Data Visualization with ggplot2 (Part 3)

Data Munging and Visualization Case Study (Chapter 4)

Seun Odeyemi

2019-04-24

Contents

Load Libraries

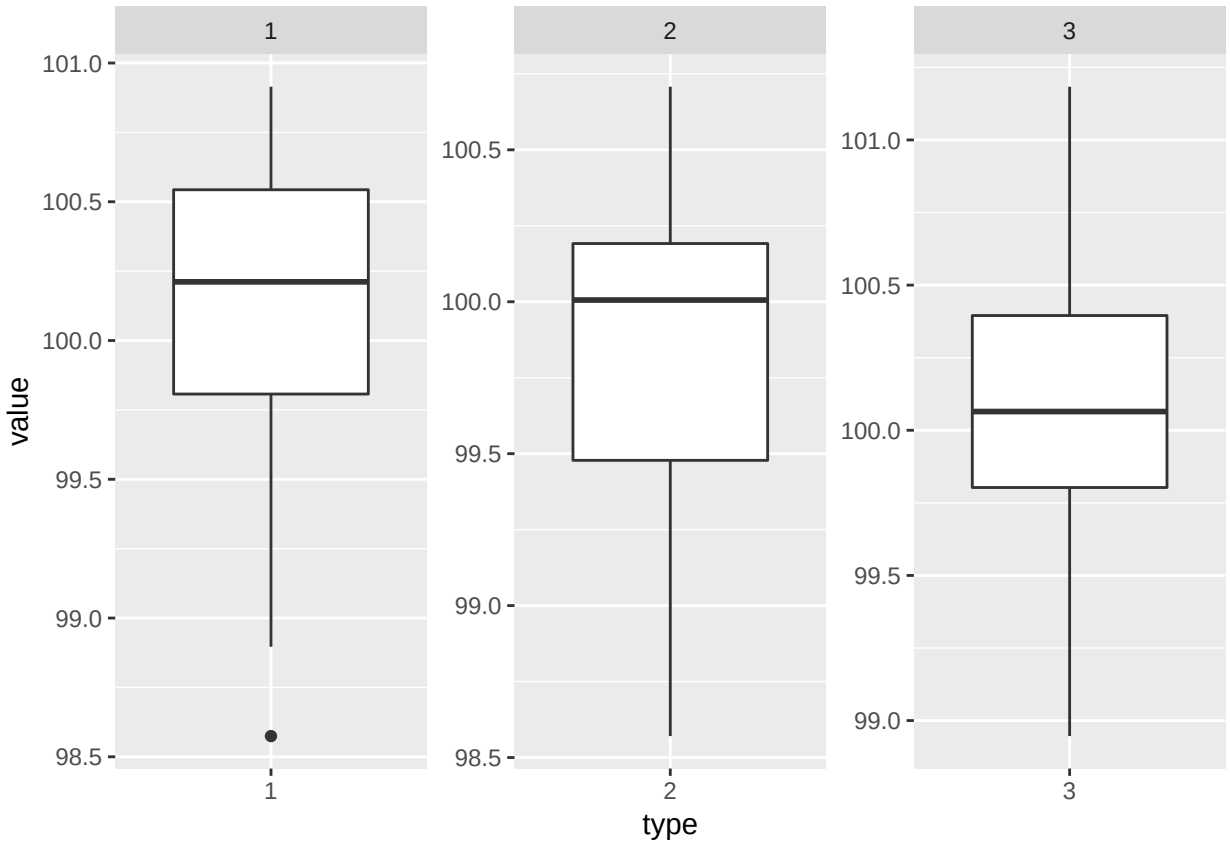
```
library(readr)
library(dplyr)
library(ggplot2)
# library(ggplot2movies)
library(tidyr)
library(skimr)
library(knitr)
library(kableExtra)
library(RColorBrewer)
library(grid)
library(ggthemes)
library(forcats)
library(GGally)
library(here)
library(hexbin)
```

Case Study I - Bag Plot

In this chapter, we're going to wrap up our three part series on `ggplot2` by learning to create and use our own layers. `ggplot(2.0)` introduced the ability to write your own extensions. This is one of the newest and most flexible features of the package and we'll see it in action in this chapter. This is a better version of what we did at the end of the second course. There we wrote a function to calculate and draw a mosaic plot. Extensions go one step further by allowing us to make our own `geom` or `stat` layer, which combines statistics and plotting. As an example we are going to create a **bag plot**. Bag plot was developed by John Tukey (the originator of the box plot). Actually, it is not really surprising since bag plots are basically 2D box plots. In the first chapter we saw 2D density plots, so here we can pick up the conversation and complete the picture with 2D box plots.

Imagine the case of having two continuous variables we want to compare. Recall that we may consider this a single continuous variable grouped to a second factor variable as shown below. If we just want two box plots then the solution is fairly straightforward. However, we don't know the relationship between any two individual data points. One solution will be to connect each data point in a **slope plot** style visualization, but there are some obvious drawbacks. For example, it is difficult to see the distribution of slopes, we'll have to make another plot with just that information and perhaps use those values to draw a box plot. This may be a good solution in some situations since we can see the distribution of the differences in each value not just two independent distributions. However, we can also imagine a situation where two groups are represented in two distinct variables. This will lend itself nicely to making a scatter plot or as we say earlier a 2D density plot.

```
# I want create a tibble with a continuous variable and a factor variable
# library(reprex)
library(tibble)
library(forcats)

df <- tibble(value = rnorm(99, 100, 0.5), type = rep(as_factor(c("1", "2", "3")), 33))

# df

ggplot(df, aes(x = type, value)) +
  geom_boxplot() +
  facet_wrap(~type, ncol = 3, scales = "free")
```



The bag plot is kind of a cross between these two types. We want to see the details of important points, but we also want to know something about the distribution. There are three concentric rings around the innermost point, which is an approximation of the 2D median. The first ring is the **hull**, which outlines the center region (not that informative). The second ring is the **bag**, which contains 50% of the data points (this is similar to the IQR in the 1D plot). The third ring is the **loop**, which contains all points inside the fence, calculated as some factor enlargement of the bag (default = 3). All points outside the loop are drawn as individual points like the extreme value in a box plot. To obtain these values, we'll the `compute.bagplot` option in the `aplpack` package. This provides a named list containing vectors, matrices, and the dataframe of the coordinates of the individual data points. Corresponding to the polygon for the bag and loop.

In the exercises we'll discover how to use these functions as a basis for a new stats layer. So, instead of using base package we can call on all the advantages of `ggplot2` to produce not only scatter plots but also bag plots, which will be just as flexible for any data set we choose to use. Although bag plots are a pretty useful tool they are not that popular partly because they are poorly understood, but so are box plots and people use them all the time. There are also trends as to what is considered a cool plot type so it maybe that bag plots become a hit someday. Perhaps having a `ggplot2` geom will facilate that.

My point here is not that you need to use bag plots instead of scatter plots. There are two quick points I want to make:

1. I want to introduce you to a new plotting type.
2. I want you to understand how to use ggplot2 extensions to create any kind of plot you may need for a specific data visualization solution.

We'll return to this in the case studies section later.

Base Package - Bag Plot

Before you create your own `stats` layer, you'll begin by understanding what a bag plot is, and how to get the data for your own plots.For this you'll use a fake dataset called `test_data`, which only contains two variables. A scatter plot is shown in the viewer.The `aplpack` package, which contains the `bagplot()` and `compute.bagplot()` functions, has been loaded for you.

```
library(aplpack)

test_data <- ch5_test_data
#> Error in eval(expr, envir, enclos): object 'ch5_test_data' not found

# Call bagplot() on test_data
bagplot(test_data)
#> Error in is.data.frame(x): object 'test_data' not found

# Call compute.bagplot on test_data, assign to bag
bag <- compute.bagplot(test_data)
#> Error in compute.bagplot(test_data): object 'test_data' not found

# Display information
bag$hull.loop
#> Error in eval(expr, envir, enclos): object 'bag' not found
bag$hull.bag
#> Error in eval(expr, envir, enclos): object 'bag' not found
bag$pxy.outlier
#> Error in eval(expr, envir, enclos): object 'bag' not found

# Highlight components
points(bag$hull.loop, col = "green", pch = 16)
#> Error in points(bag$hull.loop, col = "green", pch = 16): object 'bag' not found
points(bag$hull.bag, col = "orange", pch = 16)
#> Error in points(bag$hull.bag, col = "orange", pch = 16): object 'bag' not found
points(bag$pxy.outlier, col = "purple", pch = 16)
#> Error in points(bag$pxy.outlier, col = "purple", pch = 16): object 'bag' not found
```

Great! Bag plots are a great visualisation option, but not common, so use with caution!

