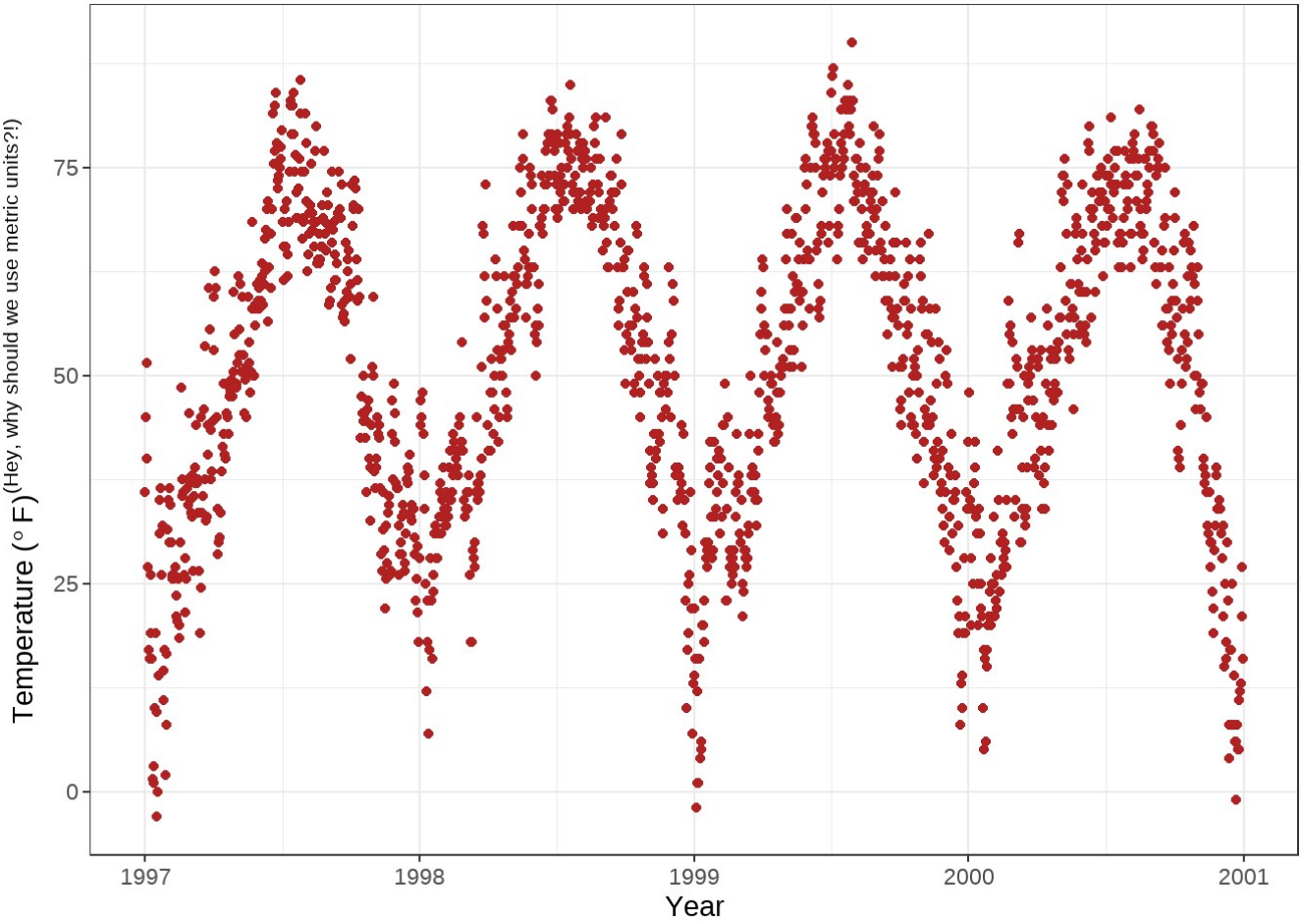
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)")
```



🤷 *You can also add each axis title via* `xlab()` *and* `ylab()`*. Expand to see example.*

Usually you can also specify symbols by simply adding the symbol itself (here "°") but the code below also allows to add not only symbols but e.g. superscripts:

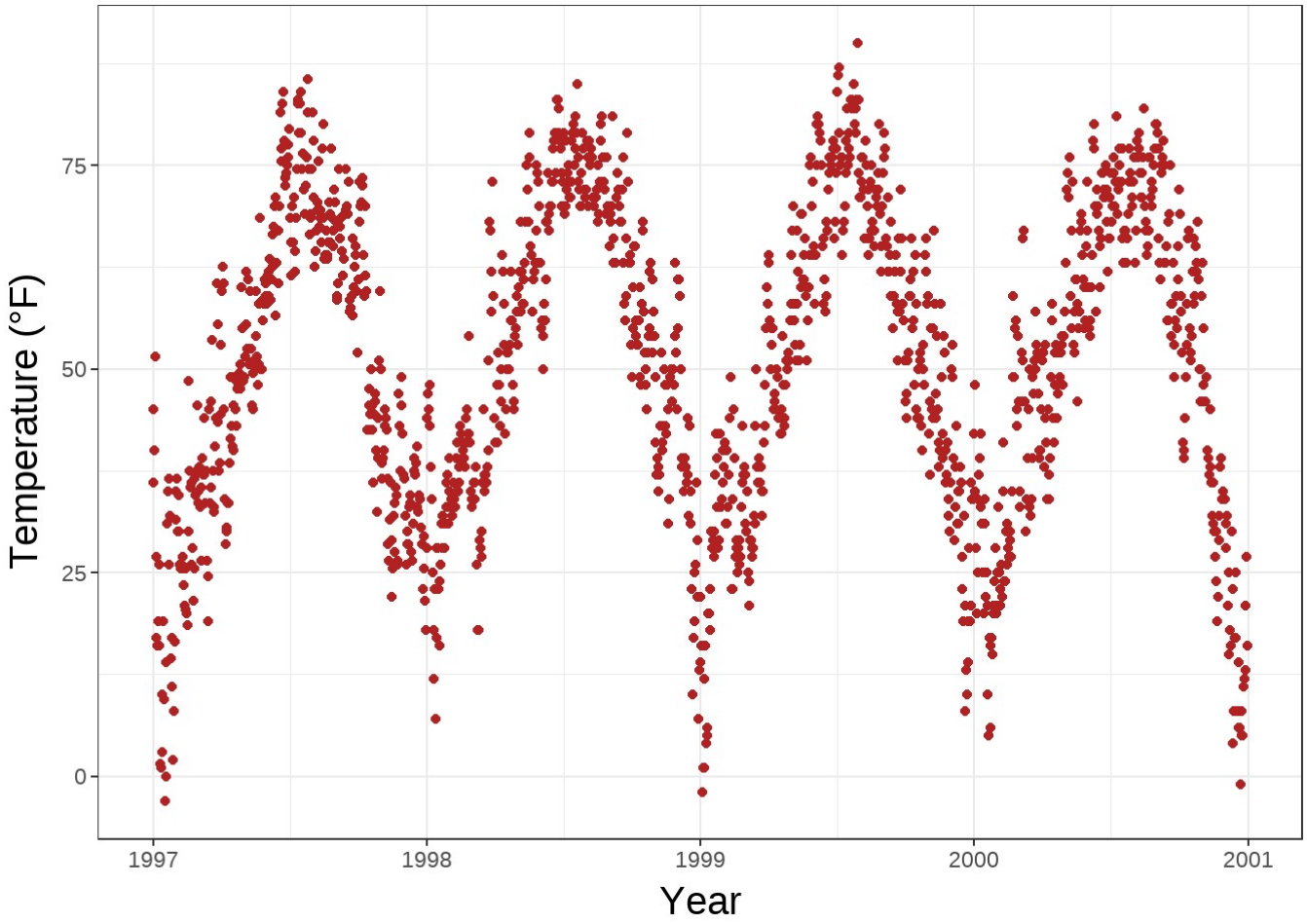
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = expression(paste("Temperature (", degree ~ F, ")"^"(Hey, why should
```



INCREASE SPACE BETWEEN AXIS AND AXIS TITLES

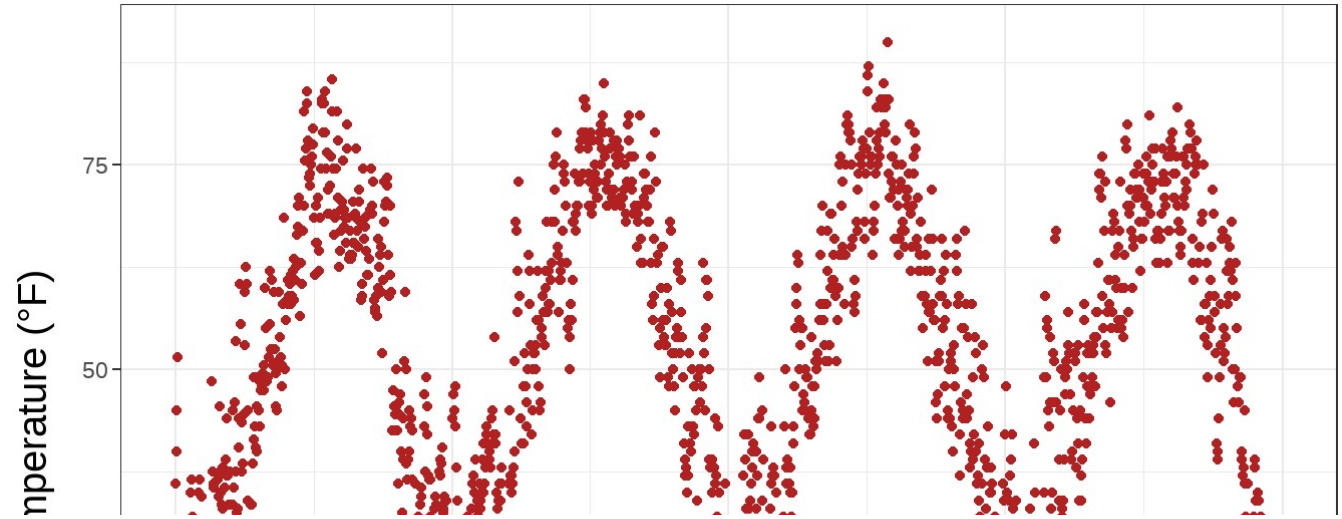
`theme()` is an essential command to modify particular theme elements (texts and titles, boxes, symbols, backgrounds, ...). We are going to use them a lot! For now, we are going to modify text elements. We can change the properties of all or particular text elements (here axis titles) by overwriting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

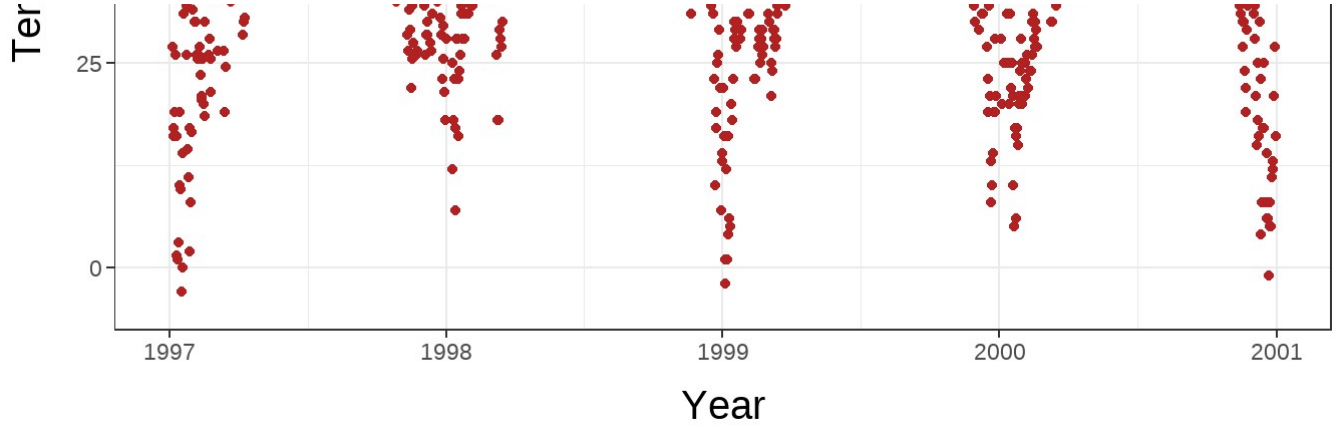
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(axis.title.x = element_text(vjust = 0, size = 15),
        axis.title.y = element_text(vjust = 2, size = 15))
```



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

```
ggplot(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))
```





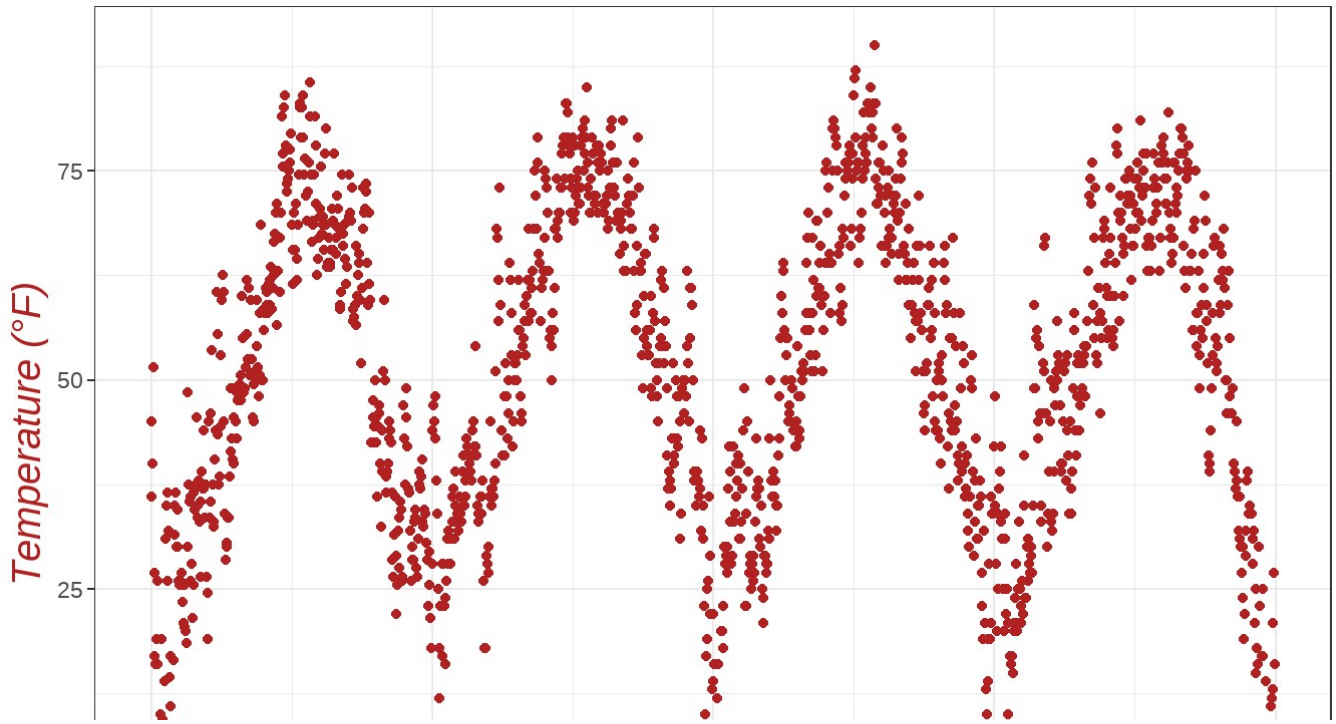
The labels `t` and `r` within the `margin()` object refer to *top* and *right*, respectively. You can also specify the four margins as `margin(t, r, b, l)`. Note that we now have to change the right margin to modify the space on the y axis, not the bottom margin.

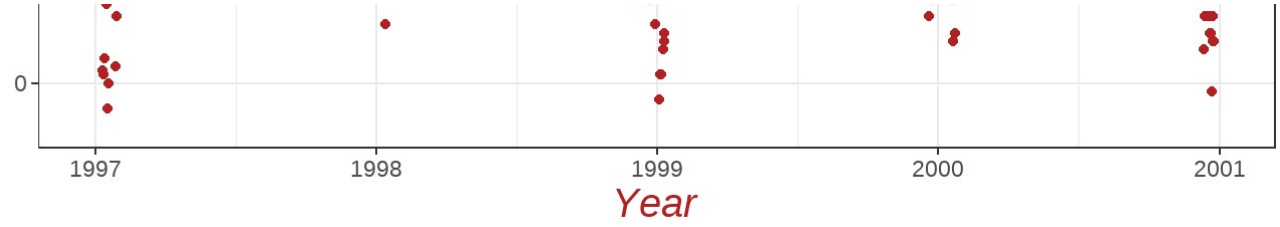
💡 **A good way to remember the order of the margin sides is "*t*-*r*-oub-*l*-*e*".**

CHANGE AESTHETICS OF AXIS TITLES

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

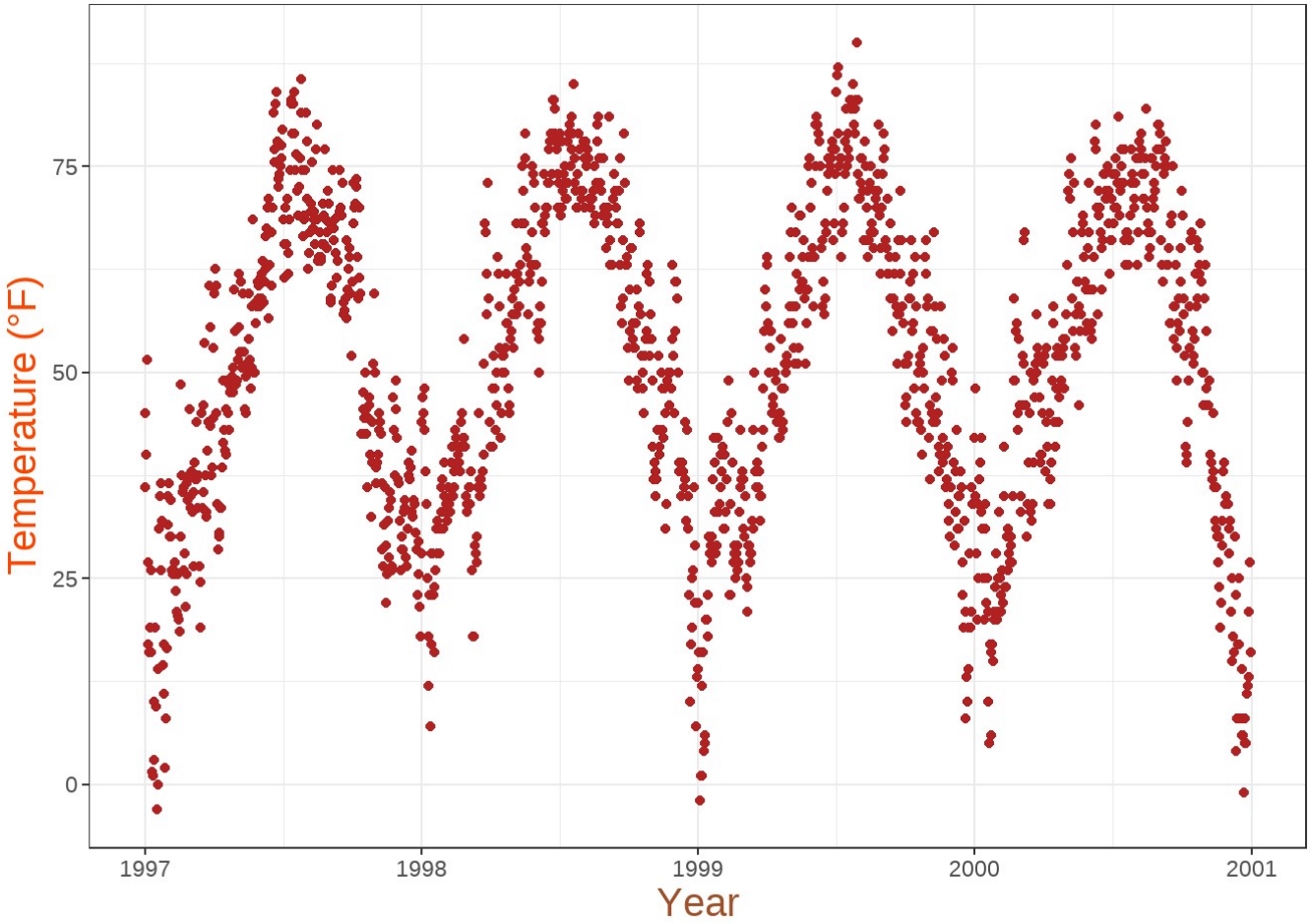
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(axis.title = element_text(size = 15, color = "firebrick",
                                  face = "italic"))
```





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

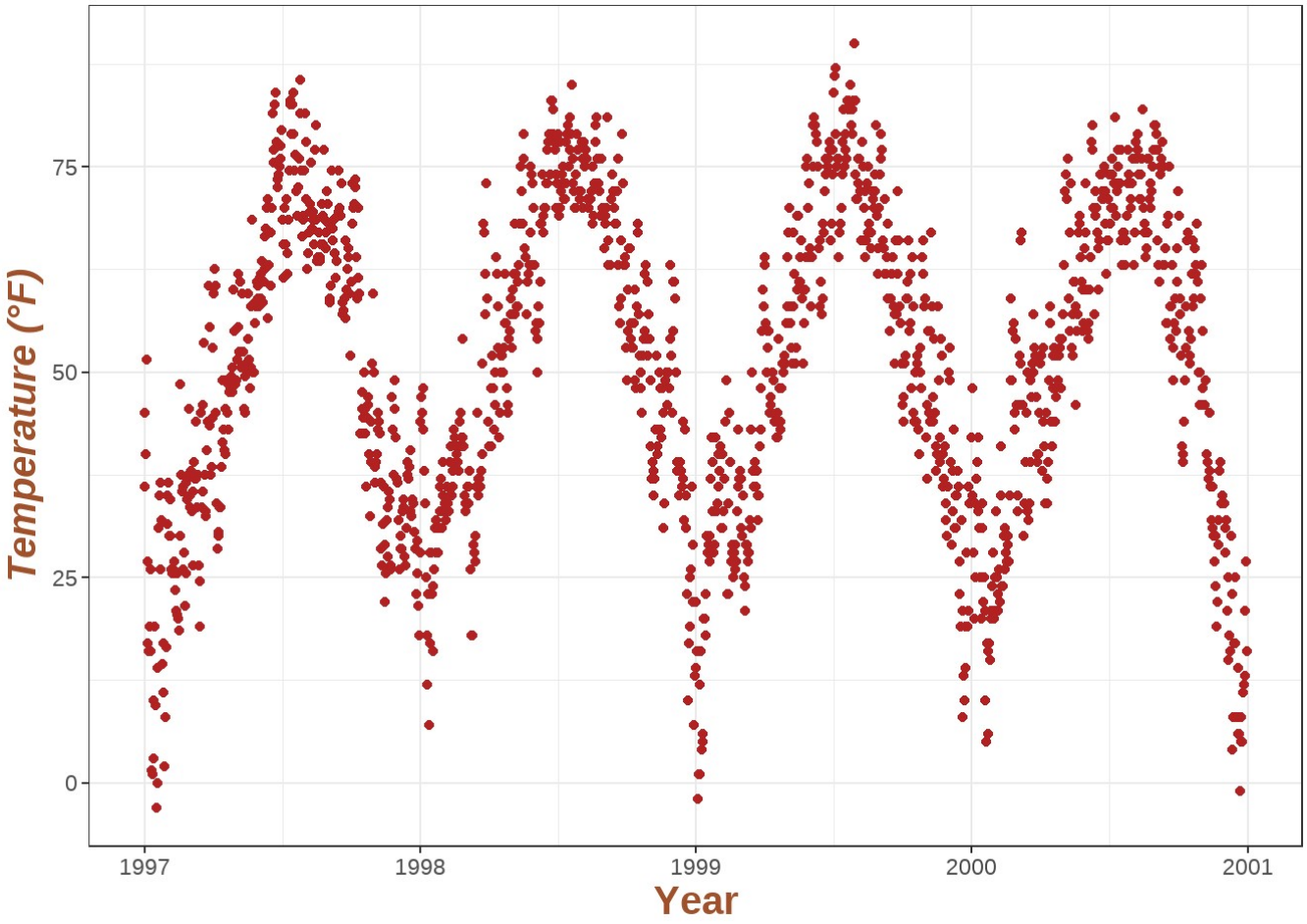
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(axis.title.x = element_text(color = "sienna", size = 15),
        axis.title.y = element_text(color = "orangered", size = 15))
```



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

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

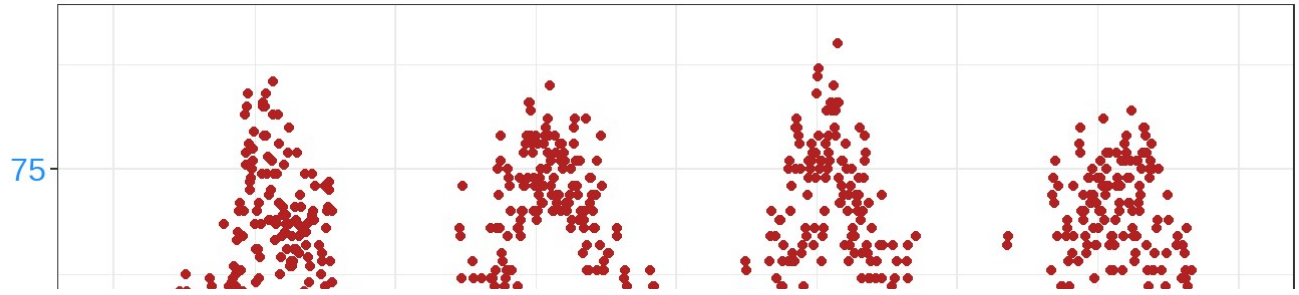
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(axis.title = element_text(color = "sienna", size = 15, face = "bold"),
        axis.title.y = element_text(face = "bold.italic"))
```

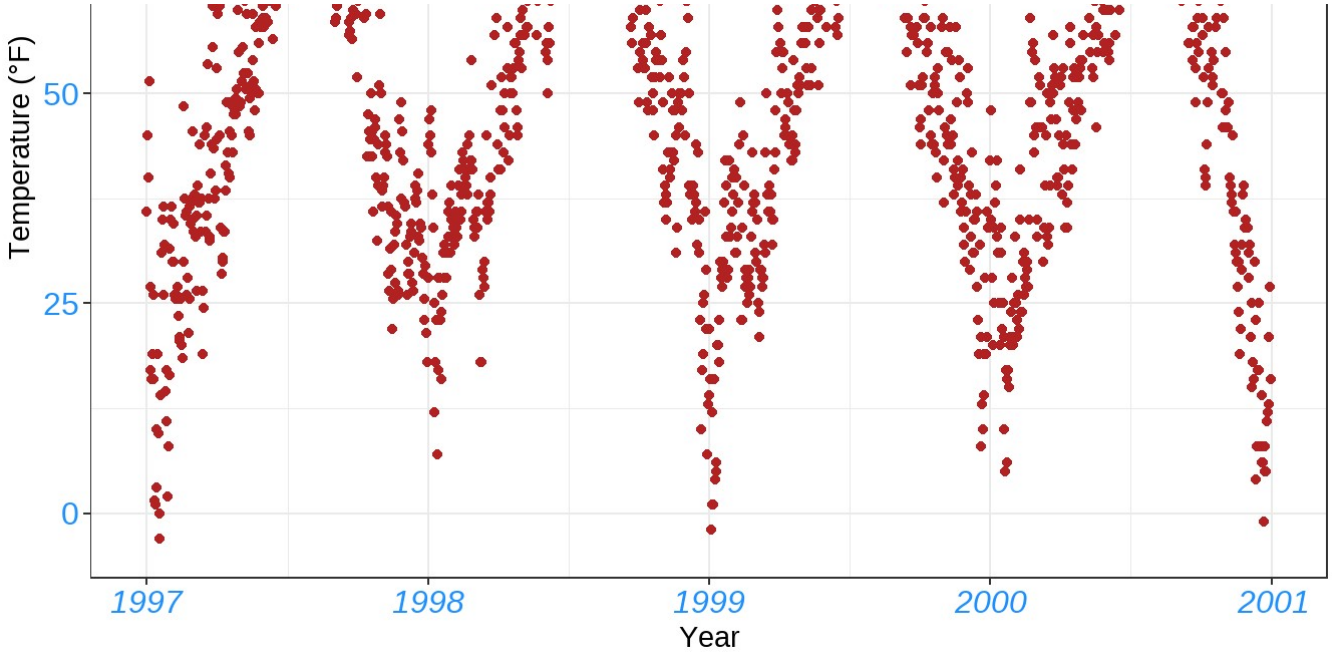


CHANGE AESTHETICS OF AXIS TEXT

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

```
ggplot(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"))
```

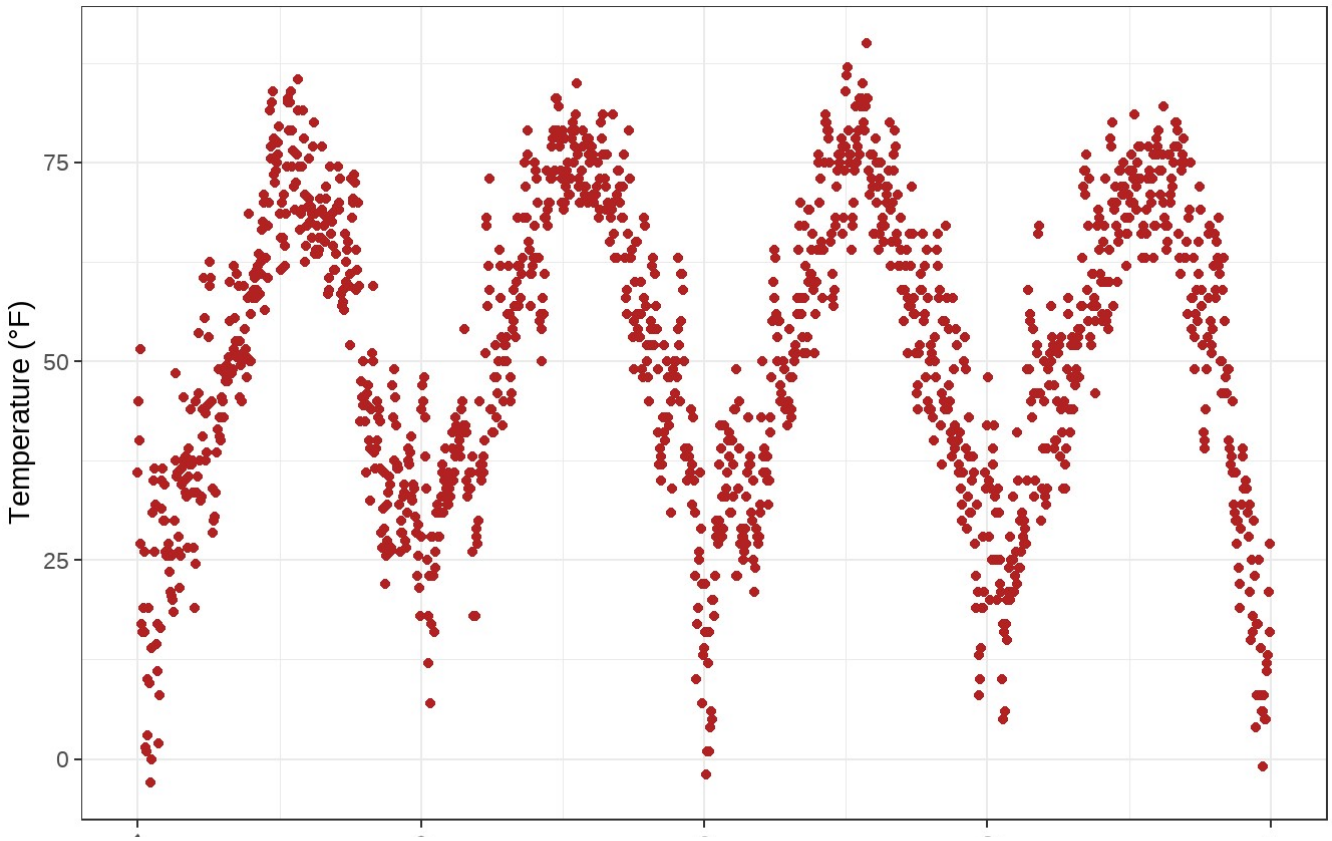




ROTATE AXIS TEXT

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

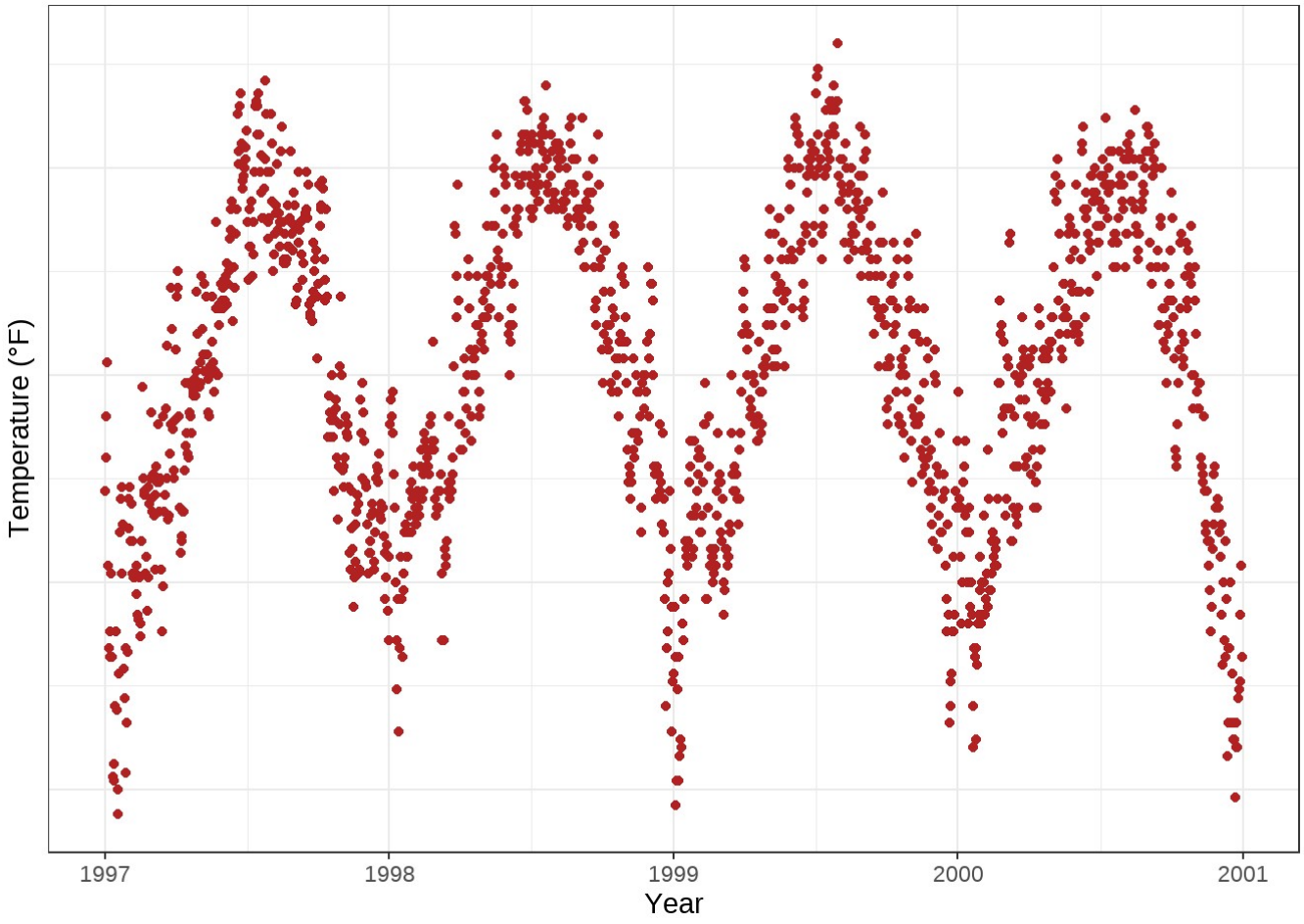
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(axis.text.x = element_text(angle = 50, vjust = 1, hjust = 1, size = 12))
```



REMOVE AXIS TEXT & TICKS

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

```
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())
```



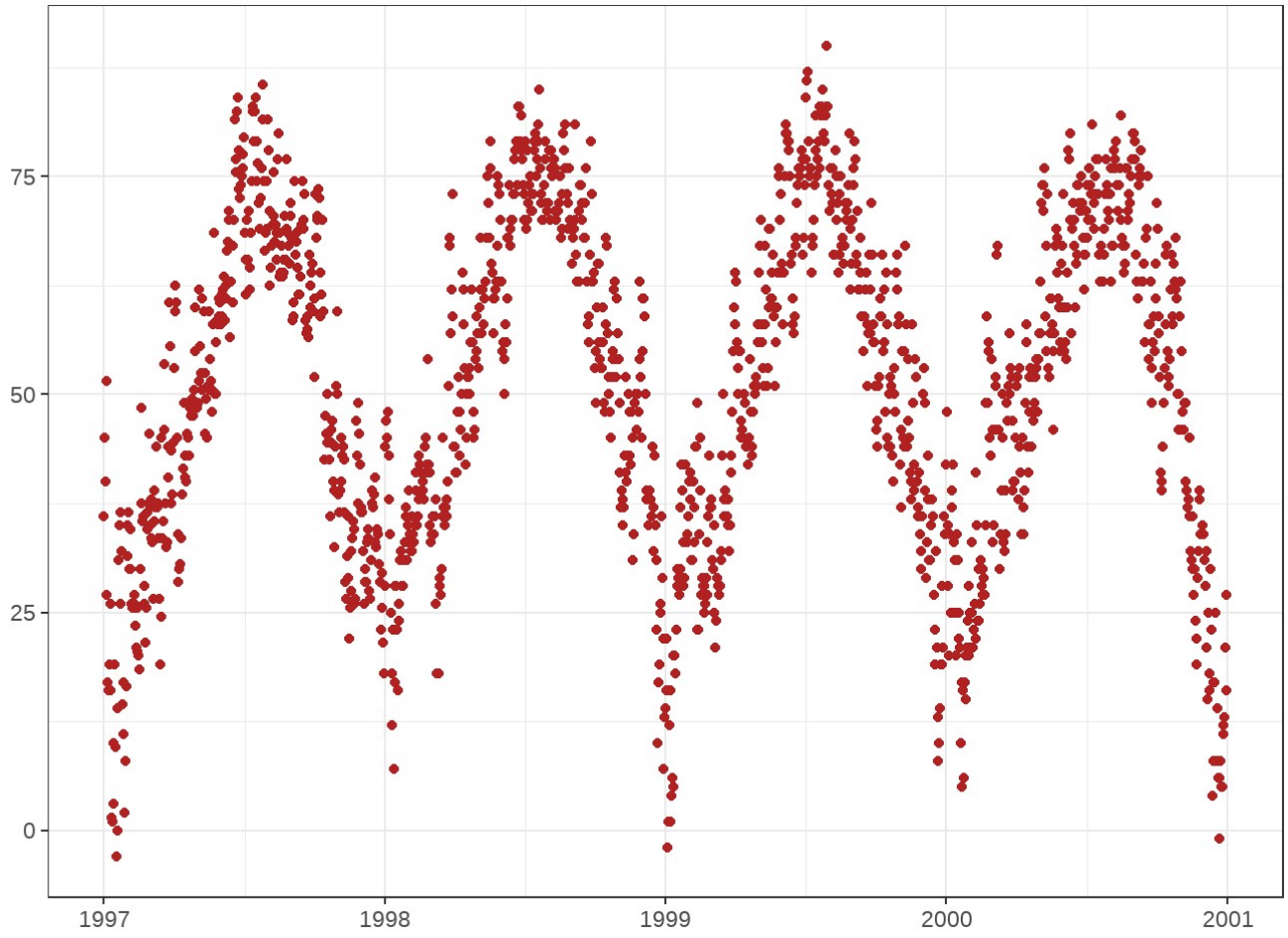
I introduced three theme elements—text, lines, and rectangles—but actually there is one more: `element_blank()` which removes the element (and thus is not considered an official element).

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

REMOVE AXIS TITLES

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

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

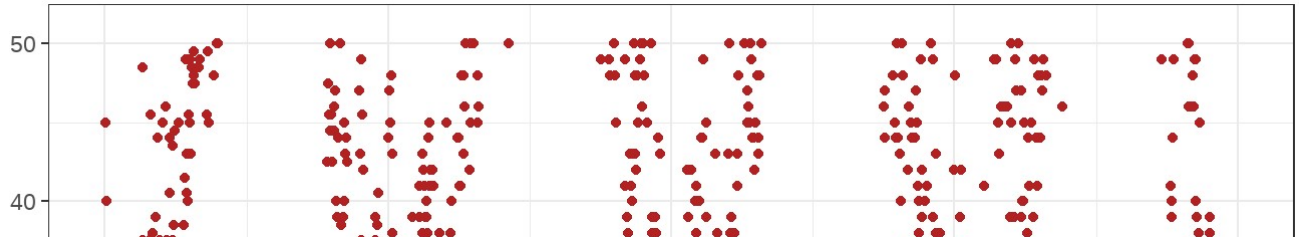


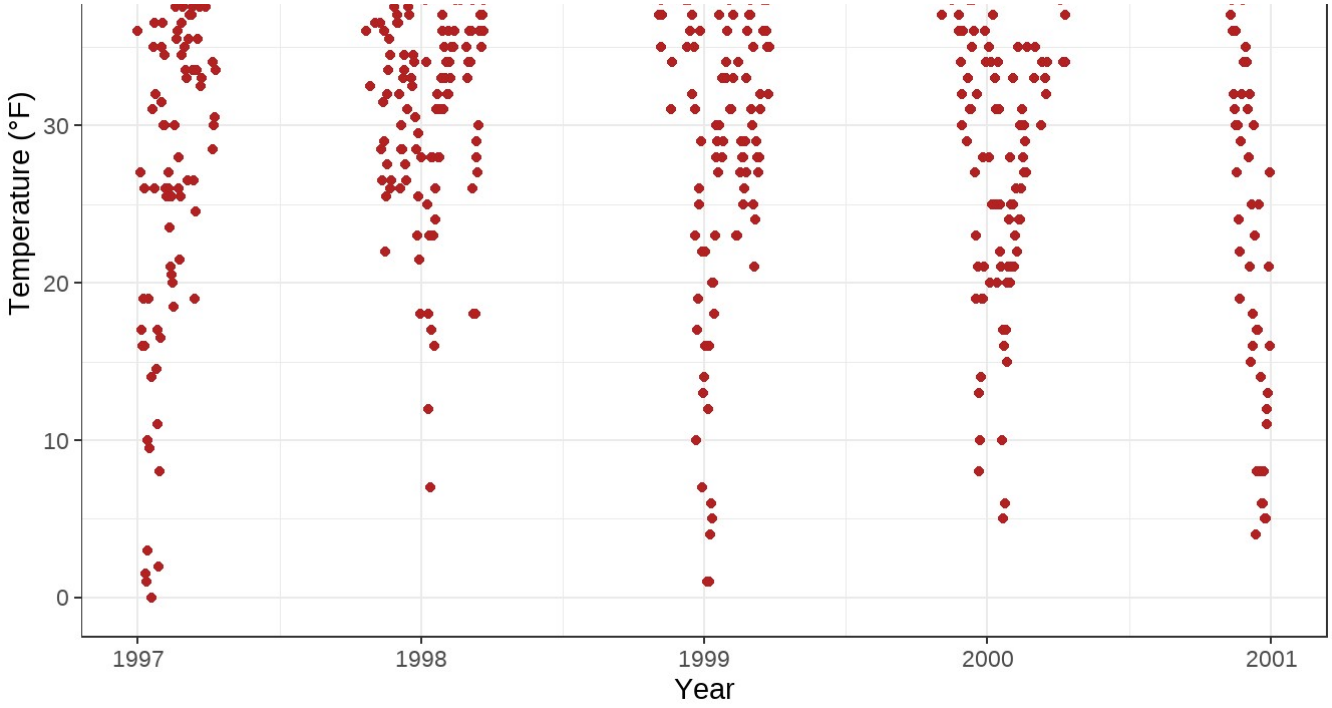
💡 **Note that `NULL` removes the element (similarly to `element_blank()`) while empty quotes `""` will keep the spacing for the axis title and simply print nothing.**

LIMIT AXIS RANGE

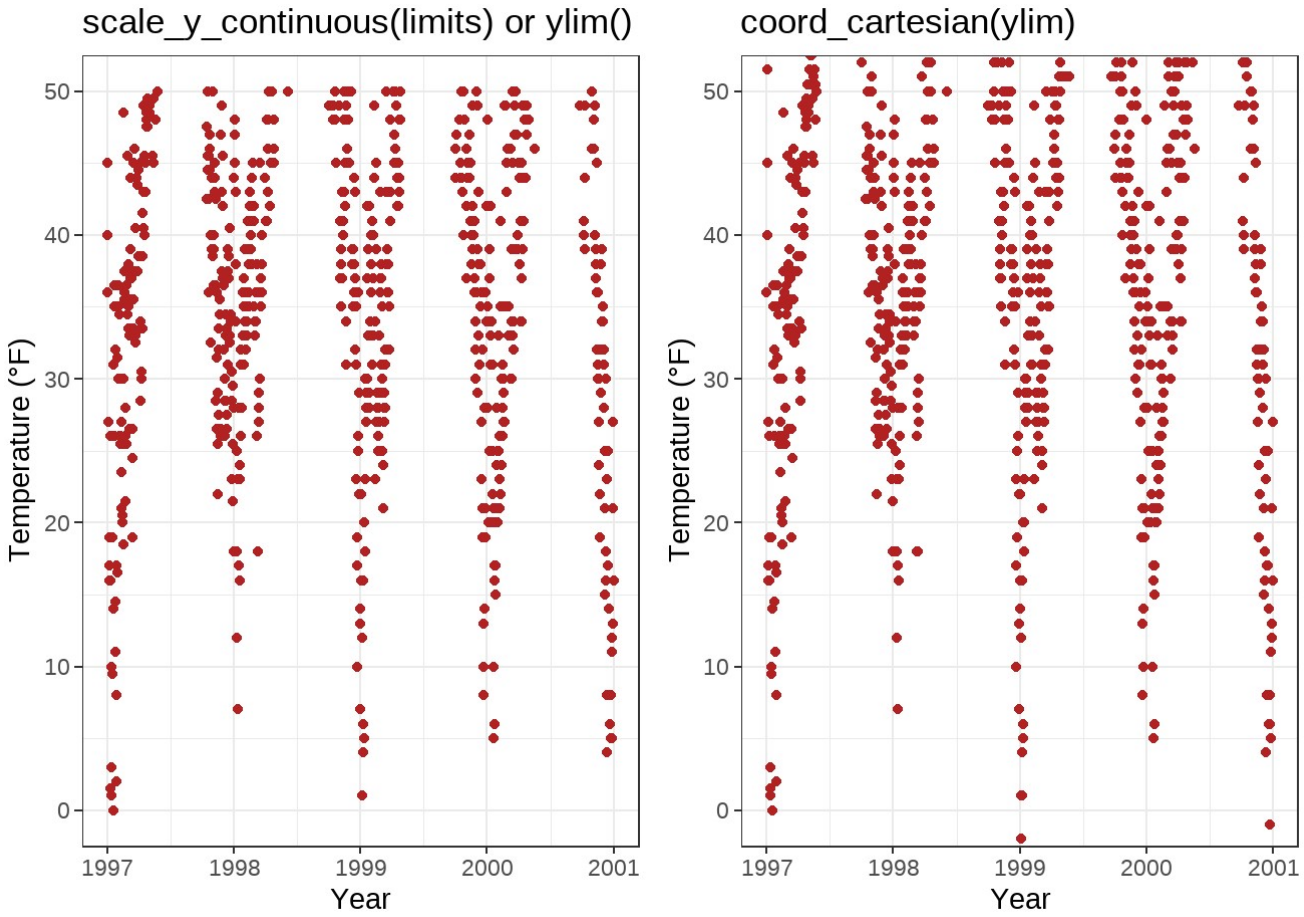
Sometimes you want to ~~zoom into~~ take a closer look at some range of your data. You can do this without subsetting your data:

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

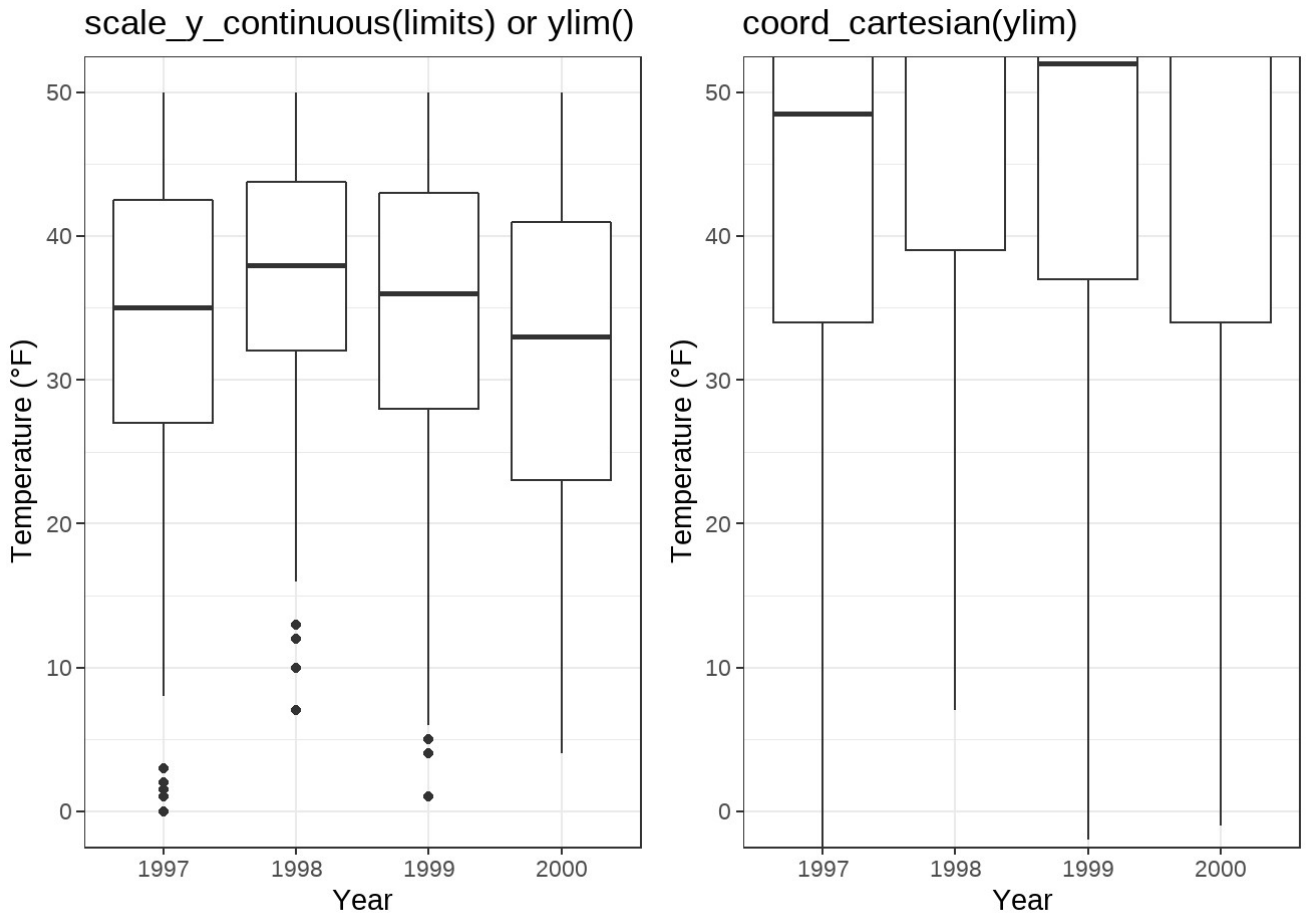




Alternatively you can use `scale_y_continuous(limits = c(0, 50))` or `coord_cartesian(ylim = c(0, 50))`. The former removes all data points outside the range while the second adjusts the visible area and is similar to `ylim(c(0, 50))`. You may wonder: *So in the end both result in the same.* But not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



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

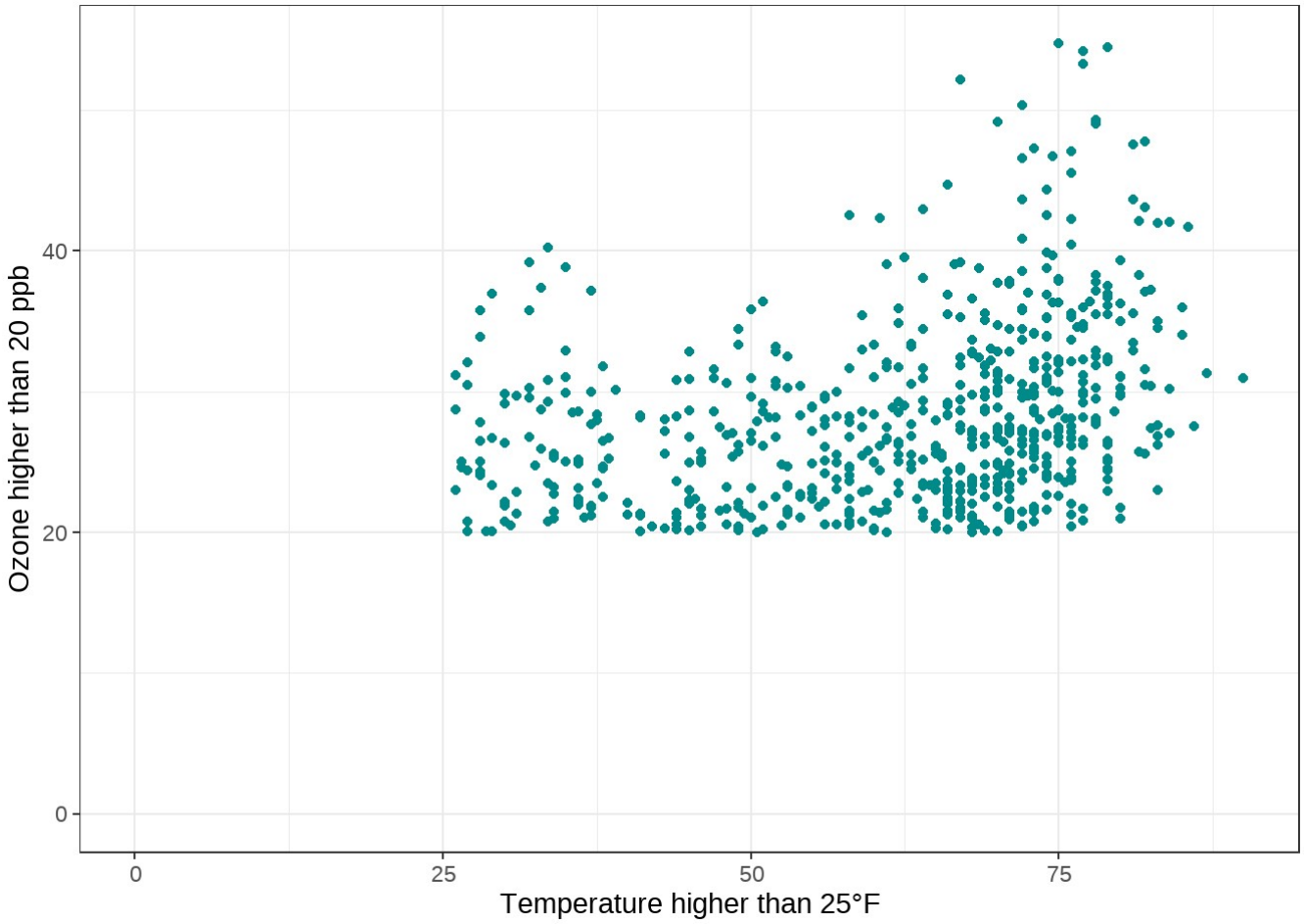


Um. Because `scale_x|y_continuous()` subsets the data first, we get completely different (and wrong, at least if in the case this was not your aim) estimates for the box plots! I hope you don't have to go back to your old scripts now and check if you *maybe* have manipulated your data while plotting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

FORCE PLOT TO START AT ORIGIN

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

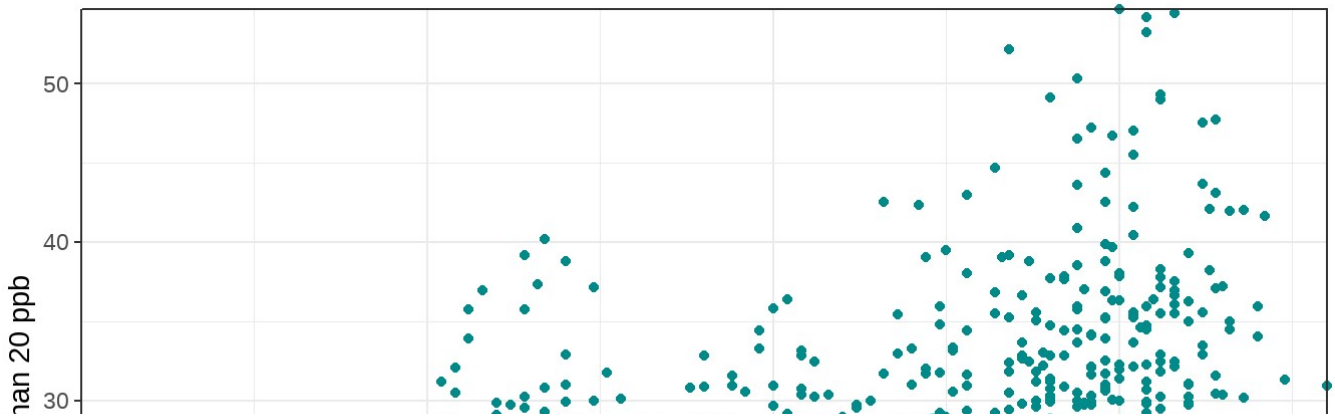
```
library(tidyverse)

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)
```

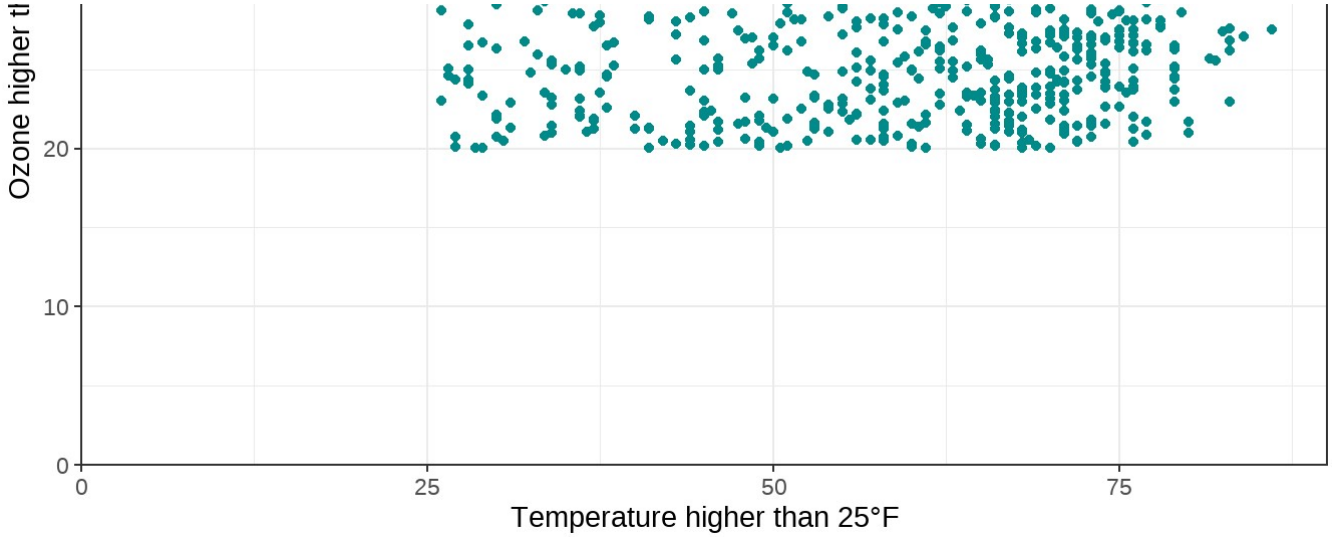


🙋 *Using* `coord_cartesian(xlim = c(0, NA), ylim = c(0, NA))` *will lead to the same result. Expand to see example.*

But we can also force it to *literally* start 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) +
  scale_x_continuous(expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  coord_cartesian(clip = "off")
```





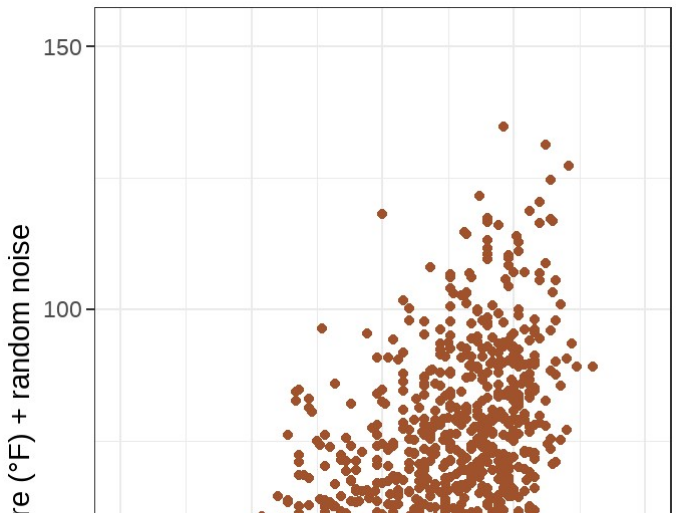
💡 **The argument `clip = "off"` in any coordinate system, always starting with `coord_*`, allows to draw outside of the panel area.**

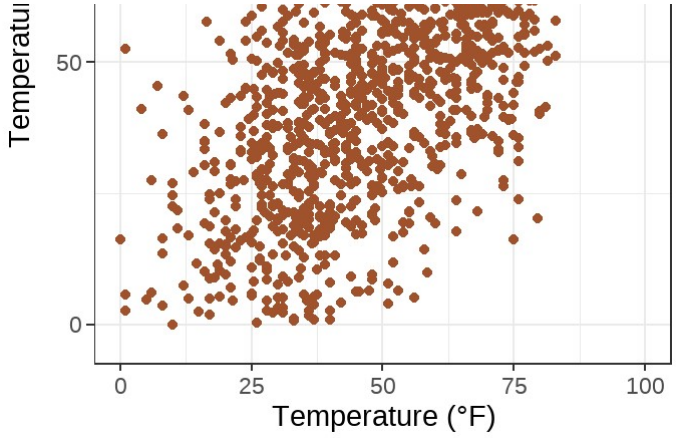
Here, I call it to make sure that the tick marks at `c(0, 0)` are not cut. See the Twitter thread by Claus Wilke (https://twitter.com/clauswilke/status/991542952802619392?lang=en) for more details.

AXES WITH SAME SCALING

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

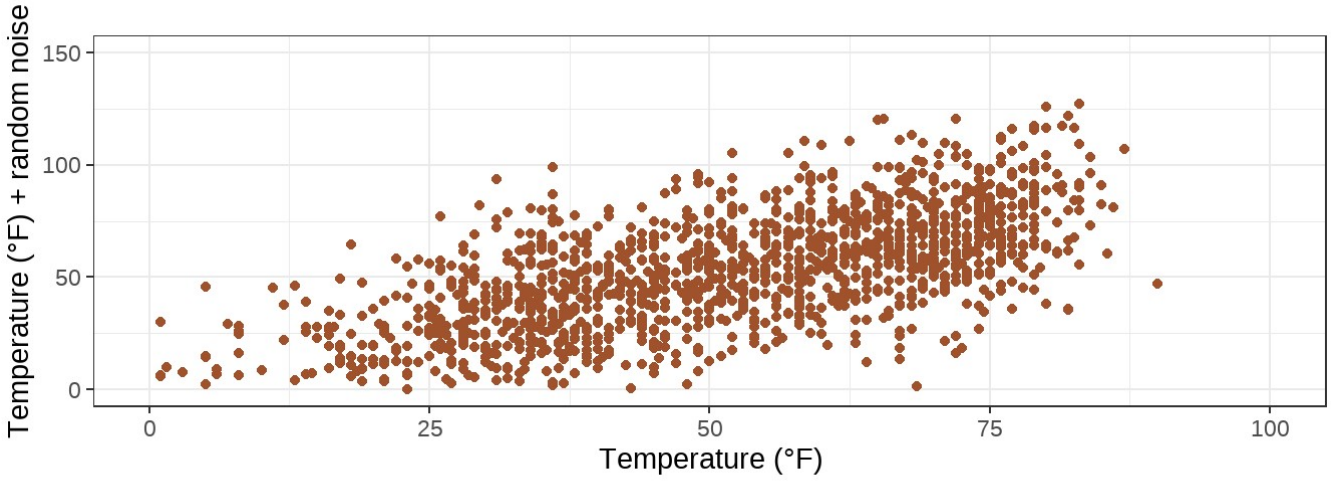
```
ggplot(chic, aes(x = temp, y = temp + rnorm(nrow(chic), sd = 20))) +
  geom_point(color = "sienna") +
  labs(x = "Temperature (°F)", y = "Temperature (°F) + random noise") +
  xlim(c(0, 100)) + ylim(c(0, 150)) +
  coord_fixed()
```





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

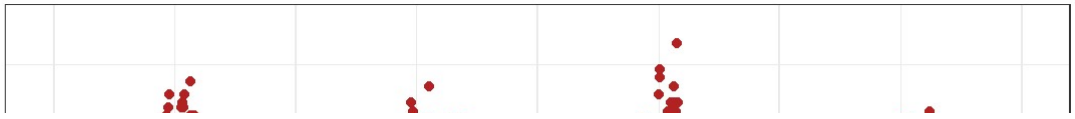
```
ggplot(chic, aes(x = temp, y = temp + rnorm(nrow(chic), sd = 20))) +
  geom_point(color = "sienna") +
  labs(x = "Temperature (°F)", y = "Temperature (°F) + random noise") +
  xlim(c(0, 100)) + ylim(c(0, 150)) +
  coord_fixed(ratio = 1/5)
```

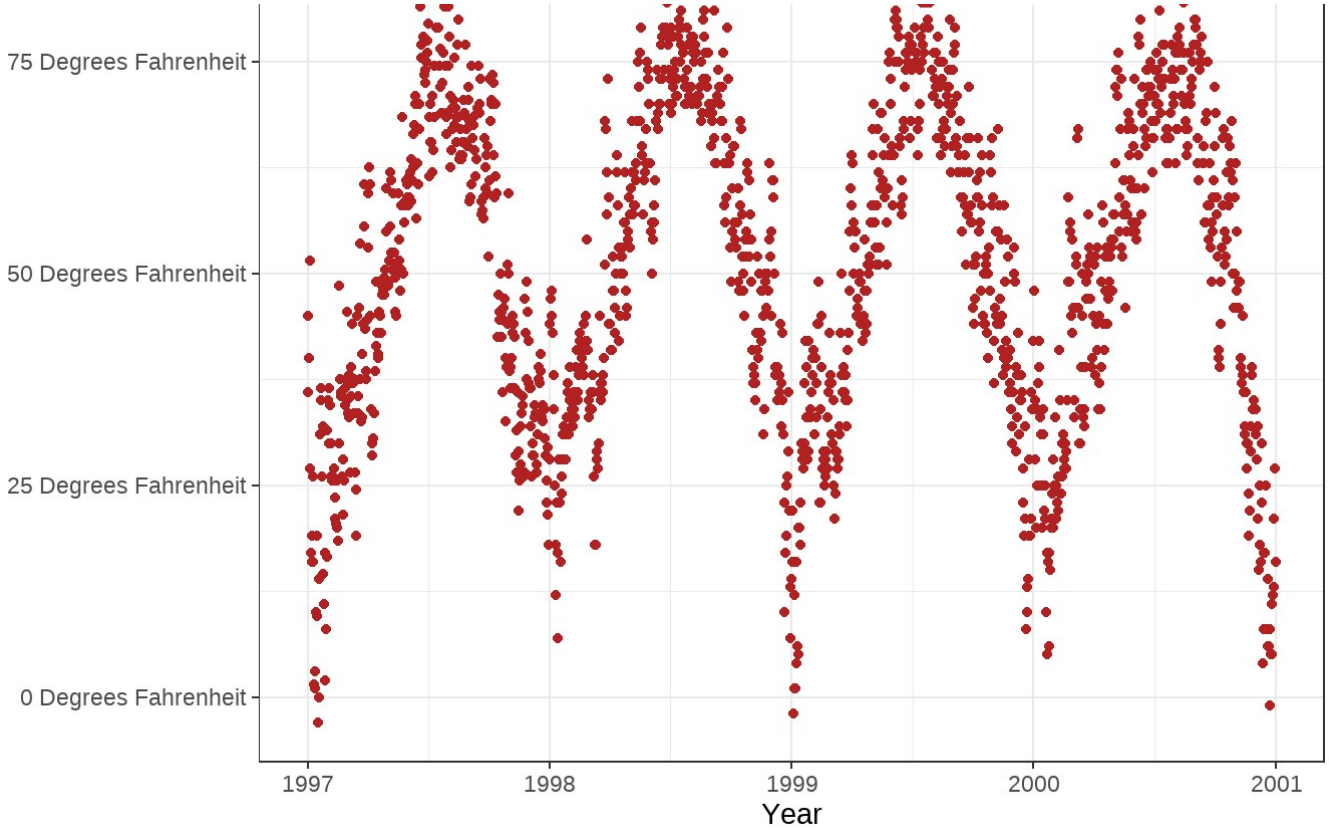


USE A FUNCTION TO ALTER LABELS

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

```
ggplot(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"))})
```





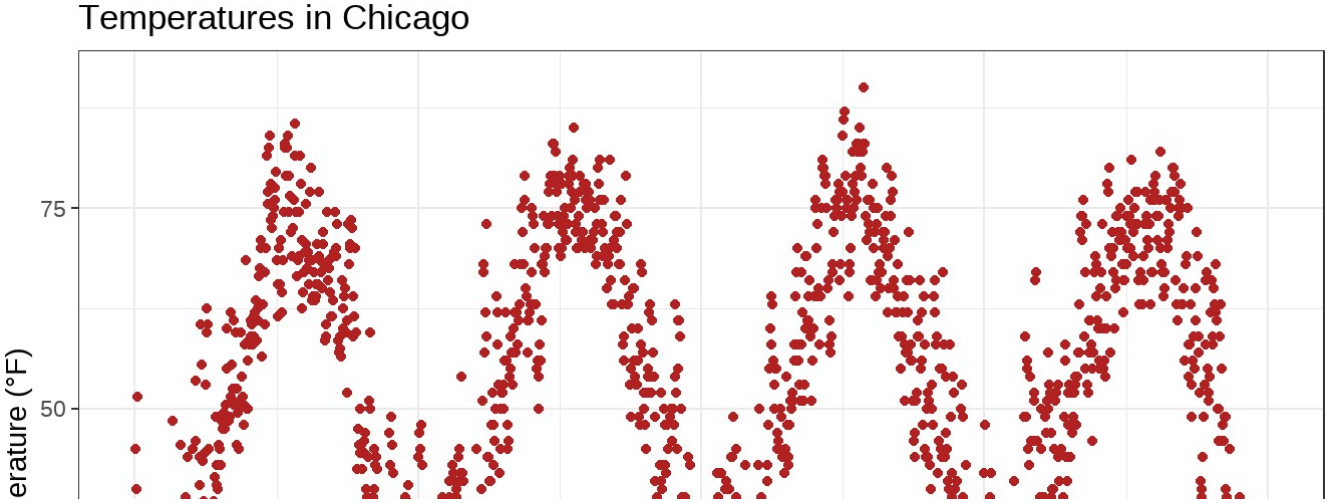
↑ Jump back to Table of Content.

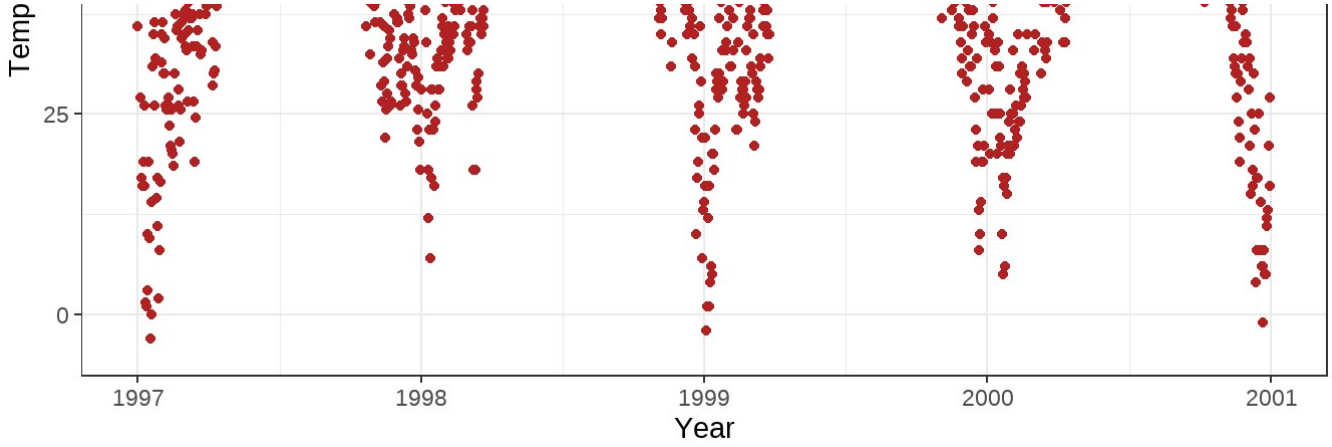
WORKING WITH TITLES

ADD A TITLE

We can add a title via the `ggtitle()` function:

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

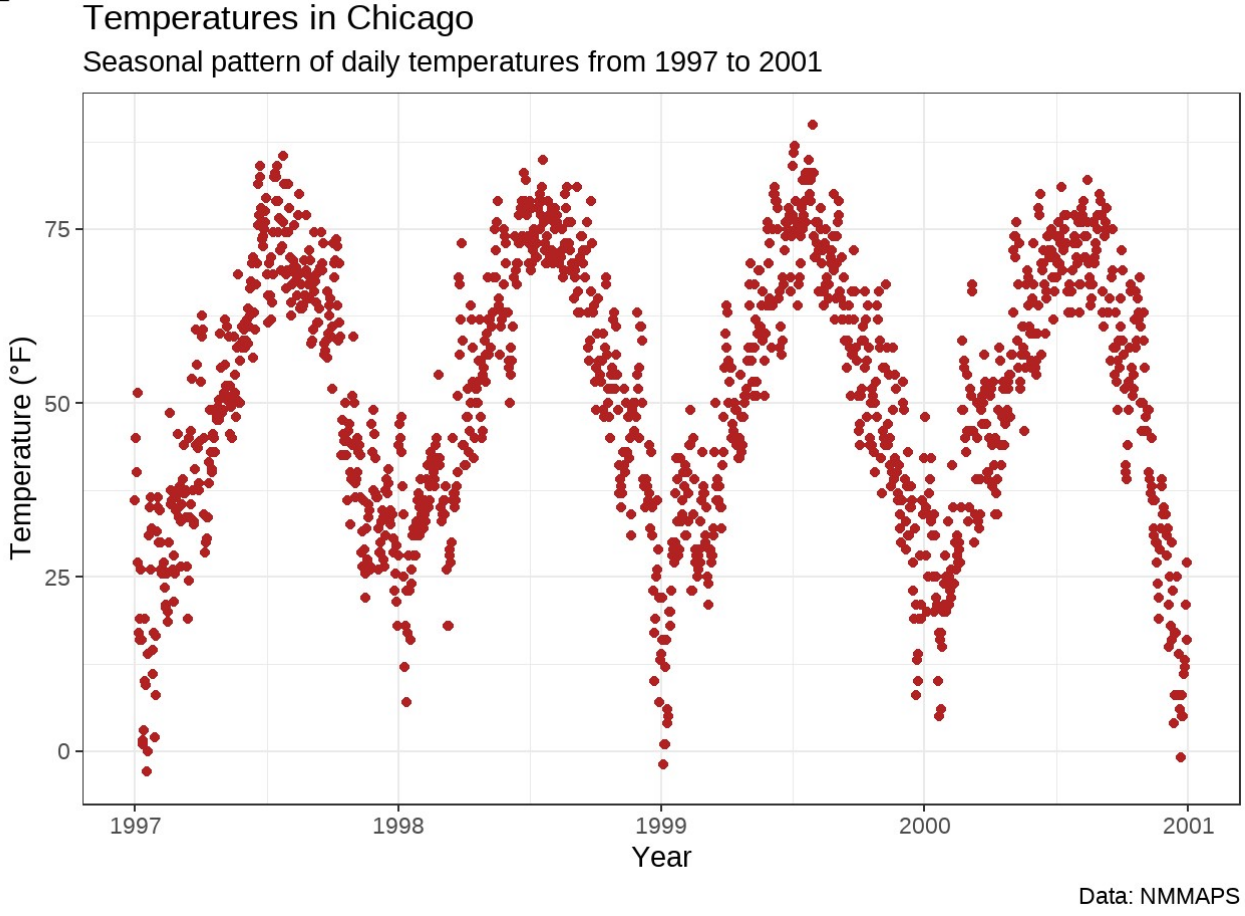




Alternatively, you can use `labs()`. Here you can add several arguments, e.g. additionally a subtitle, a caption and a tag (as well as axis titles as shown before):

```
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")
```

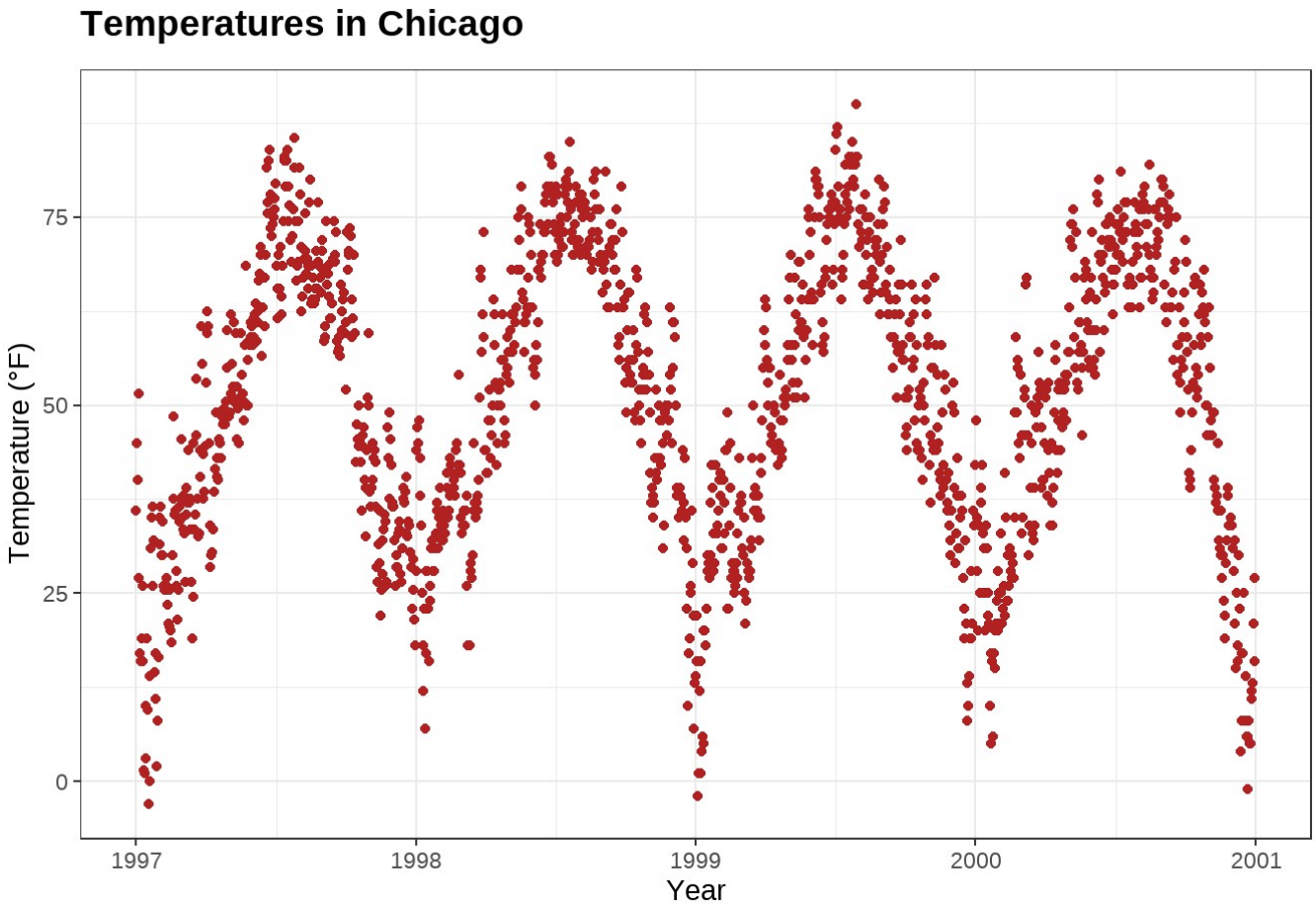
Fig. 1



MAKE TITLE BOLD & ADD A SPACE AT THE BASELINE

Again, since we want to modify the properties of a theme element, we use the `theme()` function and as for the text elements `axis.title` and `axis.text` modify the font face and the margin. All the following modifications of theme elements work not only for the title but for all other labels such as `plot.subtitle`, `plot.caption`, `plot.caption`, `legend.title`, `legend.text`, and `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))
```

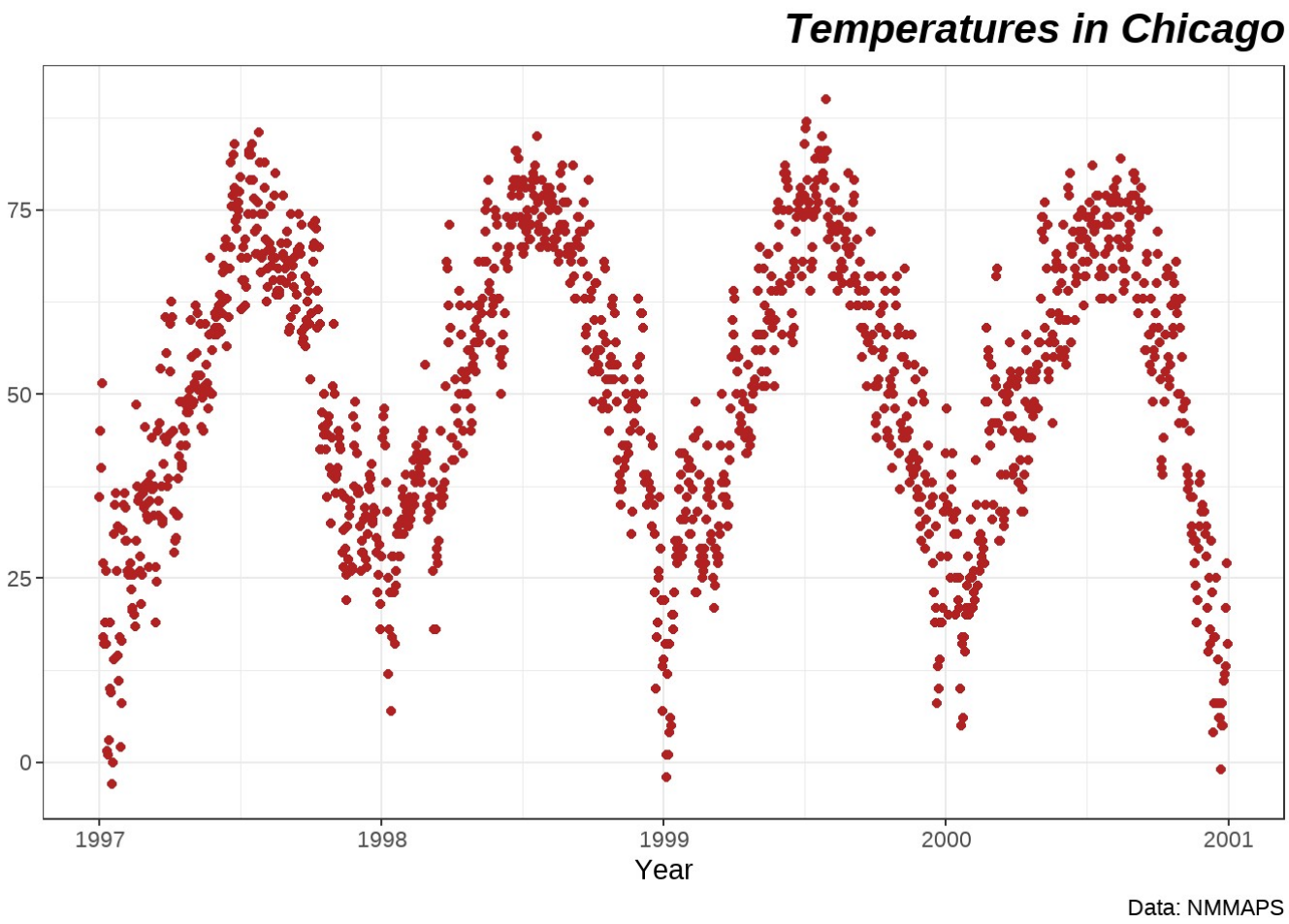


💡 **A nice way to remember the order of the margin arguments is "*t*-*r*-oub-*l*-*e*" that resembles the first letter of the four sides.**

ADJUST POSITION OF TITLES

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

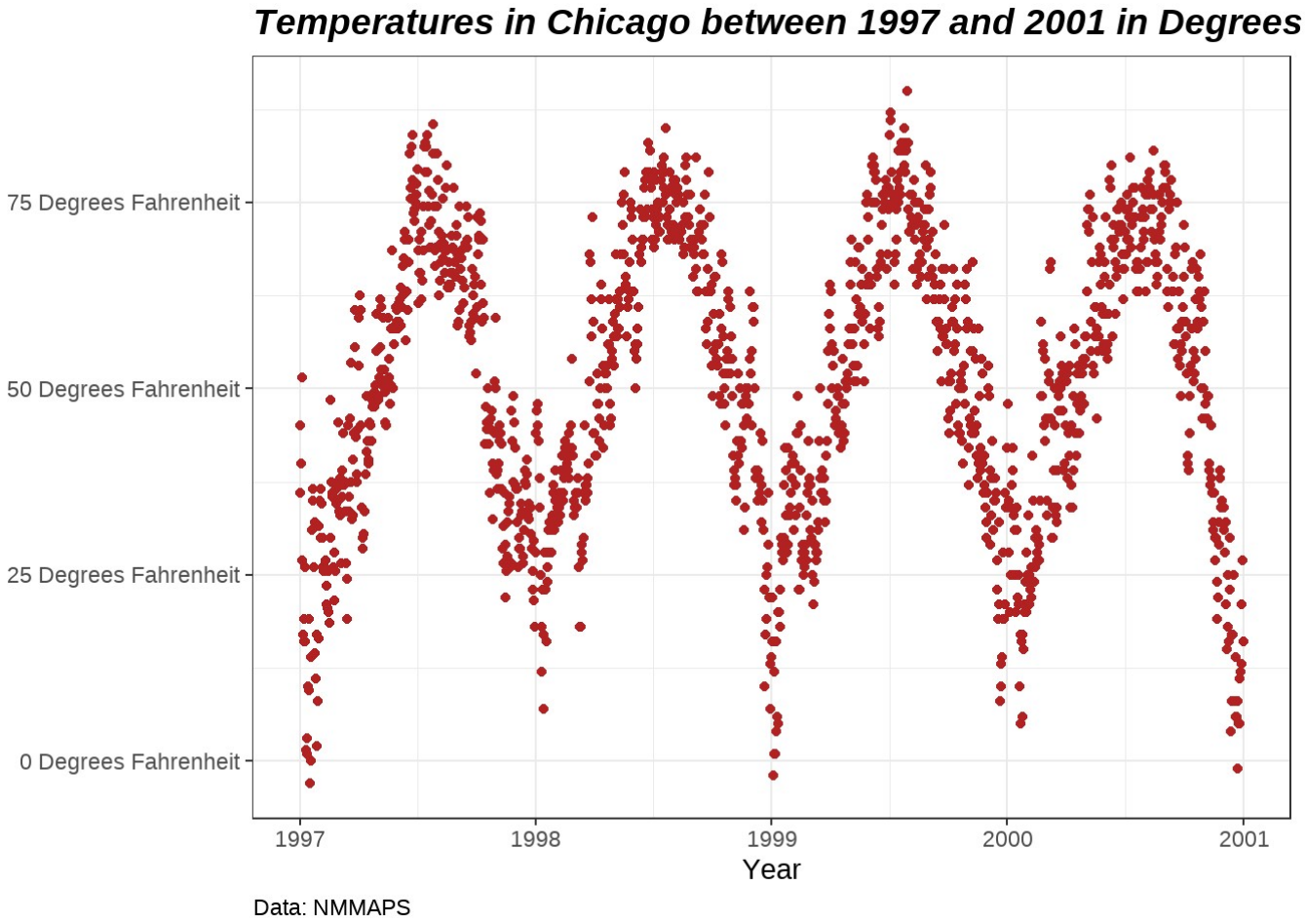
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = NULL,
       title = "Temperatures in Chicago",
       caption = "Data: NMMAPS") +
  theme(plot.title = element_text(hjust = 1, size = 16, face = "bold.italic"))
```



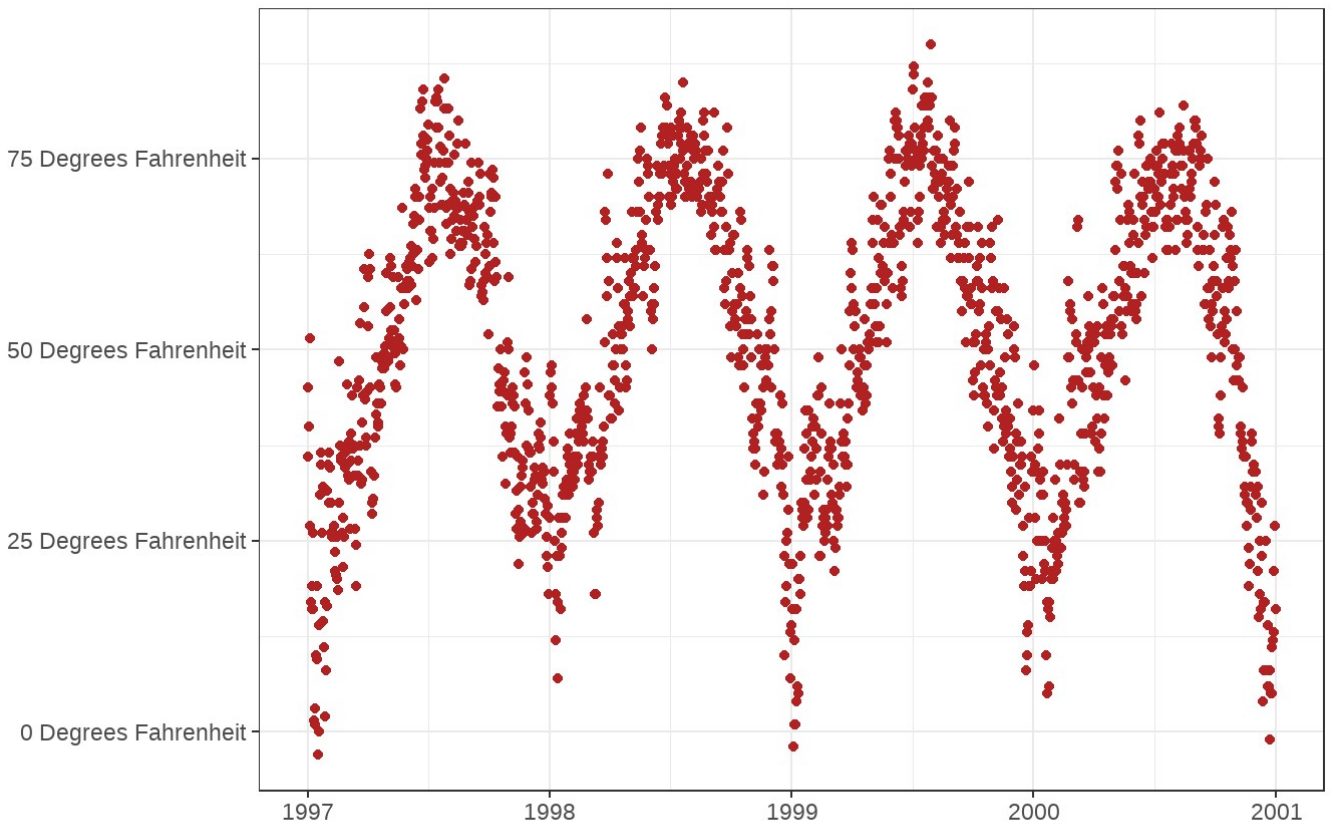
Of course, there it is also possible to adjust the vertical alignment, controlled by `vjust`.

