BROOKE ANDERSON

DATA VISUALIZATION IN R

Contents

1

Prerequisites

I have based this workshop on examples for you to try yourself, because you won't be able to learn how to program unless you try it out. I've picked example data that I hope will be interesting to Navy and Marine Corp public health researchers and practitioners.

To try out these examples, you need some set-up:

1. Download R
2. Download RStudio
3. Install some R packages
4. Create an R Project for the examples
5. Download the example data and save it in the R Project

This section will walk you through each step, explaining both *how* and *why* to do the step. Click on the **Next** button (or navigate using the links at the top of the page) to continue.

1.1 Download R

1.2 Download RStudio

1.3 Install some R packages

1.4 Create an R Project

1.5 Download example data

2
Plot

R programming has changed a lot in the past 10 years or so. This includes a big change in the tools used for visualization. The R package **ggplot2** (Wickham et al., 2018a) is at the heart of one modern style of R visualizations. I'll focus on this style throughout this workshop, as I think it's an excellent tool for creating attractive and useful visualizations in a way that is both efficient and easy to customize (once you get the hang of it).

2.1 Plot elements

ggplot2 is based on a **grammar of graphics**. To get the hang of it, you'll need to start thinking of visualizations in terms of separate elements.

2.1.1 Example data

In my own research, I study the health effects of climate-related disasters, including heat waves and hurricanes. I noticed that a lot of the sessions at this conference focus on public health surveillance, so I thought it might be interesting to combine these two ideas for the example data. On September 10, 2017, Hurricane Irma hit Florida, and before it did, it triggered evacuations for much of the state. The National Highway Traffic Safety Administration (under the US Department of Transportation) tracks all the fatal motor vehicle accidents in the US through its Fatality Analysis Reporting System (FARS).[1]

I downloaded and cleaned some data from this surveillance system. (In a later section, I'll tell you more about how to do this cleaning yourself.) I've created a fairly simple dataset for us to use here. For each date in the weeks around the hurricane's landfall, it gives the total number of motor vehicle fatalities recorded in the state. The dataset also gives the week in the year for each date (the first week in January would be "1" for this measure, etc.), as well as the day of the week. Table 2.1 shows what this data looks like.

[1] **Fatality Analysis Reporting System (FARS).** A surveillance system for all fatal motor vehicle accidents in the US, maintained by the National Highway Traffic Safety Administration. For more, see the FARS website.

Date	Week of year	Day of week	No. of motor vehicle fatalities
2017-08-27	35	Sunday	4
2017-08-28	35	Monday	5
2017-08-29	35	Tuesday	6
2017-08-30	35	Wednesday	6
2017-08-31	35	Thursday	6
2017-09-01	35	Friday	9
2017-09-02	35	Saturday	8
2017-09-03	36	Sunday	15
2017-09-04	36	Monday	7
2017-09-05	36	Tuesday	8
2017-09-06	36	Wednesday	7
2017-09-07	36	Thursday	12
2017-09-08	36	Friday	9
2017-09-09	36	Saturday	4
2017-09-10	37	Sunday	6
2017-09-11	37	Monday	4
2017-09-12	37	Tuesday	6
2017-09-13	37	Wednesday	2
2017-09-14	37	Thursday	4
2017-09-15	37	Friday	4
2017-09-16	37	Saturday	4
2017-09-17	38	Sunday	10
2017-09-18	38	Monday	7
2017-09-19	38	Tuesday	8
2017-09-20	38	Wednesday	6
2017-09-21	38	Thursday	5
2017-09-22	38	Friday	9
2017-09-23	38	Saturday	7

Table 2.1: Number of motor vehicle fatalities in Florida around the date of Hurricane Irma's Florida landfall on September 10, 2017.

2.1.2 Illustrating plot elements

I've created a simple plot of this data to use to highlight the different elements of a graph (Figure 2.1). This plot shows the number of motor vehicle fatalities in Florida per day in the weeks around Hurricane Irma, with the day of the week shown with color (since, for some health outcomes, there are patterns by the day of week).

Let's break this plot into some of its key elements:

- **data**: The data illustrated with this plot is all from the example data shown in Table 2.1.
- **geoms**:[2] The geometric objects used to plot the data are (1) points (in different colors, depending on the day of week) and (2) a line (in gray).
- **aesthetics**: For both of the geoms (points and the line), the position along

[2] **geoms.** The geometric objects (e.g., points, lines, bars, columns, polygons, rectangles, text, labels) used to display data for `ggplot` plots.

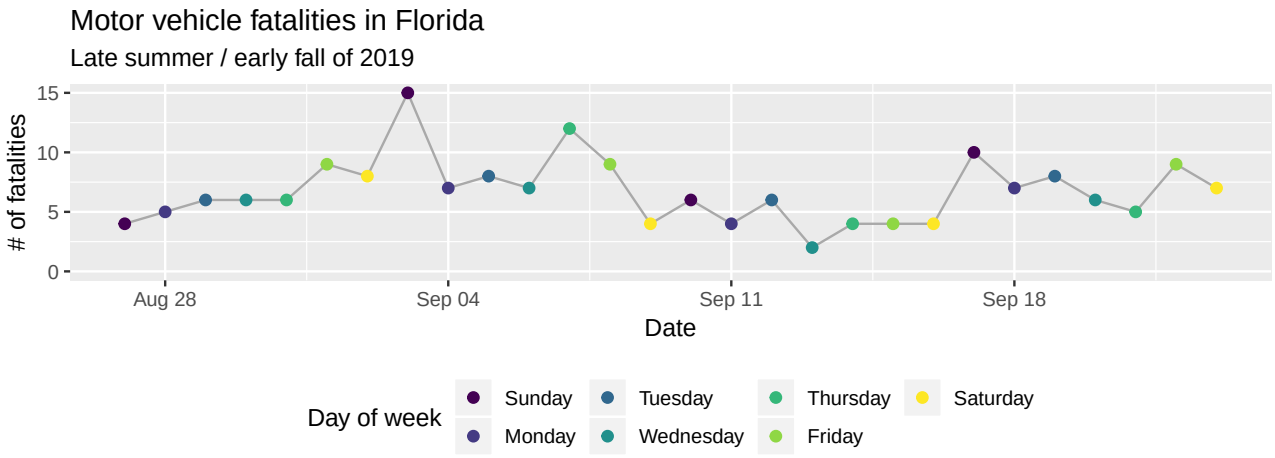


Figure 2.1: Number of motor vehicle fatalities by day in Florida in the weeks surrounding Hurricane Irma on September 10, 2019.

the x-axis shows (is **mapped to**) the date given for an observation in the data. The position along the y-axis is mapped to the number of fatalities for that observation. For the points (but not the line), the color is mapped to the day of week of the observation. For the line, the color is always gray (a **constant aesthetic** for color), rather than color being mapped to a value in the data. Other aesthetics—like size, shape, line type and transparency—have been left at their default (constant) values.

- **coordinate system**: The plot uses a **Cartesian coordinate system**, the most common coordinate system you'll use except when creating maps.
- **scales**: The plot uses a default **date scale** for the x-axis. For the y-axis, the scale is very similar to a default **continuous scale** y-axis, but has been expanded a bit to include 0. The **color scale**is more customized. It uses a color scale that's very popular right now called "viridis", rather than the default color scale.
- **labels**: This plot uses the axis titles "Date" for the x-axis, "# of fatalities" for the y-axis, and "Day of week" for the color scale. In a minute, when you start working with the example data, you'll see that these are changed from the corresponding column names in the data, to make the plot easier to understand. In addition, the plot has both a title ("Motor vehicle fatalities in Florida") and a subtitle ("Late summer / early fall of 2019").
- **theme**:[3] This plot uses the default `theme_gray` theme, with a gray background to the main plot area, white gridlines, a Sans Serif font family, and a base font size of 11. The one customization is that the legend (which here provides the key for how color maps to day of the week) is shown on the bottom of the plot rather than to the right of the plot.
- **faceting**: This plot does not take advantage of faceting. Instead, the data is plotted on a single background. The next example will show an example of faceting based on a characteristic of the data.

[3] **theme.** A collection of specifications for the background elements of the plot, including the plot background, grid lines, legend position, text size and font, margins, and axis ticks.

To help the meaning of these elements sink in, Figure 2.2 shows a second example plot, with the elements again explained below the plot.

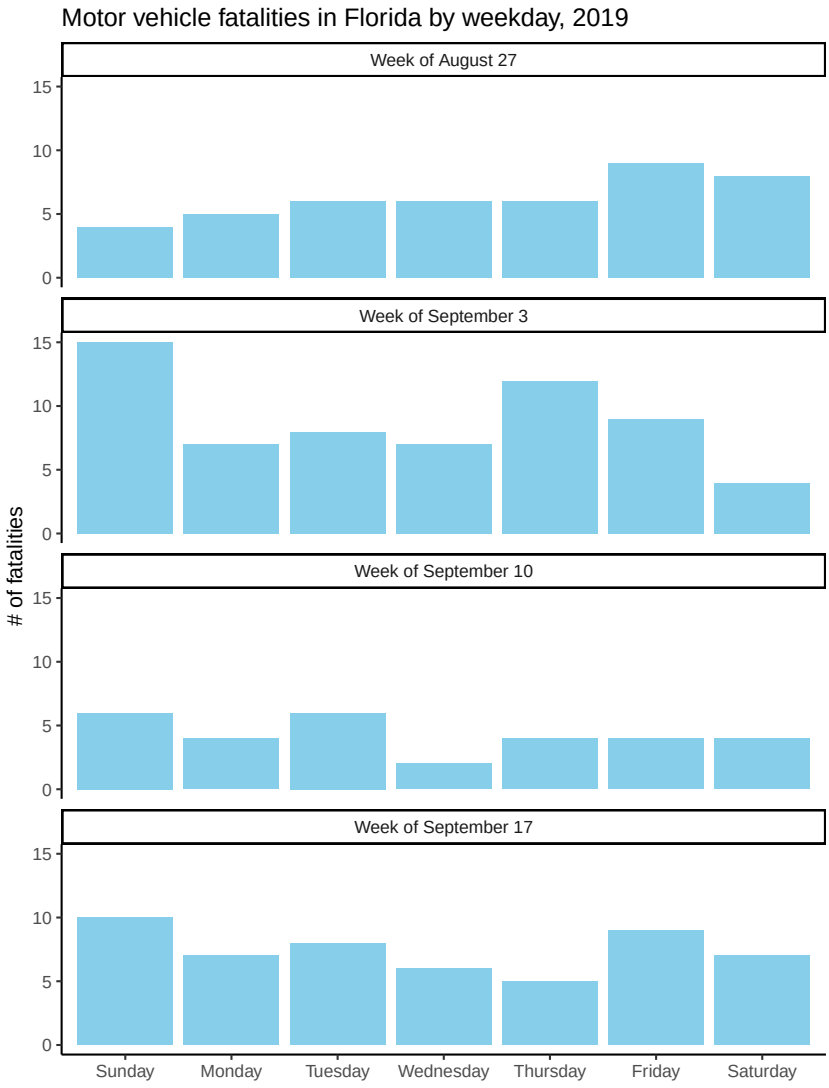


Figure 2.2: Number of motor vehicle fatalities in Florida by day of the week and week for the weeks surrounding Hurricane Irma's landfall.