Multilayer ggplot2 Bag Plot

With our current understanding, if we wanted to make a bag plot in `ggplot2`, we'd take the three data frames (for the loop, bag and outliers) and add them using three separate geom layers. Let's see how this simple solution works and in the next exercises you'll expand on this topic to make a real `stats` layer. The bag and `test_data` objects from the previous exercise are provided. test_data contains two variables: x and y.

```
# bag and test_data are available

# Create data frames from matrices
hull.loop <- data.frame(x = bag$hull.loop[,1], y = bag$hull.loop[,2])
#> Error in data.frame(x = bag$hull.loop[, 1], y = bag$hull.loop[, 2]): object 'bag' not found
hull.bag <- data.frame(x = bag$hull.bag[,1], y = bag$hull.bag[,2])
#> Error in data.frame(x = bag$hull.bag[, 1], y = bag$hull.bag[, 2]): object 'bag' not found
pxy.outlier <- data.frame(x = bag$pxy.outlier[,1], y = bag$pxy.outlier[,2])
#> Error in data.frame(x = bag$pxy.outlier[, 1], y = bag$pxy.outlier[, 2]): object 'bag' not found

# Finish the ggplot command
```

```
ggplot(test_data, aes(x = x,  y = y)) +
  geom_polygon(data = hull.loop, fill = "green") +
  geom_polygon(data = hull.bag, fill = "orange") +
  geom_point(data = pxy.outlier, col = "purple", pch = 16, cex = 1.5)
#> Error in ggplot(test_data, aes(x = x, y = y)): object 'test_data' not found
```

Looking good! Not the most efficient plot - but it's a good start!

Creating ggproto Functions

Now that you know where to find the statistics and how to use them in `ggplot2`, let's put them into the functions that will make them easier to use.

For this you'll use the `ggproto` object-oriented programming system - the basis of creating a new layer in `ggplot2`. There are four arguments for a `ggproto` object. The first two arguments are its name and what it inherits from (in this case `Stat`). Next come the required aesthetics, and then, most importantly, what the stat should do. For each *group* of data it receives from the data layer, what should be computed? This will simply be the calculations you performed in the previous exercise.

The `ggproto` object definition of `StatLoop` is already provided. Can you finish the implementations for the other ones?

```
# Watch out: you have to use generic names now: use data$x instead of test_data$x!
data <- test_data
#> Error in eval(expr, envir, enclos): object 'test_data' not found

# ggproto for StatLoop (hull.loop)
StatLoop <- ggproto("StatLoop", Stat,
                    required_aes = c("x", "y"),
                    compute_group = function(data, scales) {
                      bag <- compute.bagplot(x = data$x, y = data$y)
                      data.frame(x = bag$hull.loop[,1], y = bag$hull.loop[,2])
                    })

# ggproto for StatBag (hull.bag)
StatBag <- ggproto("StatBag", Stat,
                   required_aes = c("x", "y"),
                   compute_group = function(data, scales) {
                     bag <- compute.bagplot(x = data$x, y = data$y)
                     data.frame(x = bag$hull.bag[,1], y = bag$hull.bag[,2])
                   })

# ggproto for StatOut (pxy.outlier)
StatOut <- ggproto("StatOut", Stat,
                   required_aes = c("x", "y"),
                   compute_group = function(data, scales) {
                     bag <- compute.bagplot(x = data$x, y = data$y)
                     data.frame(x = bag$pxy.outlier[,1], y = bag$pxy.outlier[,2])
                   })
```

I like my code like I like my martinis - DRY: Don't Repeat Yourself! We're slowly getting there.

Creating stat_bag()

In the previous exercise you established three `ggproto` objects, now you need to combine them under a new `ggplot2` function that you'll call `stat_bag()`. Adding a `stat_bag()` layer will execute each of the three

ggproto objects that you just created.Your three objects are called StatLoop, StatBag, StatOut, so you'll need three layers in your stat_bag() function, which you'll make with the layer() function. When you have multiple layers, you can combine them in a list by simply calling list().

For each layer, you'll also need to specify the approrpiate geom: "polygon" or "point". The framework for the stat_bag() layer function has been provided for you.

```
# StatLoop, StatBag and StatOut are available

# Combine ggproto objects in layers to build stat_bag()
stat_bag <- function(mapping = NULL, data = NULL, geom = "polygon",
                     position = "identity", na.rm = FALSE, show.legend = NA,
                     inherit.aes = TRUE, loop = FALSE, ...) {
  list(
    # StatLoop layer
    layer(
      stat = StatLoop, data = data, mapping = mapping, geom = geom,
      position = position, show.legend = show.legend, inherit.aes = inherit.aes,
      params = list(na.rm = na.rm, alpha = 0.35, col = NA, ...)
    ),
    # StatBag layer
    layer(
      stat = StatBag, data = data, mapping = mapping, geom = geom,
      position = position, show.legend = show.legend, inherit.aes = inherit.aes,
      params = list(na.rm = na.rm, alpha = 0.35, col = NA, ...)
    ),
    # StatOut layer
    layer(
      stat = StatOut, data = data, mapping = mapping, geom = "point",
      position = position, show.legend = show.legend, inherit.aes = inherit.aes,
      params = list(na.rm = na.rm, alpha = 0.7, col = NA, shape = 21, ...)
    )
  )
}
```

Well done! Let's start using our new stat_bag!

Use stat_bag()

So far you've seen the basics for creating a new ggplot layer. It's bare-bones, but functional. You now have a working solution to the bag plot question. The ggplot2 command that you've coded before is available. now, let's use stat_bag() to make our plot!

```
# hull.loop, hull.bag and pxy.outlier are available
# stat_bag, test_data and test_data2 are available

# Previous method
ggplot(test_data, aes(x = x,  y = y)) +
  geom_polygon(data = hull.loop, fill = "green") +
  geom_polygon(data = hull.bag, fill = "orange") +
  geom_point(data = pxy.outlier, col = "purple", pch = 16, cex = 1.5)
#> Error in ggplot(test_data, aes(x = x, y = y)): object 'test_data' not found

# stat_bag
ggplot(test_data, aes(x = x, y = y)) +
  stat_bag(fill = 'black')
```

```
#> Error in ggplot(test_data, aes(x = x, y = y)): object 'test_data' not found

# stat_bag on test_data2
(test_data2 <- ch5_test_data2)
#> Error in eval(expr, envir, enclos): object 'ch5_test_data2' not found
ggplot(test_data2, aes(x = x, y = y, fill = treatment)) +
  stat_bag() +
  theme(legend.position = "bottom")# +
#> Error in ggplot(test_data2, aes(x = x, y = y, fill = treatment)): object 'test_data2' not found
  #theme_tufte()
```

This is awesome! That's a DRY plot!

Case Study II - Weather (Part 1)

In this case study we're going to see concepts and techniques you've encounter over the three `ggplot2` courses in action. In particular, **I want to drive home the point that as part of our analytical toolkit, the underlying research question dictates the data visualization we'll develop**. The data set is something we've all seen a thousand times: the Weather. Our case study will go through the process of developing a data visualization that was developed by Edward Tufte for the New York Times. The data visualization emphasized highs and lows in a given year.