Since 2019, the user is 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 later is actually the better choice designwise in most cases and many people were very happy about that new feature since especially with very long y axis labels the alignment looks awful:

```
(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")
```



Year

Data: NMMAPS

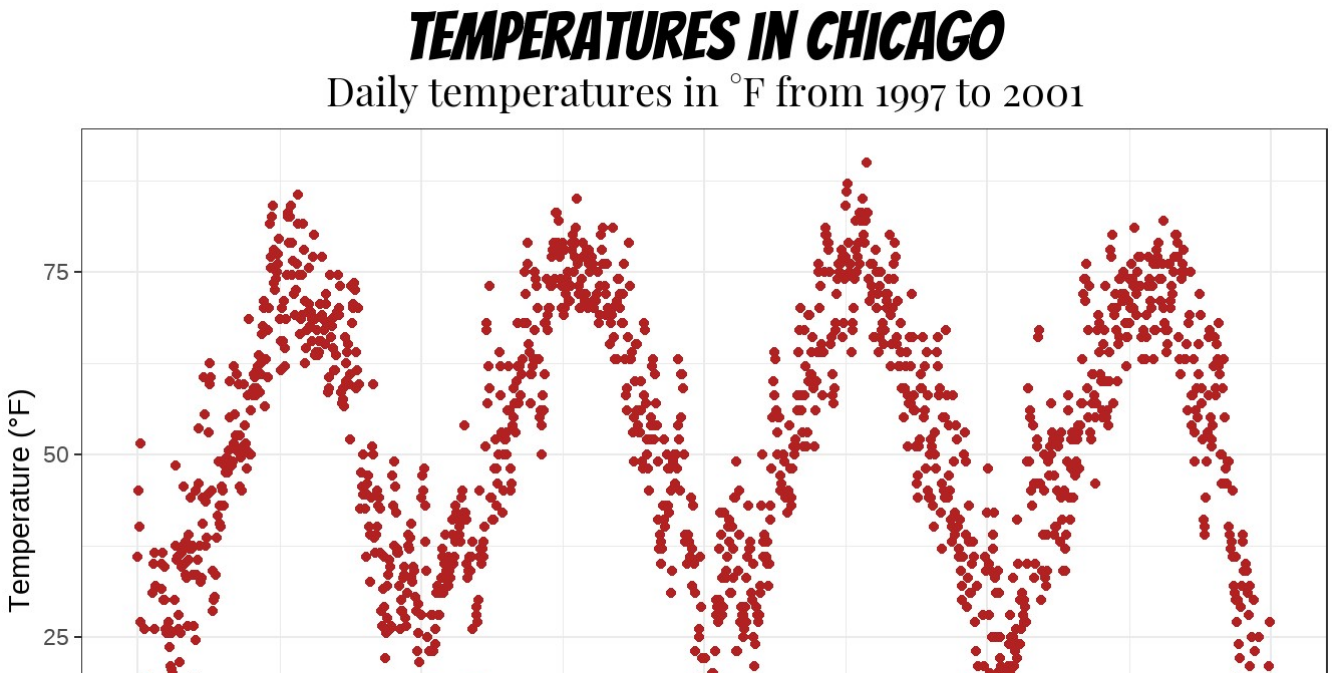
USE A NON-TRADITIONAL FONT IN YOUR TITLE

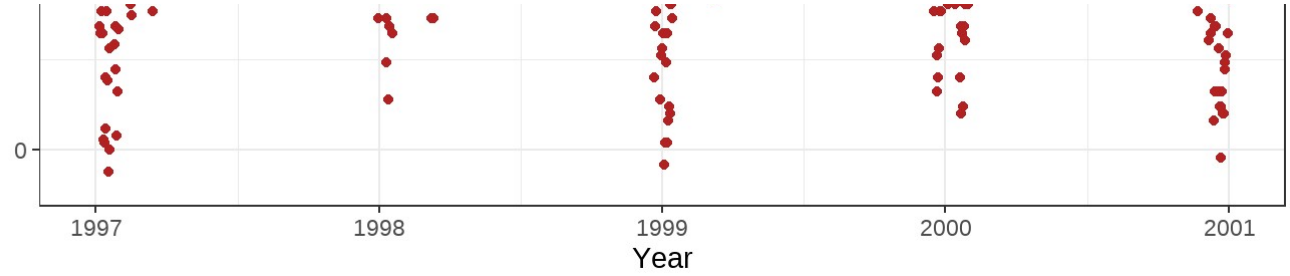
You can also use different fonts not only the default one provided by ggplot (and which differs between operating systems). There are several packages that help you to use fonts which are installed on your machine (and you may be using in your office program). Here, I use the `showtext` package (https://github.com/yixuan/showtext) that makes it easy to use various types of fonts (TrueType, OpenType, Type 1, web fonts, etc.) in R plots. After we have loaded the package, you need to import the font that has to be installed on your device as well. I regularly use Google fonts (https://fonts.google.com/) that can be imported with the function `font_add_google()` but you can also add other fonts with `font_add()`. (Note that even in case of using Google fonts you must install the font—and restart Rstudio—to use the font.)

```
library(showtext)
font_add_google("Playfair Display", ## name of Google font
                "Playfair")  ## name that will be used in R
font_add_google("Bangers", "Bangers")
```

Now, we can use those font families using—yeah, you guessed right— `theme()`:

```
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", hjust = .5, size = 15))
```





You can also set a non-default font for all text elements of your plots, for more details see section "Working with Themes". I am going to use *Roboto Condensed* as new default font for all the plots that follow.

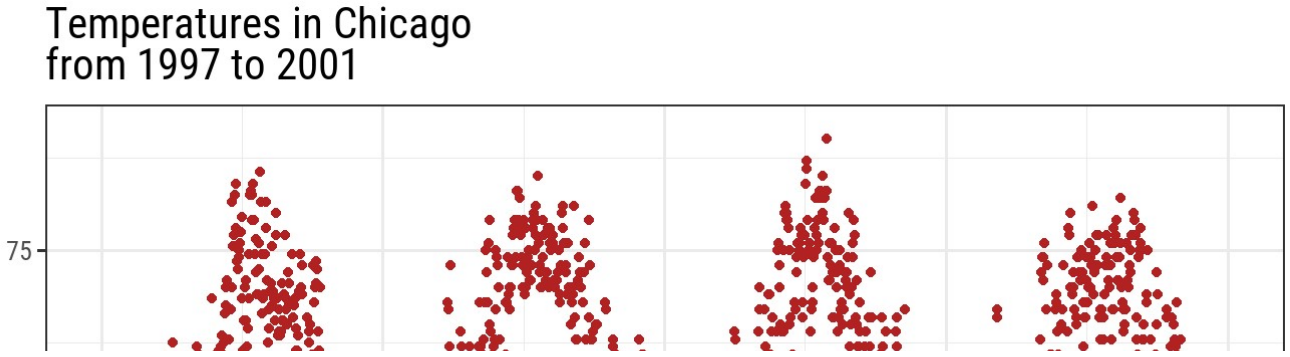
```
font_add_google("Roboto Condensed", "Roboto Condensed")
theme_set(theme_bw(base_size = 12, base_family = "Roboto Condensed"))
```

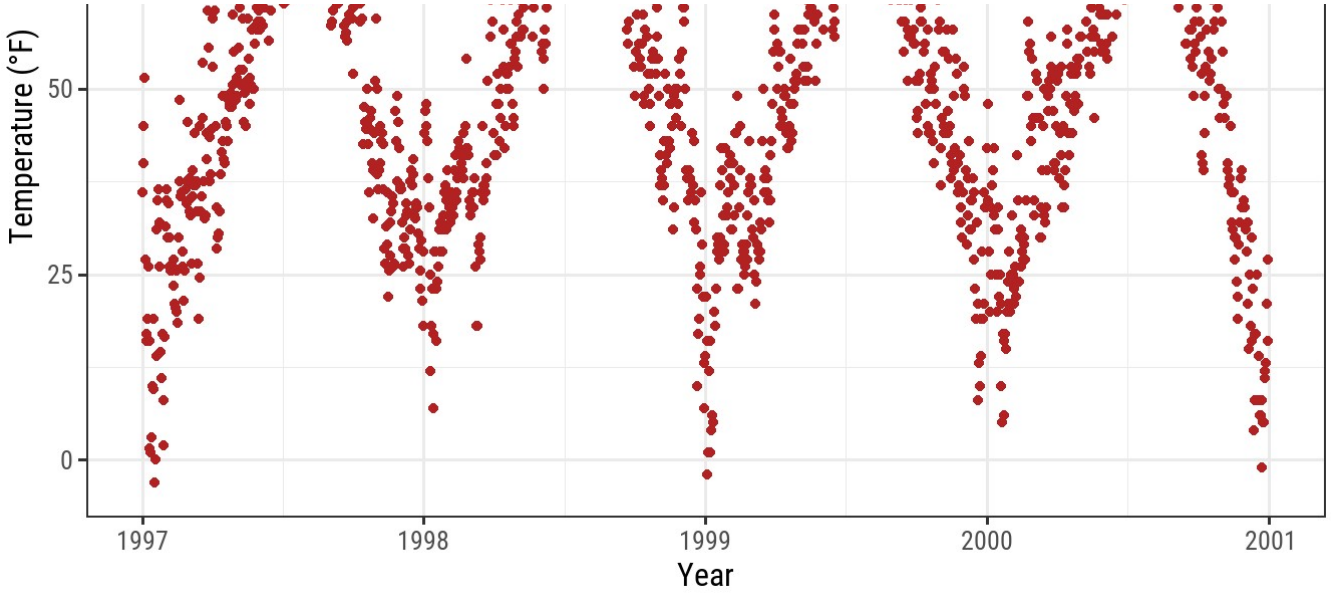
(Previously, this tutorial used the {extrafont} package (https://cran.r-project.org/web/packages/extrafont/README.html), which did a great job until last year. All of the sudden I couldn't add any new fonts anymore and after getting a new laptop, the package did not find any fonts at all... I usually suggest the {ragg} package (https://ragg.r-lib.org/) now. However, I did not succeed to make it work for my homepage so I use the `{showtext}` package which is great as well with the only main difference that you need to import the font you want to use explicitly with `{showtext}`. (However, it seems there are some technical details that are not solved optimally by `{showtext}` (https://twitter.com/thomasp85/status/1355083725156077571) so you may want to use the package as a very last resort.)

CHANGE SPACING IN MULTI-LINE TEXT

You can use the `lineheight` argument to change the spacing between lines. In this example, I have squished 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))
```



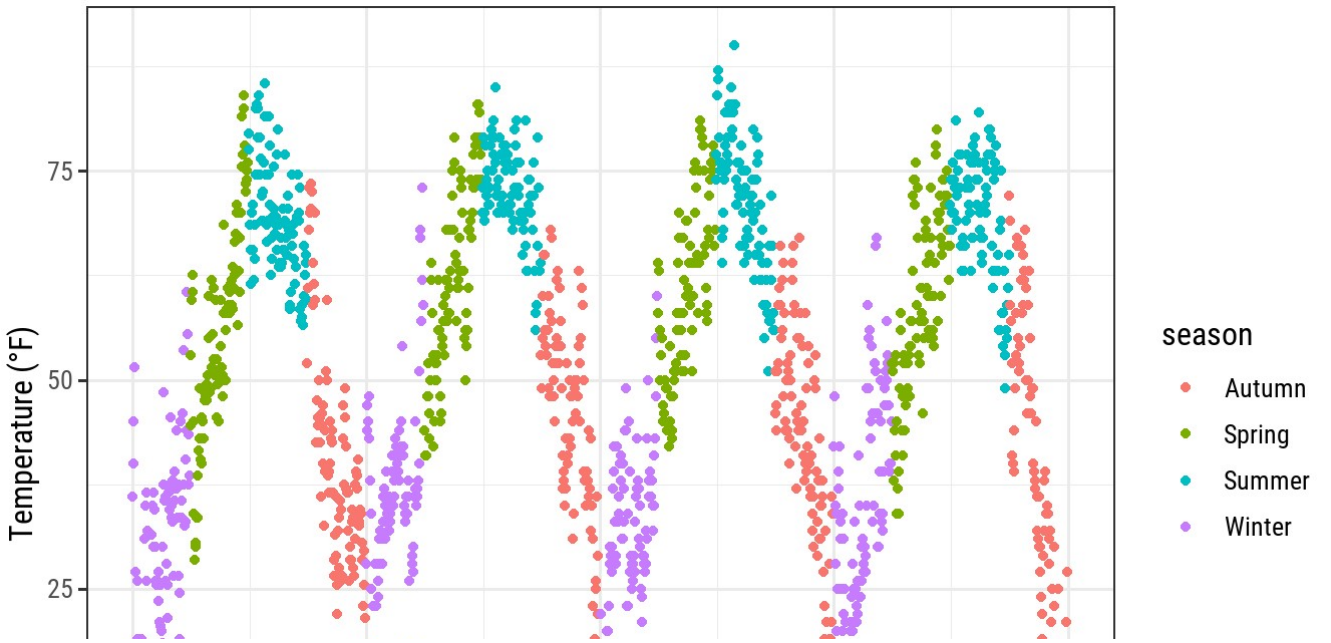


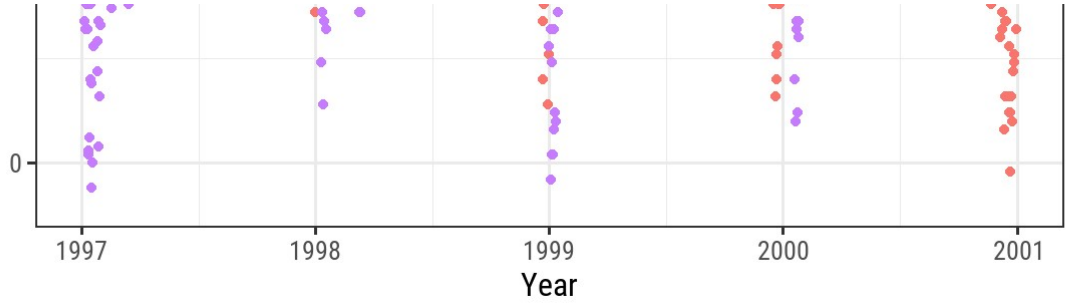
↑ Jump back to Table of Content.

WORKING WITH LEGENDS

We will color code the plot based on season. Or to phrase it in a more ggplot'ish way: we map the variable `season` to the `aes`thetic `color`. One nice thing about `{ggplot2}` is that it adds a legend by default when mapping a variable to an aesthetic. You can see that by default the legend title is what we specified in the color argument:

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



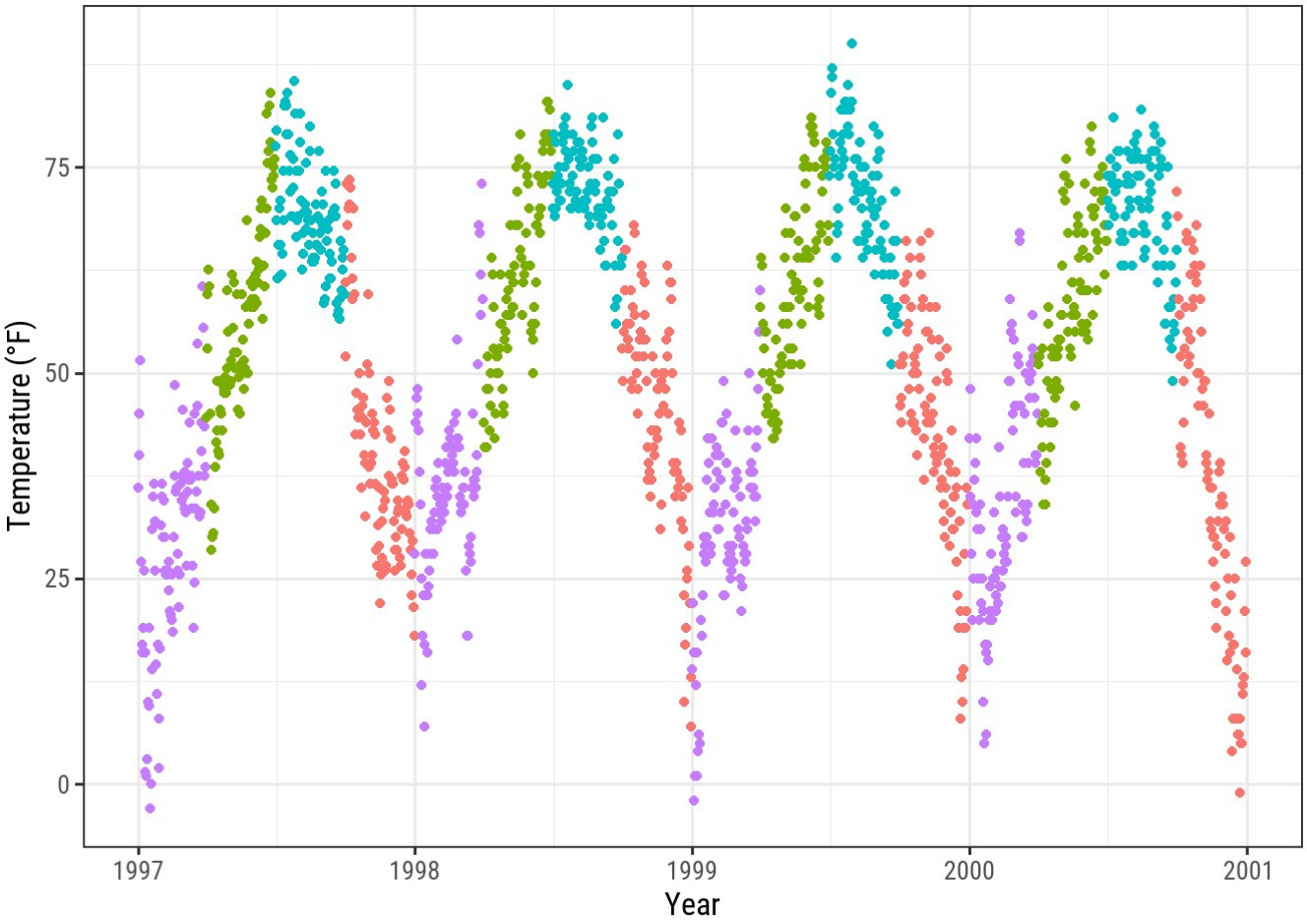


TURN OFF THE LEGEND

Always one of the first question is: “How can I get rid of the legend?”.

It is quite easy and always works with `theme(legend.position = "none")`:

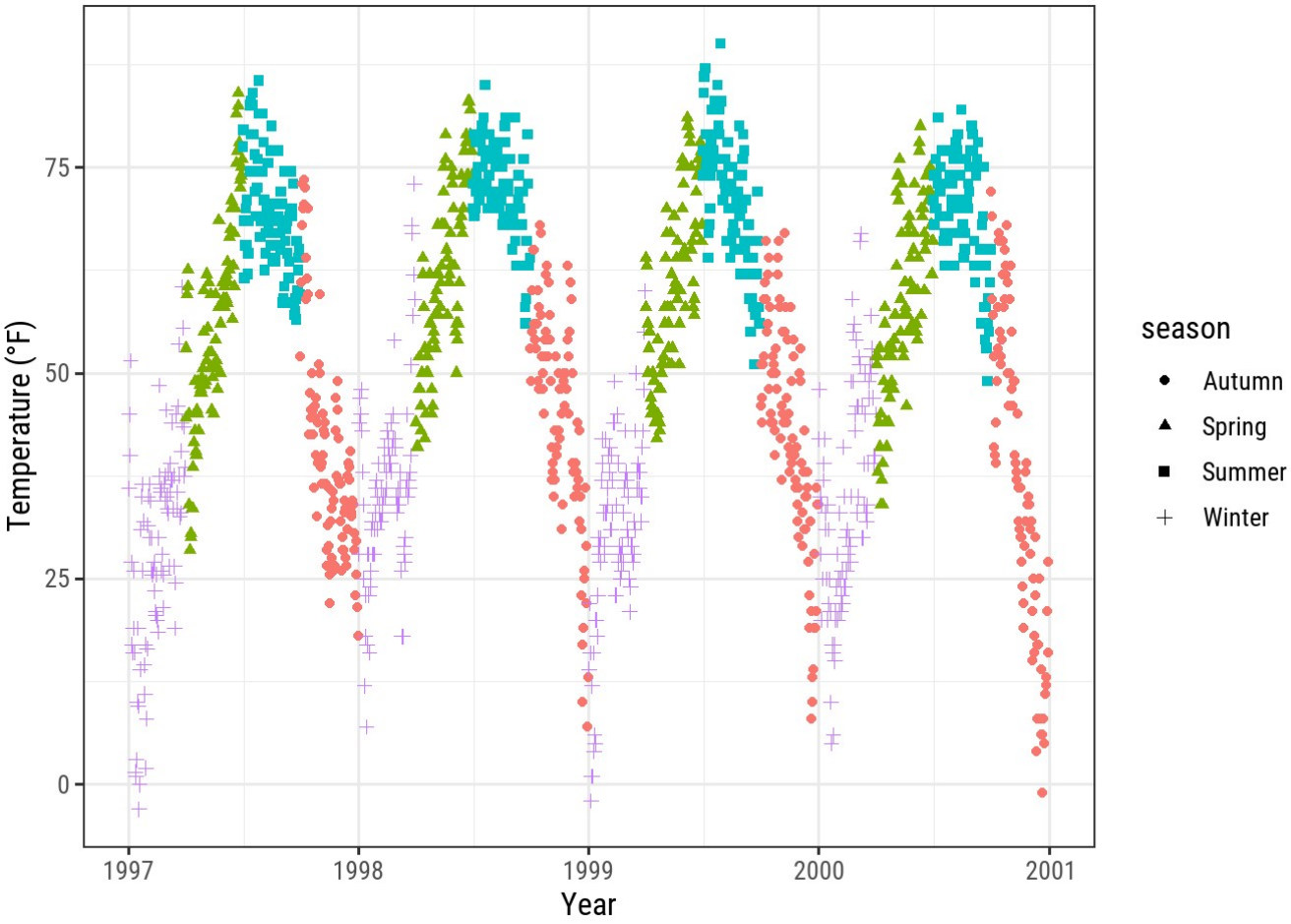
```
ggplot(chic,
       aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(legend.position = "none")
```



You can also use `guides(color = "none")` or `scale_color_discrete(guide = "none")` depending on the specific case. While the change of the theme element removes all legends at once, you can

remove particular legends with the latter options while keeping some others:

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

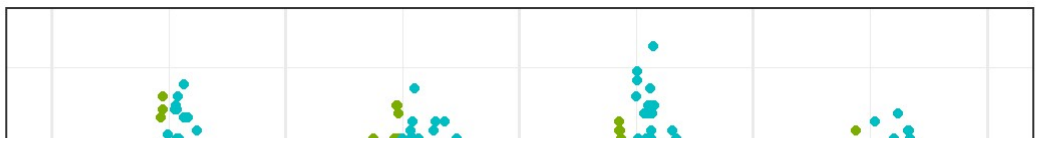


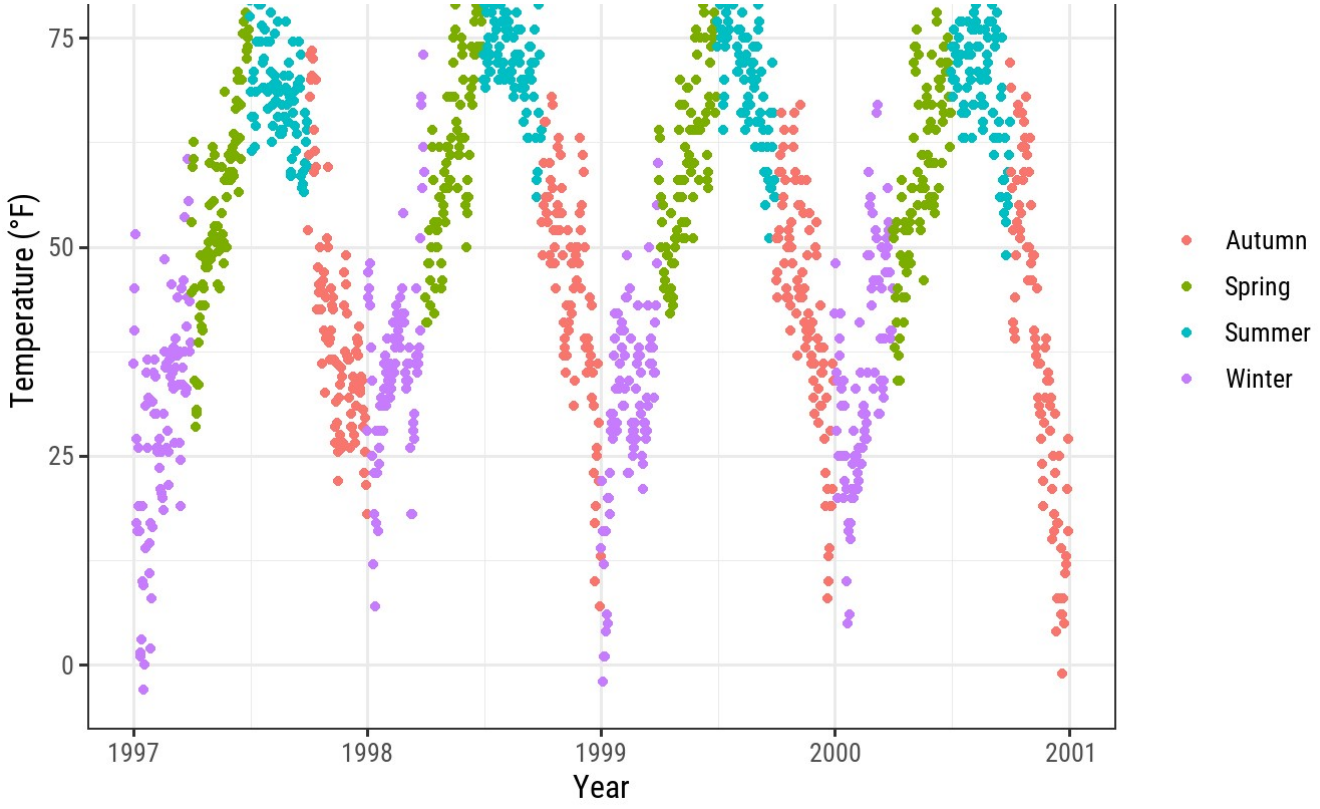
Here, for example, we keep the legend for the shapes while discarding the one for the colors.

REMOVE LEGEND TITLES

As we already learned, use `element_blank()` to draw *nothing*:

```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(legend.title = element_blank())
```



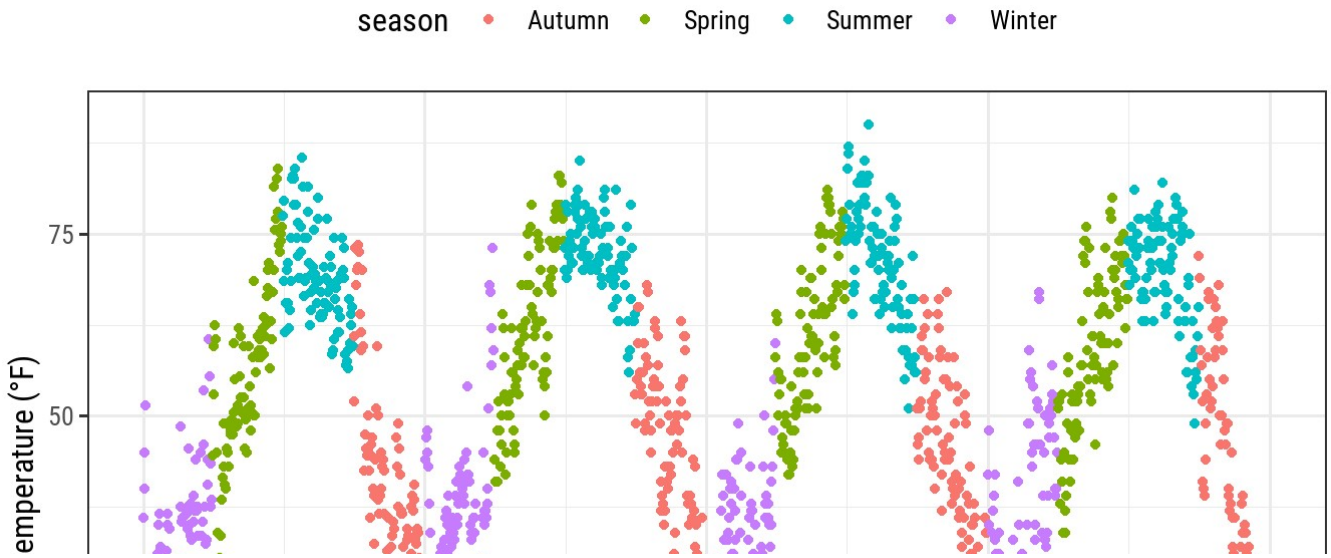


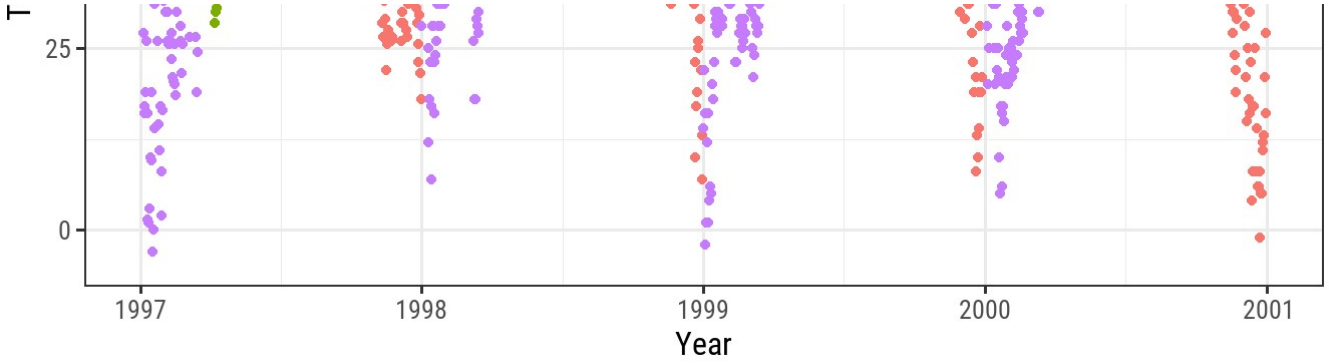
🤷 *You can achieve the same by setting the legend name to `NULL`, either via `scale_color_discrete(name = NULL)` or `labs(color = NULL)`. Expand to see examples.*

CHANGE LEGEND POSITION

If you want to place the legend not on the right, one uses `legend.position` as argument in `theme`. Possible positions are “top”, “right” (which is the default), “bottom”, and “left”.

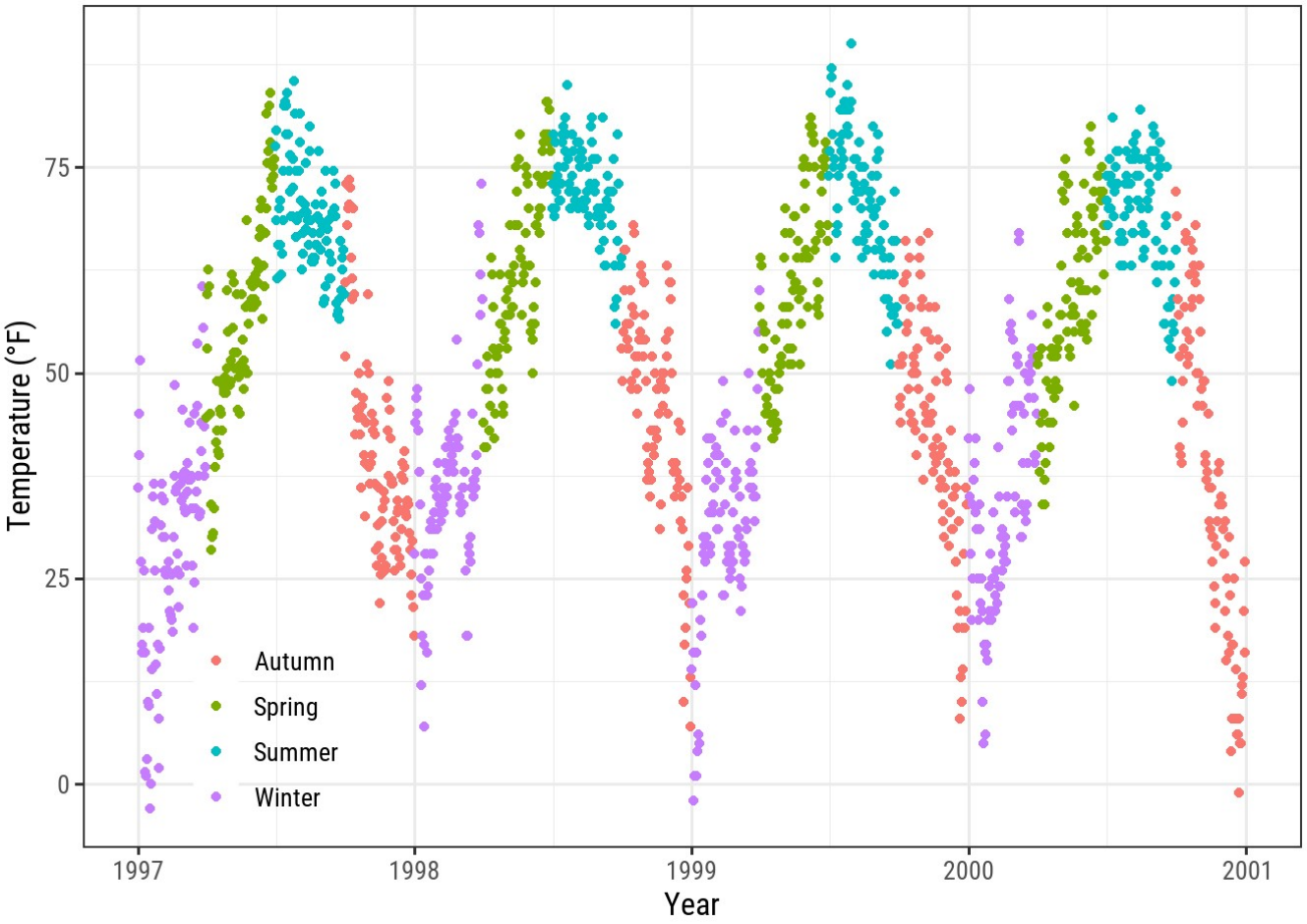
```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(legend.position = "top")
```





You can also place 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"))
```

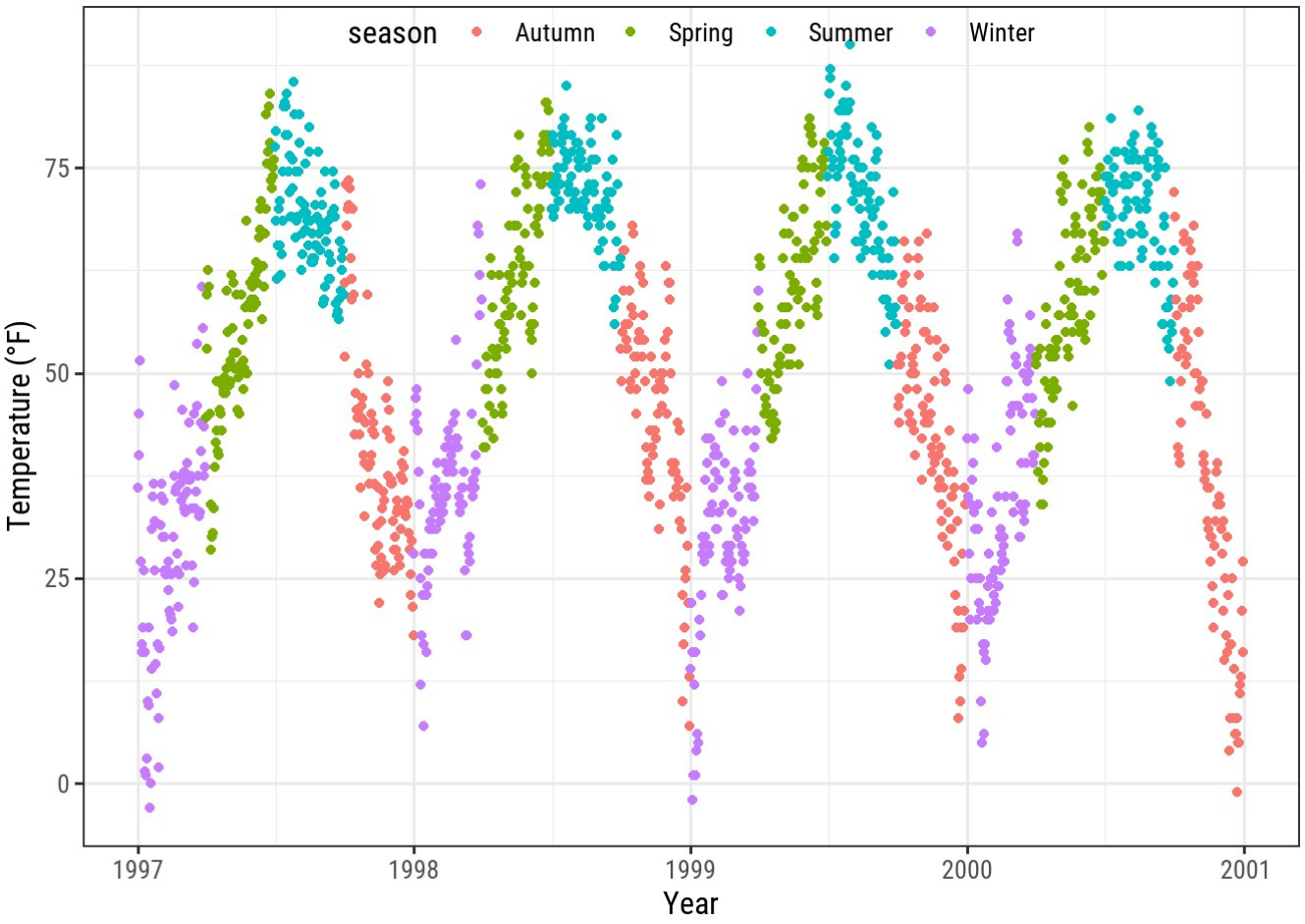


Here, I also overwrite the default white legend background with a transparent fill to make sure the legend does not hide any data points.

CHANGE LEGEND DIRECTION

As you have seen, the legend direction is by default vertical but horizontal when you choose either the “top” or “bottom” position. But you can also switch the direction as you like:

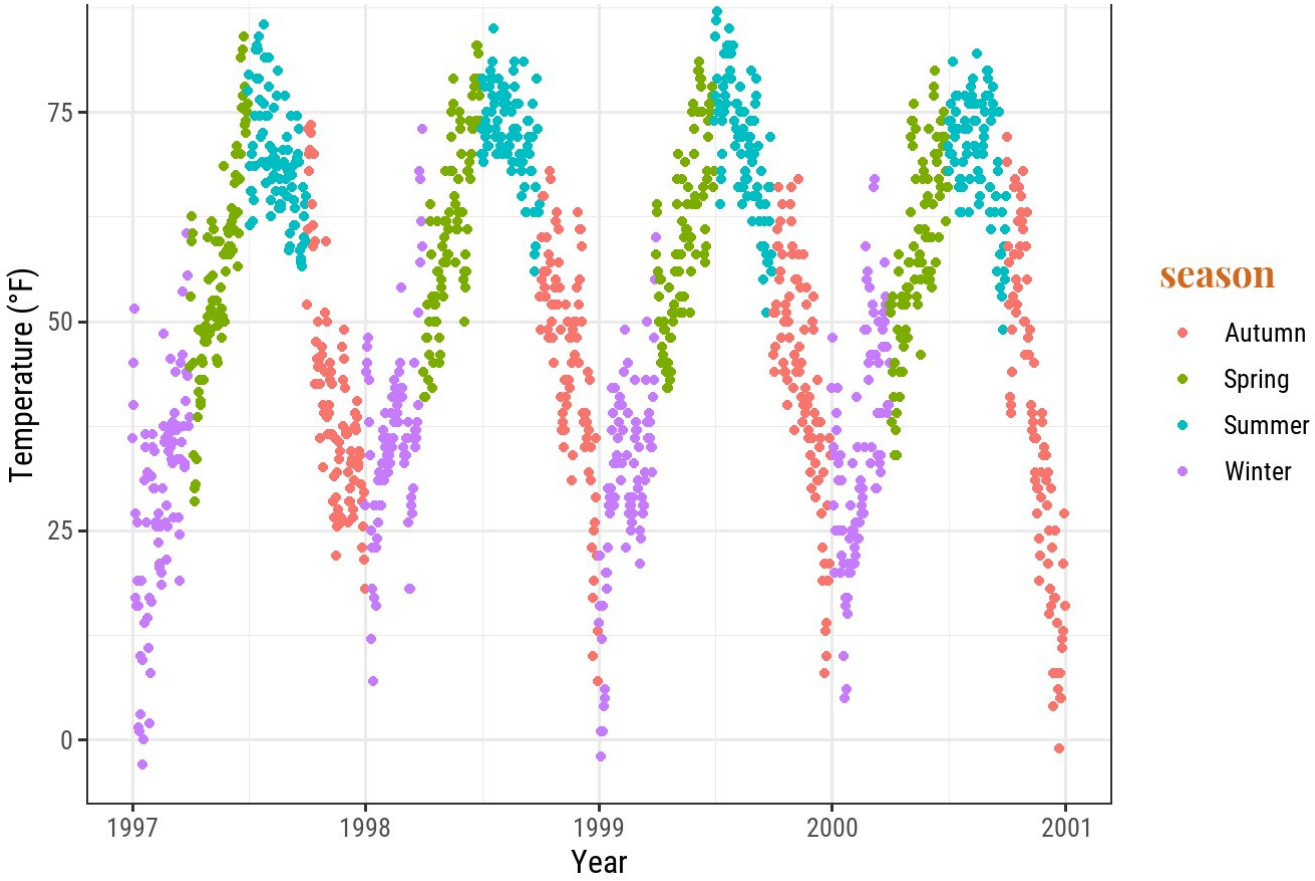
```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(legend.position = c(.5, .97),
        legend.background = element_rect(fill = "transparent")) +
  guides(color = guide_legend(direction = "horizontal"))
```



CHANGE STYLE OF THE LEGEND TITLE

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

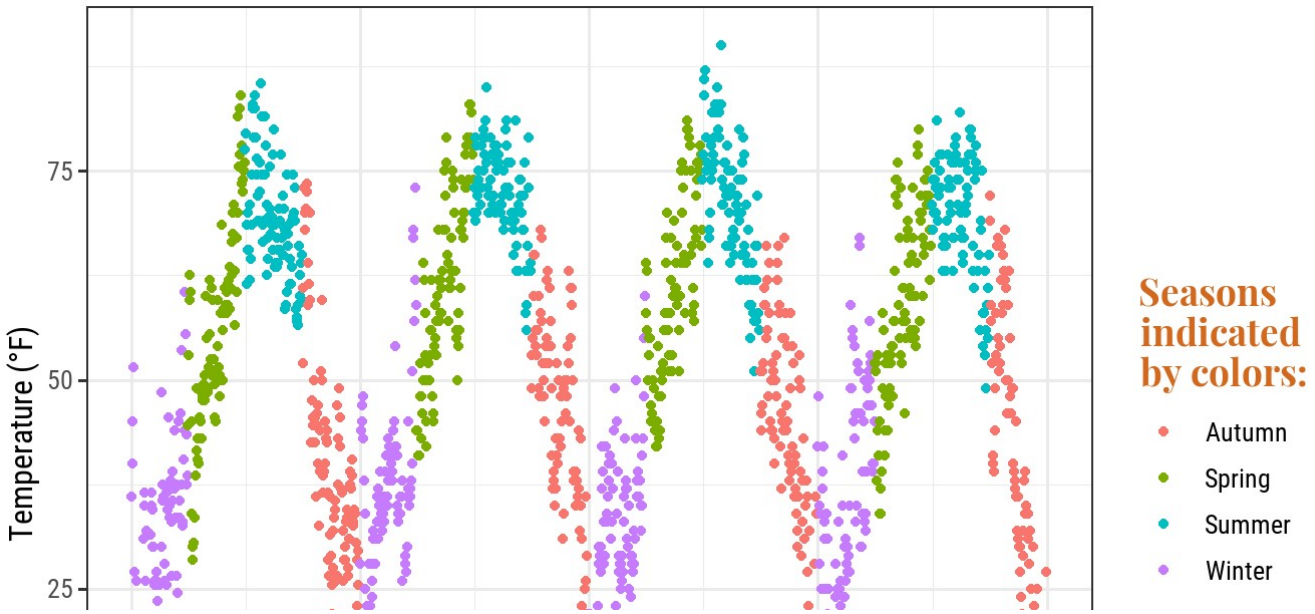
```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(legend.title = element_text(family = "Playfair",
                                    color = "chocolate",
                                    size = 14, face = "bold"))
```

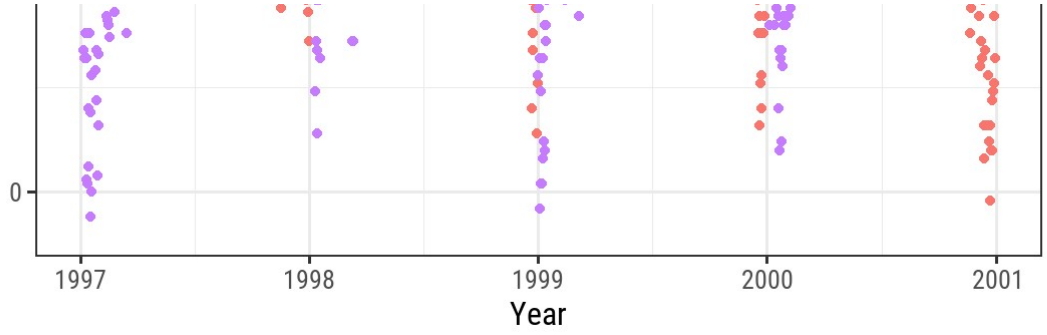


CHANGE LEGEND TITLE

The easiest way to change the title of the legend is 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",
                                    color = "chocolate",
                                    size = 14, face = "bold"))
```





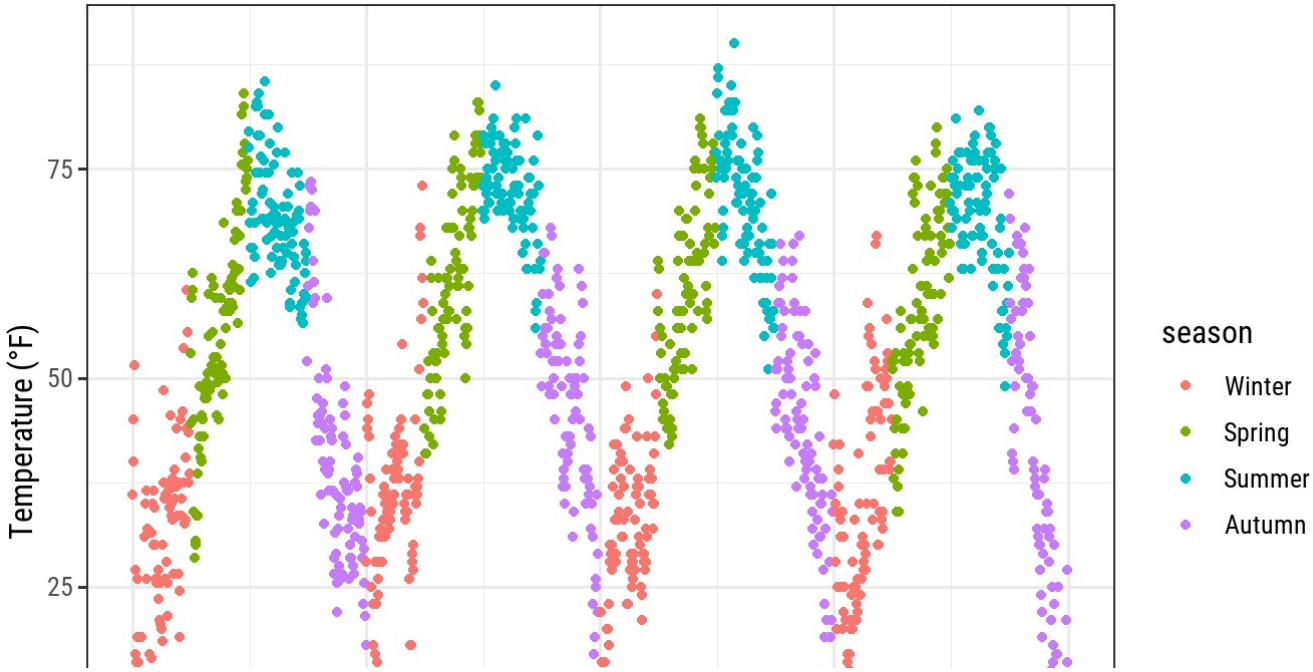
The legend details can be changed via `scale_color_discrete(name = "title")` or `guides(color = guide_legend("title"))`:

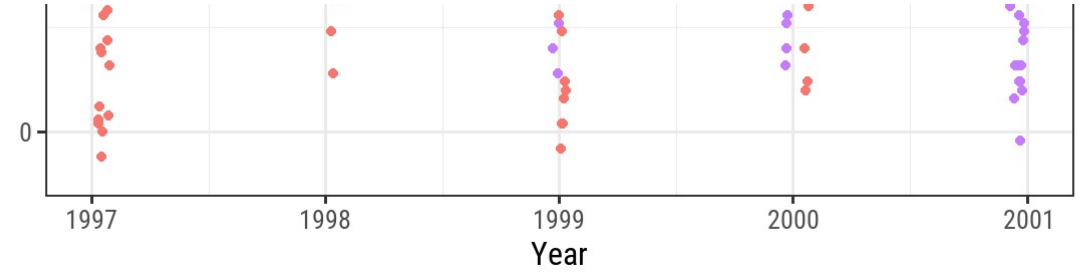
```
ggplot(chic, aes(x = date, y = temp, color = season))) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(legend.title = element_text(family = "Playfair",
                                    color = "chocolate",
                                    size = 14, face = "bold")) +
  scale_color_discrete(name = "Seasons\nindicated\nby colors:")
```

CHANGE ORDER OF LEGEND KEYS

We can achieve this 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)")
```

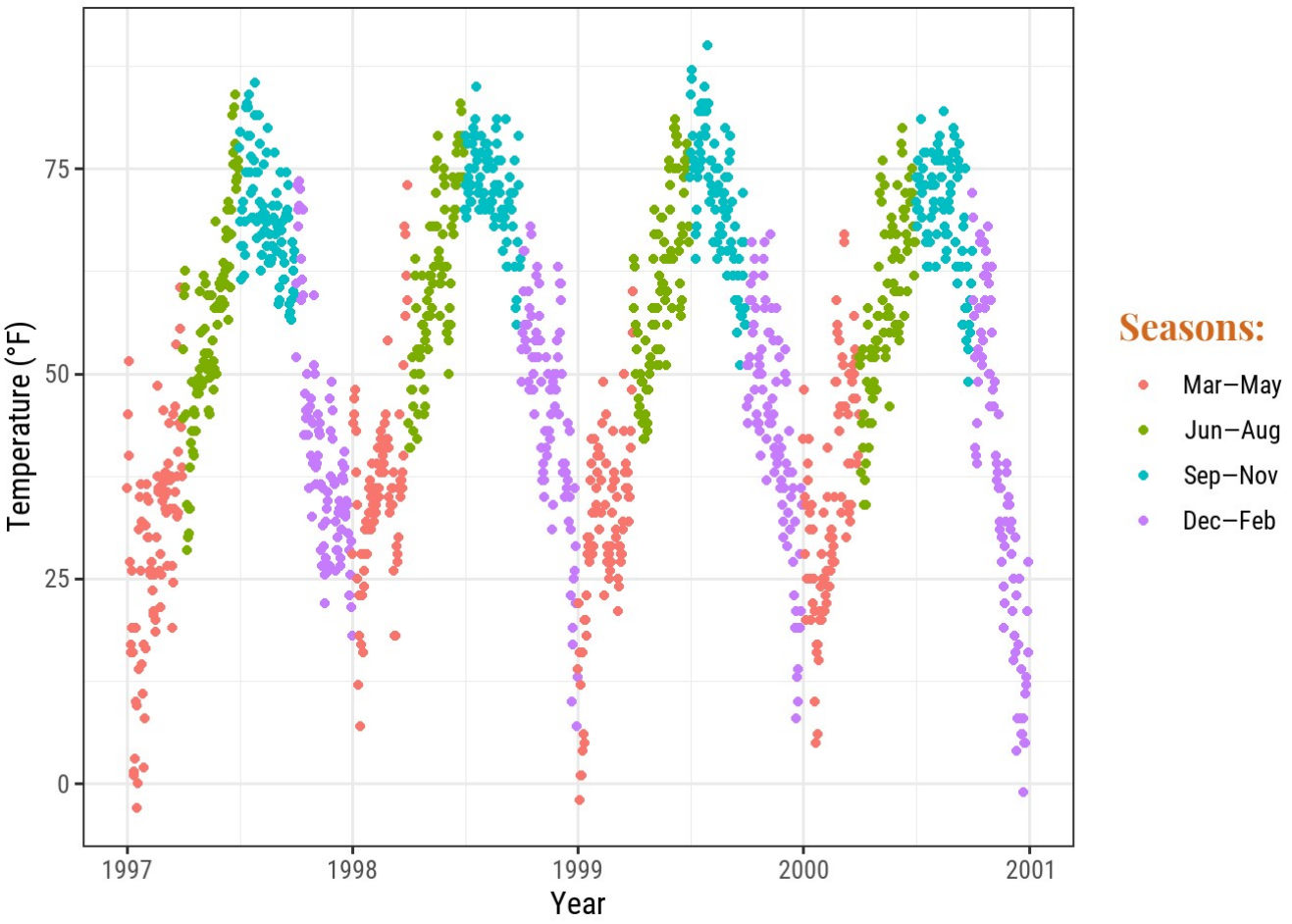




CHANGE LEGEND LABELS

We are going to replace the seasons by the months which they are covering by providing a vector of names in the `scale_color_discrete()` call:

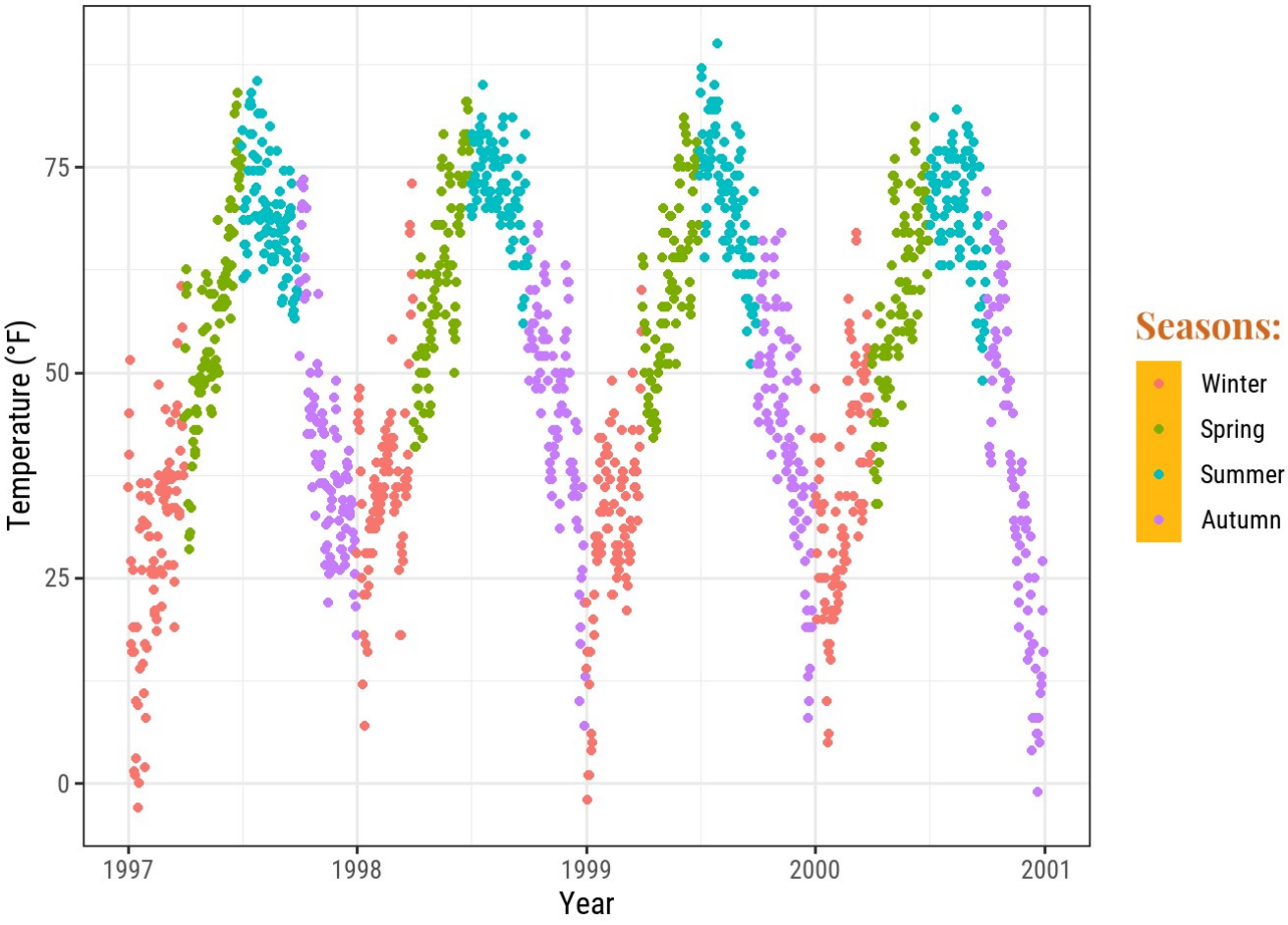
```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  scale_color_discrete(
    name = "Seasons:",
    labels = c("Mar–May", "Jun–Aug", "Sep–Nov", "Dec–Feb")
  ) +
  theme(legend.title = element_text(
    family = "Playfair", color = "chocolate", size = 14, face = 2
  ))
```



CHANGE BACKGROUND BOXES IN THE LEGEND

To change the background color (fill) of the legend keys, we adjust 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",
                                    color = "chocolate",
                                    size = 14, face = 2)) +
  scale_color_discrete("Seasons:")
```

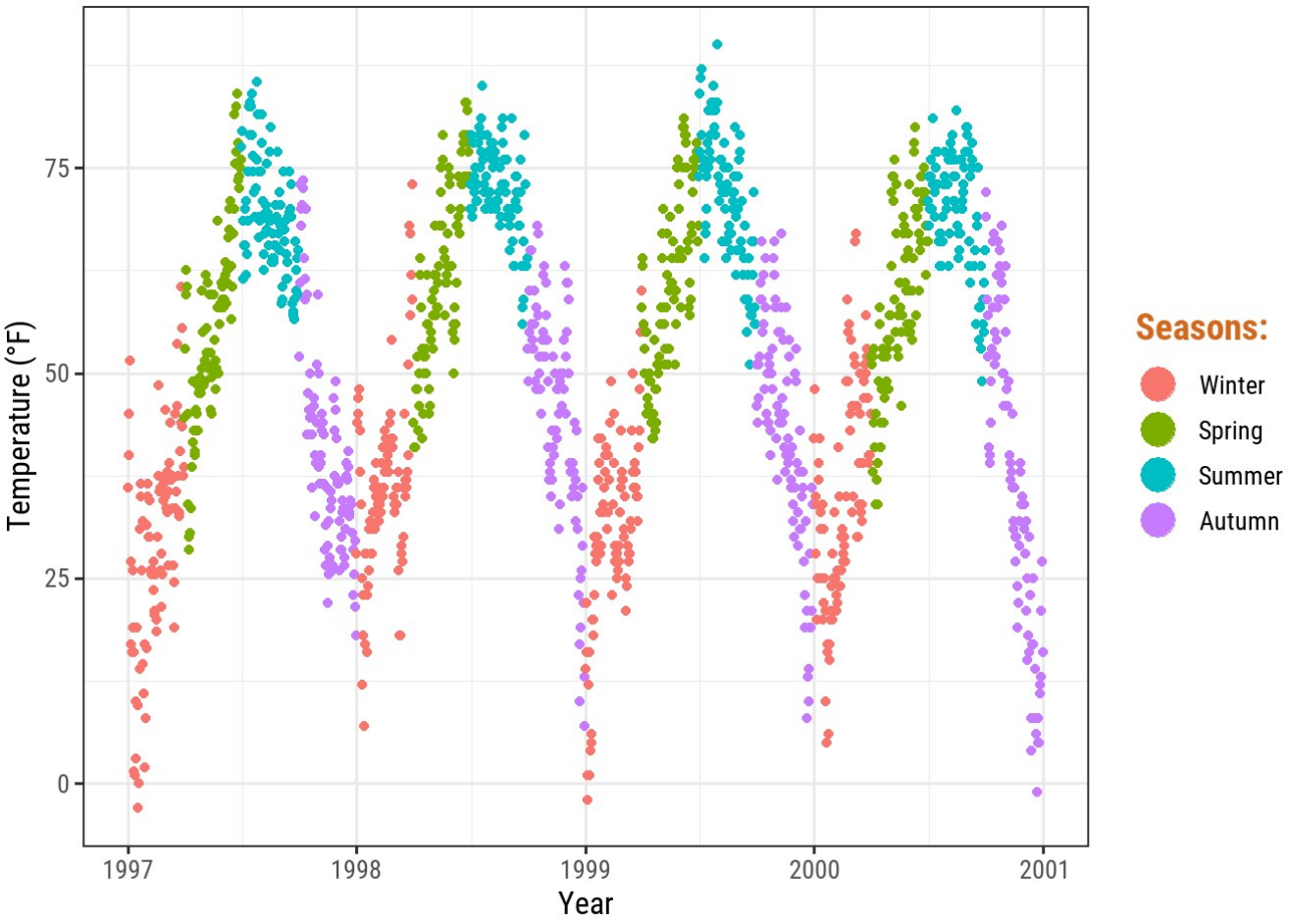


If you want to get rid of them entirely use `fill = NA` or `fill = "transparent"`.

CHANGE SIZE OF LEGEND SYMBOLS

Points in the legend can get a little lost with the default size, especially without the boxes. To override the default one uses again the `guides` layer like this:

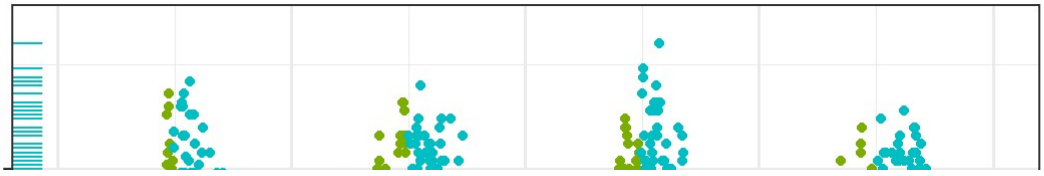
```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(legend.key = element_rect(fill = NA),
        legend.title = element_text(color = "chocolate",
                                    size = 14, face = 2)) +
  scale_color_discrete("Seasons:") +
  guides(color = guide_legend(override.aes = list(size = 6)))
```

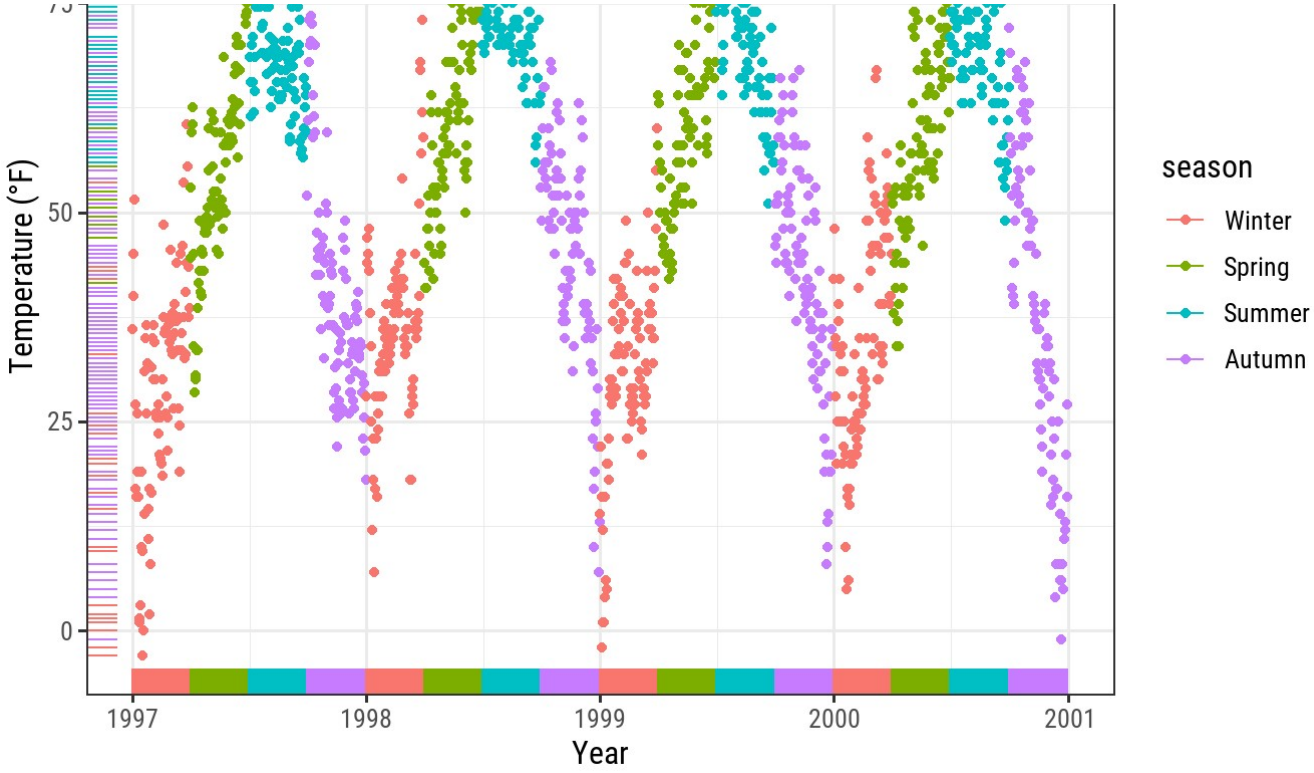


LEAVE A LAYER OFF THE LEGEND

Let's say you have two different geoms mapped to the same variable. For example, color 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:

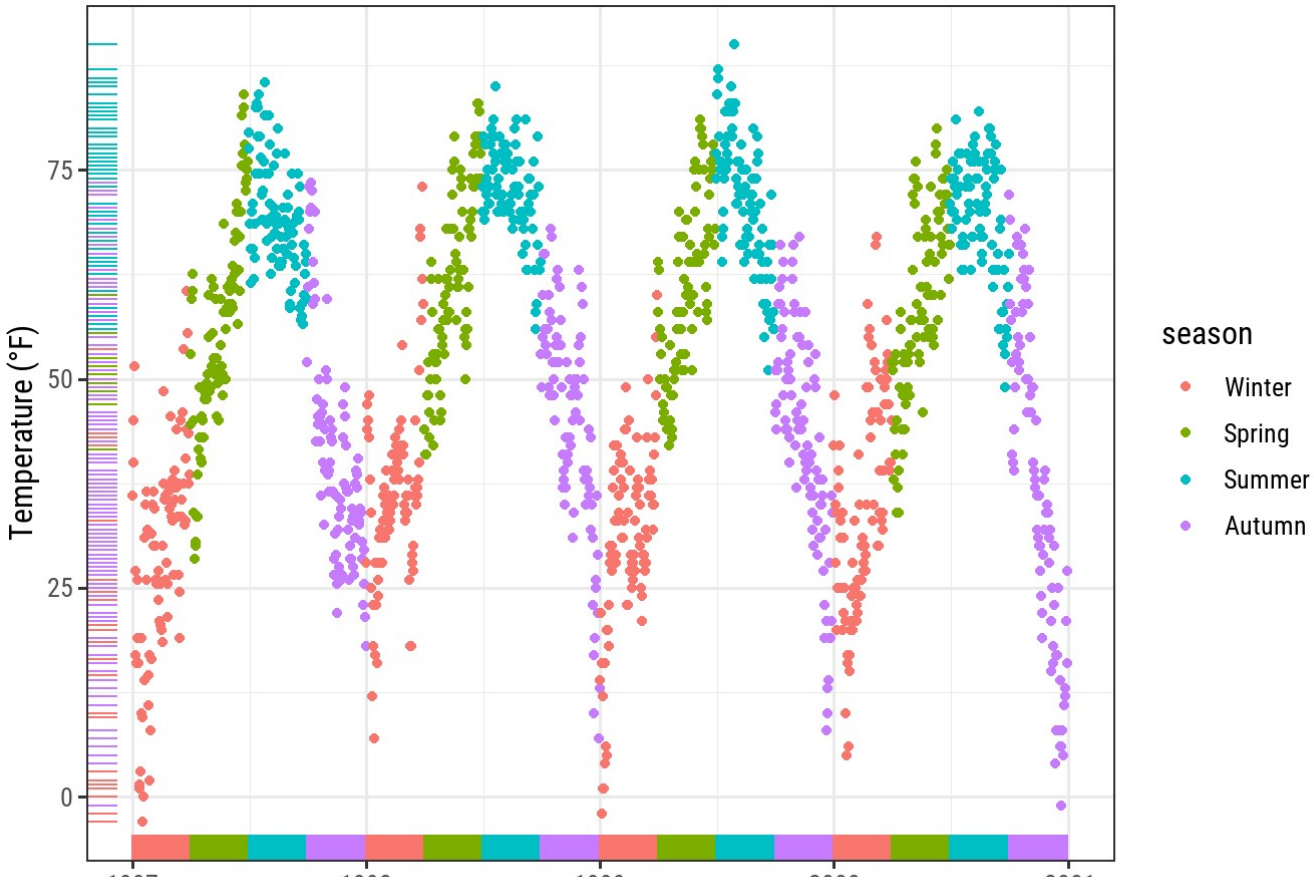
```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  geom_rug()
```





You can use `show.legend = FALSE` to turn off a layer in the legend:

```
ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  geom_rug(show.legend = FALSE)
```

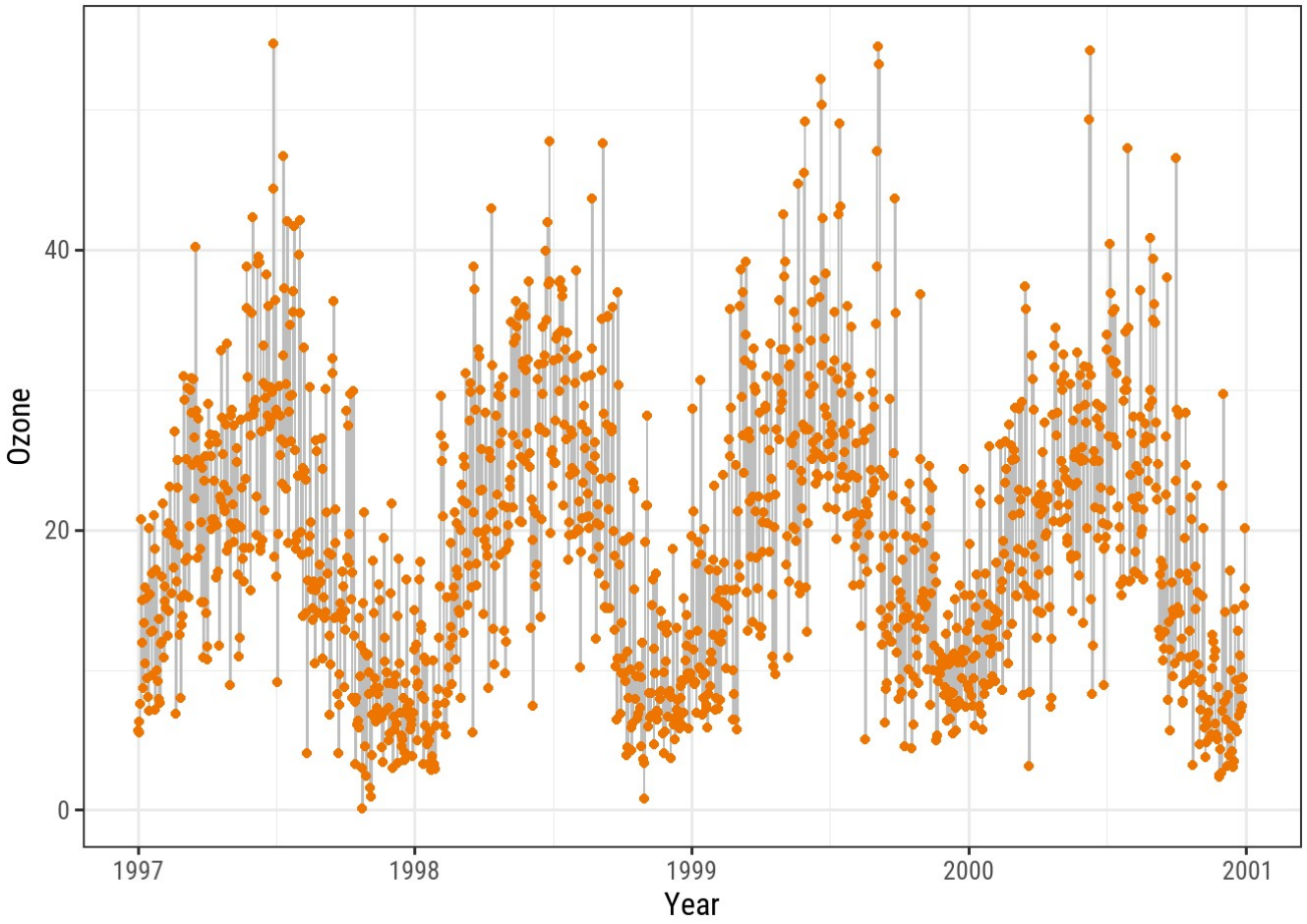


MANUALLY ADDING LEGEND ITEMS

`{ggplot2}` will not add a legend automatically unless you map aesthetics (color, size etc.) to a variable. There are times, though, that I want to have a legend so that it is clear what you are plotting.

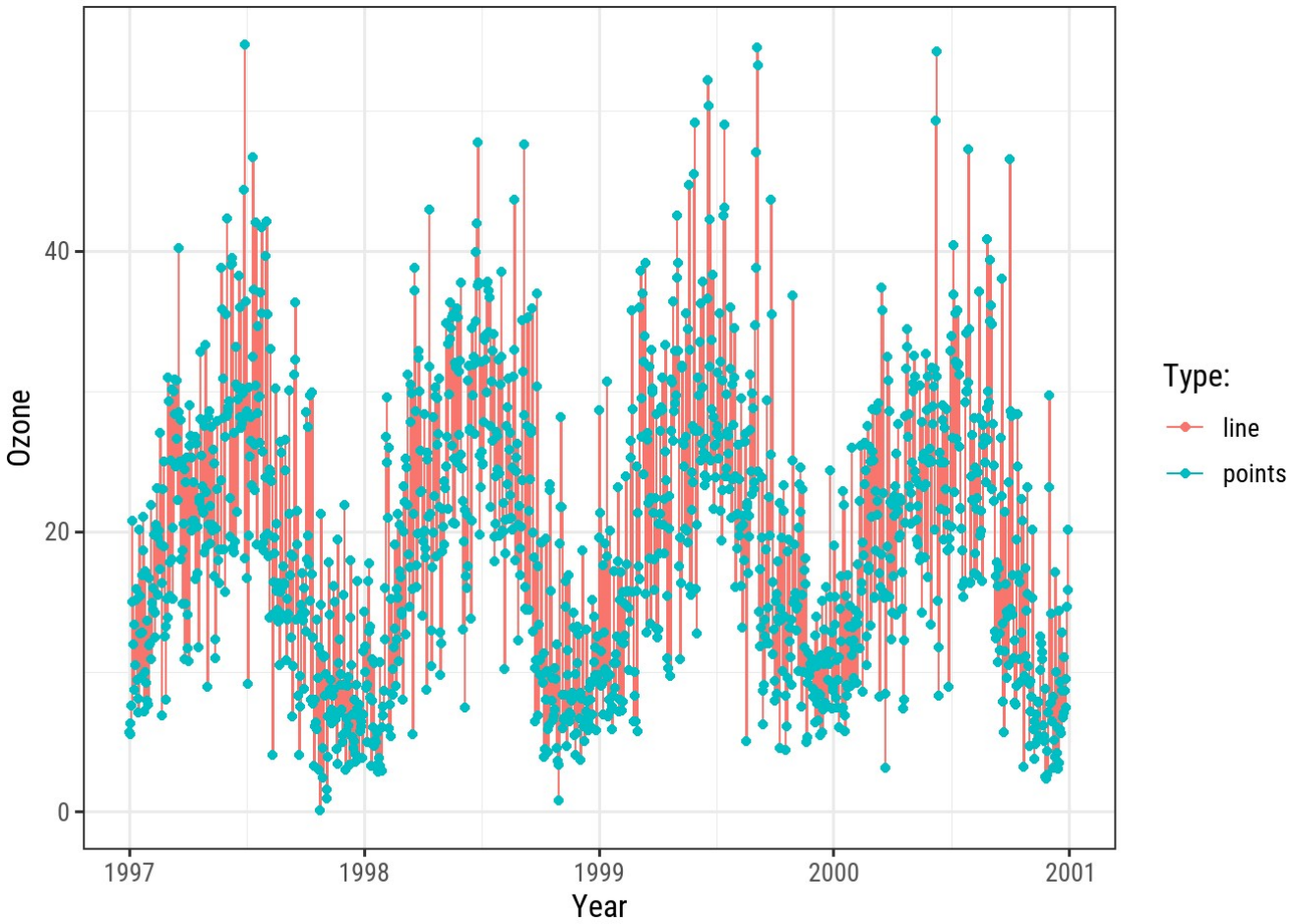
Here is the default:

```
ggplot(chic, aes(x = date, y = o3)) +
  geom_line(color = "gray") +
  geom_point(color = "darkorange2") +
  labs(x = "Year", y = "Ozone")
```



We can force a legend by mapping a guide to a *variable*. We are mapping the lines and the points using `aes()` and we are mapping **not** to a variable in our dataset but to a single string (so that we get just one color for each).

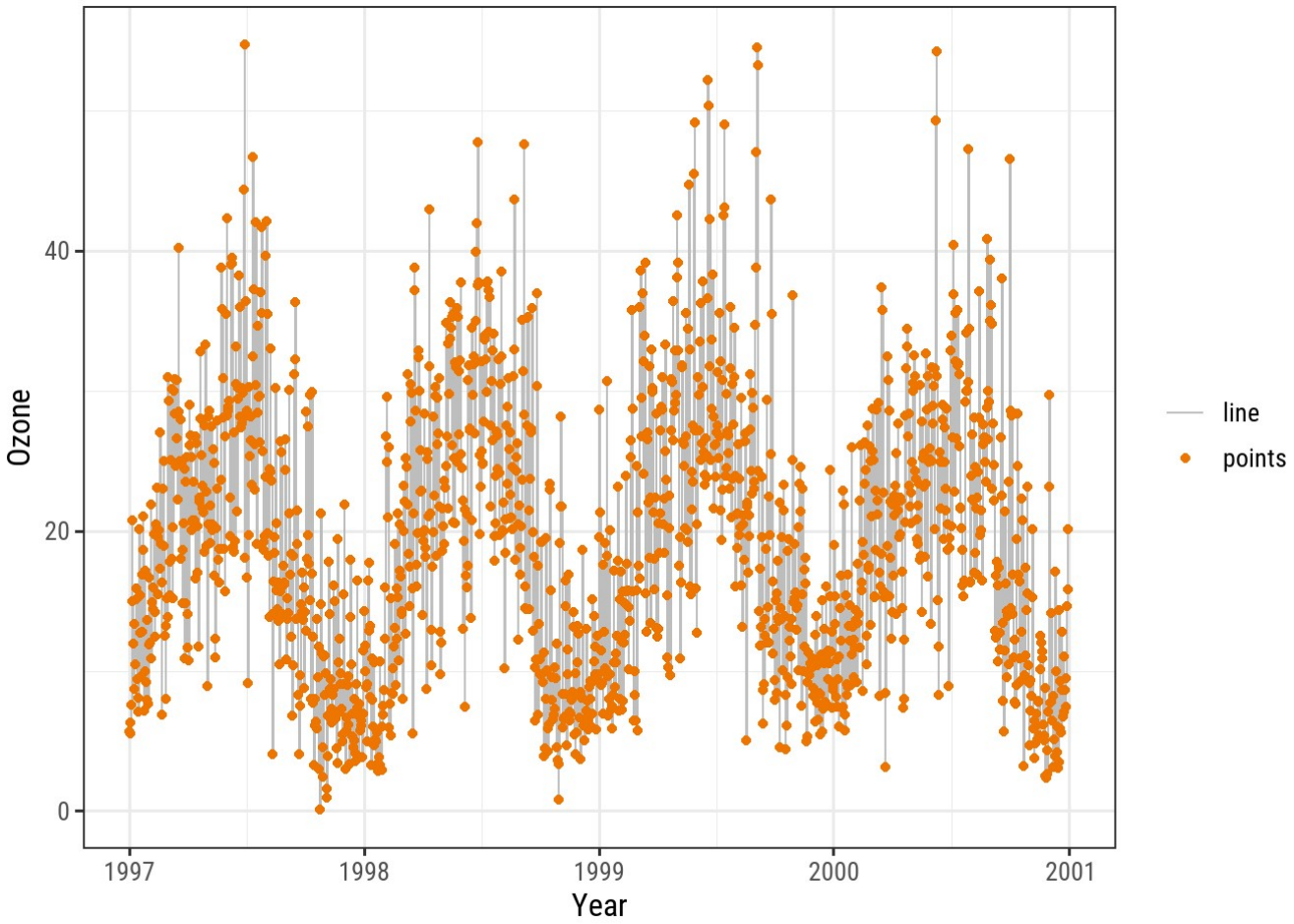
```
ggplot(chic, aes(x = date, y = o3)) +
  geom_line(aes(color = "line")) +
  geom_point(aes(color = "points")) +
  labs(x = "Year", y = "Ozone") +
  scale_color_discrete("Type:")
```



We are getting close but this is not what we want. We want gray and red! To change the color, we use `scale_color_manual()`. Additionally, we override the legend aesthetics using the `guide()` function.

Voila! Now, we have a plot with gray lines and red pints as well as a single gray line and a single red point as legend symbols:

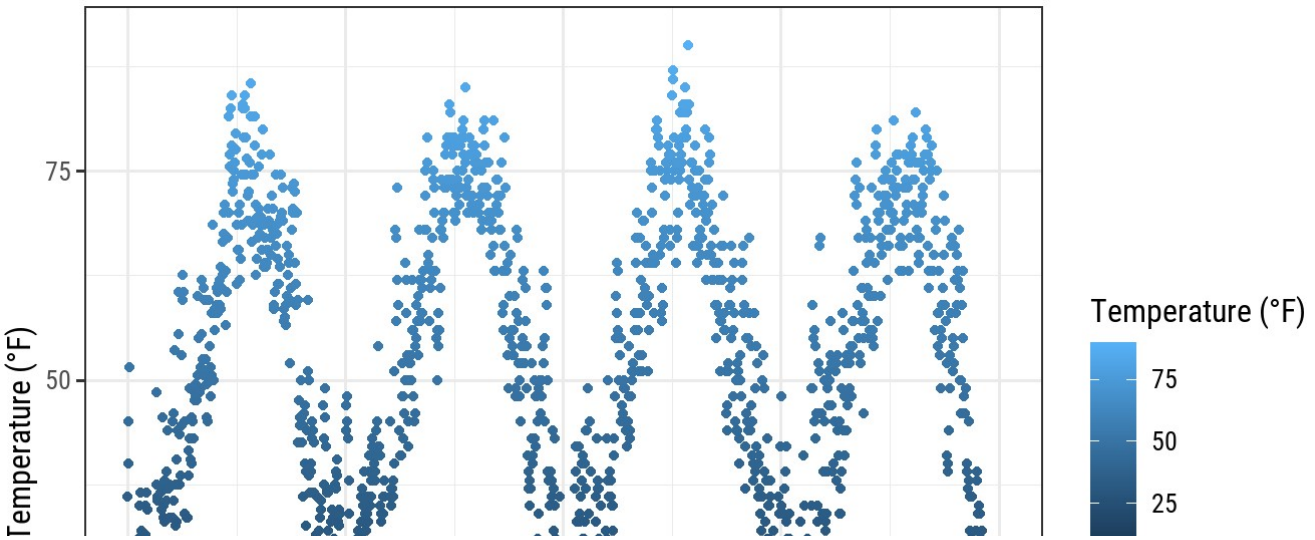
```
ggplot(chic, aes(x = date, y = o3)) +
  geom_line(aes(color = "line")) +
  geom_point(aes(color = "points")) +
  labs(x = "Year", y = "Ozone") +
  scale_color_manual(name = NULL,
                     guide = "legend",
                     values = c("points" = "darkorange2",
                                "line" = "gray")) +
  guides(color = guide_legend(override.aes = list(linetype = c(1, 0),
                                                  shape = c(NA, 16))))
```

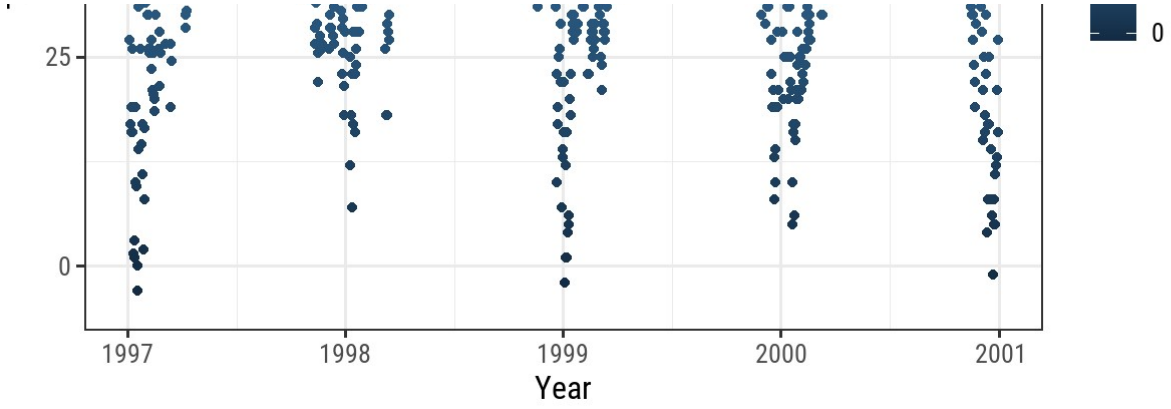


USE OTHER LEGEND STYLES

The default legend for categorical variables such as `season` is a `guide_legend()` as you have seen in several previous examples. If you map a continuous variable to an aesthetic, `{ggplot2}` will by default not use `guide_legend()` but `guide_colorbar()` (or `guide_colourbar()`):

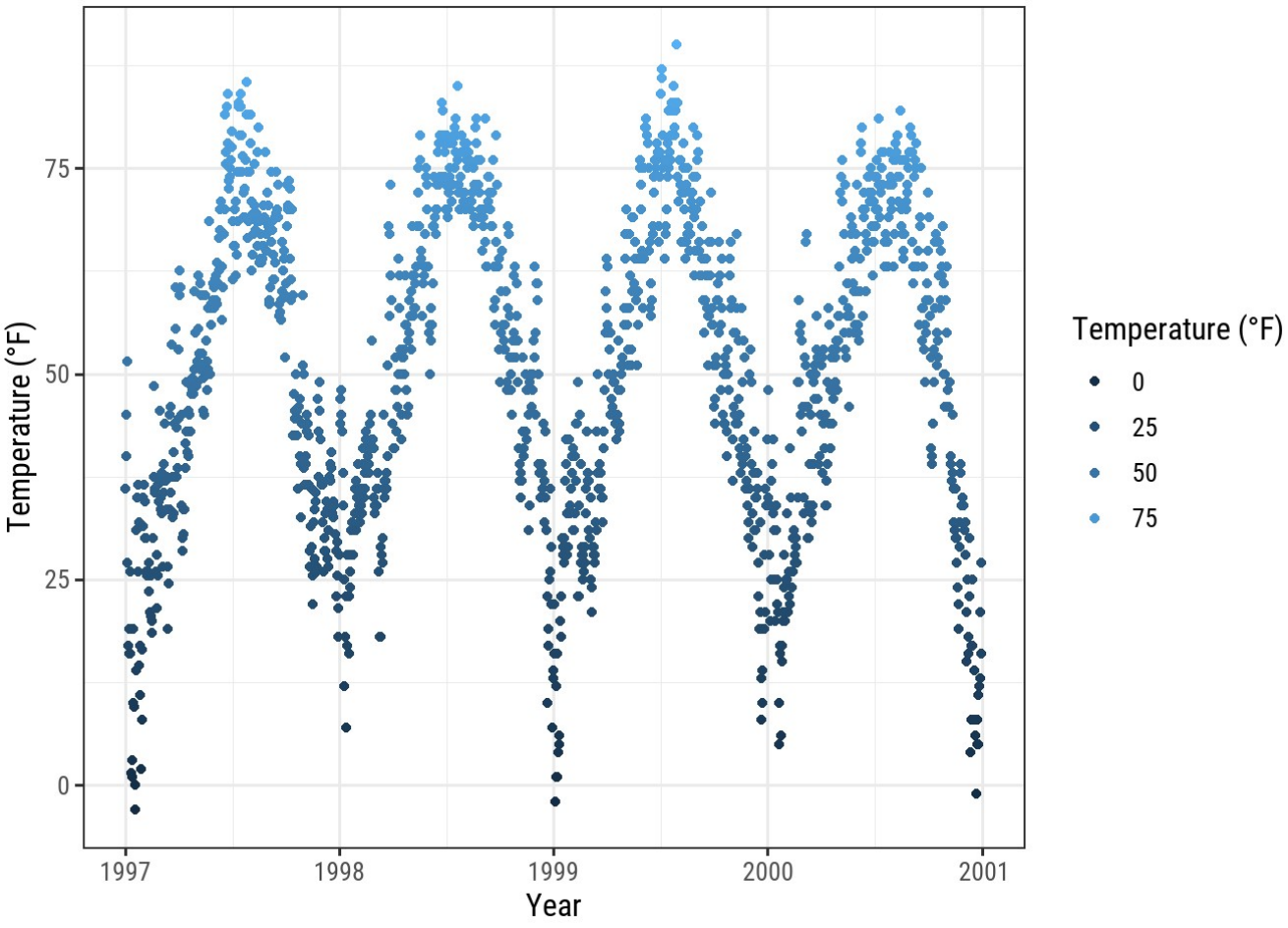
```
ggplot(chic,
       aes(x = date, y = temp, color = temp)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F)")
```





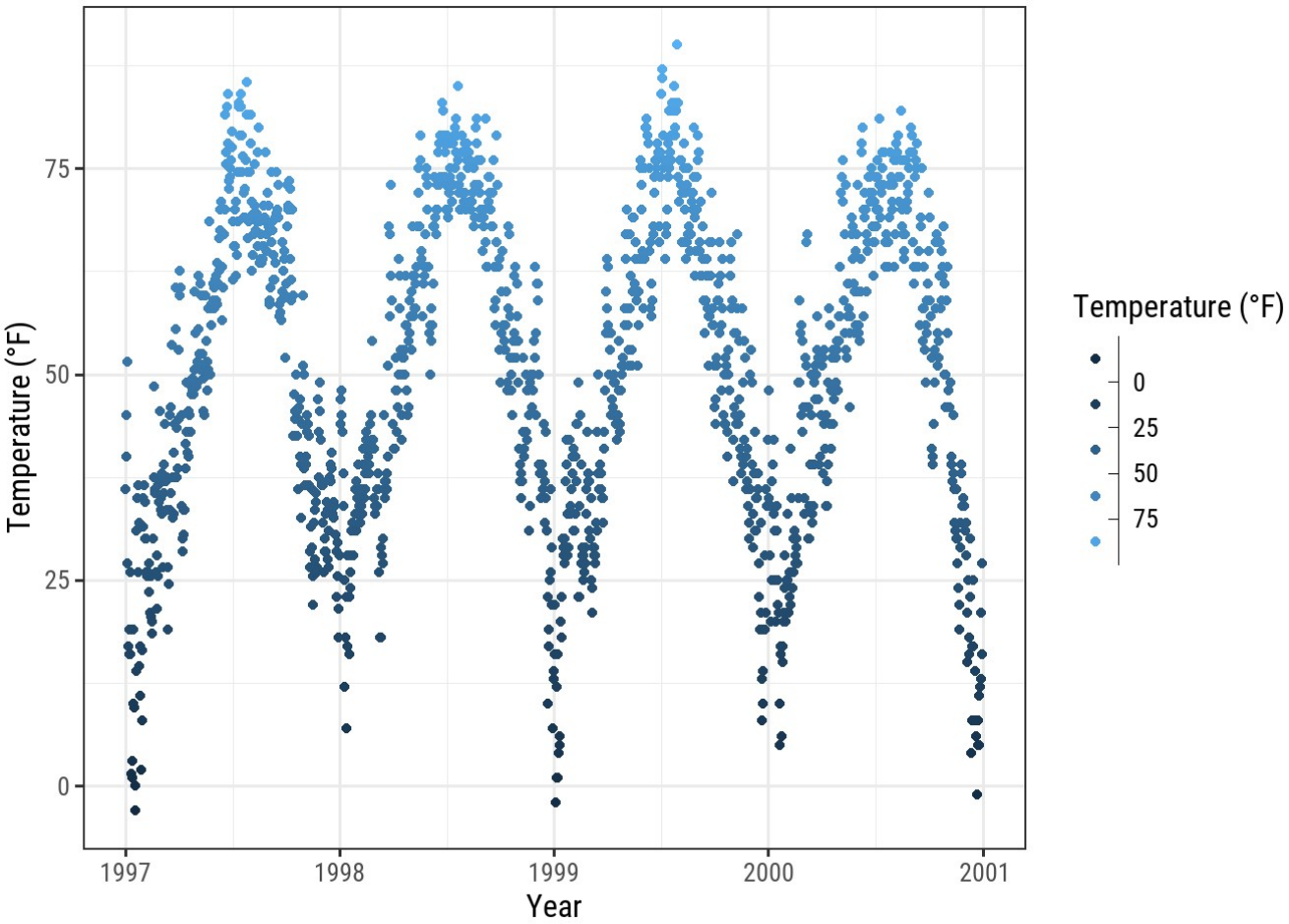
However, by using `guide_legend()` you can force the legend to show discrete colors for a given number of breaks as in case of a categorical variable:

```
ggplot(chic,
       aes(x = date, y = temp, color = temp)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F)") +
  guides(color = guide_legend())
```



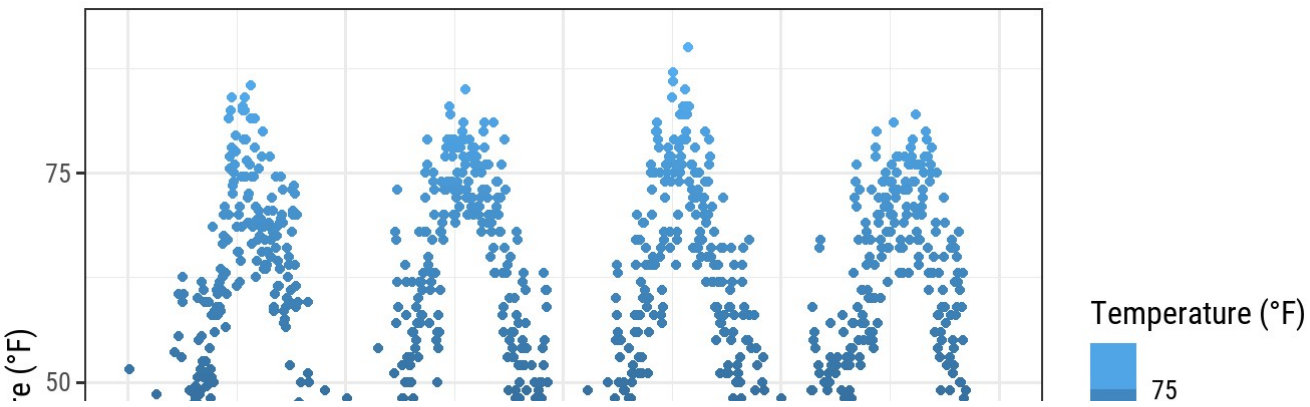
You can also use *binned scales*:

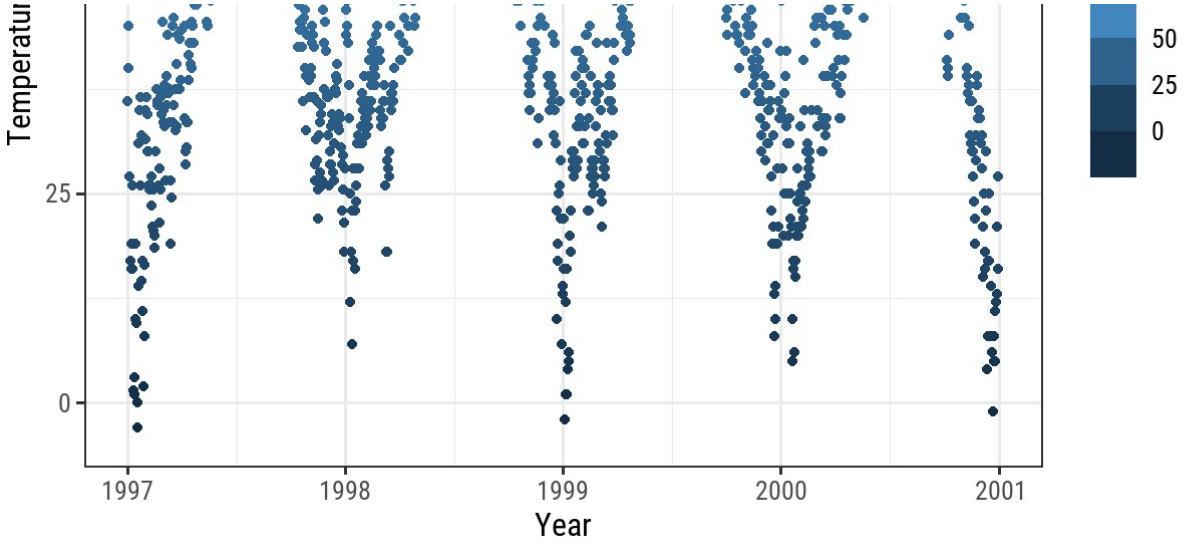
```
ggplot(chic,
       aes(x = date, y = temp, color = temp)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)", color = "Temperature (°F)") +
  guides(color = guide_bins())
```



… 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())
```





↑ Jump back to Table of Content.

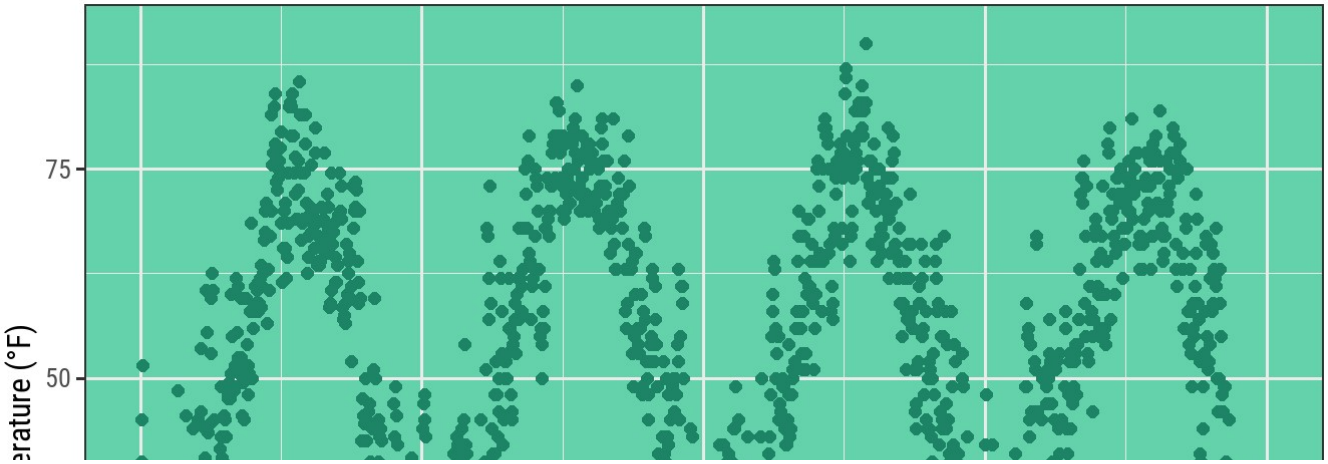
WORKING WITH BACKGROUNDS & GRID LINES

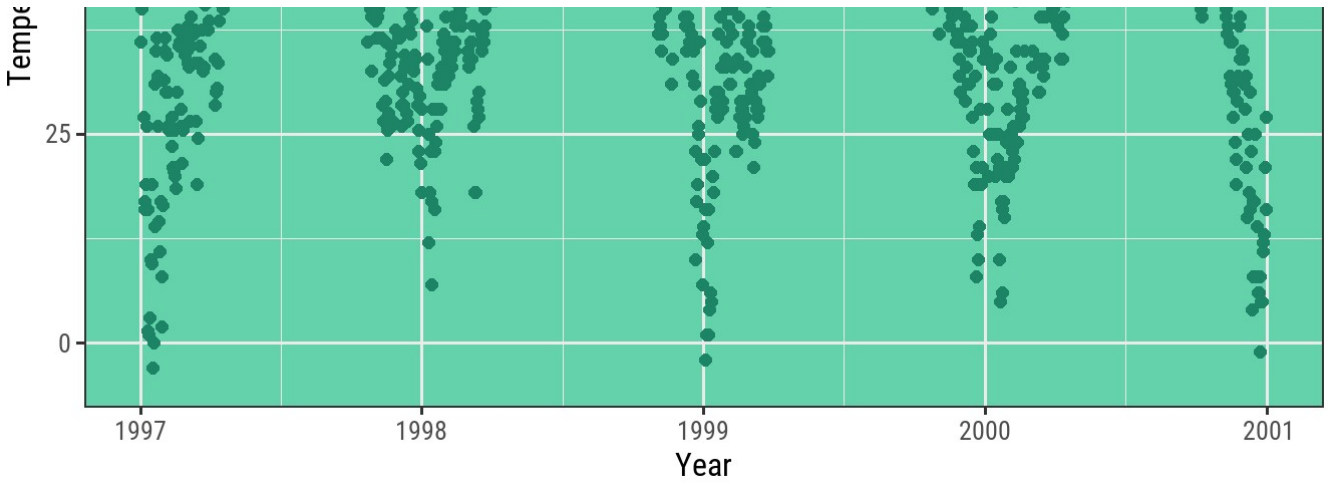
There are ways to change the entire look of your plot with one function (see "Working with Themes" section below) but if you want to simply change the colors of some elements, you can also do that.

CHANGE THE PANEL BACKGROUND COLOR

To change the background color (fill) of the panel area (i.e. the area where the data is plotted), one needs to adjust the theme element `panel.background`:

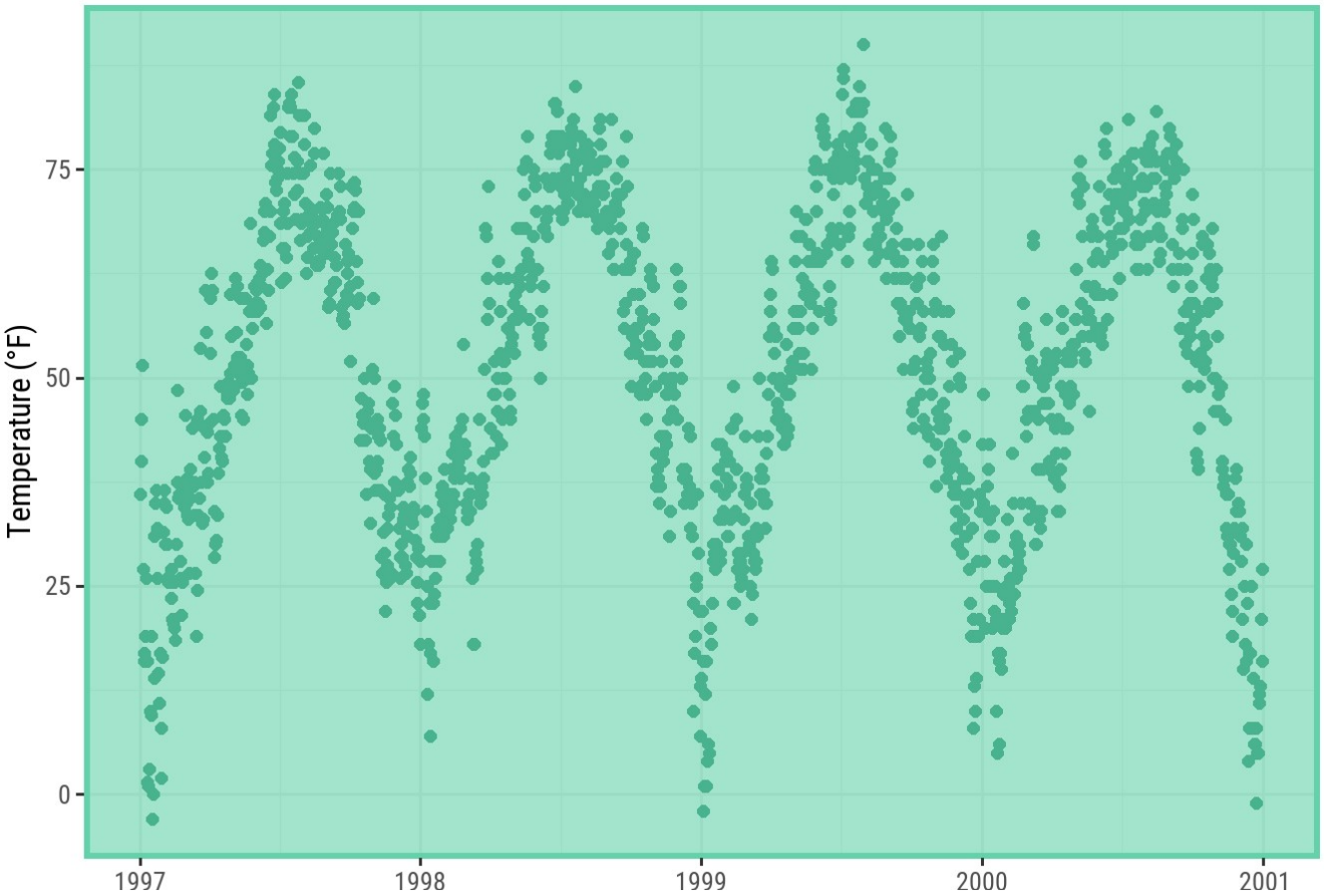
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "#1D8565", size = 2) +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(panel.background = element_rect(
    fill = "#64D2AA", color = "#64D2AA", size = 2)
  )
```





Note that the true color—the outline of the panel background—did not change even though we specified it. This is because there is a layer on top of the `panel.background`, namely `panel.border`. However, when make sure to use a transparent fill here, otherwise your data is hidden behind this layer. In the following example, I illustrate that 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", size = 2)
  )
```

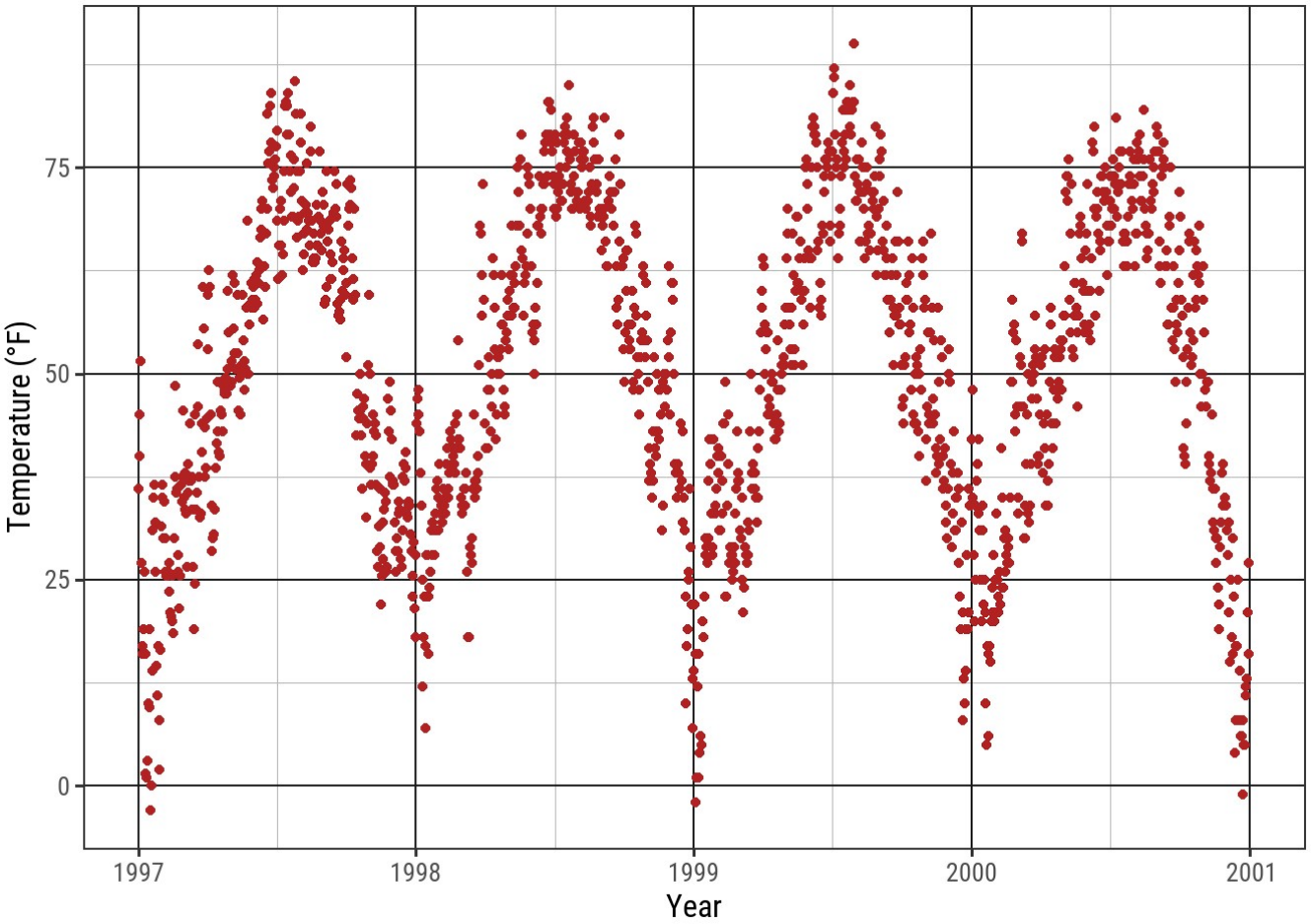


Year

CHANGE GRID LINES

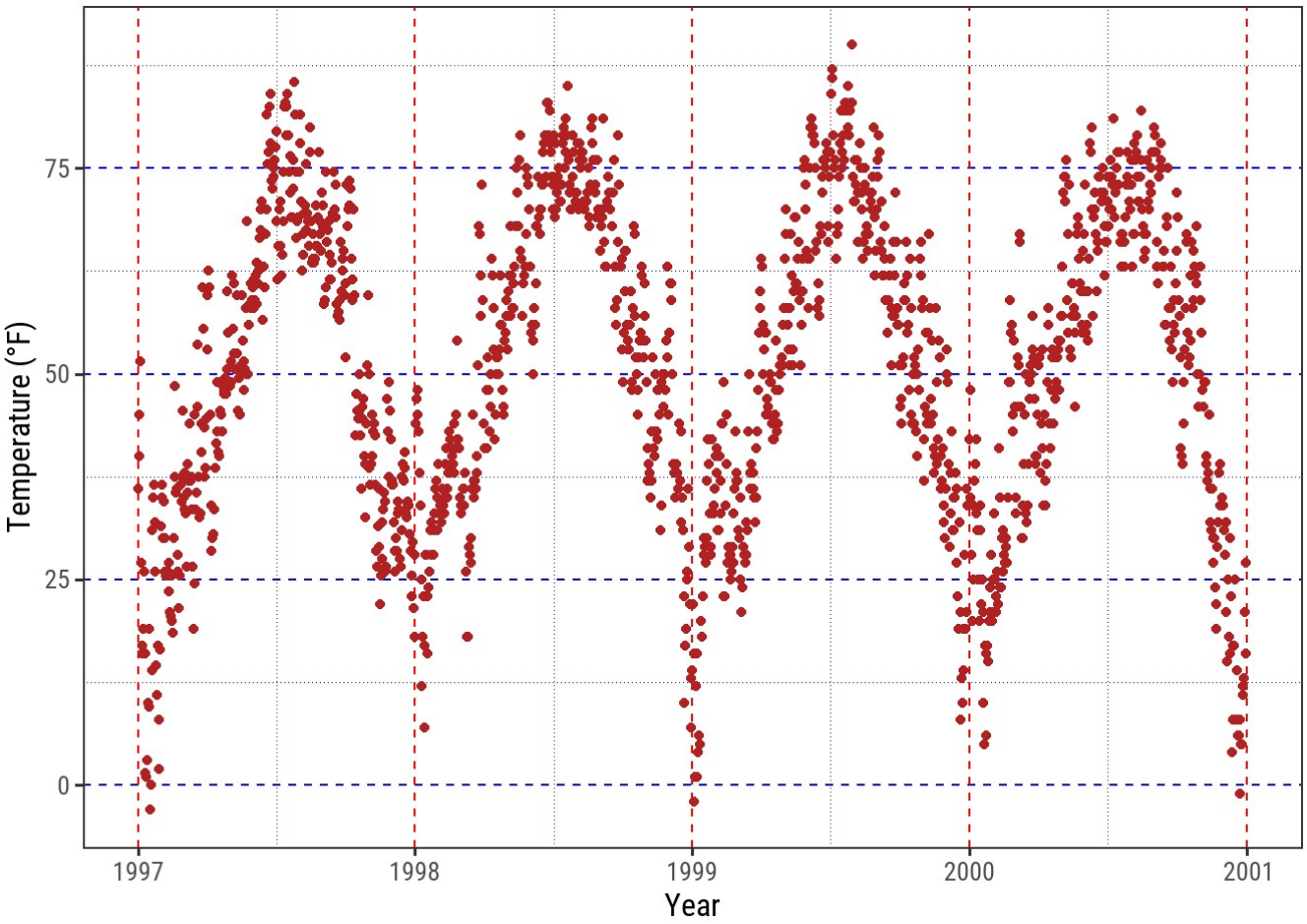
There are two types of grid lines: major grid lines indicating the ticks and minor grid lines between the major ones. You can change all of these by overwriting the defaults for `panel.grid` or for each set of gridlines separately, `panel.grid.major` and `panel.grid.minor`.

```
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", size = .5),
        panel.grid.minor = element_line(color = "gray70", size = .25))
```



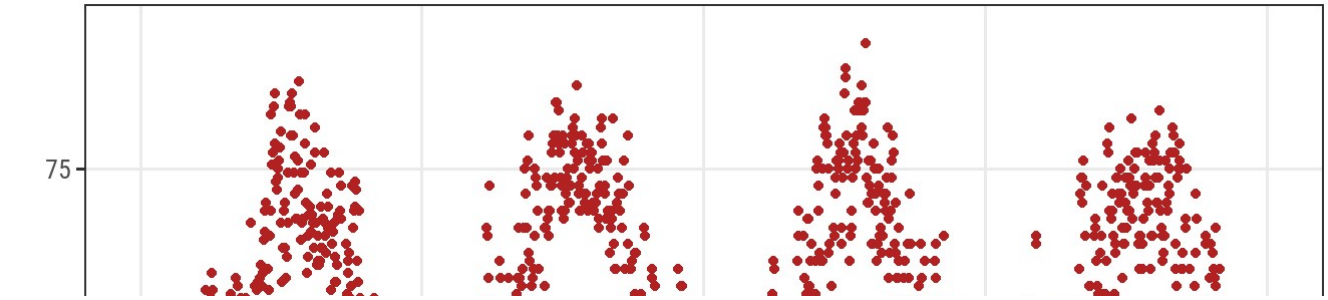
You can even specify settings for all four different levels:

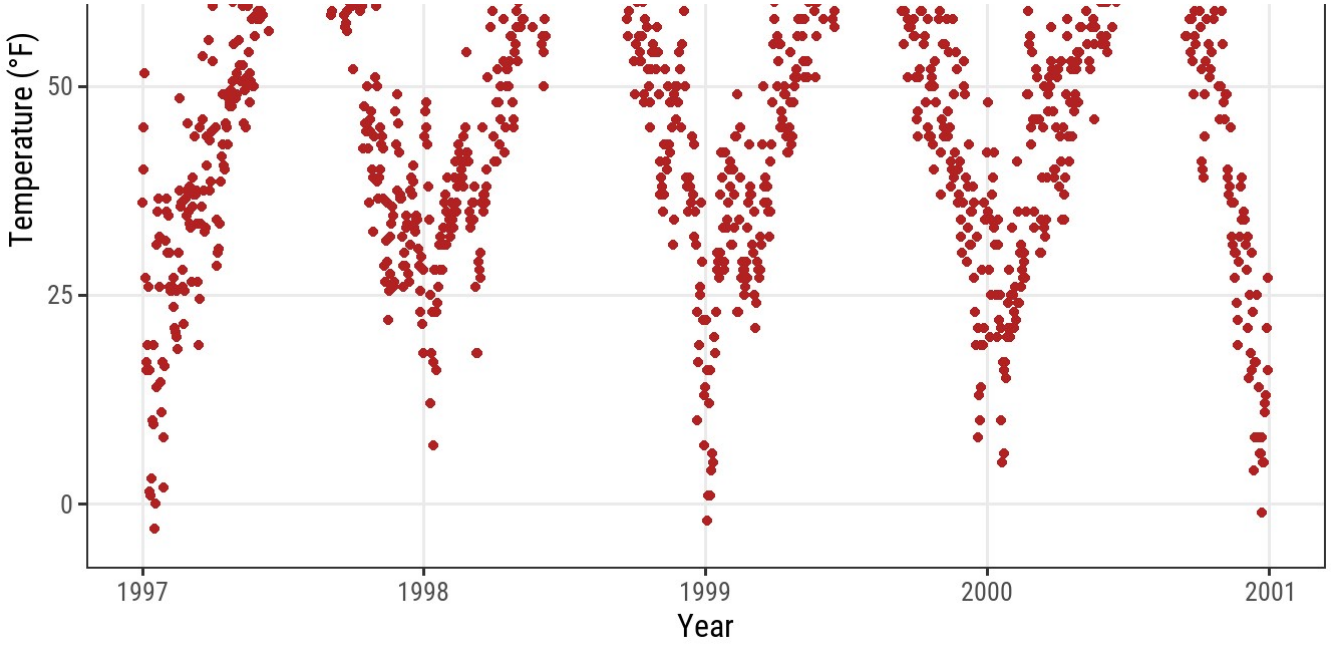
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(panel.grid.major = element_line(size = .5, linetype = "dashed"),
        panel.grid.minor = element_line(size = .25, linetype = "dotted"),
        panel.grid.major.x = 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"))
```



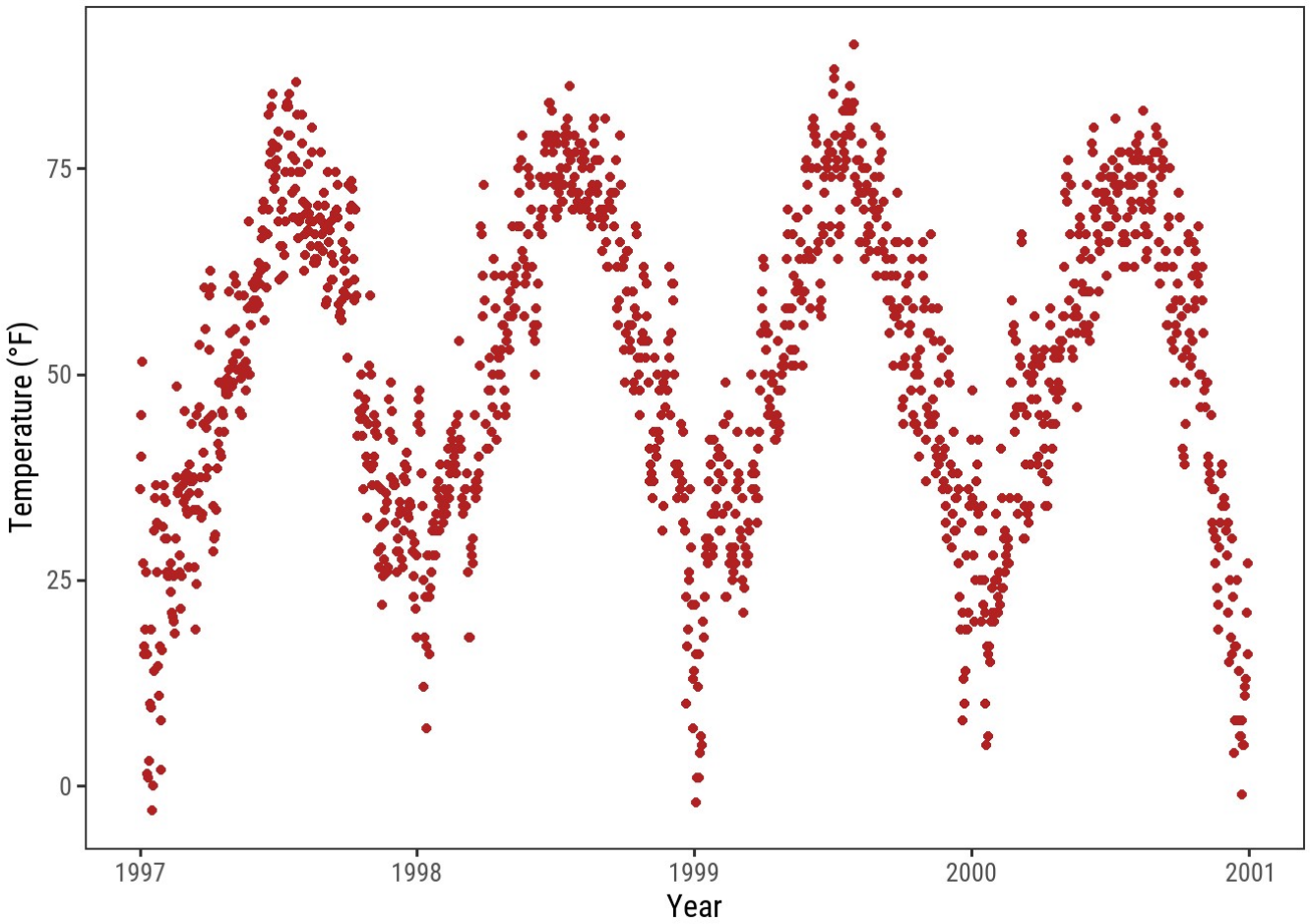
And, of course, you can remove some or all grid lines if you like:

```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(panel.grid.minor = element_blank())
```





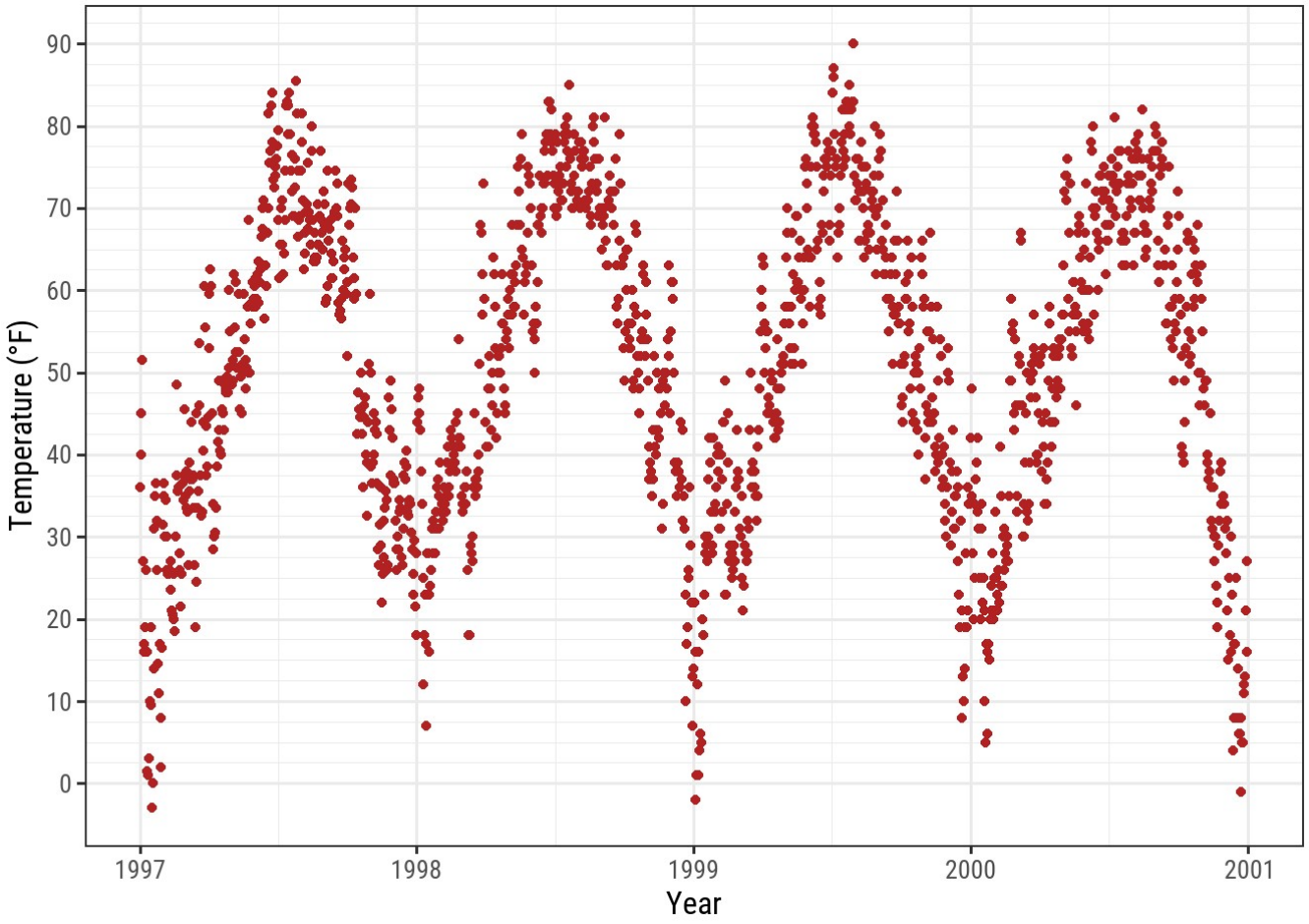

```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(panel.grid = element_blank())
```



CHANGE SPACING OF GRIDLINES

Furthermore, you can also define the breaks between both, major and minor grid lines:

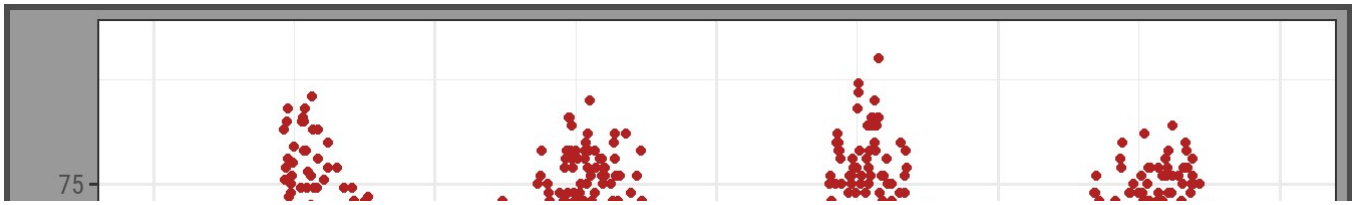
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  scale_y_continuous(breaks = seq(0, 100, 10),
                     minor_breaks = seq(0, 100, 2.5))
```

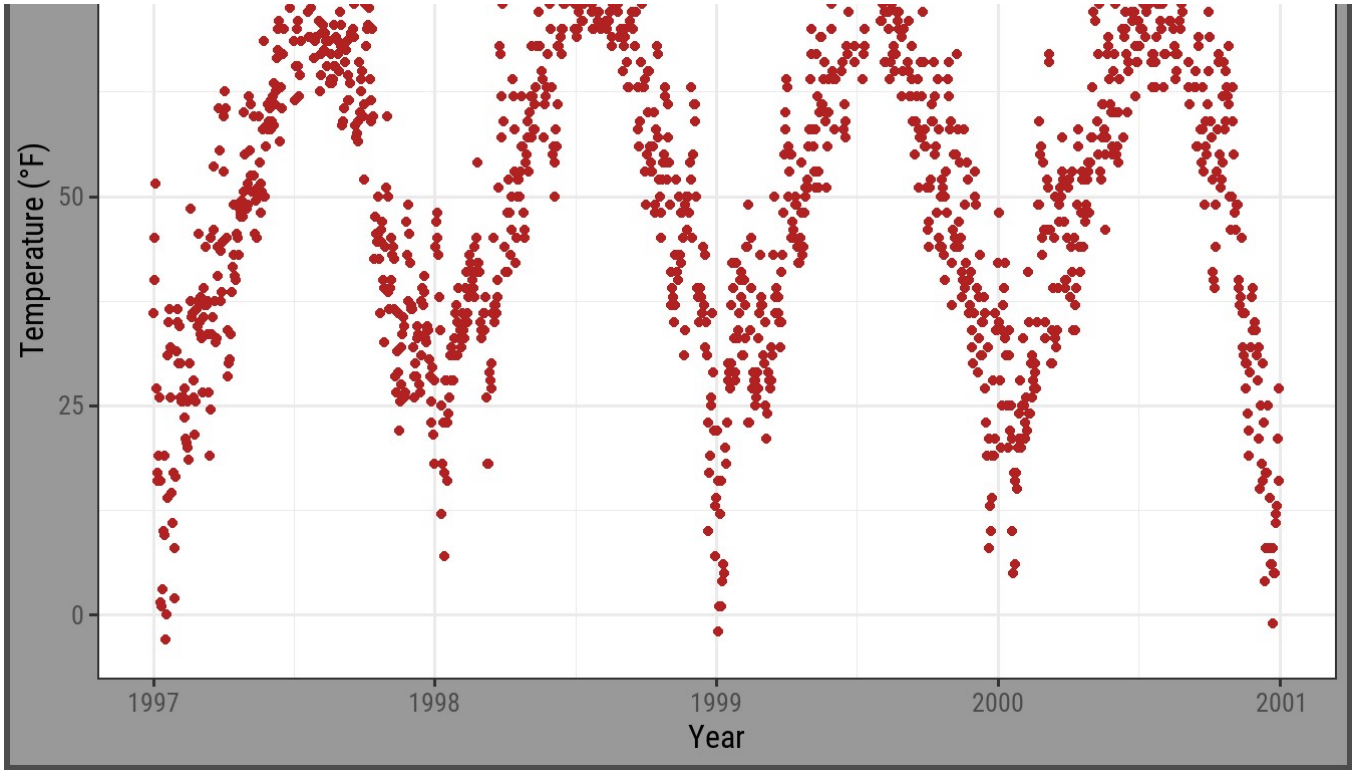


CHANGE THE PLOT BACKGROUND COLOR

Similarly, to change the background color (fill) of the plot area, one needs to modify the theme element `plot.background`:

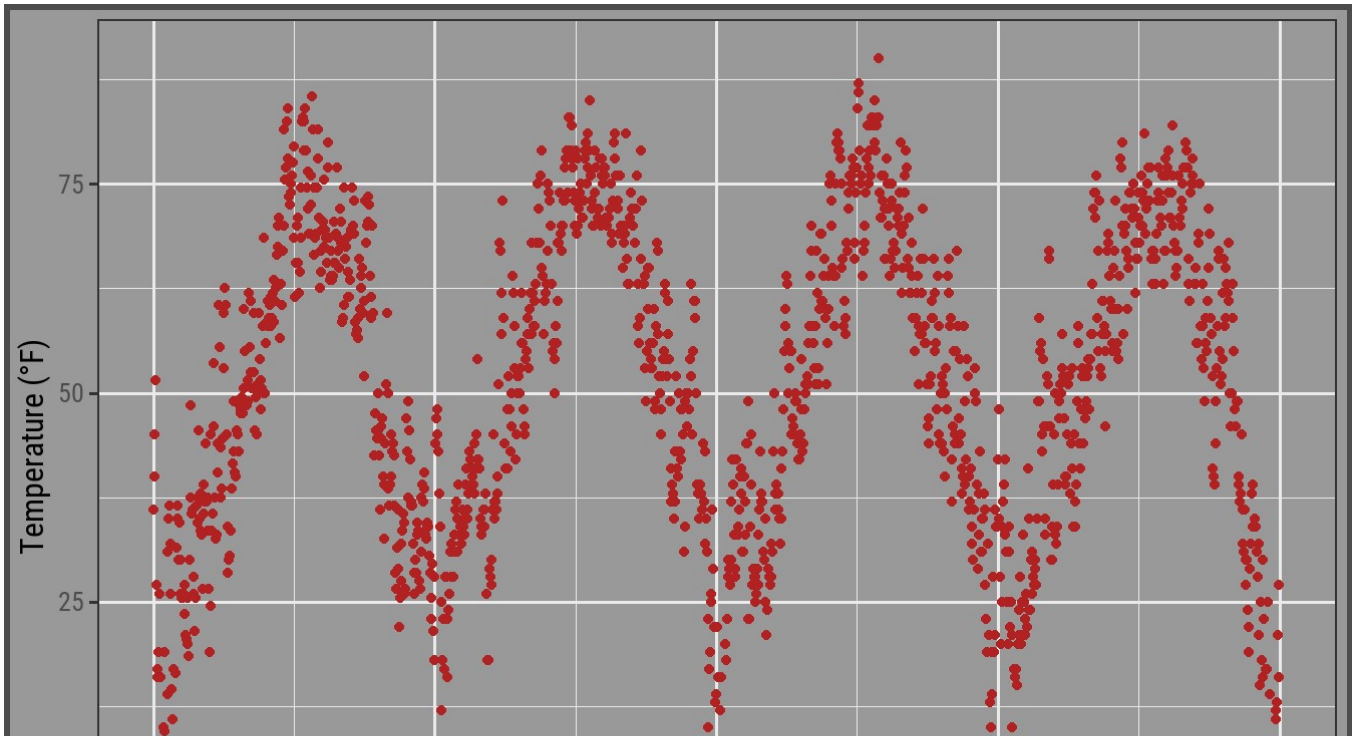
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "firebrick") +
  labs(x = "Year", y = "Temperature (°F)") +
  theme(plot.background = element_rect(fill = "gray60",
                                       color = "gray30", size = 2))
```

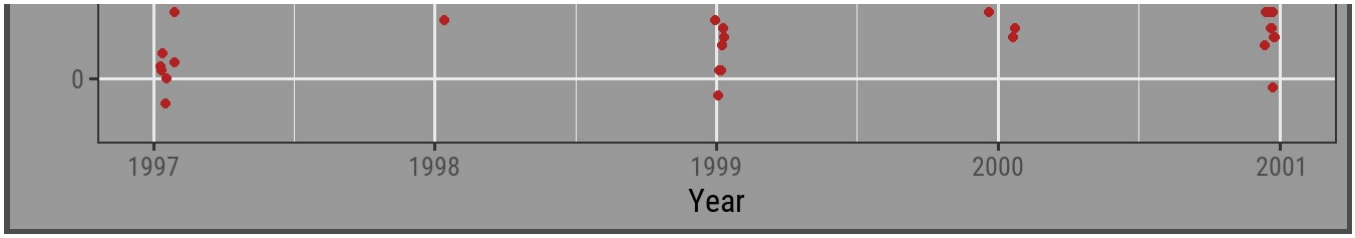




You can achieve a unique background color by either setting the same colors in both `panel.background` and `plot.background` or by setting the background filling of the panel to `"transparent"` or `NA`:

```
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", size = 2))
```





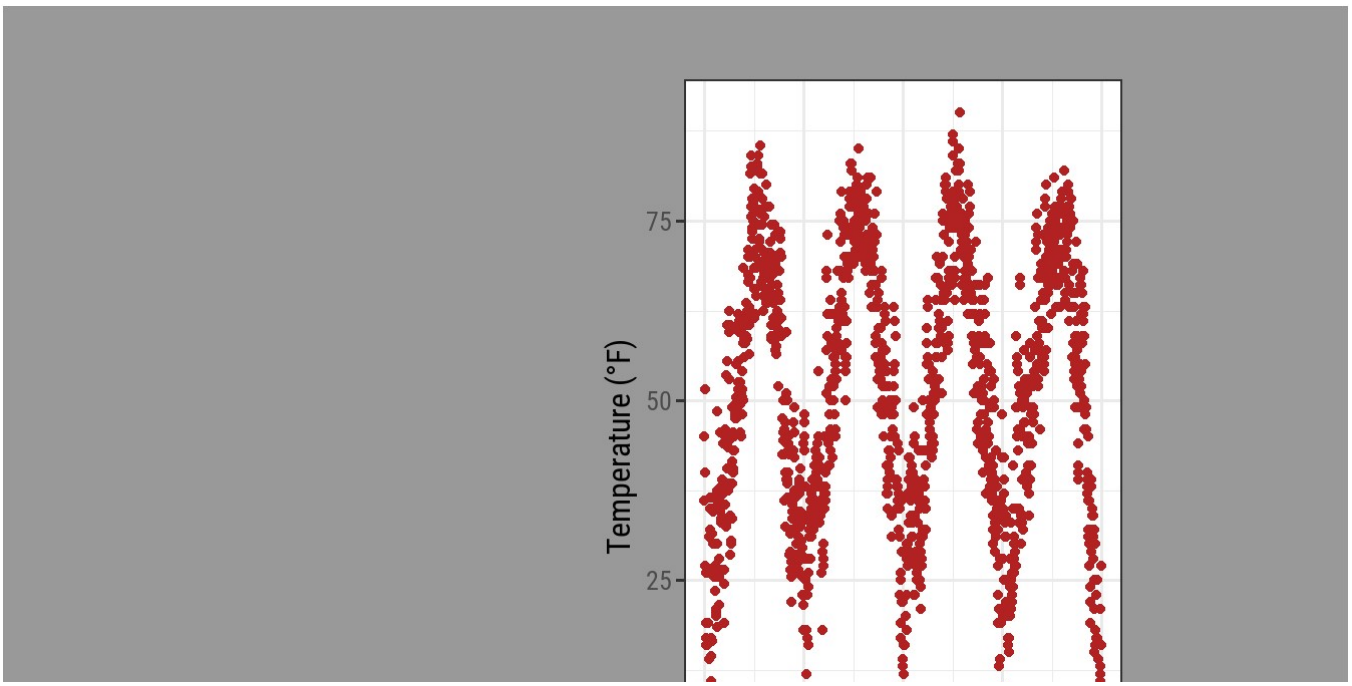
↑ Jump back to Table of Content.

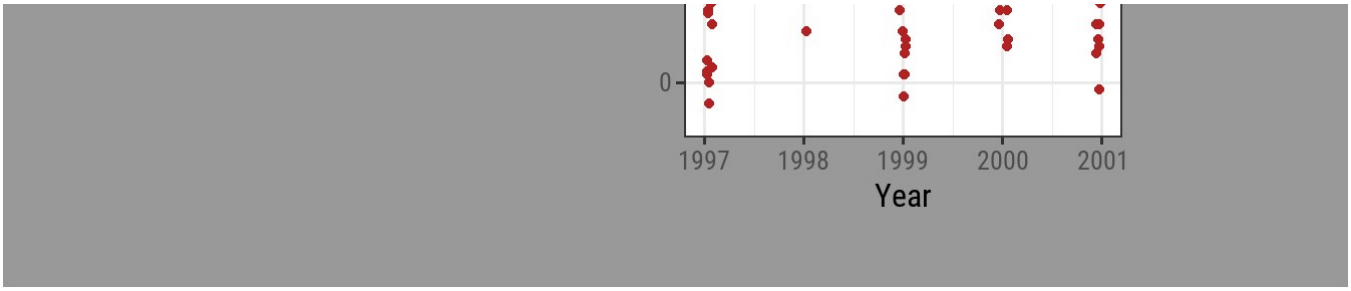
WORKING WITH MARGINS

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 In the previous example we already illustrated the default margin by changing the background color using `plot.background`.

Now let us add extra space to both the left and right. The argument, `plot.margin`, can handle a variety of different units (cm, inches, etc.) but it requires the use of the function unit 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 a 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"))
```





The order of the margin sides is top, right, bottom, left—a nice way to remember this order is "**trouble** that sorts the first letter of the four sides.

🤷 *You can also use* `unit()` *instead of* `margin()`*. Expand to see example.*

↑ Jump back to Table of Content.

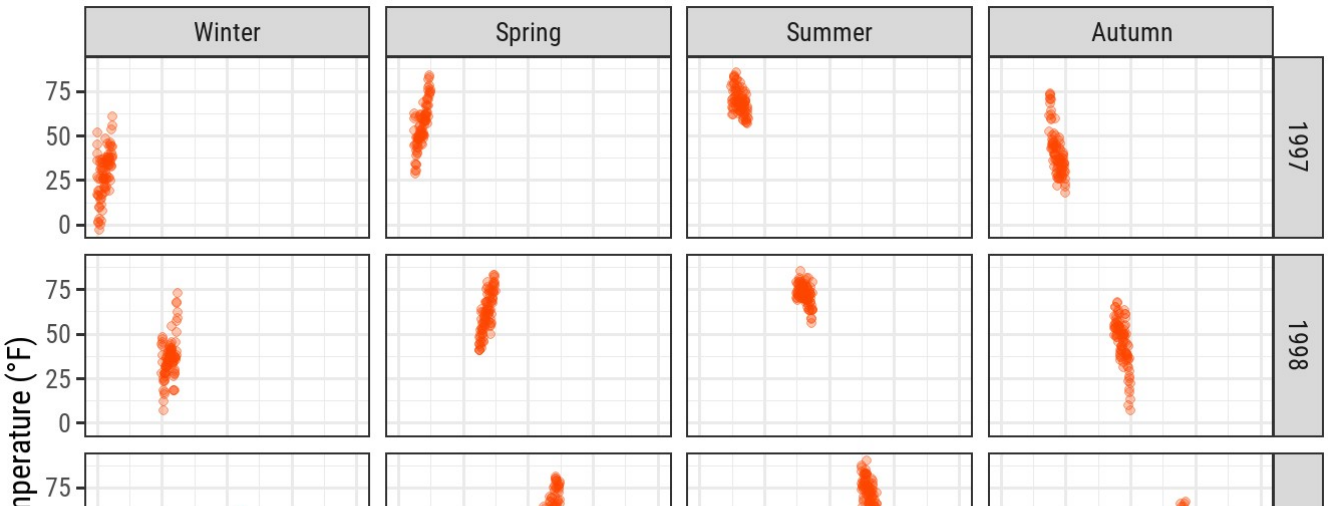
WORKING WITH MULTI-PANEL PLOTS

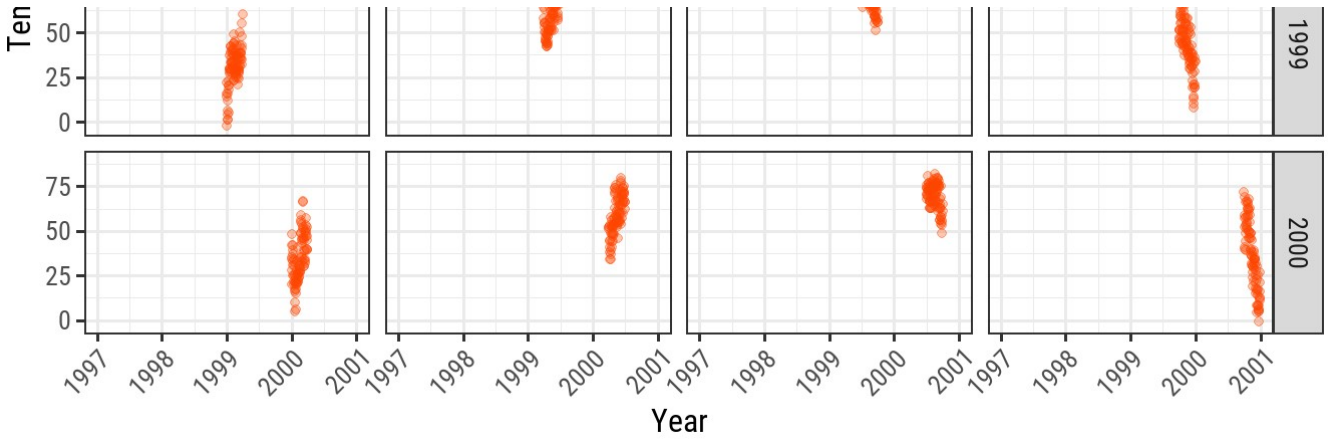
The `{ggplot2}` package has two nice functions for creating multi-panel plots, called *facets*. They are related but a little different: `facet_wrap` creates essentially a ribbon of plots based on a single variable while `facet_grid` spans a grid of two variables.

CREATE A GRID OF SMALL MULTIPLES BASED ON TWO VARIABLES

In case of two variables, `facet_grid` does the job. Here, the order of the variables determines the number of rows and columns:

```
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)
```



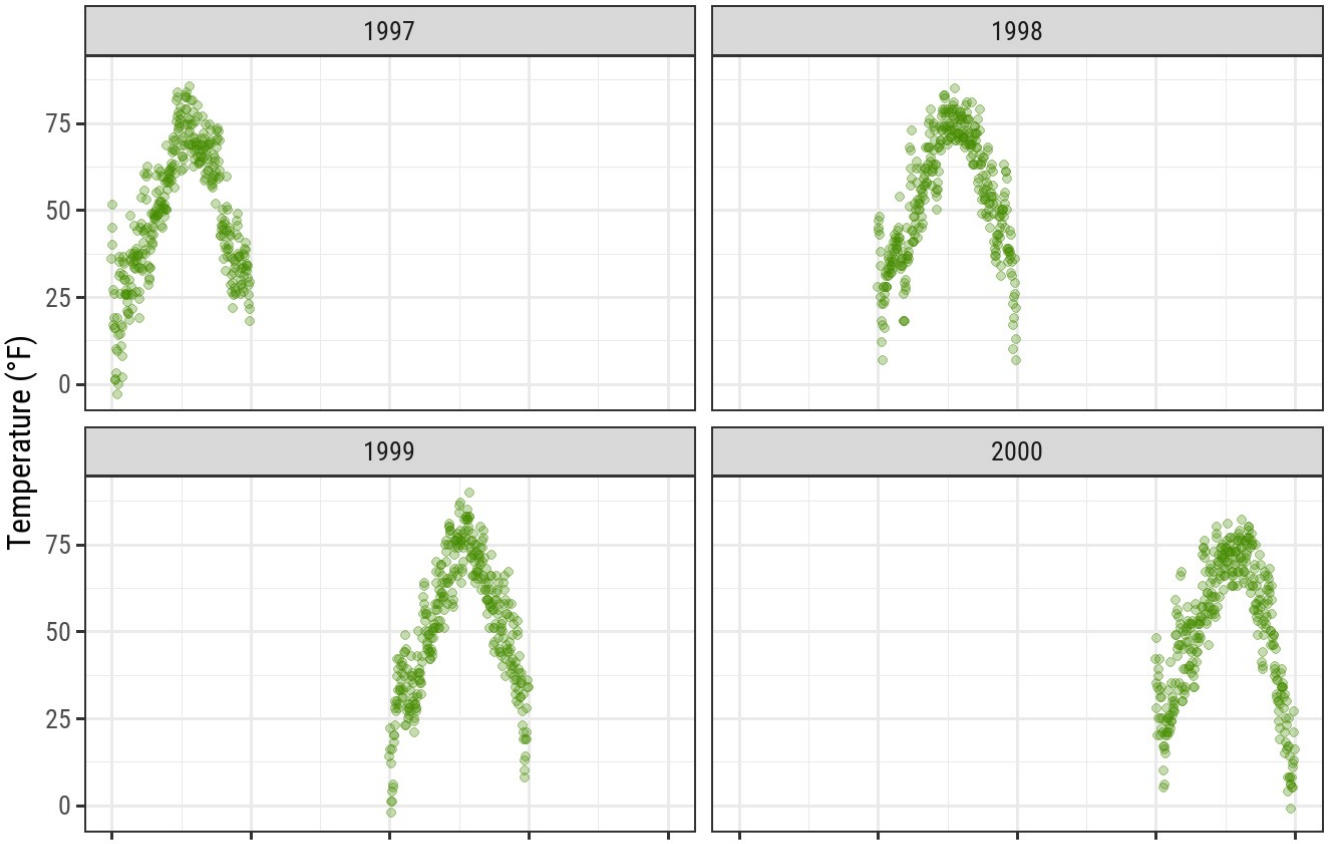


To change from row to column arrangement you can change `facet_grid(year ~ season)` to `facet_grid(season ~ year)`.

CREATE SMALL MULTIPLES BASED ON ONE VARIABLE

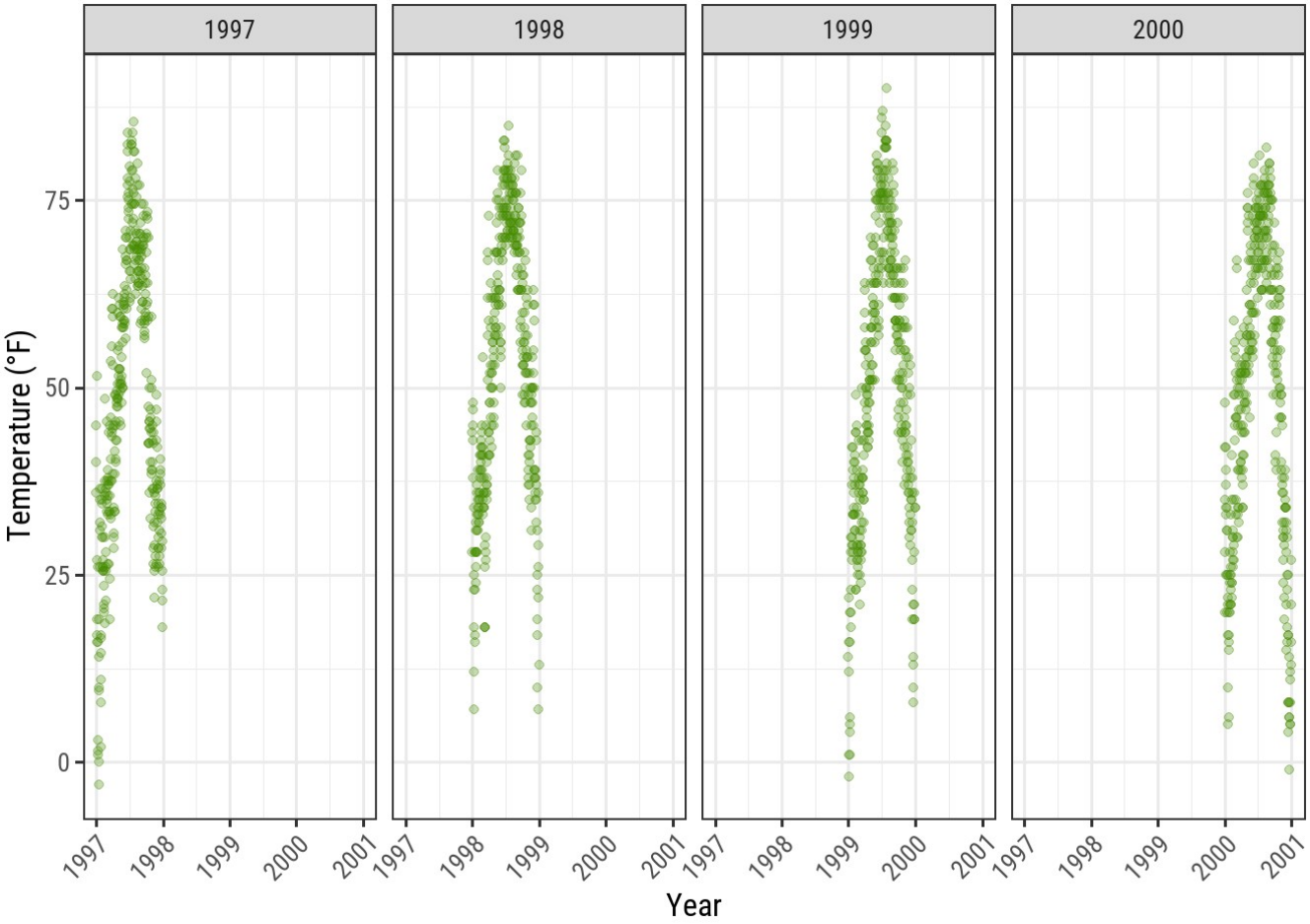
`facet_wrap` creates a facet of a single variable, written with a tilde in front: `facet_wrap(~ variable)`. The appearance of these subplots is controlled by the arguments `ncol` and `nrow`:

```
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)
```



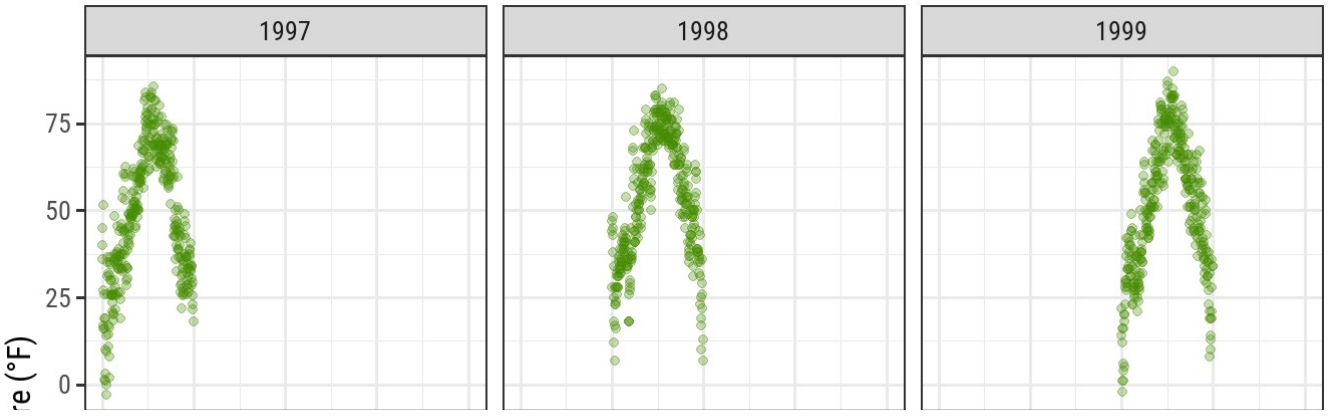
Accordingly, you can arrange the plots as you like, instead as a matrix in one row...

