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. You can download the slides from the workshop by clicking here.

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

1. Download R
2. Download RStudio
3. Install some R packages
4. Download example R Project

This section will walk you through each step, explaining both *how* and *why* to do the step.

1. Download R

R is free and open-source software. You can download a copy for just about any operating system at the]Comprehensive R Archive Network (CRAN)] (`https://cran.r-project.org/`). Look for the links on this page to "Download for …" and your operating system.[1]

2. Download RStudio

You do not have to have RStudio[2] to run R, but it makes the experience much nicer. I also think it's much easier to learn R when using RStudio to run it rather than using R by itself.

RStudio is an **Integrated Development Environment (IDE)**[3] You should download it after you have downloaded R. You can download it here. You want the Desktop, open source edition. Download this and install it on your computer as you would any other software package from a website.

3. Install some R packages

[1] If your computer has restrictions on what you can download, you may need to take to your system administrator or IT team to discuss whether you can download and install R and, if so, how. In that case, you may also need to talk with them about a strategy for installing R packages, as well.

[2] In this section, I'm referring to RStudio, the software. It is produced by a company that is also called "RStudio". RStudio, the company, includes a team of some of the best and most prolific R programmers currently contributing packages to extend R. A lot of the packages covered in this booklet were created and are maintained by members of the RStudio team.

[3] **Integrated Development Environment (IDE).** A program that provides a helpful interface for developing code, often geared to a specific programming language. It typically includes tools and a visual set-up for the programmer's convenience.

When you downloaded the R software from CRAN, you downloaded what's called **base R**. This is the "engine" to run the heart of R, plus a few key extension packages. There are now several thousand packages available for R. These packages extend the core functionality of R, and we'll be using a lot of these extra packages in this booklet.

When you need to use an R package that did not come with base R (most of them), you will first need to **install** it to your computer. This can be done from R with the `install.packages` function. This installs a package from an online repository to your computer, saving the code in a special location that R can find. Since this function is downloading code from online, your computer needs to have an internet connection when you run `install.packages`, or you'll get an error.

This booklet uses a number of R packages beyond base R. To install all the packages that you'll need, run the following code in your version of R:

```
install.packages()
```

4. Download example R Project

I've created a repository on **GitHub**[4] You can find this example repository by clicking here. On the page takes you to, click on the "Clone or download" button and then select "Download ZIP".

This will download a single zipped file to your computer. When you unzip the file, it will be a special type of directory, an R Project directory. To open the R Project and start on the examples, open RStudio, then go to "File" -> "Open Project". A pop-up window will open to let you navigate through your files and find an R Project to open. Navigate to the directory you downloaded, which should be called "navy_public_health_examples" and doubleclick on the file in this directory called "navy_public_health_examples.Rproj".

This will open the project. In the "Files" pane of RStudio, you should see some subdirectories for "R" and "data". These have the example R code and data, respectively, for you to try the examples in this booklet. The code in each of the R files should run independently, including the code to load all required packages. Figure 1.1 shows what this package should look like once you've downloaded and opened it, as well as opened the "plot.R" file in the project's "R" subdirectory.

Click on the **Next** button (or navigate using the links at the top of the page) to continue.

[4] **GitHub.** An online platform for directories tracked with the version control software `git`. It has become a very popular place for coders to post code from their projects, especially for open-source projects like R packages. You may need to get a GitHub account to be able to clone this example directory. Even if not, it's not a bad idea to get an account if you work a lot with data and open-source code.

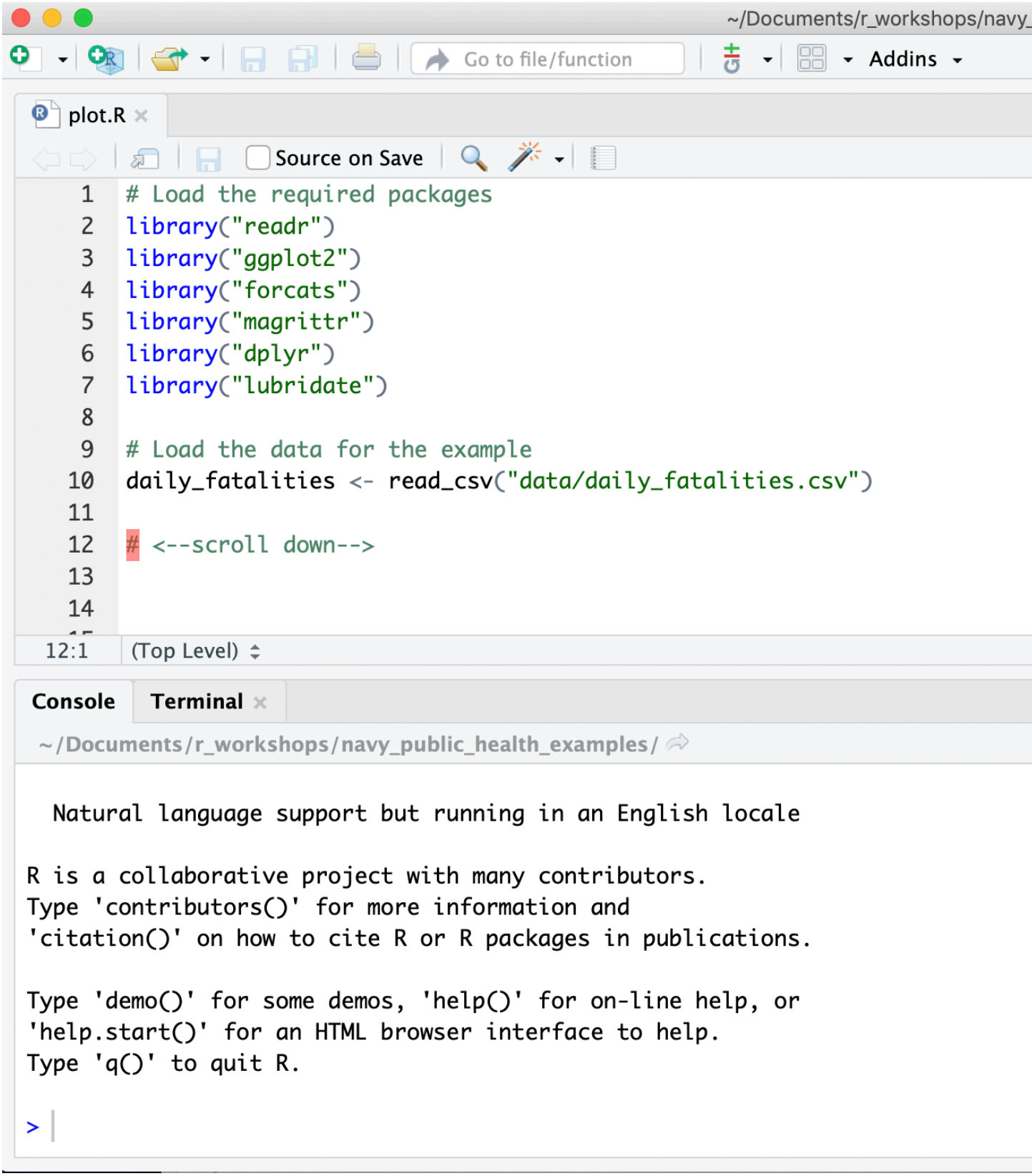


Figure 1.1: What the example R Project for this booklet should look like once you've downloaded and opened it, as well as opened the 'plot.R' file in the project's 'R' subdirectory.

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.