Our dataframe contains the average temperature for New York City for the first half of 2016. This a pretty limited data since all that we can really do here is plot the temperature as time series. We should always think about what kind of comparisons we want to make. For example, we can look at geography comparing cities in different regions. Or, we can choose to compare the current year to historical record, which is what we are going to do. For that we have another dataframe which contains records from 1995 to 2015. If we plot each year as a separate line, we can start to see some trends. There is an expected rise in the middle of each year to correspond to the summer months and we have a range of values for any given day. A typical comparison will be to plot the current as a trend on top of the series. This kind of works, but just asking how 2016 compares to the historical record doesn't really offer much insight since anyways there is a lot of variability in the data set.

At this point we can start refining our research question. Tufte decided to ask: how many days in the current year set a new daily record for highs and lows? Although, we can see days in which 2016 was clearly hotter than the record it is still difficult to see any clear result. The first step was to realize that although we have a time series, we don't have to use a line. Actually, we want to know something about the historical range of values for each given day. In that case we are going to a `linerange` `ggplot2` `geom` that should gives a 95% CI for the historical record in dark brown and all the raw values in light brown. Both the color and geom adjustments allow us to see the current year compared to the record more clearly.

To really emphasize the positions where 2016 exceeds the historical record. We can create another dataset before plotting, which identifies the days of interest and use a point geom to highlight them as we will see shortly. Our custom visualization requires a custom legend, which we saw how to make by taking advantage of grid graphics functions we learnt in the previous chapter. With this plot we start to blur the lines between exploratory and explanatory plots. We see how data visualization is really a part of exploratory data analysis and more than just looking at distributions. This plot is pretty informative, but it can use some cleaning up. In the exercises we'll go through the steps of developing and cleaning up the theme of this plot.

Step 1: Read in Data and Examine

Before you can begin with your visualization you need to obtain and clean up the data. The data you're using comes from the University of Dayton.

In this exercise you'll read in our data and clean it up. To start, you'll focus on the weather data from New York. It is available as a fixed-width format file called NYNEWYOR.txt.

Since leap years mean that dates don't line up perfectly, you're just going to remove all occurrences of February 29. In addition, you're going to split your data in two pieces: one for the historical record, and one for 2016. You'll see how to avoid doing this later on.

```
# Import weather data
weather <- read.fwf("datasets/NYNEWYOR.txt",
                    header = FALSE,
                    col.names = c("month", "day", "year", "temp"),
                    widths = c(14, 14, 13, 4))

# Check structure of weather
str(weather)
#> 'data.frame':    7824 obs. of  4 variables:
#>  $ month: num  1 1 1 1 1 1 1 1 1 1 ...
#>  $ day  : num  1 2 3 4 5 6 7 8 9 10 ...
#>  $ year : num  1995 1995 1995 1995 1995 ...
#>  $ temp : num  44 41 28 31 21 27 42 35 34 29 ...

# Create past with two filter() calls
# past <- weather %>%
#   filter(!(month == 2 & day == 29)) %>%
#   filter(year != as.character(2016))

past <- weather %>%
  filter(!(month == 2 & day == 29)) %>%
  filter(year != max(year))

# Check structure of past
str(past)
#> 'data.frame':    7665 obs. of  4 variables:
#>  $ month: num  1 1 1 1 1 1 1 1 1 1 ...
#>  $ day  : num  1 2 3 4 5 6 7 8 9 10 ...
#>  $ year : num  1995 1995 1995 1995 1995 ...
#>  $ temp : num  44 41 28 31 21 27 42 35 34 29 ...
```

Good work! We could already think about making our script more dynamic by using `lubridate::year(Sys.Date())` to extract the current year, but since our dataset ends in 2016, we can just take the max value in the year column.

Step 2: Summarize History

Using the past data frame, you'll calculate the 95% CI for the temperatures on each date. To do this, you'll assign a unique ID, called yearday to each date. This way, you can group according to yearday, then calculate aggregrate statistics on the correct values, over all years. You could group according to month and year, but yearday will make plotting on the x axis easier later on.

The past data frame that you've created before is already available in your workspace.

Note: You'll use a bunch of dplyr calls here; if you want to refresh your memory on them, you can have a look at our course on dplyr.

```
# Create new version of past
past_summ <- past %>%
  group_by(year) %>%
  mutate(yearday = 1:length(day)) %>%
  ungroup() %>%
  filter(temp != -99) %>%
```

```
  group_by(yearday) %>%
  mutate(max = max(temp),
         min = min(temp),
         avg = mean(temp),
         CI_lower = Hmisc::smean.cl.normal(temp)[2],
         CI_upper = Hmisc::smean.cl.normal(temp)[3]) %>%
  ungroup()

# Structure of past_summ
str(past_summ)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    7645 obs. of  10 variables:
#>  $ month   : num  1 1 1 1 1 1 1 1 1 1 ...
#>  $ day     : num  1 2 3 4 5 6 7 8 9 10 ...
#>  $ year    : num  1995 1995 1995 1995 1995 ...
#>  $ temp    : num  44 41 28 31 21 27 42 35 34 29 ...
#>  $ yearday : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ max     : num  51 48 57 55 56 62 52 57 54 47 ...
#>  $ min     : num  17 15 16 15 21 14 14 12 21 8.5 ...
#>  $ avg     : num  35.6 35.4 34.9 35.1 35.9 ...
#>  $ CI_lower: num  31 31.6 29.7 29.9 31.9 ...
#>  $ CI_upper: num  40.1 39.2 40 40.4 39.9 ...
```

Great job! Time to make a first plot!

Step 3: Plot History

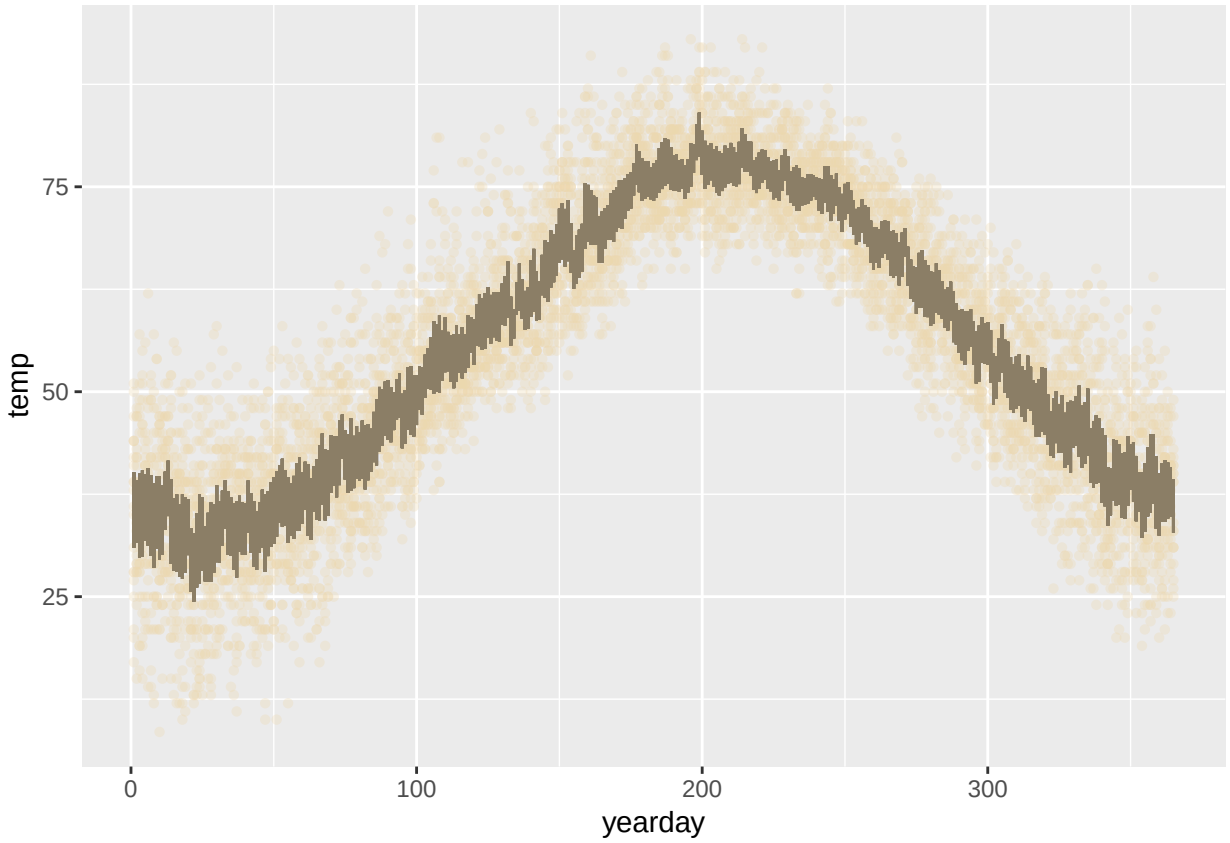
Now you're ready to plot the historical record. There are couple things you can do here. The typical thing, but also pretty boring, would be to plot the average as a solid line:

ggplot(past, aes(x = yearday, y = avg)) + geom_line()

You can do better than that by plotting a line range for the historical min-max and also the 95% CI. Very coincidentally, that's exactly the data that you've prepared in the previous exercise! The `ggplot()` command in the editor is a first attempt.