Here's the breakdown of plot elements for this plot:

- **data**: The data illustrated with this plot is the same as for Figure 2.2, the example data shown in Table 2.1.
- **geoms**: The geometric objects used to plot the data are columns.
- **aesthetics**: For the column geoms, the x-axis position is mapped to the day of the week and the y-axis position is mapped to the number of fatalities. The color is mapped to a constant aesthetic, sky blue.
- **coordinate system**: The plot uses a **Cartesian coordinate system**.
- **scales**: The plot uses a default **discrete scale** for the x-axis and the default **continuous scale** for the y-axis.

- **labels**: This plot uses the axis titles "Date" for the x-axis, "# of fatalities" for the y-axis, and "Day of week" for the color scale. In a minute, when you start working with the example data, you'll see that these are changed from the corresponding column names in the data, to make the plot easier to understand. In addition, the plot has both a title ("Motor vehicle fatalities in Florida") and a subtitle ("Late summer / early fall of 2019").
- **theme**: This plot uses the `theme_classic` theme, with a white background to the main plot area, no gridlines, a Sans Serif font family, and axis lines only on the left and bottom sides of the plot area.
- **faceting**: This plot facets by week. This variable was obtained from the "week" column in the dataset, although some changes were made to have better labeling of the facets (e.g., "Week of August 27" rather than "35").

2.2 Building a plot

Now that you have and idea of how different elements combine to create a plot, I'll walk you through the steps to "layer" these elements together to create these two example plots.

2.2.1 Reading data

First, you'll need to bring the example data into your R session. If you followed the steps in the "Prerequisites" section, you should have a **comma-separated file**[4] in a "data" subdirectory of your current working directory. Also, if you followed the steps in the "Prerequisites" section, you should have installed all the necessary R packages for this example.

While there are functions in base R that import comma-separated files, I think the functions in the **readr** package (Wickham et al., 2018b), a package in the **tidyverse**, has some nicer defaults. To use functions in this package, you first need to load it into your R session using the `library` function:

```
library("readr")
```

The `library` function must specify the name of the package you'd like to load (in this case, "readr"). If you have forgotten to install the "readr" package before you run this function, you'll get the message:

```
Error in library(readr) : there is no package called 'readr'
```

If you get that message, go back and re-read the "Prerequisites" to make sure you've installed all the required packages.[5]

Now that you've loaded `readr`, you can use the package's `read_csv` function[6] to read in the data. The one argument this function requires is the **file path**[7] to the data file.

```
daily_fatalities <- read_csv("data/daily_fatalities.csv")
```

[4] **comma-separated file format.** A flat file format (try opening the file in a plain text editor—unlike a binary file format like Excel, you should be able to read all the content) where each column entry is separated by a comma. This is a common file format for tabular data that can be read into and written from most statistical programs (including R, Excel, and SAS).

[5] There is also a small chance, especially if you're using a computer for which you don't have superuser privileges, that something else is going on. Double-check that you installed the package you're trying to load, and if loading the package still doesn't work, talk with your IT team about installing and loading R packages on your computer. For some company- or government-issued computers, the computer might have been set up to restrict installing new software, or to save installed software somewhere other than the default locations where R searches with `library`.

[6] For this and any other R function, you can open a help file in RStudio to give you more information about the function, usually including some examples of how to use it. Just

```
## Parsed with column specification:
## cols(
##   date = col_date(format = ""),
##   week = col_double(),
##   weekday = col_character(),
##   fatals = col_double()
## )
```

Don't be alarmed by the message that's printed out after you run the function! In data frames in R, each columns can have one of several different classes (some examples: character: "Florida", date: 2017-09-10, integer: 17, double: 17.0). The `read_csv` function looks at the values in each column in the data and tries to guess what class each column should be, and this message tells you what it guessed, so you can check. In this case, the number of fatalities will always be a whole number, so the "integer" class would have also worked well, but there should be no problem with the column having a "double" class for the plotting we'll be doing, so everything looks fine.

The previous code used a **gets arrow**[8] to save the data you read in to an R object called `daily_fatalities`. Now, anytime you want to use this data, you can reference it with the name `daily_fatalities` instead of needing to read it in again. For example, you can print out the start of the data by calling the object name by itself:

[8] **gets arrow.** The function <−, which allows you to save the output from running the code on the right-hand side of the arrow to an object with the name on the left of the arrow. This is a funny kind of function called an **infix function**, which goes between two function arguments instead of putting the arguments inside parentheses. If you want to look up the helpfile for an infix function, wrap it in backticks: `?`<-`.

```
daily_fatalities
```

```
## # A tibble: 28 x 4
##    date        week weekday   fatals
##    <date>     <dbl> <chr>      <dbl>
##  1 2017-08-27    35 Sunday         4
##  2 2017-08-28    35 Monday         5
##  3 2017-08-29    35 Tuesday        6
##  4 2017-08-30    35 Wednesday      6
##  5 2017-08-31    35 Thursday       6
##  6 2017-09-01    35 Friday         9
##  7 2017-09-02    35 Saturday       8
##  8 2017-09-03    36 Sunday        15
##  9 2017-09-04    36 Monday         7
## 10 2017-09-05    36 Tuesday        8
## # ... with 18 more rows
```

By default, `read_csv` read the data into a structure called a **tibble**,[9] and you can see that the top of the print-out notes this. It also gives the dimensions (28 rows and 4 columns) and, under each column name, the class of the column. When you start plotting this data, you'll use the dataframe's object name (`daily_fatalities`) to reference the full dataset and each column name

[9] **tibble.** A slightly fancier dataframe, which is a tabular data format in R, where there are rows and columns, with each column having data with the same class (e.g., character, integer, date) and all columns having the same length. Values in the columns should "line up" across the rows—for example, in this case, the second value in the `fatals` column should be the number of fatalities for the date given by the second value in the `date` column. Compared to a dataframe, a tibble prints out more nicely when you call the object name alone, among some other advantages.

(e.g., `date` for the date of the observations, `fatals` for the number of fatalities observed on that date) to reference specific elements of the data.

2.2.2 *Plotting by layers*

To create a data visualization using `ggplot2`, we'll add up "layers" for each of the plot elements described earlier. In this section, I'll step through this process. First, we'll create a **ggplot object** using the `ggplot` call, and then we'll add layers to it with +. To use these functions, you'll need to load the `ggplot2` package:

```
library(ggplot2)
```

For the first step, create the `ggplot` object. When you create this object, specify the dataframe with the data you'd like to plot using the `data` parameter:

```
ggplot(data = daily_fatalities)
```


Figure 2.3: Step 1 of layering a plot: Creating the ggplot object. At this point, nothing's actually plotted, because we haven't added any geoms yet.

Figure 2.3 shows the output with this single, initial layer. We haven't added any geoms yet, so the plot isn't showing anything. Since we've specified the data, however, we'll be able to add geoms where we map aesthetics to columns in the dataset. Let's do that next, and add a layer with a line (`geom_line`) for the number of fatalities per day. We'll use the `aes` function *inside* the `geom_line` call to specify that we want the x-axis to show the value in the `date` column and the y-axis to show the value in the `fatals` column:

```
ggplot(data = daily_fatalities) + geom_line(aes(x = date,
    y = fatals))
```

The result is in Figure 2.4. You can see that you now have a line showing the number of fatalities per day. Next, we can add a layer on top of the line with a

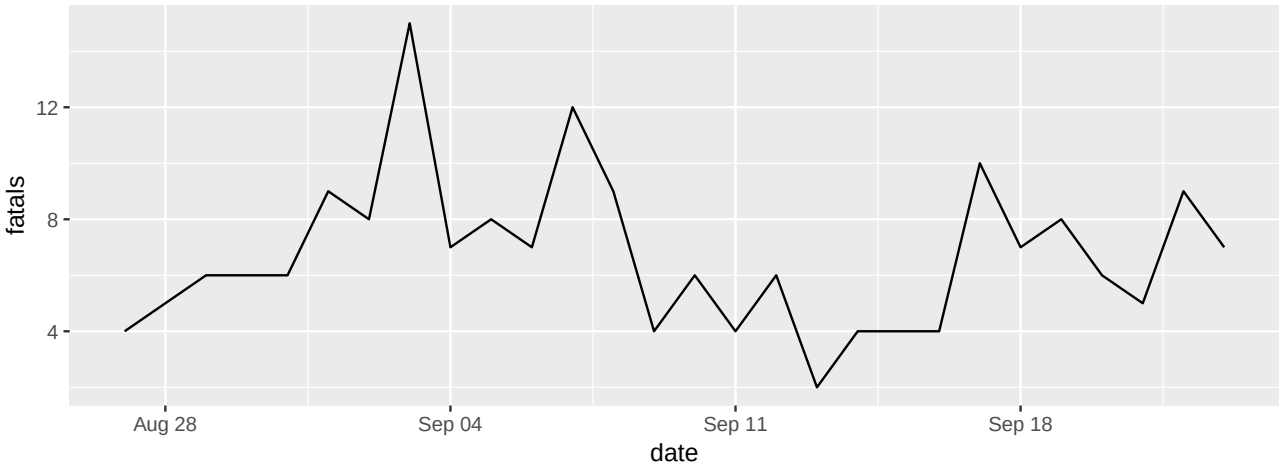


Figure 2.4: Step 2 of layering a plot: Adding a line geom. In this case, the x-axis (x aesthetic) is mapped to the 'date' column in the data, while the y-axis (y aesthetic) is mapped to the 'fatals' column.

point (`geom_point`) for each date showing the number of fatalities. Two of the aesthetics for this geom (x and y) will be the same as for the line. However, we also want to map color to day of the week. Since day of the week is in a column called `weekday`, we can specify this aesthetic as `color = weekday` within the `aes` call. Try running the following code to add this layer:

```
ggplot(data = daily_fatalities) + geom_line(aes(x = date,
    y = fatals)) + geom_point(aes(x = date, y = fatals,
    color = weekday))
```

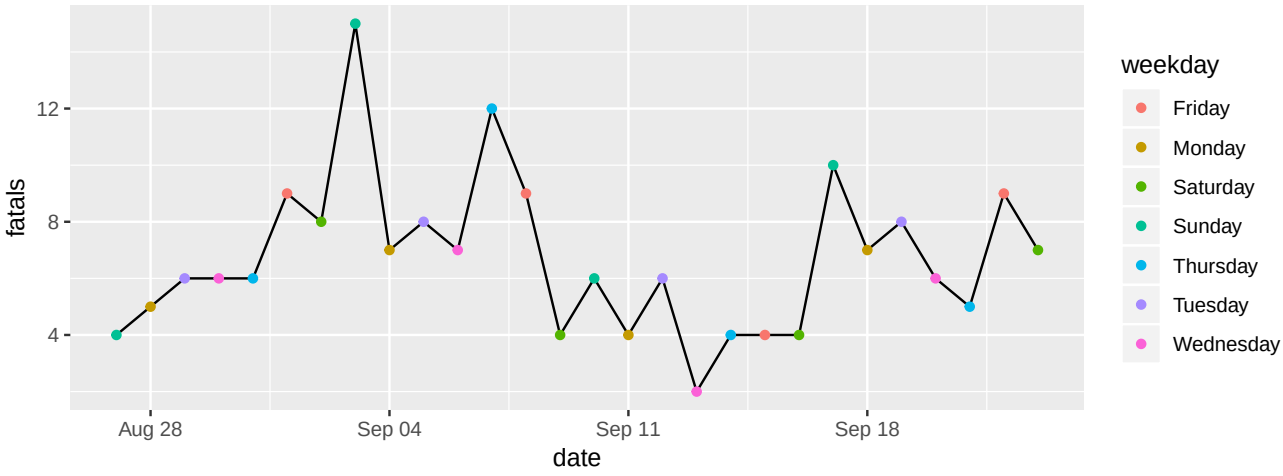


Figure 2.5: Step 3 of layering a plot: Adding a point geom. The x and y aesthetics are the same as for the line geom, but now we're also mapping color to the 'weekday' column in the data.