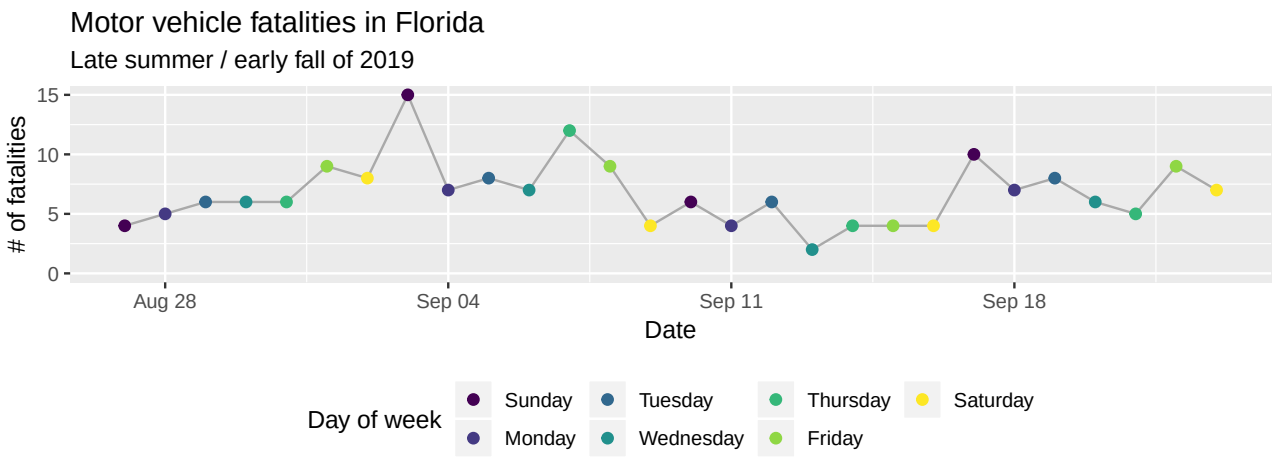


Figure 2.1: Number of motor vehicle fatalities by day in Florida in the weeks surrounding Hurricane Irma on September 10, 2019.

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.

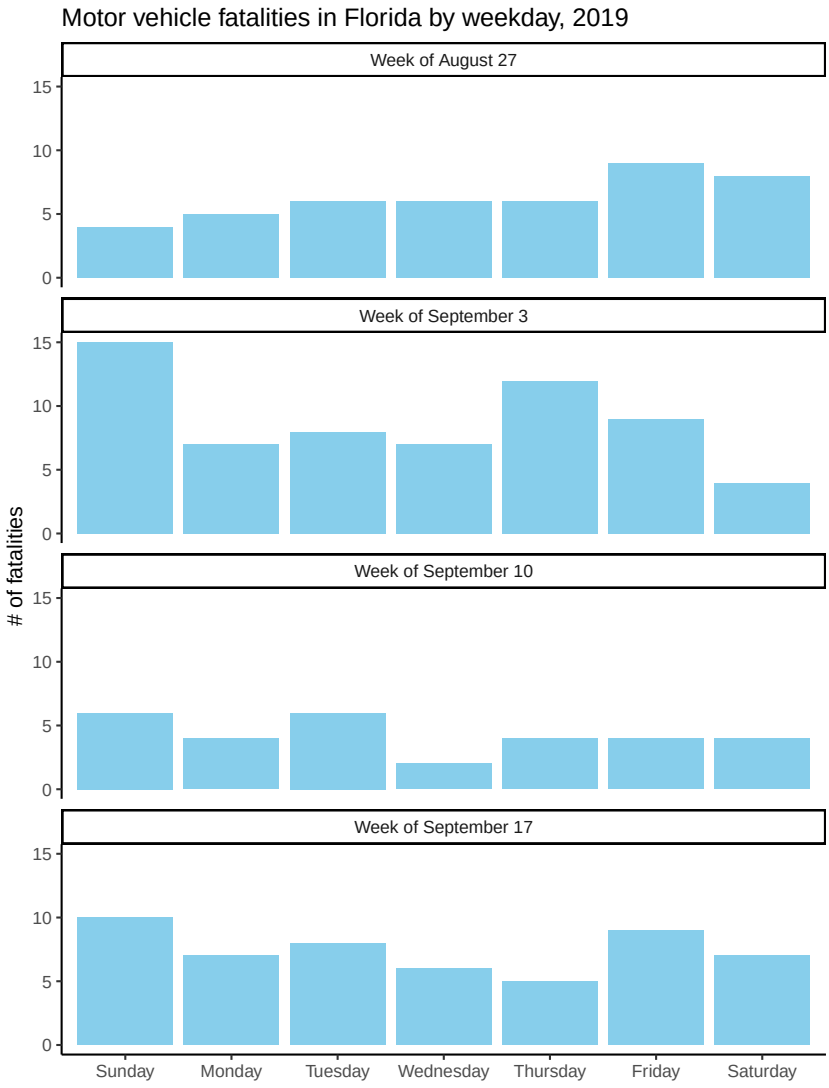


Figure 2.2: Number of motor vehicle fatalities in Florida by day of the week and week for the weeks surrounding Hurricane Irma's landfall.

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

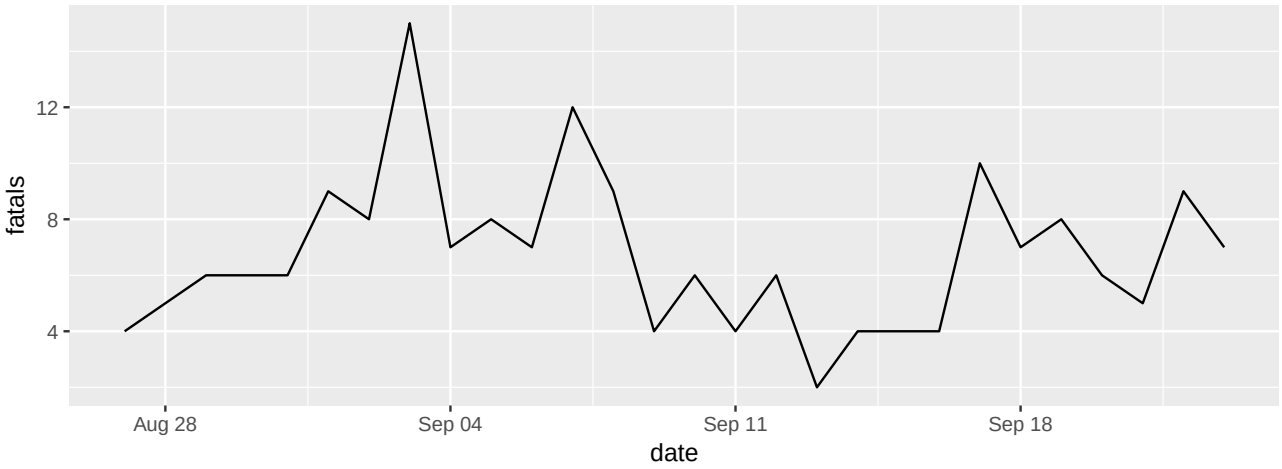


Figure 2.4: Step 2 of layering a plot: Adding a line geom. In this case, the x-axis (x aesthetic) is mapped to the 'date' column in the data, while the y-axis (y aesthetic) is mapped to the 'fatals' column.

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))
```

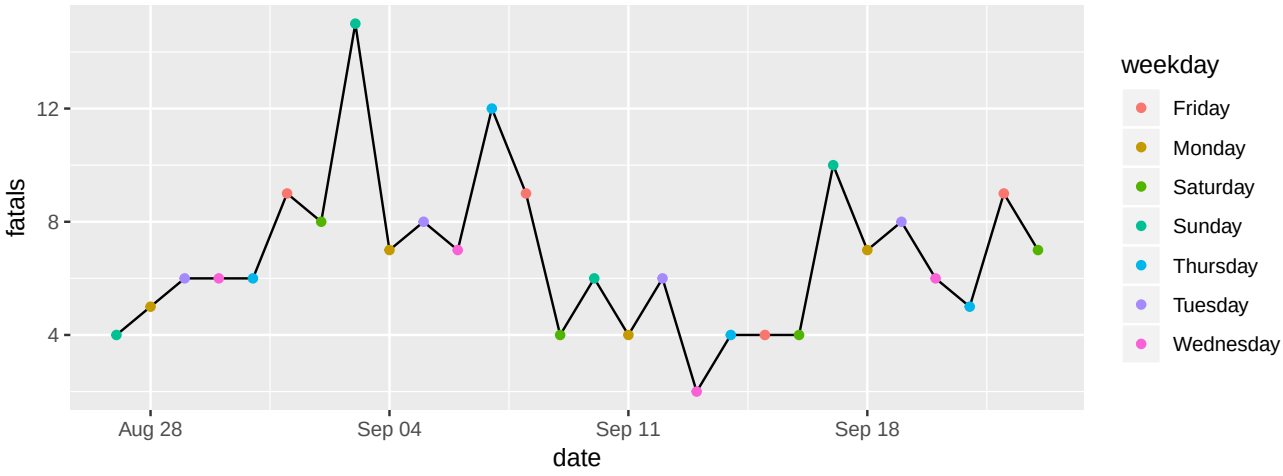


Figure 2.5: Step 3 of layering a plot: Adding a point geom. The x and y aesthetics are the same as for the line geom, but now we're also mapping color to the 'weekday' column in the data.

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)
```