However, since the range of values at the extremes is so sparse, it would be more accurate to represent this using points. Can you update the plot accordingly?

```
# Adapt historical plot
ggplot(past_summ, aes(x = yearday, y = temp)) +
  geom_point(col = "#EED8AE", alpha = 0.3, shape = 16) +
  geom_linerange(aes(ymin = CI_lower, ymax = CI_upper), col = "#8B7E66")
```

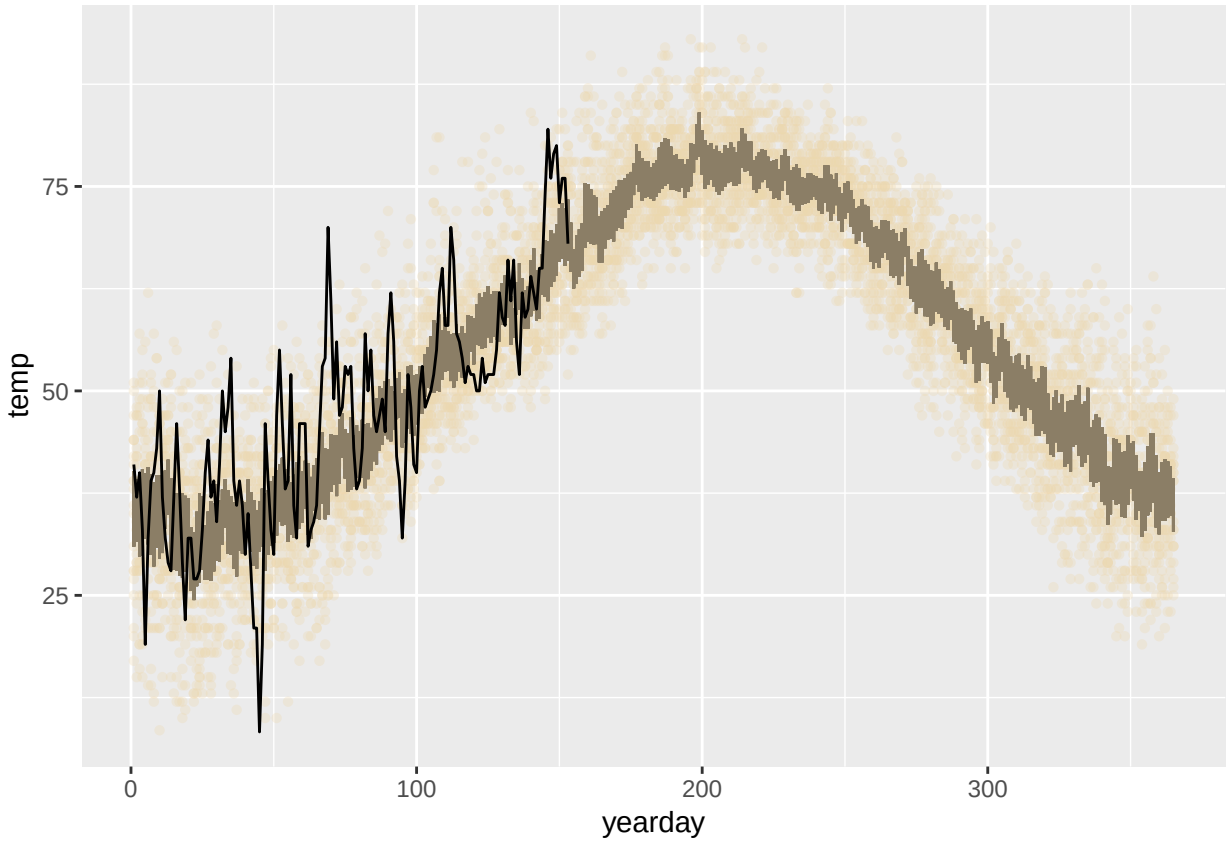


Looks awesome! This is going to get even more detailed.

Step 4: Plot Present

Now that you have an idea of the historical record, you'll want to compare it to the temperature measurements of the current year. `dplyr` code similar to the one you've coded up to create the past data frame has been provided. Up to you to add another layer to the `ggplot()` command from the previous exercise!

```
# weather and past are available in your workspace

# Create present
present <- weather %>%
  filter(!(month == 2 & day == 29)) %>%
  filter(year == max(year)) %>%
  group_by(year) %>%
  mutate(yearday = 1:length(day)) %>%
  ungroup() %>%
  filter(temp != -99)

# Add geom_line to ggplot command
ggplot(past_summ, aes(x = yearday, y = temp)) +
  geom_point(col = "#EED8AE", alpha = 0.3, shape = 16) +
  geom_linerange(aes(ymin = CI_lower, ymax = CI_upper), col = "#8B7E66") +
  geom_line(data = present)
```



Great job! We're slowly creating a unique and informative plot.