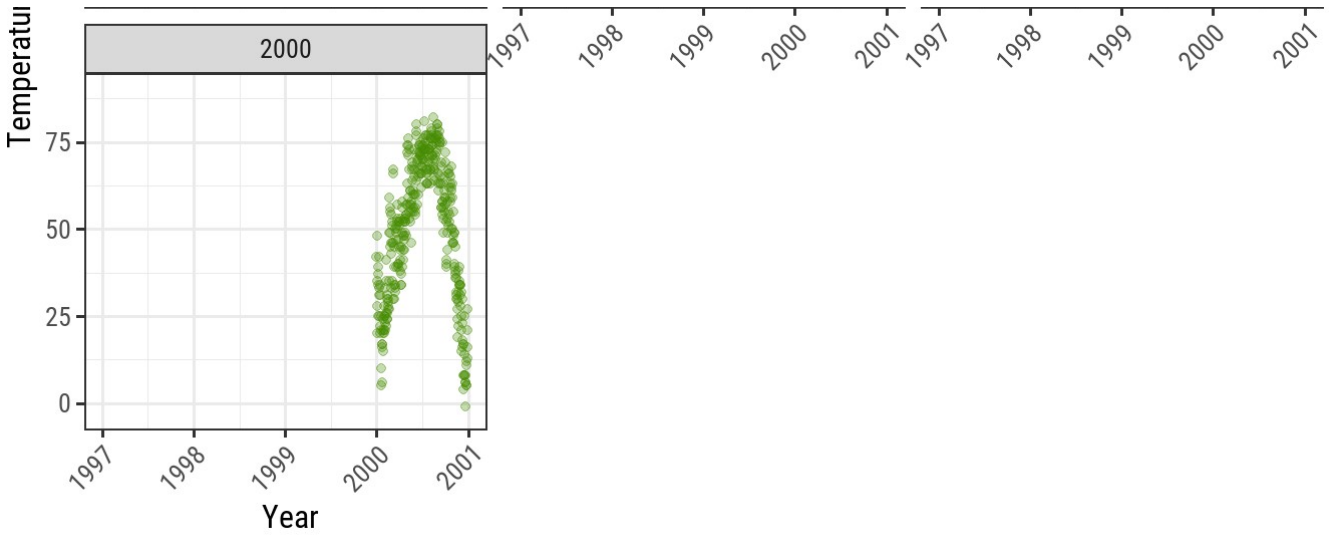
```
g + facet_wrap(~ year, nrow = 1)
```



... or even as a asymmetric grid of plots:

```
g + facet_wrap(~ year, ncol = 3) + theme(axis.title.x = element_text(hjust = .15))
```

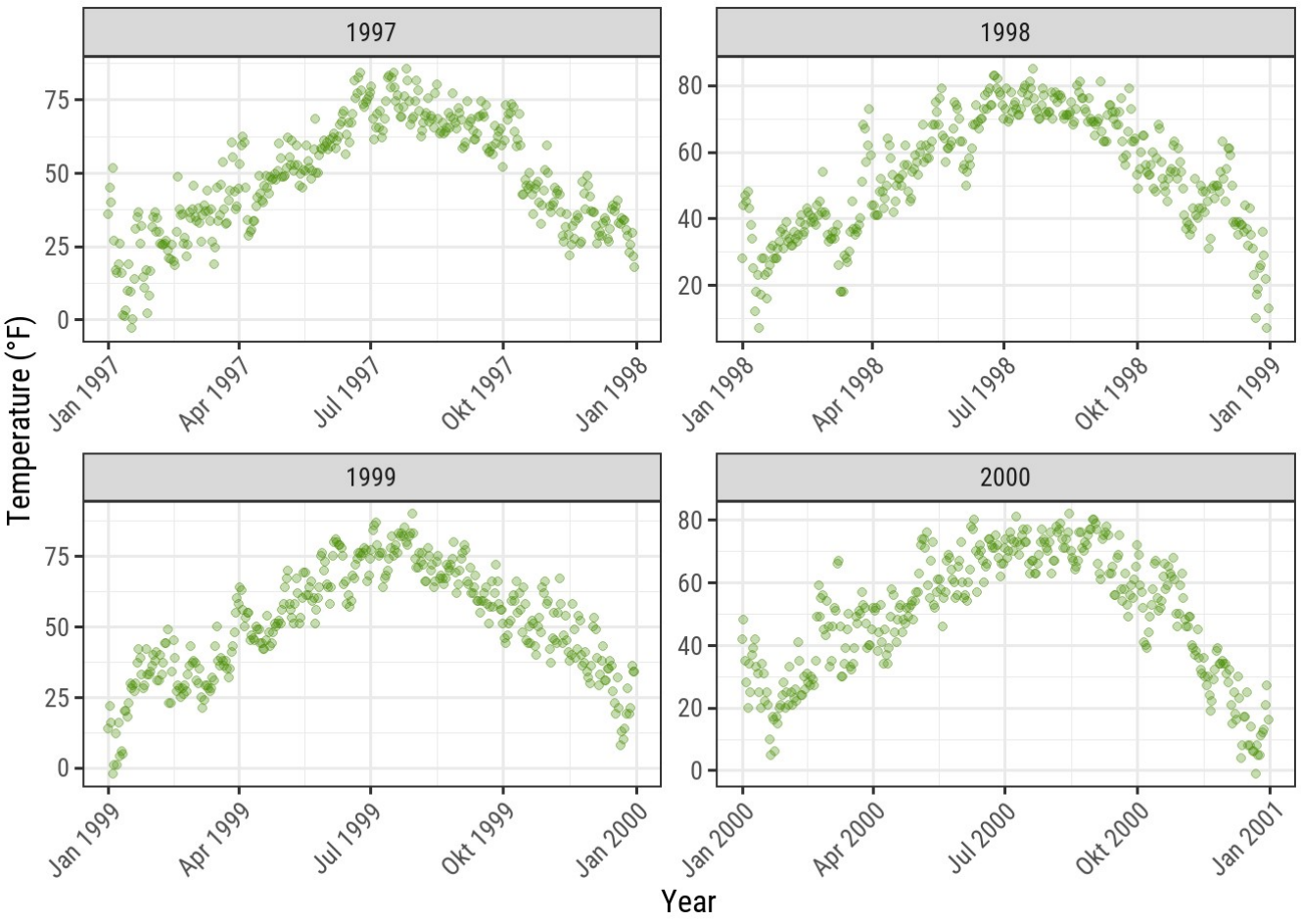




ALLOW AXES TO ROAM FREE

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")
```

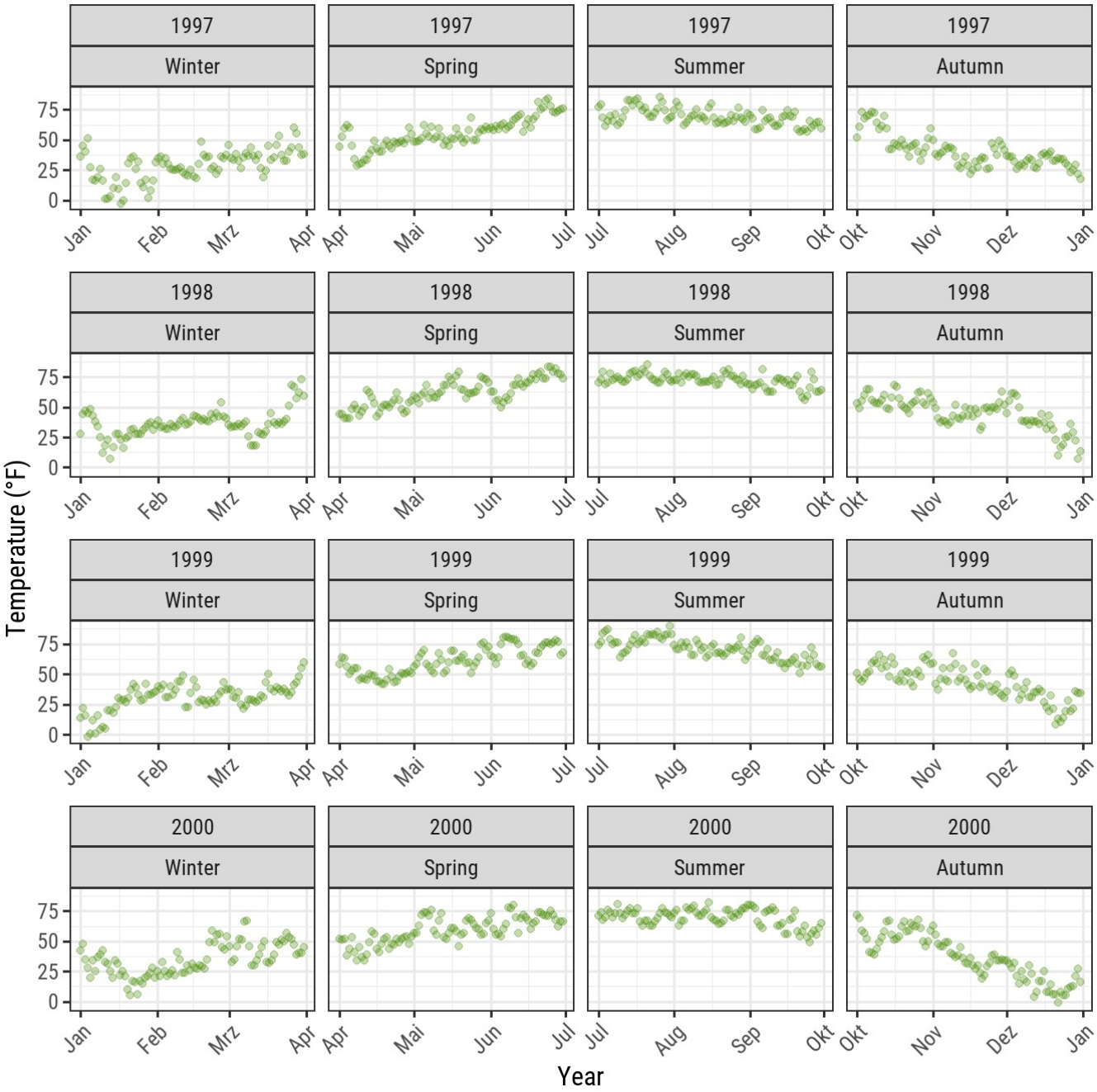


Note that both, x and y axes differ in their range!

USE `facet_wrap` WITH TWO VARIABLES

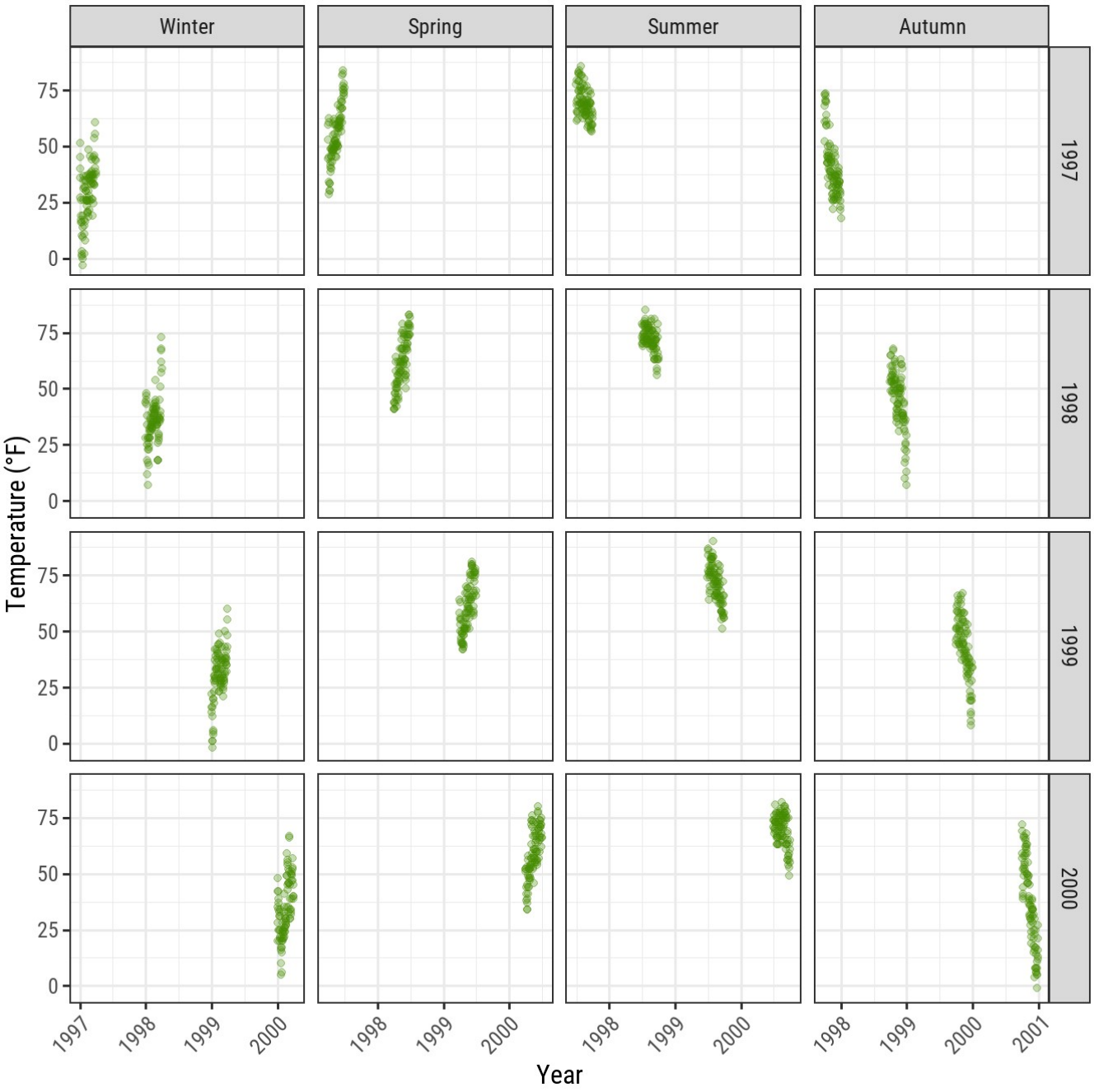
The function `facet_wrap` can also take two variables:

```
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:

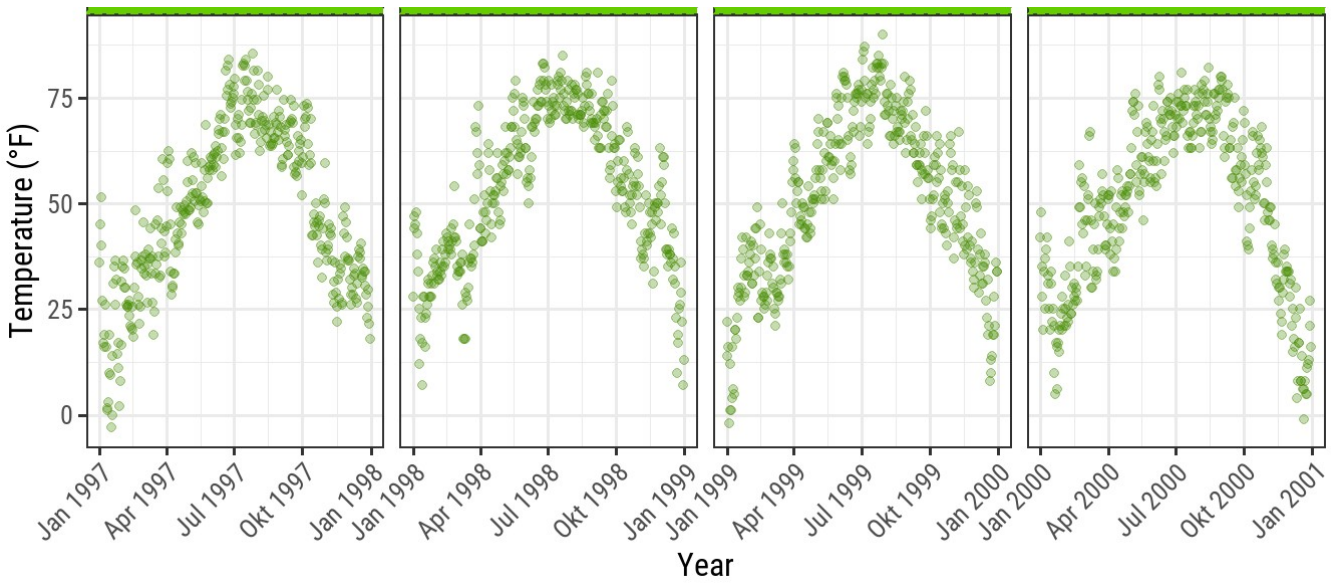
```
g + facet_grid(year ~ season, scales = "free_x")
```



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"))
```

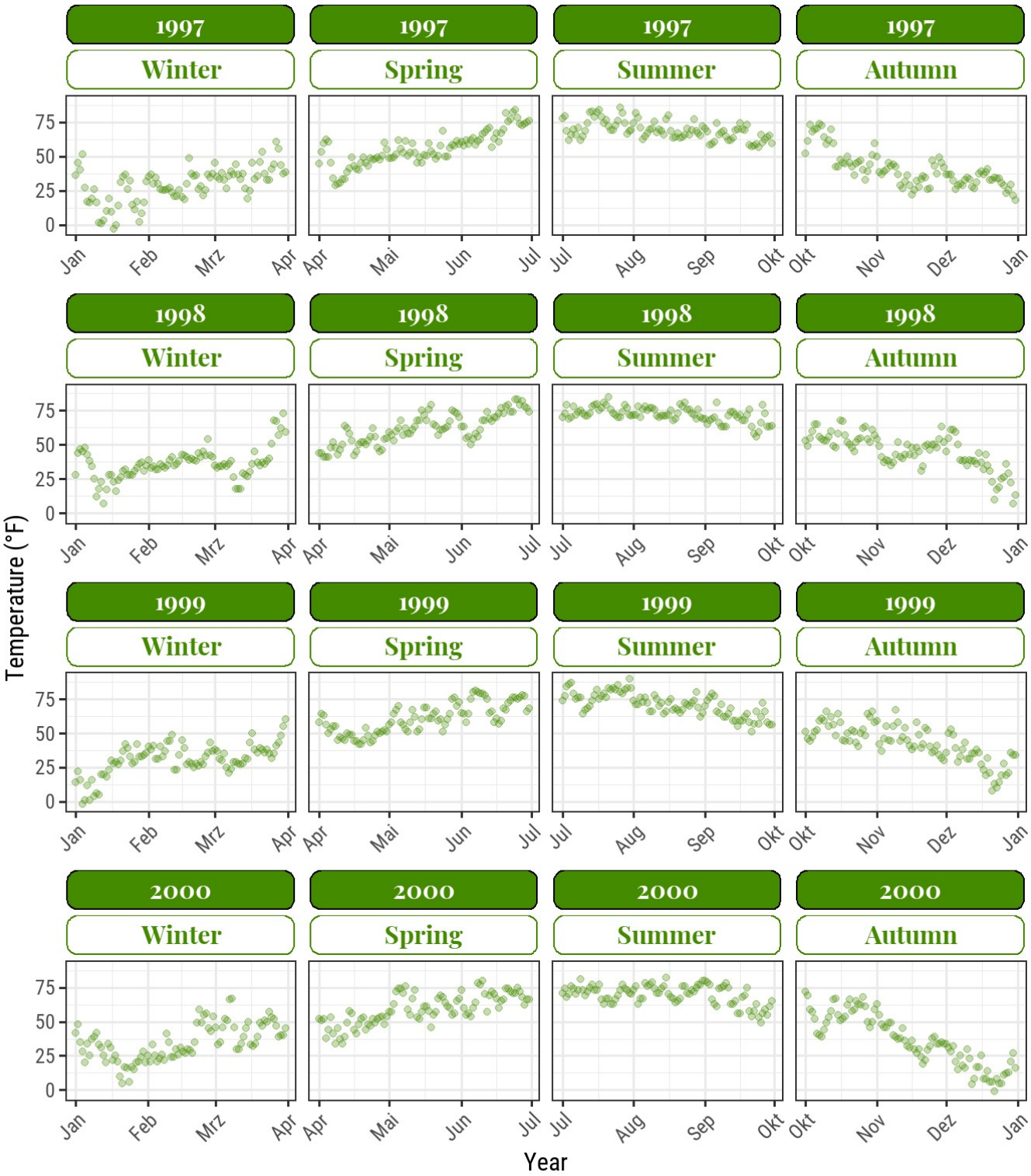


The following two functions adapted from this answer by Claus Wilke (https://stackoverflow.com/questions/60332202/conditionally-fill-ggtext-text-boxes-in-facet-wrap), the author of the `{ggtext}` package (https://wilkelab.org/ggtext/), allow to highlight specific labels in combination with `element_textbox()`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

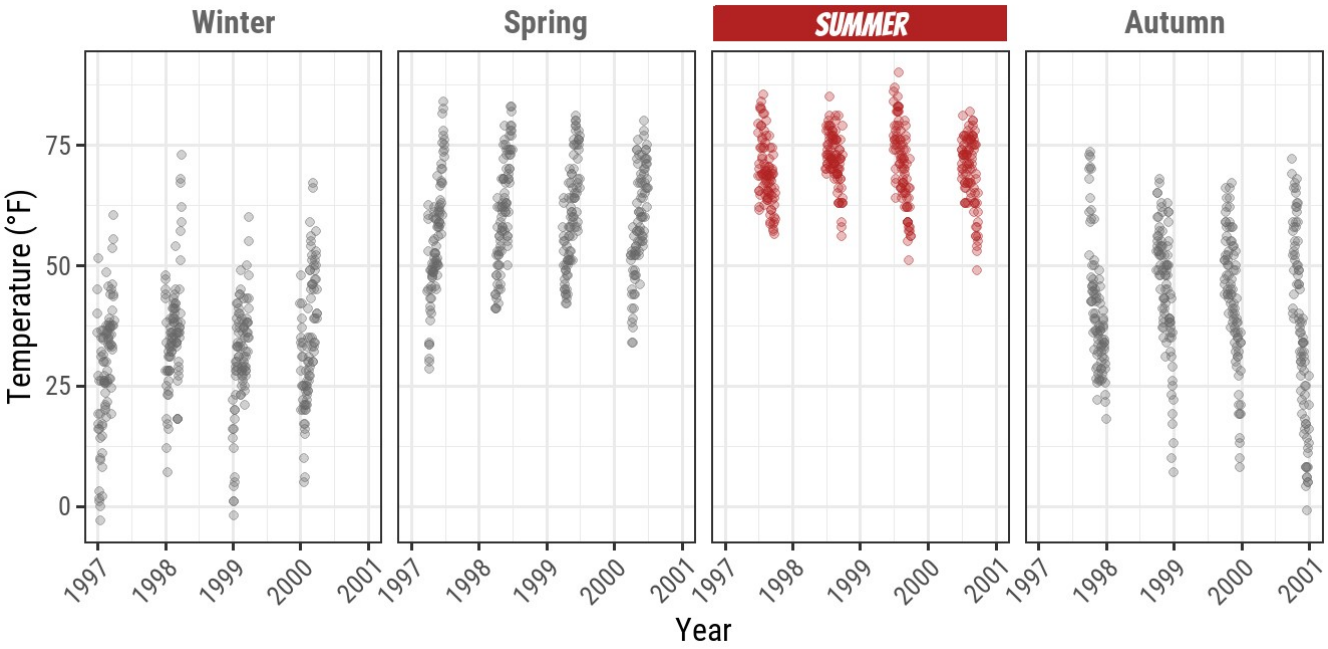
```
library(ggtext)
library(rlang)

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.c
    ),
    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", 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"
    )
  )
```



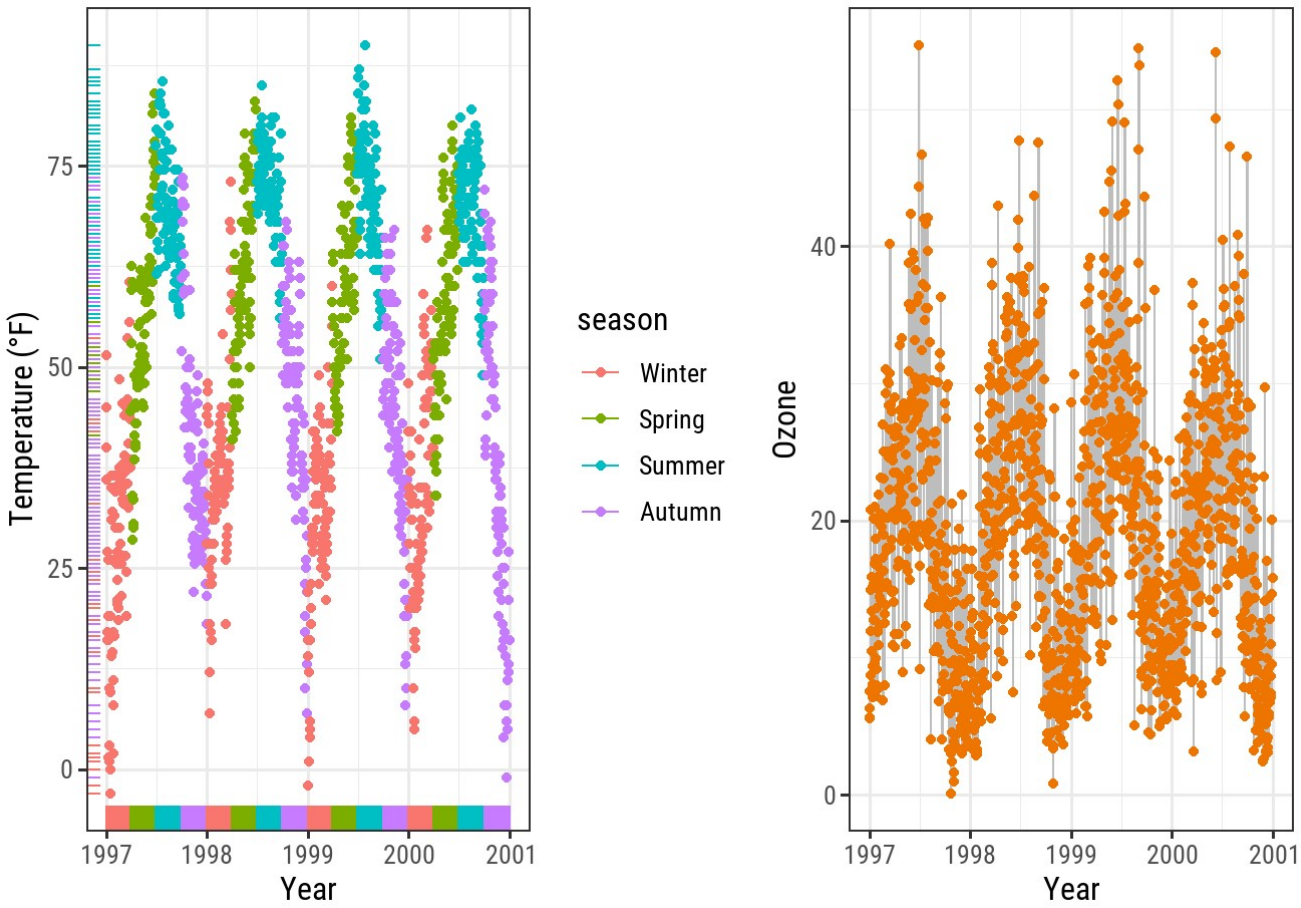
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(aes(color = season == "Summer"), alpha = .3) +
  labs(x = "Year", y = "Temperature (°F)") +
  facet_wrap(~ season, nrow = 1) +
  scale_color_manual(values = c("gray40", "firebrick"), guide = "none") +
  theme(
    axis.text.x = element_text(angle = 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"
    )
  )
```



CREATE A PANEL OF DIFFERENT PLOTS

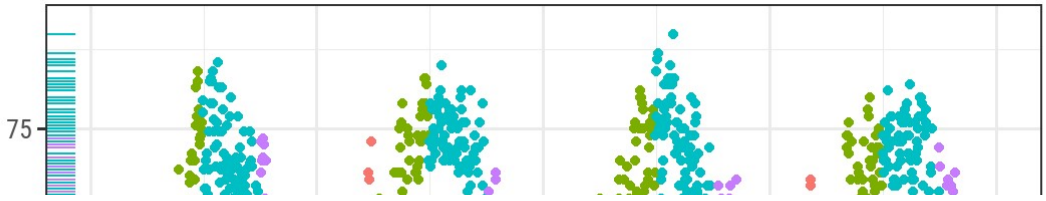
There are several ways how plots can be combined. The easiest approach in my opinion is the `{patchwork}` package (https://github.com/thomasp85/patchwork) by Thomas Lin Pedersen:

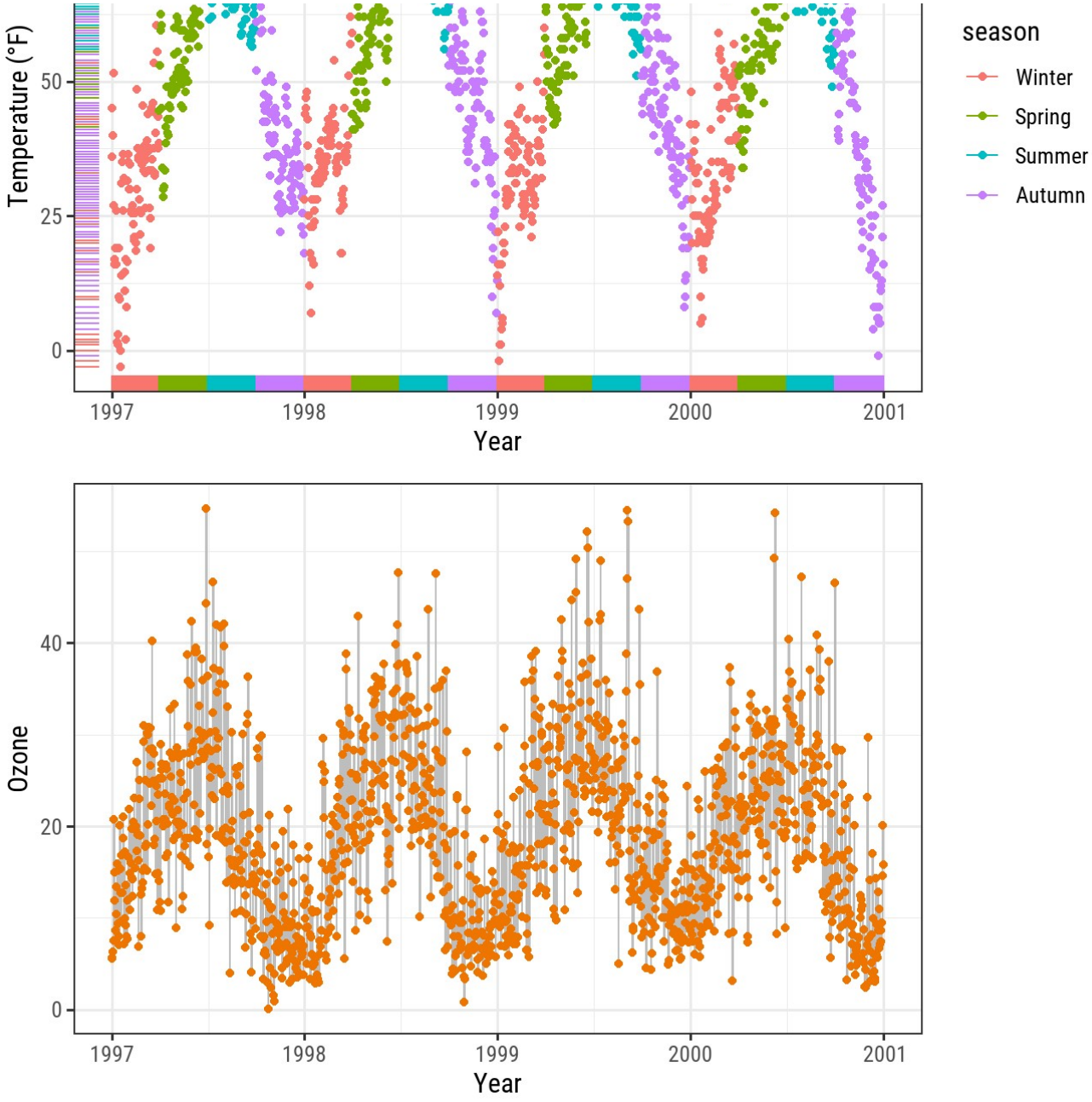
```
p1 <- ggplot(chic, aes(x = date, y = temp,
                       color = season)) +
        geom_point() +
        geom_rug() +
        labs(x = "Year", y = "Temperature (°F)")

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!):

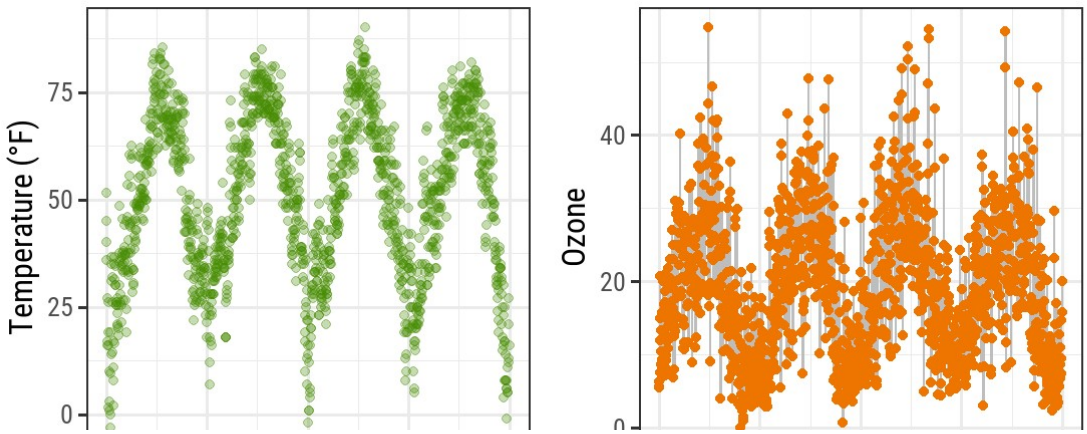
```
p1 / p2
```

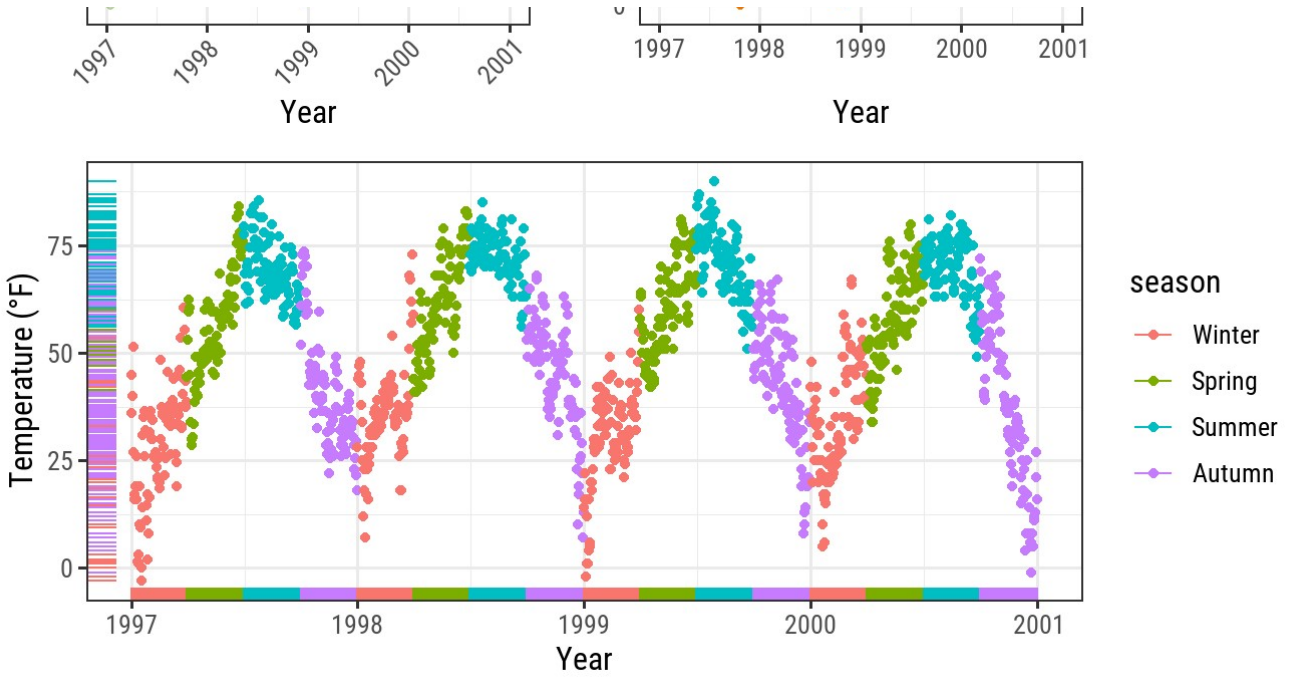




And also nested plots are possible!

```
(g + p2) / p1
```

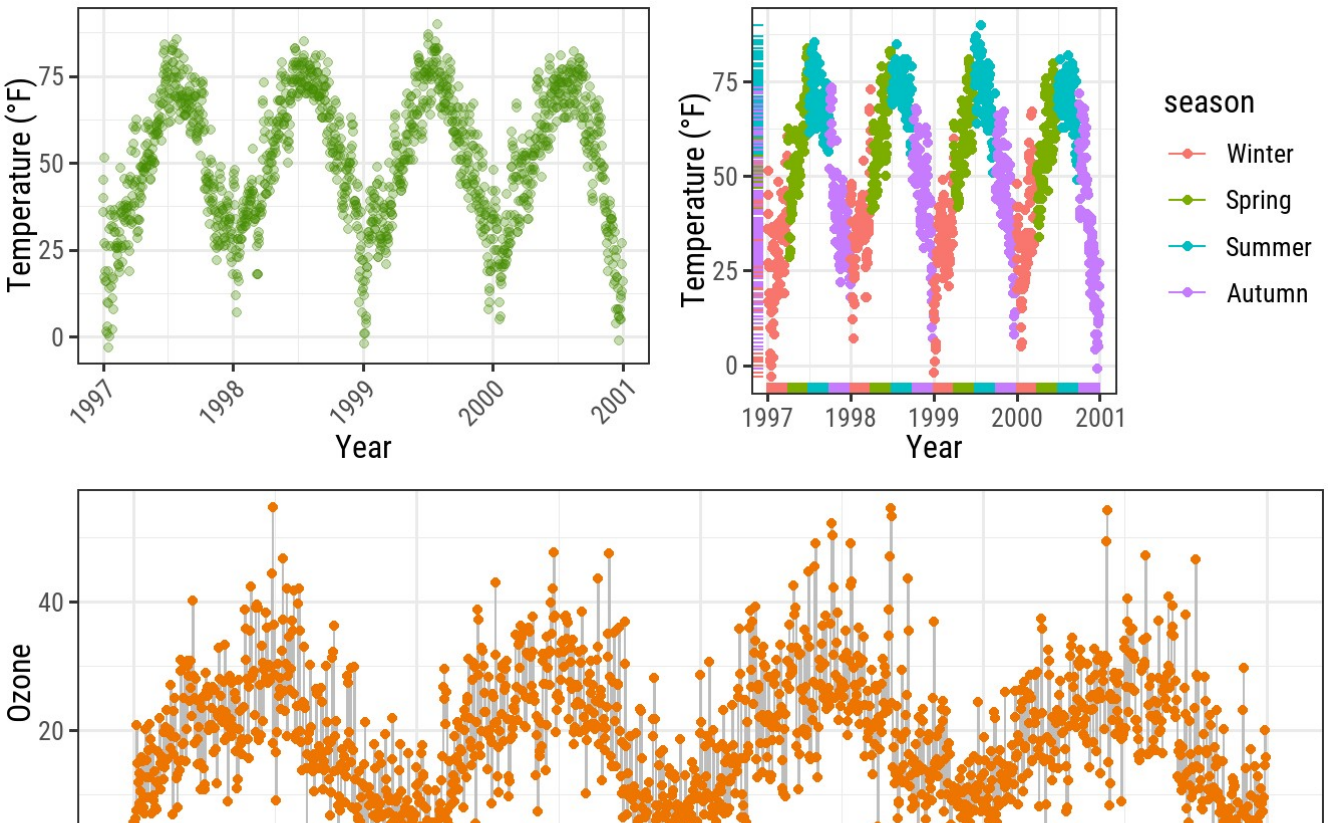


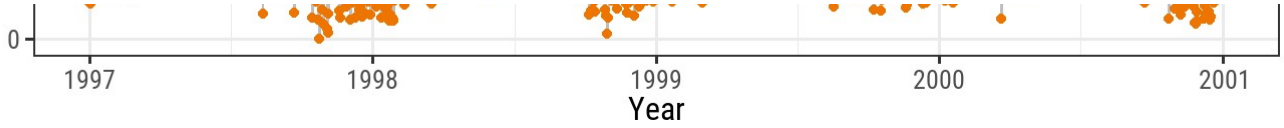


(Note the alignment of the plots even though only one plot includes a legend.)

Alternatively, the `{cowplot}` package (https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html) by Claus Wilke provides the functionality to combine multiple plots (and lots of other good utilities):

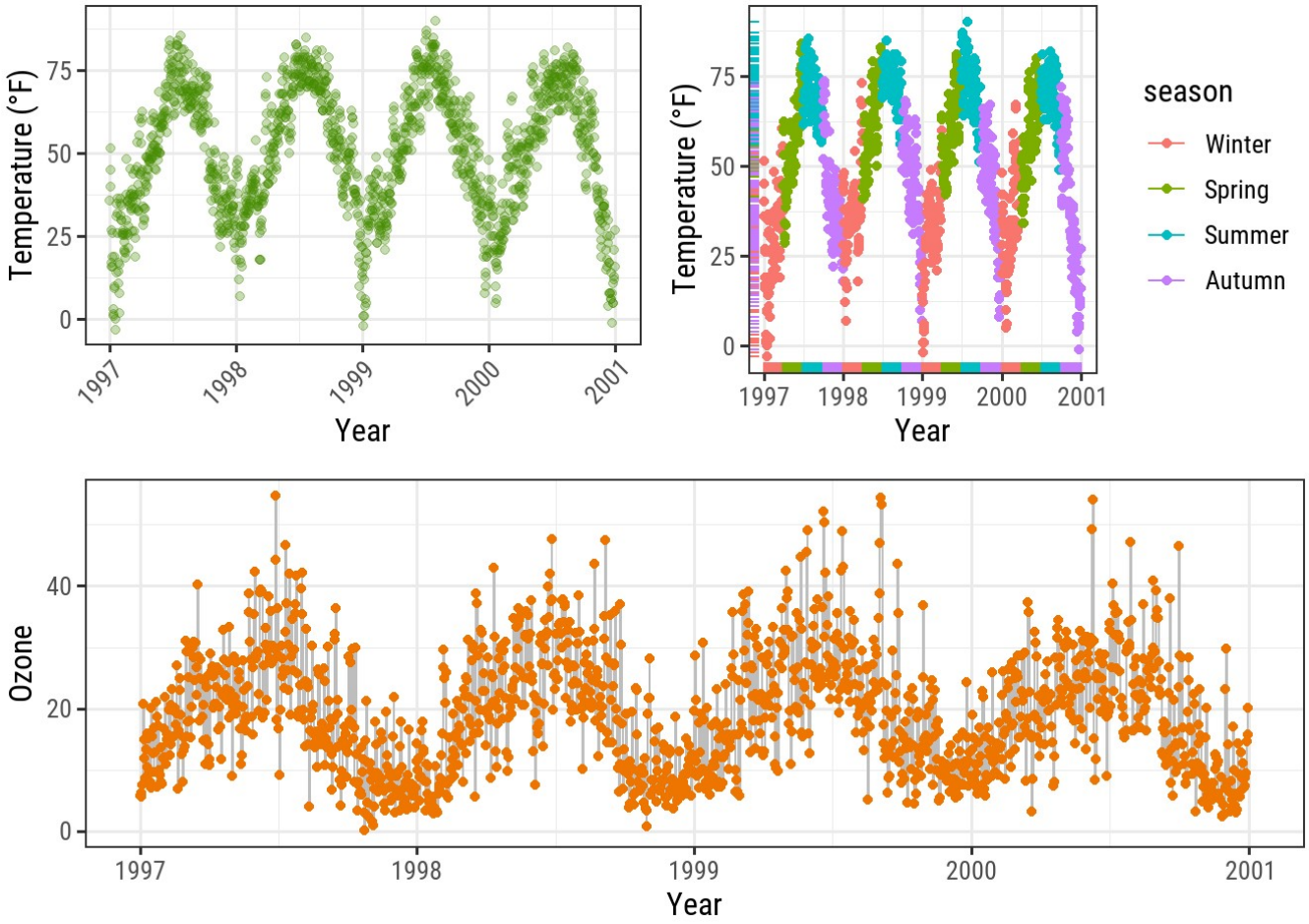
```
library(cowplot)
plot_grid(plot_grid(g, p1), p2, ncol = 1)
```





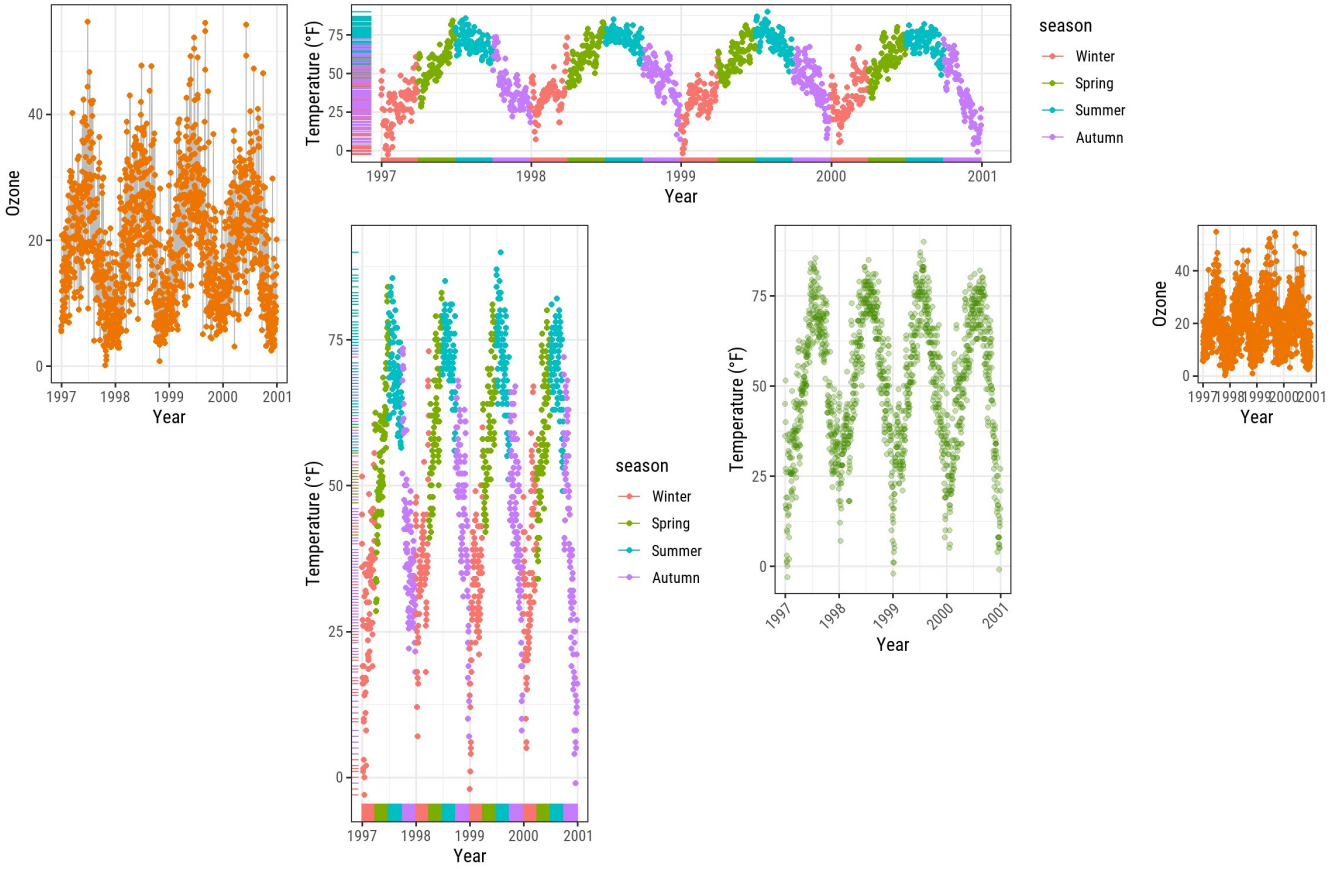
... and so does the `{gridExtra}` package (https://cran.r-project.org/web/packages/gridExtra/vignettes/arrangeGrob.html) as well:

```
library(gridExtra)
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)
```



↑ Jump back to Table of Content.

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.

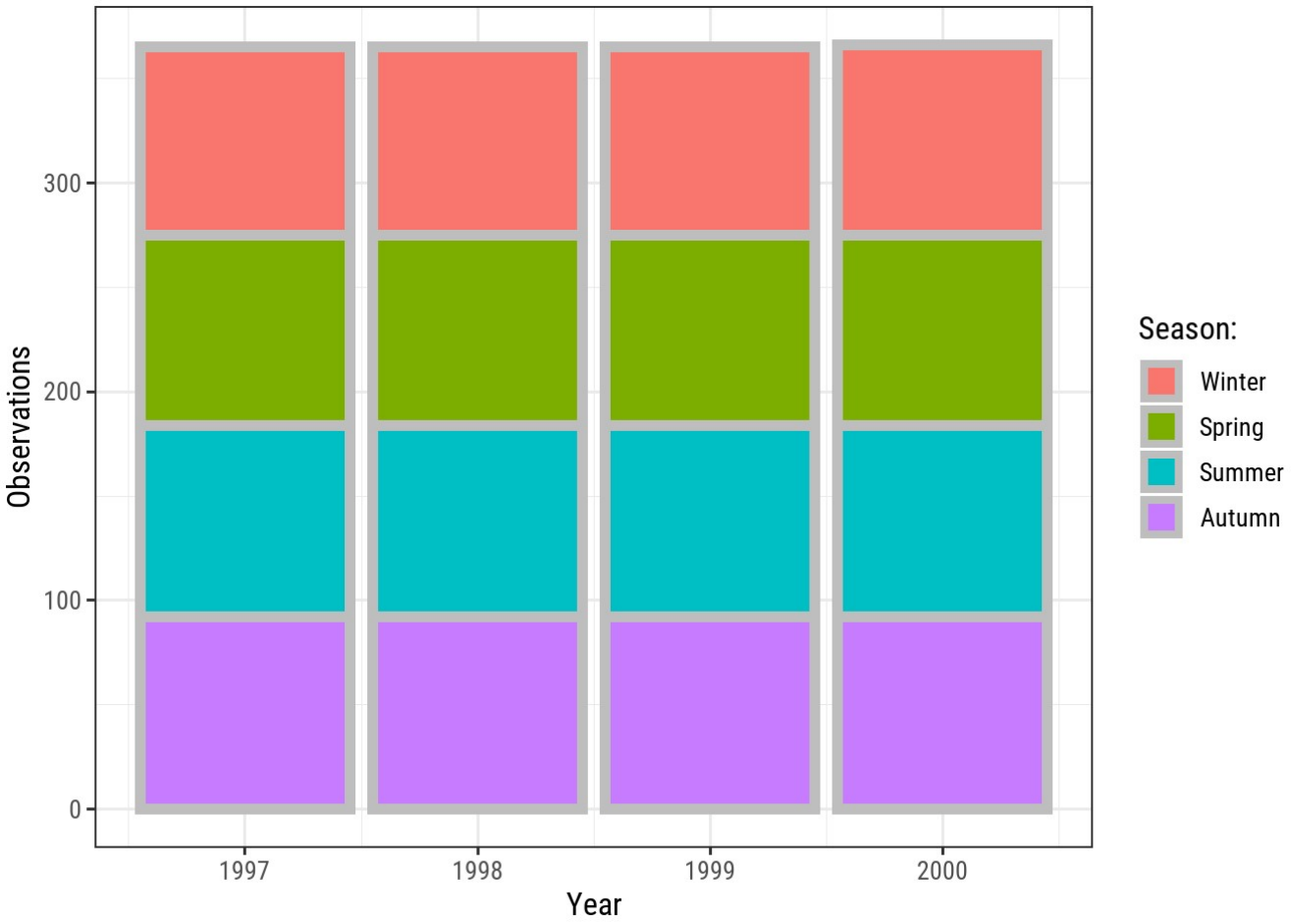
There are two main differences when it comes to colors in `{ggplot2}`. Both arguments, `color` and `fill`, can be

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 `aes`thetics 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:

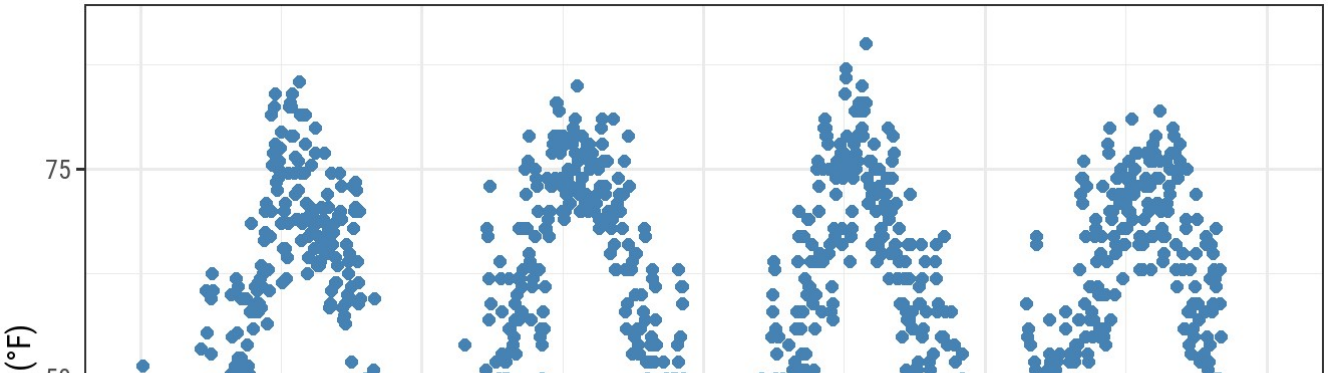
```
ggplot(chic, aes(year)) +
  geom_bar(aes(fill = season), color = "grey", size = 2) +
  labs(x = "Year", y = "Observations", fill = "Season:")
```

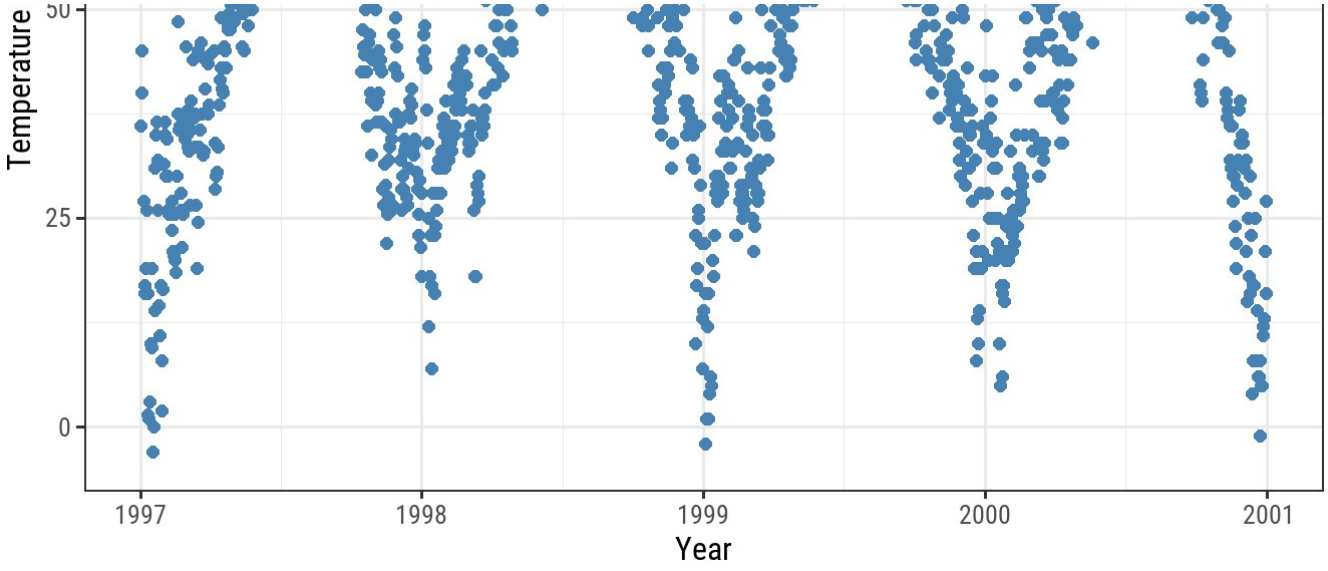


SPECIFY SINGLE COLORS

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

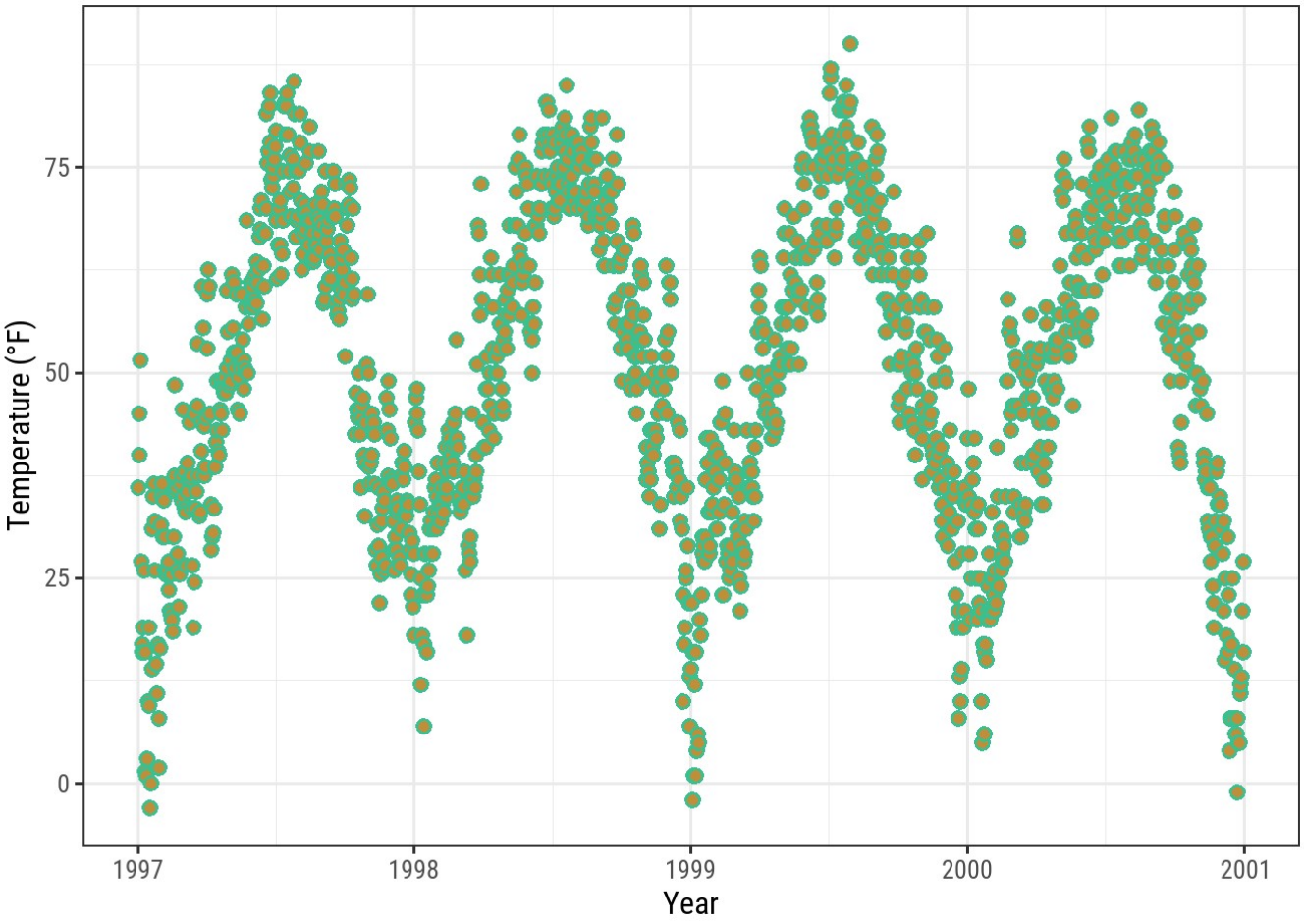
```
ggplot(chic, aes(x = date, y = temp)) +
  geom_point(color = "steelblue", size = 2) +
  labs(x = "Year", y = "Temperature (°F)")
```





... 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)")
```

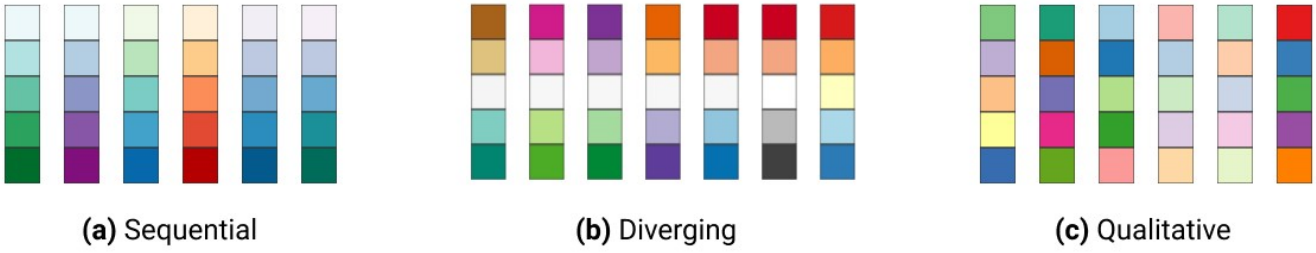


Tian Zheng at Columbia has created a useful PDF of R colors (http://www.stat.columbia.edu

/~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)`).

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:



Source: "Hands-On Data Visualization" by Jack Dougherty & Ilya Ilyankou

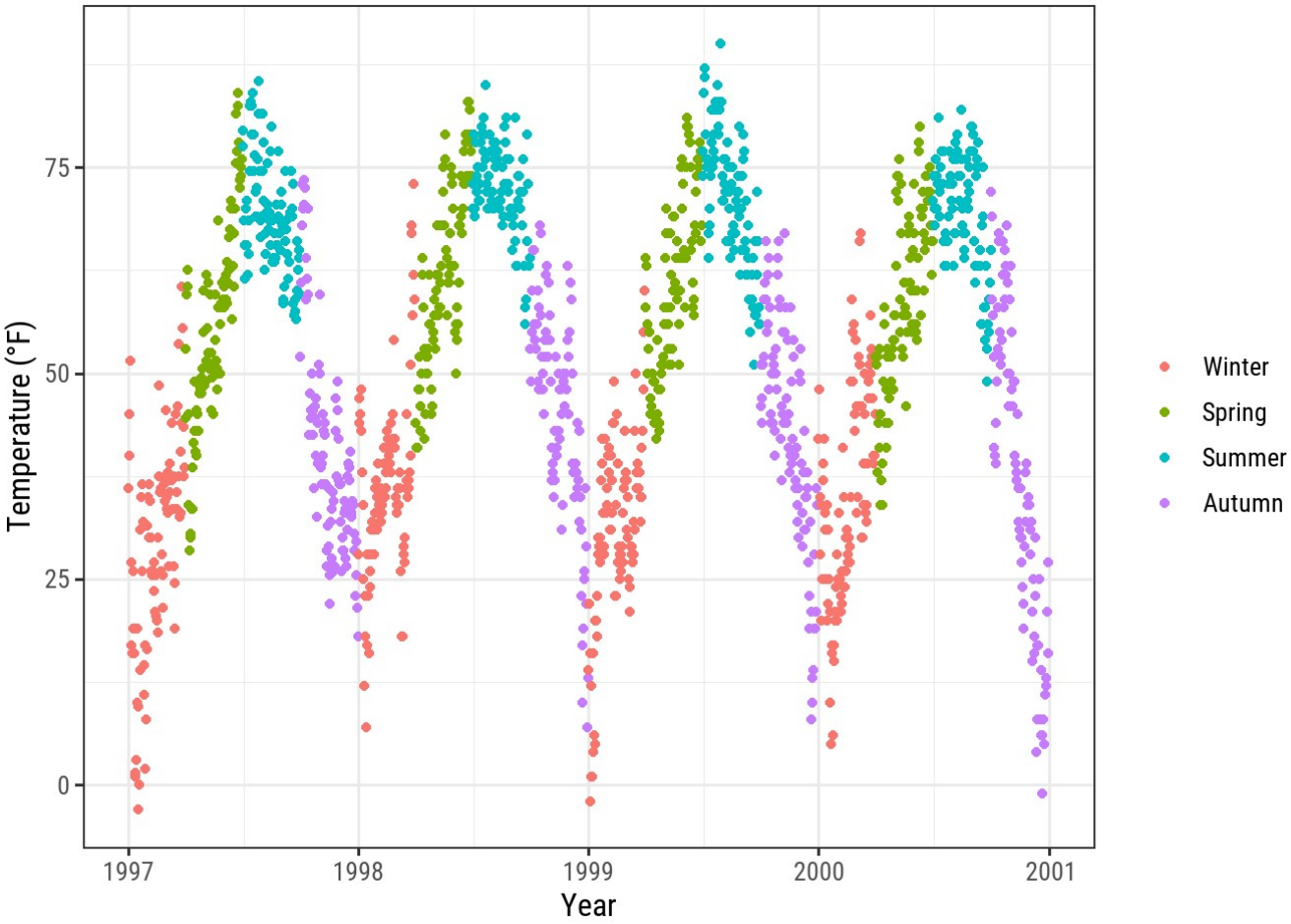
QUALITATIVE VARIABLES

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:

Artwork by Allison Horst

The default categorical color palette looks like this:

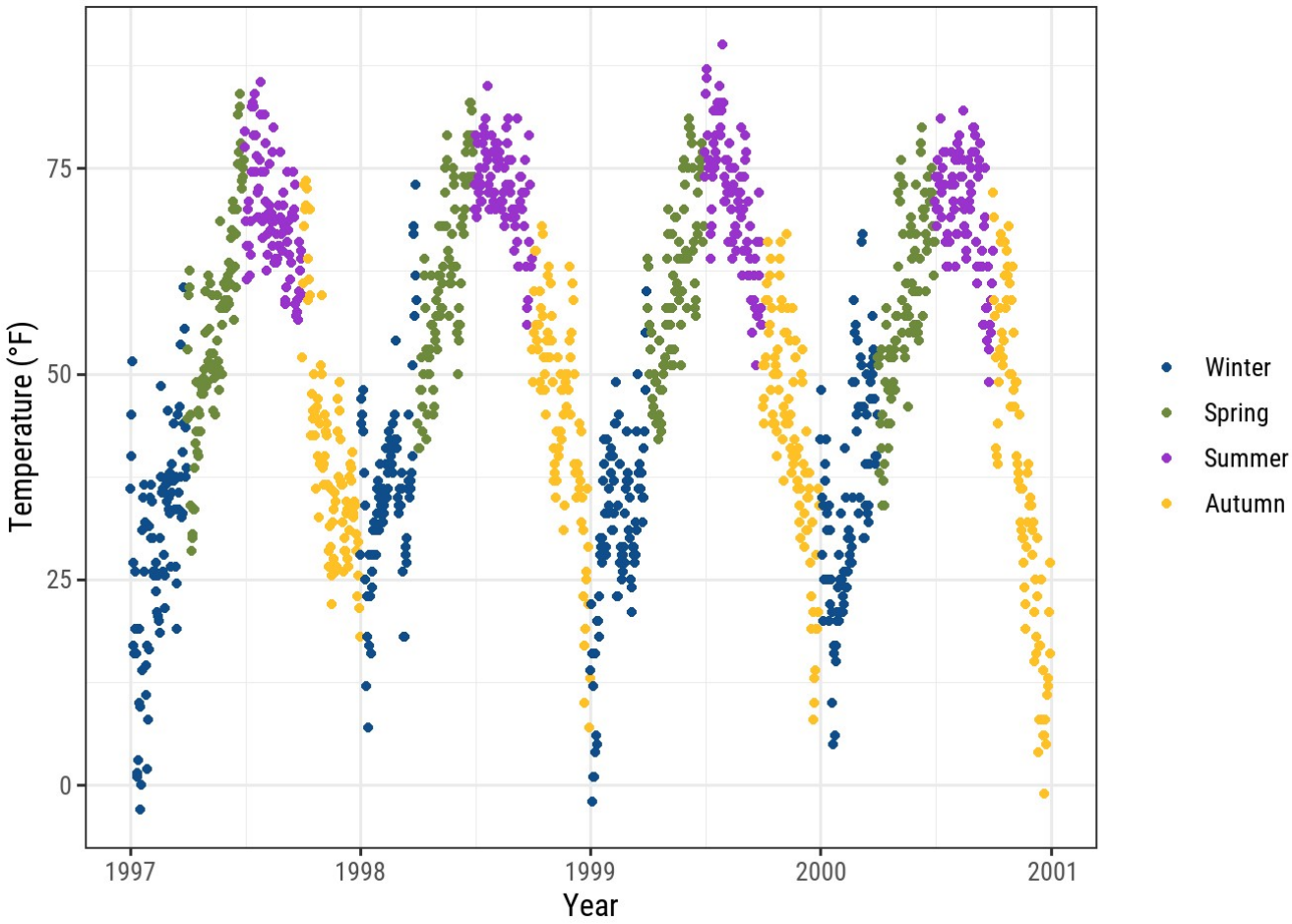
```
(ga <- ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)", color = NULL))
```



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:

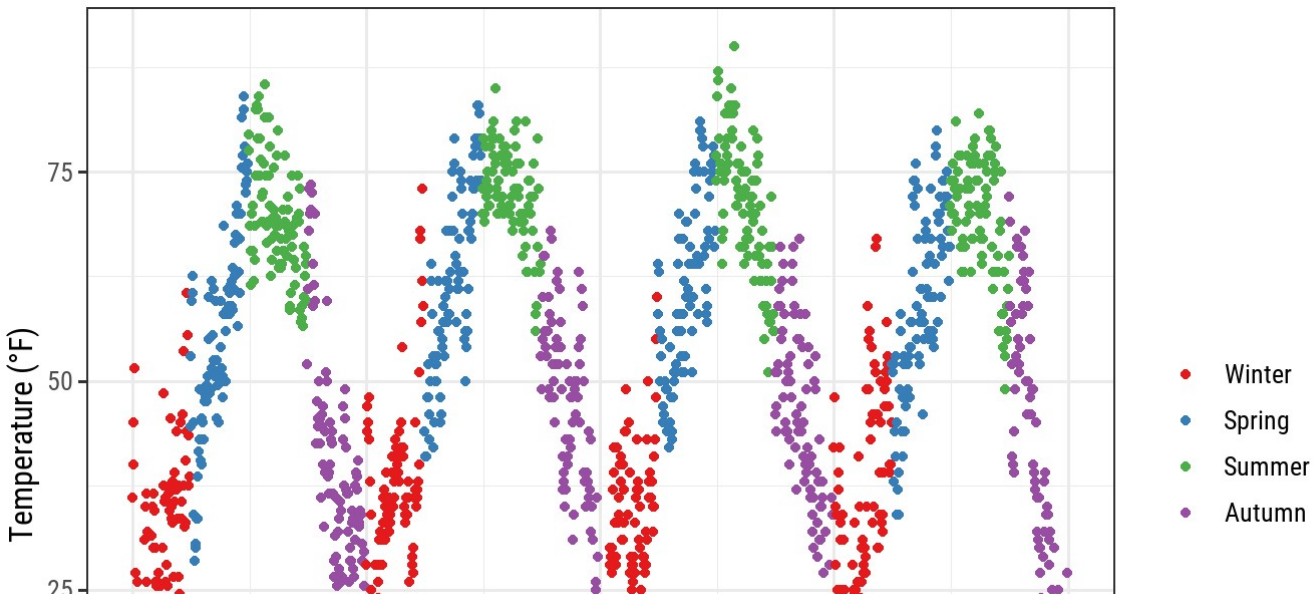
```
ga + scale_color_manual(values = c("dodgerblue4",
                                   "darkolivegreen4",
                                   "darkorchid3",
                                   "goldenrod1"))
```

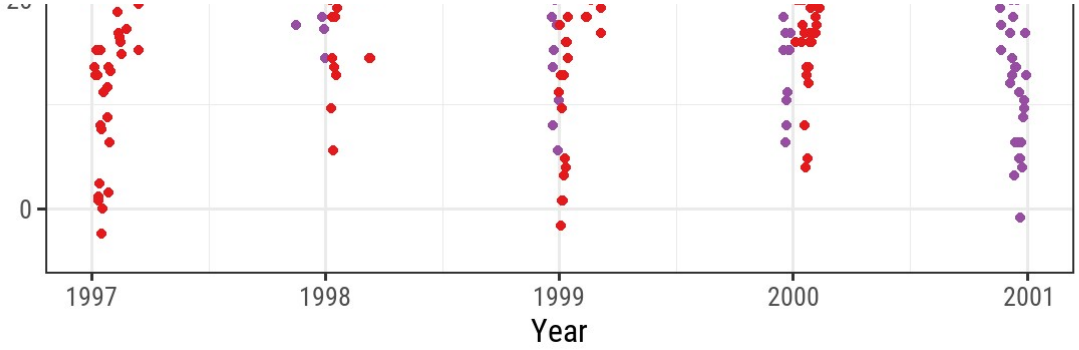


USE BUILT-IN QUALITATIVE COLOR PALETTES

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")
```





💡 **You can explore all schemes available via `RColorBrewer::display.brewer.all()`.**

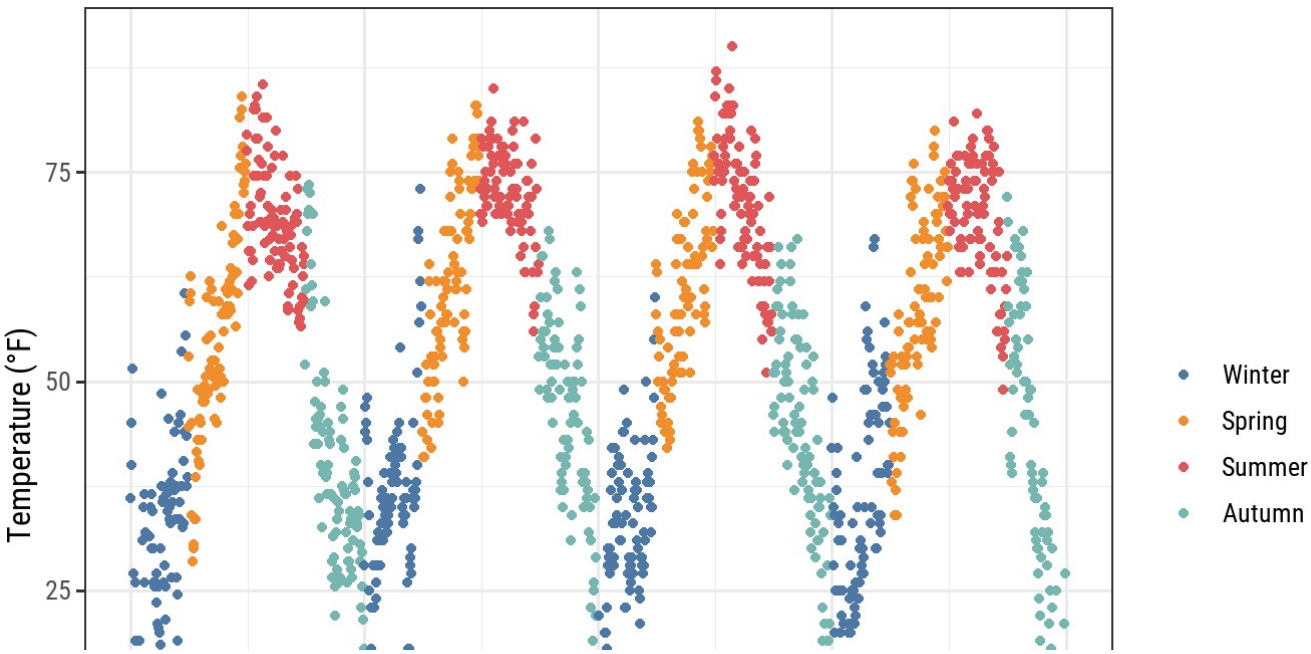
USE QUALITATIVE COLOR PALETTES FROM EXTENSION PACKAGES

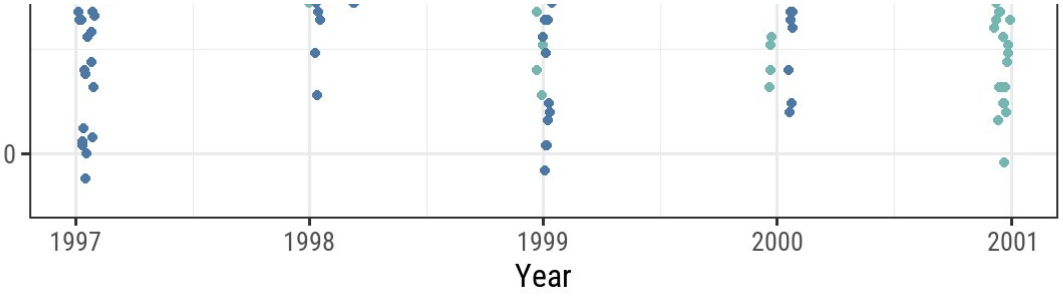
There are many extension packages that provide additional color palettes. Their use differs depending on the way the package is designed. For an extensive overview of color palettes available in R, check the collection provided by Emil Hvitfeldt (https://github.com/EmilHvitfeldt/r-color-palettes/blob/master/README.md#comprehensive-list-of-color-palettes-in-r).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

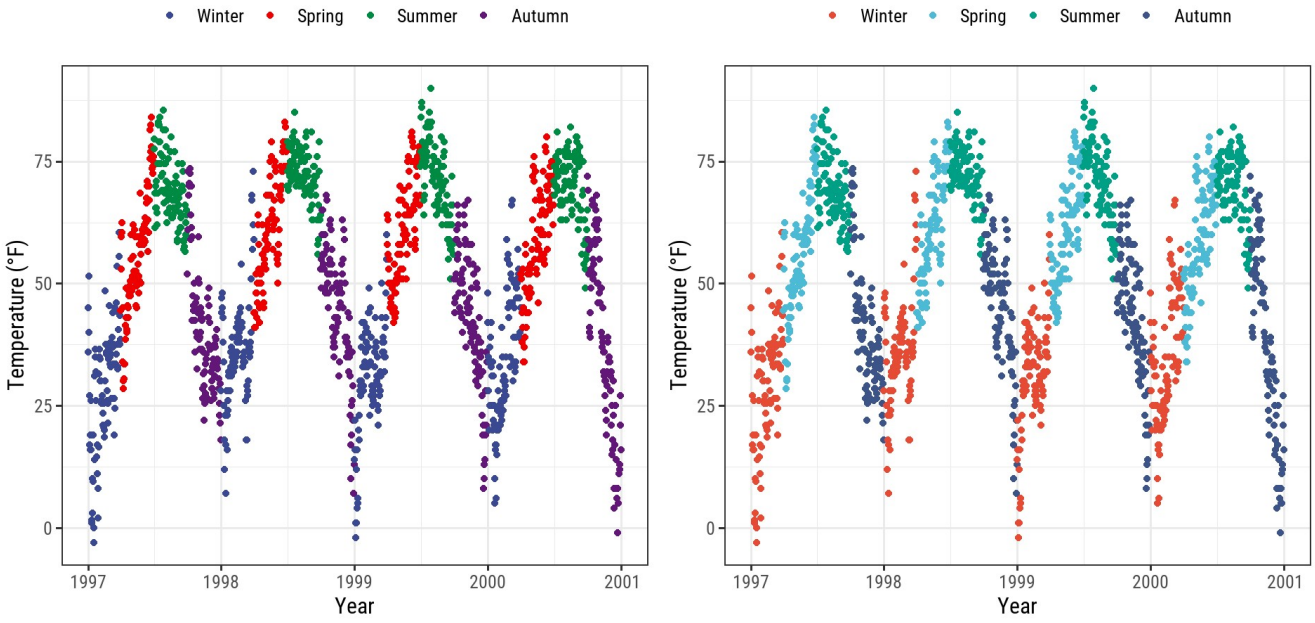
```
library(ggthemes)
ga + scale_color_tableau()
```





The `{ggsci}` package (https://cran.r-project.org/web/packages/ggsci/vignettes/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")
```



QUANTITATIVE VARIABLES

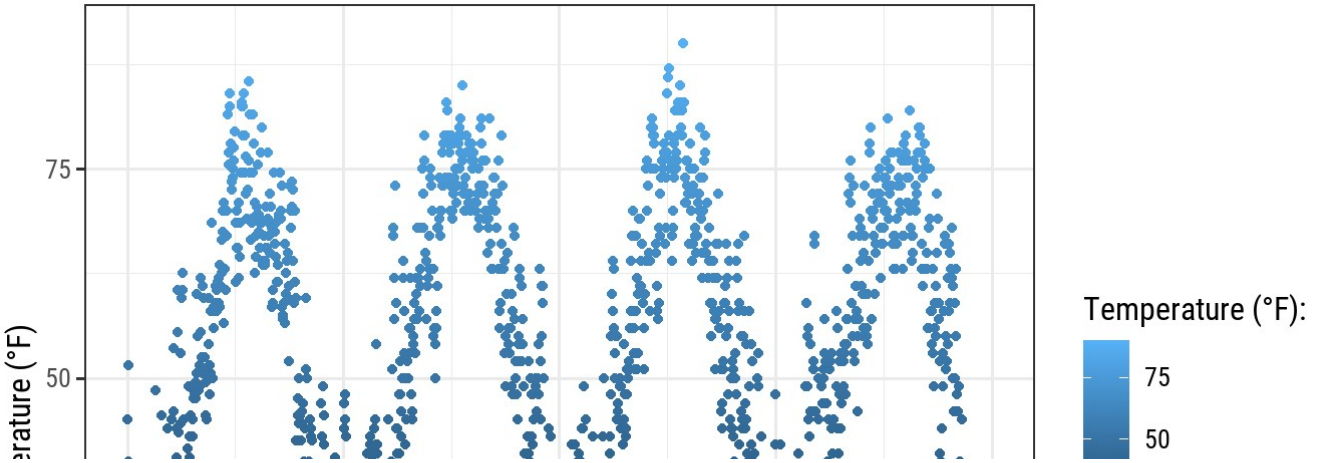
Quantitative variables represent a measurable quantity and are thus numerical. Quantitative data can be further classified as being either continuous (floating numbers possible) or discrete (integers only):