You should get the plot shown in Figure 2.5. You may have noticed that the set of colors that is used for weekdays is different than in the plot in Figure 2.1. This set of colors is actually specified by the "scale" element of the plot, so we'll change that in a different layer.

So far, we've customized some of the plot aesthetics by mapping them so that their values are based on observations in the data. However, sometimes you'll want to change an aesthetic to a **constant**, where the aesthetic changes for the aesthetic for all of the observations from the data, but in the same way.

For example, for this plot, we'd like the line to be dark gray for all observations, and we'd like the points to be a little bigger. To specify a **constant aesthetic**, move it outside of the `aes` call. For colors, you can set a constant value to one of the many named "R Colors"[10] (make sure you put the color name inside parentheses—otherwise, R will think you're refering to an R object). Constant point size values can be specified using numbers, where larger numbers will make bigger points and the default value is somewhere around 1.[11] Try numbers bigger than 1 (e.g., 1.5, 2) for bigger points and smaller values (e.g., 0.8, 0.5) for smaller points.

[10] See this website for examples and names

[11] For an in-depth look at aesthetic specifications, see the ggplot2 specs vignette

The following code will set the line to be dark gray and the points to be a bit larger (result in Figure 2.6):

```
ggplot(data = daily_fatalities) + geom_line(aes(x = date,
    y = fatals), color = "darkgray") + geom_point(aes(x = date,
    y = fatals, color = weekday), size = 2)
```

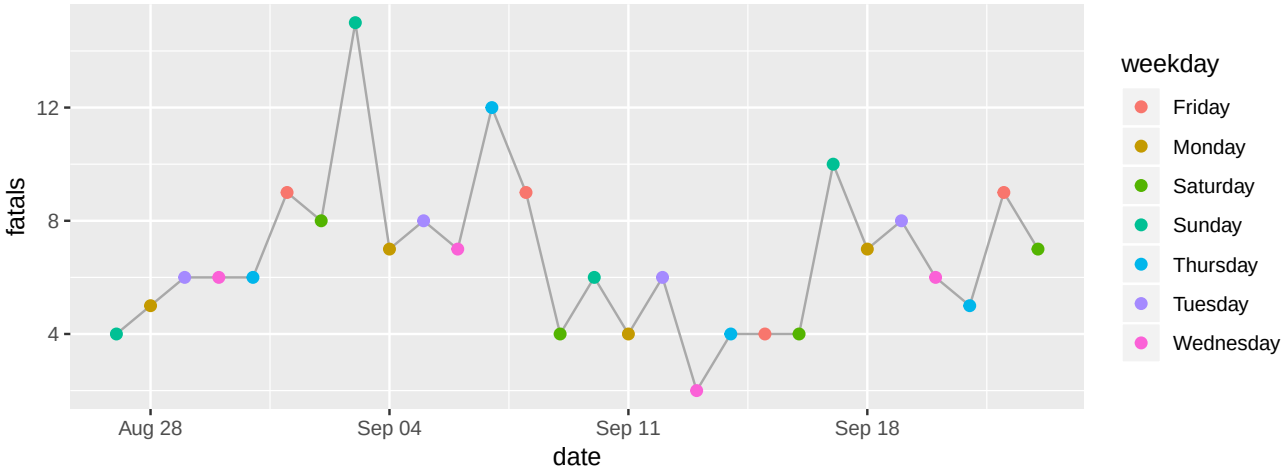


Figure 2.6: Step 4 of layering a plot: Adding constant aesthetics. In this step, we're making the line dark gray for all observations and the points a bit larger. Note that these aesthetics, since they're constant, are set outside of the 'aes' call. Also, note that thee color is specified inside quotation marks.

We're using the default coordinate system for this plot, so we don't need to add a layer for the coordinate system. For plots where you need to change from this default coordinate system, you'll add a layer that starts with `coord_`. For example, in the "Map" section, you'll see how to use a geographic coordinate system using `coord_map`. If you want to flip your x- and y-axis (there are a few examples where this is useful), you can add a layer with `coord_flip`.

While we're also using the default scales for the x aesthetic, we aren't for the y or color aesthetic. The change to the y scale is very minimal: we're just expanding it to include 0. This can be done by adding the layer `expand_limits` with the `y` parameter set to 0.

To change the color scale, you need to add a layer to specify the alternative color scale. We'll use a color scale called "viridis". This is good for discrete data (like here, where we're showing day of the week rather than a continuous number), it really shines when you're plotting continuous values. The order of the scale is clear to those who are colorblind, and it's also clear when printed out on a black-and-white printer.

To change the color scale to use this scale, add a layer with `scale_color_viridis_d`. The "d" here is for "discrete"; if you were using color to show a continuous value in the data (e.g., a column with an integer or double class), you'd add a layer called `scale_color_viridis_c` instead. The final result of these scale customizations is shown in Figure 2.7.

```
ggplot(data = daily_fatalities) + geom_line(aes(x = date,
    y = fatals), color = "darkgray") + geom_point(aes(x = date,
    y = fatals, color = weekday), size = 2) +
    expand_limits(y = 0) + scale_color_viridis_d()
```

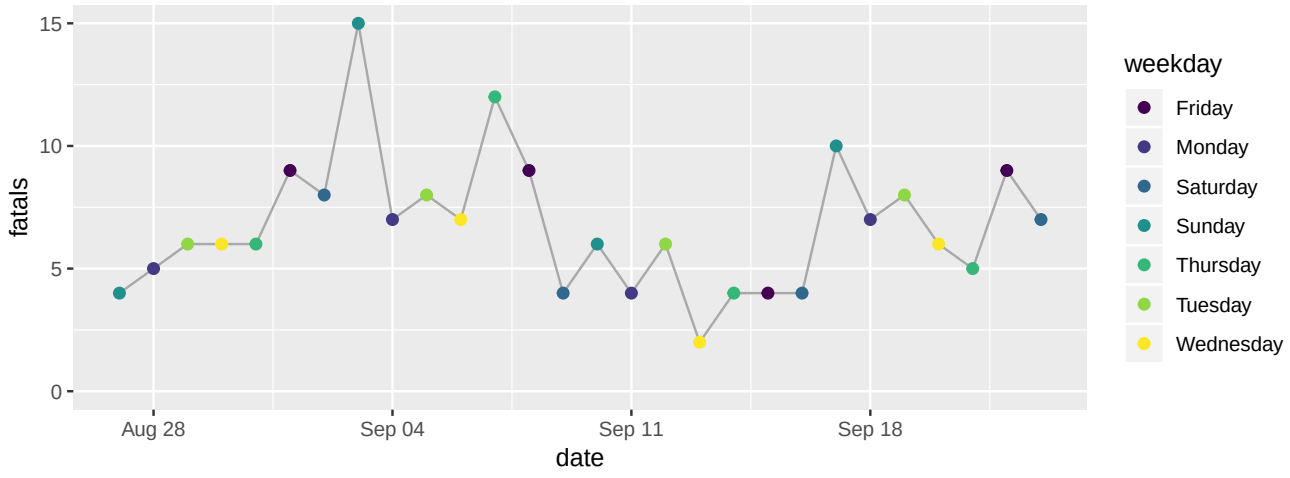


Figure 2.7: Step 5 of layering a plot: Changing the color scale. This step changes from the default color scale to the 'viridis' color scale.

The function for this layer has a few options for customization. For example, try changing it to `scale_color_viridis_d(option = "A")` or `scale_color_viridis_d(direction = -1)`. To see all its options, check its helpfile with `?scale_color_viridis_d`.

Next, we'll change the scale labels. By default, the label for each scale is the name of the column in the data that the aesthetic was mapped to (x: "date", y: "fatals", color: "weekday"). You can add a `labs` layer to change these to labels that are easier to understand. The other labeling we'd like to do is to add a title and subtitle, which we can do with a `ggtitle` layer (the subtitle is added with the `sub` parameter of this layer). The final result of adding these layers is shown in Figure 2.8.

```
ggplot(data = daily_fatalities) + geom_line(aes(x = date,
```

```
    y = fatals), color = "darkgray") + geom_point(aes(x = date,
    y = fatals, color = weekday), size = 2) +
    expand_limits(y = 0) + scale_color_viridis_d() +
    labs(x = "Date", y = "# of fatalities", color = "Day of week") +
    ggtitle("Motor vehicle fatalities in Florida",
        subtitle = "Late summer / early fall of 2019")
```

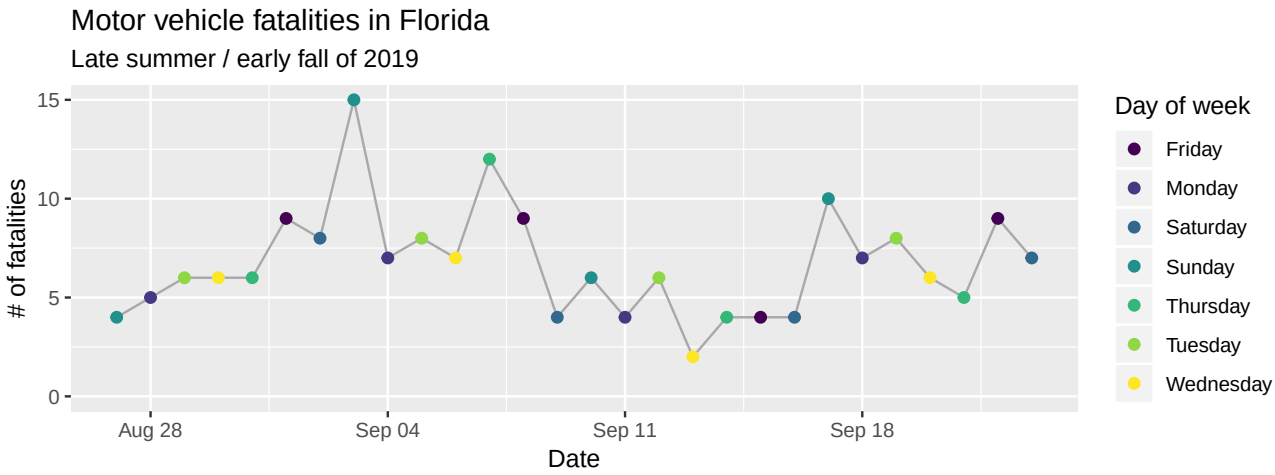


Figure 2.8: Step 6 of layering a plot: Customizing labels. The 'labs' layer customizes not only the x and y axis labels, but also the legend title for the color scale. The title and subtitle are added with a 'ggtitle' layer.