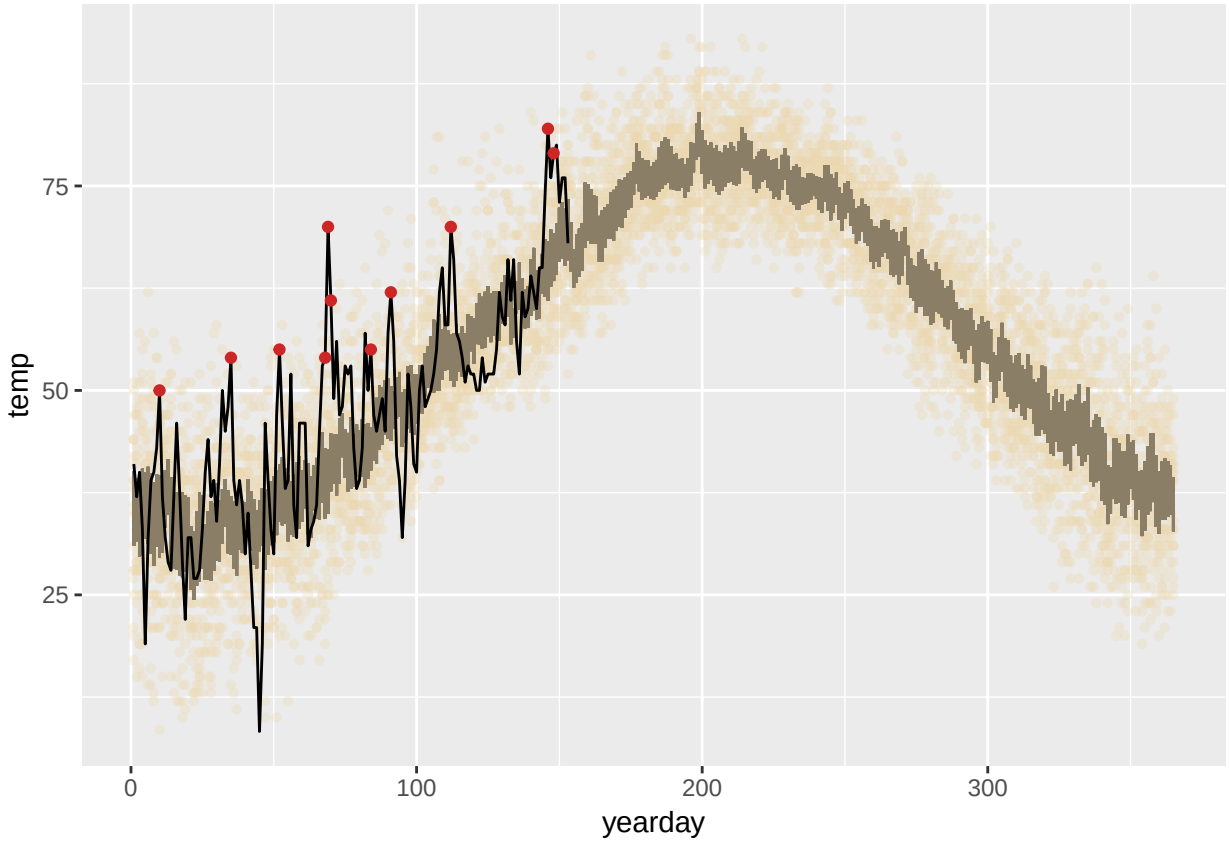
Step 5: Find New Record Highs

You're going to look at a couple ways of detecting interesting features in your dataset. In this case, the interesting features are going to be which dates in the current year exceed the historical record (either a new record high or low for each yearday).

The first method, which you'll use here and in the next exercise, works fine, but you'll explore how to make it more efficient in later exercises.

```
# Create past_highs
past_highs <- past_summ %>%
  group_by(yearday) %>%
  summarise(past_high = max(temp))

# Create record_high
record_high <- present %>%
  left_join(past_highs) %>%
  filter(temp > past_high)
#> Joining, by = "yearday"

# Add record_high information to plot
ggplot(past_summ, aes(x = yearday, y = temp)) +
  geom_point(col = "#EED8AE", alpha = 0.3, shape = 16) +
  geom_linerange(aes(ymin = CI_lower, ymax = CI_upper), col = "#8B7E66") +
  geom_line(data = present) +
```

```
  geom_point(data = record_high, col = "#CD2626")
```



Great! There are quite some record highs in here!

Efficiently Calculate Record Highs and Lows

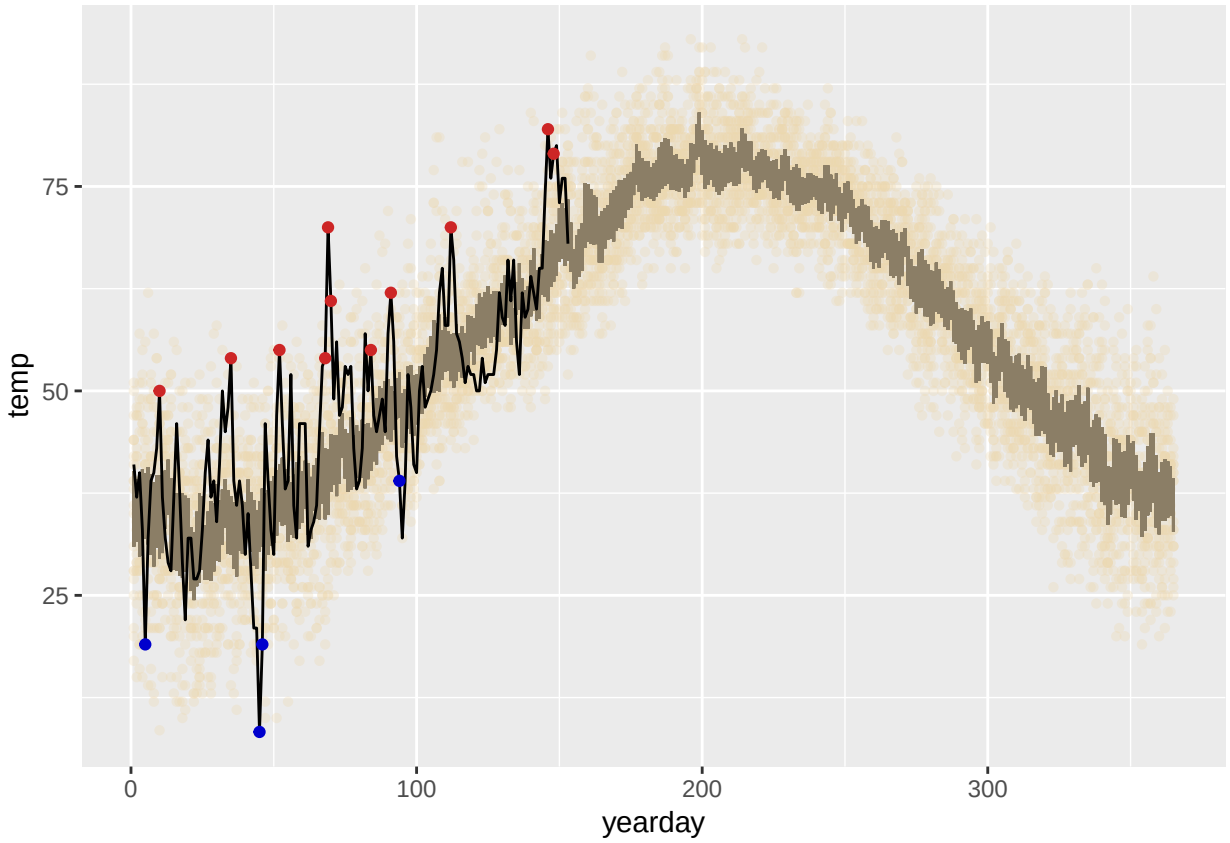
To also add the record lows to the plot, you could do the same things as in the previous exercise: create a data frame `past_lows`, join it with `present`, figure out the record lows, and add yet another layer, with a blueish color.

This is not really the `ggplot2` way to do things. Instead of adding two layers, and manually assigning a color, you can do something else: you can map a variable denoting a record high or low onto the color aesthetic!

Here you'll combine the previous two exercises to identify the record highs and lows in one step, assign them to a new data frame called extremes, and use this to map a color aesthetic. This will make both your data munging and plotting code more efficient.

```
# Create past_extremes
past_extremes <- past_summ %>%
  group_by(yearday) %>%
  summarise(past_low = min(temp),
            past_high = max(temp))

# Create record_high_low
record_high_low <- present %>%
  left_join(past_extremes) %>%
  mutate(record = ifelse(temp < past_low,
                         "#0000CD",
```

```
                        ifelse(temp > past_high,
                               "#CD2626",
                               "#00000000")))
#> Joining, by = "yearday"

# Structure of record_high_low
str(record_high_low)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    153 obs. of  8 variables:
#>  $ month    : num  1 1 1 1 1 1 1 1 1 1 ...
#>  $ day      : num  1 2 3 4 5 6 7 8 9 10 ...
#>  $ year     : num  2016 2016 2016 2016 2016 ...
#>  $ temp     : num  41 37 40 33 19 32 39 40 43 50 ...
#>  $ yearday  : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ past_low : num  17 15 16 15 21 14 14 12 21 8.5 ...
#>  $ past_high: num  51 48 57 55 56 62 52 57 54 47 ...
#>  $ record   : chr  "#00000000" "#00000000" "#00000000" "#00000000" ...

# Add point layer of record_high_low
p <- ggplot(past_summ, aes(x = yearday, y = temp)) +
  geom_point(col = "#EED8AE", alpha = 0.3, shape = 16) +
  geom_linerange(aes(ymin = CI_lower, ymax = CI_upper), col = "#8B7E66") +
  geom_line(data = present) +
  geom_point(data = record_high_low, aes(col = record)) +
  scale_color_identity()

p
```



Great! Those are a lot of geom layers.