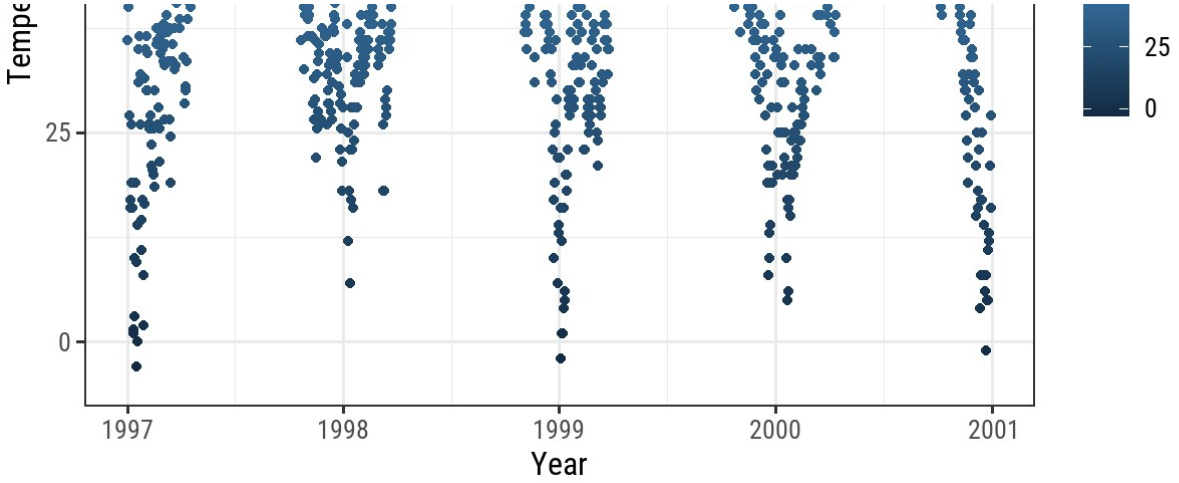
Artwork by Allison Horst

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()
```



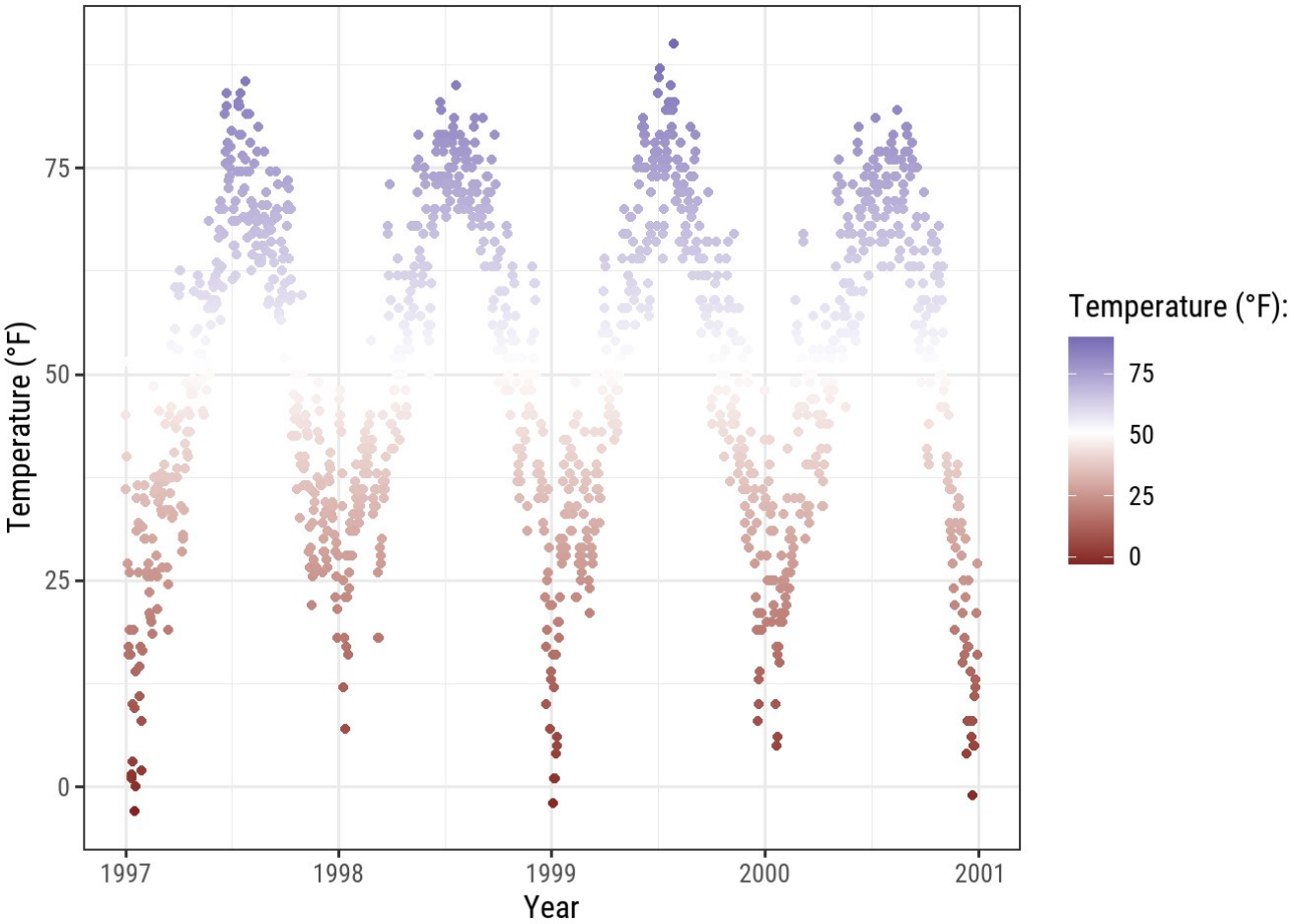


This code produces the same plot:

```
gb + scale_color_gradient()
```

And here is the diverging default color scheme:

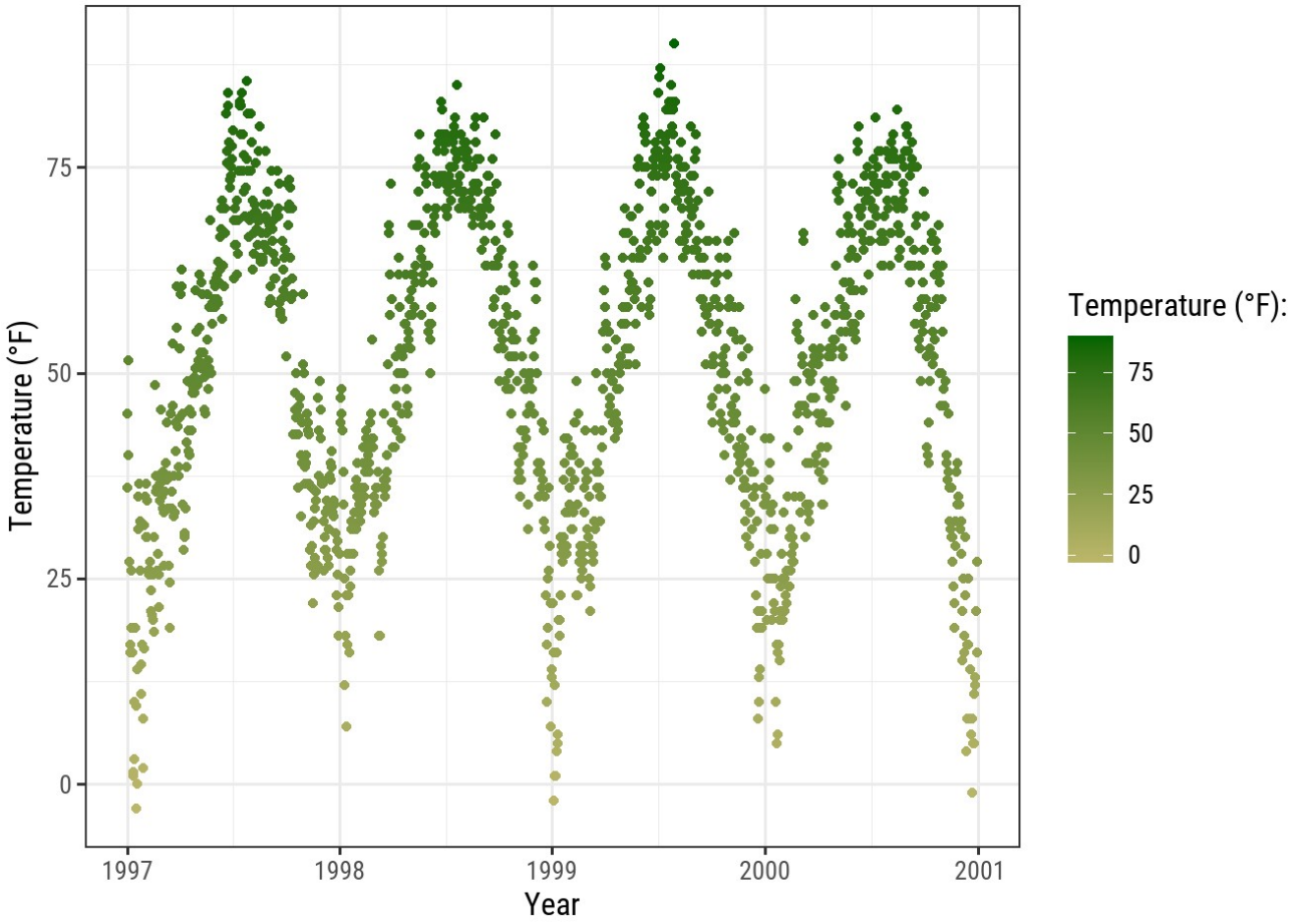
```
mid <- mean(chic$temp)  ## midpoint

gb + scale_color_gradient2(midpoint = mid)
```



MANUALLY SET A SEQUENTIAL COLOR SCHEME

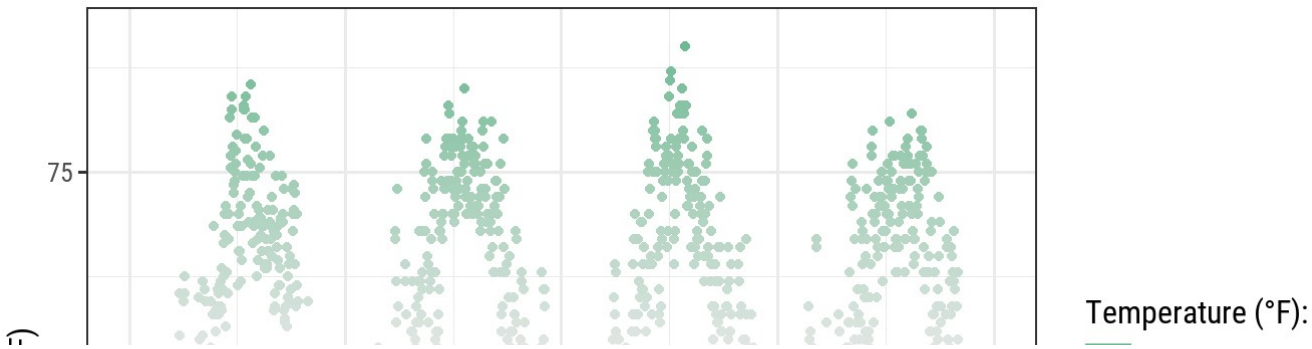
You can manually set gradually changing color palettes for continuous variables via `scale_*_gradient()`:

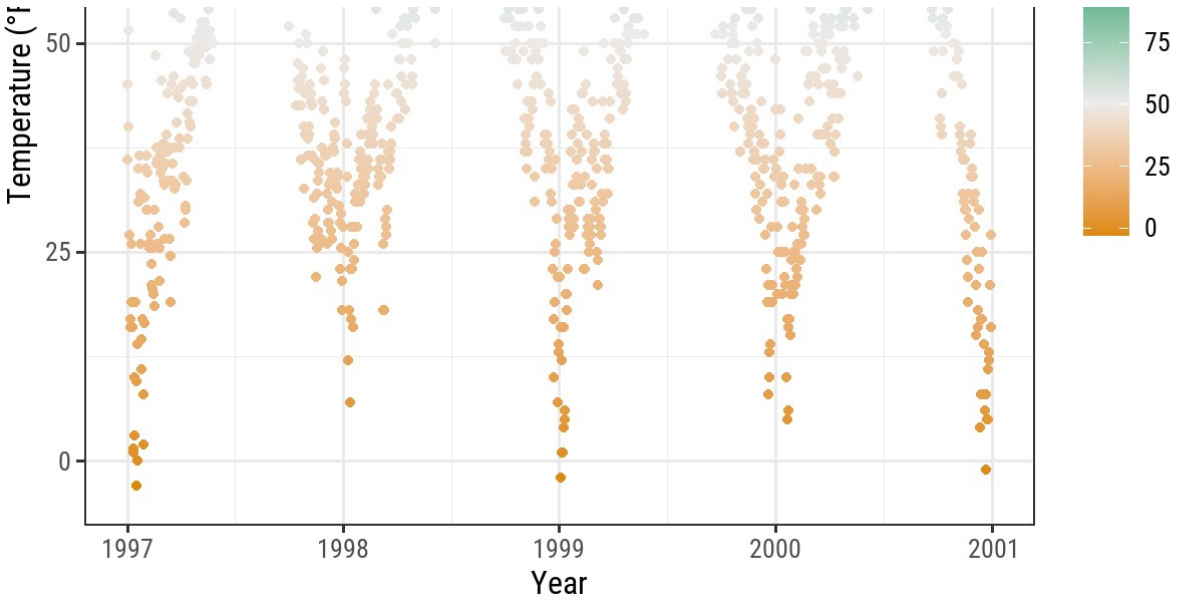
```
gb + scale_color_gradient(low = "darkkhaki",
                          high = "darkgreen")
```



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")
```



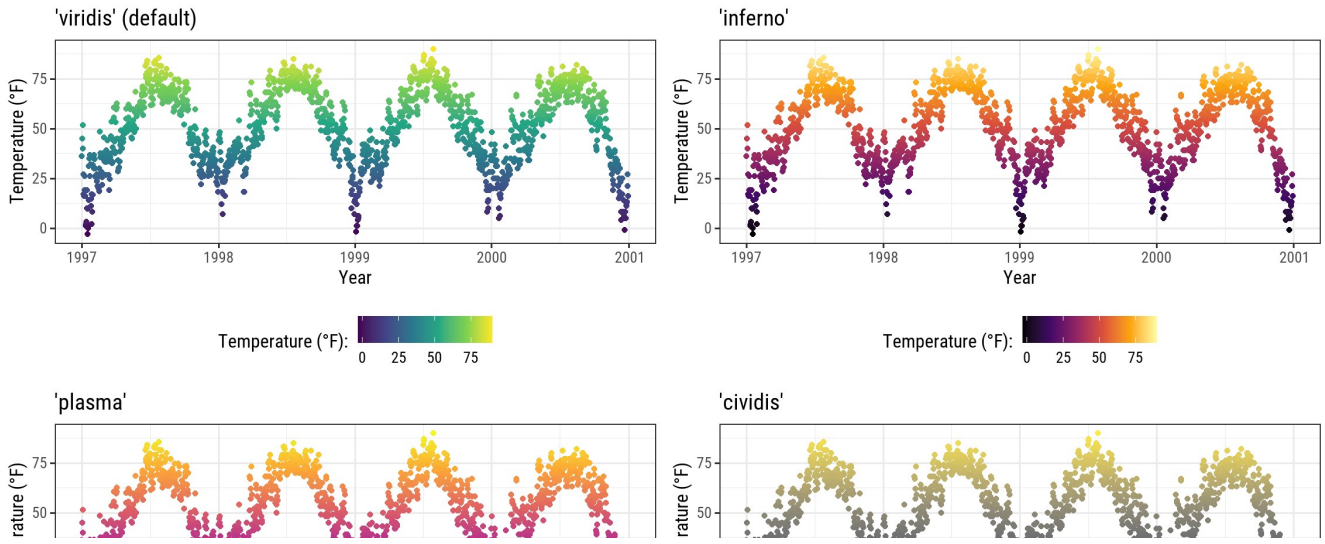


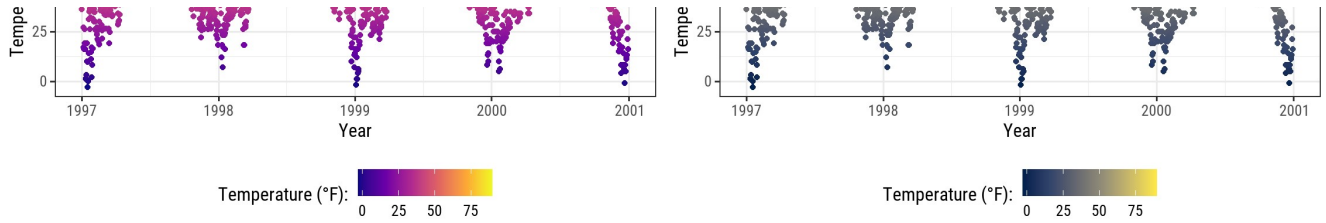
THE BEAUTIFUL VIRIDIS COLOR PALETTE

The **viridis** color palettes (https://cran.r-project.org/web/packages/viridis/vignettes/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 `dichromate` (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:

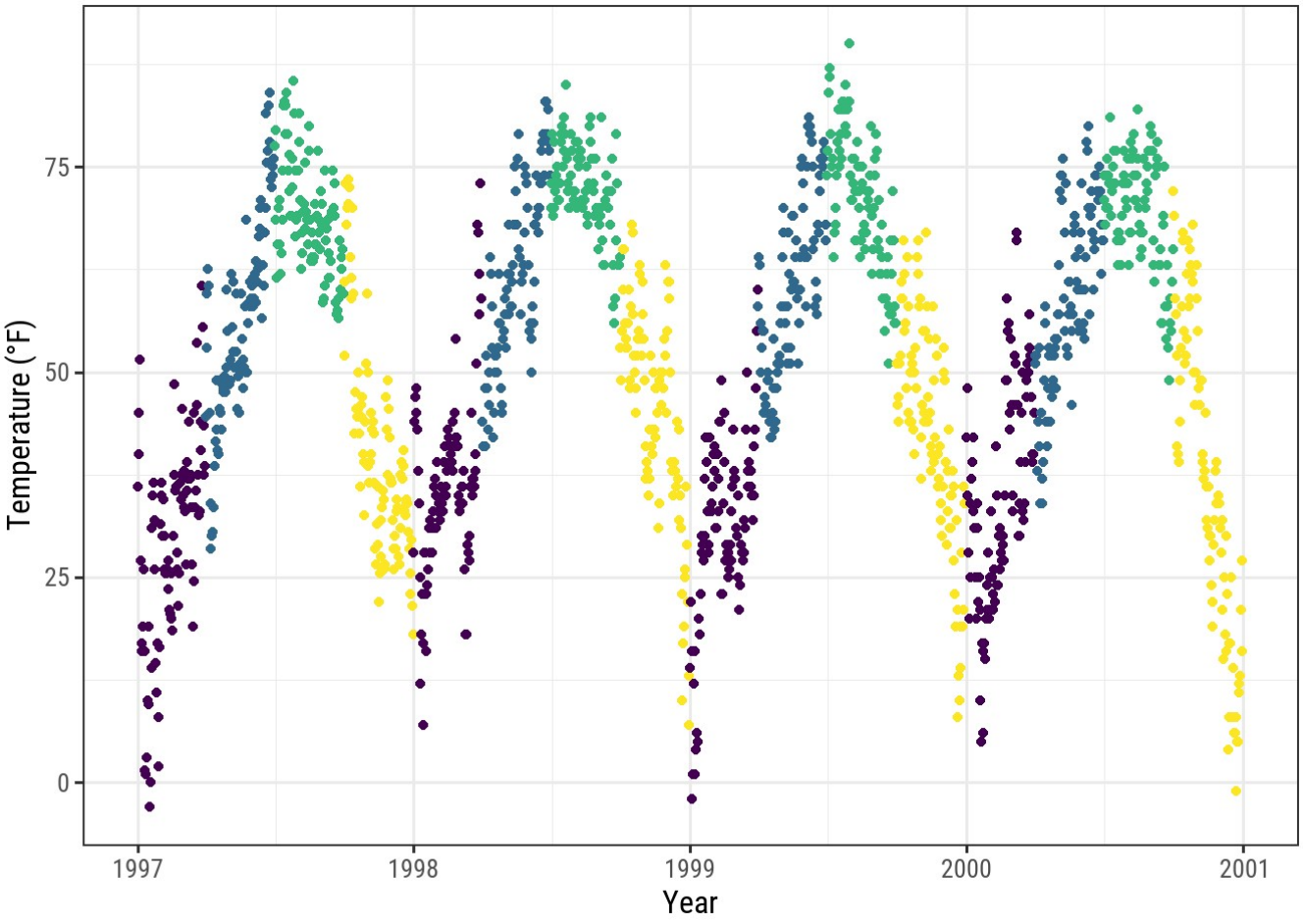
```
p1 <- gb + scale_color_viridis_c() + ggtitle("'viridis' (default)")
p2 <- gb + scale_color_viridis_c(option = "inferno") + ggtitle("'inferno'")
p3 <- gb + scale_color_viridis_c(option = "plasma") + ggtitle("'plasma'")
p4 <- gb + scale_color_viridis_c(option = "cividis") + ggtitle("'cividis'")

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")
```



USE QUANTITATIVE COLOR PALETTES FROM EXTENSION PACKAGES

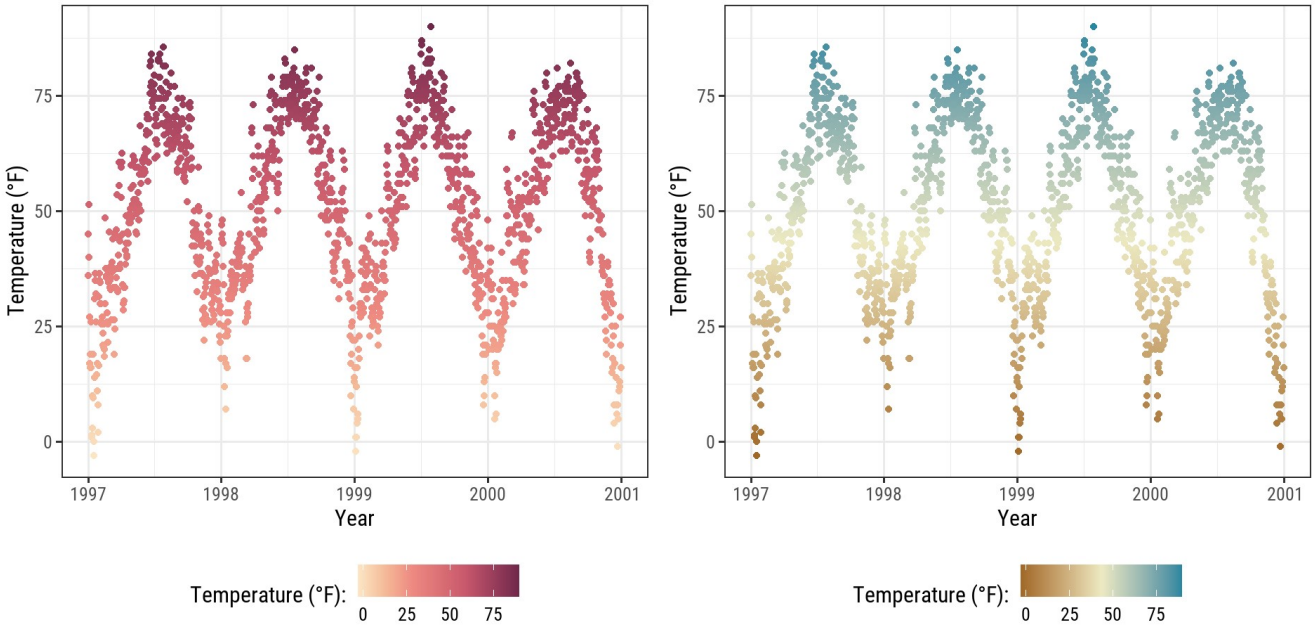
The many extension packages provide not only additional categorical color palettes but also sequential, diverging and even cyclical palettes. Again, I point you to the great collection provided by Emil Hvitfeldt (https://github.com/EmilHvitfeldt/r-color-palettes/blob/master/README.md#comprehensive-list-of-color-palettes-in-r)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

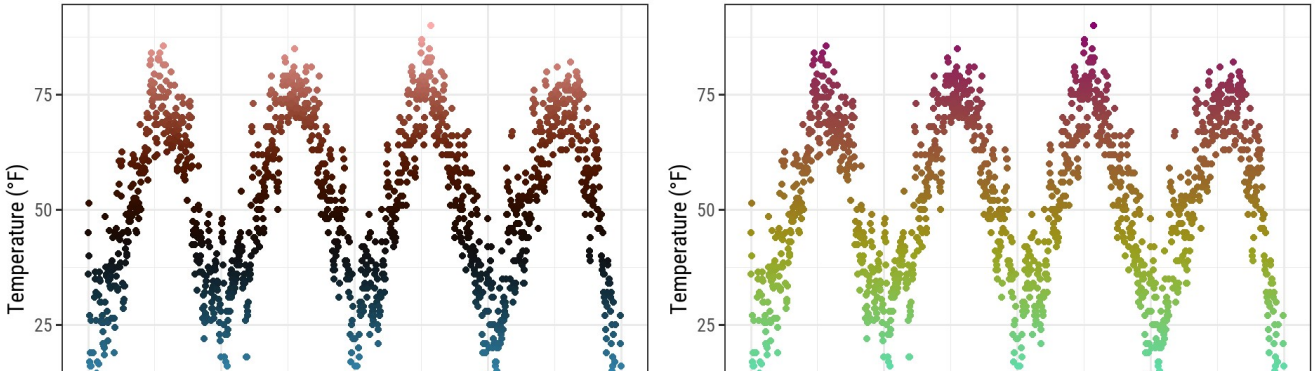
and contains several of my most-used palettes:

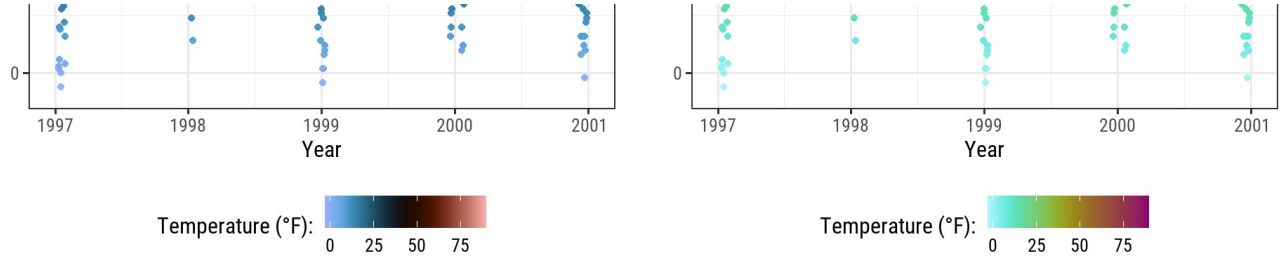
```
library(rcartocolor)
g1 <- gb + scale_color_carto_c(palette = "BurgYl")
g2 <- gb + scale_color_carto_c(palette = "Earth")

(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")
```



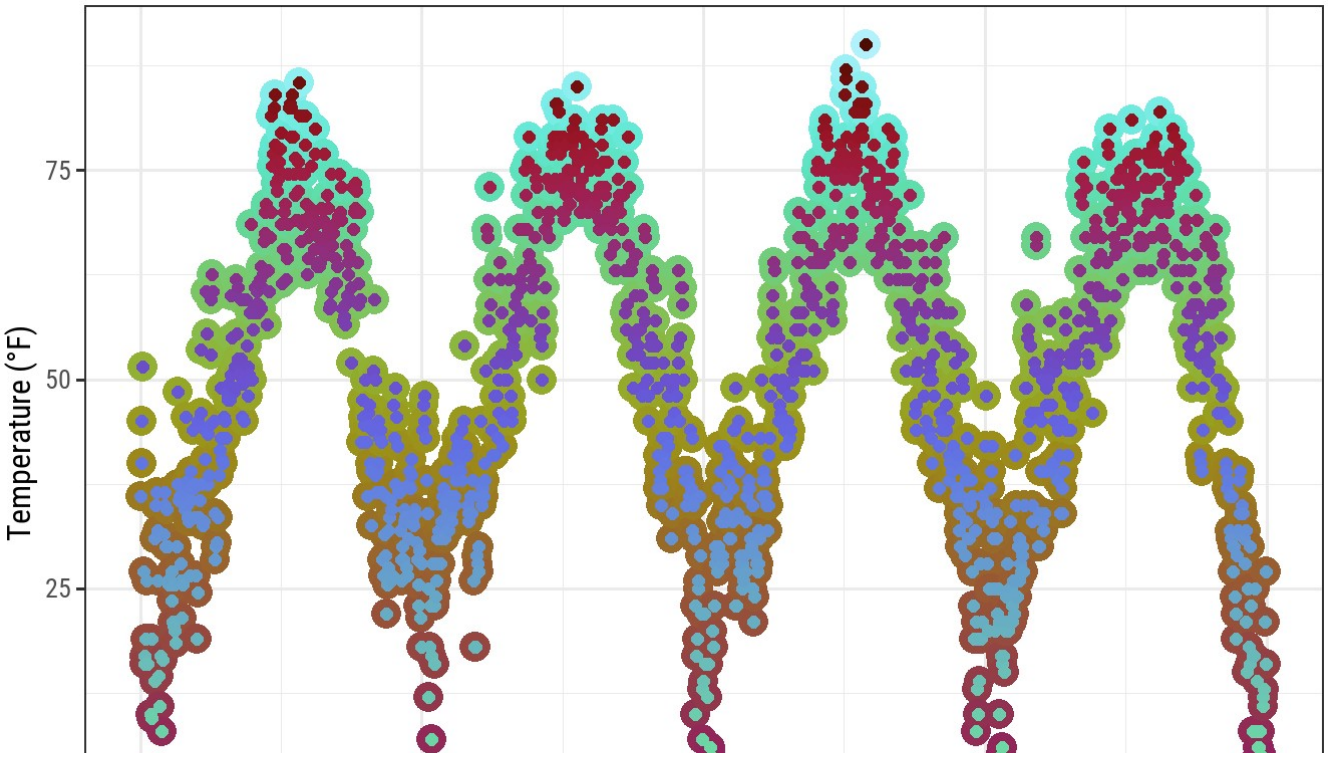


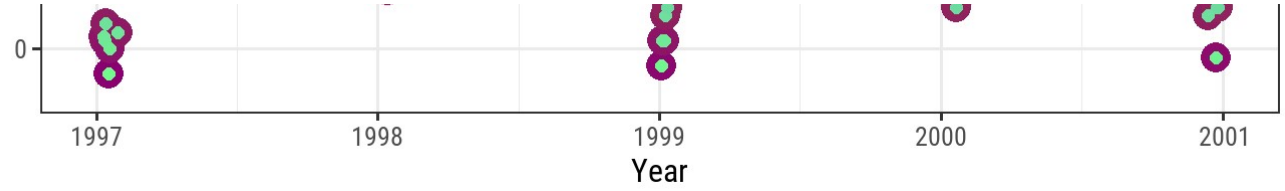
MODIFY COLOR PALETTES AFTERWARDS

Since the latest 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 `invert_color()` from the `{ggdark}` package (https://github.com/nsgrantham/ggdark):

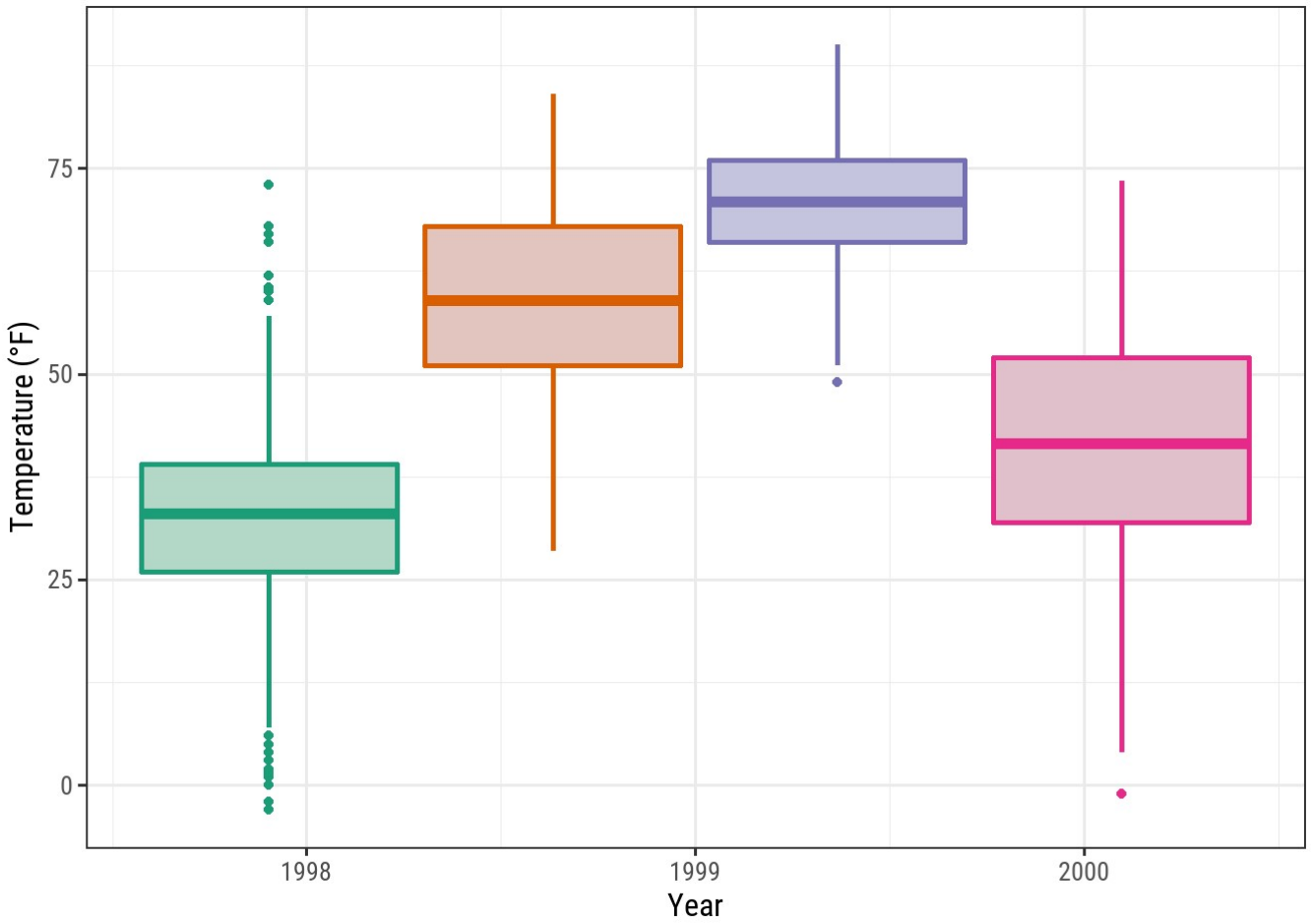
```
library(ggdark)

ggplot(chic, aes(date, temp, color = temp)) +
  geom_point(size = 5) +
  geom_point(aes(color = temp,
                 color = after_scale(invert_color(color))),
             size = 2) +
  scale_color_scico(palette = "hawaii", guide = "none") +
  labs(x = "Year", y = "Temperature (°F)")
```





Changing the color scheme afterwards is especially fun with functions from the `{ggdark}` and `{colorspace}` packages, namely `invert_color()`, `lighten()`, `darken()` and `desature()`. You can even combine those functions. Here, we plot a box plot that has both arguments, `color` and `fill`:

```
library(colorspace)

ggplot(chic, aes(date, temp)) +
  geom_boxplot(aes(color = season,
                   fill = after_scale(desaturate(lighten(color, .6), .6))),
               size = 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.

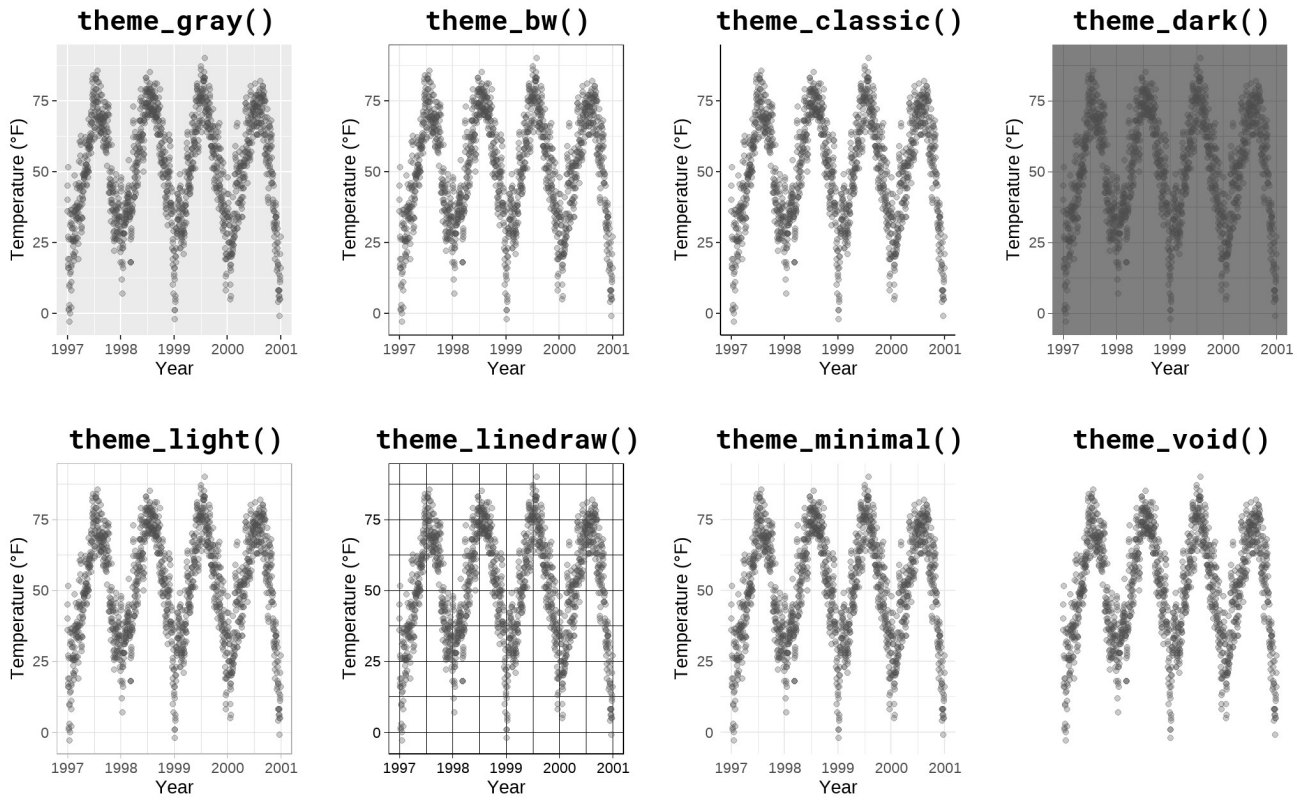
💡 **This seems a bit complicated for now—one could simply use the `color` and `fill` scales for both. Yes, that is true but think about use cases where you need several `color` and/or `fill` scales. In such a case, it would be senseless to occupy the `fill` scale with a slightly darker version of the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

↑ Jump back to Table of Content.

WORKING WITH THEMES

CHANGE THE OVERALL PLOTTING STYLE

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

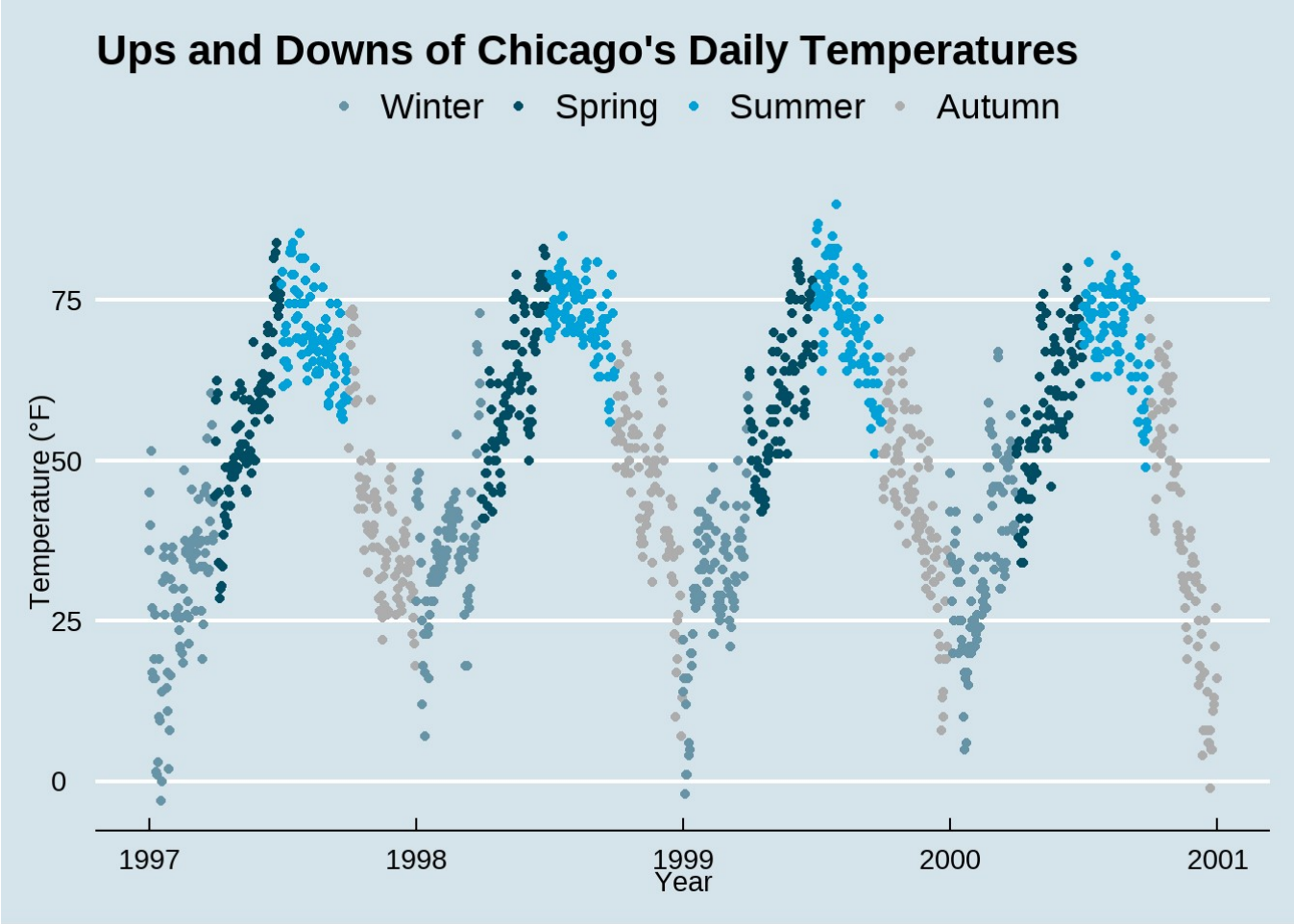


There are several packages that provide additional themes, some even with different default color palettes. As an example, Jeffrey Arnold has put together the library `{ggthemes}` with several custom themes imitating popular designs. For a list you can visit the `{ggthemes}` package site (https://github.com/jrnold/ggthemes). 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/)

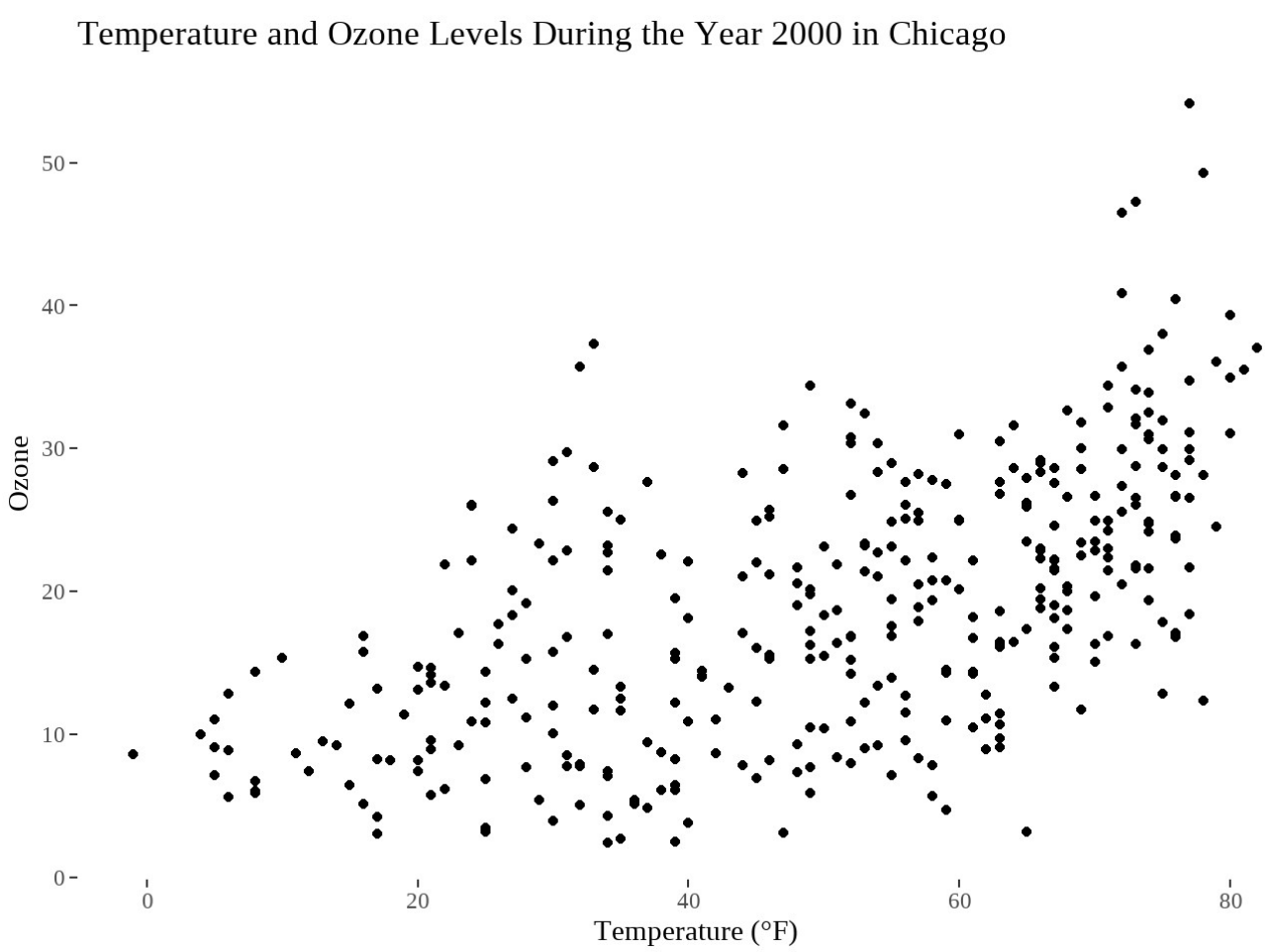
magazine by using `theme_economist()` and `scale_color_economist()`:

```
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:

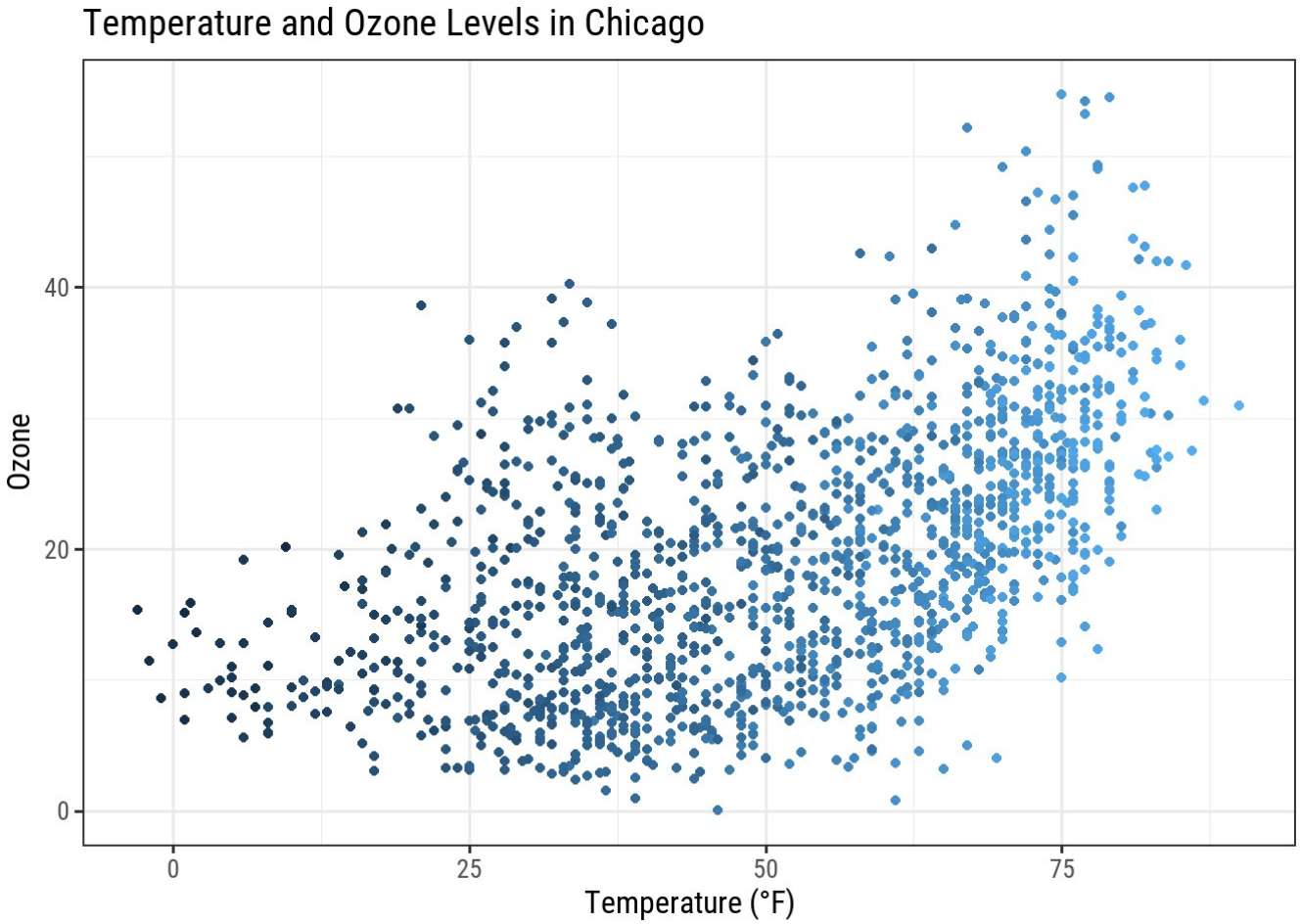
```
library(dplyr)
chic_2000 <- filter(chic, year == 2000)

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:

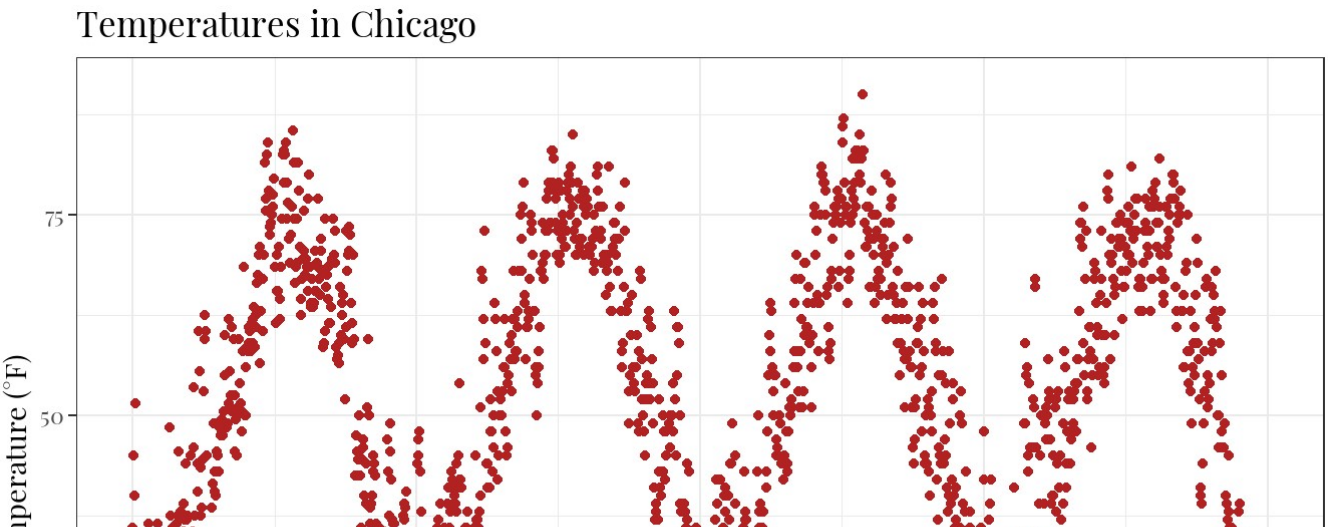
```
library(hrbrthemes)

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")
```

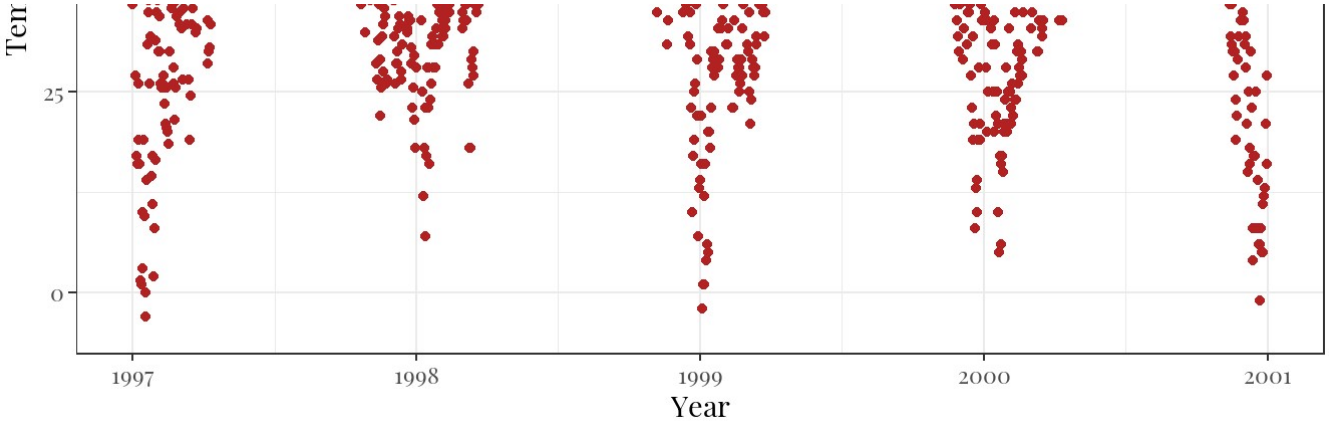


CHANGE THE FONT OF ALL TEXT ELEMENTS

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")
```

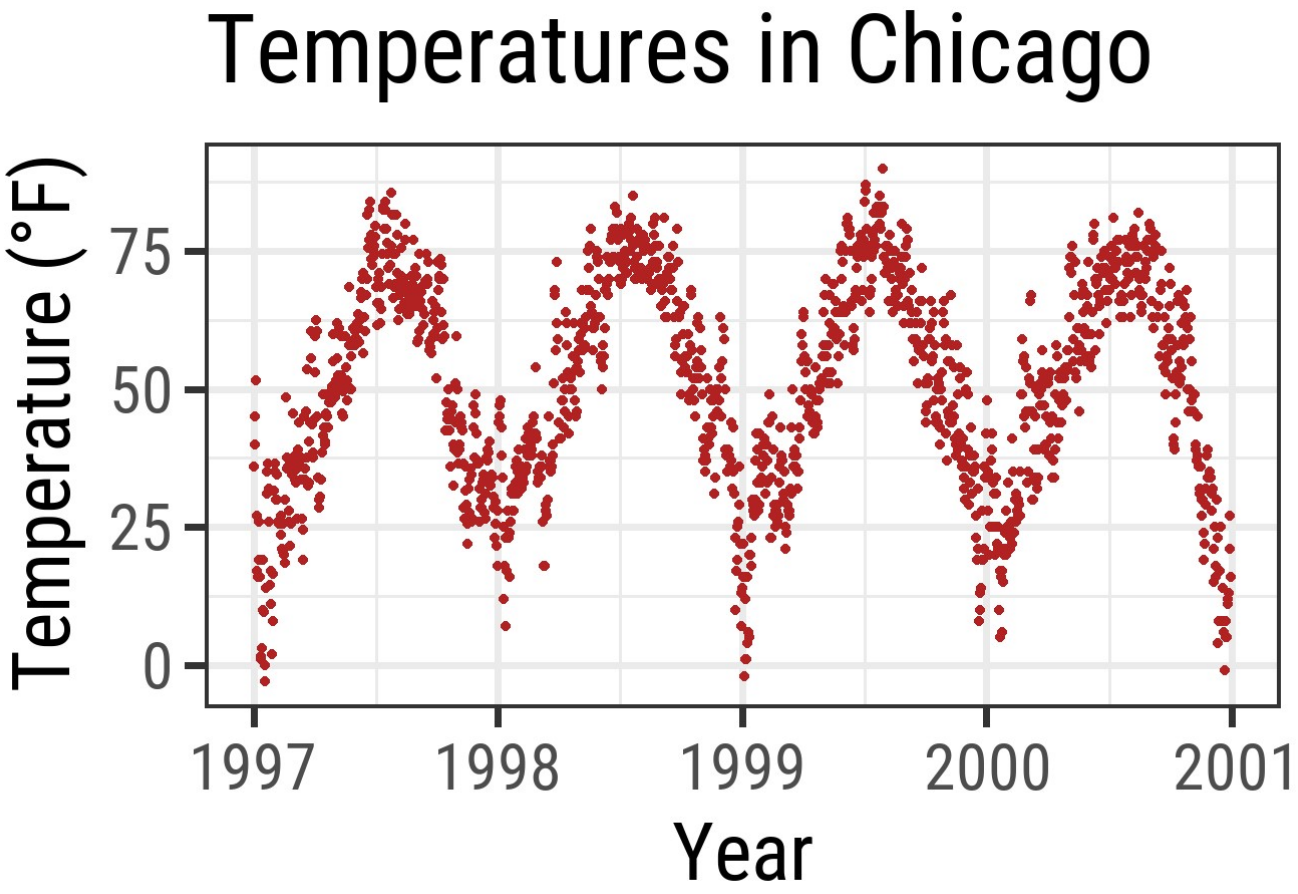




CHANGE THE SIZE OF ALL TEXT ELEMENTS

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:

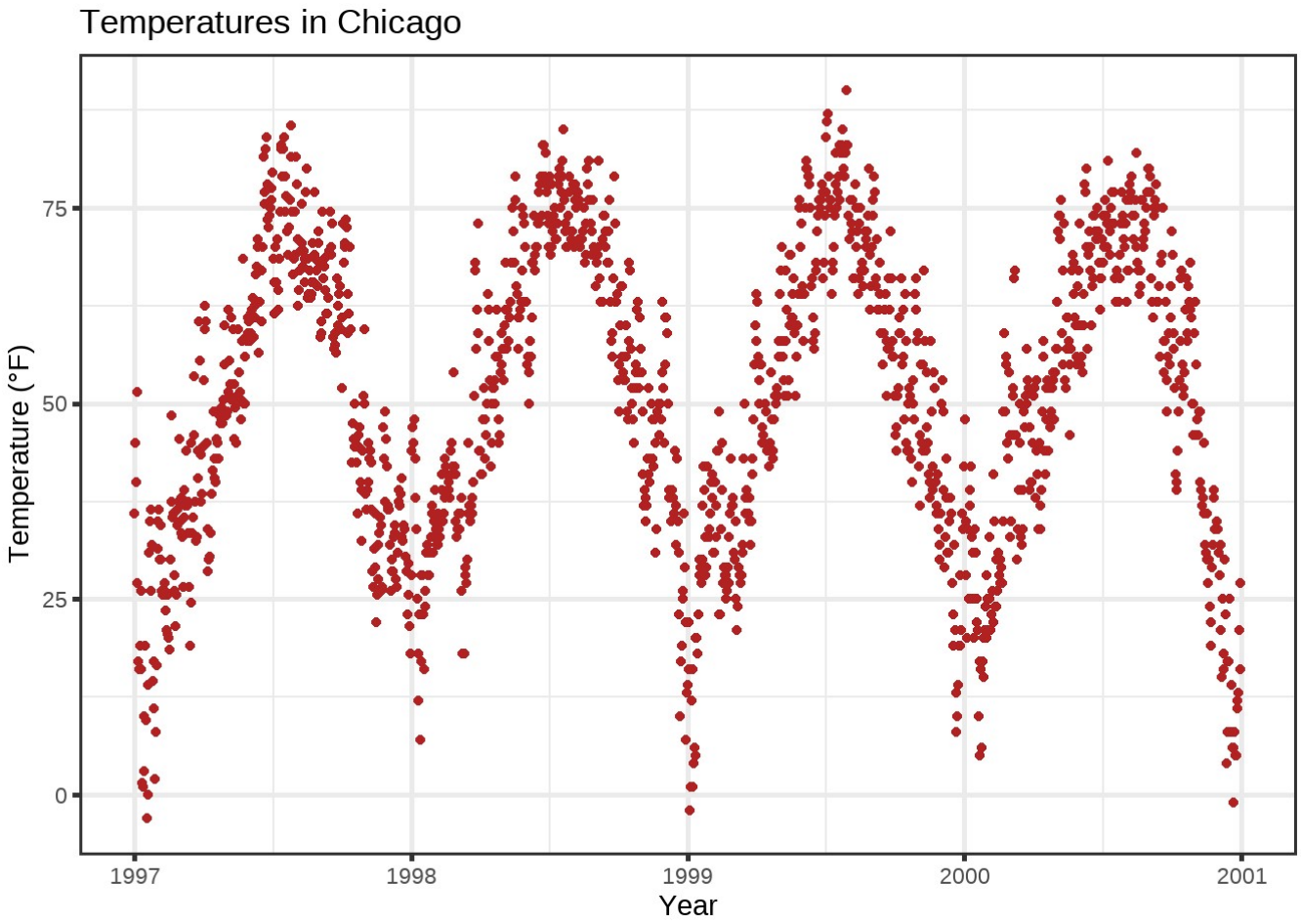
```
g + theme_bw(base_size = 30, base_family = "Roboto Condensed")
```



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)
```



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", size = base_line_size,
##         linetype = 1, lineend = "butt"), rect = element_rect(fill = "white",
##         colour = "black", size = 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 =
##             half_line/2), vjust = 0), axis.text.y = element_text(margin = margin(r = 0.
##             half_line/2), hjust = 1), axis.text.y.right = element_text(margin = margin(
##             half_line/2), hjust = 0), 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.title.x = element_text(margin = margin(t
##             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_
##             margin = margin(l = half_line/2), vjust = 0), legend.background = element_r
##         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 = element_rect(fill = "grey95"
##             colour = NA), legend.key.size = unit(1.2, "lines"),
##         legend.key.height = NULL, legend.key.width = NULL, legend.text = element_text(s
##         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(2 * half_line, "pt"), panel.background = element_rect
##             colour = NA), panel.border = element_blank(), panel.grid = element_line(col
##         panel.grid.minor = element_line(size = rel(0.5)), panel.spacing = unit(half_lin
##             "pt"), panel.spacing.x = NULL, panel.spacing.y = NULL,
##         panel.ontop = FALSE, strip.background = element_rect(fill = "grey85",
##             colour = NA), 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 = uni
##             "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_lin
##         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: 0x0000000024e08af0>
## <environment: namespace:ggplot2>
```

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

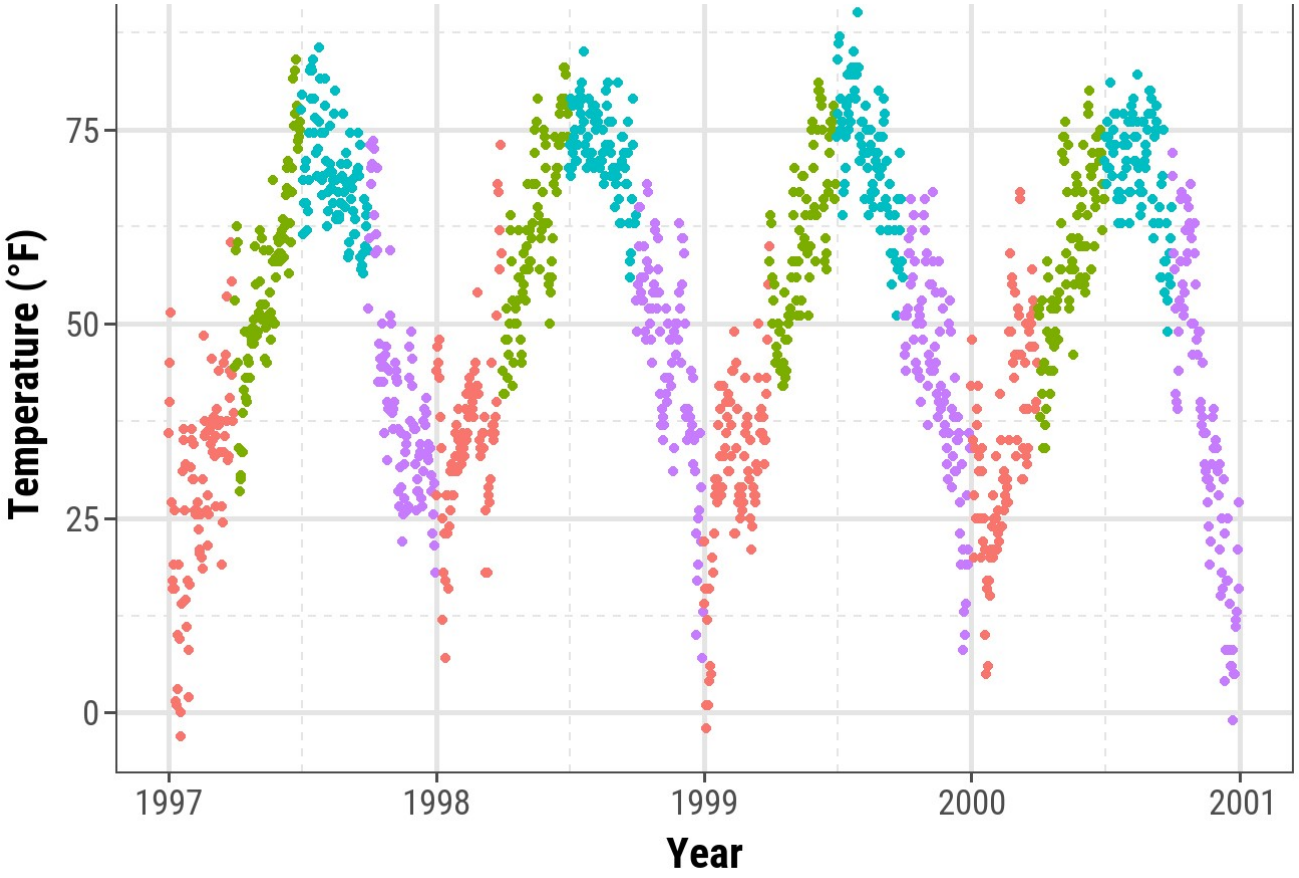
```
theme_custom <- function (base_size = 12, base_family = "Roboto Condensed") {
  half_line <- base_size/2
  theme(
    line = element_line(color = "black", size = .5,
                        linetype = 1, lineend = "butt"),
    rect = element_rect(fill = "white", color = "black",
                        size = .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", size = .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, size = .7),
    panel.grid.major = element_line(color = "gray90", size = 1),
    panel.grid.minor = element_line(color = "gray90", size = .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(base_size, base_size, base_size, base_size),
    complete = TRUE
  )
}
```

💡 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* change 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_custom())

ggplot(chic, aes(x = date, y = temp, color = season)) +
  geom_point() + labs(x = "Year", y = "Temperature (°F)") + guides(color = FALSE)
```

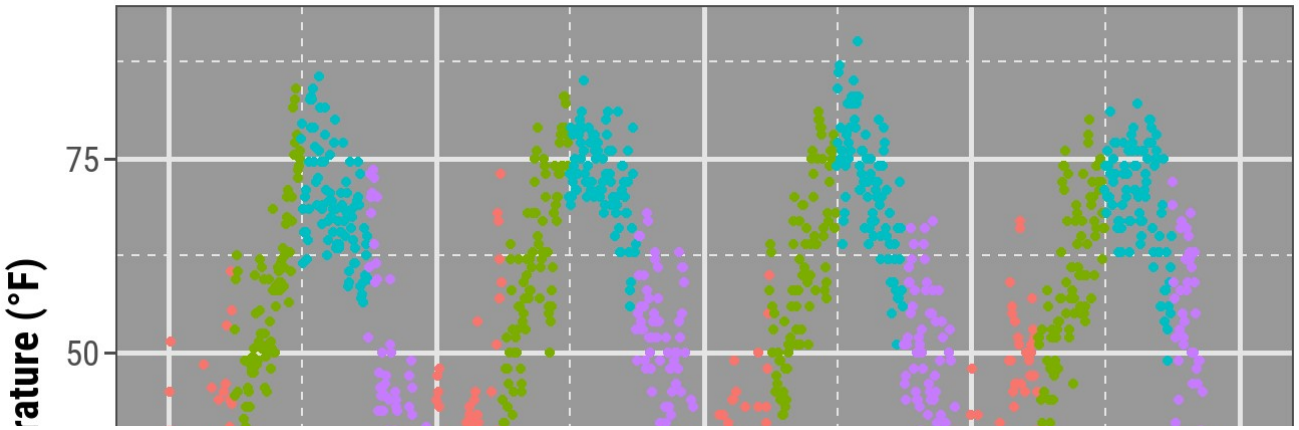


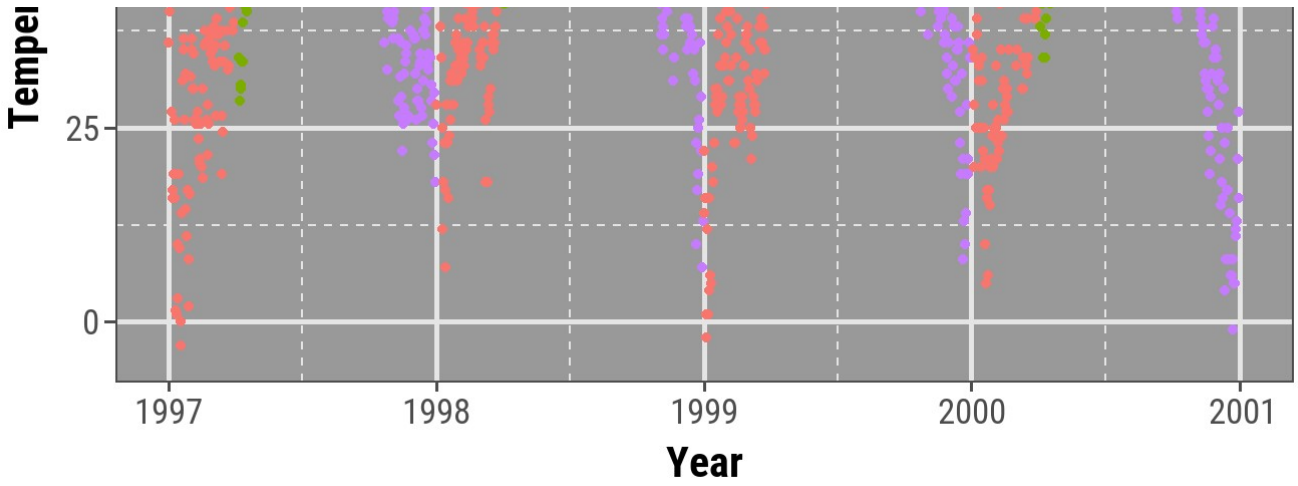
This way of changing the plot design is highly recommended! It allows you to quickly change any element of your plots by changing it once. You can within a few seconds plot all your results in a congruent style and adapt it to other needs (e.g. a presentation with bigger font size or journal requirements).

UPDATE THE CURRENT THEME

You can also set quick changes using `theme_update()`:

```
theme_custom <- 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 = FALSE)
```





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

```
theme_custom <- theme_update(panel.background = element_rect(fill = "white"),
                             panel.grid.major = element_line(size = .5),
                             panel.grid.minor = element_blank())
```

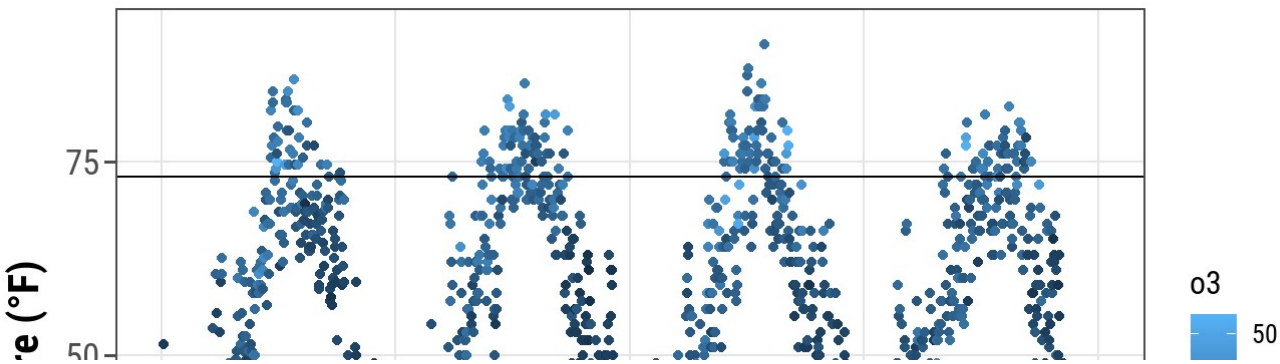
↑ Jump back to Table of Content.

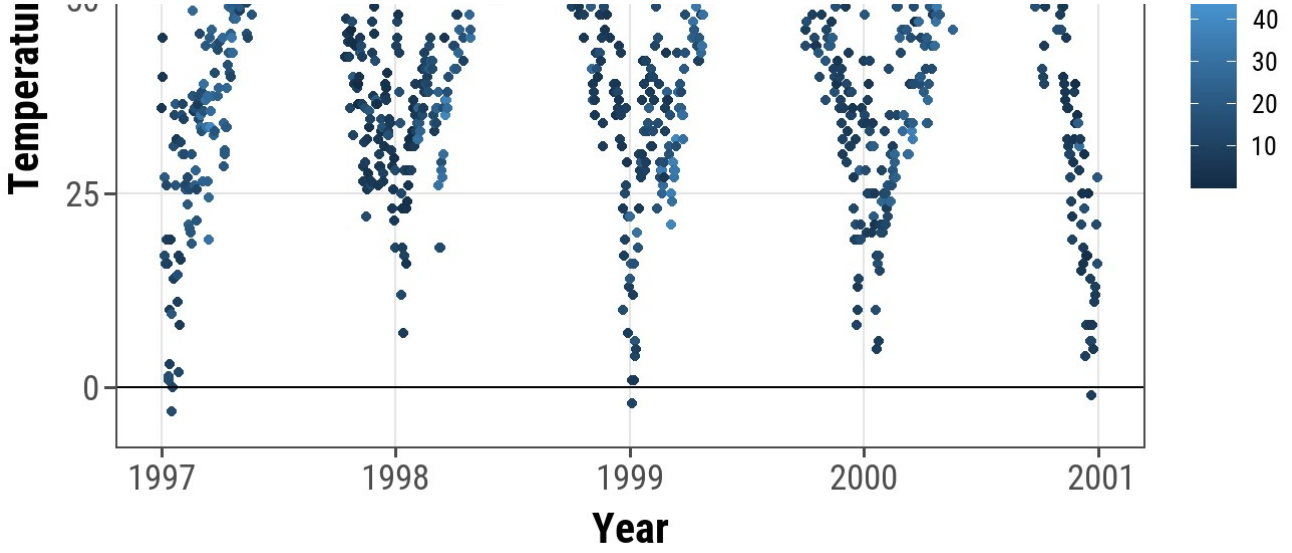
WORKING WITH LINES

ADD HORIZONAL OR VERTICAL LINES TO A PLOT

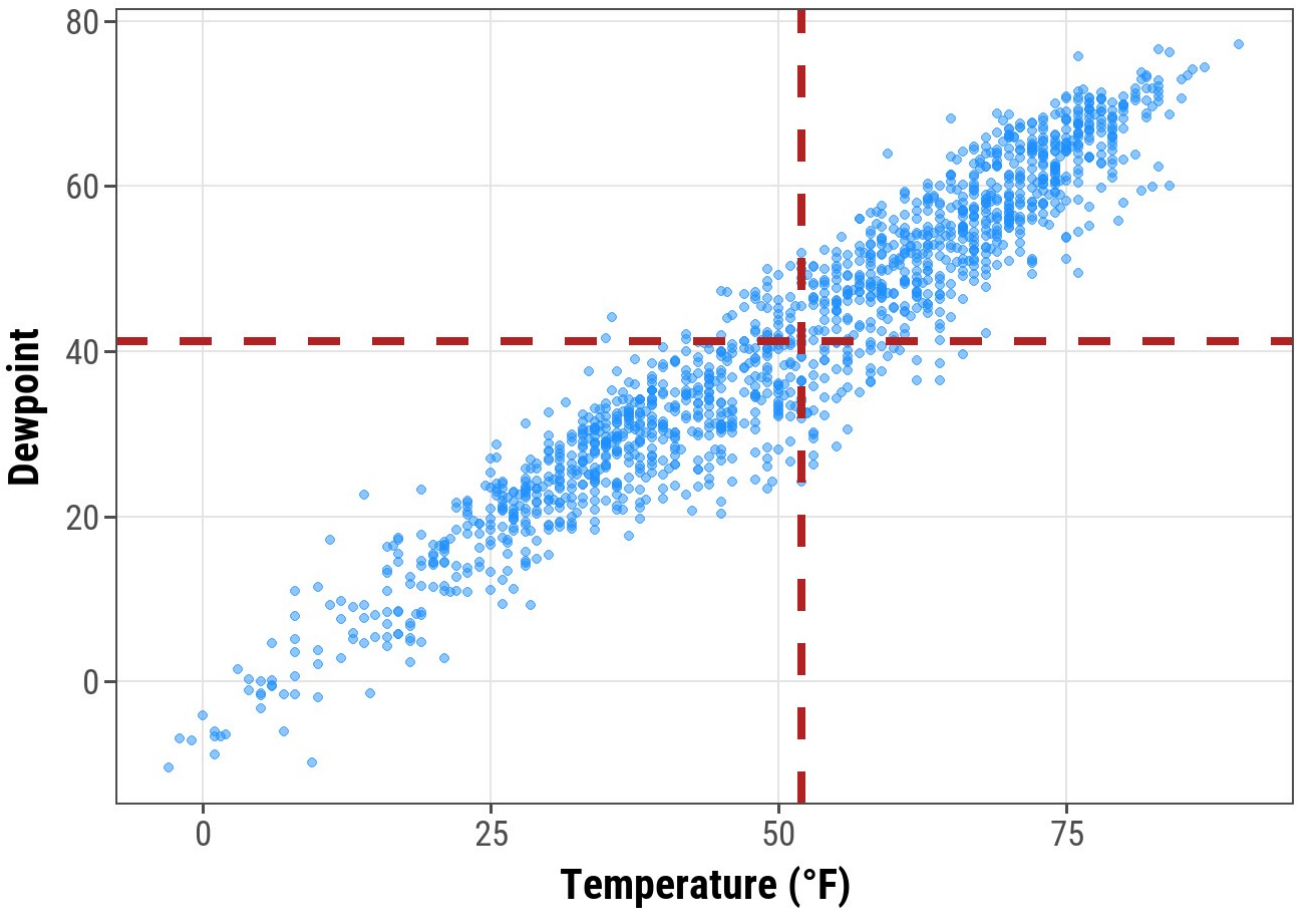
You might want to highlight a given range or threshold, which can be done plotting a line at defined coordinates using `geom_hline()` (for “horizontal lines”) or `geom_vline()` (for “vertical lines”):

```
ggplot(chic, aes(x = date, y = temp, color = o3)) +
  geom_point() +
  geom_hline(yintercept = c(0, 73)) +
  labs(x = "Year", y = "Temperature (°F)")
```



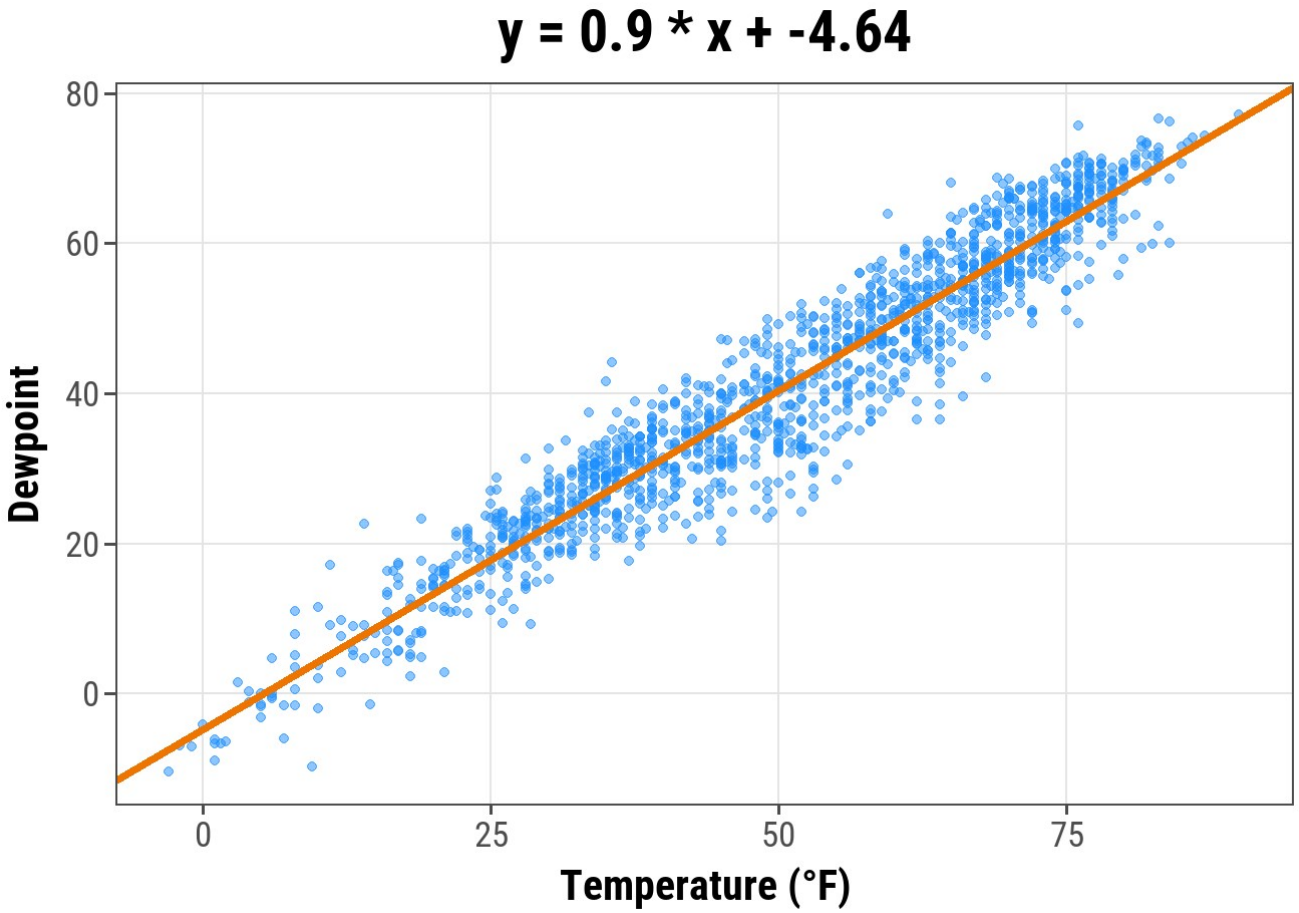