The last layer we need to add is a theme layer. While we're using most of the elements from the default theme (`theme_gray`), we do want to change the position of the legend. For a time series plot like this, the change can be helpful, as it lets us create a plot that's much wider than it is tall. You can move the legend using the `theme` layer with an argument specified for `legend.position`. With the `theme` function, you can customize almost any of the background elements of a plot.[12] However, you'll usually only want to do that for a few elements—if you want to change a lot of elements, there is a set of functions that start `theme_` that will let you change to one of several "themes" that change many elements at once (as I'll show in the next example). If you find you're often using `theme` to specify lots of elements by hand, you can create your own `theme_*` function (fill in * with the name of your choice!).

[12] See the helpfile for "theme" with `?theme` for a full listing.

```
ggplot(data = daily_fatalities) + geom_line(aes(x = date,
    y = fatals), color = "darkgray") + geom_point(aes(x = date,
    y = fatals, color = weekday), size = 2) +
    expand_limits(y = 0) + scale_color_viridis_d() +
    labs(x = "Date", y = "# of fatalities", color = "Day of week") +
    ggtitle("Motor vehicle fatalities in Florida",
        subtitle = "Late summer / early fall of 2019") +
    theme(legend.position = "bottom")
```

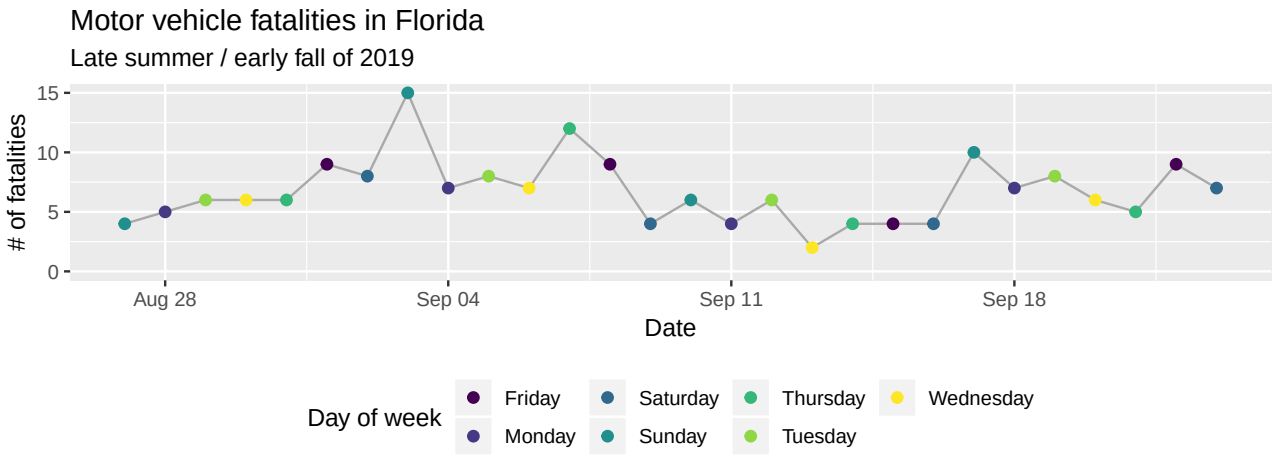


Figure 2.9: Step 7 of layering a plot: Customizing the theme. Move the color legend to below the plot.

As one final detail, if you were looking very closely, you may have noticed that the order of the days of week are different in the original plot from our final version here. That's because, by default, discrete character-class values are ordered alphabetically. We can change this order, and should, to be the order that week days occur. To do this, we need to change the column class to a **factor class**[13] and then change the order of that factor's levels.

To do this, we'll need to load a few additional R packages (you'll also use these in the "Tidy" section, which has much more on manipulating datasets). The **forcats** package (Wickham, 2019) has functions for working with factors, including a function we can use to reorder the factor levels. The **magrittr** package (Bache and Wickham, 2014) includes two infix functions we'll use to make the code cleaner: the pipe operator (%>%) and the compound assignment pipe operator (%<>%). The **dplyr** package (Wickham et al., 2019) includes a number of simple but powerful functions for manipulating tibbles.

The "Tidy" section will go into detail for how to use all these functions. As a brief summary, the `mutate` function used twice to change the values in the "weekday" column: first, to convert the class of the column to a factor (`as_factor`) and second to change the order of those levels (`fct_relevel`) by hand, to start with Monday and go in order through Sunday. The compound assignment pipe operator (%<>%) allows us to perform all those operations to columns inside the "daily_fatalities" tibble, and then to save the result *back to the same R object* (overwriting the earlier version of the tibble).

[13] **factor class.** An R class for values that take character names, but that describe categories, for which you expect values to show up more than once in your data. In this data, the day of week is an example, since it is expressed as a character variable ("Monday", "Tuesday", etc.), but we expect there to be multiple observations with, e.g., "Monday". If you have a varible that you expect to be unique (e.g., name of study subjects, unique ID number), the column should be in a character, not a factor, class. In R, each possible catogory for a factor is called a **level**. You can change the order of the levels of a factor, and this will change the order they're shown on a plot.

```
library("forcats")
library("magrittr")
library("dplyr")

daily_fatalities %<>% mutate(weekday = as_factor(weekday)) %>%
    mutate(weekday = fct_relevel(weekday, "Sunday",
```

```
        "Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday"))
```

Now when we run the same `ggplot` code, you can see that the order of the days of week in the color legend, and the order the colors are assigned to week day, have changed (Figure 2.10).

```
ggplot(data = daily_fatalities) + geom_line(aes(x = date,
    y = fatals), color = "darkgray") + geom_point(aes(x = date,
    y = fatals, color = weekday), size = 2) +
    expand_limits(y = 0) + scale_color_viridis_d() +
    labs(x = "Date", y = "# of fatalities", color = "Day of week") +
    ggtitle("Motor vehicle fatalities in Florida",
        subtitle = "Late summer / early fall of 2019") +
    theme(legend.position = "bottom")
```

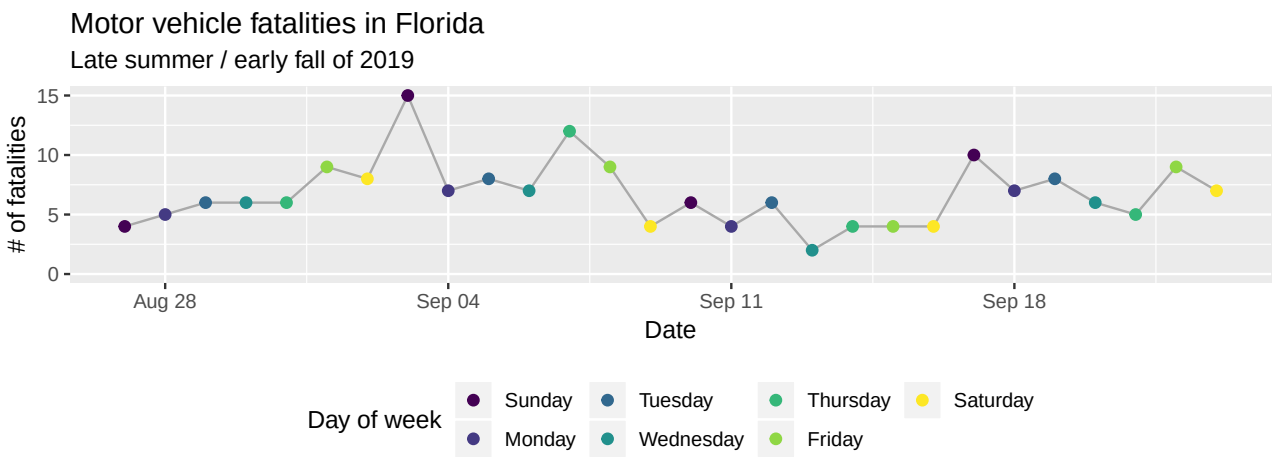


Figure 2.10: Change the order of the days of week from alphabetical to temporal by transforming the data before plotting.

The second example plot we can go through much more quickly. There are two elements I'd like to point out: the layer for faceting and the layer for the theme. The `facet_wrap` function separates the plot into **small multiples** based on the values in a column of the data. By default, all the scales (e.g., x-axis, y-axis) will be the same across all the small multiples, allowing for an easier comparison across the plots. The other layers in this code should now be somewhat familiar to you: the `ggplot` call initializes a `ggplot` object with the "daily_fatalities" dataset, just like in the last plot, while the `geom_col` layer adds a column geom where the x aesthetic is mapped to the value in the "weekday" column and the y aesthetic is mapped to the value in the "fatals" column.

To change to the `theme_classic`, all you need to do is add a `theme_classic` layer (Figure 2.2.2).