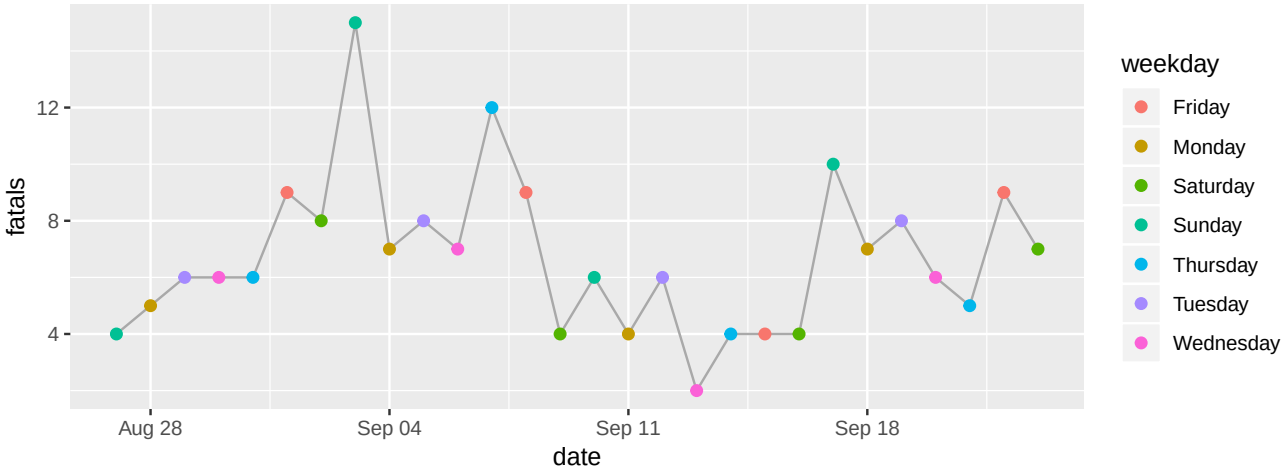


Figure 2.6: Step 4 of layering a plot: Adding constant aesthetics. In this step, we're making the line dark gray for all observations and the points a bit larger. Note that these aesthetics, since they're constant, are set outside of the 'aes' call. Also, note that thee color is specified inside quotation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

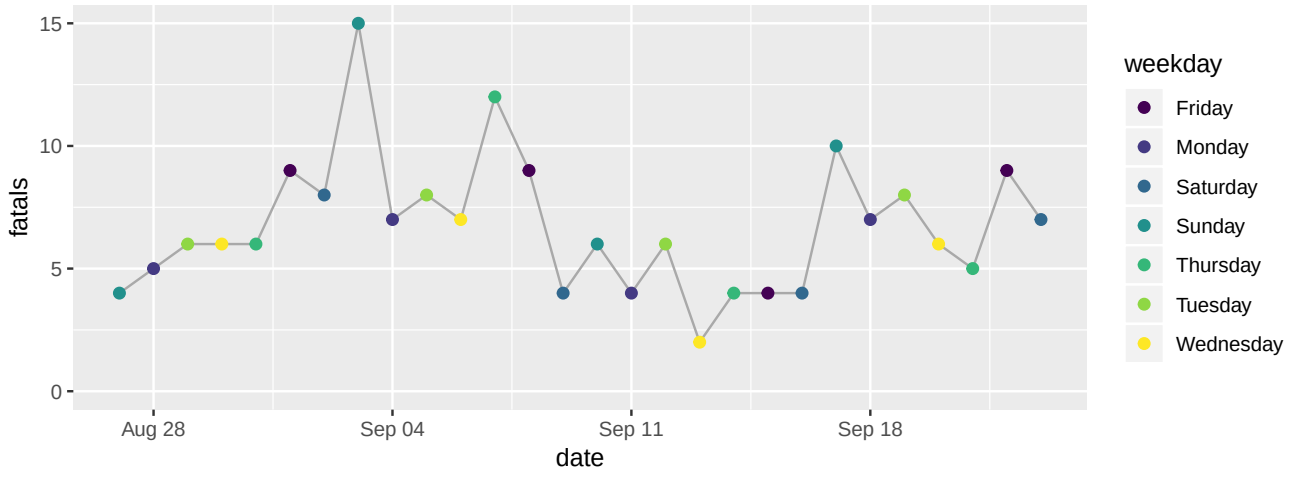


Figure 2.7: Step 5 of layering a plot: Changing the color scale. This step changes from the default color scale to the 'viridis' color scale.

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")
```

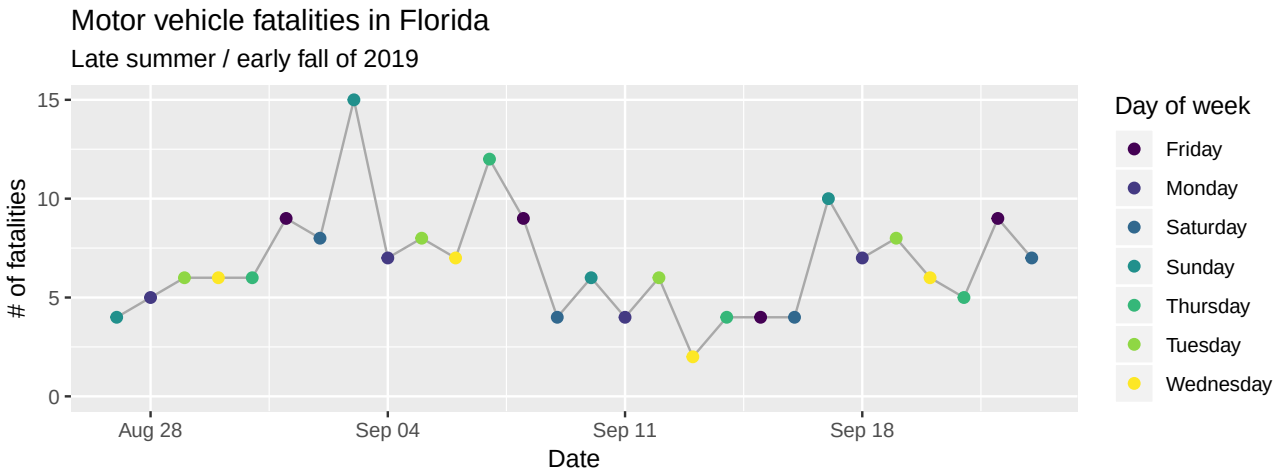


Figure 2.8: Step 6 of layering a plot: Customizing labels. The 'labs' layer customizes not only the x and y axis labels, but also the legend title for the color scale. The title and subtitle are added with a 'ggtitle' layer.

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")
```