```
g <- ggplot(chic, aes(x = temp, y = dewpoint)) +
  geom_point(color = "dodgerblue", alpha = .5) +
  labs(x = "Temperature (°F)", y = "Dewpoint")

g +
  geom_vline(aes(xintercept = median(temp)), size = 1.5,
             color = "firebrick", linetype = "dashed") +
  geom_hline(aes(yintercept = median(dewpoint)), size = 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", size = 1.5) +
  labs(title = paste0("y = ", round(coefficients(reg)[2], 2),
                      " * x + ", round(coefficients(reg)[1], 2)))
```

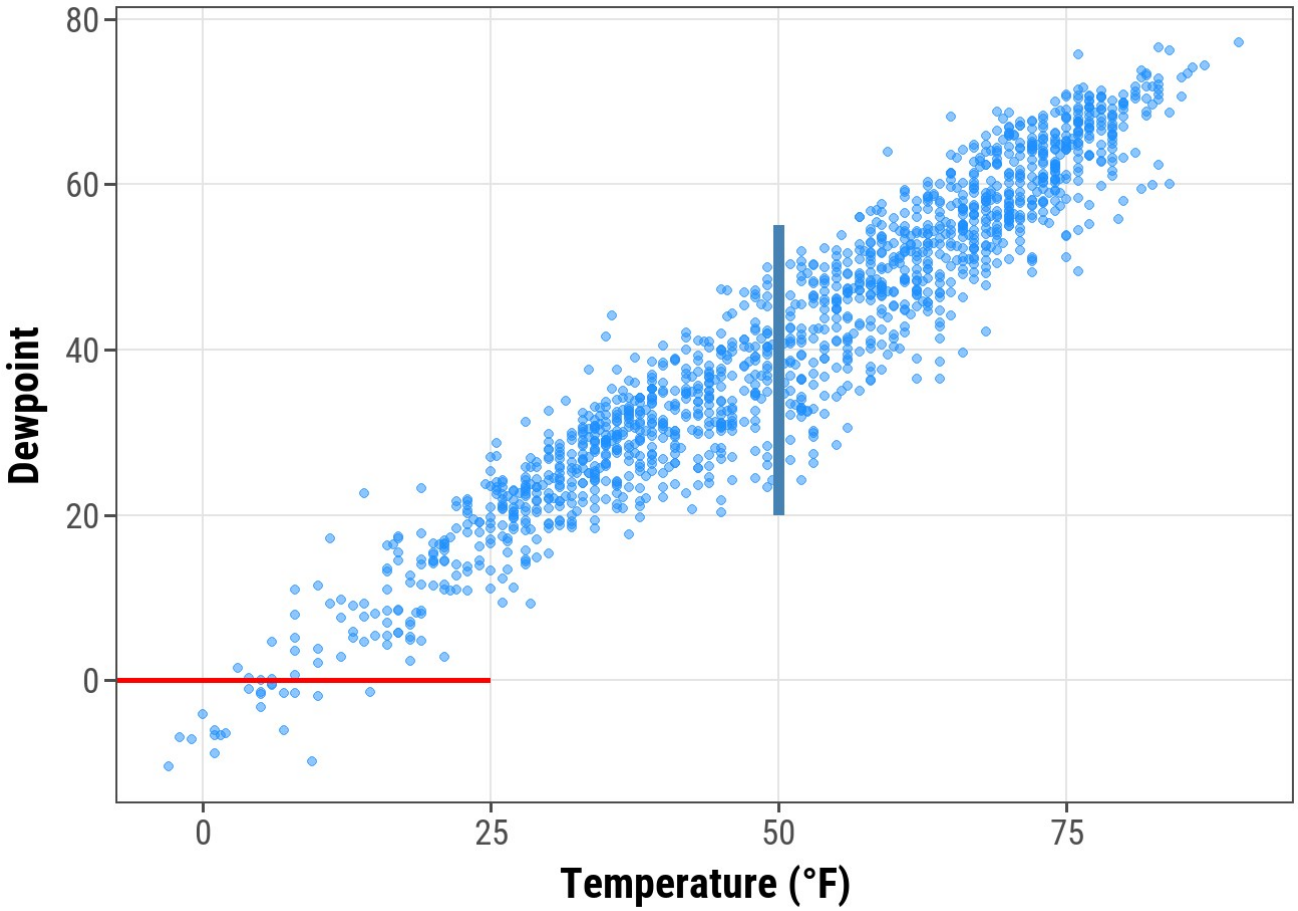


Later, we will learn how to add a linear fit with one command using `stat_smooth(method = "lm")`. However, there might be other reasons to add a line with a given slope and this is how one does it 🤷

ADD A LINE WITHIN A PLOT

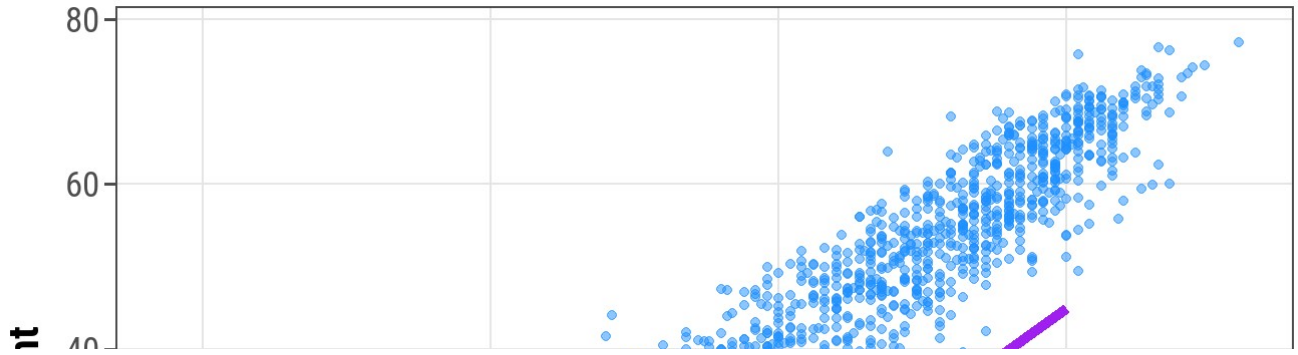
The previous approaches always covered the whole range of the plot panel, but sometimes one wants to highlight only a given area or use lines for annotations. In this case, `geom_linerange()` is here to help:

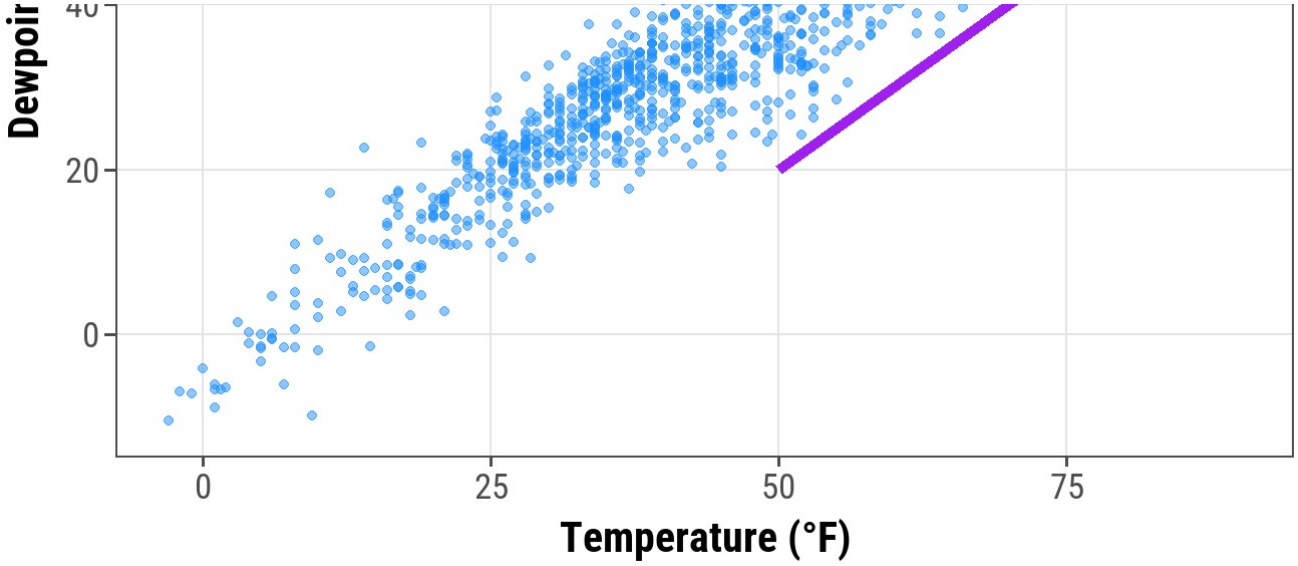
```
g +
  ## vertical line
  geom_linerange(aes(x = 50, ymin = 20, ymax = 55),
                 color = "steelblue", size = 2) +
  ## horizontal line
  geom_linerange(aes(xmin = -Inf, xmax = 25, y = 0),
                 color = "red", size = 1)
```



Or you can use `geom_segment()` to draw lines with a slope differing from 0 and 1:

```
g +
  geom_segment(aes(x = 50, xend = 75,
                   y = 20, yend = 45),
               color = "purple", size = 2)
```

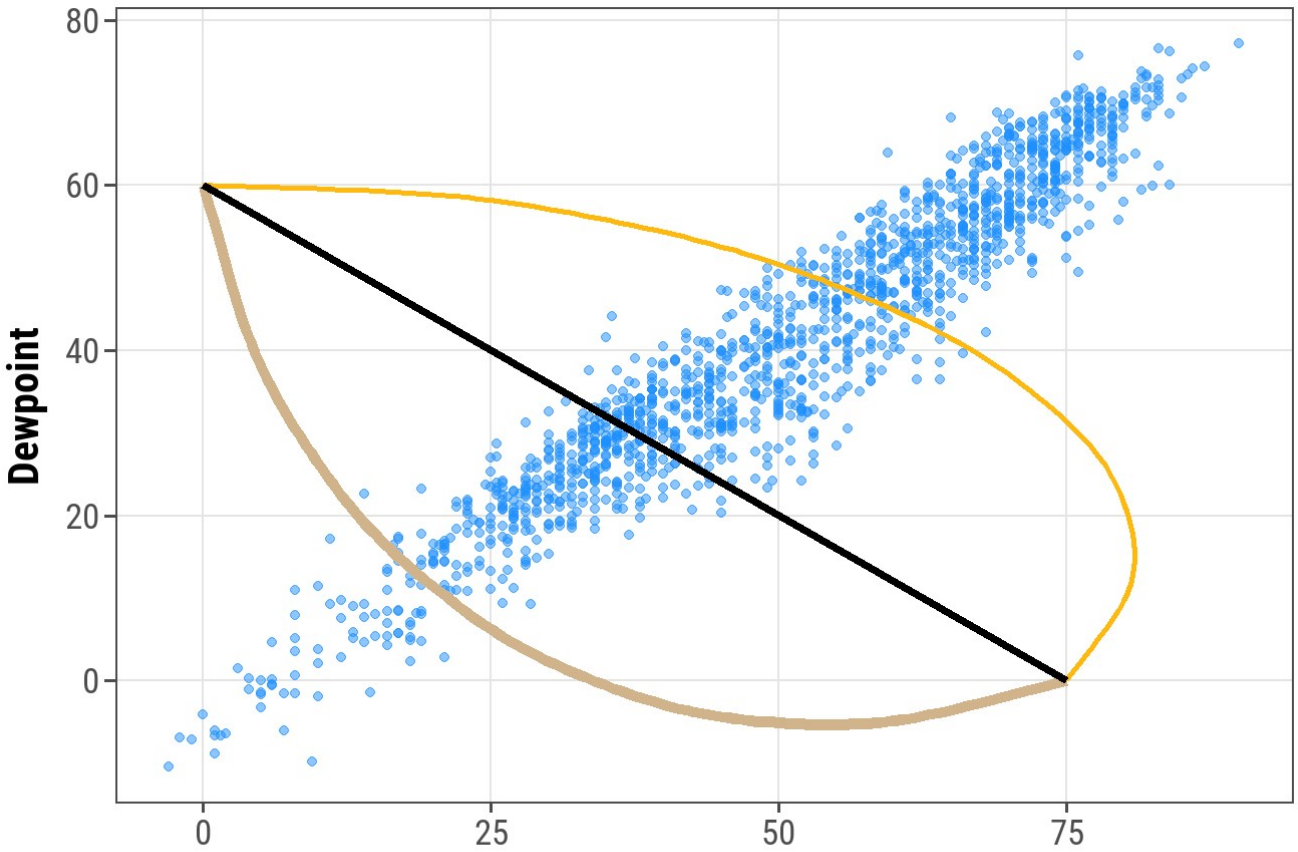




ADD CURVED LINES AND ARROWS TO A PLOT

`geom_curve()` adds curves. Well, and straight lines if you like:

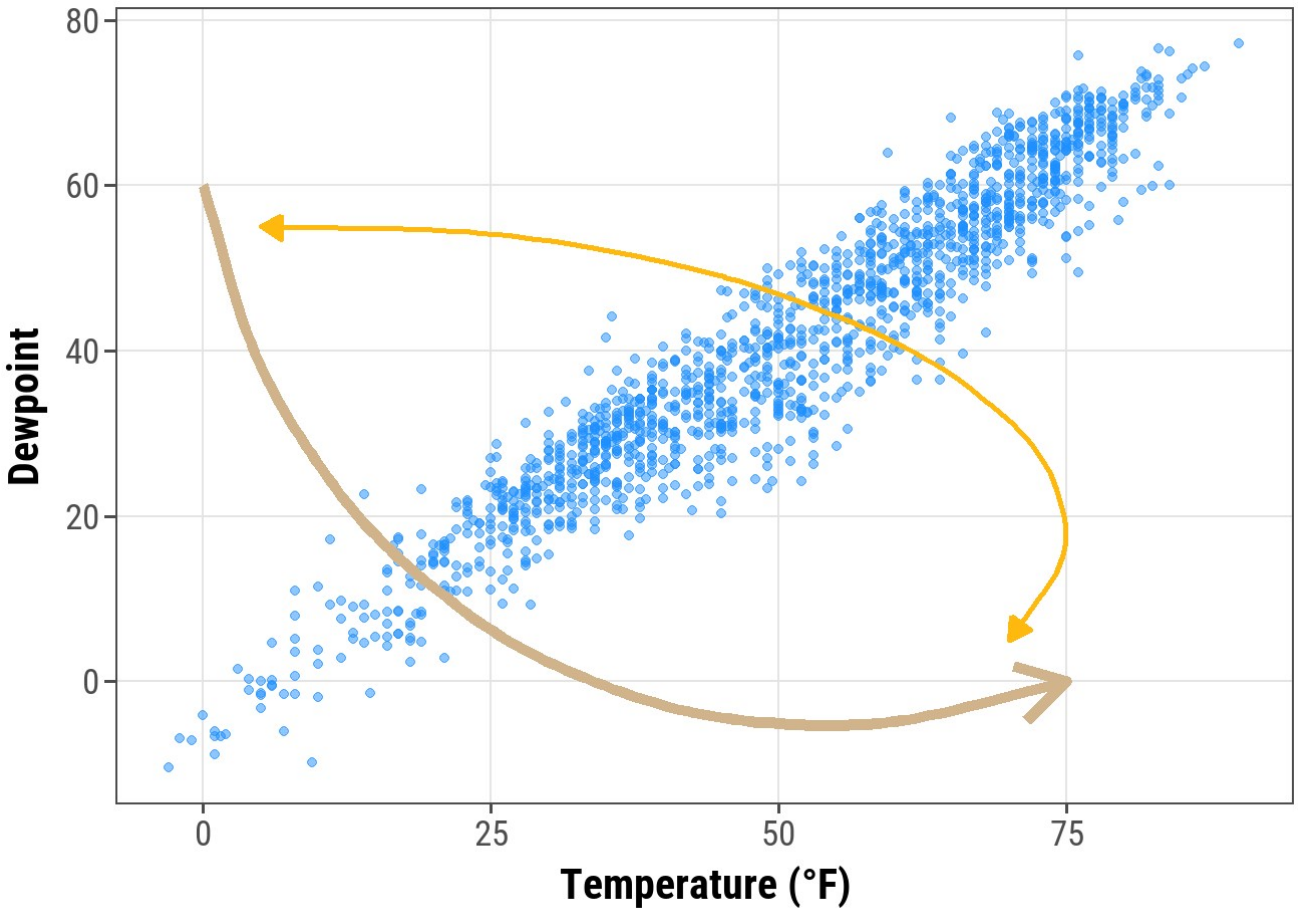
```
g +
  geom_curve(aes(x = 0, y = 60, xend = 75, yend = 0),
             size = 2, color = "tan") +
  geom_curve(aes(x = 0, y = 60, xend = 75, yend = 0),
             curvature = -0.7, angle = 45,
             color = "darkgoldenrod1", size = 1) +
  geom_curve(aes(x = 0, y = 60, xend = 75, yend = 0),
             curvature = 0, size = 1.5)
```



Temperature (°F)

The same geom can be used to draw arrows:

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



↑ Jump back to Table of Content.

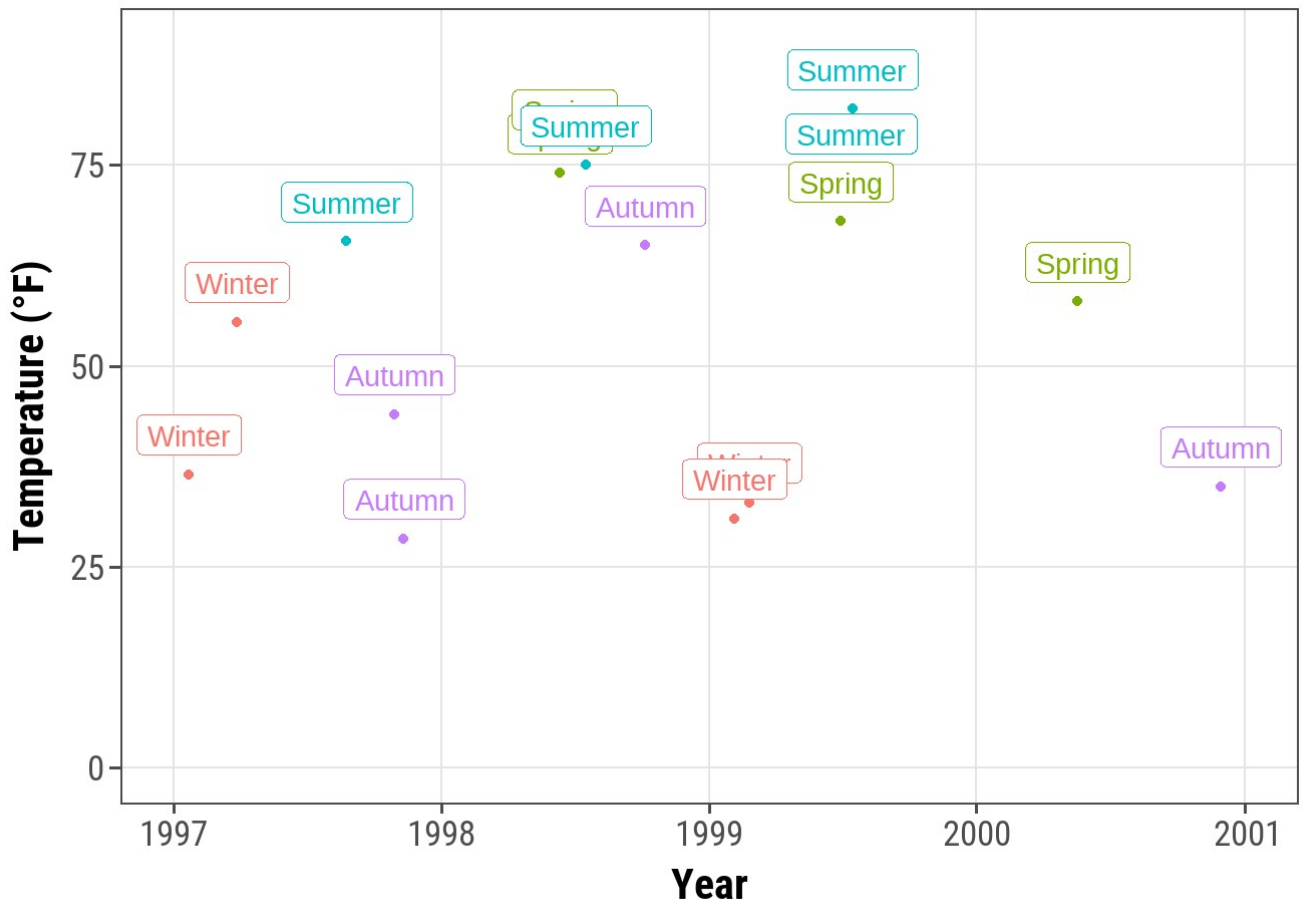
WORKING WITH TEXT

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)

library(dplyr)
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")
```

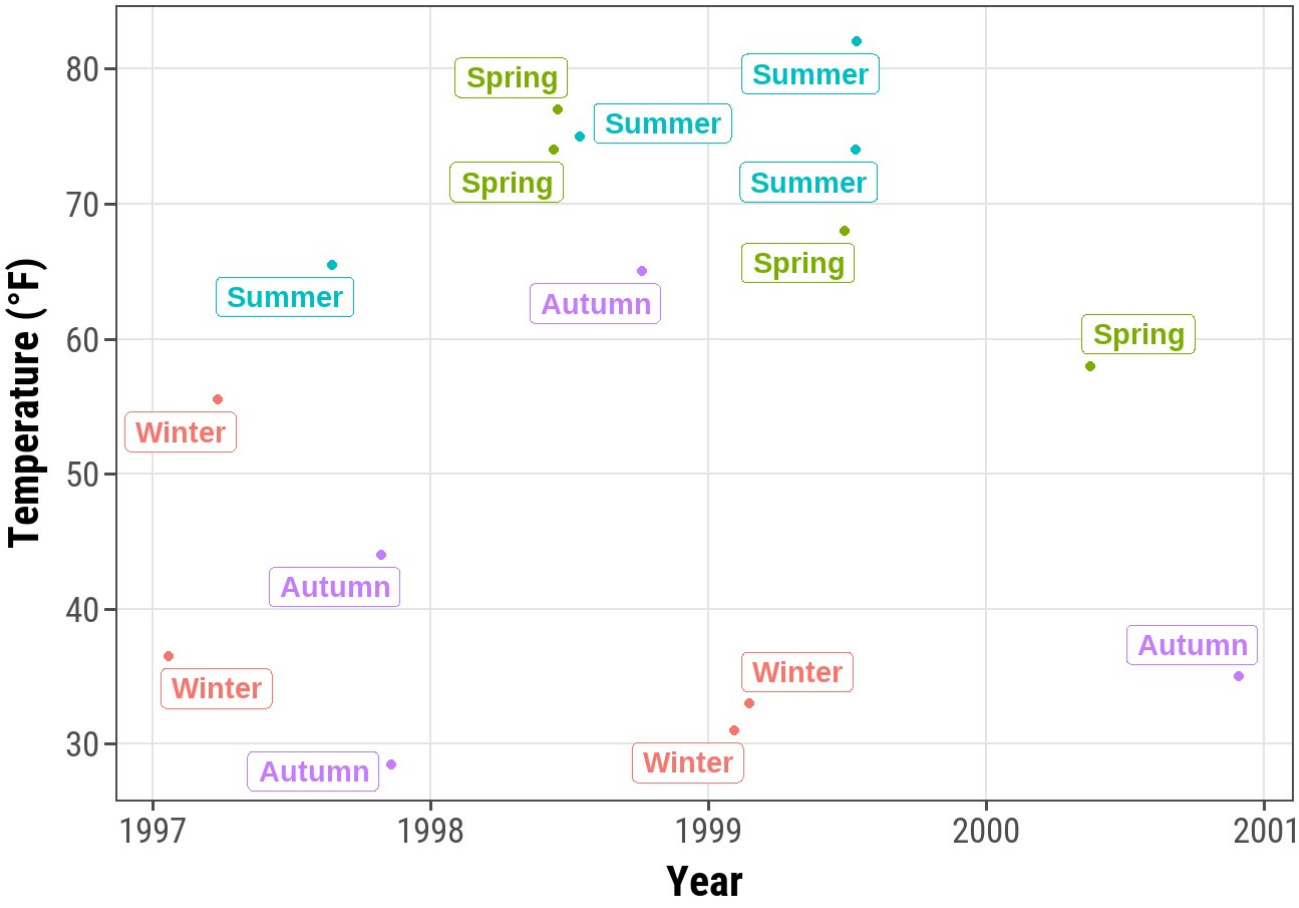


Okay, avoiding overlap of labels did not work out. But don't worry, we are going to fix it in a minute!

🤷 *You can also use `geom_text()` if you don't like boxes around your labels. Expand to see example.*

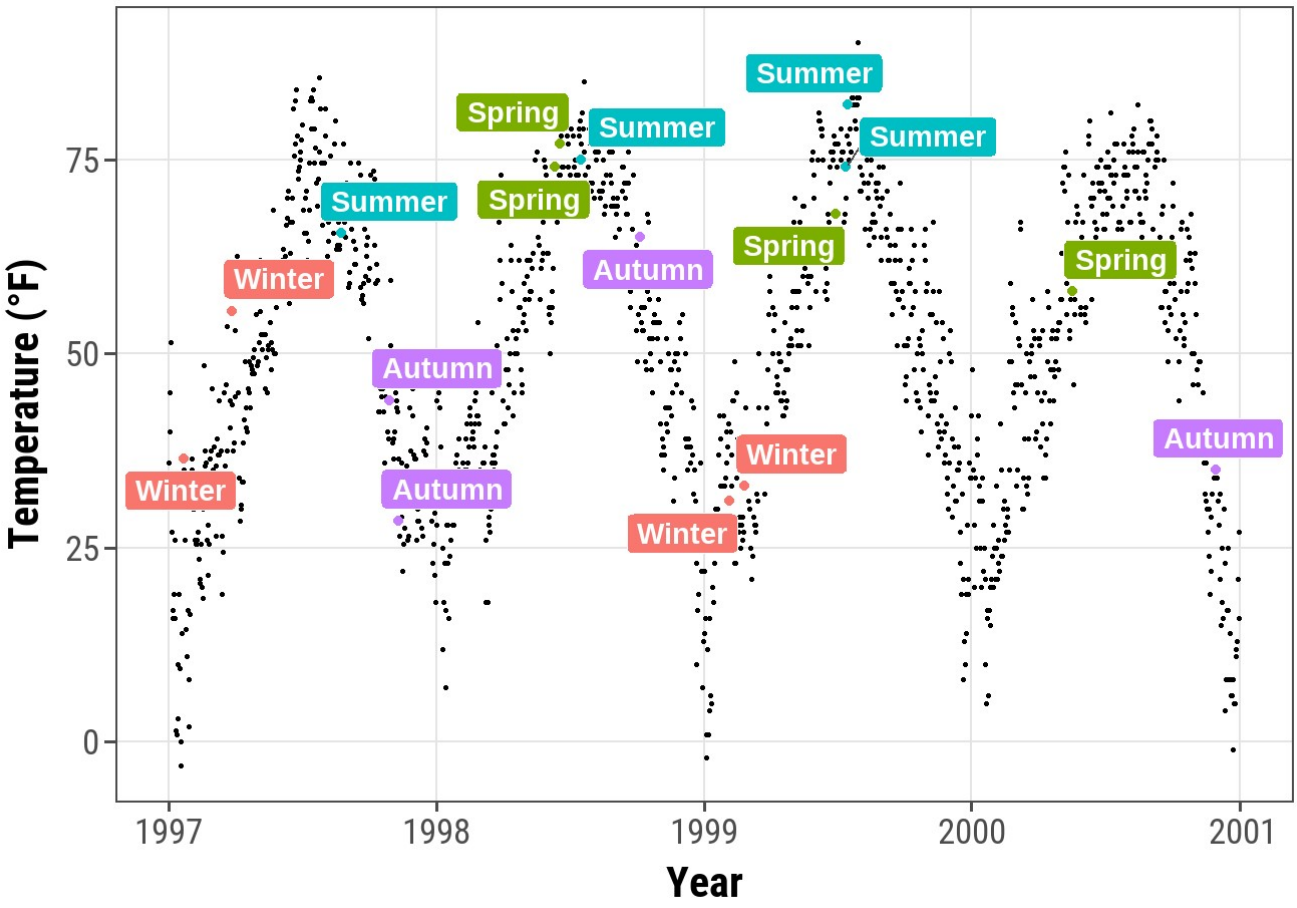
A cool thing is the `{ggrepel}` package which provides geoms for `{ggplot2}` to repel overlapping text as in our examples above. We simply replace `geom_text()` by `geom_text_repel()` and `geom_label()` by `geom_label_repel()`:

```
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")
```

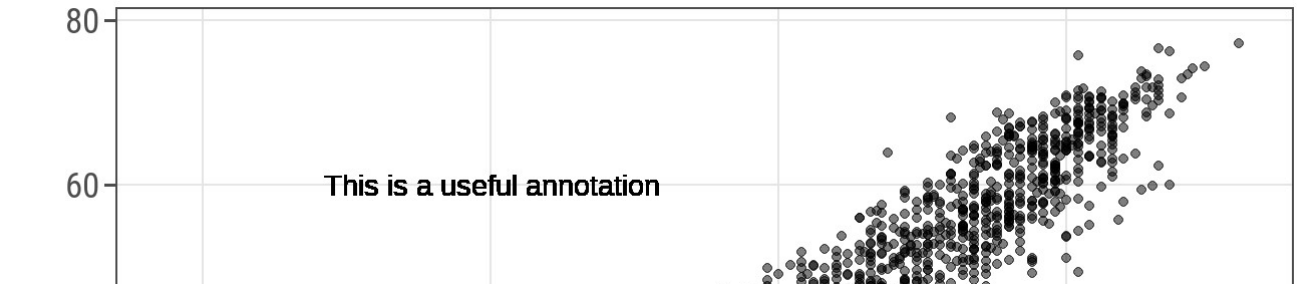


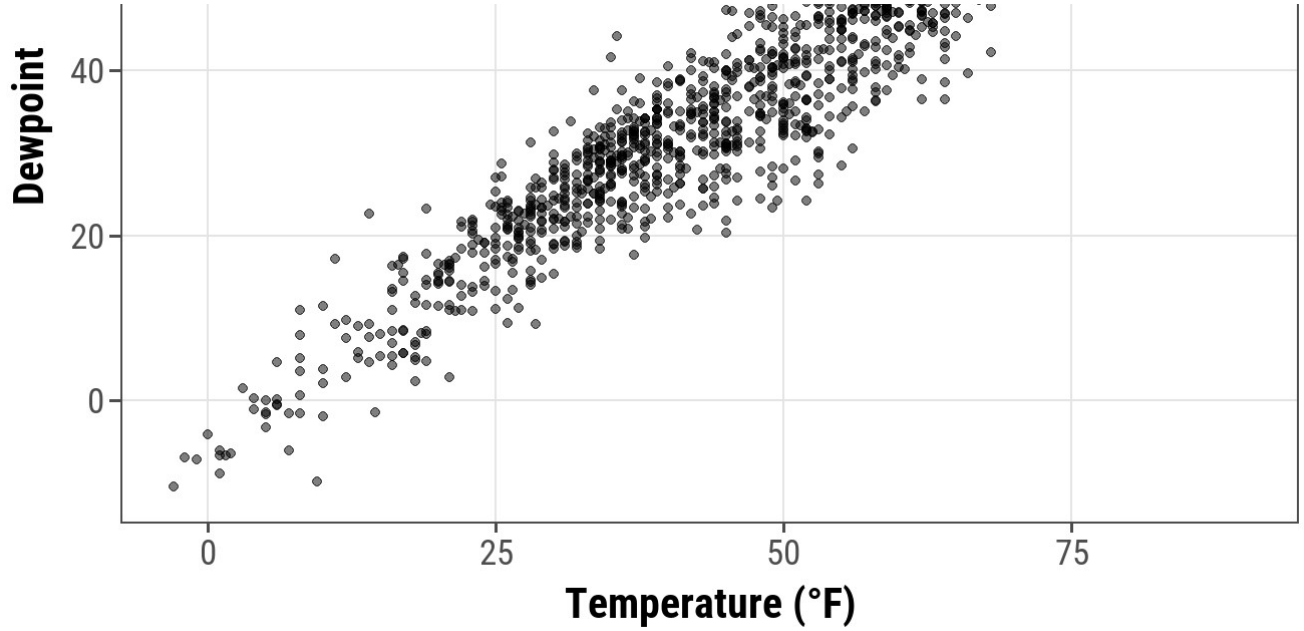
This also works for the pure text labels by using `geom_text_repel()`. Have a look at all the usage examples (https://cran.r-project.org/web/packages/ggrepel/vignettes/ggrepel.html).

ADD TEXT ANNOTATIONS

There are several ways how one can add annotations to a ggplot. We can again use `geom_text()` or `geom_label()`:

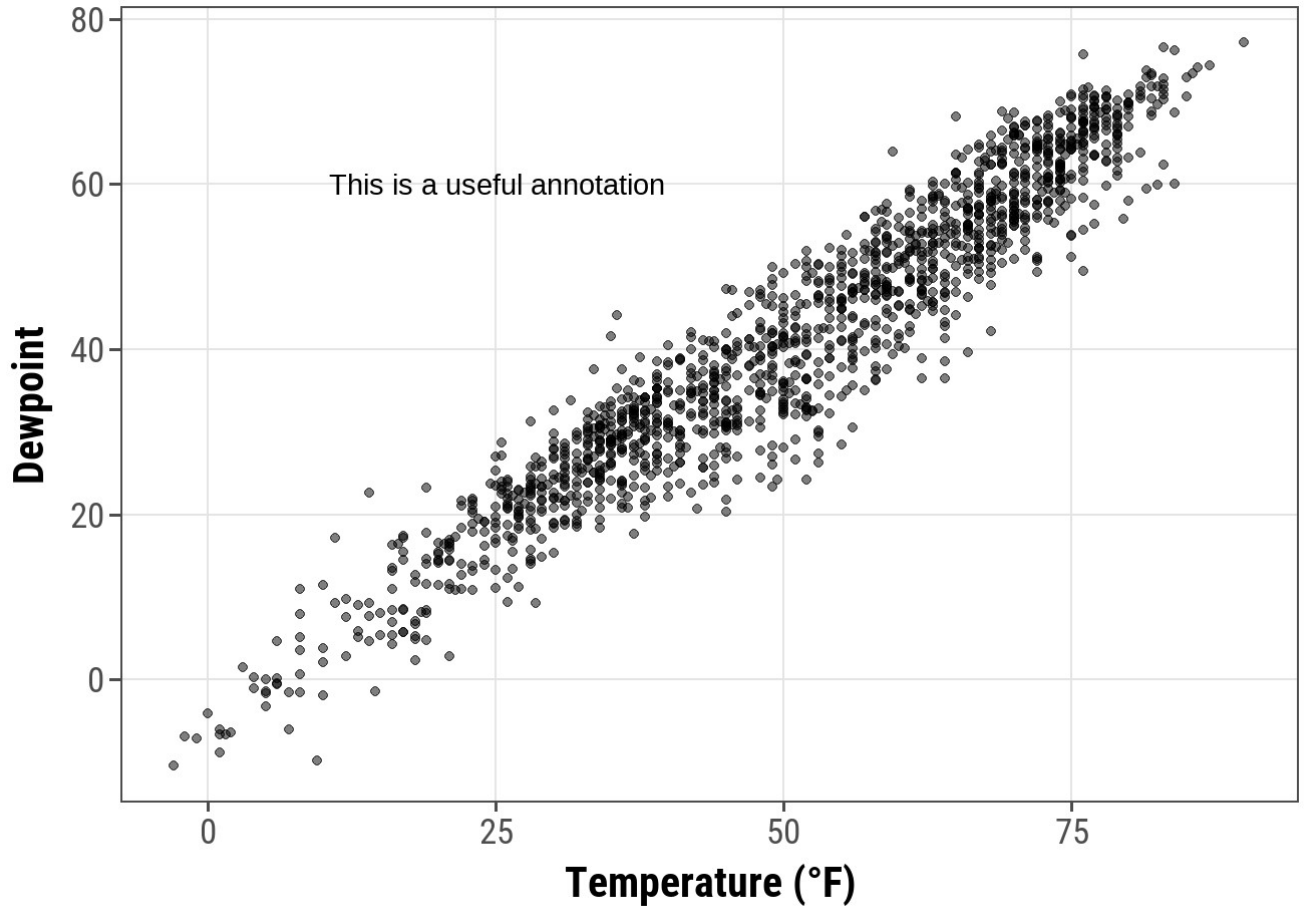
```
g <-
  ggplot(chic, aes(x = temp, y = dewpoint)) +
  geom_point(alpha = .5) +
  labs(x = "Temperature (°F)", y = "Dewpoint")

g +
  geom_text(aes(x = 25, y = 60,
                label = "This is a useful annotation"))
```





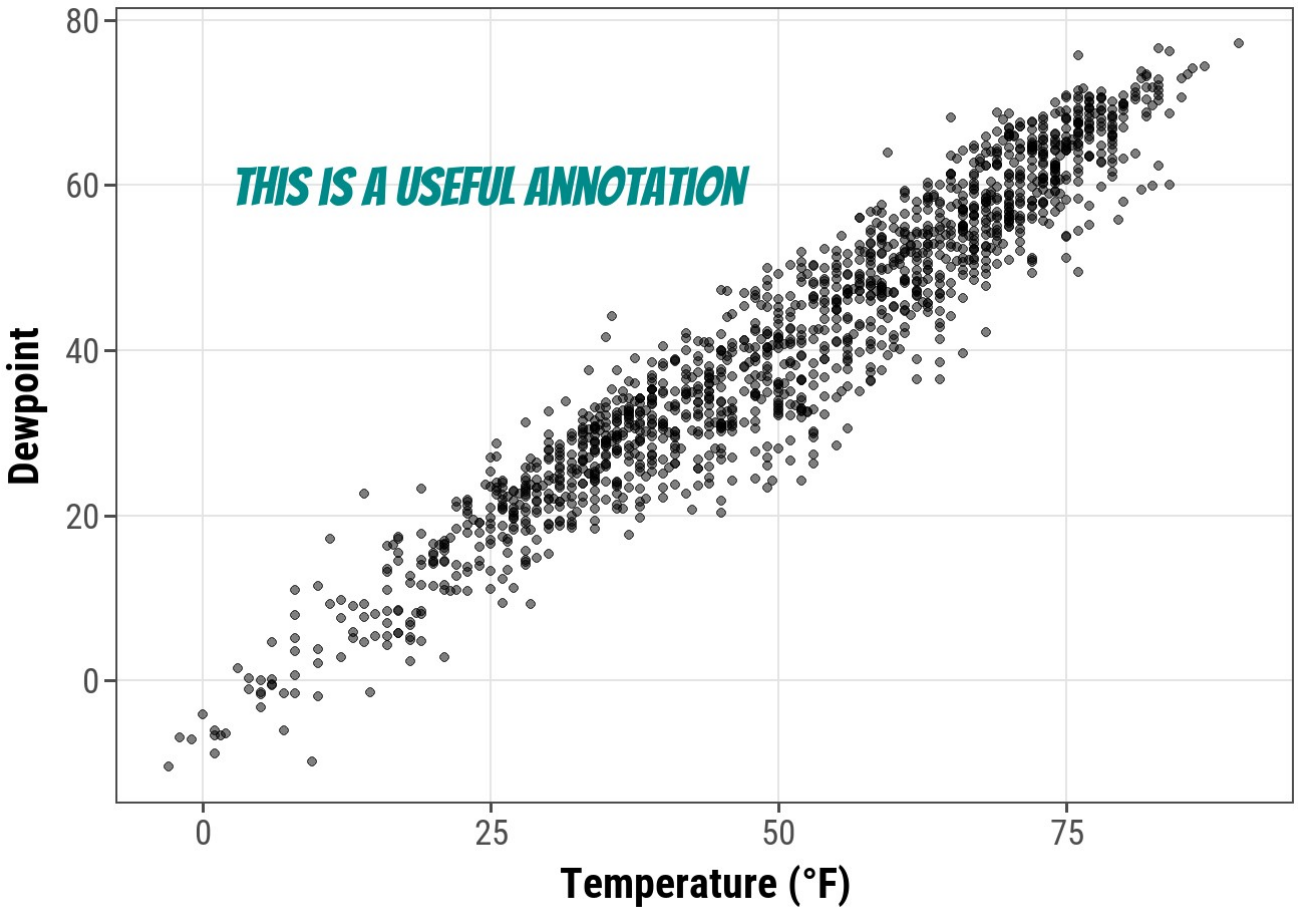
However, now ggplot has drawn one text label per data point—that's 1,461 labels and you only see one! You can solve that by setting the `stat` argument to `"unique"`:

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



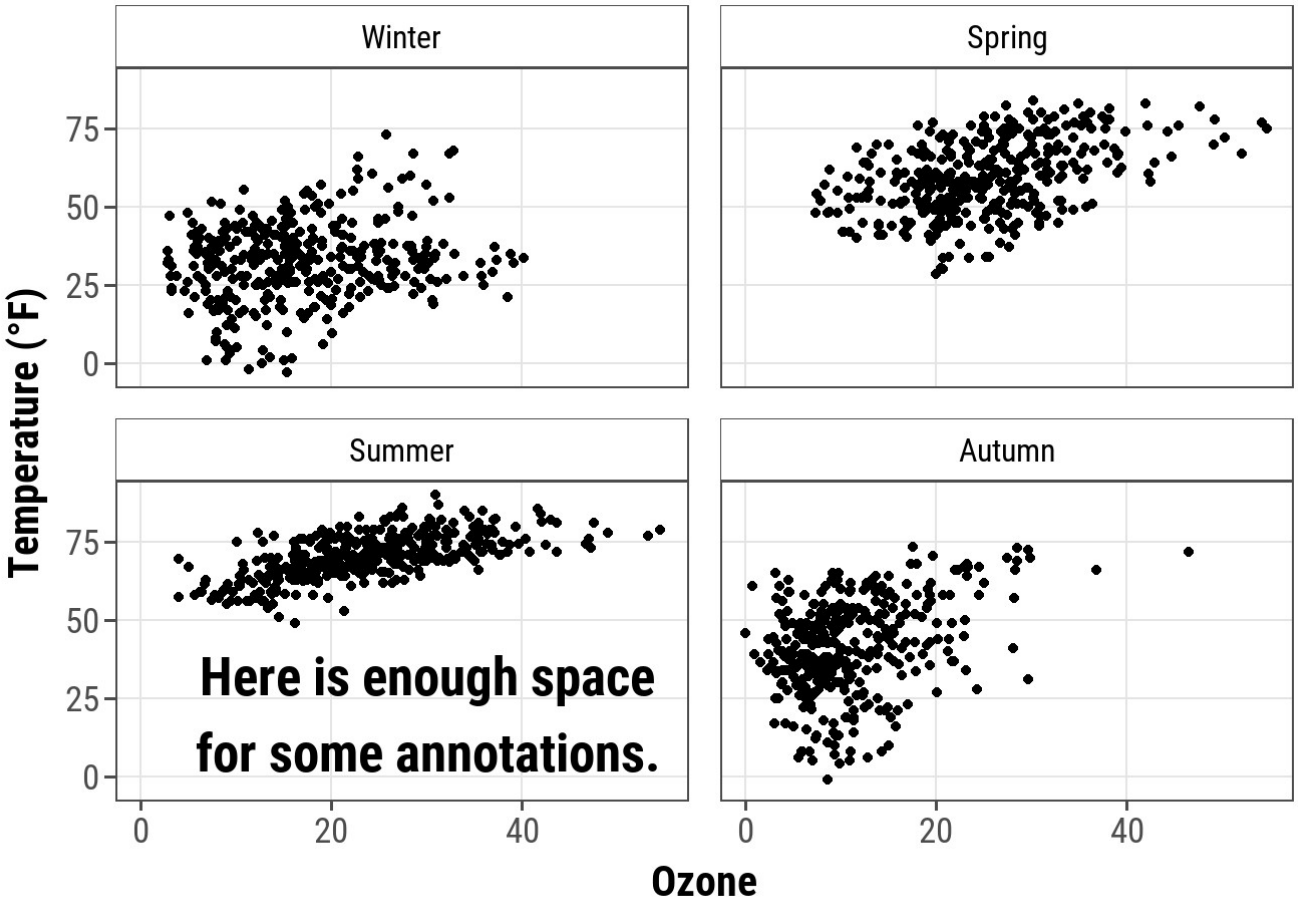
By the way, of course one can change the properties of the displayed text:

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



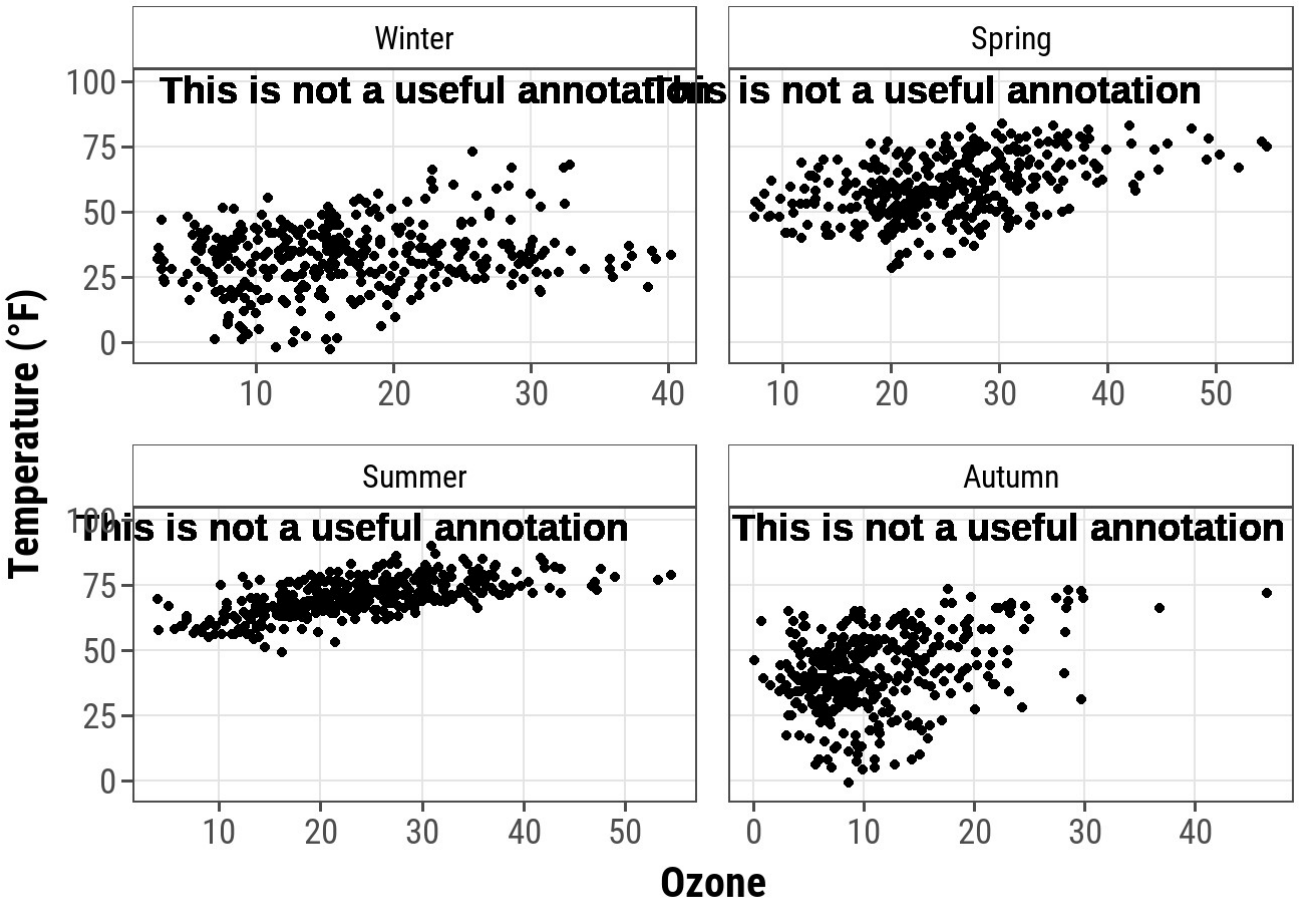
In case you use one of the facet functions to visualize your data you might run into trouble. One thing is that you may want to include the annotation only once:

```
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")
```



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

```
library(tidyverse)
(ann <-
  chic %>%
  group_by(season) %>%
  summarize(o3 = min(o3, na.rm = TRUE) +
              (max(o3, na.rm = TRUE) - min(o3, na.rm = TRUE)) / 2))
```

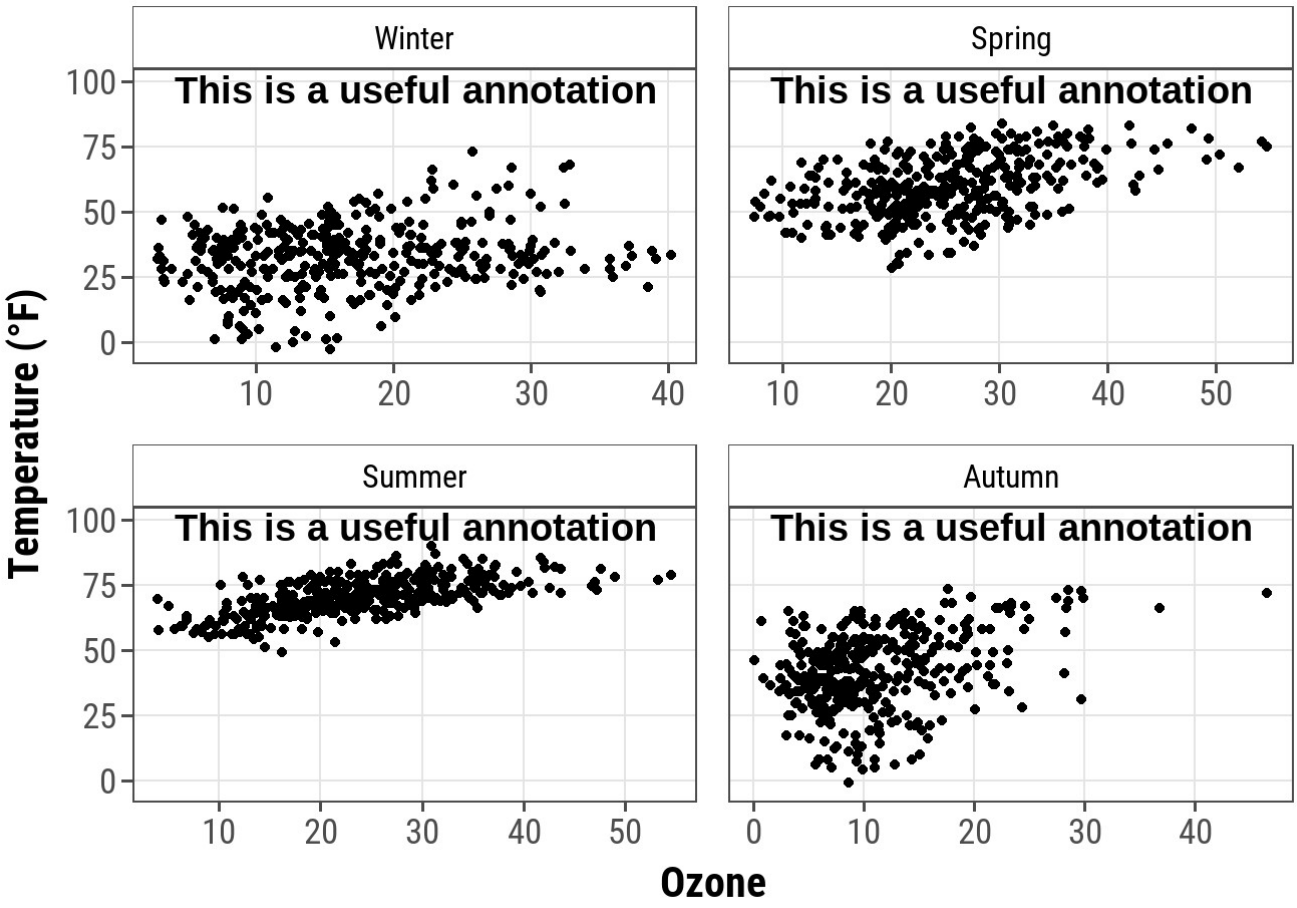
```
## # A tibble: 4 x 2
##   season    o3
## * <fct>  <dbl>
## 1 Winter  21.5
## 2 Spring  31.0
## 3 Summer  29.2
## 4 Autumn  23.3
```

```
ann
```

```
## # A tibble: 4 x 2
##   season    o3
## * <fct>  <dbl>
## 1 Winter  21.5
## 2 Spring  31.0
## 3 Summer  29.2
## 4 Autumn  23.3
```

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

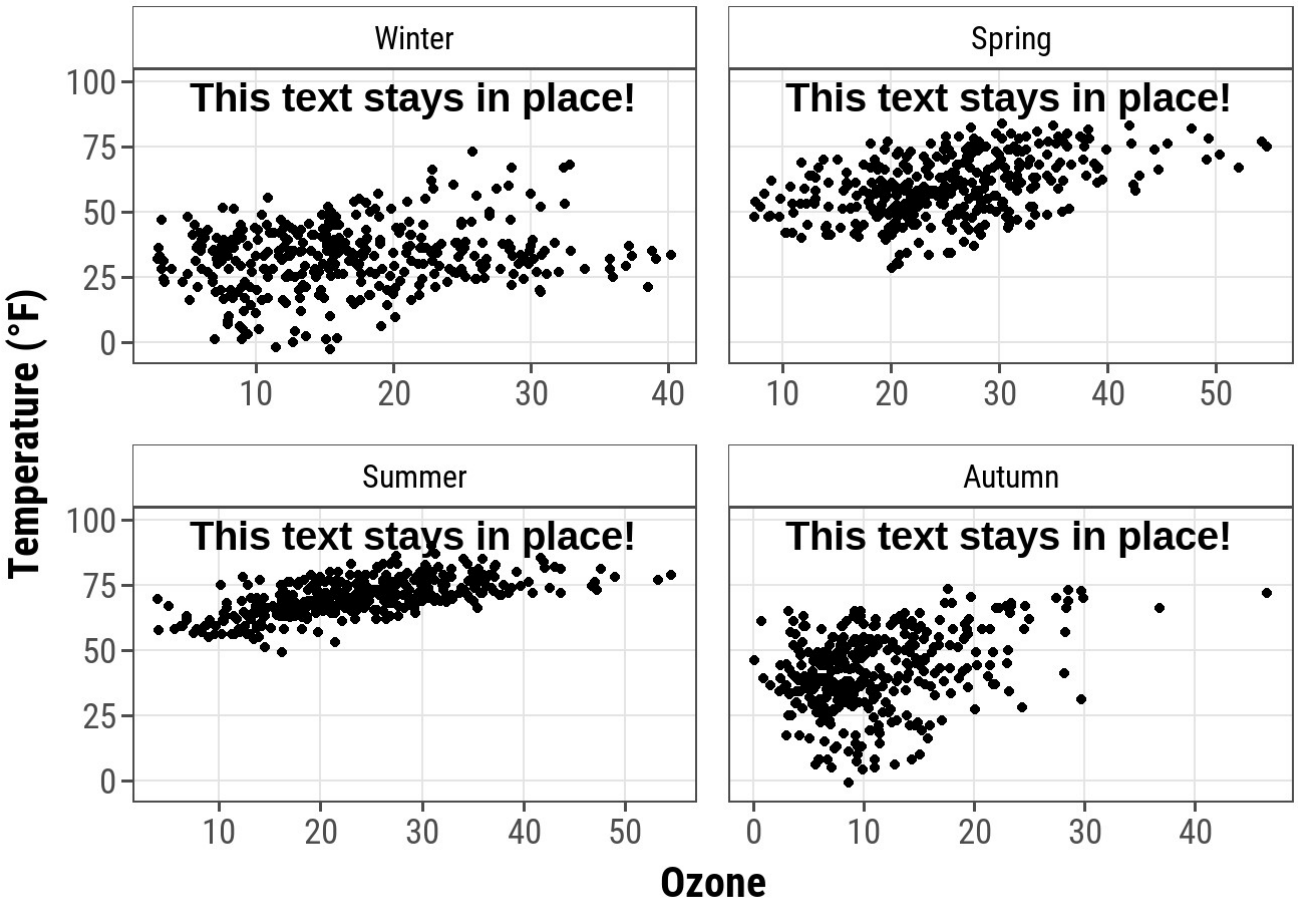
```
g +
  geom_text(data = ann,
            aes(x = o3, y = 97,
                label = "This is a useful annotation"),
            size = 5, fontface = "bold") +
  scale_y_continuous(limits = c(NA, 100)) +
  facet_wrap(~season, scales = "free_x")
```



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.

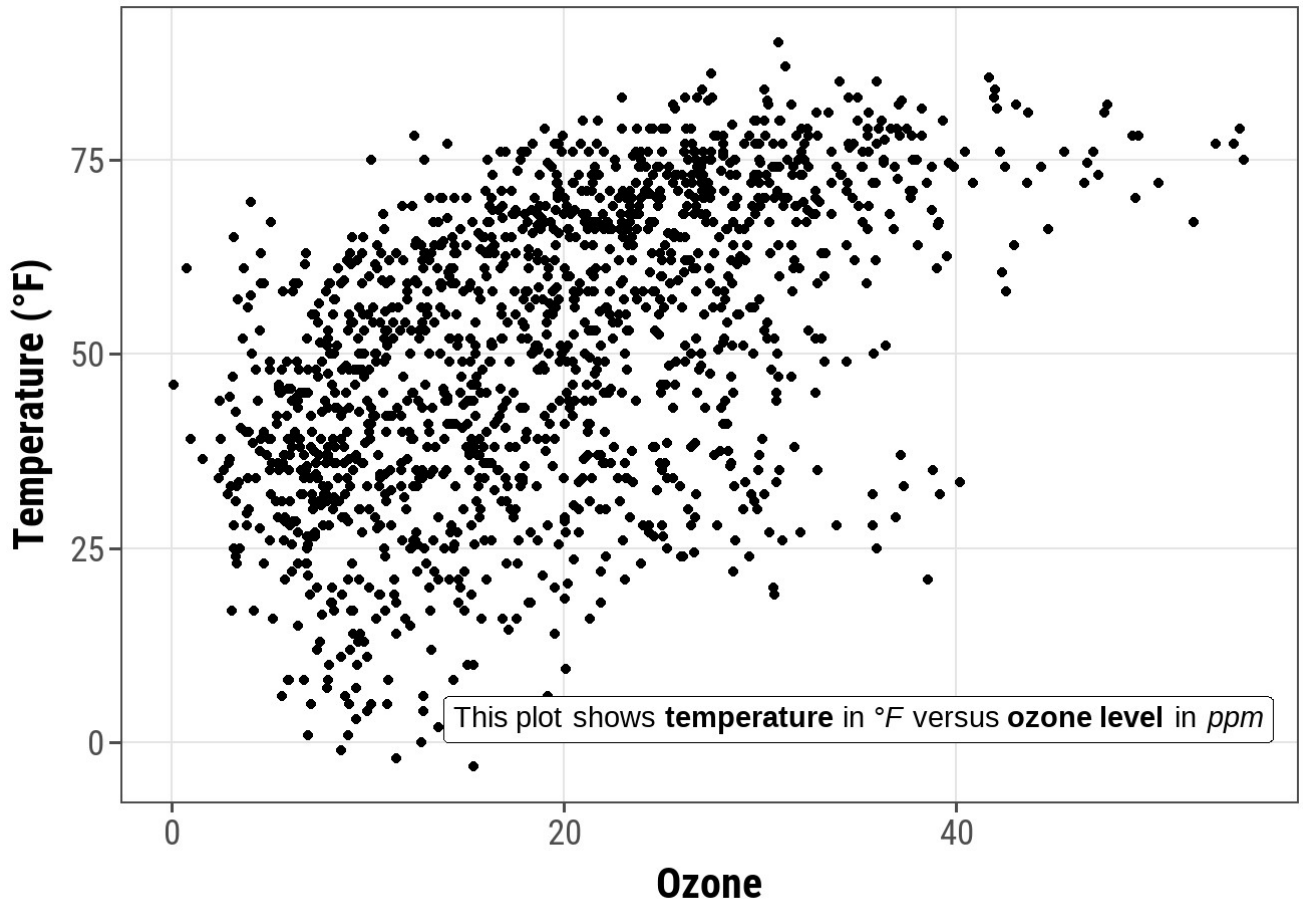
```
library(grid)
my_grob <- grobTree(textGrob("This text stays in place!",
                             x = .1, y = .9, hjust = 0,
                             gp = gpar(col = "black",
                                       fontsize = 15,
                                       fontface = "bold")))

g +
  annotation_custom(my_grob) +
  facet_wrap(~season, scales = "free_x") +
  scale_y_continuous(limits = c(NA, 100))
```



USE MARKDOWN AND HTML RENDERING FOR ANNOTATIONS

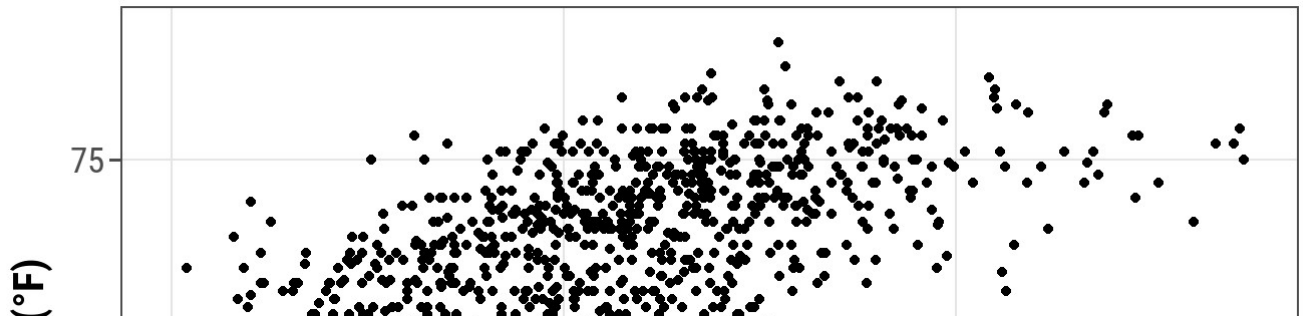
Again, we are using Claus Wilke's `{ggtext}` package (https://wilkelab.org/ggtext/) that is designed for improved text rendering support for `{ggplot2}`. The `{ggtext}` package defines two new theme elements, `element_markdown()` and `element_textbox()`. The package also provides additional geoms. `geom_richtext()` is a replacement for `geom_text()` and `geom_label()` and renders text as markdown…

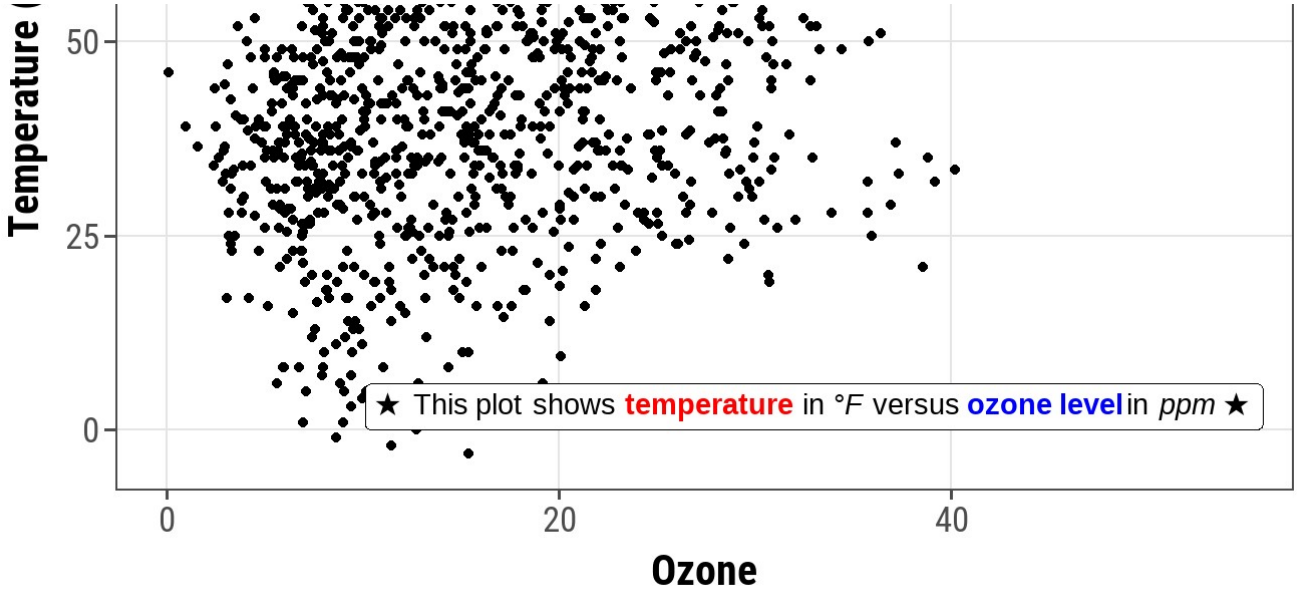
```
library(ggtext)

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")
```



... or html:

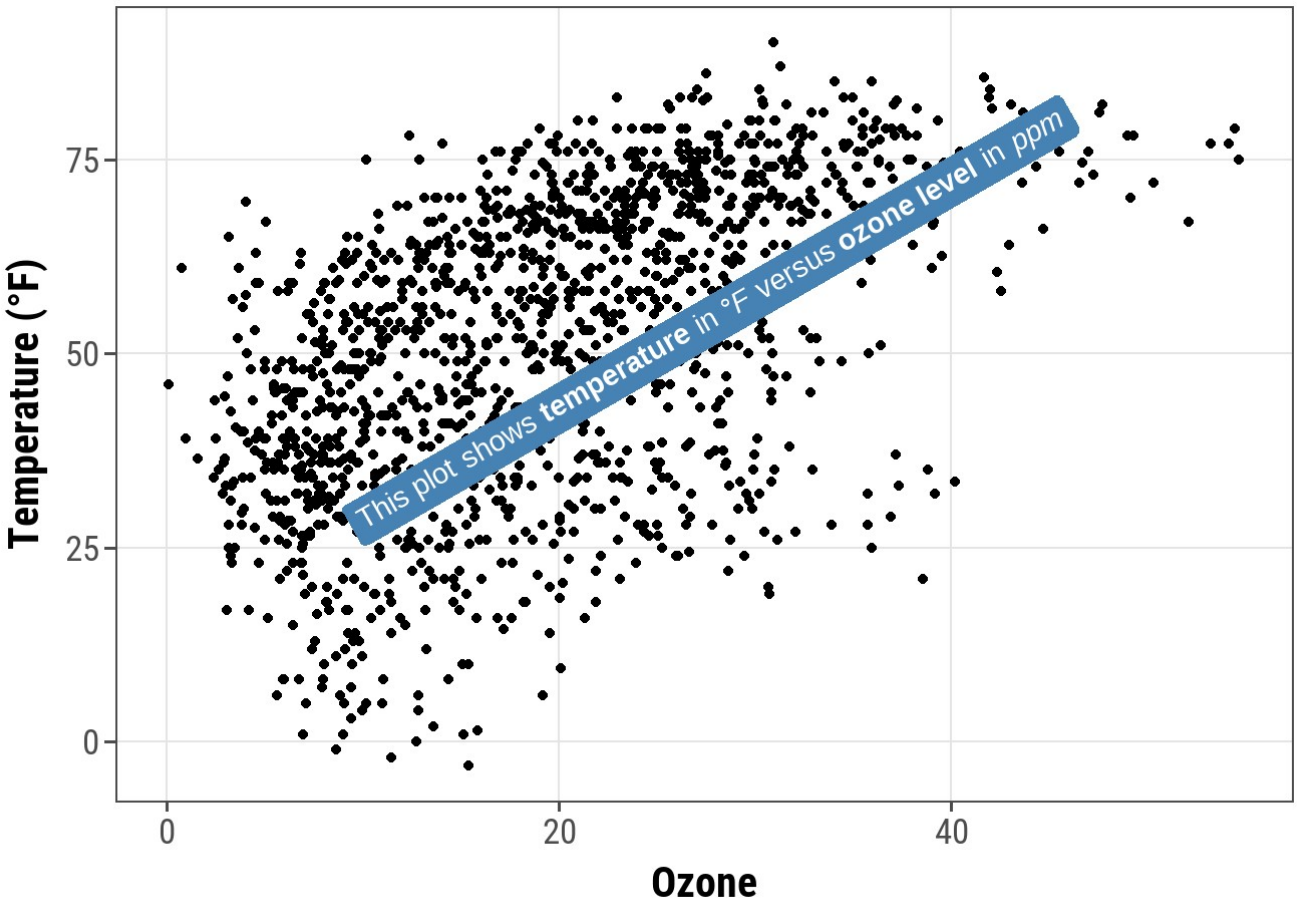
```
lab_html <- "&#9733; This plot shows <b style='color:red;'>temperature</b> in <i>°F</i> ver

g +
  geom_richtext(aes(x = 33, y = 3, label = lab_html),
                stat = "unique")
```





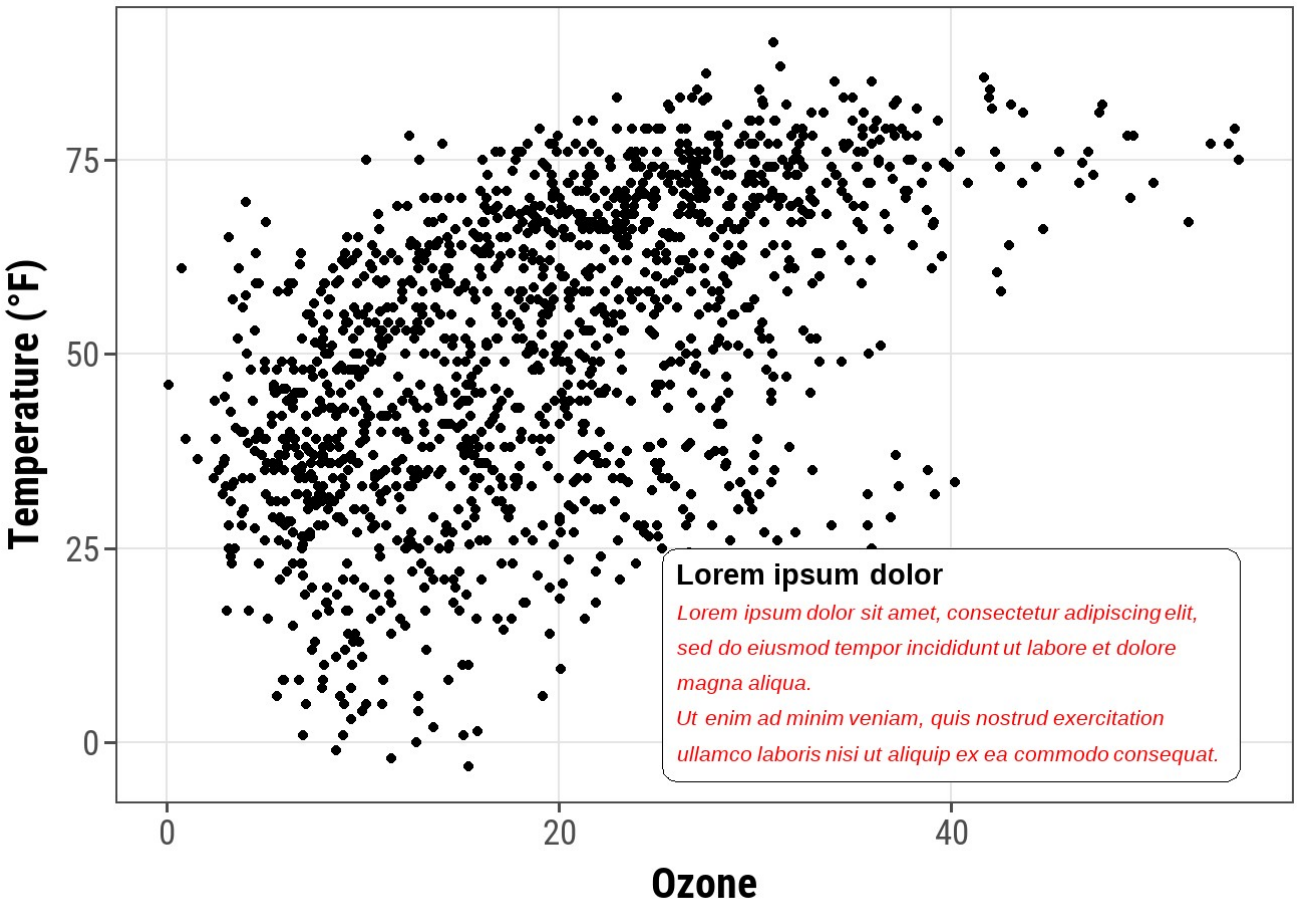
The geom comes with a lot of details one can modify, such as angle (which is not possible in the default `geom_text()` and `geom_label()`), properties of the box and properties of the text.

```
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")
```



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 dolo

g +
  geom_textbox(aes(x = 40, y = 10, label = lab_long),
               width = unit(15, "lines"), stat = "unique")
```



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

↑ Jump back to Table of Content.

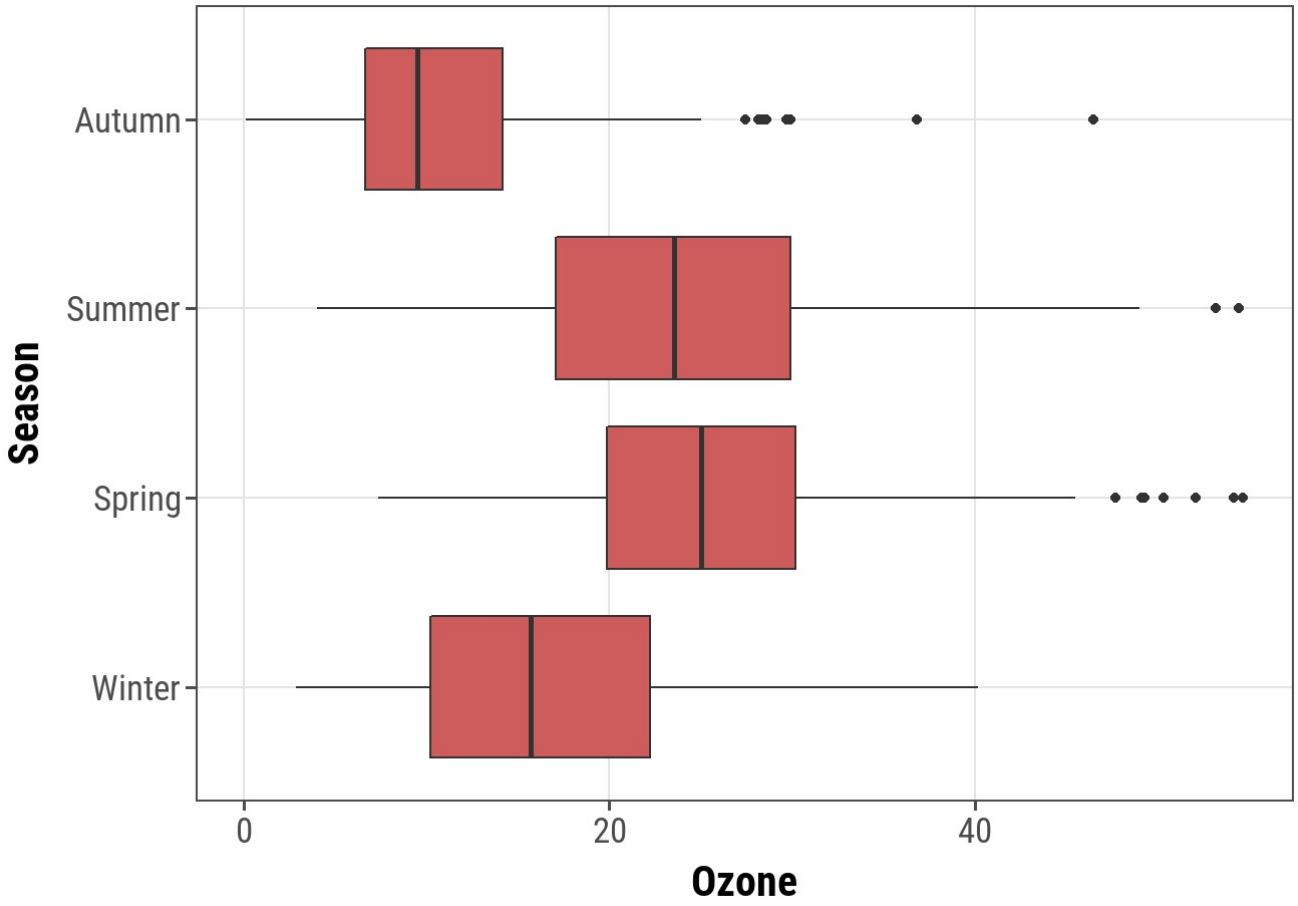
WORKING WITH COORDINATES

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()
```

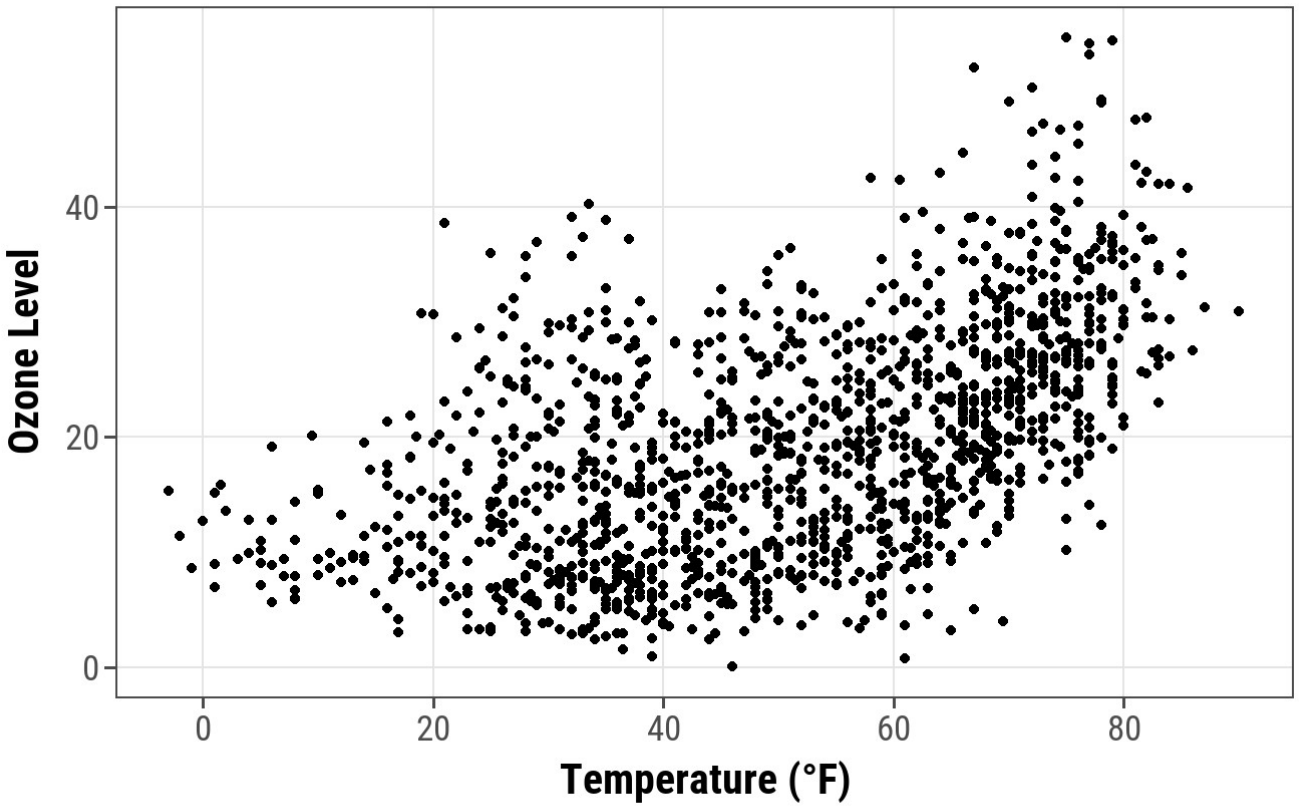


🤷 *Since `{ggplot2}` version 3.0.0 it is also possible to draw geom's horizontally via the argument `orientation = "y"`. Expand to see example.*

FIX AN AXIS

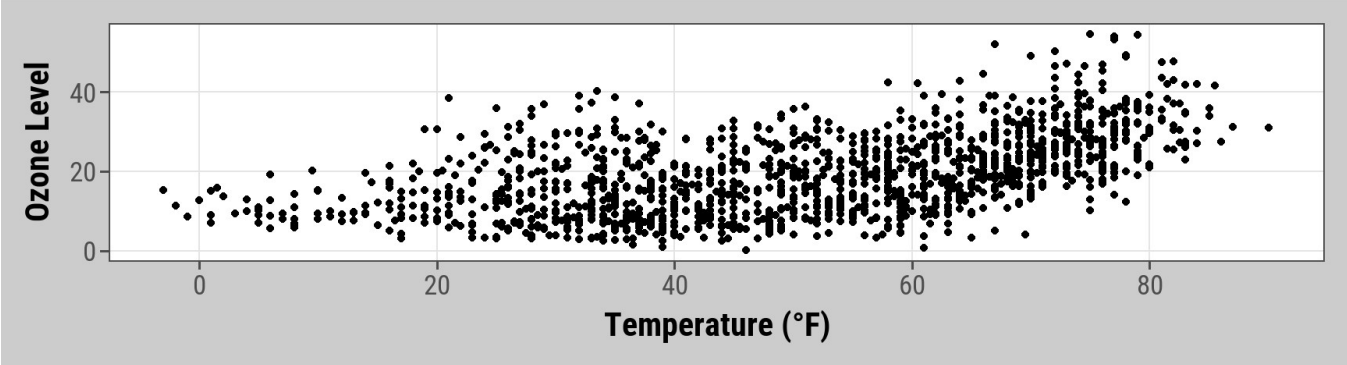
One can fix the aspect ratio of the Cartesian coordinate system and literally force a physical representation of the units along the x and y axes:

```
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:

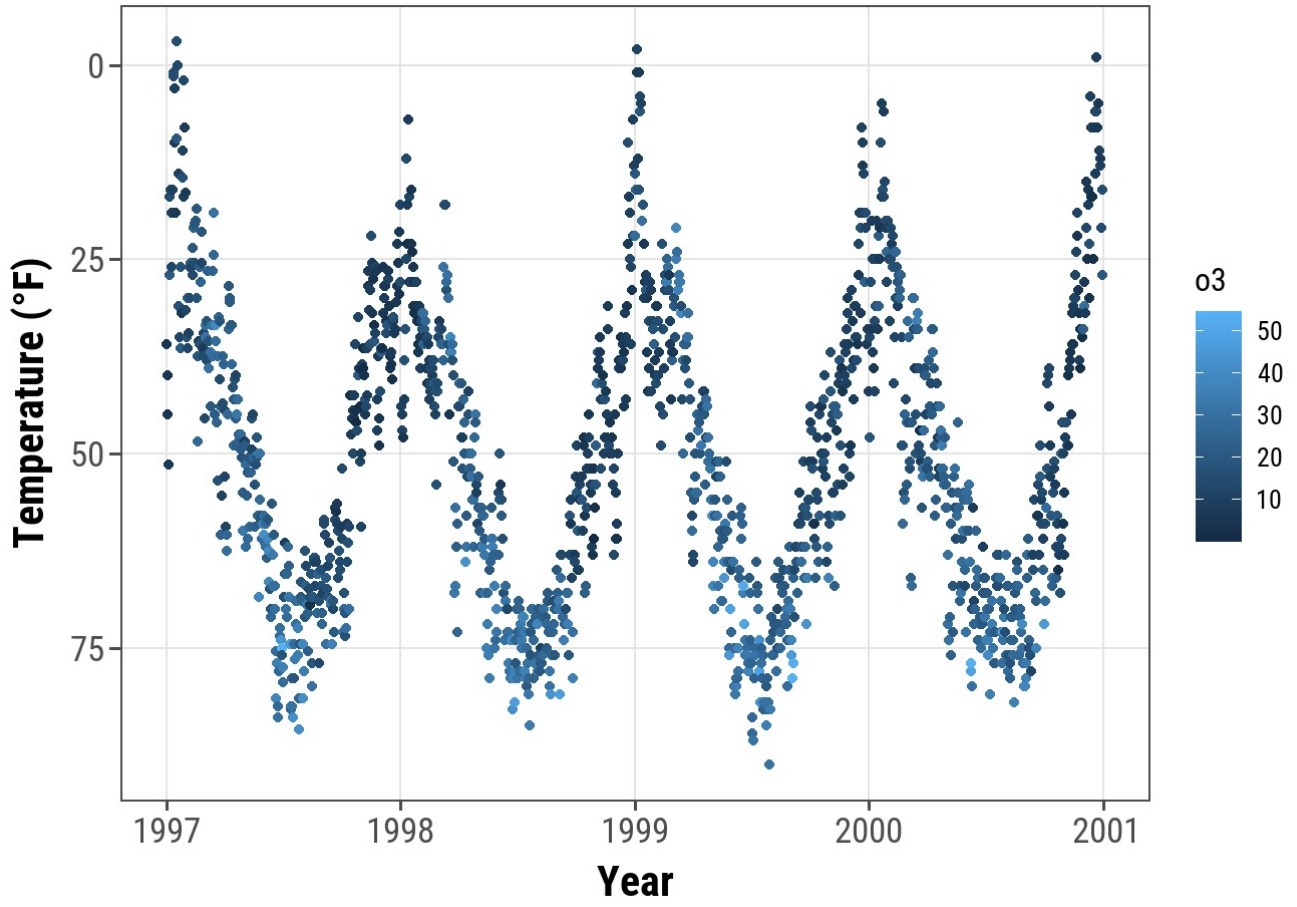
```
ggplot(chic, aes(x = temp, y = o3)) +
  geom_point() +
  labs(x = "Temperature (°F)", y = "Ozone Level") +
  scale_x_continuous(breaks = seq(0, 80, by = 20)) +
  coord_fixed(ratio = 1/3) +
  theme(plot.background = element_rect(fill = "grey80"))
```



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()
```

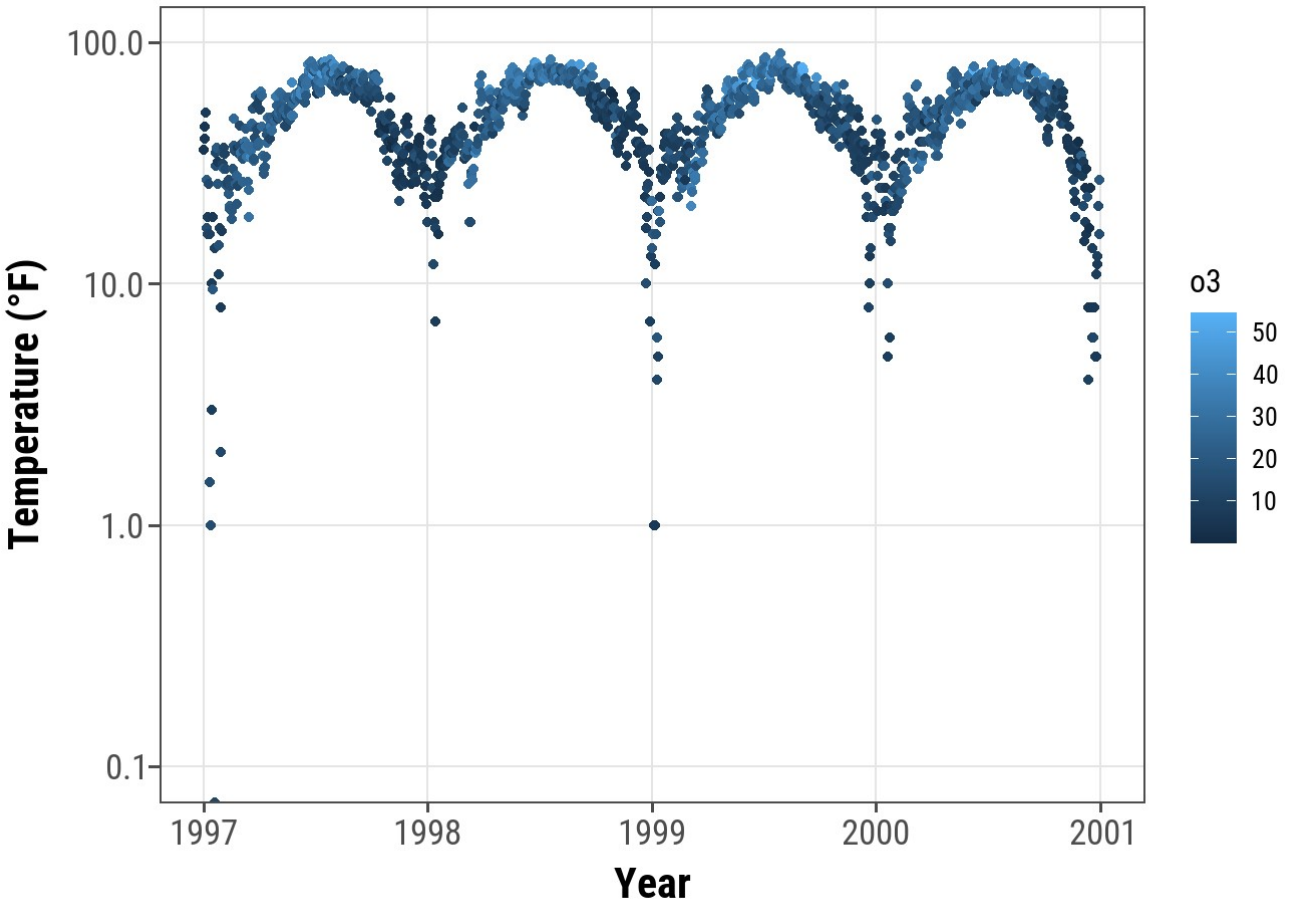


🤷 *Note that this will only work for continuous data. If you want to reverse discrete data, use the `fct_rev()` function from the {forcats} package (https://forcats.tidyverse.org/). Expand to see example.*

TRANSFORM AN AXIS

… or transform the default linear mapping by using `scale_y_log10()` or `scale_y_sqrt()`. As an example, here is a $\log_{10}$-transformed axis (which introduces NA's in this case so be careful):

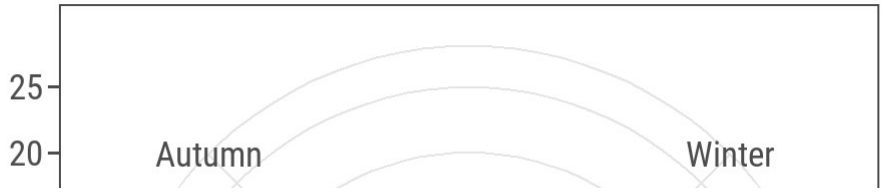
```
ggplot(chic, aes(x = date, y = temp, color = o3)) +
  geom_point() +
  labs(x = "Year", y = "Temperature (°F)") +
  scale_y_log10(lim = c(0.1, 100))
```

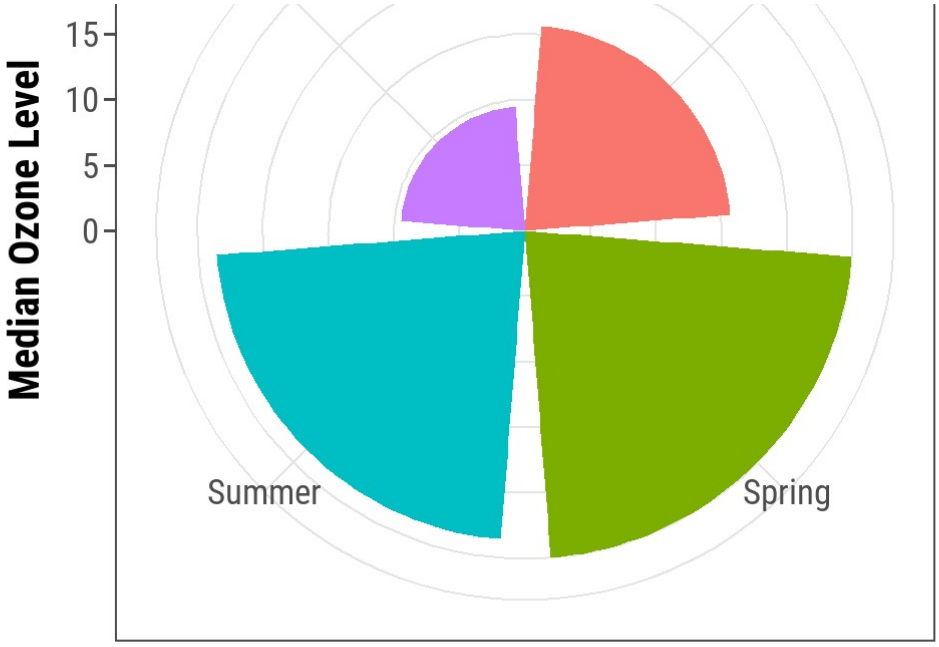


CIRCULARIZE A PLOT

It is also possible to circularize (polarize?) the coordinate system by calling `coord_polar()`.