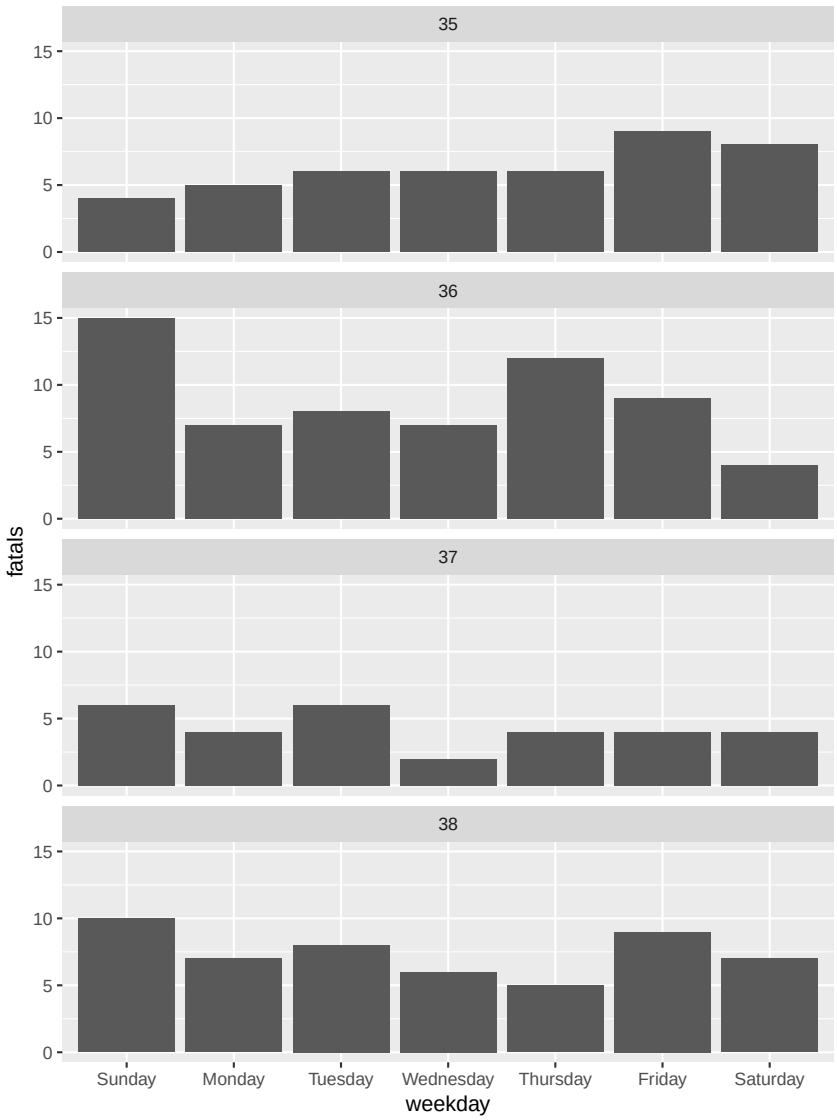
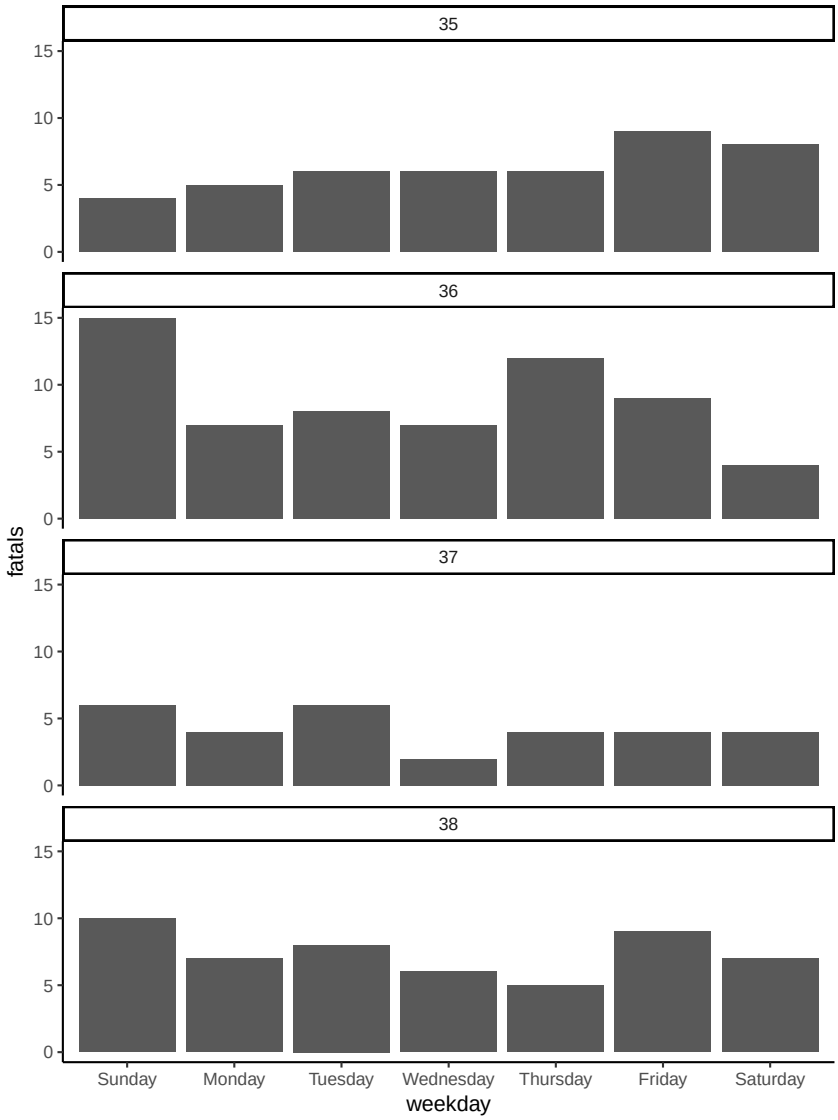


Figure 2.11: Adding a facet layer to a plot. The plot is now faceted by the 'week' column in the dataframe, with all the facets lined up vertically in a single column ('ncol = 1').



\begin{figure}
\caption[Changing the theme]{Changing the theme. The 'theme_*' family of functions can quickly change many of the background elements of a plot with a single layer call.} \end{figure}

Most of the rest of the layers for the plot are very similar to the first plot. They include changing the **fill**[14] of the geom by mapping it to a constant fill aesthetic (`color = "skyblue"` in the `geom_col` call), customizing the labels with a `lab` layer, and adding a title with `ggtitle`. The result of adding these layers is shown in Figure 2.12

[14] **fill aesthetic.** While all geoms have a **color aesthetic**, some also have a **fill** aesthetic. These include columns, bars, polygons, and points with certain shapes. In these cases, 'color' will specify the outline of the geom while 'fill' will specify the inside. Each geom has a set of required aesthetics and a set of possible aesthetics. To find each set for a geom, go to its helpfile (e.g., `?geom_col`) and scroll down to the "Aesthetics" section.

```
ggplot(data = daily_fatalities) + geom_col(aes(x = weekday,
    y = fatals), fill = "skyblue") + facet_wrap(~week,
    ncol = 1) + theme_classic() + labs(x = "",
    y = "# of fatalities") + ggtitle("Motor vehicle fatalities in Florida by weekday, 2019")
```

As with the first graph, to take the plot to its final stage, you'll need to

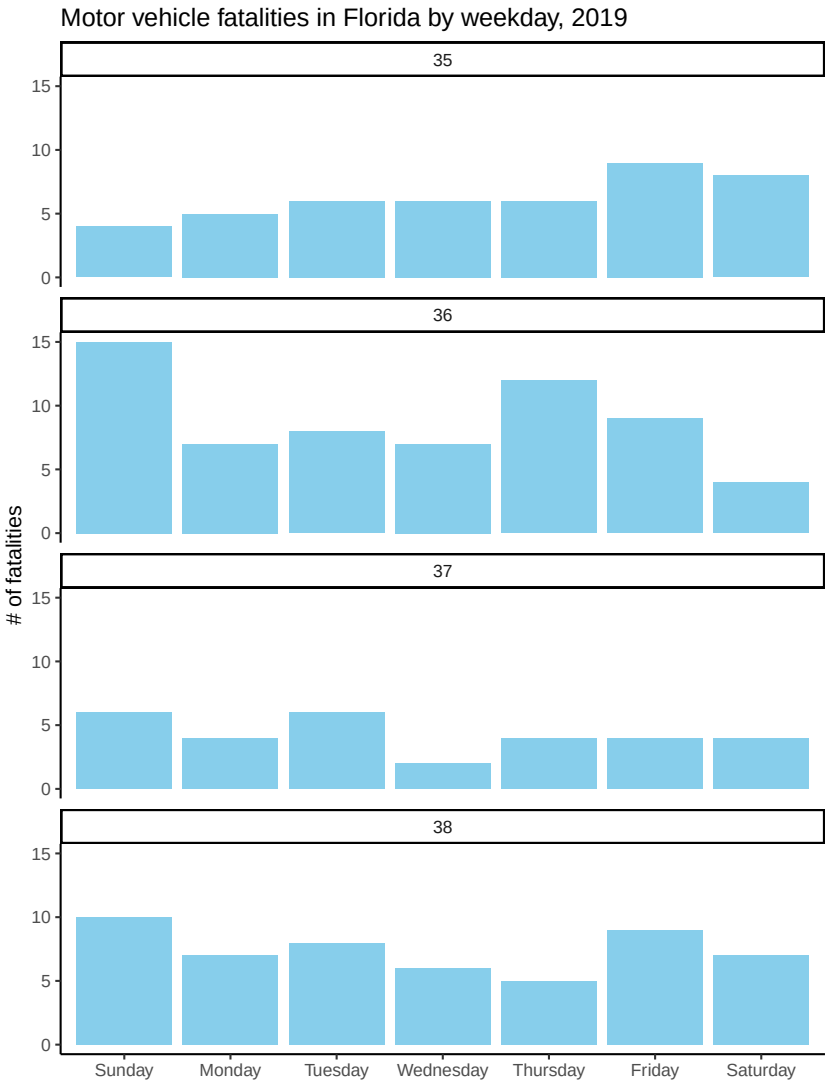


Figure 2.12: Customizing the labeling and adding a constant fill aesthetic. Note that the constant fill aesthetic is specified outside the 'aes' call for the geom, and that the color is specified inside quotation marks.

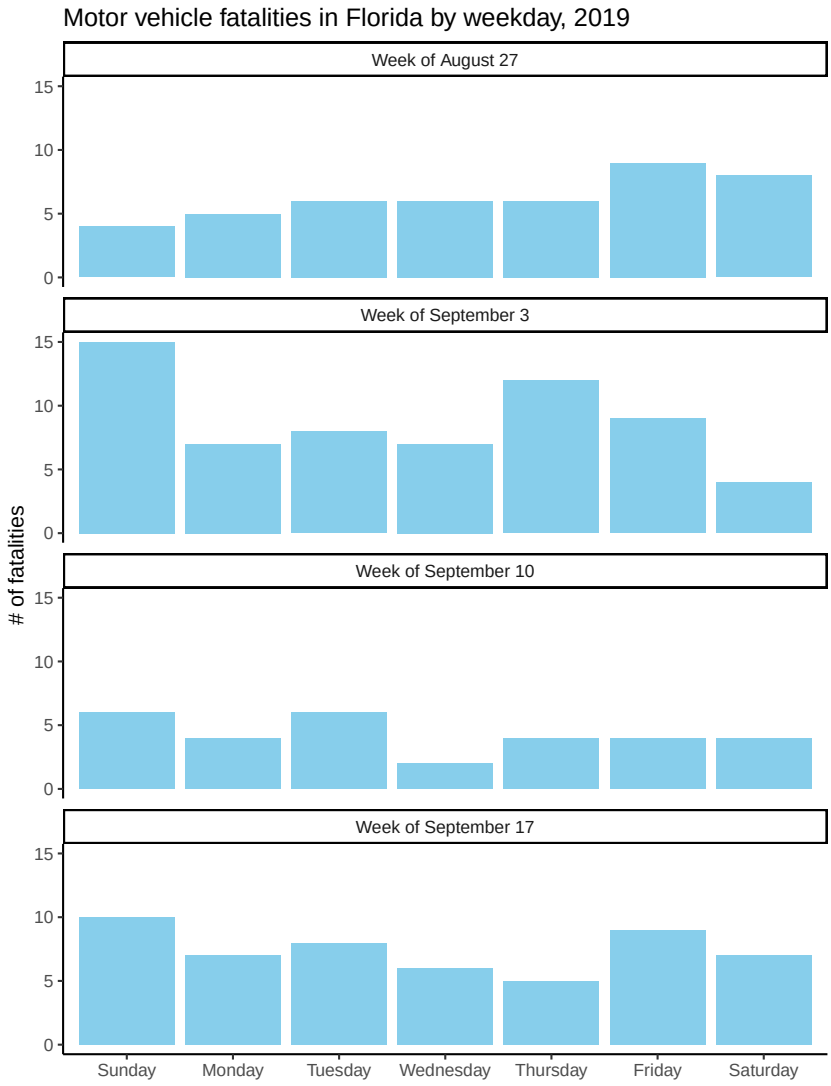
change the input data a bit. In this case, this step is to create clearer labels for the facets. The week number (e.g., 35) won't mean much to most viewers. A label giving the date of the first day in the week ("Week of August 27") will be more helpful. To create these new labels, you can use `group_by`, `mutate`, and `first` functions from the `dplyr` package (which you have already loaded if you've worked through the examples in order) to create a column called 'first_day' with the date of the first day in each week. Then you can use `mutate` (also from `dplyr`) to create the 'week_label' column from the existing columns of the tibble. This code uses the `month` and `day` functions from the `lubridate` package (very useful for working with dates) (Spinu et al., 2018) to extract the month and day from the 'first_day' dates, and then `paste` (from base R) pastes these together with "Week of". Finally, the `as_factor` and `fct_reorder` calls (both from `forcats`, which you also should have loaded if you've followed the examples in order) get this new column in the right order, so that the facets show up in temporal order rather than alphabetical.