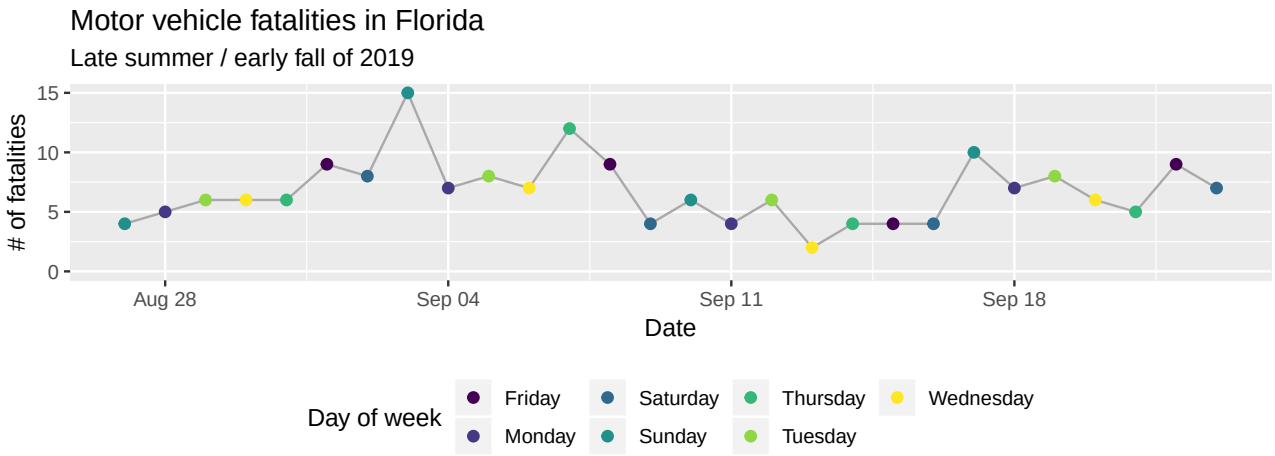


Figure 2.9: Step 7 of layering a plot: Customizing the theme. Move the color legend to below the plot.