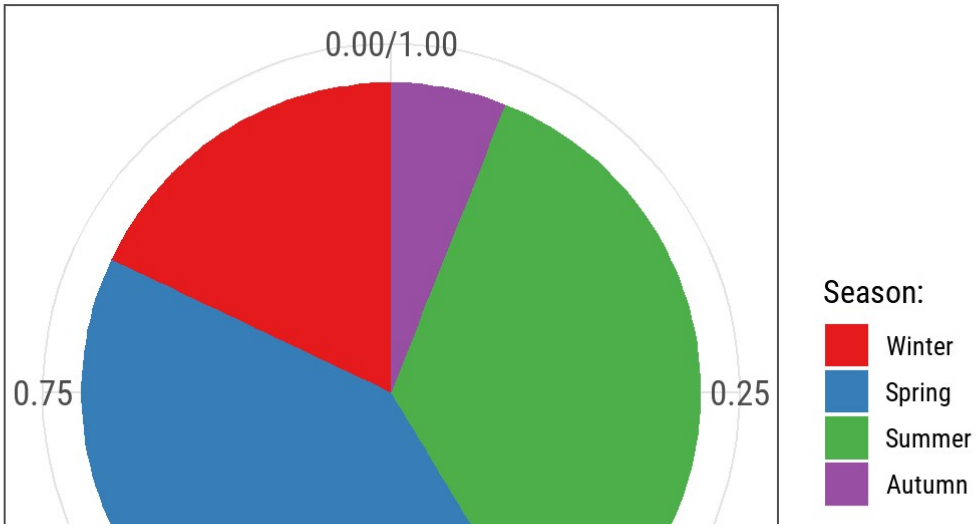
```
library(tidyverse)

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 = FALSE)
```

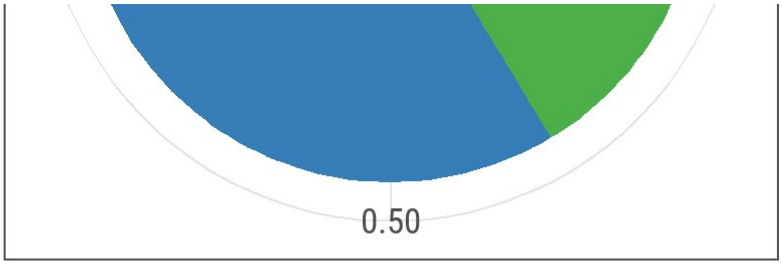




This coordinate system allows to draw pie charts as well:

```
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())
```

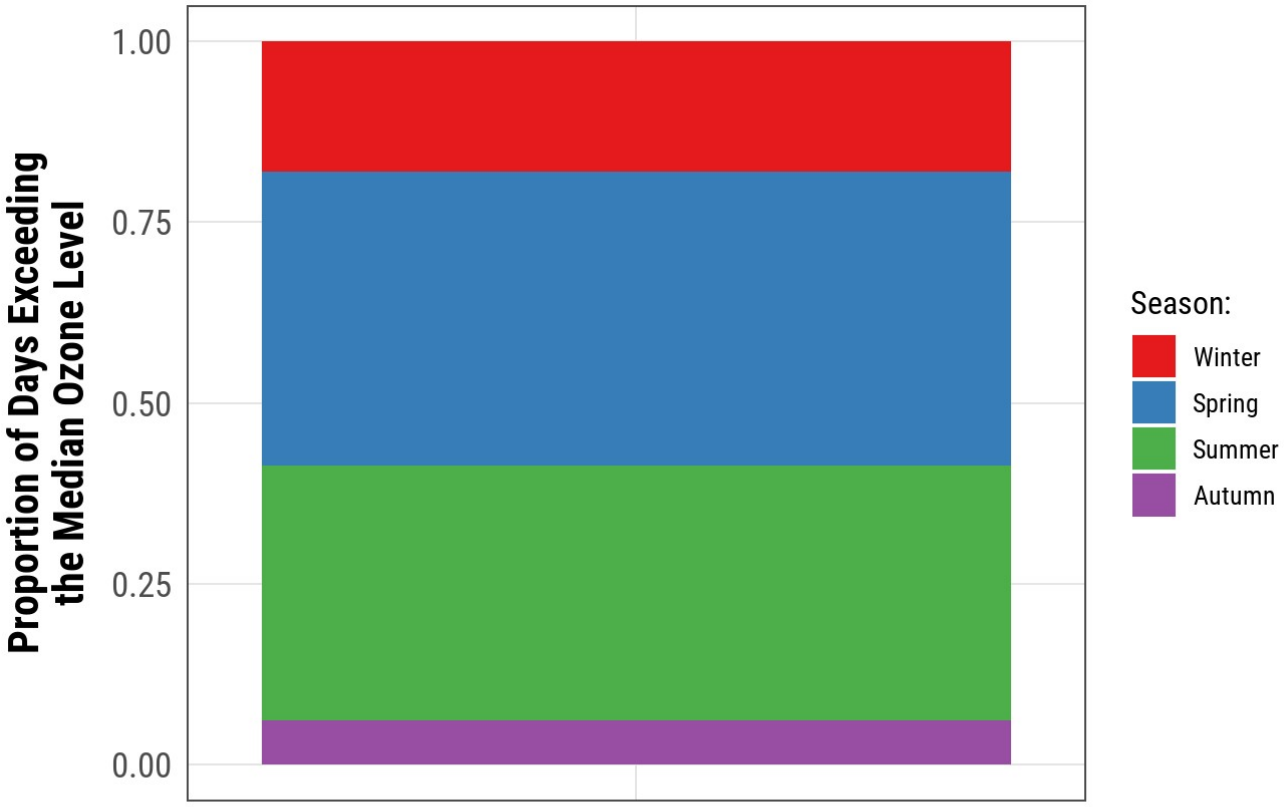




Proportion of Days Exceeding the Median Ozone Level

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())
```



↑ Jump back to Table of Content.

WORKING WITH CHART TYPES

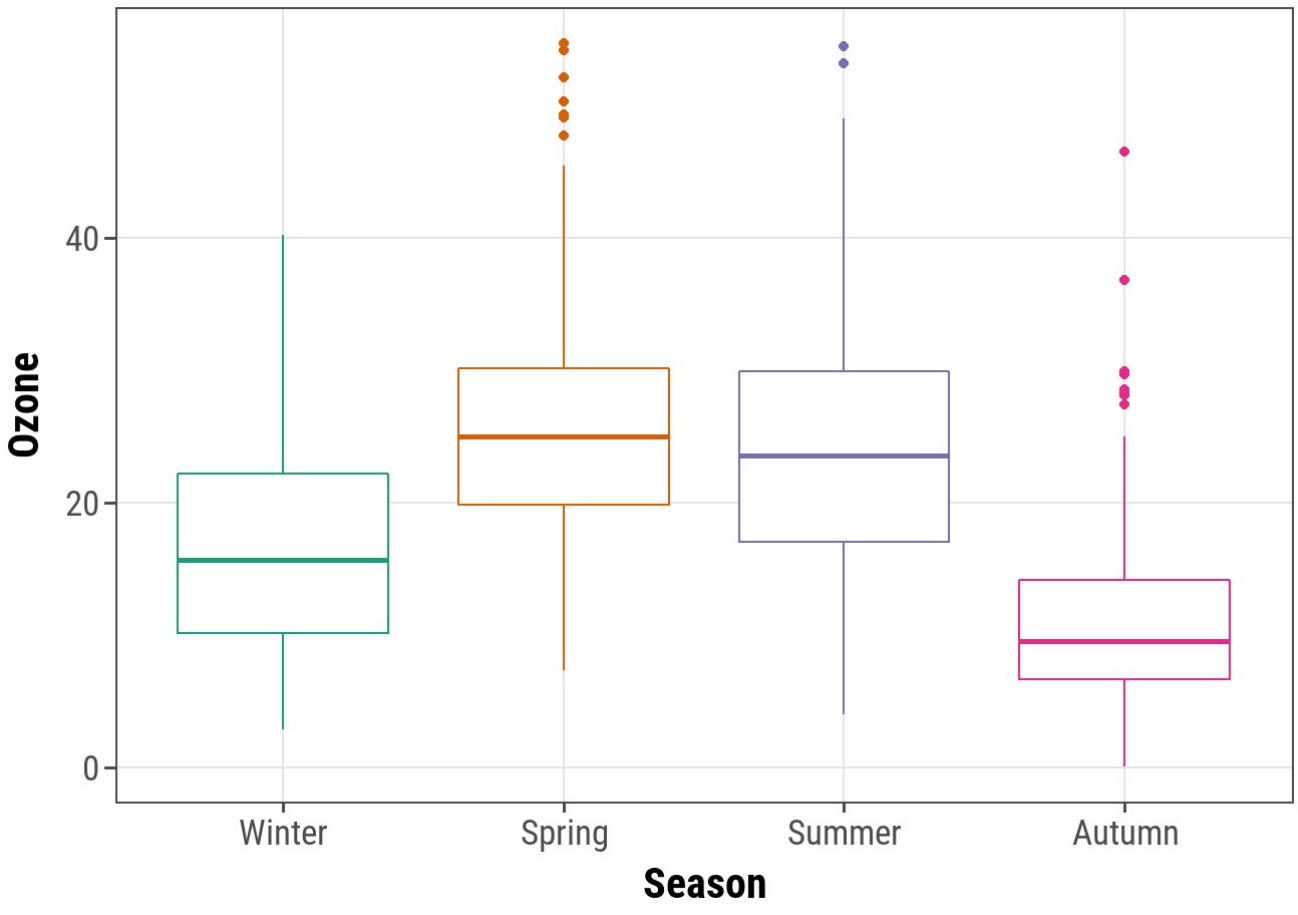
ALTERNATIVES TO A BOX PLOT

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.

🤷 *Expand for a short recap on box and whiskers plots.*

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

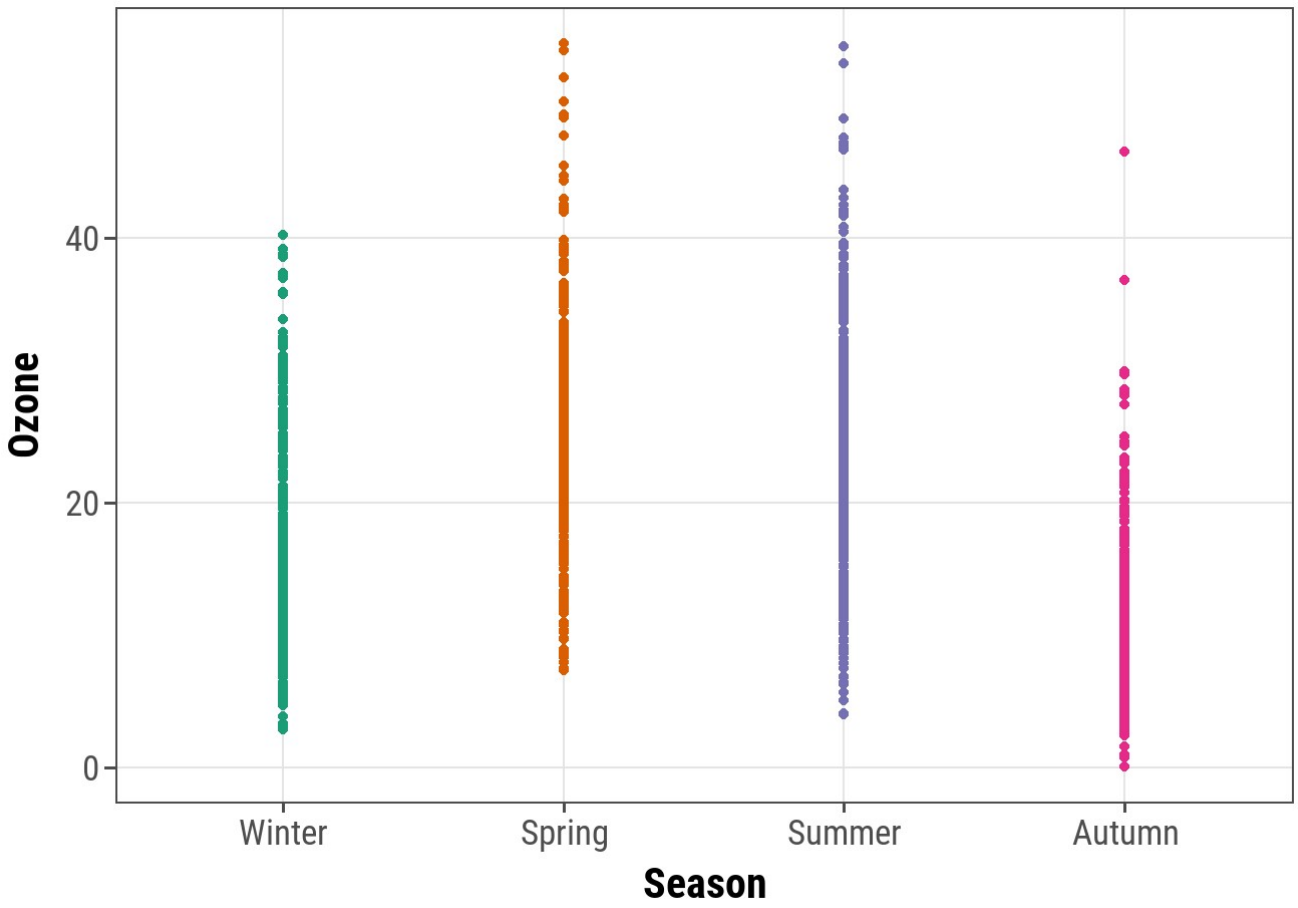
```
g <-
  ggplot(chic, aes(x = season, y = o3,
                   color = season)) +
    labs(x = "Season", y = "Ozone") +
    scale_color_brewer(palette = "Dark2", guide = "none")

g + geom_boxplot()
```



I. ALTERNATIVE: PLOT OF POINTS

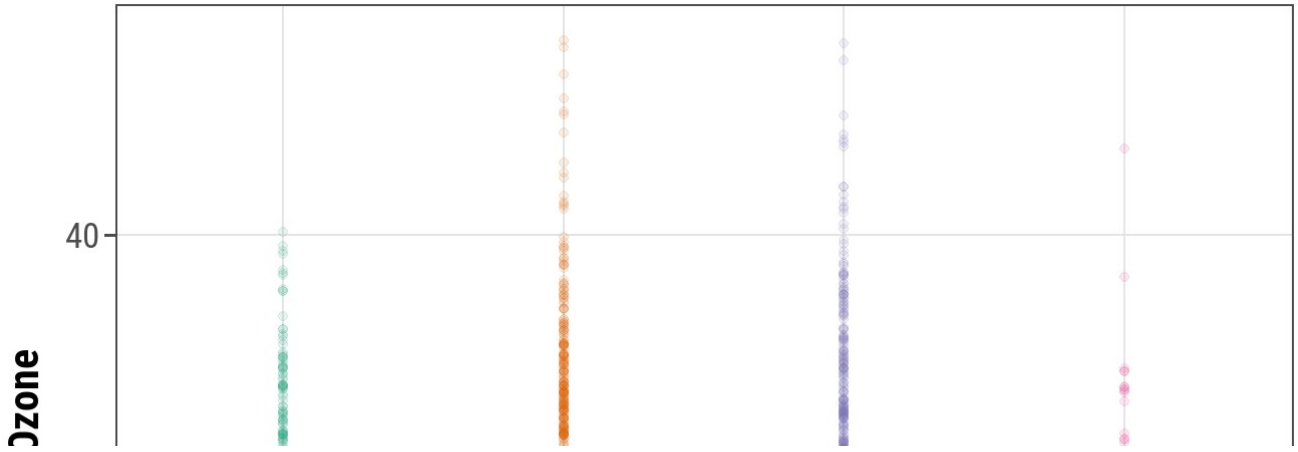
Let's plot just each data point of the raw data:

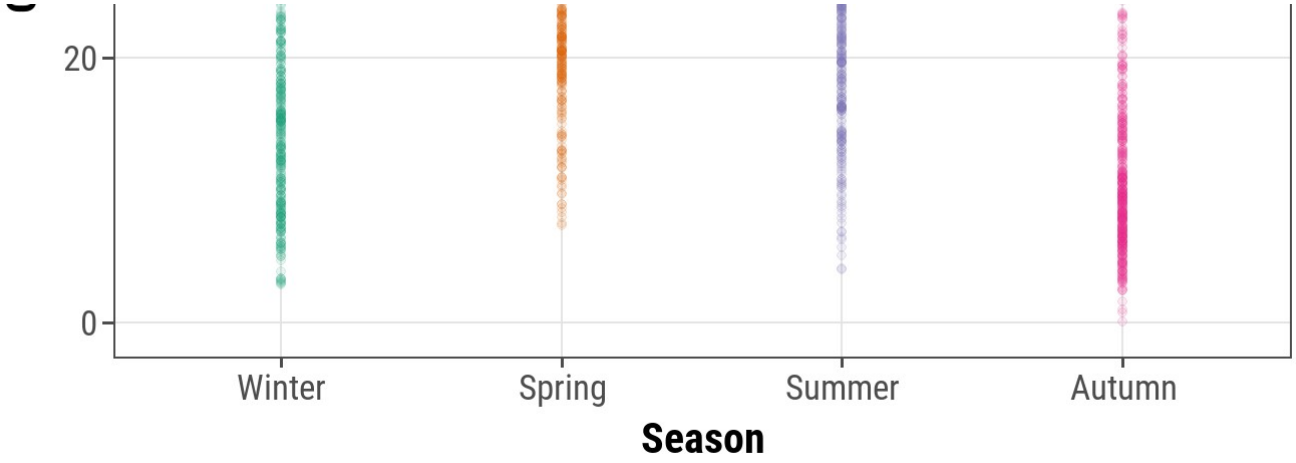
```
g + geom_point()
```



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

```
g + geom_point(alpha = .1)
```



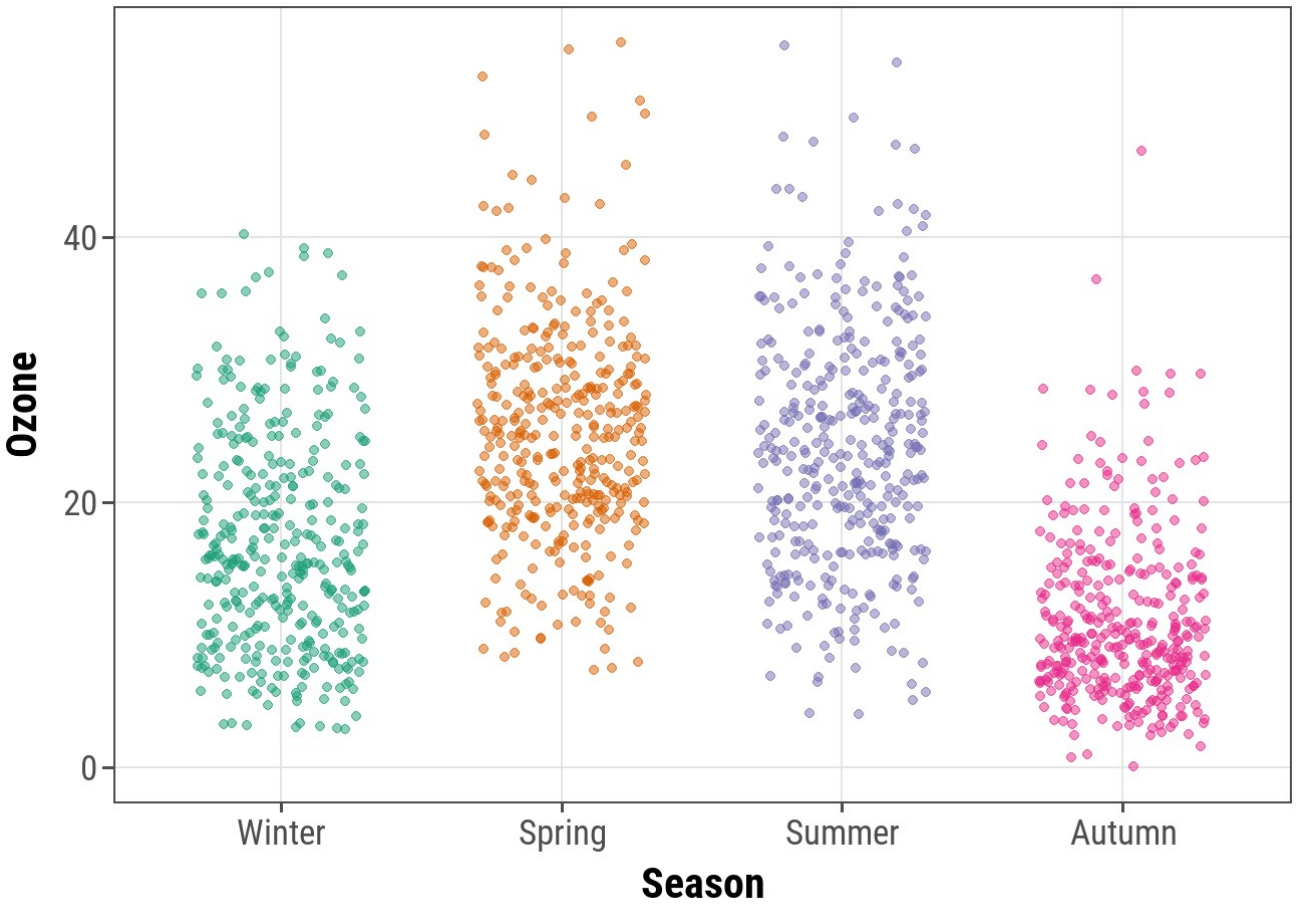


However, setting transparency is difficult here since either the overlap is still too high or the extreme values are not visible. Bad, so let's try something else.

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.

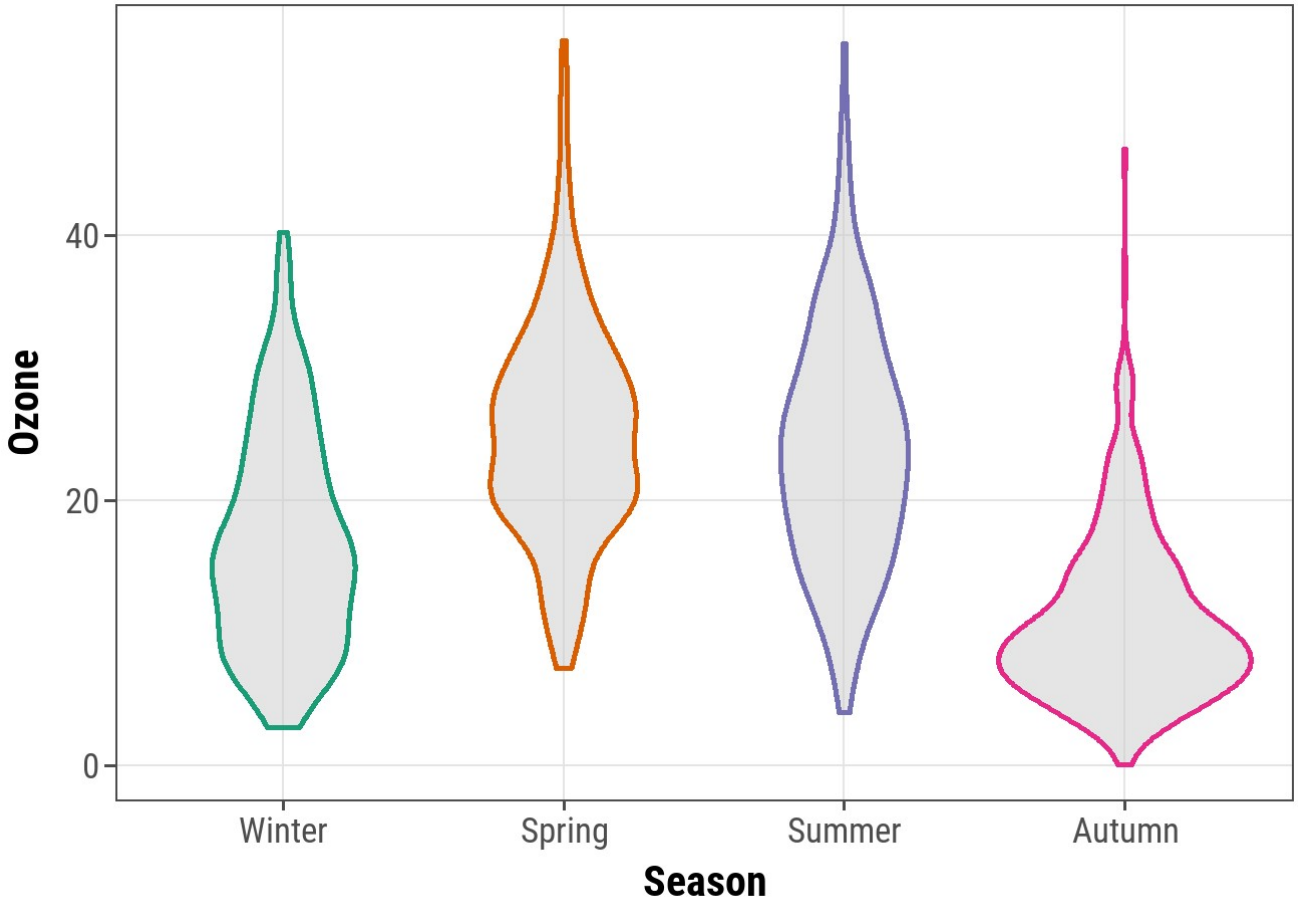
```
g + geom_jitter(width = .3, alpha = .5)
```



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.

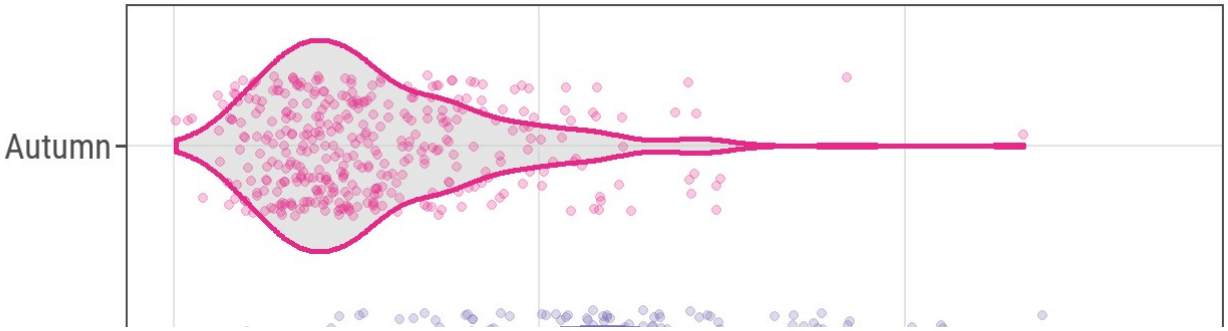
```
g + geom_violin(fill = "gray80", size = 1, alpha = .5)
```

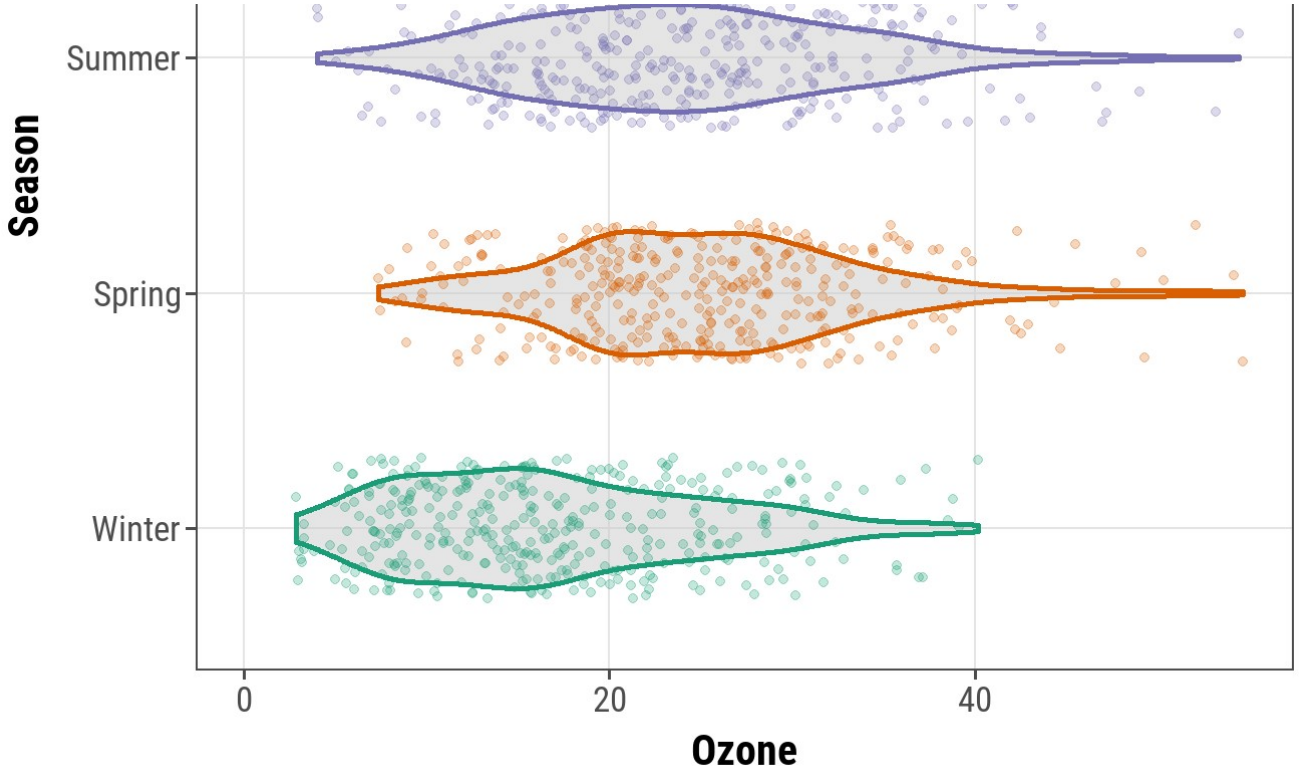


4. ALTERNATIVE: COMBINING VIOLIN PLOTS WITH JITTER

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

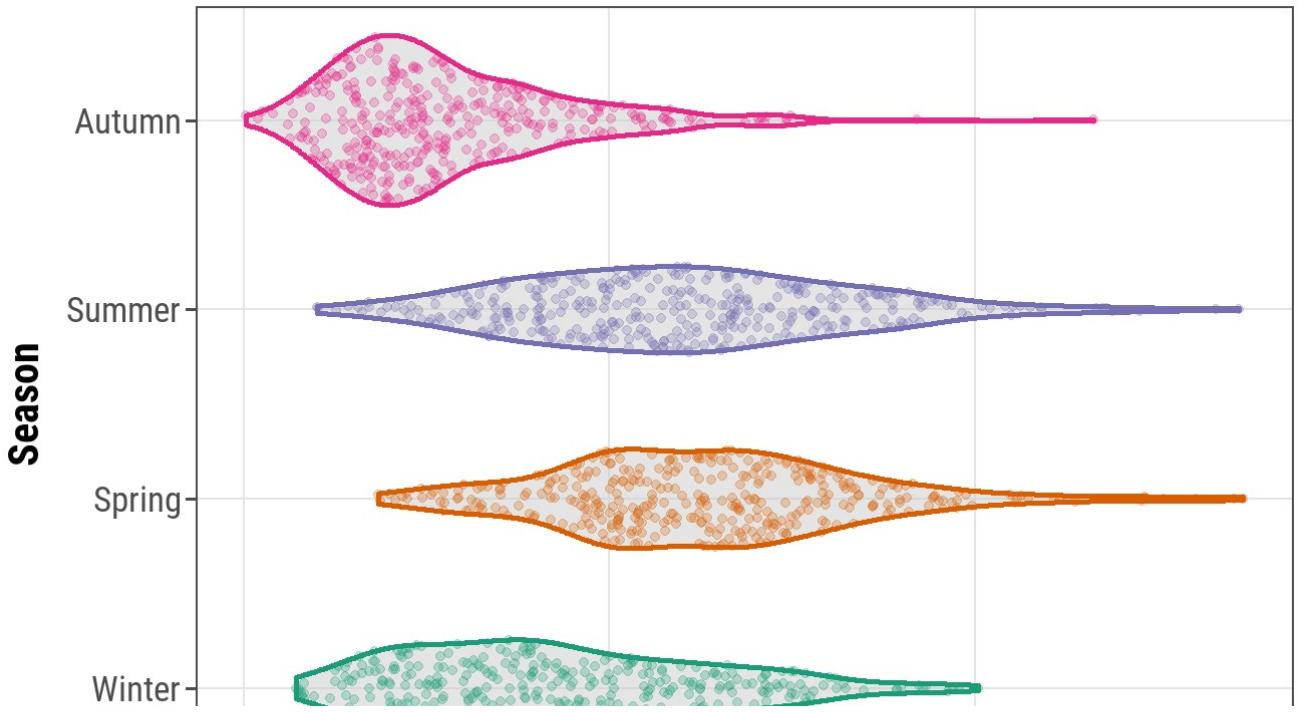
```
g + geom_violin(fill = "gray80", size = 1, alpha = .5) +
    geom_jitter(alpha = .25, width = .3) +
    coord_flip()
```





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:

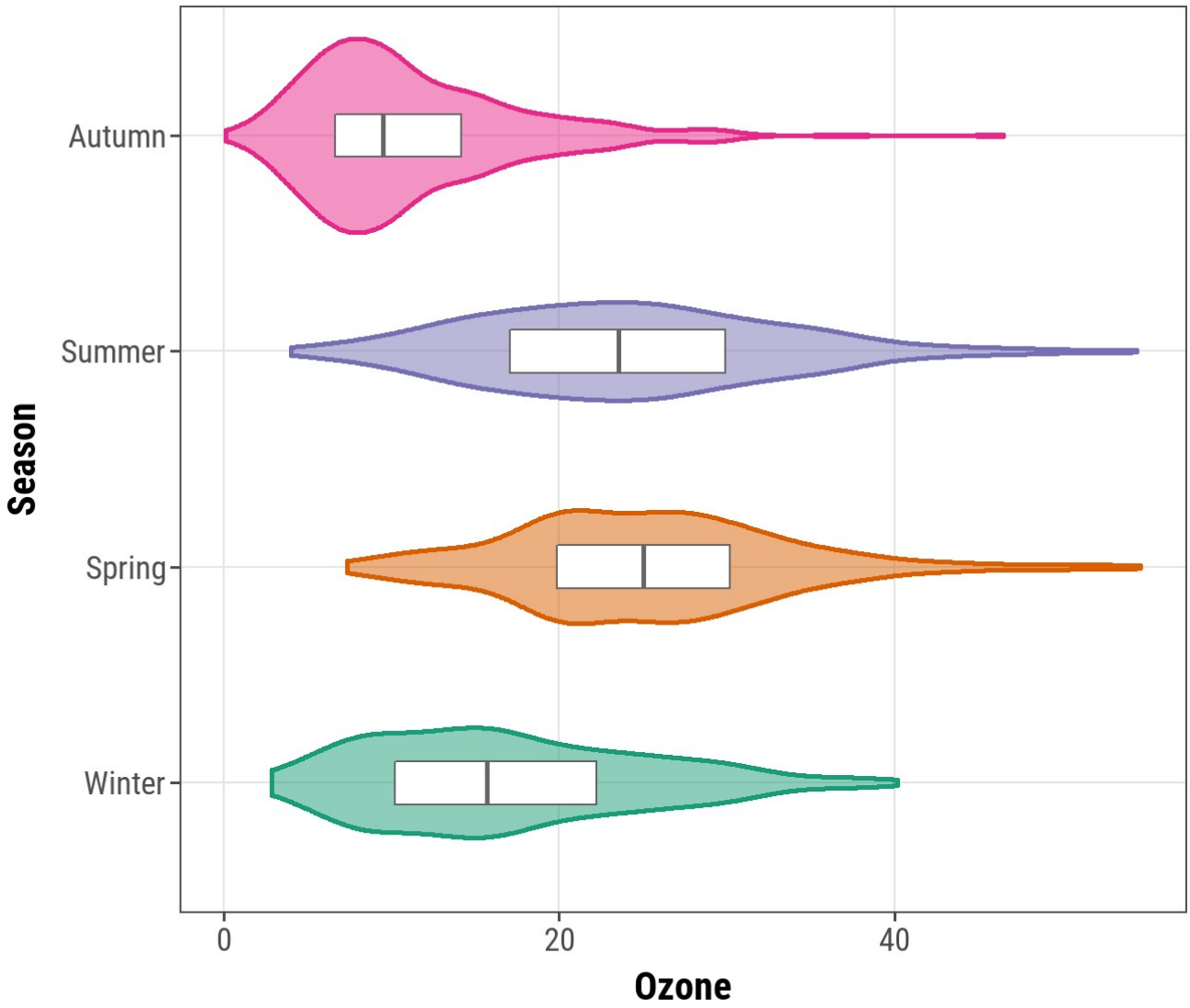
```
library(ggforce)

g + geom_violin(fill = "gray80", size = 1, alpha = .5) +
    geom_sina(alpha = .25) +
    coord_flip()
```



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:

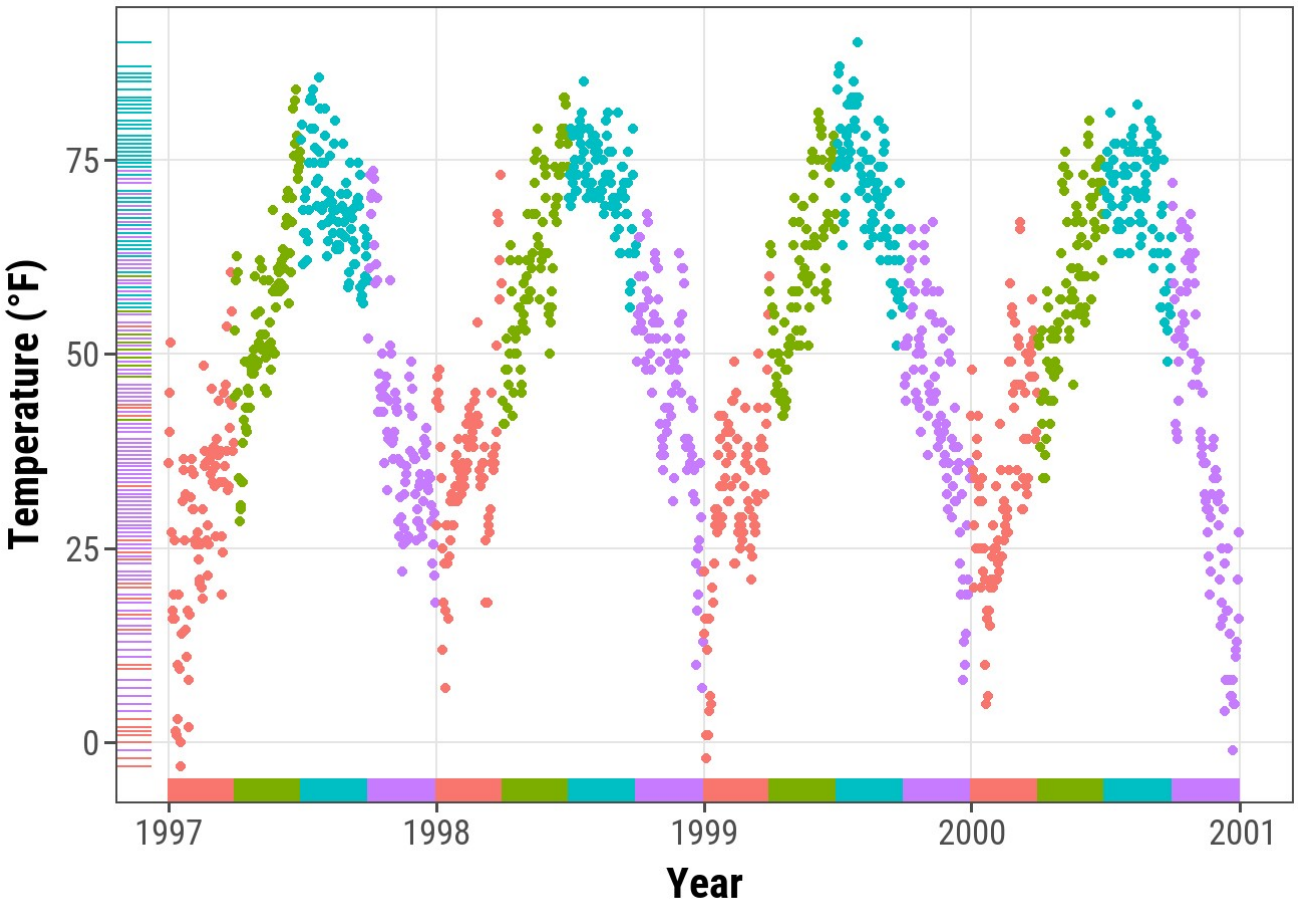
```
g + geom_violin(aes(fill = season), size = 1, alpha = .5) +
    geom_boxplot(outlier.alpha = 0, coef = 0,
                 color = "gray40", width = .2) +
    scale_fill_brewer(palette = "Dark2", guide = "none") +
    coord_flip()
```



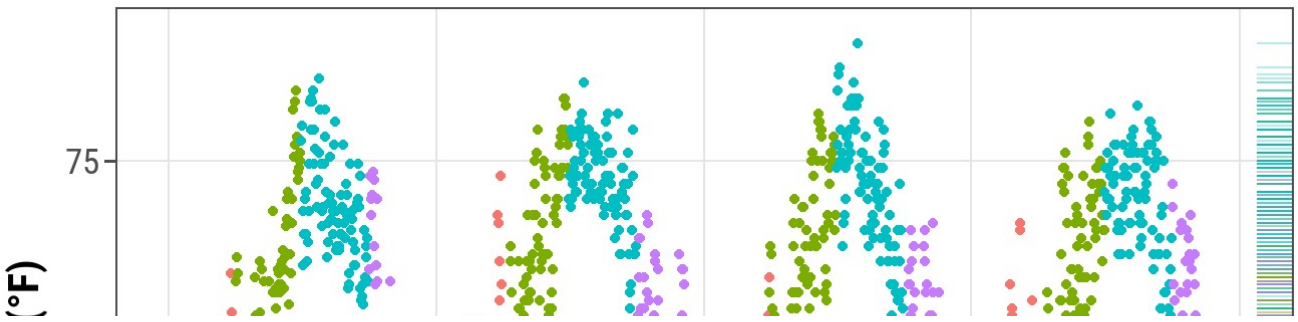
CREATE A RUG REPRESENTATION TO A PLOT

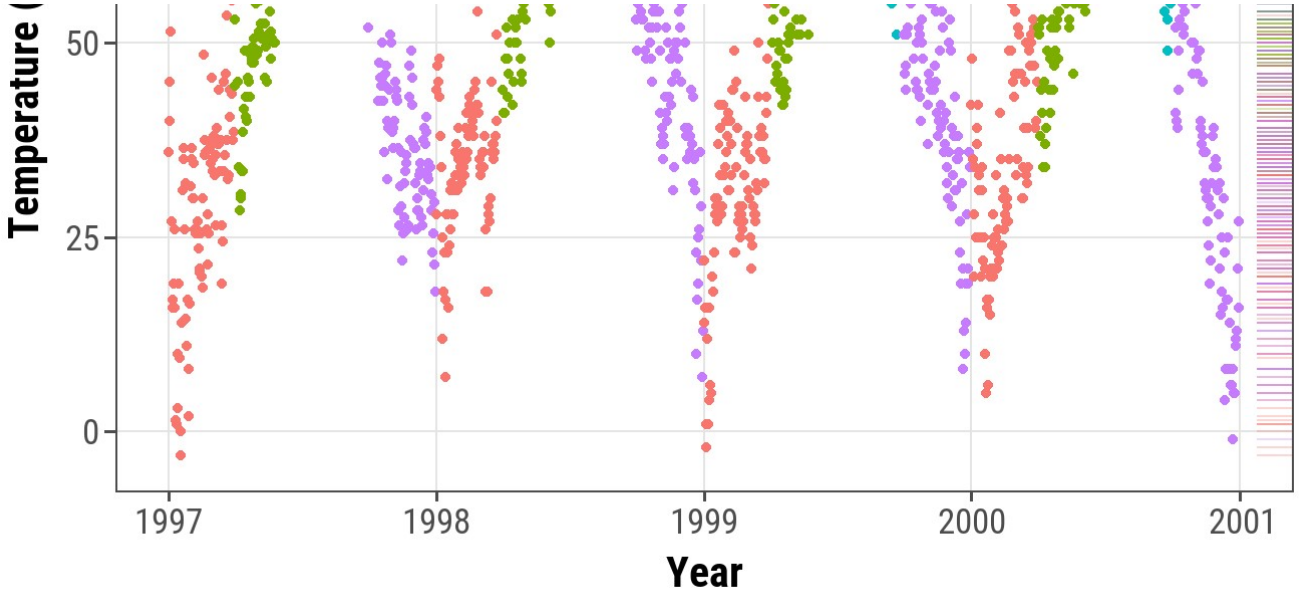
A rug represents the data of a single quantitative variable, displayed as marks along an axis. In most cases, it is used in addition to scatter plots or heatmaps to visualize the overall distribution of one or both of the variables:

```
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)")
```





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`.

```
library(tidyverse)

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

```
## # A tibble: 5 x 6
##   term      death   temp dewpoint     pm10     o3
##   <chr>     <dbl>  <dbl>    <dbl>    <dbl>  <dbl>
## 1 death         1 -0.486   -0.465 -0.00294 -0.238
## 2 temp         NA  1         0.958  0.368    0.535
## 3 dewpoint     NA NA         1      0.327    0.454
## 4 pm10         NA NA        NA      1        0.206
## 5 o3           NA NA        NA     NA        1
```

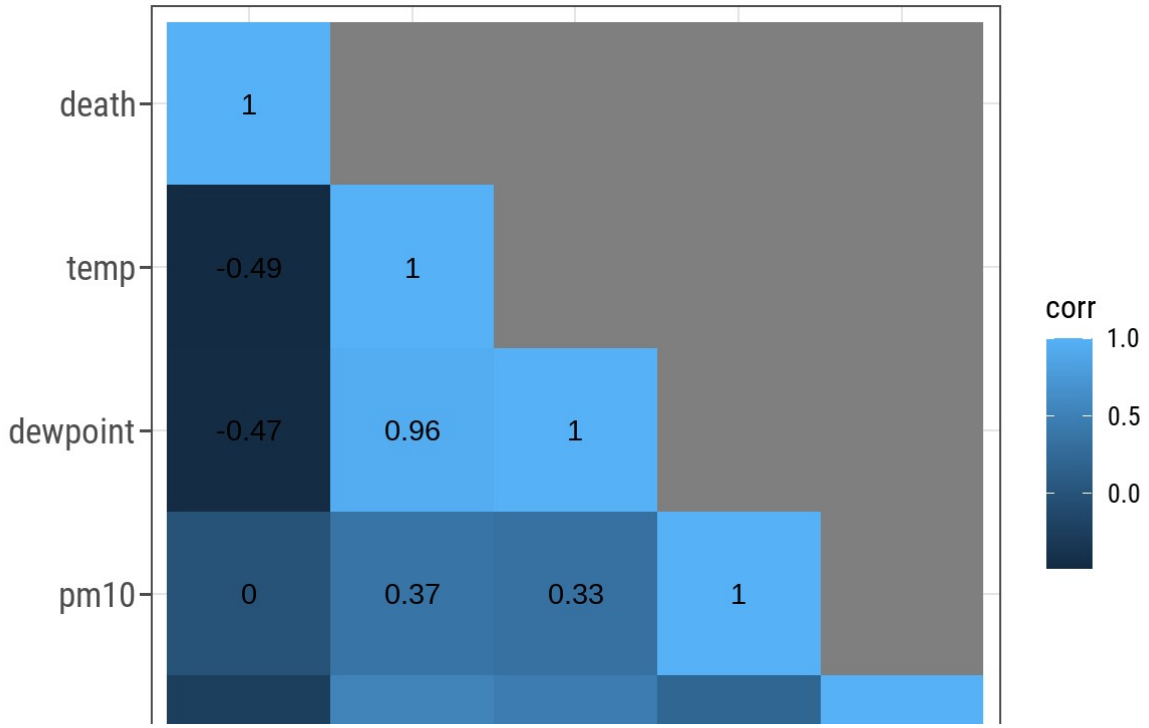
Now we put the resulting matrix in **long** format using the `pivot_longer()` function from the `{tidyr}` package:

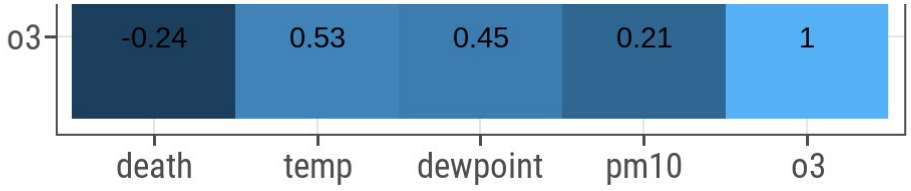
```
corm <- corm %>%
  pivot_longer(
    cols = -term,
    names_to = "colname",
    values_to = "corr"
  ) %>%
  mutate(rowname = fct_inorder(term),
         colname = fct_inorder(colname))
```

```
## # A tibble: 25 x 4
##    term  colname      corr rowname
##    <chr> <fct>       <dbl> <fct>
##  1 death death     1       death
##  2 death temp     -0.486   death
##  3 death dewpoint -0.465   death
##  4 death pm10     -0.00294 death
##  5 death o3       -0.238   death
##  6 temp  death    NA       temp
##  7 temp  temp      1       temp
##  8 temp  dewpoint  0.958   temp
##  9 temp  pm10      0.368   temp
## 10 temp  o3        0.535   temp
## # ... with 15 more rows
```

For the plot we will use `geom_tile()` for the heatmap and `geom_text()` for the labels:

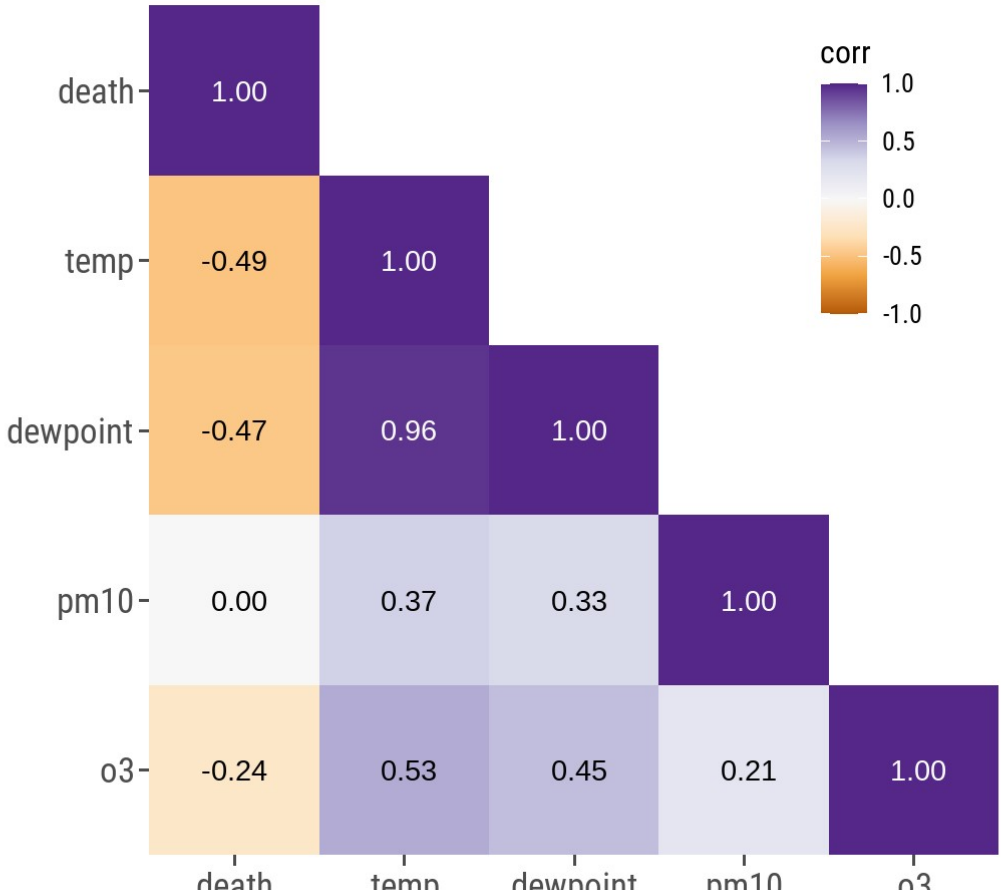
```
ggplot(corm, aes(rowname, fct_rev(colname),
                 fill = corr)) +
  geom_tile() +
  geom_text(aes(label = round(corr, 2))) +
  coord_fixed() +
  labs(x = NULL, y = NULL)
```





I like to have a diverging color palette, 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 nicely formatted labels that are colored depending on the underlying fill:

```
ggplot(corm, aes(rowname, fct_rev(colname),
                 fill = corr)) +
  geom_tile() +
  geom_text(aes(
    label = format(round(corr, 2), nsmall = 2),
    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)
  ) +
  labs(x = NULL, y = NULL) +
  theme(panel.border = element_rect(color = NA, fill = NA),
        legend.position = c(.85, .8))
```

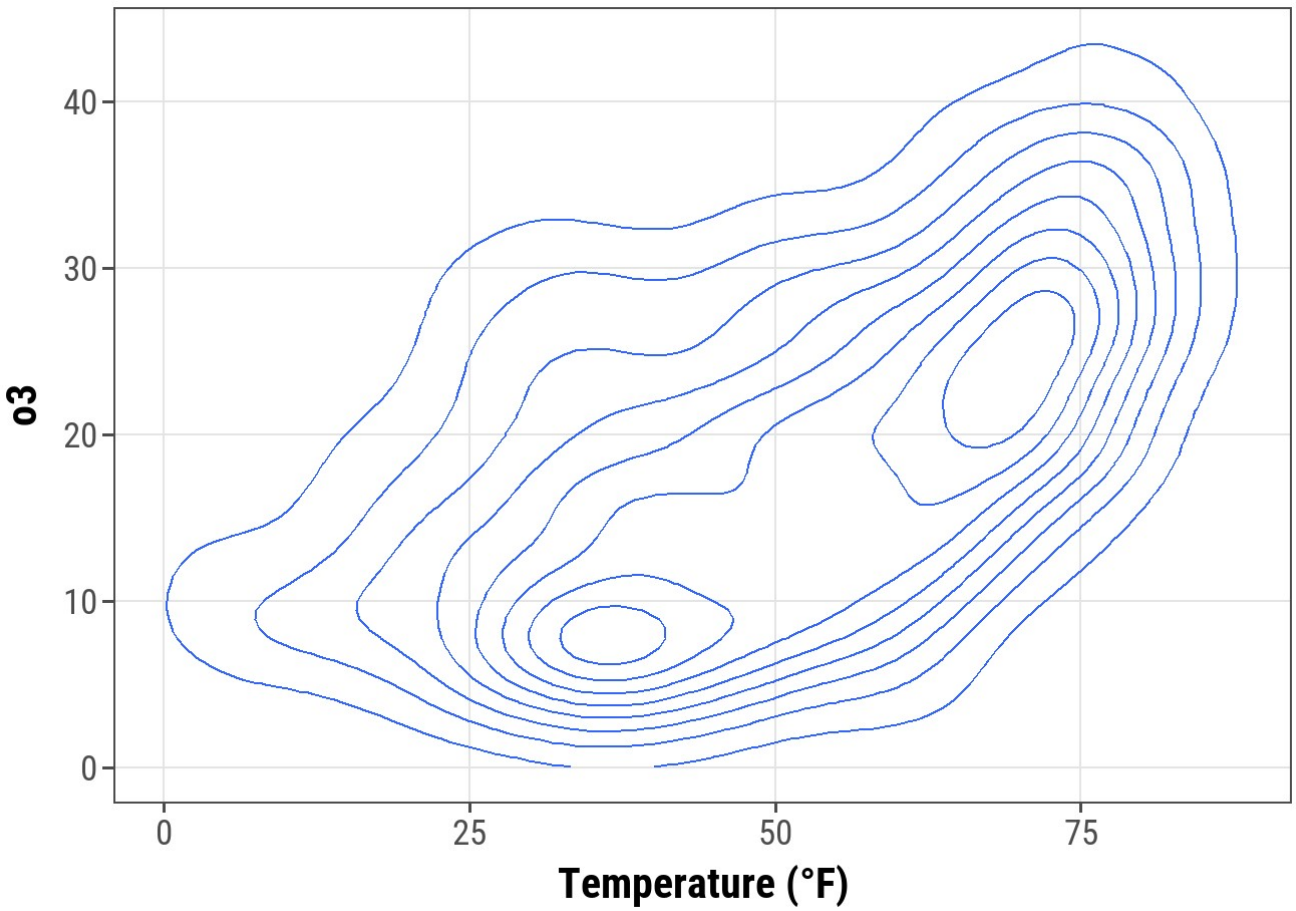


death temp dewpoint pm10 o3

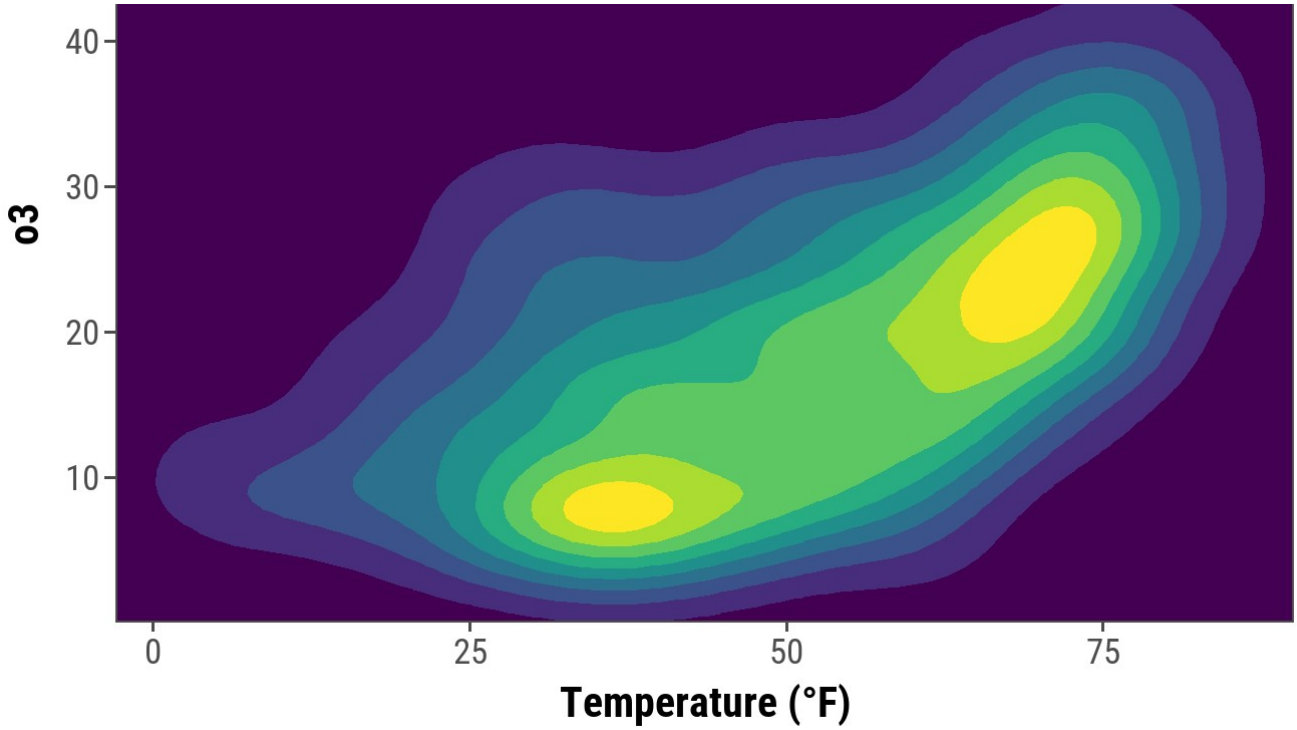
CREATE A CONTOUR PLOT

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")
```

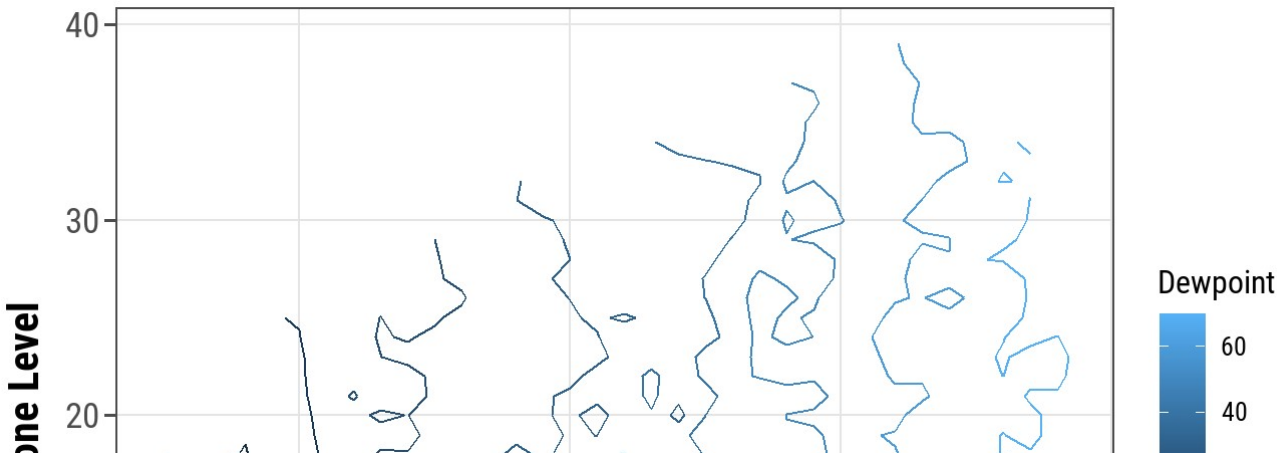


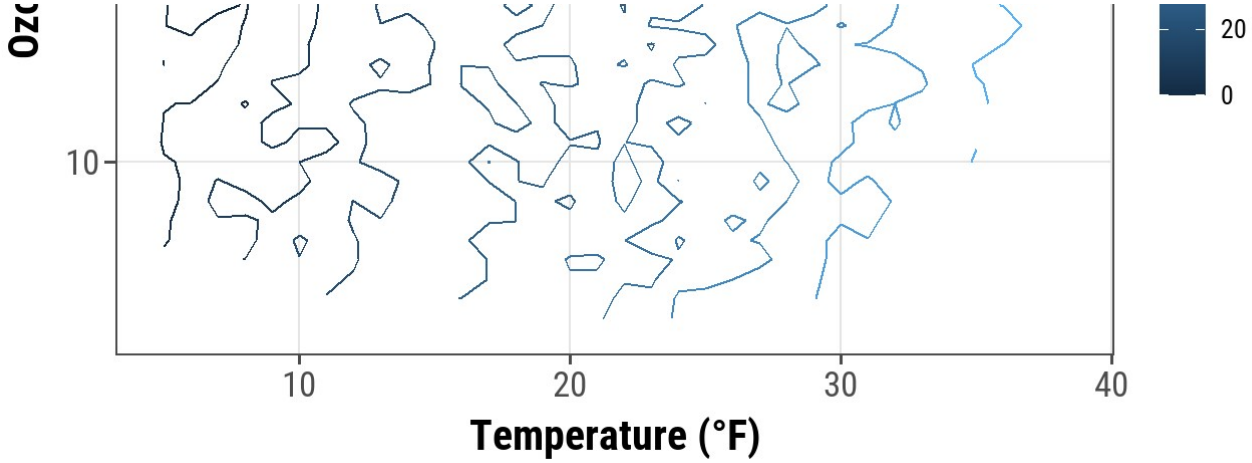

```
ggplot(chic, aes(temp, o3)) +
  geom_density_2d_filled(show.legend = FALSE) +
  coord_cartesian(expand = FALSE) +
  labs(x = "Temperature (°F)", x = "Ozone Level")
```



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
library(akima)
fld <- with(chic, interp(x = temp, y = o3, z = dewpoint))

## prepare data in long format
library(reshape2)
df <- melt(fld$z, na.rm = TRUE)
names(df) <- c("x", "y", "Dewpoint")

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 = ..level.., fill = Dewpoint))
```

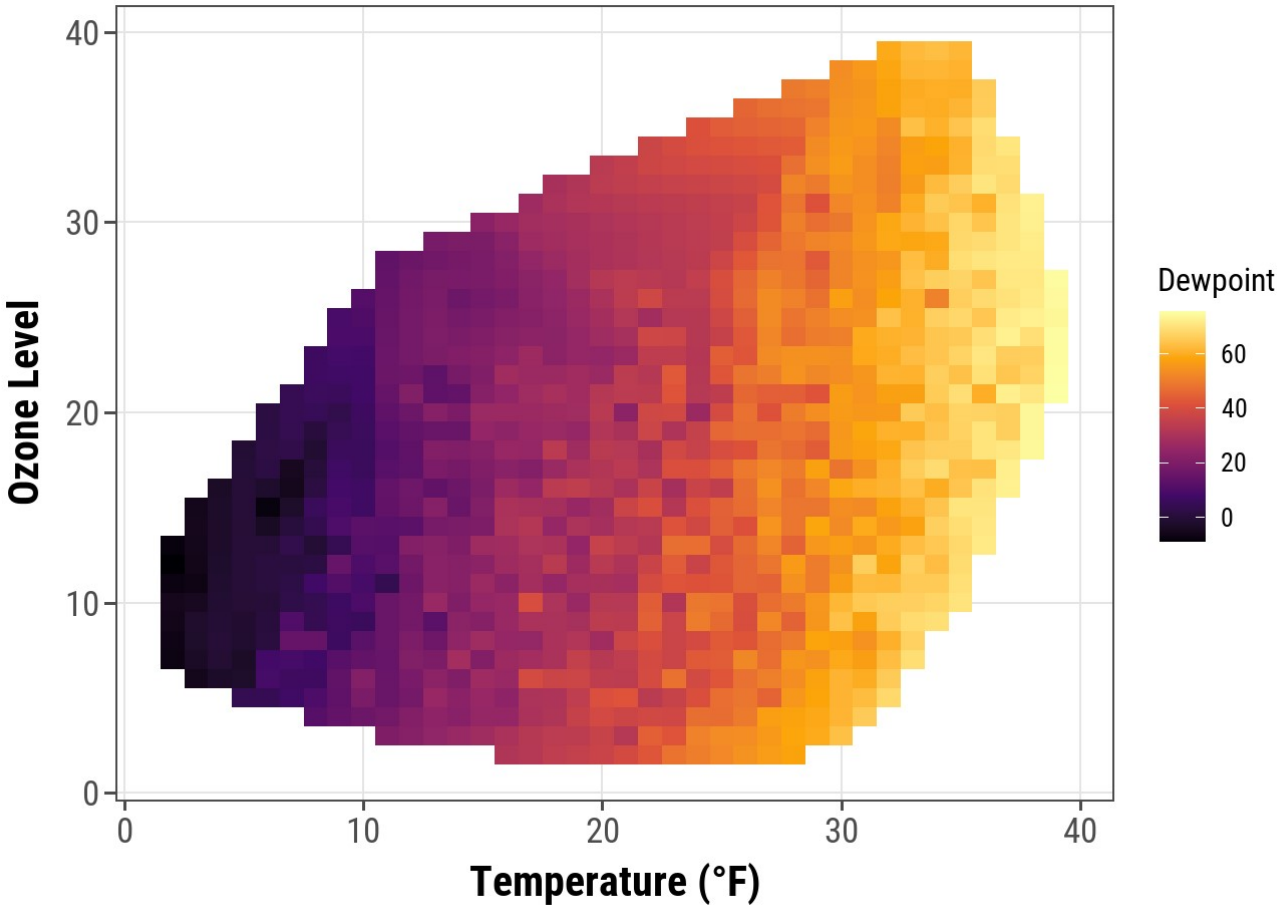




Surprise! As it is defined, the drew point is in most cases equal to the measured temperature.