Once you've made these changes to the 'daily_fatalities' tibble, you should get the final version of the plot when you re-run the plotting code used in the last step, getting Figure 2.2.2.

```
library(lubridate)

daily_fatalities %<>% group_by(week) %>% mutate(first_day = first(date)) %>%
    ungroup() %>% mutate(week_label = paste("Week of",
    month(first_day, label = TRUE, abbr = FALSE),
    day(first_day))) %>% mutate(week_label = as_factor(week_label),
    week_label = fct_reorder(week_label, week,
        .fun = min))

ggplot(data = daily_fatalities) + geom_col(aes(x = weekday,
    y = fatals), fill = "skyblue") + facet_wrap(~week_label,
    ncol = 1) + labs(x = "", y = "# of fatalities") +
    theme_classic() + ggtitle("Motor vehicle fatalities in Florida by weekday, 2019")
```



\begin{figure}
\caption[To create better facet labels, you can first make some changes to the dataset and then facet by the newly created 'week_label' column rather than the 'week' column]{To create better facet labels, you can first make some changes to the dataset and then facet by the newly created 'week_label' column rather than the 'week' column.} \end{figure}

2.3 Saving plots

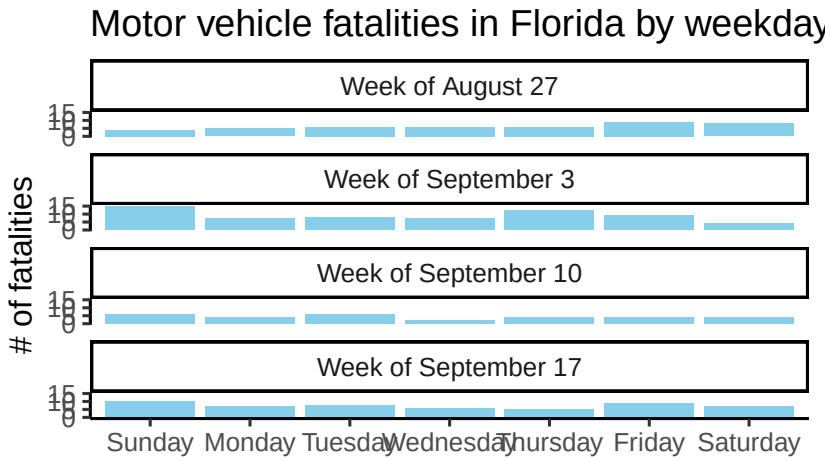
You can assign the output of a `ggplot` call to an R object using the gets arrow (<−), just like you can assign data you've read in to an R object. If you do this, the plot won't print immediately after you run the code:

```
irma_fatalities_plot <- ggplot(data = daily_fatalities) +
    geom_col(aes(x = weekday, y = fatals), fill = "skyblue") +
    facet_wrap(~week_label, ncol = 1) + labs(x = "",
```

```
    y = "# of fatalities") + theme_classic() +
    ggtitle("Motor vehicle fatalities in Florida by weekday, 2019")
```

However, now you can print the plot anytime you want by calling the object:

```
irma_fatalities_plot
```



To save this plot to a file, you can use the `ggsave` function. This function can save the plot in several different formats (e.g., pdf, jpeg, png, svg). Further, it allows you to specify the size you'd like for the height and width of the plot as well as the plot resolution.

The following call will save the `irma_fatalities_plot` to your working directory, which should be the R project directory you're using for the examples if you followed the set-up in "Prerequisites". It will save the file as a pdf that is 6 inches tall by 5 inches wide.

```
ggsave(irma_fatalities_plot, filename = "irma_fatalities.pdf",
    device = "pdf", height = 6, width = 5, units = "in")
```

2.4 Learn more

The `ggplot` framework has become extremely popular, and there are a lot of excellent resources for learning more about how to use it. Many of these are through the website bookdown.org, which hosts a collection of free, author-submitted online books, mostly about R programming.

R for Data Science is a global look at effectively using R's "tidyverse", including sections on plotting with `ggplot2`. Kiernan Healy's Data Visualization: A Practical Introduction is a great book on using R for plotting, with extensive examples in R. This book also covers a lot of the principles of creating effective and attractive plots—it's well worth reading.

If you want to dig deeper into plotting in R using the `ggplot` framework, you might want to take a look at the official `ggplot2` book, ggplot2: Elegant Graphics for Data Analysis and at Paul Murrell's book R Graphics, with extensive coverage of the grid graphics system that `ggplot2` builds on.

Finally, one of the best ways to learn more is to check out RStudio's "Data Visualization" cheatsheet,[15] Once you've started to get the hang of the basics of plotting in R, download this cheatsheet and work through the examples. All of them use datasets that come with R, so you should be able to run them all, and you'll get a good idea of the range of plots `ggplot2` can be used to create.

[15] **cheatsheet.** A two-sided, one-page sheet crammed with all the main functions for a particular topic of coding. RStudio has a large collection available here, including cheatsheets for cleaning data, working with factors and strings, and a range of other topics. This webpage also includes "contributed" cheatsheets, developed by people outside of RStudio.

3
Map

R has had tools for mapping spatial data for a long time, but some of these tools, but they could take a while to learn if you were just used to basic plotting in R. Recently, some packages have been developed for mapping spatial data that fit within the `ggplot` framework, so they allow you to take what you've learned about creating non-geographical plots and apply them to create maps.

The `sf` package (Pebesma, 2019) is a fantastic new(-ish) package for mapping in R. The "Tidy" section of this handout describes the "tidy" data framework used as an input for `ggplot` code. The "tidy" data framework is also convenient for cleaning, merging, and manipulating data before plotting or modeling it. The `sf` package allows you to read in and work with geographical data in a tidy format. It turns out that this is very powerful, as you can learn how to do a few things well (plotting [see the "Plot" section] and working with data [see the "Tidy" section]), and then apply these tools in the same way, whether you're working with geographical or non-geographical data.

3.1 Geographical data in a tidy format

The `sf` package allows you to create a dataframe object with one special characteristic: a special **list column**[1] called "geometry" that contains the geometrical data needed to draw an observation. For example, if you have a dataset of motor vehicle accidents, where each row gives the data for an accident, the `geometry` column might include the latitude and longitude for the location of the accident. As another example, if you have a data set of the total number of motor vehicle fatalities in a year in each county in a state, the `geometry` column might have all the latitude and longitude points to form the boundary of each county.

[1] **list column**

These special dataframes are given a class called **sf**[2]

There are several ways for you to create this special type of dataframe. We'll create a few to use in the later mapping examples. First, if you have a regular dataframe, you can convert it into an `sf` object, specifying which parts of the dataframe include geographical information.

[2] **sf.** Short for "simple features", the name of both an R package and the object class created and used by the data. This class includes a special column called "geometry" for geographic information about each observation. Objects with this class can be used for mapping spatial data, but also can be manipulated using tidyverse tools very similarly to tibbles.

If you followed all the set-up instructions in the "Prerequisites", you should have downloaded a dataset called "fl_accidents.csv" and saved it in the "data" subdirectory of the R Project directory for the examples. You can use `readr` to read it in. If you print out the start of it, you'll see that it's got observations (rows) of fatal motor vehicle accidents. These accidents all occurred in Florida within a week of Hurricane Irma's landfall on September 10, 2017. The columns give the county code (`fips`), date (`date`), location (`latitude` and `longitud`), and the number of fatalities (`fatals`).

```
library(readr)
fl_accidents <- read_csv("data/fl_accidents.csv")
```

```
## Parsed with column specification:
## cols(
##   fips = col_double(),
##   date = col_date(format = ""),
##   latitude = col_double(),
##   longitud = col_double(),
##   fatals = col_double()
## )
```

```
fl_accidents
```

```
## # A tibble: 37 x 5
##     fips date       latitude longitud fatals
##    <dbl> <date>        <dbl>    <dbl>  <dbl>
##  1 12031 2017-09-08     30.2    -81.5      1
##  2 12095 2017-09-07     28.5    -81.4      1
##  3 12097 2017-09-08     28.3    -81.3      1
##  4 12095 2017-09-07     28.6    -81.2      1
##  5 12031 2017-09-08     30.2    -81.8      1
##  6 12033 2017-09-07     30.6    -87.4      2
##  7 12023 2017-09-10     30.1    -82.7      1
##  8 12075 2017-09-08     29.6    -82.9      1
##  9 12045 2017-09-09     30.1    -85.3      2
## 10 12031 2017-09-12     30.4    -81.8      1
## # ... with 27 more rows
```

Even though this data has geographical information in it (latitude and longitude), it's currently just a regular dataframe. To convert it to an `sf` class object, you can use the `st_as_sf` function,[3] specifying the columns with the geogrphical coordinates using the `coords` parameter.[4]

```
library(sf)
fl_accidents %<>% st_as_sf(coords = c("longitud",
    "latitude"))
```

[3] Most of the functions in the `st` package start with `st_`. Since R studio allows **tab completion**, this makes it very easy to look up a function in the package whose name you might have forgotten. Just try typing `st_` and then the Tab key in your R console—you should get a pop-up with a list of suggestions for possible function names.

[4] Here, the compound pipe operator, `%<>%`, applies the function to `fl_accidents` and then overwrites the `fl_accidents` with the modified version, updating the object so you're ready to use it for later code.