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")
```

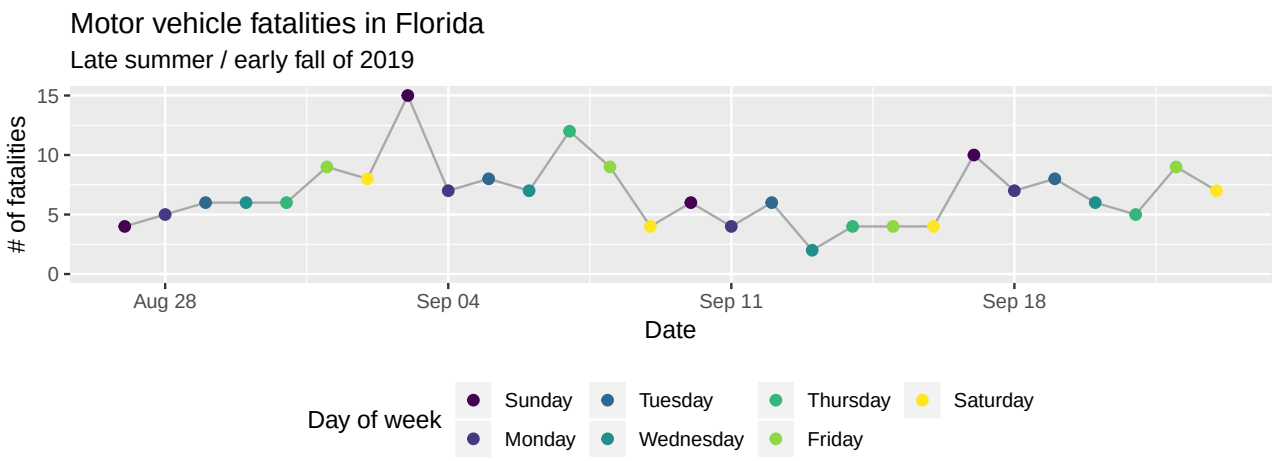


Figure 2.10: Change the order of the days of week from alphabetical to temporal by transforming the data before plotting.

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).

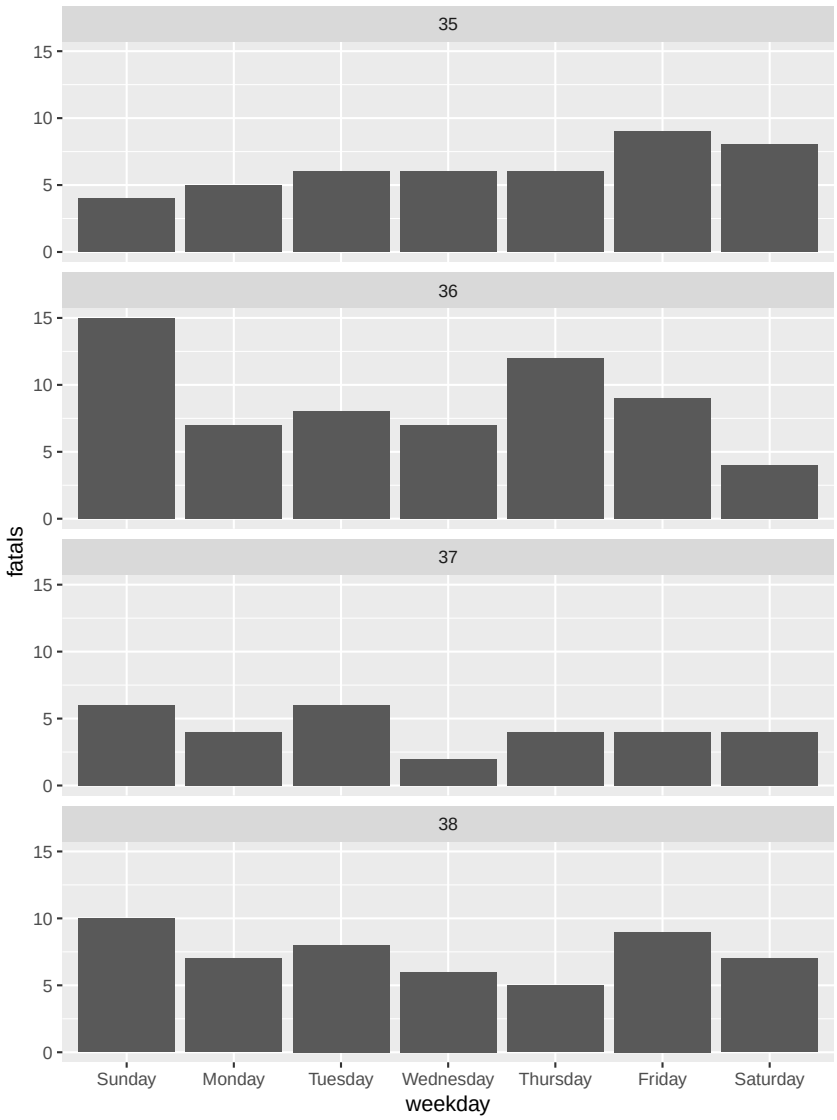
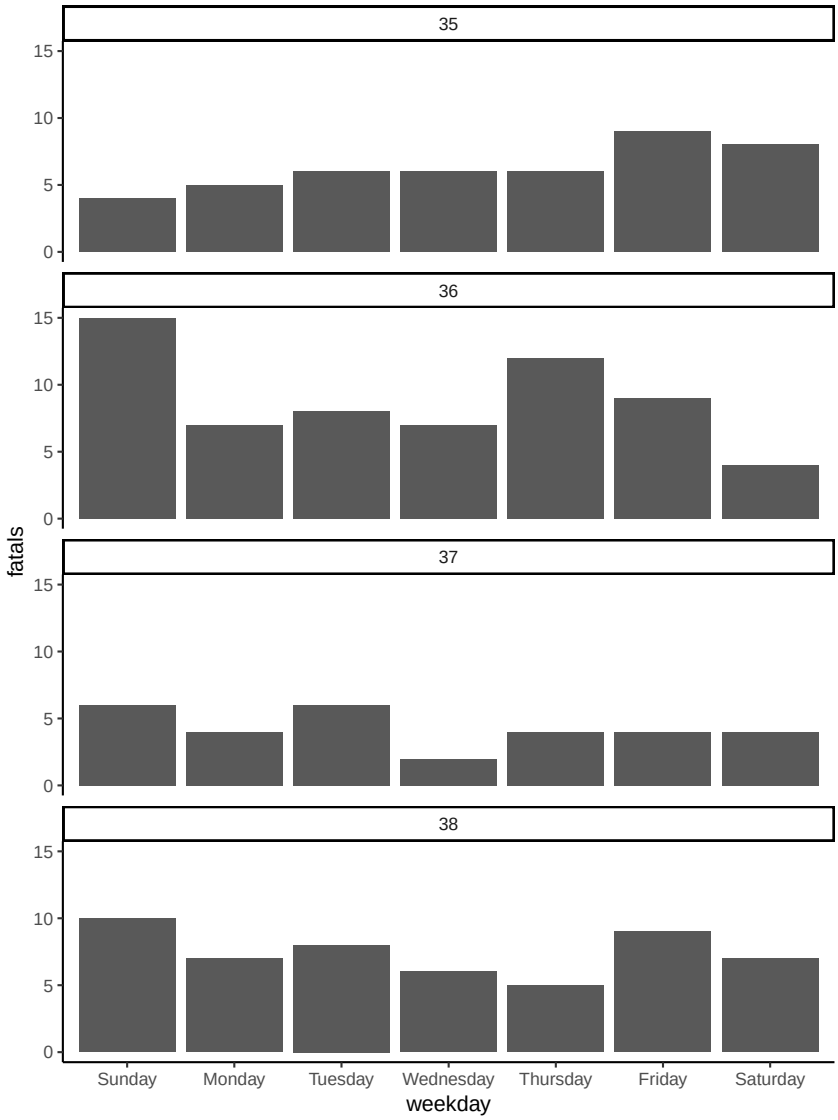


Figure 2.11: Adding a facet layer to a plot. The plot is now faceted by the 'week' column in the dataframe, with all the facets lined up vertically in a single column ('ncol = 1').



\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

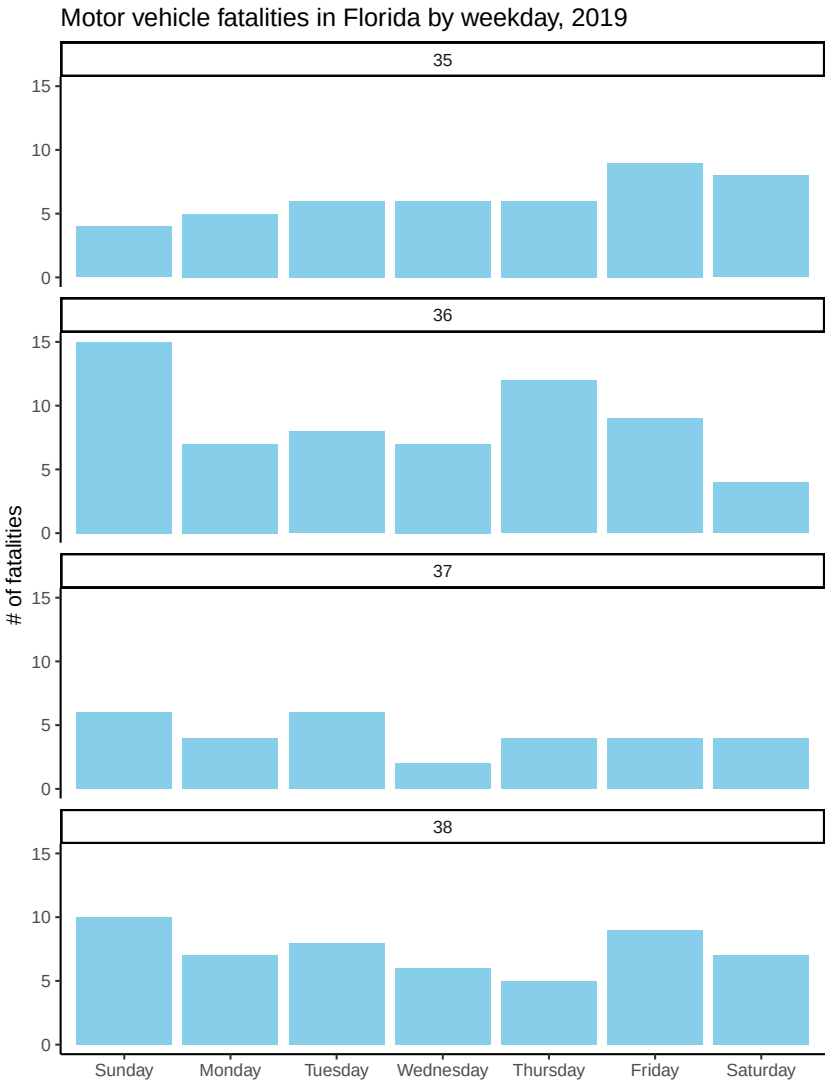


Figure 2.12: Customizing the labeling and adding a constant fill aesthetic. Note that the constant fill aesthetic is specified outside the 'aes' call for the geom, and that the color is specified inside quotation marks.

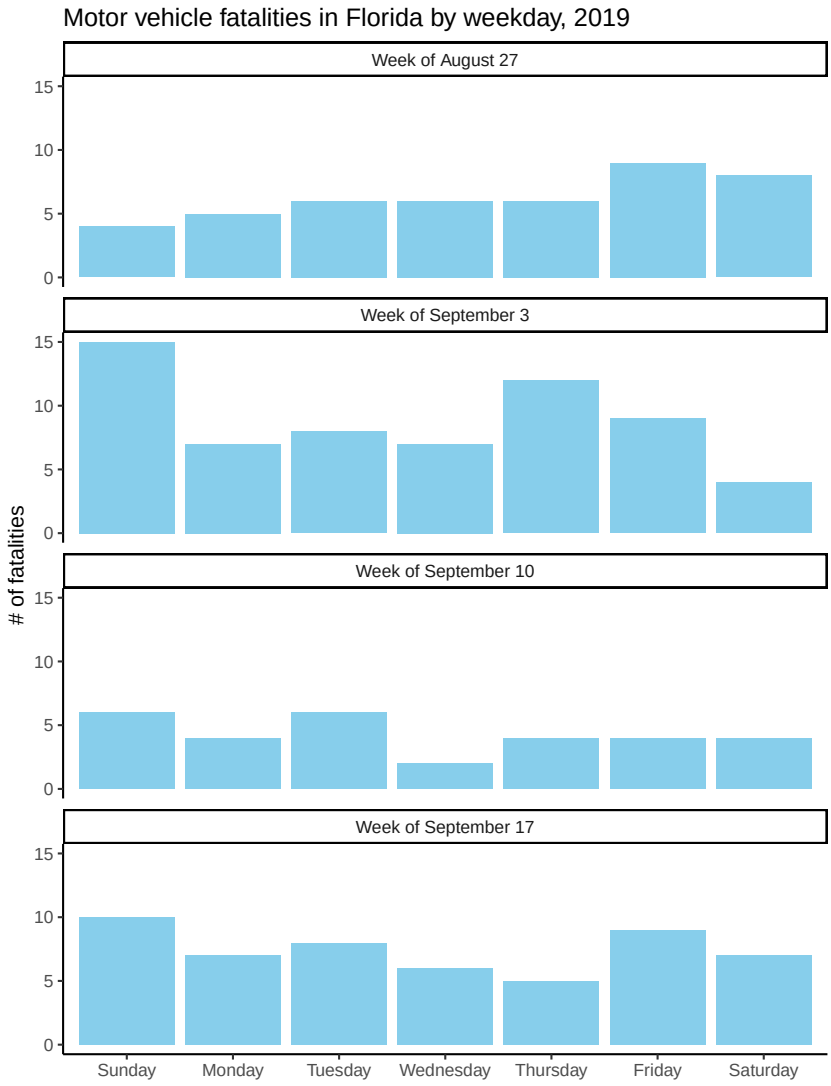
change the input data a bit. In this case, this step is to create clearer labels for the facets. The week number (e.g., 35) won't mean much to most viewers. A label giving the date of the first day in the week ("Week of August 27") will be more helpful. To create these new labels, you can use `group_by`, `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

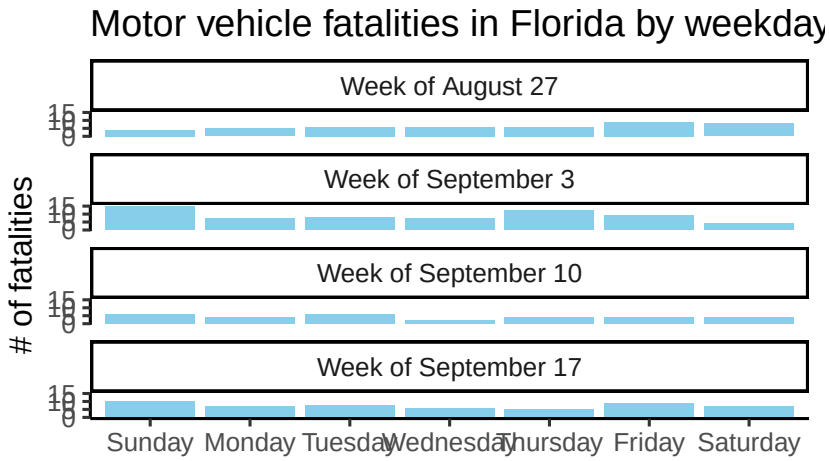
You can assign the output of a `ggplot` call to an R object using the gets arrow (<−), just like you can assign data you've read in to an R object. If you do this, the plot won't print immediately after you run the code:

```
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 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 how the tidyverse framework is based on a "tidy" data format. 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 its "Learn More" references]), 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 column 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.

These special dataframes are given a class called **sf**[1] 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 to demonstrate some of the ways to get an `sf` object.

First, if you have a regular dataframe, you can convert it into an `sf` object,

[1] **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.

specifying which parts of the dataframe include geographical information. If you followed all the set-up instructions in the "Prerequisites", you should have downloaded a dataset called "fl_accidents.csv" and 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")
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,[2] specifying the columns with the geogrphical coordinates using the `coords` parameter.[3] We also want to go ahead and set the projection to something reasonable for latitude-longitude data; the "4326" code is a good pick. That can be set with the `st_sf` function (for a lot more on map projections, see the resources in "Learn More").

```
library("sf")
fl_accidents %<>% st_as_sf(coords = c("longitud",
    "latitude")) %>% st_sf(crs = 4326)
```

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`).

```
fl_accidents
```

[2] 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.

[3] 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.

```
## 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):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## # 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**[4] If you followed the "Prerequisites", you should have downloaded a shapefile as a subdirectory called "al112017_best_track" in the "data" subdirectory. I downloaded this as a zipped data file from the National Hurricane Center and unzipped it. The shapefile includes shapefiles for the track of Hurricane Irma.

[4] **shapefile.** A format for storing geographical data common for GIS. The data will typically be stored in a directory (often zipped to a single file), with separate files that give geographic data and other characteristics of the data.

If you look at the "al112017_best_track" directory, 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 `st_read` and specifying the layer you'd like from the shapefile directory with the `layer` parameter (put the root of the filenames for the layer you want). Since we'll be plotting this with the accident data, we should transform the data's projection from its original projection to the one we set for the `fl_accidents` data. You can do that with the `st_transform` function. 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") %>% 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
```

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 of each accident.

```
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):    4326
```

```
## proj4string:    +proj=longlat +datum=WGS84 +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...
```

Finally, there are some packages available now that allow you to read data, including spatial data, directly into R from large open data databases, using R functions that wrap the database's Application Programming Interfaces (APIs). For example, the `tigris` package lets you pull spatial data into R directly from the US Census. To get spatial data for all the counties in Florida, you can run:

```
library(tigris)
fl_counties <- counties(state = "FL", cb = TRUE,
    class = "sf")
```

If, for any reason, you're not able to get the above code to work, I pulled and saved this data with the example data, and you can load it with the following code if you followed all the directions in the "Prerequisites" and are having problems with the previous code:

```
load("data/fl_counties.RData")
```

3.2 Basic mapping

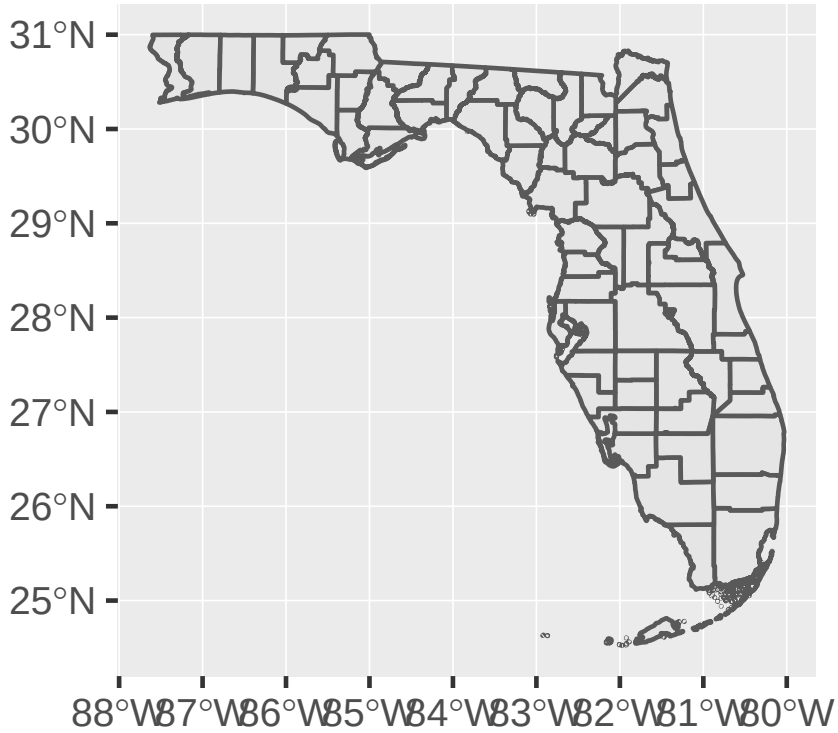
If you don't have `ggplot2` still loaded from the exercises in the "Plot" section, you'll need to load it. Then, you can create a `ggplot` object and add an `sf`

layer to it with `geom_sf`, specifying the `sf` data set to plot.[5] Add a coordinate layer (`coord_sf`) appropriate for a map (otherwise, you might end up with a map that's "stretched" out in either the x- or y-direction):

[5] If you are running this on a Mac, there's a chance that you might get an error message about "polygon edge not found." There seems to be a bug that's still being resolved involving the default graphics driver on Macs with this package. If you get this error message, try fixing it by adding the following layer to the `ggplot` code: `+ theme(axis.text = element_blank(), axis.ticks = element_blank()`

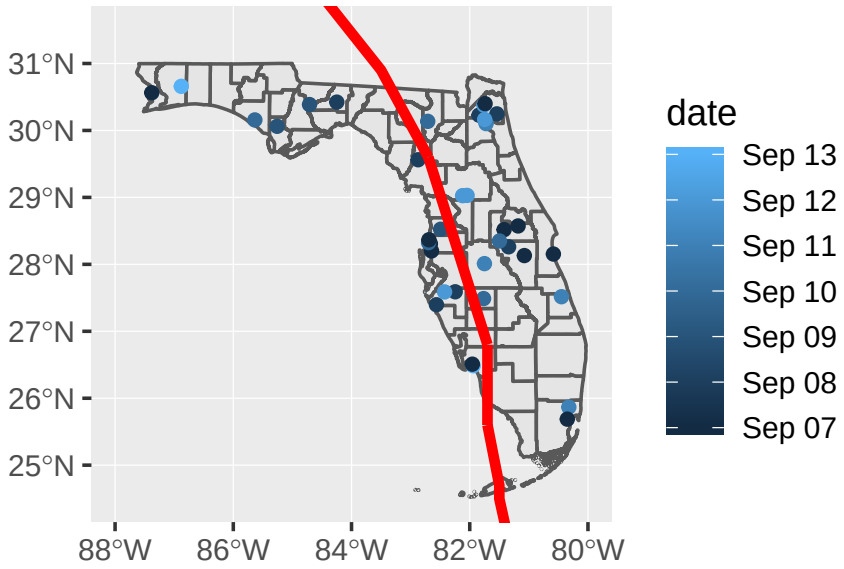
```
library("ggplot2")

ggplot() + geom_sf(data = fl_counties) + coord_sf()
```



You can add other `sf` layers to this map, to add points for the location of each fatal accident and a line for the track of Hurricane Irma. Since the track for Irma began much further south than Florida and continues north of Florida, you'll need to set `xlim` and `ylim` in the `coord_sf` call to ensure that the map is zoomed to show Florida. You can map aesthetics to values in the data, or to constant values, just like with a regular `ggplot` object. In this example, the color of each point is mapped to its date, while the track of Irma is shown in red and a bit larger than the default size.

```
ggplot() + geom_sf(data = fl_counties) + geom_sf(data = fl_accidents,
    aes(color = date)) + geom_sf(data = irma_tracks,
    color = "red", size = 1.5) + coord_sf(xlim = c(-88,
    -80), ylim = c(24.5, 31.5))
```



You can do anything to this map that you would to a regular `ggplot` object. For example, instead of using color to show the date of an accident, you could use faceting to add a small map of each date using `facet_wrap`. This example code also adds some elements for the plot's theme, including changing the background color and taking out the axis ticks and labels.