Custom Legend

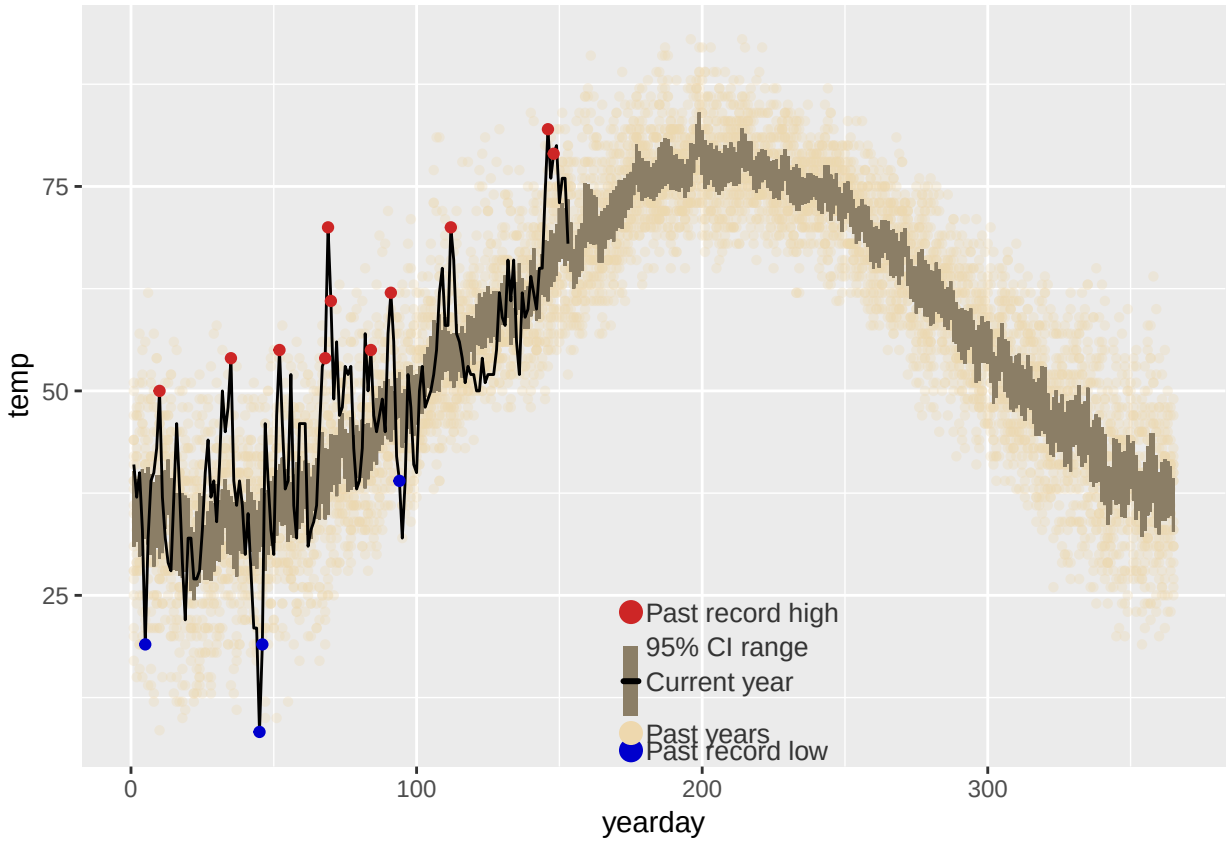
Although you have a lot of information on your plot, the only aesthetic that you used was color, so the legend won't reflect all geoms and color attributes that you've used.

This means you'll have to create your own legend. You'll do this with the grid package plotting functions that you can call after generating the `ggplot` itself.

We've set up a new function, `draw_pop_legend()`, that takes 5 arguments. Your task it to complete the rest. The function will push a `viewport` using `pushViewport(viewport())`. Code to position the points, rectangle and black line has been provided for you. You should be able to understand what's happening here from the previous chapter on `grid` graphics. Feel free to play around with the arguments, but the defaults should work fine.

```
# Finish the function draw_pop_legend
draw_pop_legend <- function(x = 0.6, y = 0.2, width = 0.2, height = 0.2, fontsize = 10) {

  # Finish viewport() function
  pushViewport(viewport(x = x, y = y, width = width, height = height, just = "center"))

  legend_labels <- c("Past record high",
                     "95% CI range",
                     "Current year",
                     "Past years",
                     "Past record low")
```

```
  legend_position <- c(0.9, 0.7, 0.5, 0.2, 0.1)

  # Finish grid.text() function
  grid.text(label = legend_labels, x = 0.12, y = legend_position,
            just = "left",
            gp = gpar(fontsize = fontsize, col = "grey20"))

  # Position dots, rectangle and line
  point_position_y <- c(0.1, 0.2, 0.9)
  point_position_x <- rep(0.06, length(point_position_y))
  grid.points(x = point_position_x, y = point_position_y, pch = 16,
              gp = gpar(col = c("#0000CD", "#EED8AE", "#CD2626")))
  grid.rect(x = 0.06, y = 0.5, width = 0.06, height = 0.4,
            gp = gpar(col = NA, fill = "#8B7E66"))
  grid.lines(x = c(0.03, 0.09), y = c(0.5, 0.5),
             gp = gpar(col = "black", lwd = 3))

  # Add popViewport() for bookkeeping
  popViewport()
}

# Plotting object p, from previous exercise
p

# Call draw_pop_legend()
draw_pop_legend()
```



Great! Note: There are a couple other ways you could have done this. You could have used `ggplot2::annotate()`, which is rather tedious and not as reproducible. You could have also used a layout when making the viewport, filling each segment in the viewport layout with specific elements. You could have also named each viewport so as to access the graphics objects, grobs, later on. Here, you're also neglecting checks and error messages which would ensure your function always works properly - so it's not quite ready for release into the wild. For our purposes, this working version will suffice, but in a package you'd probably want something more robust.

Case Study II - Weather (Part 2)

The previous steps we considered worked, but that's not the `ggplot2` way. So, what is? In end we had many dataframes, which we used to assemble the final plot. It's kind of like making a summary dataframe and then plotting that as a separate layer on top of raw values. We could do that, but in the second course we saw that we can use functions like `stat_summary()` to do all that for us. Here, we're going to wrap up our case study by making use of some stat layers, which will do a similar thing.

We'll have the `stat_historical` to plot the historical record. `stat_present` to plot the most recent year and `stat_extremes` to mark the new records highs and lows. We saw something similar to this in the case study of the previous course where we created a mosaic plot. There the solution was the wrap the steps into a big function. This worked well enough, but here, by going one step further, our methods are more generalized and flexible. For example, this means I can choose specific layers and use them in the context of other ggplot2 layers such as faceting without having to produce large chunks of code.

Step 1: clean_weather()

Now that you've created a unique visualization to answer a specific data analysis question, it would be nice if we can make our lives even easier. Having to continually run all the code you created previously to make

your plot is a big hassle! What if you wanted to do this for many different cities?

Instead of trying to combine all steps into a single function, which is what you did with the mosaic plot at the end of course 2, you'll make several `stat` functions which will act as layers in `ggplot2` plots. This will help you to make flexible plots in the future.

Let's start by making a quick and dirty function to read in and clean up the data.

```
# Finish the clean_weather function
clean_weather <- function(file) {
  weather <- read.fwf(file,
                      header = FALSE,
                      col.names = c("month", "day", "year", "temp"),
                      widths = c(14, 14, 13, 4))
  weather %>%
    filter(!(month == 2 & day == 29)) %>%
    group_by(year) %>%
    mutate(yearday = 1:length(day)) %>%
    ungroup() %>%
    filter(temp != -99)
}

# Import NYNEWYOR.txt: my_data
my_data <- clean_weather("datasets/NYNEWYOR.txt")
```

Well done! From now on, getting clean weather data will be as easy as calling `clean_weather()`!

Step 2: Historical Data

Let's develop the first layer to show the historical data called stat_historical(). Like before, you'll create a stats layer starts by creating a stats object with ggproto(), and then you'll create a function where you'll define a number of layers.