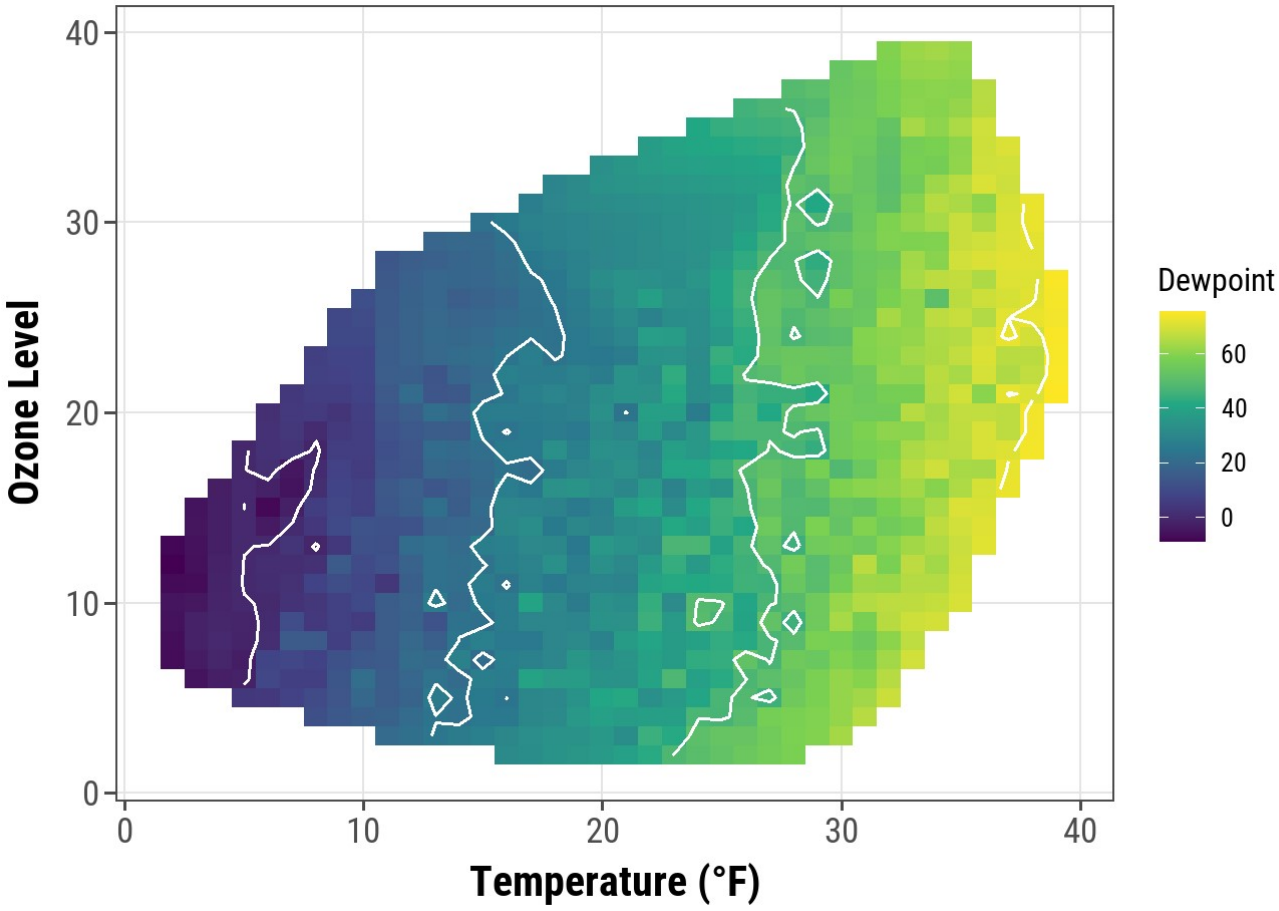
The lines are indicating different levels of drew points, but this is not a pretty plot and also hard to read due to missing borders. Let's try a tile plot using the viridis color palette to encode the dewpoint of each combination of ozone level and temperature:

```
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?

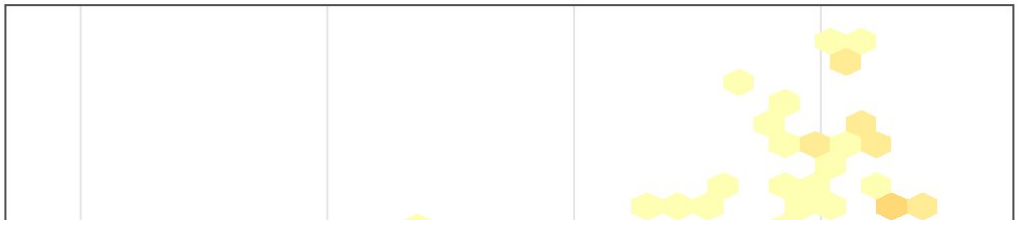
```
g + geom_tile(aes(fill = Dewpoint)) +
    stat_contour(color = "white", size = .7, bins = 5) +
    scale_fill_viridis_c()
```

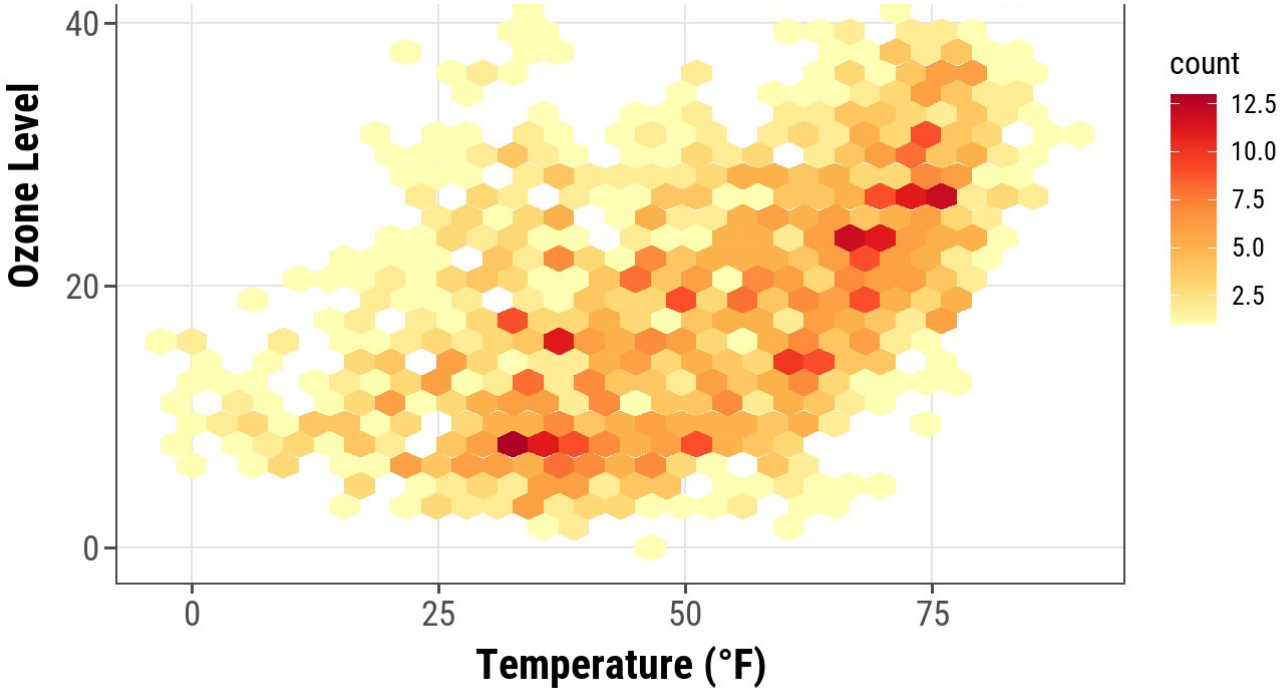


CREATE A HEATMAP

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

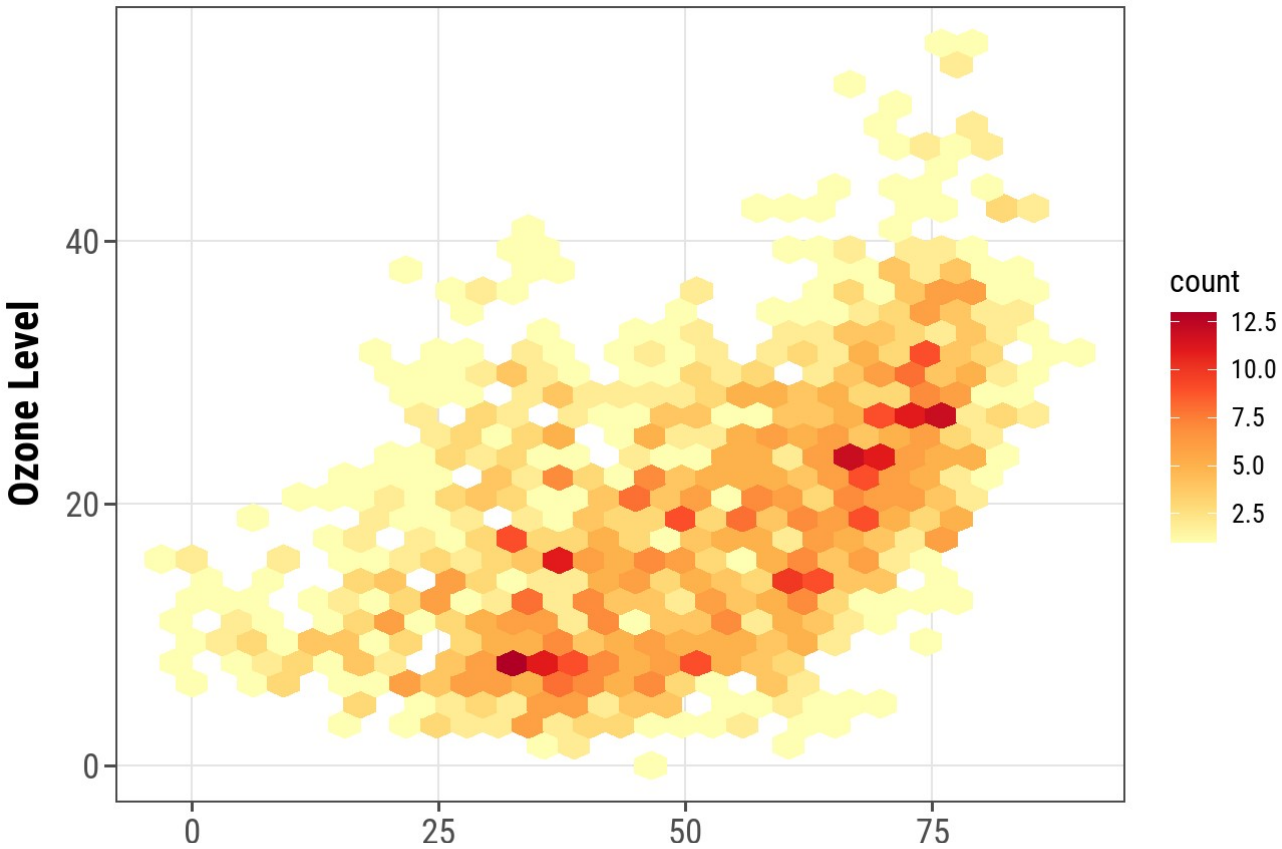
```
ggplot(chic, aes(temp, o3)) +
  geom_hex() +
  scale_fill_distiller(palette = "YlOrRd", direction = 1) +
  labs(x = "Temperature (°F)", y = "Ozone Level")
```





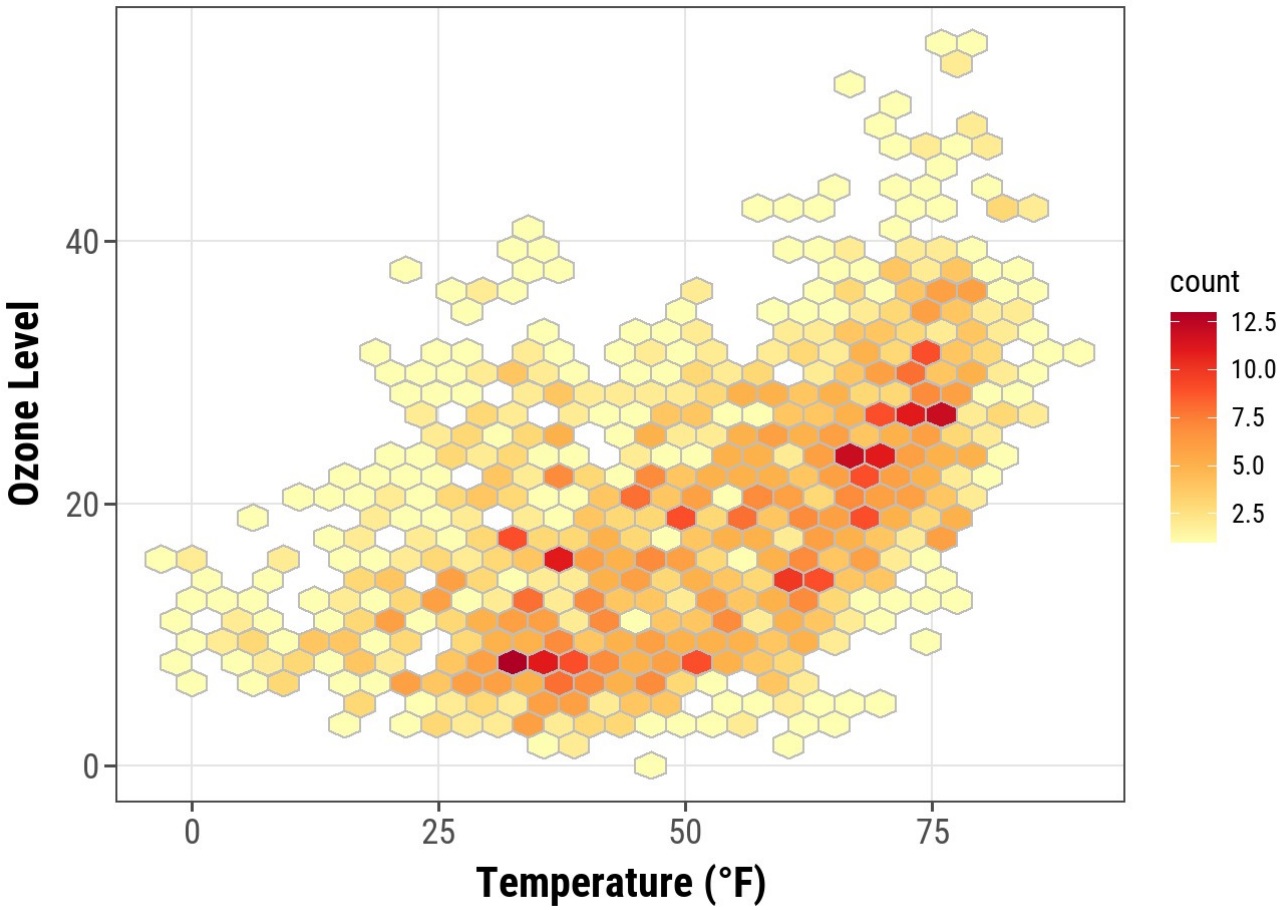
Often, white lines pop up in the resulting plot. One can fix that by mapping also color to either `..count..` (the default) or `..density..` ...

```
ggplot(chic, aes(temp, o3)) +
  geom_hex(aes(color = ..count..)) +
  scale_fill_distiller(palette = "YlOrRd", direction = 1) +
  scale_color_distiller(palette = "YlOrRd", direction = 1) +
  labs(x = "Temperature (°F)", y = "Ozone Level")
```



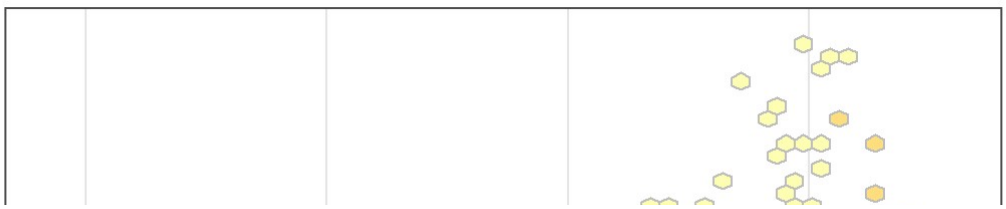
... or by setting the same color as outline for all hexagonal cells:

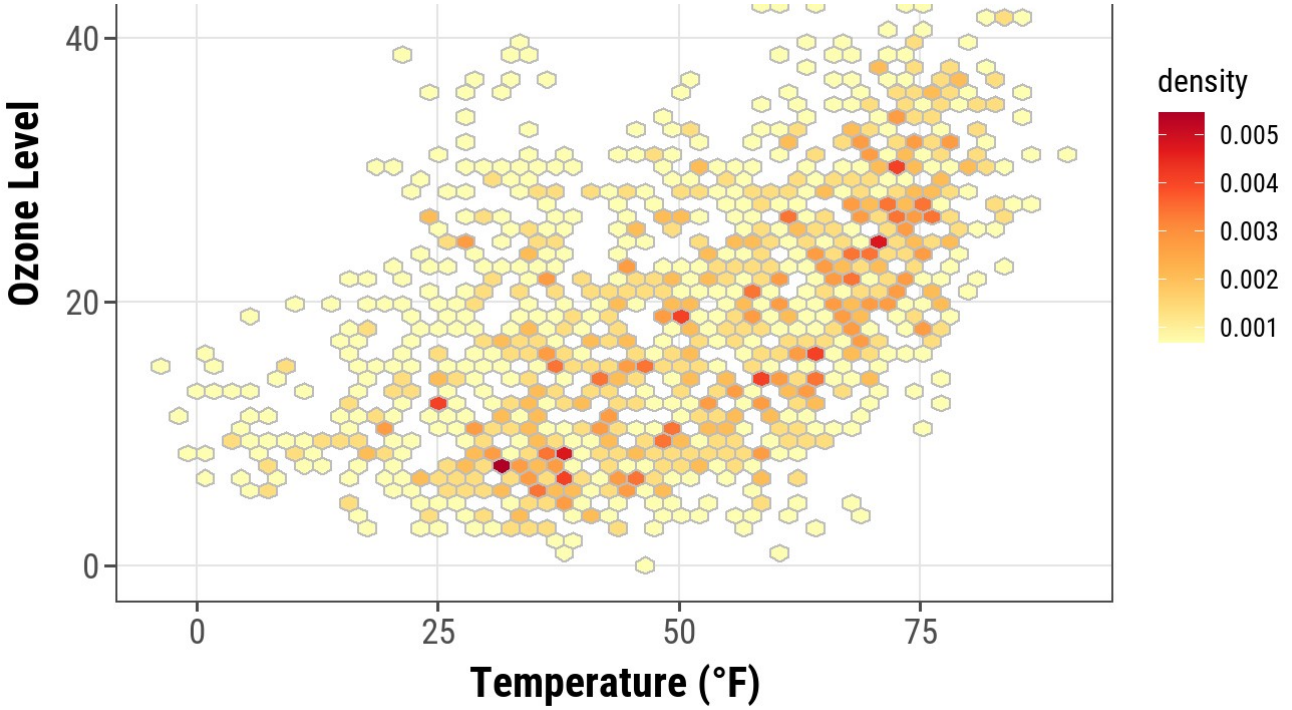
```
ggplot(chic, aes(temp, o3)) +
  geom_hex(color = "grey") +
  scale_fill_distiller(palette = "YlOrRd", direction = 1) +
  labs(x = "Temperature (°F)", y = "Ozone Level")
```



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

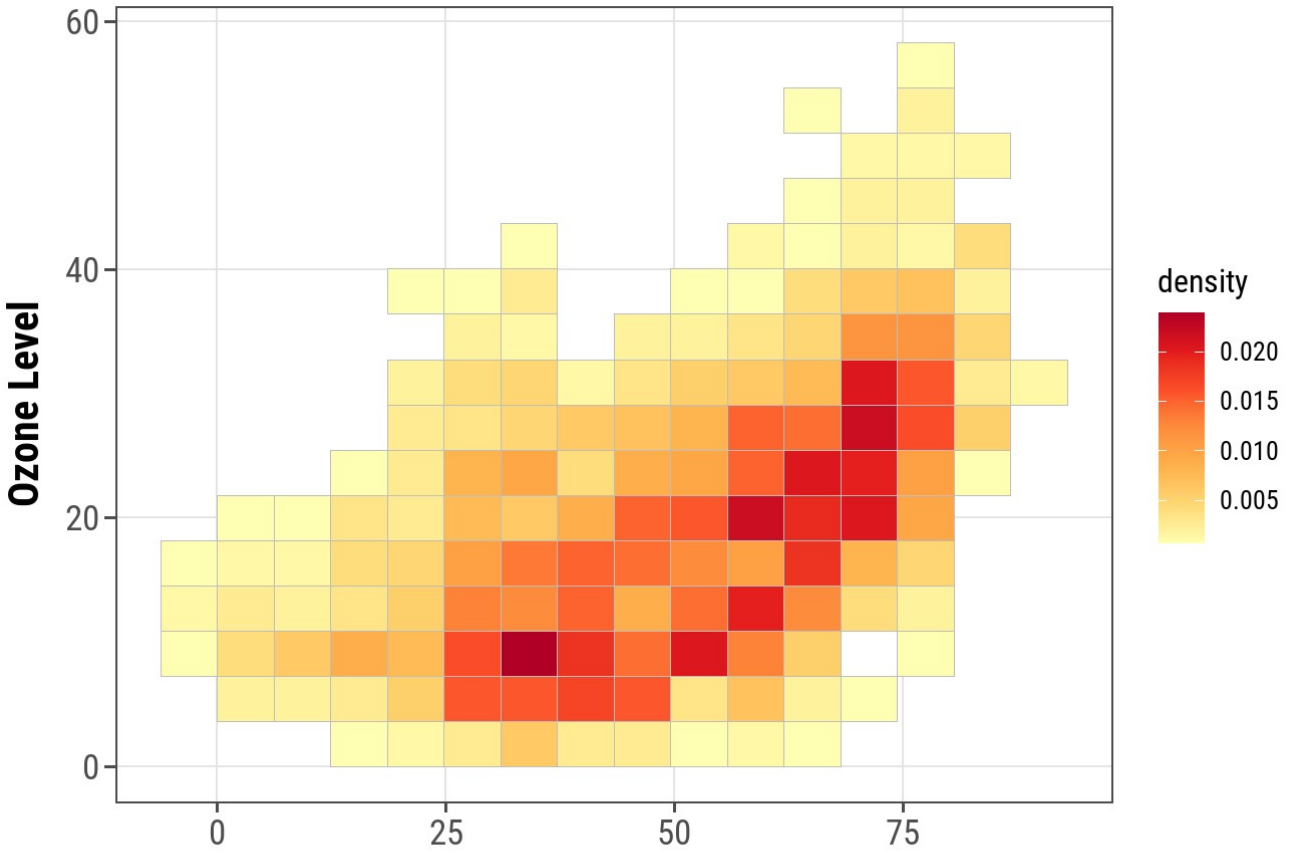
```
ggplot(chic, aes(temp, o3, fill = ..density..)) +
  geom_hex(bins = 50, color = "grey") +
  scale_fill_distiller(palette = "YlOrRd", direction = 1) +
  labs(x = "Temperature (°F)", y = "Ozone Level")
```





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 = ..density..)) +
  geom_bin2d(bins = 15, color = "grey") +
  scale_fill_distiller(palette = "YlOrRd", direction = 1) +
  labs(x = "Temperature (°F)", y = "Ozone Level")
```



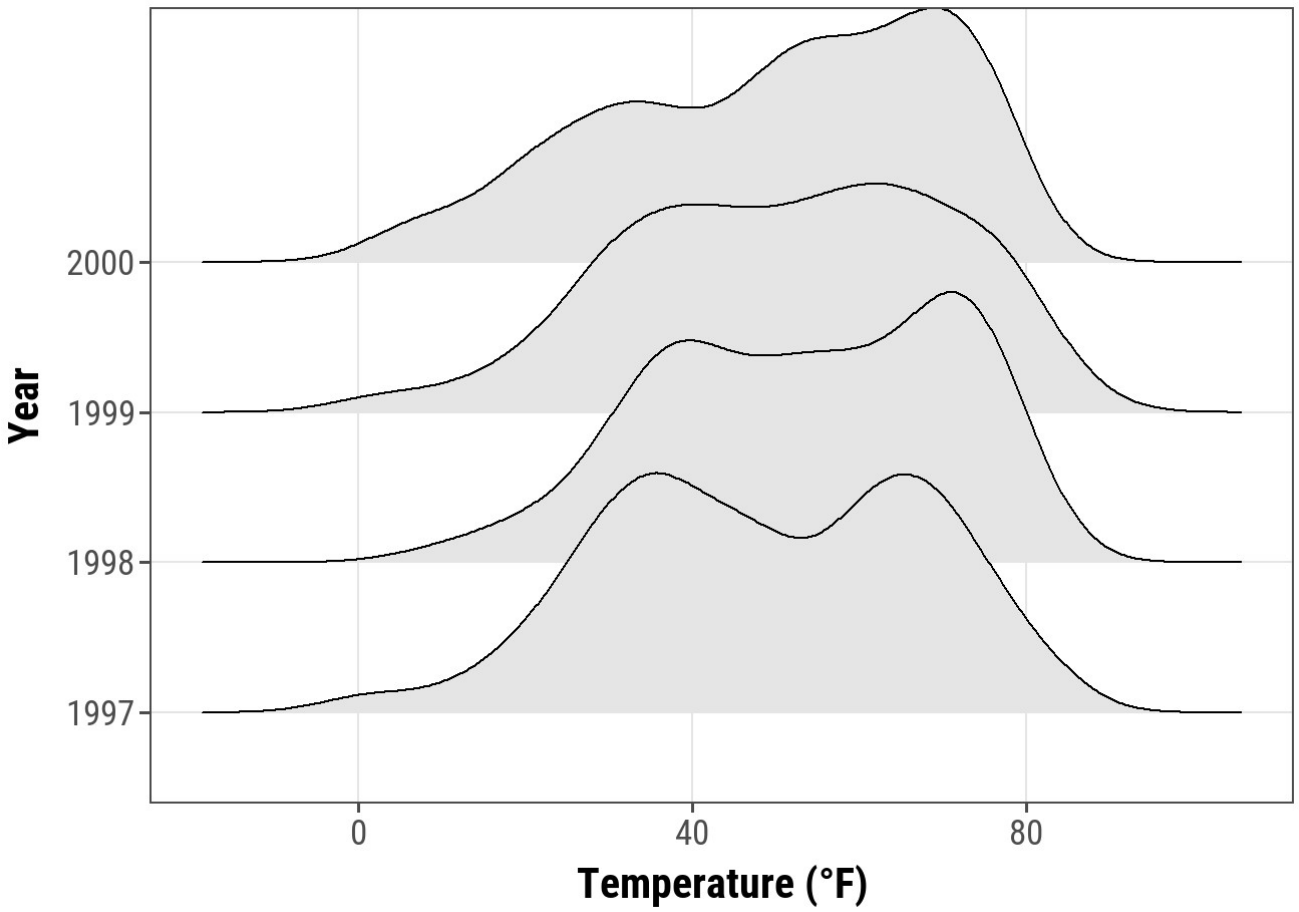
Temperature (°F)

CREATE A RIDGE PLOT

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

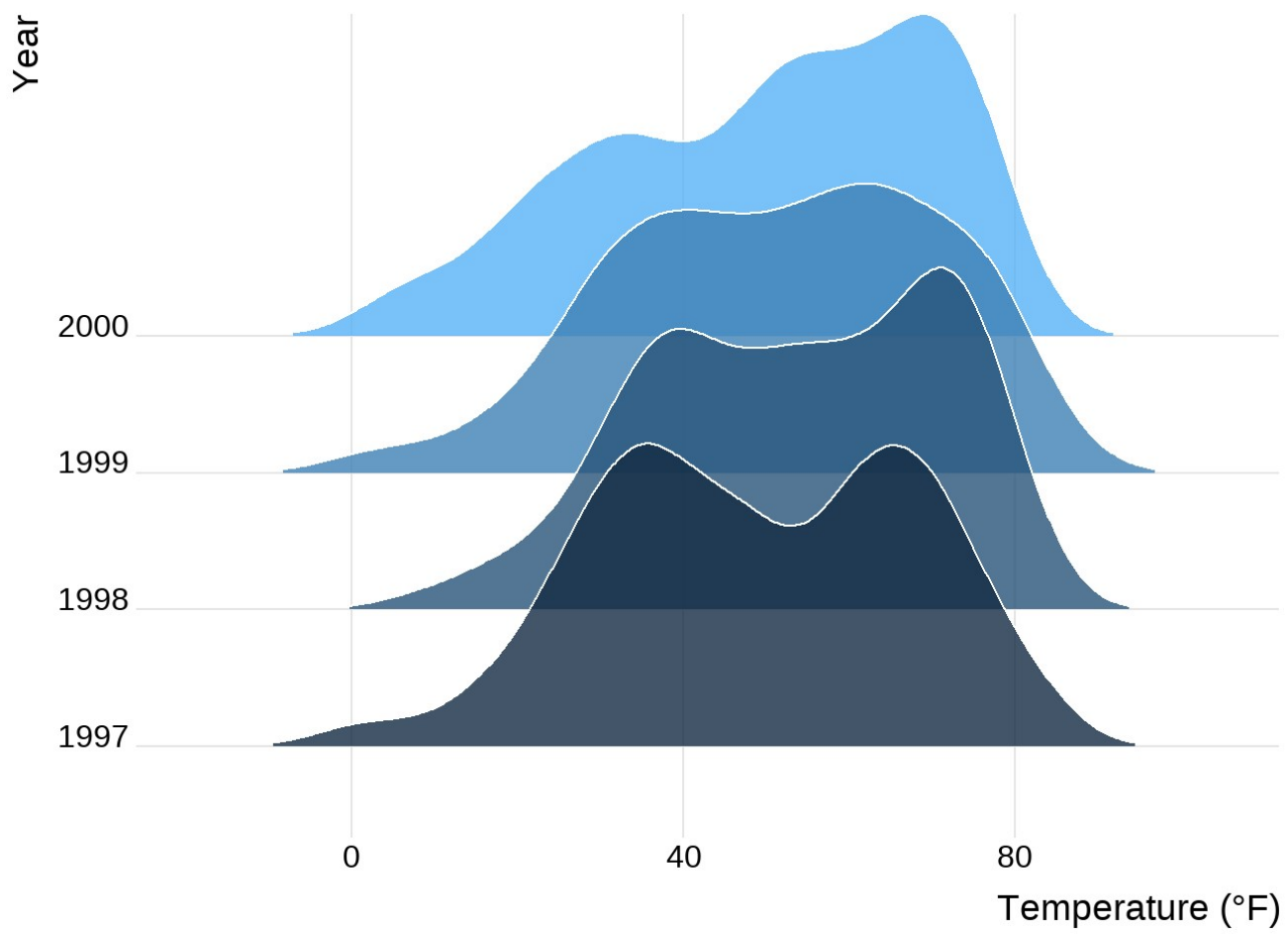
While you can create those plots with basic `{ggplot2}` commands (https://github.com/halhen/viz-pub/blob/master/sports-time-of-day/2_gen_chart.R) the popularity lead to a package that make it easier create those plots: `{ggridges}` (https://cran.r-project.org/web/packages/ggridges/index.html). We are 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")
```



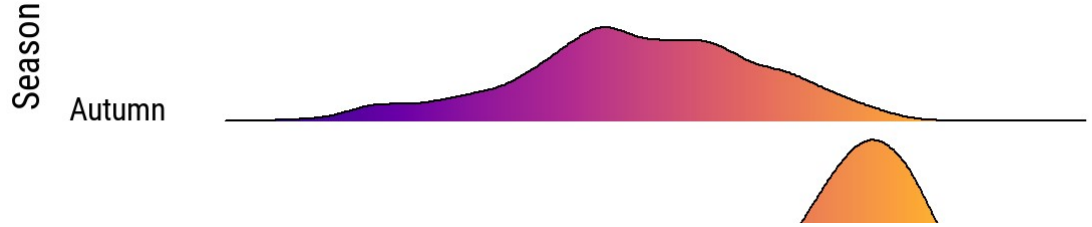
You can easily specify the overlap and the trailing tails by using the arguments `rel_min_height` and `scale`, respectively. The package also comes with its own theme (but I would prefer to build my own, see chapter “Create and Use Your Custom Theme”). Additionally, we change the colors based on year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

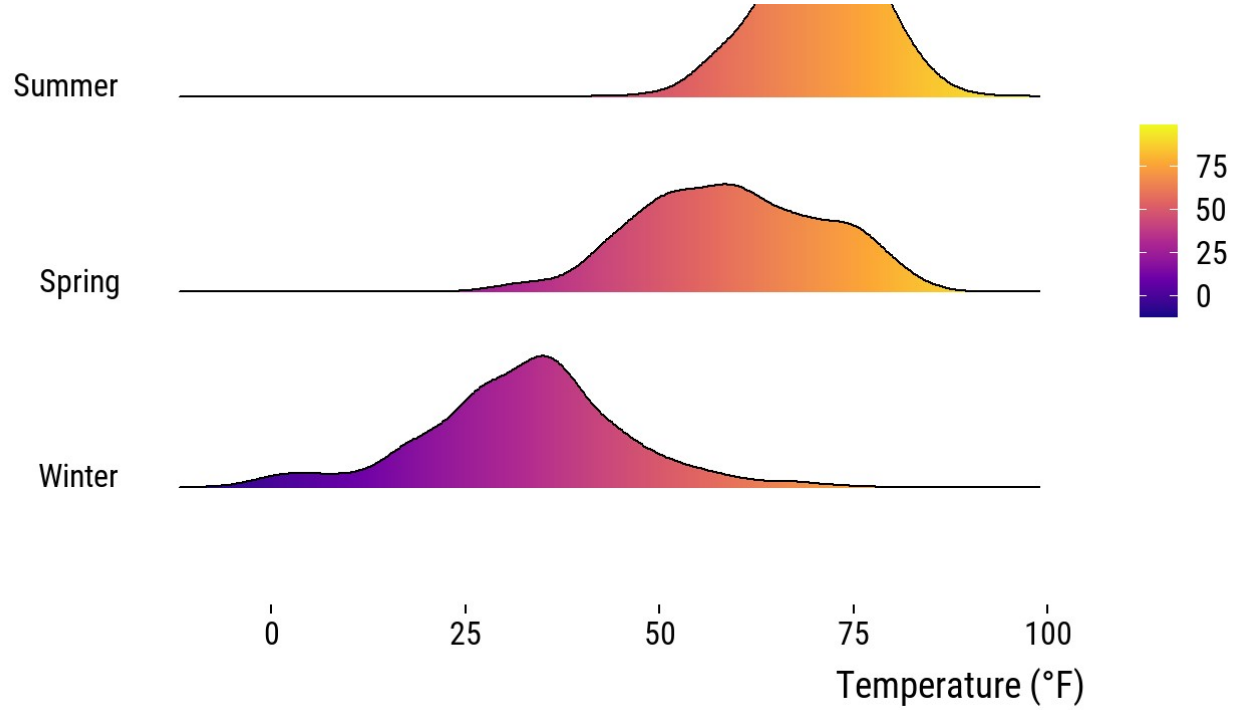
```
ggplot(chic, aes(x = temp, y = factor(year), fill = year)) +
  geom_density_ridges(alpha = .8, color = "white",
                      scale = 2.5, rel_min_height = .01) +
  labs(x = "Temperature (°F)", y = "Year") +
  guides(fill = FALSE) +
  theme_ridges()
```



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:

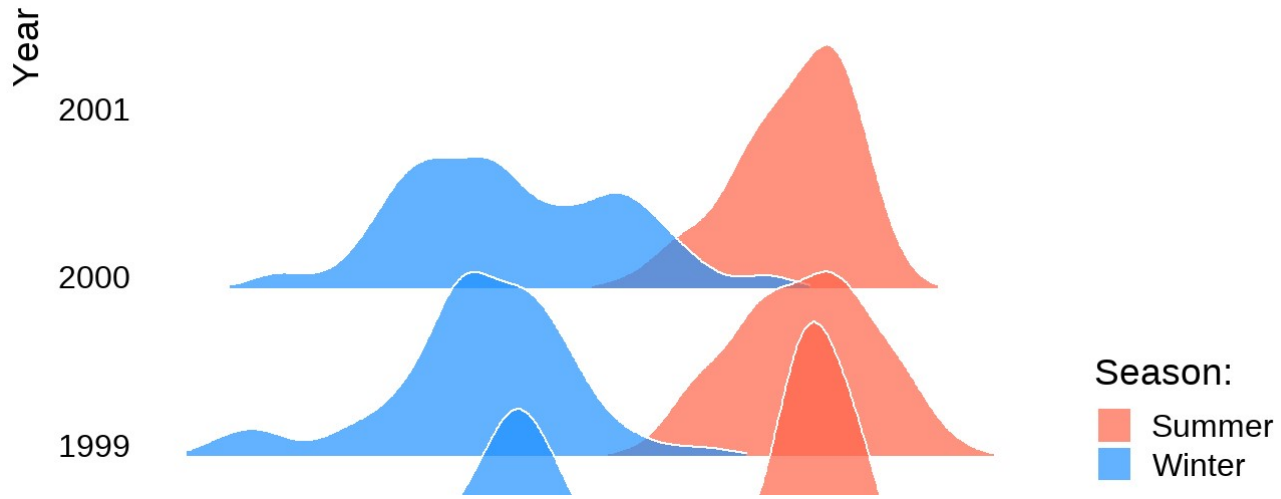
```
ggplot(chic, aes(x = temp, y = season, fill = ..x..)) +
  geom_density_ridges_gradient(scale = .9, gradient_lwd = .5,
                               color = "black") +
  scale_fill_viridis_c(option = "plasma", name = "") +
  labs(x = "Temperature (°F)", y = "Season") +
  theme_ridges(font_family = "Roboto Condensed", grid = FALSE)
```

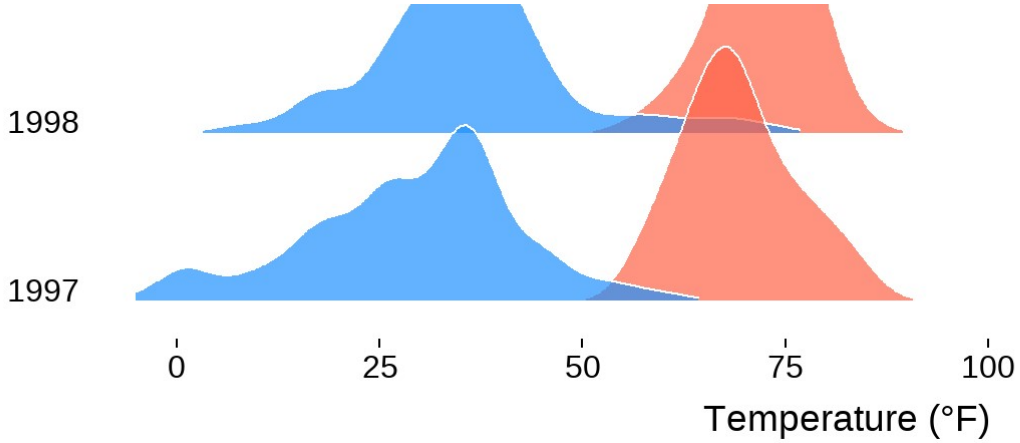




We can also compare several groups per ridgeline and coloring them according to their group. This follows the idea of Marc Belzunces (https://twitter.com/marcbeldata/status/888697140268204032).

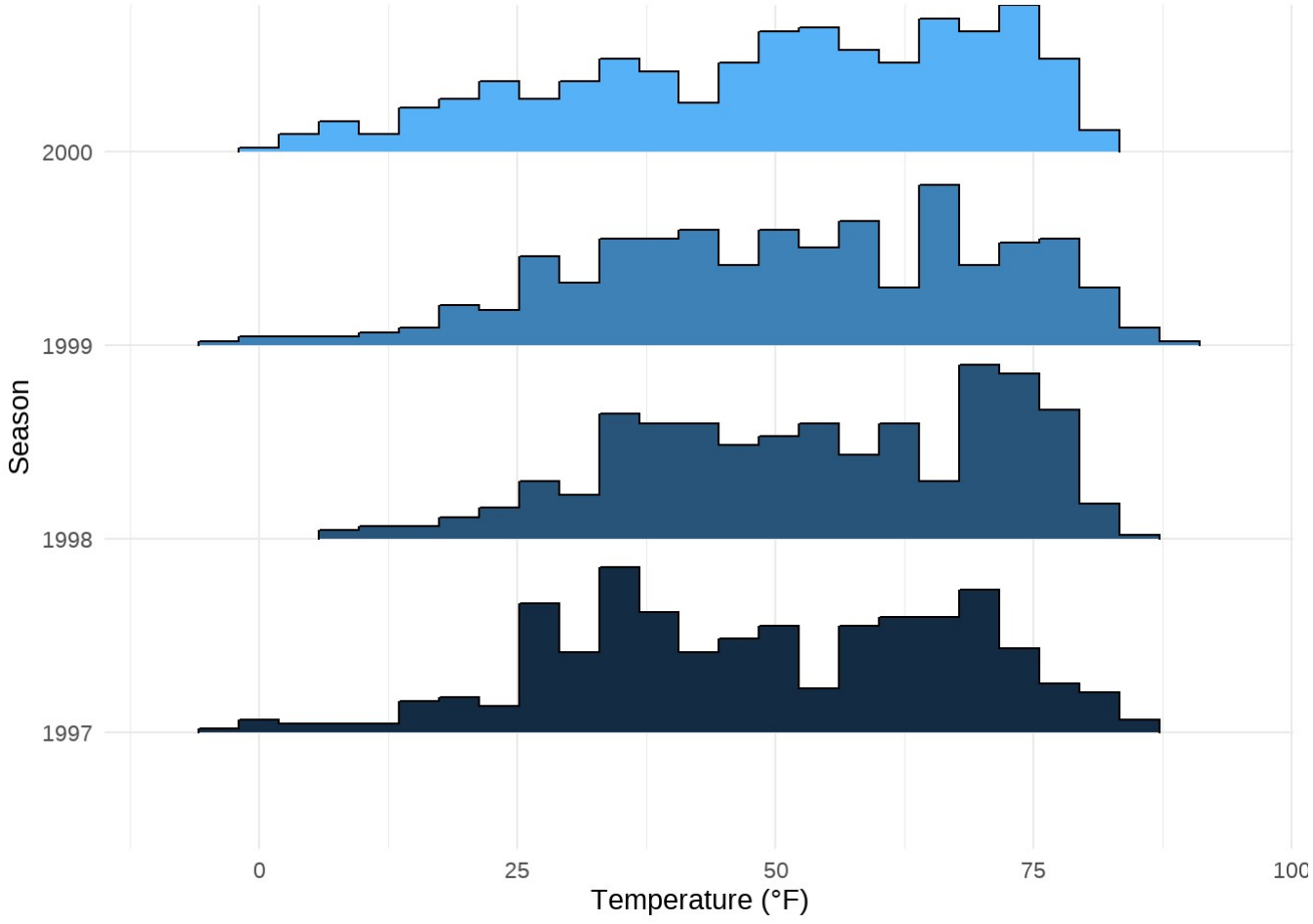
```
library(tidyverse)

## only plot extreme season using dplyr from the tidyverse
ggplot(data = 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")
```





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")
```

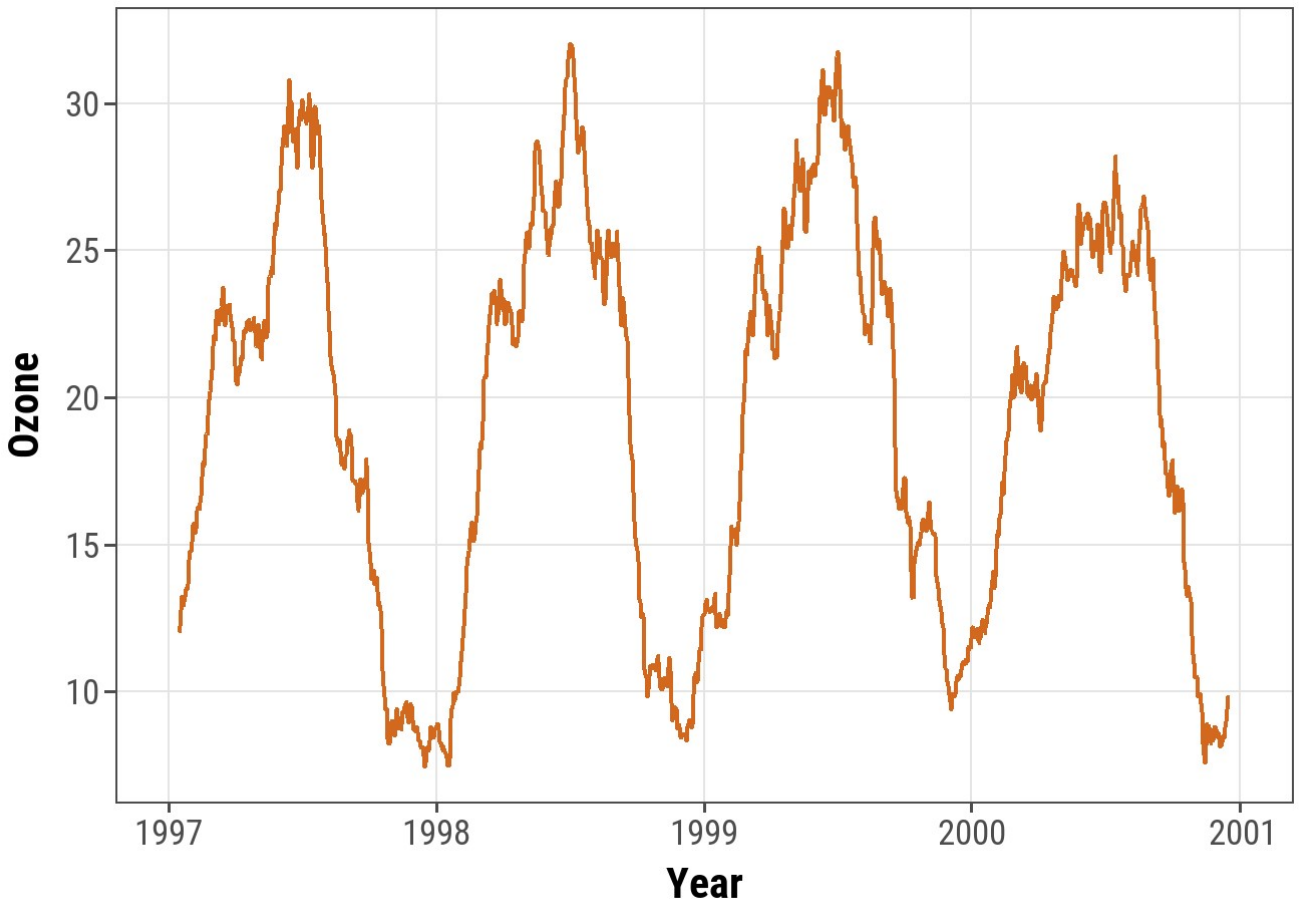


↑ Jump back to Table of Content.

WORKING WITH RIBBONS (AUC, CI, ETC.)

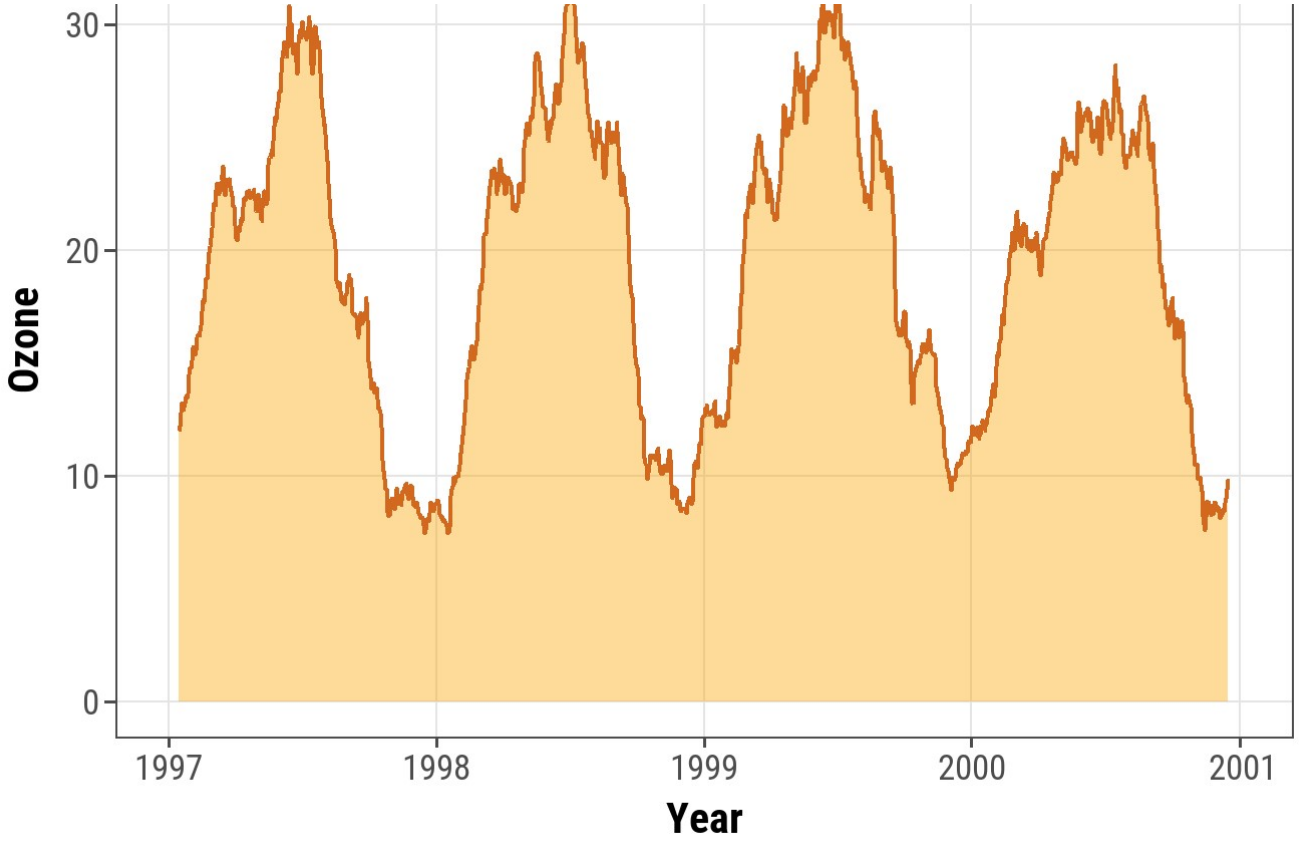
This is not a perfect dataset for demonstrating this, but using ribbon can be useful. In this example we will create a 30-day running average using the filter() function so that our ribbon is not too noisy.

```
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")
```



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")
```



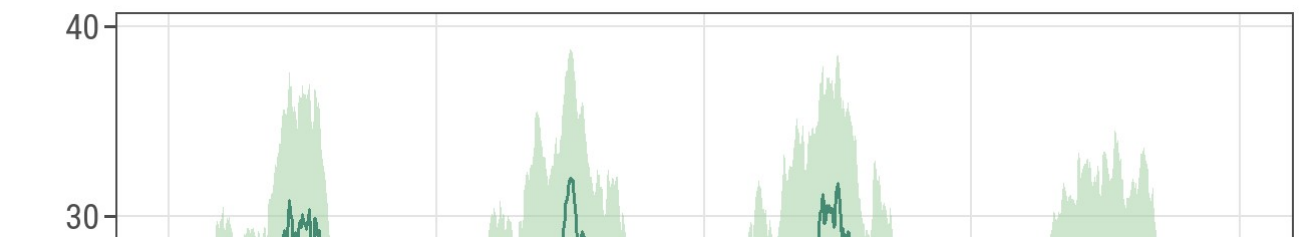
Nice to indicate the area under the curve (AUC) (https://en.wikipedia.org/wiki/Area_under_the_curve_(pharmacokinetics)) but this is not the conventional way to use `geom_ribbon()`.

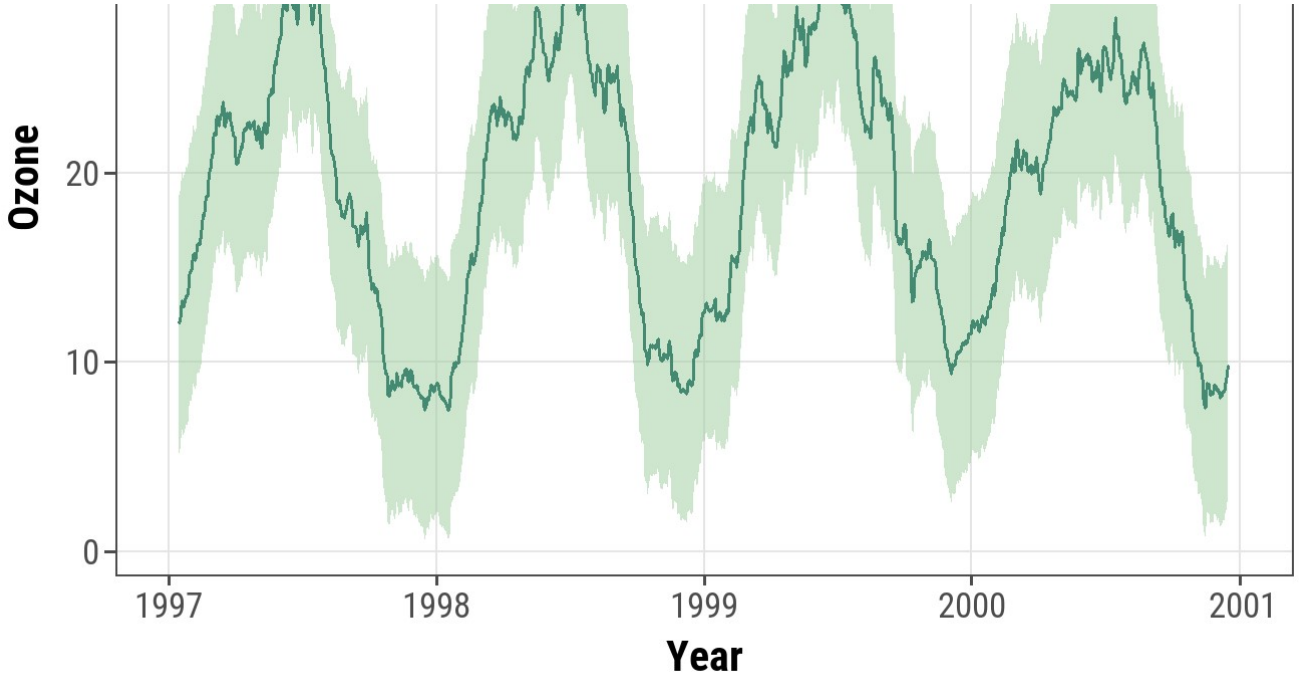
🤷 *And actually a nicer way to achieve the same is* `geom_area()`.

Expand to see example.

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")
```





↑ Jump back to Table of Content.

WORKING WITH SMOOTHINGS

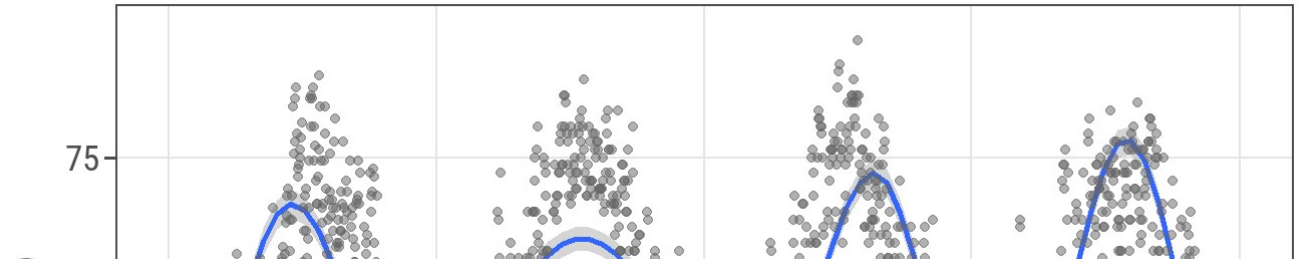
It is amazingly easy to add smoothing to your data using `{ggplot2}`.

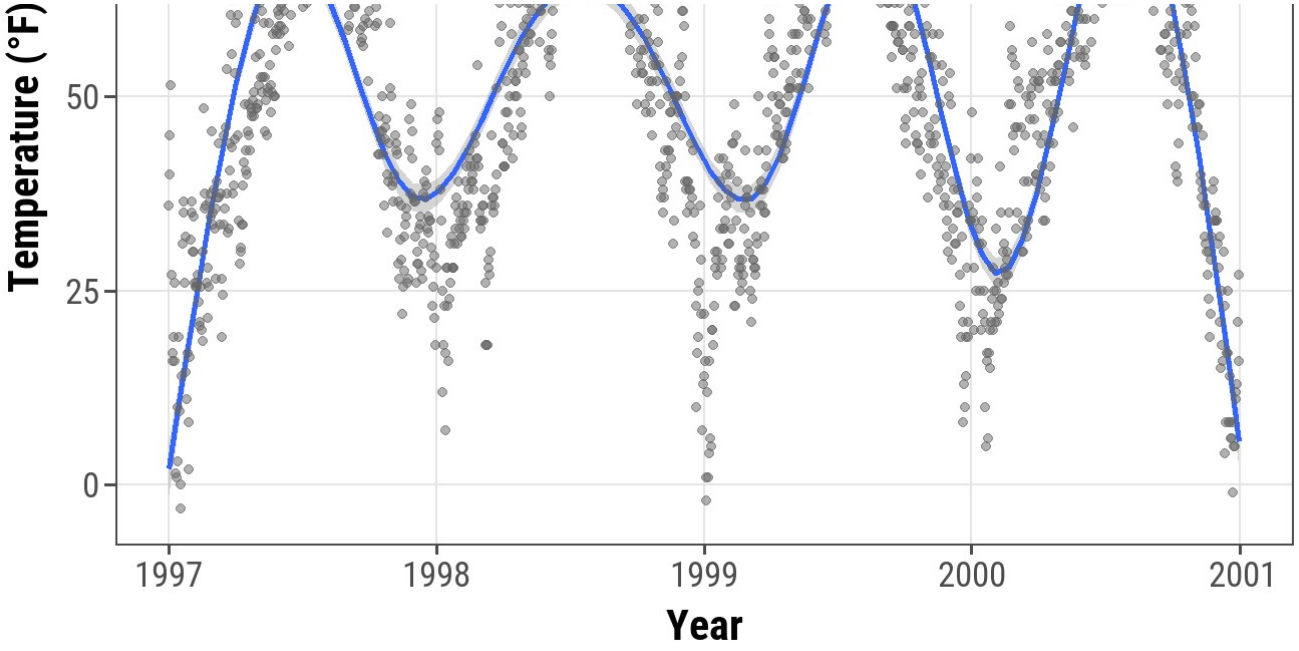
DEFAULT: ADDING A LOESS OR GAM SMOOTHING

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)) +
  labs(x = "Year", y = "Temperature (°F)") +
  stat_smooth() +
  geom_point(color = "gray40", alpha = .5)
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



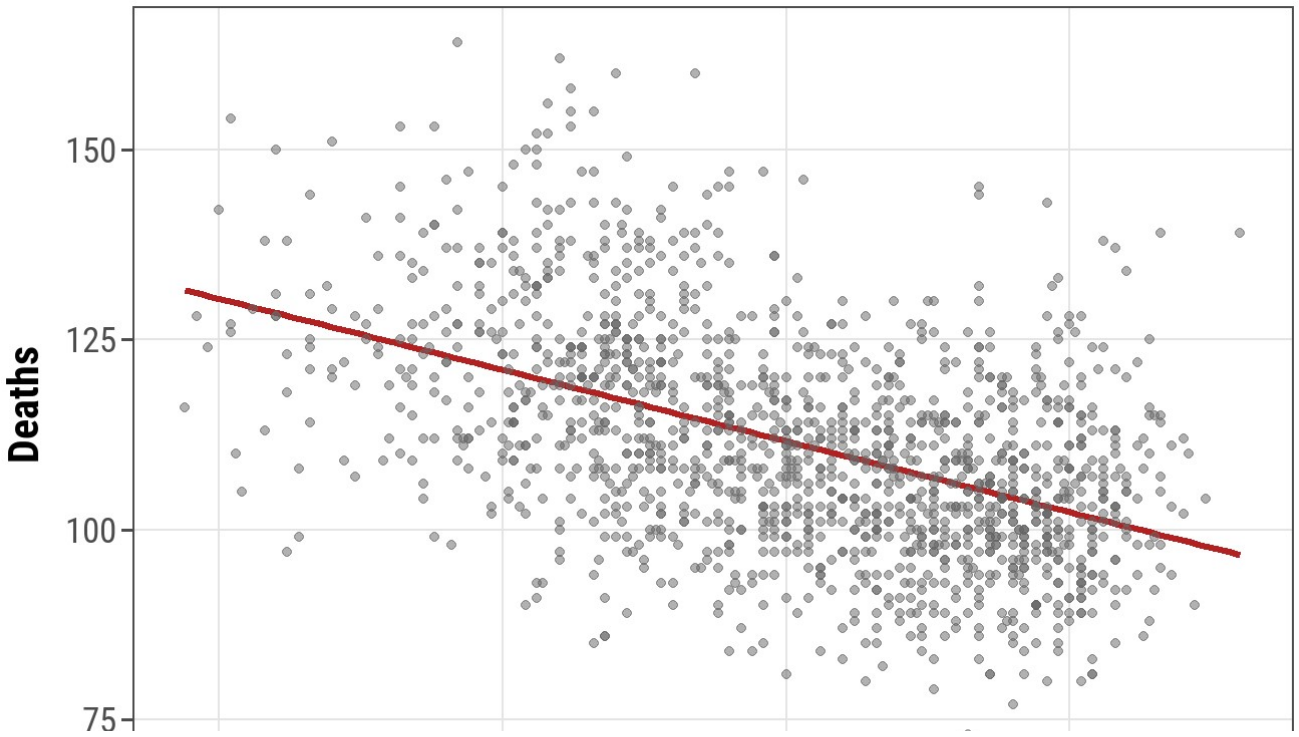


💡 **In most cases one wants the points to be on top of the ribbon so make sure you always call the smoothing before you add the points.**

ADDING A LINEAR FIT

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

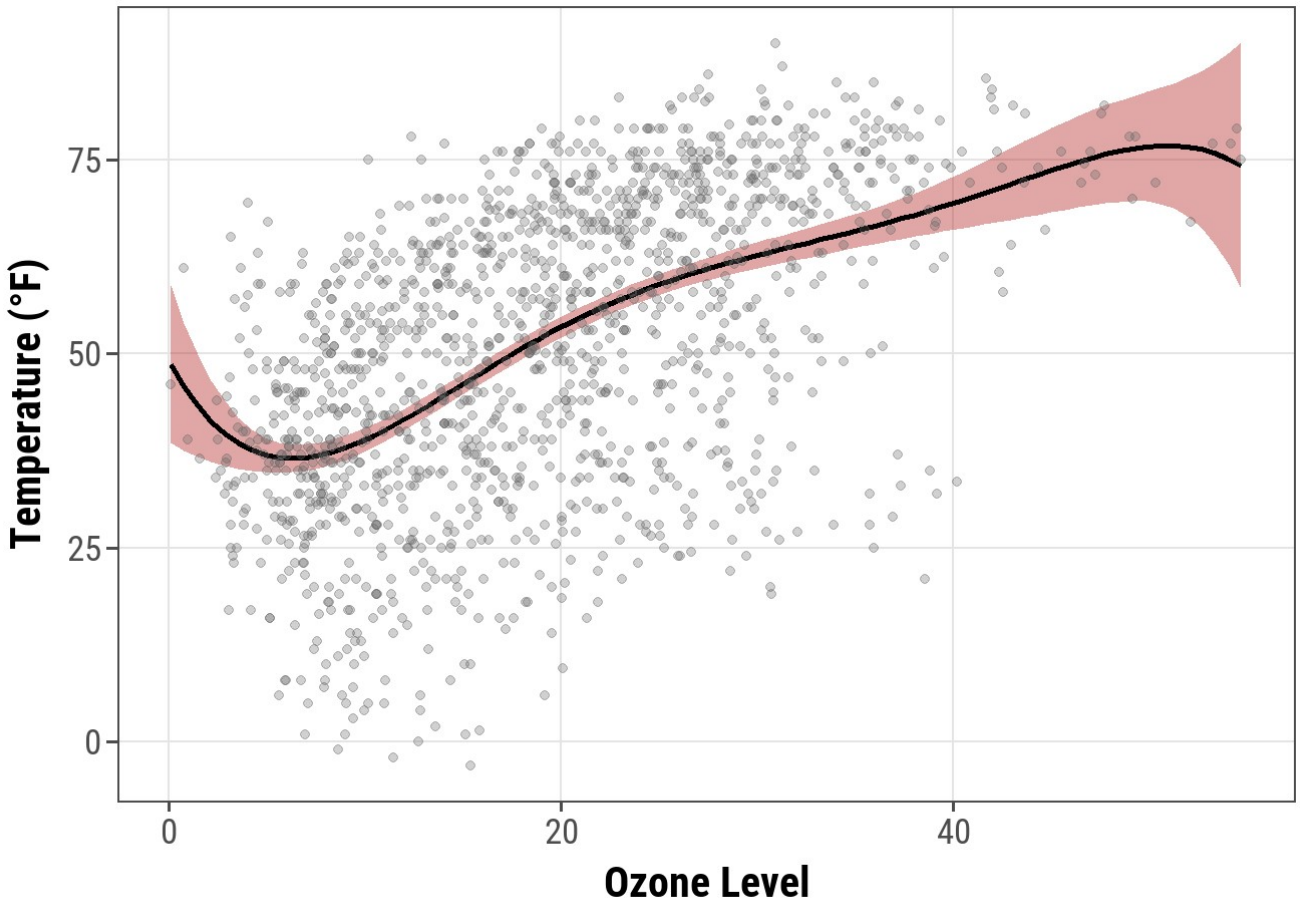
```
ggplot(chic, aes(x = temp, y = death)) +
   labs(x = "Temperature (°F)", y = "Deaths") +
   stat_smooth(method = "lm", se = FALSE,
               color = "firebrick", size = 1.3) +
   geom_point(color = "gray40", alpha = .5)
```



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)?

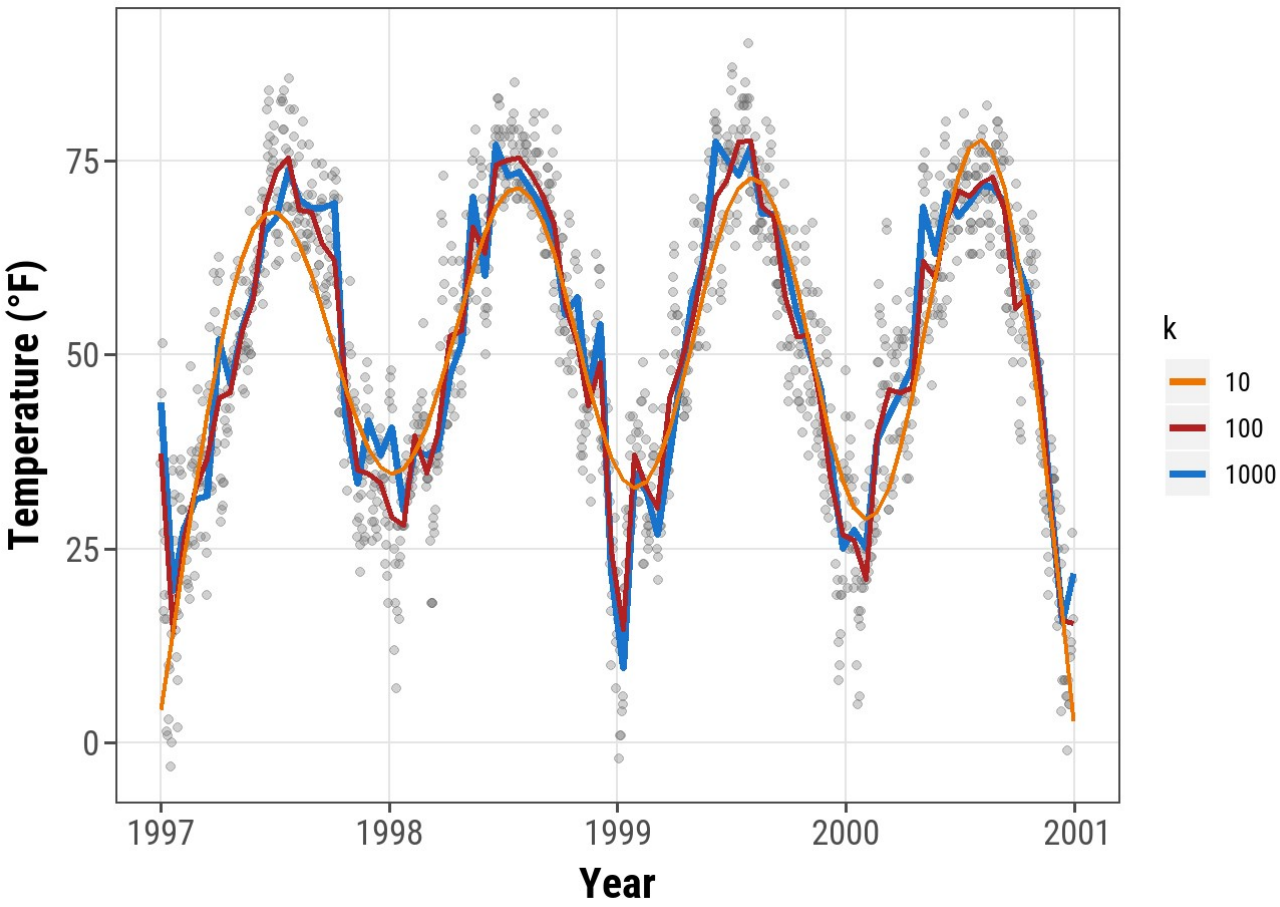
```
ggplot(chic, aes(x = o3, y = temp))+
  labs(x = "Ozone Level", y = "Temperature (°F)") +
  geom_smooth(
    method = "lm",
    formula = y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5),
    color = "black",
    fill = "firebrick"
  ) +
  geom_point(color = "gray40", alpha = .3)
```



🤷 *Huh, `geom_smooth()`? There is an important difference between `geom` and `stat` layers but here it really doesn't matter which one you use. Expand to compare both.*

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) +
  labs(x = "Year", y = "Temperature (°F)") +
  stat_smooth(aes(col = "1000"),
              method = "gam",
              formula = y ~ s(x, k = 1000),
              se = FALSE, size = 1.3) +
  stat_smooth(aes(col = "100"),
              method = "gam",
              formula = y ~ s(x, k = 100),
              se = FALSE, size = 1) +
  stat_smooth(aes(col = "10"),
              method = "gam",
              formula = y ~ s(x, k = 10),
              se = FALSE, size = .8) +
  scale_color_manual(name = "k", values = cols)
```



↑ Jump back to Table of Content.

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).

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")
```

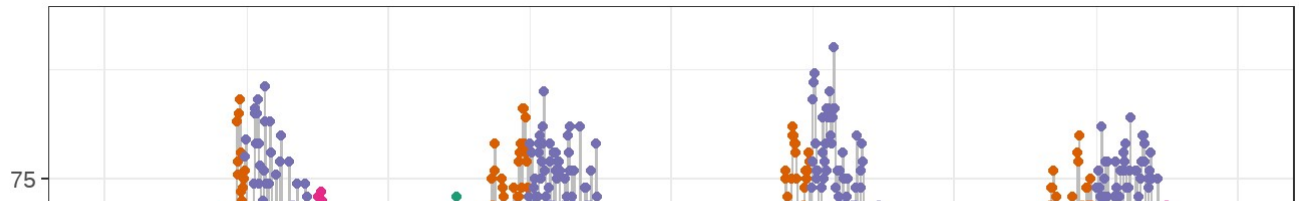
Of course, one can use ggplots in these apps. This example demonstrates the possibility to add some interactive user experience:

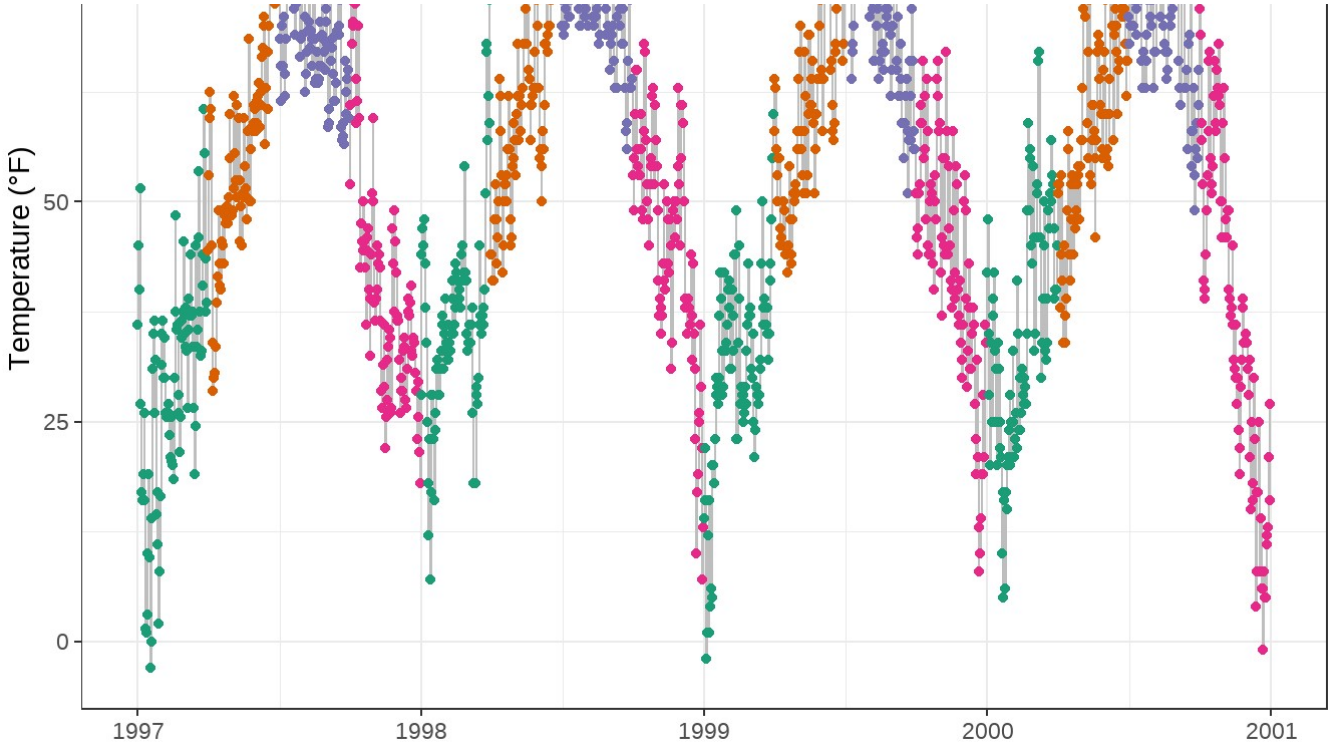
```
runExample("04_mpg")
```

PLOT.LY VIA {PLOTLY} AND {GGPLOT2}

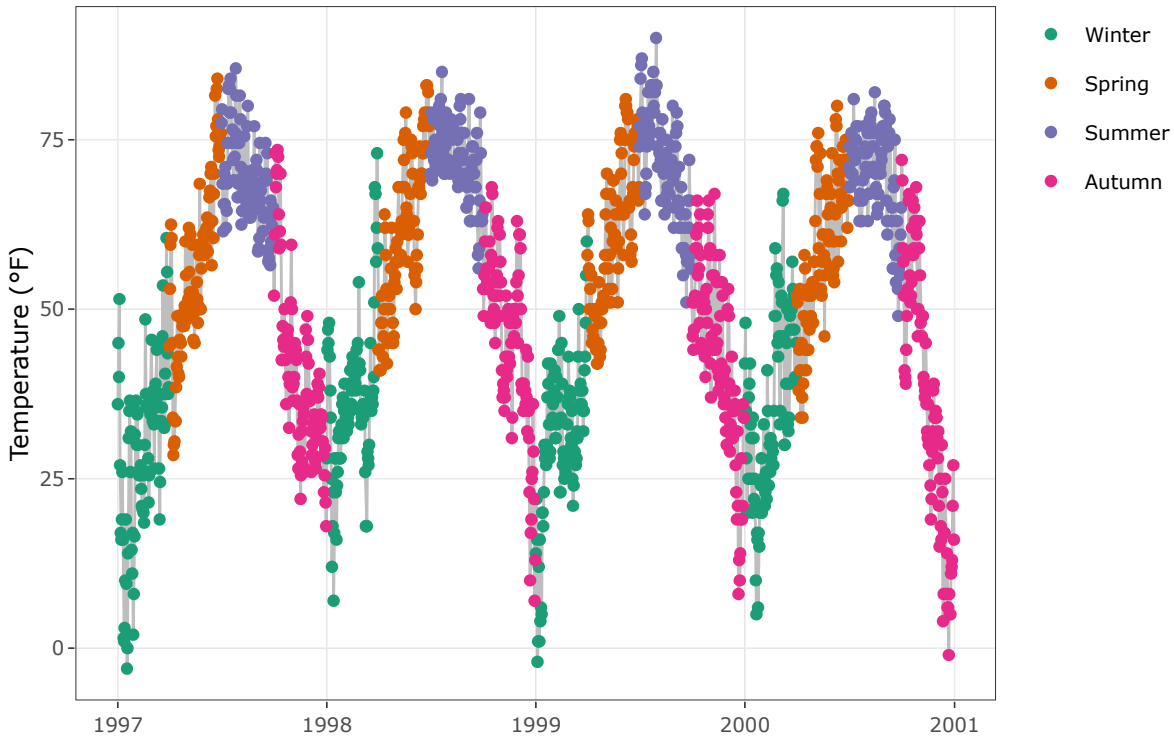
Plot.ly (https://chart-studio.plotly.com/feed/#/) is a tool for creating online, interactive graphics and web apps. The `{plotly}` package (https://plot.ly/r/getting-started/) enables you to create those directly from your `{ggplot2}` plots and the workflow is surprisingly easy and can be done from within 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()
```






```
library(plotly)

ggplotly(g)
```



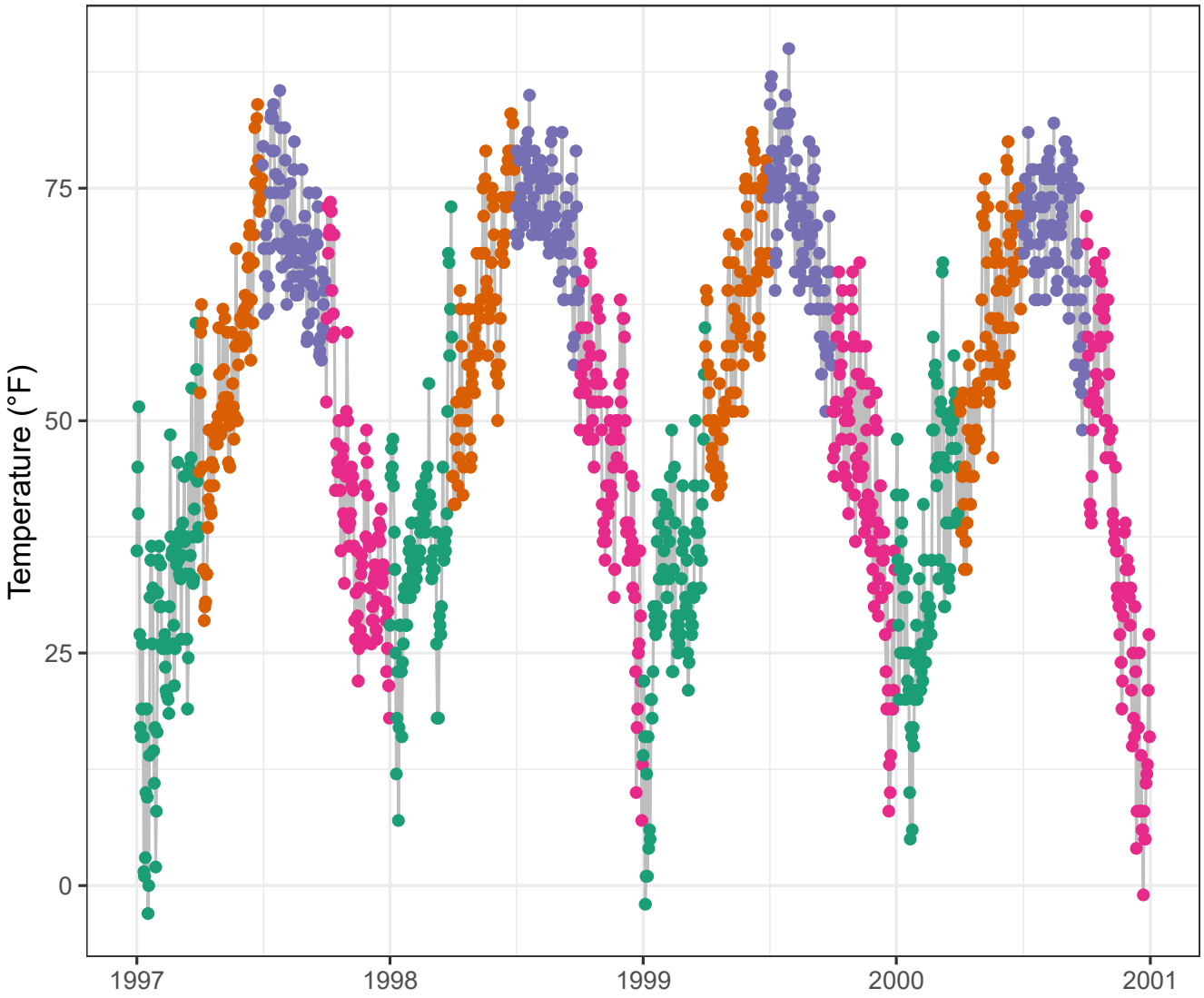
Here, for example, it keeps the overall theme setting but adds the legend again.

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)
```

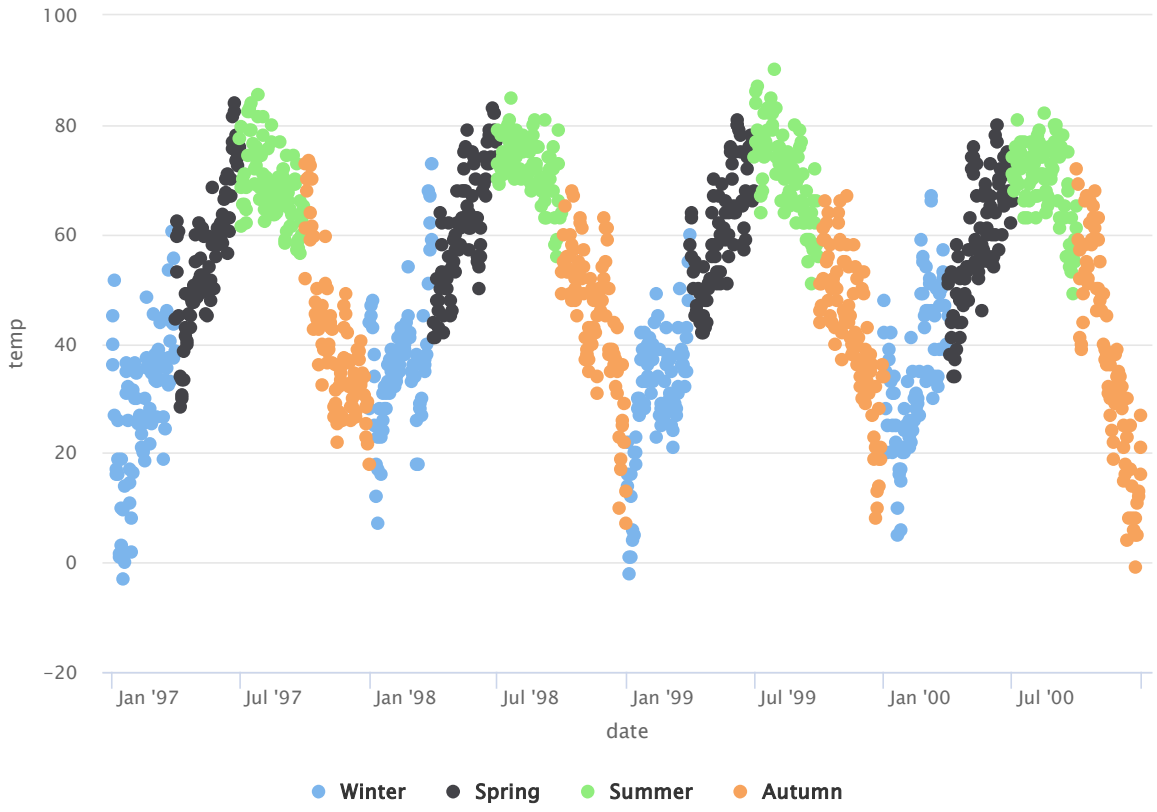


HIGHCHARTS VIA {HIGHCHARTER}

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)

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



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 these two apps (App 1 (https://johncoene.shinyapps.io/fopi-contest/) and App 2 (https://berlinbikes.correlaid.org/)) making use of the `{echarts4r}` functionality.

```
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)
```

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"))
```

(The example doesn't work in Rmarkdown.)

↑ Jump back to Table of Content.

REMARKS, TIPPS & RESOURCES

USING `ggplot()` 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, but in

`source()` or inside your own functions you will need an explicit `print()` statement, i.e. `print(g)` in most of our examples. See also the Q&A page of R (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))

↑ Jump back to Table of Content.

R Session Info

← PREVIOUS POST (/2019/05/17/THE-EVOLUTION-OF-A-GGPLOT-EP.-1/)

NEXT POST → (/2019/09/04/WORKSHOP-ANNOUNCEMENT-DATASCIENCE-DATAVIZ-TIDYVERSE/)

FEATURED TAGS (/TAGS/)

BERLIN (/TAGS/BERLIN) DATAVIZ (/TAGS/DATAVIZ) R (/TAGS/R) TIDYTUESDAY (/TAGS/TIDYTUESDAY) ANIMATIONS (/TAGS/ANIMATIONS) ANNOUNCEMENT (/TAGS/ANNOUNCEMENT)

GGPLOT2 (/TAGS/GGPLOT2) MAPS (/TAGS/MAPS) TIDYVERSE (/TAGS/TIDYVERSE) TUTORIAL (/TAGS/TUTORIAL) WEATHER (/TAGS/WEATHER)

FRIENDS

DataVizSociety (https://www.datavisualizationsociety.com/) R4DS Community (https://www.rfordatasci.com/)
CorrelAid (https://correlaid.org/en/) Will Chase (https://www.williamrchase.com/tags/dataviz/)
Georgios Karamanis (https://karaman.is/) Marco Sciaini (https://marcosci.github.io/)
Matthias Stahl (https://www.higsch.com/) Heureka Labs (https://www.heurekalabs.org/)

A HUGO (https://gohugo.io/) CleanWhite (https://themes.gohugo.io/hugo-theme-cleanwhite) page build with ♥ and powered by Netlify (https://www.netlify.com/) • Header images by Richard Strozynski (https://www.instagram.com/richard.strozynski/?hl=en)