Now, when you print out `fl_accidents`, you'll see some extra information at the top of the print out, including the objects **bounding box** (`bbox`) and **projection** (`epsg`, `proj4string`) (which we currently haven't set).

```
fl_accidents
```

```
## Simple feature collection with 37 features and 3 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: -87.3797 ymin: 25.6876 xmax: -80.32332 ymax: 30.65894
## epsg (SRID):    NA
## proj4string:    NA
## # A tibble: 37 x 4
##     fips date       fatals
##    <dbl> <date>      <dbl>
##  1 12031 2017-09-08      1
##  2 12095 2017-09-07      1
##  3 12097 2017-09-08      1
##  4 12095 2017-09-07      1
##  5 12031 2017-09-08      1
##  6 12033 2017-09-07      2
##  7 12023 2017-09-10      1
##  8 12075 2017-09-08      1
##  9 12045 2017-09-09      2
## 10 12031 2017-09-12      1
## # ... with 27 more rows, and 1 more
## #   variable: geometry <POINT>
```

A second way to create an `sf` object in R is to read in data from a geographic data file, like a **shapefile**[5] If you followed the "Prerequisites", you should have downloaded a file called "al112017_best_track.zip". This is a zipped file that includes shapefiles for the track of Hurricane Irma from the National Hurricane Center ([website]).

[5] **shapefile.** …

First, you'll need to unzip this file to "unpack" it as a directory of files. On many computers, you can do this by double-clicking on the zipped file. However, you can also unzip files from R. Just run:

```
unzip("data/al112017_best_track.zip", exdir = "data/al112017_best_track")
```

Once the file is unzipped, if you look at the "al112017_best_track" directory created, you will see that it contains a collection of files starting with one of several roots (e.g., "al112017_lin") and with one of several suffixes (e.g., ".dbf", ".prj", ".shx").

```
list.files("data/al112017_best_track/")
```

```
##  [1] "al112017_lin.dbf"
##  [2] "al112017_lin.prj"
##  [3] "al112017_lin.shp"
##  [4] "al112017_lin.shp.xml"
##  [5] "al112017_lin.shx"
##  [6] "al112017_pts.dbf"
##  [7] "al112017_pts.prj"
##  [8] "al112017_pts.shp"
##  [9] "al112017_pts.shp.xml"
## [10] "al112017_pts.shx"
## [11] "al112017_radii.dbf"
## [12] "al112017_radii.prj"
## [13] "al112017_radii.shp"
## [14] "al112017_radii.shp.xml"
## [15] "al112017_radii.shx"
## [16] "al112017_windswath.dbf"
## [17] "al112017_windswath.prj"
## [18] "al112017_windswath.shp"
## [19] "al112017_windswath.shp.xml"
## [20] "al112017_windswath.shx"
```

Each set of files with the same root provides a "layer" of the shapefile, giving a set of geographic information. For example, the "al112017_lin." *files provides the line of the hurricane's track, while the "al112017_windswarth."* layer provides windswaths (how severe the wind was in certain locations surrounding the storm track).

You can read in any of these layers into R as an `sf` object using `irma_tracks` and specifying the layer you'd like from the directory with the `layer` parameter (put the root of the filenames for the layer you want). For example, to read in a line with the track of irma, you can run:

```
irma_tracks <- st_read("data/al112017_best_track/",
    layer = "al112017_lin")
```

```
## Reading layer `al112017_lin' from data source `/Users/georgianaanderson/Documents/r_workshops/navy_pu
## Simple feature collection with 20 features and 3 fields
## geometry type:  LINESTRING
## dimension:      XY
## bbox:           xmin: -90.1 ymin: 16.1 xmax: -26.9 ymax: 36.8
## epsg (SRID):    NA
## proj4string:    +proj=longlat +a=6371200 +b=6371200 +no_defs
```

If you print out `irma_tracks`, you can see that it looks like a dataframe, but with extra geographic information (bounding box, projection, etc.) as well as a special column for geometry, which gives the latitude and longitude …

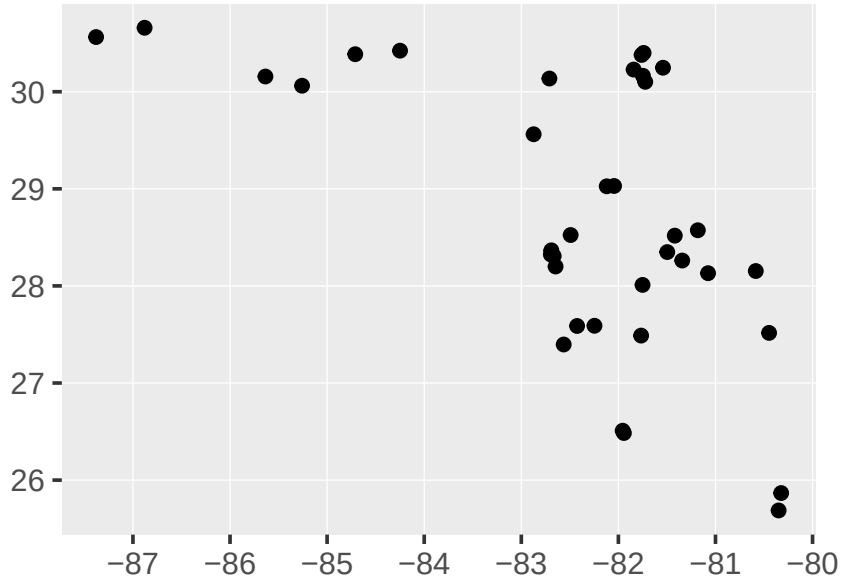
```
irma_tracks
```

```
## Simple feature collection with 20 features and 3 fields
## geometry type:  LINESTRING
## dimension:      XY
## bbox:           xmin: -90.1 ymin: 16.1 xmax: -26.9 ymax: 36.8
## epsg (SRID):    NA
## proj4string:    +proj=longlat +a=6371200 +b=6371200 +no_defs
## First 10 features:
##    STORMNUM           STORMTYPE SS
## 1        11 Tropical Depression  0
## 2        11      Tropical Storm  0
## 3        11           Hurricane1  1
## 4        11           Hurricane2  2
## 5        11           Hurricane3  3
## 6        11           Hurricane2  2
## 7        11           Hurricane3  3
## 8        11           Hurricane4  4
## 9        11           Hurricane5  5
## 10       11           Hurricane4  4
##                            geometry
## 1  LINESTRING (-26.9 16.1, -28...
## 2  LINESTRING (-28.3 16.2, -29...
## 3  LINESTRING (-32.5 16.4, -33...
## 4  LINESTRING (-34.2 17.1, -35...
## 5  LINESTRING (-35.1 17.5, -36...
## 6  LINESTRING (-42.6 18.9, -44...
## 7  LINESTRING (-47.9 17.9, -49...
## 8  LINESTRING (-53.9 16.7, -55...
## 9  LINESTRING (-57.8 16.7, -59...
## 10 LINESTRING (-73 21.5, -73.2...
```

3.2 Basic mapping

```
library(ggplot2)

ggplot() + geom_sf(data = fl_accidents)
```



3.3 Learn more

The `sf` package is rapidly developing, and so it is worthwhile to seek out the latest help guides and tutorials to learn more about the system, particularly if you are doing this a while after the spring 2019 workshop.

The package authors have created a website with more on the package. They also have a book called Geocomputation with R, which is currently available online and will be available in print sometime in 2019.

Both these sources go in depth to describe the `sf` package and how to use it. If you want other examples of using the package, many people have recently written blog posts with examples of using it, so it's worth googling something like "blog post map r sf".

4

Interact

4.1 DT: Datatables

```
library(readr)
library(dplyr)
library(sf)
fl_accidents <- read_csv("data/fl_accidents.csv")
```

```
library(DT)
datatable(fl_accidents)
```

Show 10 entries Search:

	fips	date	latitude	longitud	fatals
1	12031	2017-09-08	30.2474	-81.5407	1
2	12095	2017-09-07	28.51796389	-81.41926667	1
3	12097	2017-09-08	28.26148889	-81.34221111	1
4	12095	2017-09-07	28.573275	-81.18188056	1
5	12031	2017-09-08	30.22845833	-81.84293889	1
6	12033	2017-09-07	30.56341111	-87.3797	2
7	12023	2017-09-10	30.13608333	-82.71024444	1
8	12075	2017-09-08	29.56196111	-82.87318056	1
9	12045	2017-09-09	30.06197778	-85.25815	2
10	12031	2017-09-12	30.37913056	-81.76085556	1

Showing 1 to 10 of 37 entries Previous 1 2 3 4 Next

```
datatable(fl_accidents, class = "compact", caption = "Fatal motor vehicle accidents in Florida the week
    colnames = c("County FIPS", "Date", "Latitude",
        "Longitude", "# fatalities"))
```

Show 10 entries Search:

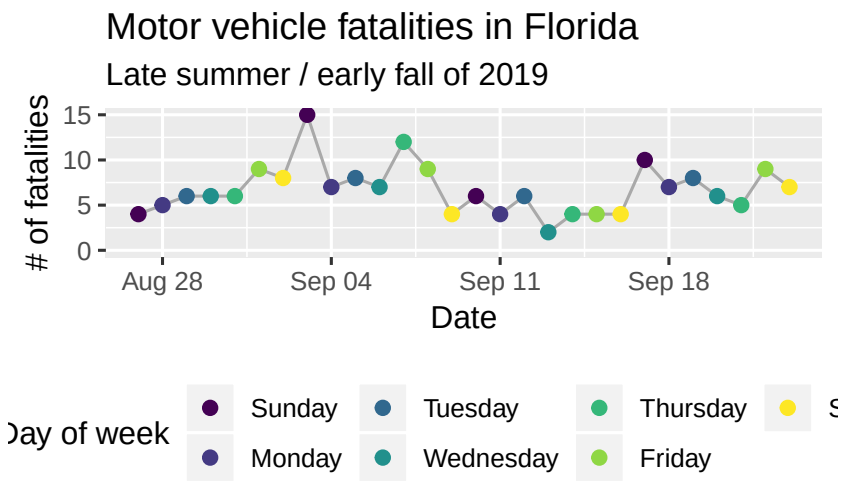
Fatal motor vehicle accidents in Florida the week of Hurricane Irma

	County FIPS	Date	Latitude	Longitude	# fatalities
1	12031	2017-09-08	30.2474	-81.5407	1
2	12095	2017-09-07	28.51796389	-81.41926667	1
3	12097	2017-09-08	28.26148889	-81.34221111	1
4	12095	2017-09-07	28.573275	-81.18188056	1
5	12031	2017-09-08	30.22845833	-81.84293889	1
6	12033	2017-09-07	30.56341111	-87.3797	2
7	12023	2017-09-10	30.13608333	-82.71024444	1
8	12075	2017-09-08	29.56196111	-82.87318056	1
9	12045	2017-09-09	30.06197778	-85.25815	2
10	12031	2017-09-12	30.37913056	-81.76085556	1