To do this properly, you are going to define a new aesthetic inside ggproto(). This will be called "year", and you can map the variable year from your dataset onto this aesthetic. You'll see in a bit that this a very elegant solution.

```
# Create the stats object
StatHistorical <- ggproto("StatHistorical", Stat,
                    compute_group = function(data, scales, params) {
                      data <- data %>%
                        filter(year != max(year)) %>%
                        group_by(x) %>%
                        mutate(ymin = Hmisc::smean.cl.normal(y)[3],
                               ymax = Hmisc::smean.cl.normal(y)[2]) %>%
                        ungroup()
                    },
                    required_aes = c("x", "y", "year"))

# Create the layer
stat_historical <- function(mapping = NULL, data = NULL, geom = "point",
                            position = "identity", na.rm = FALSE, show.legend = NA,
                            inherit.aes = TRUE, ...) {
  list(
    layer(
      stat = "identity", data = data, mapping = mapping, geom = geom,
      position = position, show.legend = show.legend, inherit.aes = inherit.aes,
```

```
      params = list(na.rm = na.rm, col = "#EED8AE", alpha = 0.3, shape = 16, ...)
    ),
    layer(
      stat = StatHistorical, data = data, mapping = mapping, geom = "linerange",
      position = position, show.legend = show.legend, inherit.aes = inherit.aes,
      params = list(na.rm = na.rm, col = "#8B7E66", ...)
    )
  )
}

# Build the plot
my_data <- clean_weather("datasets/NYNEWYOR.txt")
ggplot(my_data, aes(x = yearday, y = temp, year = year)) +
  stat_historical()
```

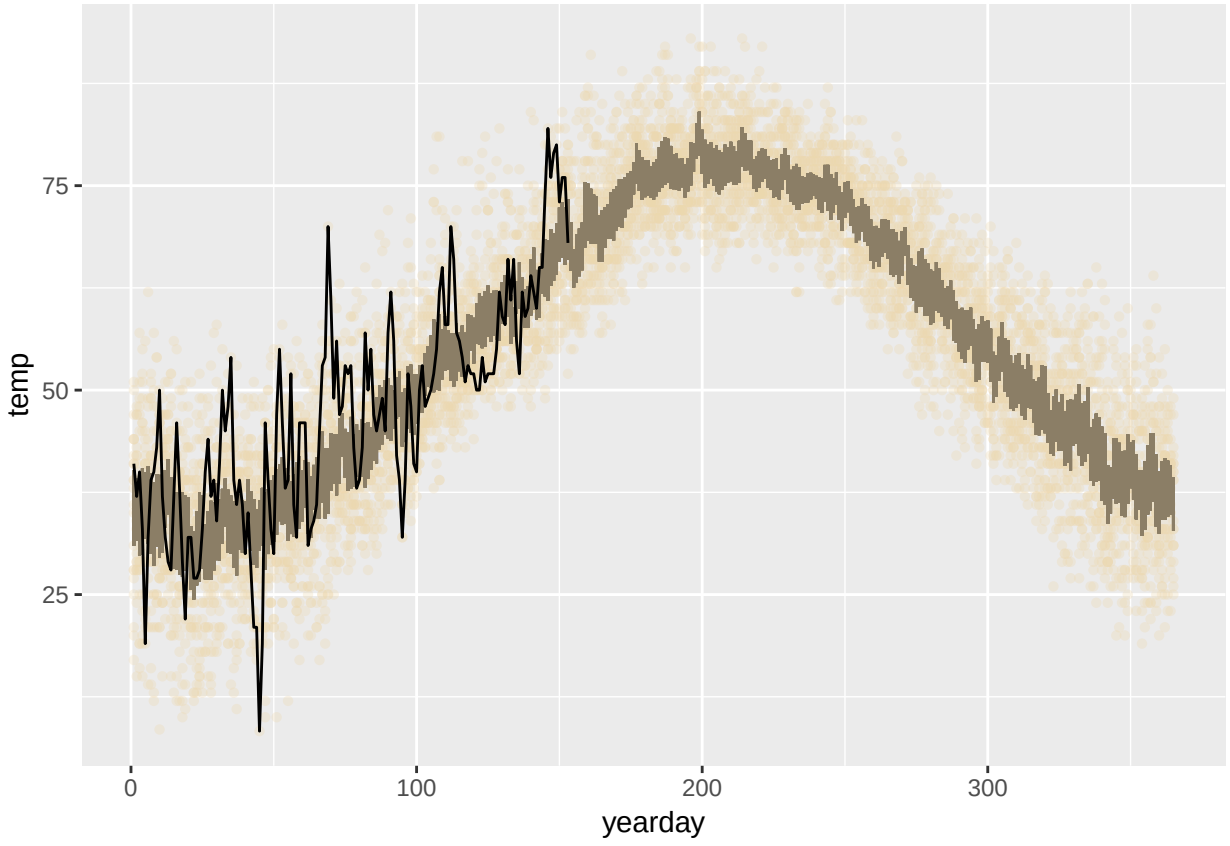
Great job! We've got a new layer: stat_historical()

Step 3: Present Data

Nice. Now you have the historic record. What about getting the current year? This is really straight-forward! You just need to do the same trick of defining a new "year" aesthetic and then filter the dataset accordingly. This time you only need one layer, there's no notion of a line range geom in this case!

```
# Create the stats object
StatPresent <- ggproto("StatPresent", Stat,
                       compute_group = function(data, scales, params) {
                         data <- filter(data, year == max(year))
```

```
                        },
                        required_aes = c("x", "y", "year"))

# Create the layer
stat_present <- function(mapping = NULL, data = NULL, geom = "line",
                        position = "identity", na.rm = FALSE, show.legend = NA,
                        inherit.aes = TRUE, ...) {
  layer(
    stat = StatPresent, data = data, mapping = mapping, geom = geom,
    position = position, show.legend = show.legend, inherit.aes = inherit.aes,
    params = list(na.rm = na.rm, ...)
  )
}

# Build the plot
my_data <- clean_weather("datasets/NYNEWYOR.txt")
ggplot(my_data, aes(x = yearday, y = temp, year = year)) +
  stat_historical() +
  stat_present()
```



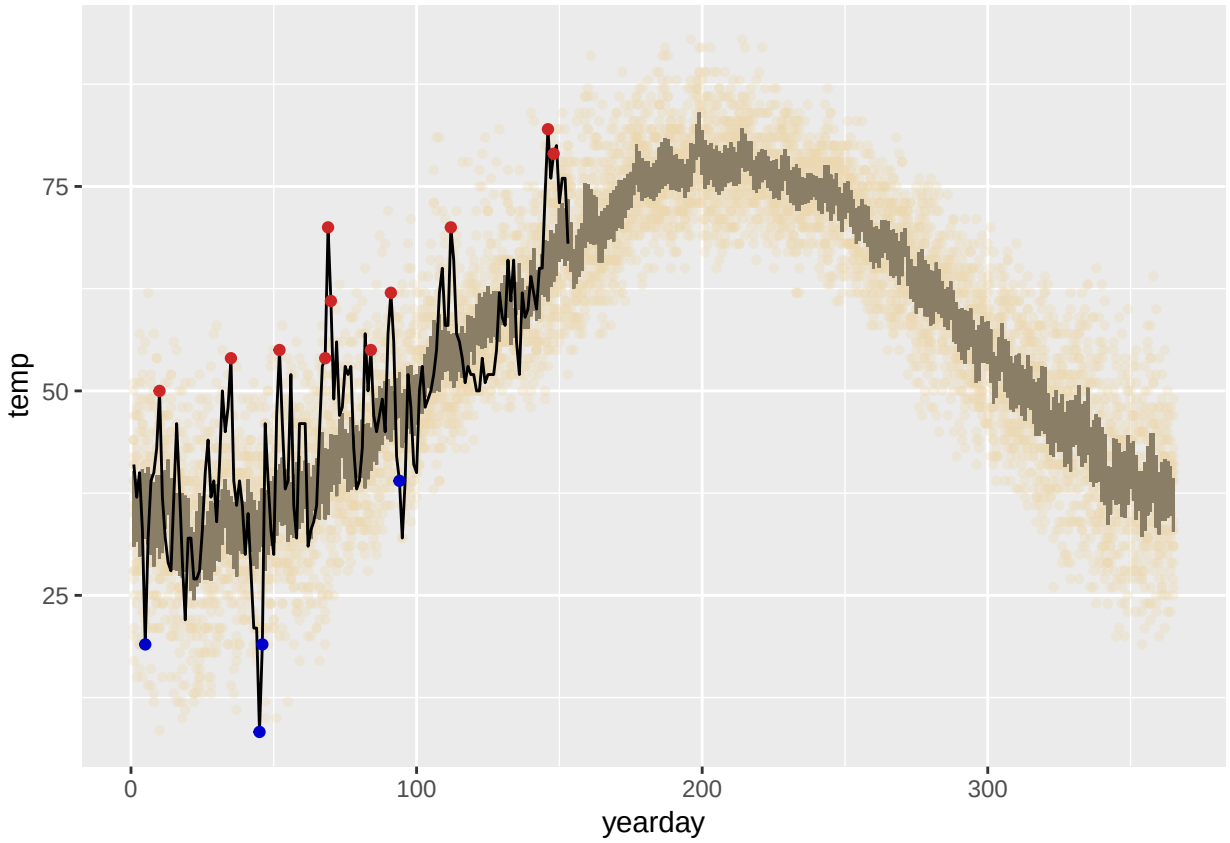
I love it - things are shaping up nicely!