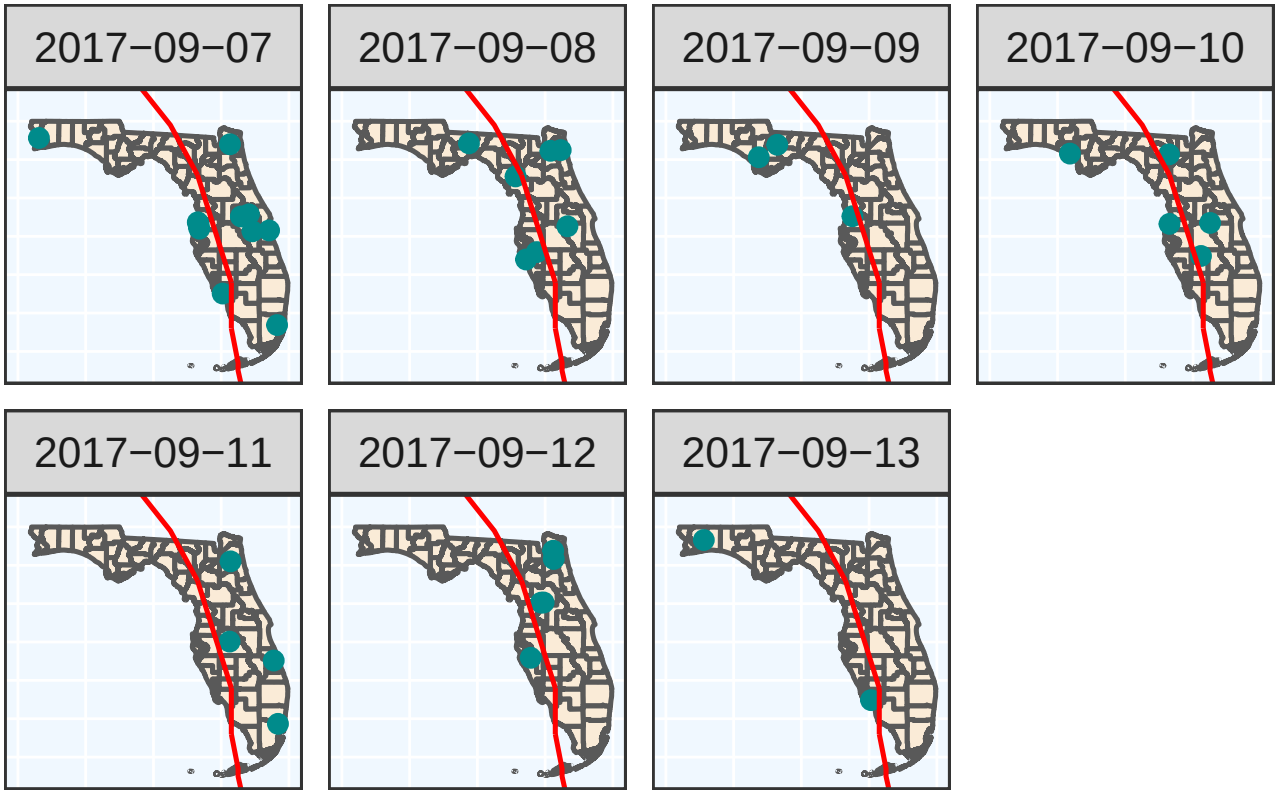
```
ggplot() + geom_sf(data = fl_counties, fill = "antiquewhite") +
    geom_sf(data = fl_accidents, color = "darkcyan") +
    geom_sf(data = irma_tracks, color = "red") +
    coord_sf(xlim = c(-88, -80), ylim = c(24.5,
        31.5)) + facet_wrap(~date, ncol = 4) +
    theme_bw() + theme(panel.background = element_rect(fill = "aliceblue"),
    axis.ticks = element_blank(), axis.text = element_blank(),
    panel.grid = element_line(color = "white",
        size = 0.8))
```

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". Finally, some of the `ggplot` references listed in the "Plot" section's "Learn More" include sections on mapping.



There's an article on the `tigris` package, if you'd like to find out more about how to use that package to read in US Census spatial data.

4

Interact

For a few years now, you have been able to use Javascript to create interactive plots to include on websites. These plots respond to actions from your mouse. When you scroll over a point, for example, some information about the point might show up, or when you click on a marker on a map, you might get a pop-up box with some information.

More recently, R packages have been developed that allow you to create these interactive visualizations directly from R. Many of these packages were developed by members of the R Studio team in a collection of packages called `htmlwidgets` (Vaidyanathan et al., 2018). Once you've mastered how to plot in R with `ggplot2` and how to map with `sf`, some of these packages are almost embarrasingly easy to use to create really fantastic data visualizations.

You can use these interactive visualizations on your own computer to explore your data. You can also include them in webpages and post them for others to see. In the "Report" section, I'll talk more about how you can create reports using R to share or post as webpages. While you're developing a visualization, you'll be able to check it out in the "Viewer" pane in R Studio, just like you can see plots in the "Plots" pane.

4.1 DT: Datatables

One of the easiest interactive visualizations to create in R is an interactive datatable, which you can create using the `DT` package (Xie et al., 2018). If you went through the "Prerequisites", you should have installed that package to your computer. You can scroll down to see one example. Explore this table a little bit. You'll see that you can click the numbers at the bottom right of the table to page through all the data in the table. You can use the arrows beside the column names to rearrange the rows based on the values they have in a column. With the search box in the top right, you can try searching for specific elements in the data.[1]

You can build this table using the `DT` package in the `htmlwidgets` family of packages. First, you need to read in the data you'd like to print. We're using one

[1] As an example, try searching for "2017-09-10" in the search box to get the accidents that occurred on the day of Hurricane Irma's Florida landfall, September 10, 2017. Then try clicking on the "fatals" column name until the arrow beside it points down, to see what the maximum number of fatalities during an accident was.

of the datasets from the "Plot" section, so you can read it in the same way you did in the previous section (if you already have `readr` and `magrittr` loaded from working on a previous section, you can skip those lines):

```
library("readr")

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

Now, to print this dataframe as an interactive data table, just run the data frame through the `datatable` function.