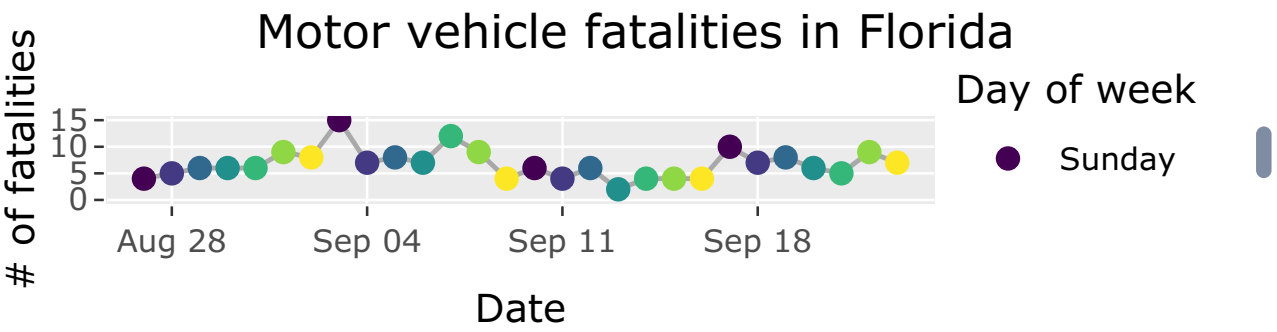
Showing 1 to 10 of 37 entries Previous 1 2 3 4 Next

4.2 Plotly

```
fatality_plot <- ggplot(data = daily_fatalities) +
    geom_line(aes(x = date, y = fatals), color = "darkgray") +
    geom_point(aes(x = date, y = fatals, color = weekday),
        size = 2) + expand_limits(y = 0) + scale_color_viridis_d() +
    labs(x = "Date", y = "# of fatalities", color = "Day of week") +
    ggtitle("Motor vehicle fatalities in Florida",
        subtitle = "Late summer / early fall of 2019") +
    theme(legend.position = "bottom")
fatality_plot
```



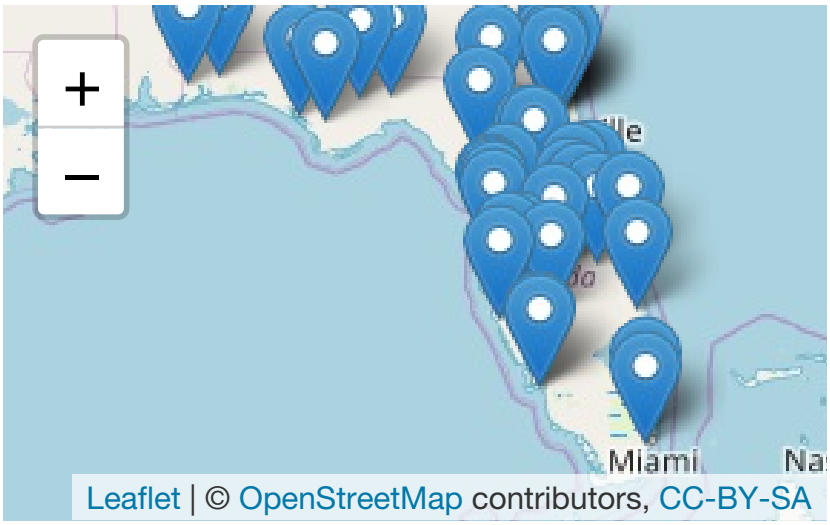

```
library(plotly)
fatality_plot %>% ggplotly()
```



4.3 Leaflet

```
library(magrittr)
library(leaflet)
fl_accidents %<>% st_as_sf(coords = c("longitud",
```

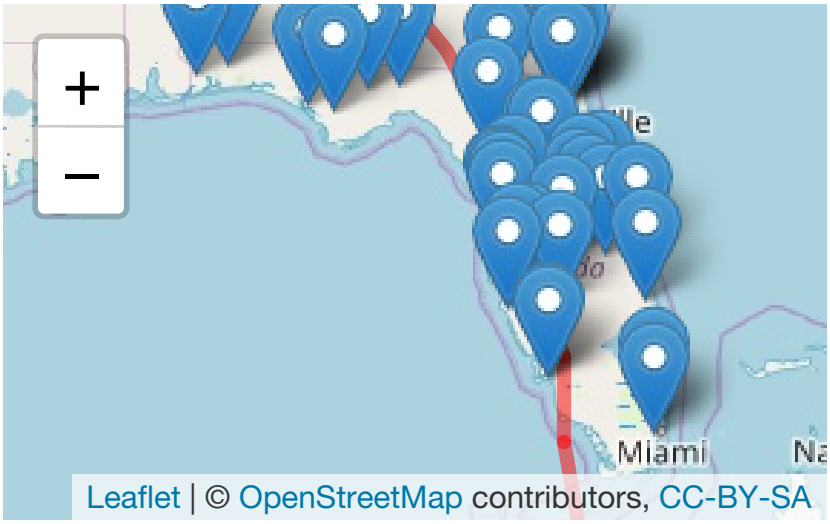
```
    "latitude")) %>% st_sf(crs = 4326)
leaflet() %>% addTiles() %>% addMarkers(data = fl_accidents)
```




```
library(purrr)
irma_track <- st_read("data/al112017_best_track",
    layer = "al112017_lin") %>% st_transform(crs = 4326)
```

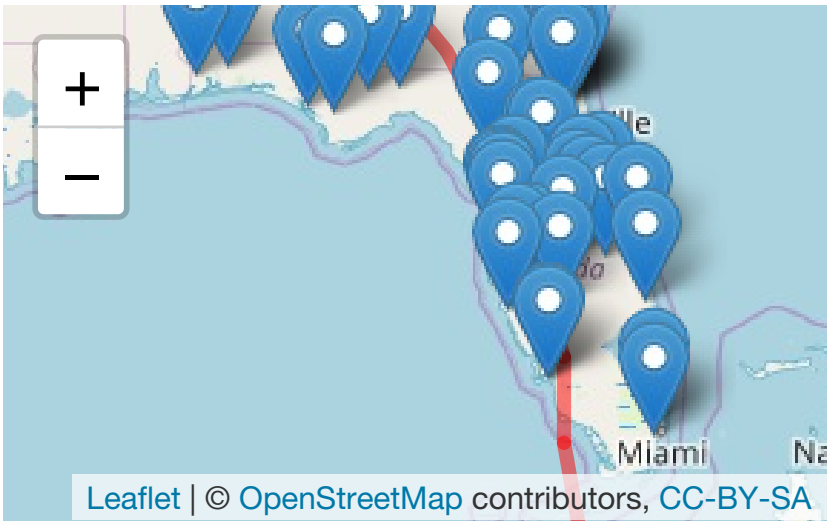
```
## Reading layer `al112017_lin' from data source `/Users/georgianaanderson/Documents/r_workshops/navy_pu
## Simple feature collection with 20 features and 3 fields
## geometry type:  LINESTRING
## dimension:      XY
## bbox:           xmin: -90.1 ymin: 16.1 xmax: -26.9 ymax: 36.8
## epsg (SRID):    NA
## proj4string:    +proj=longlat +a=6371200 +b=6371200 +no_defs
```

```
leaflet() %>% addTiles() %>% fitBounds(lng1 = -88,
    lng2 = -80, lat1 = 24.5, lat2 = 31.5) %>%
    addMarkers(data = fl_accidents, popup = ~date) %>%
    addPolylines(data = irma_track, color = "red")
```




```
library(htmltools)
fl_accidents %<>% mutate(popup = paste("<b>Date:</b>",
    date, "<br/>", "<b># fatalities:</b>", fatals))

leaflet() %>% addTiles() %>% fitBounds(lng1 = -88,
    lng2 = -80, lat1 = 24.5, lat2 = 31.5) %>%
    addMarkers(data = fl_accidents, popup = ~popup) %>%
    addPolylines(data = irma_track, color = "red")
```



4.4 Learn more

5
Report

5.1 RMarkdown

5.2 Dashboards

6
Tidy

The **ggplot2** framework is a very efficient and powerful framework for creating visualizations. This comes in part from the fact that it sticks to a specific data format for its input. It requires you to start with data in a what's called a "tidy" format.

In the previous section, I used an example dataset that was already in this format, to make it easier for you to get started with plotting. However, to leverage the power of **ggplot2** for real datasets, you have to know how to get them into this tidy format. This section will explain this format, as well as how you can clean real datasets to convert them into this format.

6.1 Tidy format

When you hear the term "tidy data", you might just think of a "clean" dataset—one where unneeded columns have been removed, for example, and perhaps that's in a nice **rectangular format**.[1]

[More about tidy data]

[1] **rectangular format.** A data format where each column has data in the same class (e.g., date, number) and is the same length. (Think of the format of data in a table of a paper, or in a clean Excel spreadsheet.

6.2 Tidyverse tools

As an epidemiologist, I meet many people who learned SAS as students and continue to use it. A common misperception is that R is good for visualizations, but bad for cleaning data. While in the past this might have been (somewhat) valid, now it couldn't be further from the truth. With a collection of tools available through the **tidyverse**,[2] you can write clean and compact code to clean even very large and messy datasets.

This collection of tools follows the principle that hard problems can be best solved by small tools that can be combined together, very much in-line with the **Unix philosophy**.

[2] **tidyverse.** A collection of packages to work with data in a "tidy" format, or to convert it to that format if needed. Many of these packages are developed and maintained by people at RStudio. If you run `library("tidyverse")`, you can load the core tidyverse packages in your R session. This way, you avoid having to load them one by one.

The tidyverse works as well as it does because, for many parts of it, it requires a common input and output, and those input and output specifications are identical (the tidy data format). If you want to get a better idea of this concept, and why it's so powerful, think of some of the classic toys, like Legos (train sets and Lincoln logs also work here). Each piece takes the same input and produces the same output. Think of the bottom of a Lego—it "inputs" small, regularly-spaced pegs, which are exactly what's at the top ("output") of each Lego block. This common input and output means that the blocks can be joined together in an extraordinary number of different combinations, and that you can imagine and then make very complex structures with the blocks.

6.3 Subsetting

6.3.1 Selecting columns

6.3.2 Slicing rows

6.3.3 Filtering rows

6.4 Adding

6.5 Summarizing

6.6 Reformatting

7
Final Words

We have finished a nice book.

8

Bibliography

Bache, S. M. and Wickham, H. (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5.

Pebesma, E. (2019). *sf: Simple Features for R*. R package version 0.7-3.

Spinu, V., Grolemund, G., and Wickham, H. (2018). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.7.4.

Wickham, H. (2019). *forcats: Tools for Working with Categorical Variables (Factors)*. R package version 0.4.0.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., and Woo, K. (2018a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.1.0.

Wickham, H., François, R., Henry, L., and Müller, K. (2019). *dplyr: A Grammar of Data Manipulation*. R package version 0.8.0.1.

Wickham, H., Hester, J., and Francois, R. (2018b). *readr: Read Rectangular Text Data*. R package version 1.3.1.