Step 4: Extremes

The last step is to create a `stat_extreme()` layer. This takes advantage of the calculations you did earlier, which combined the definitions of new highs and lows into one variable, `record`. This variable contains the

colors of the dots: deep blue for record lows, dark red for record highs, and transparent dots otherwise. Because this `record` variable is 'internally computed', you'll have to refer to it as `..record..` when you're creating the plot.

```
# Create the stats object
StatExtremes <- ggproto("StatExtremes", Stat,
                        compute_group = function(data, scales, params) {

                          present <- data %>%
                            filter(year == max(year))

                          past <- data %>%
                            filter(year != max(year))

                          past_extremes <- past %>%
                            group_by(x) %>%
                            summarise(past_low = min(y),
                                      past_high = max(y))

                          # transform data to contain extremes
                          data <- present %>%
                            left_join(past_extremes) %>%
                            mutate(record = ifelse(y < past_low,
                                                   "#0000CD",
                                                   ifelse(y > past_high,
                                                          "#CD2626",
                                                          "#00000000")))
                        },
                        required_aes = c("x", "y", "year"))

# Create the layer
stat_extremes <- function(mapping = NULL, data = NULL, geom = "point",
                          position = "identity", na.rm = FALSE, show.legend = NA,
                          inherit.aes = TRUE, ...) {
  layer(
    stat = StatExtremes, data = data, mapping = mapping, geom = geom,
    position = position, show.legend = show.legend, inherit.aes = inherit.aes,
    params = list(na.rm = na.rm, ...)
  )
}

# Build the plot
my_data <- clean_weather("datasets/NYNEWYOR.txt")
ggplot(my_data, aes(x = yearday, y = temp, year = year)) +
  stat_historical() +
  stat_present() +
  stat_extremes(aes(col = ..record..)) +
  scale_color_identity() # Colour specification
#> Joining, by = "x"
```



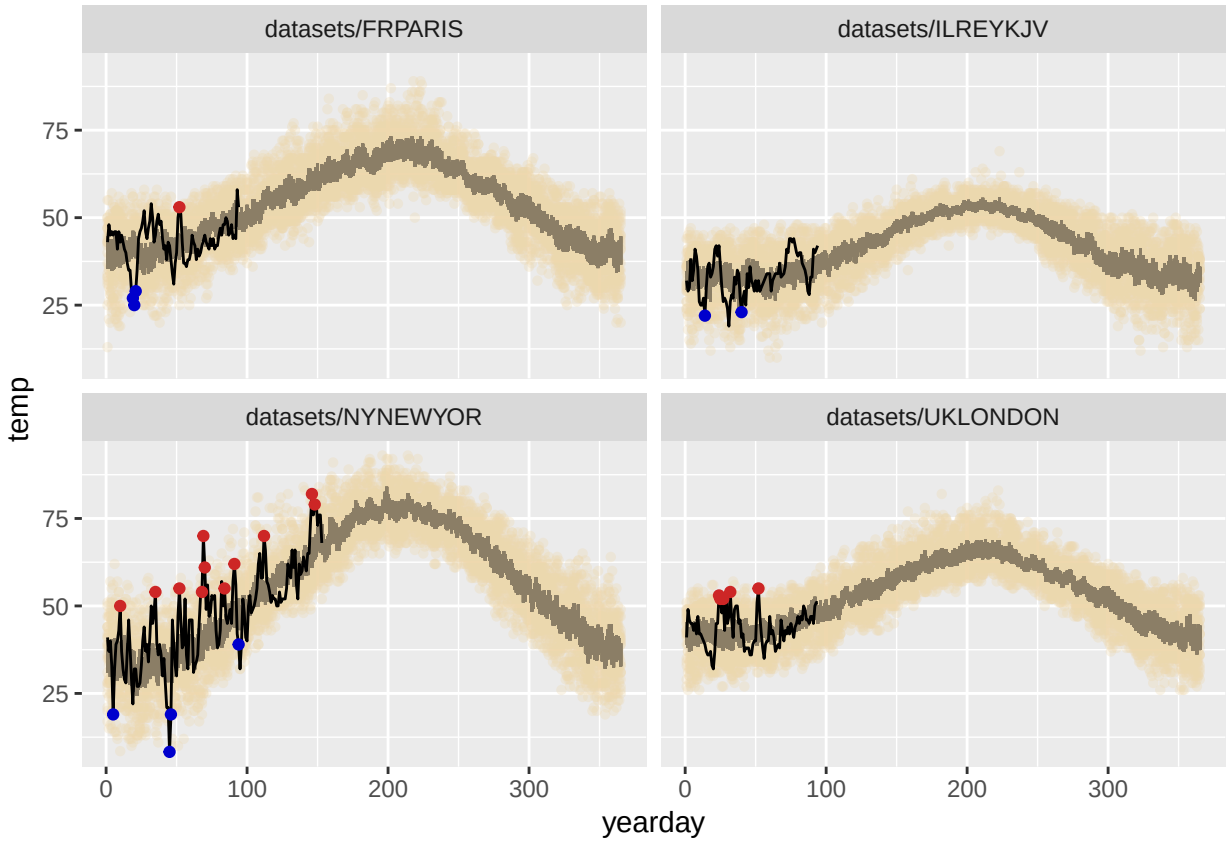
Awesome! Can you see how these all fit together? That plot is as DRY as a good martini!

Step 5: Re-use Plotting Style

You're at the end of your case study: you have built up three `stat_` functions that you can use in any combination you like and with any dataset that fits the type of stats you are doing. Let's try it out on some other cities! In your working directory, there are four files, related to the weather data for New York, Paris, Reykavik and London. Their filenames are stored in the `my_files` vector.

```
# File paths of all datasets

my_files <- list.files("datasets/")

my_files <- c("datasets/NYNEWYOR.txt", "datasets/FRPARIS.txt", "datasets/ILREYKJV.txt", "datasets/UKLONI

# Build my_data with a for loop
my_data <- NULL
for (file in my_files) {
  temp <- clean_weather(file)
  temp$id <- sub(".txt", "", file)
  my_data <- rbind(my_data, temp)
}

# Build the final plot, from scratch!
ggplot(my_data, aes(x = yearday, y = temp, year = year)) +
  stat_historical() +
  stat_present() +
```

```
stat_extremes(aes(col = ..record..)) +
scale_color_identity() +  # specify color here
facet_wrap(~id, ncol = 2)
```



Congratulations! You've completed all three ggplot2 courses. Grab yourself a DRY martini and head over to the final video!

Wrap Up

Data visualization lies at the interesection of statistics and design.