```
library("magrittr")
library("DT")
fl_accidents %>% datatable()
```

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

In your R Studio session, the interactive table should show up in the "Viewer" pane. If you want to see it in a separate window, click on the "Zoom" button, and it will open the table in a separate window that you can rescale.

You can do a lot of customization through the `datatable` call. For example, you can add a caption to the table, change the appearance using the `class` parameter, change the table width, and use clearer names for the columns. Try the following example code to see an example of how these changes can change the table's appearance:

```
datatable(fl_accidents, class = "compact", caption = "Example of an interactive data table. Each observa
    colnames = c("County FIPS", "Date", "Latitude",
        "Longitude", "# fatalities"), width = 800)
```

Show 10 entries Search:

Example of an interactive data table. Each observation (row) is the information for one of the fatal motor vehicle accidents in Florida the week of Hurricane Irma's landfall, with columns for the county where the accident occurred, the date of the accident, and the number of fatalities.

	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

To get more details on all the options available, see the helpfile for `datatable` (run `?datatable`) or read through the online tutorial for the package.

4.2 *Plotly*

One of the easiest ways to make a `ggplot` plot interactive is with the `plotly` package (Sievert et al., 2018). If you went through the "Prerequisites", you should have installed that package to your computer. This package has a function, `ggplotly`, that allows you to make a `ggplot` plot interactive with a single line of code.

For example, try it out with the Florida motor vehicle fatality time series from the "Plot" section.[2] First, save the plot to an object called `fatality_plot` using the gets arrow:

[2] To run this code, you should make sure you've run the code in the "Plot" section in your current R session. If the "Environment" pane of your RStudio session includes "daily_fatalities" in its list, you should be okay. The code itself should look familiar from the "Plot" section.

```
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")
```

As a reminder, when you assign a plot object a name, it won't print out, but you can print it out anytime you want by calling the name you assigned it (Figure 4.1).

```
fatality_plot
```

To use plotly to make this plot interactive, all you need to do is load the `plotly` package and run this `ggplot` object through its `ggplotly` function:

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

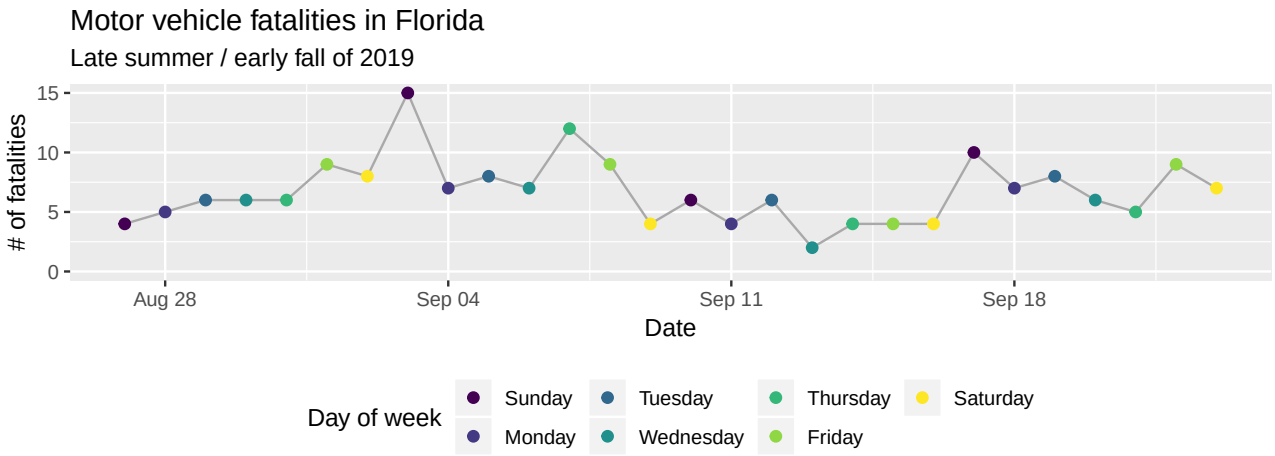


Figure 4.1: Example of printing out a ggplot object by calling its assigned name.

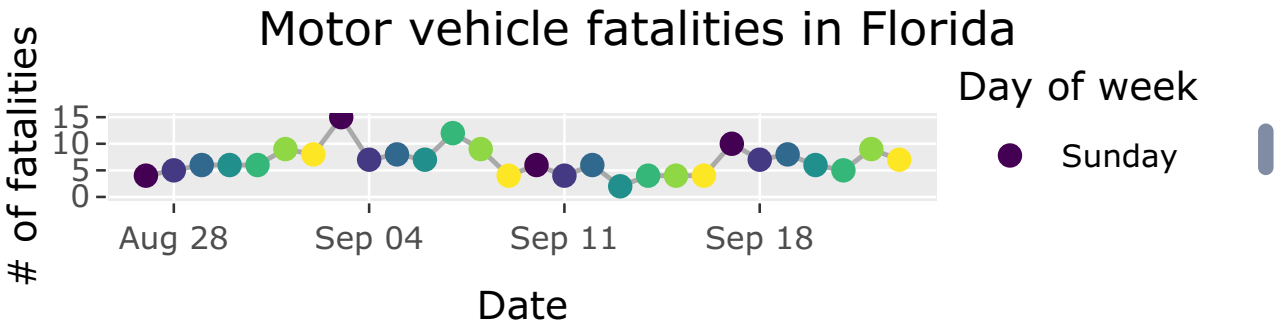


Figure 4.2: Example of an interactive plot created with 'plotly'. Try scrolling over the points and playing around with the buttons in the top right.

Try scrolling over Figure 4.2: when you scroll over a point, a pop-up should appear with some information about the plot. You should also have some buttons on the top right of the plot that allow you to zoom in and out, to download the figure as an image file (png), and to do a few other things.

This is just the tip of the iceburg for what you can do in terms of making interactive plots in R. I show one more example of a package for creating interactive plots (`leaflet`, for creating interactive maps), and the I'll give you a lot of resources at the end of this section for where you can find out more about how to make interactive plots using R.

4.3 Leaflet

One of my favorite packages for interactive plotting in R is the `leaflet` package (Cheng et al., 2018). This package allows you to create interactive maps very similar to the maps you see on Google maps. As the background, it pulls in tiles from a collection of tiles at different zoom levels, allowing you to zoom in and out and pan around the resulting map.

You can create a `leaflet` map using data that's in an `sf` class, which we covered in the "Map" section. For example, you can use the following code to read in the data, convert it to an `sf` object by specifying the columns with geographical information, and set projection information using the `st_sf` function:

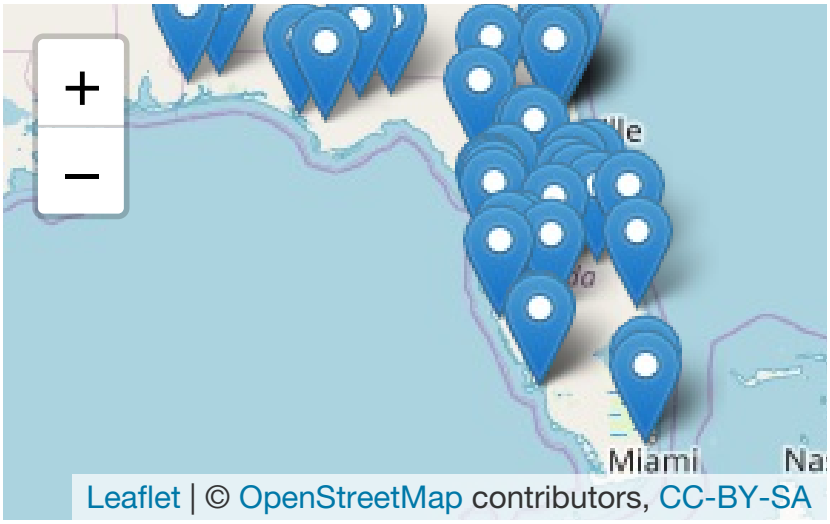
```
library("sf")

fl_accidents %<>% st_as_sf(coords = c("longitud",
    "latitude")) %>% st_sf(crs = 4326)
```

Now you can create the map with the data. The code should look similar to `ggplot2` code for plotting, although notice that it uses a pipe operator (%>%) rather than a plus sign (+) to add on the layers. The `leaflet` call creates a leaflet object, and the `addTiles` function adds in the background tiles.[3] You can add markers showing the location of each accident using the `addMarkers` call, specifying the dataset to use with the `data` parameter:

[3] In this example, we're using the default background tiles. You can pick from a variety of styles for background tiles, however. See the online documentation for `leaflet`, listed later in this section, for more.

```
library("leaflet")
leaflet() %>% addTiles() %>% addMarkers(data = fl_accidents)
```



The result is an interactive map, with a marker at the location of each accident. Try using the plus and minus buttons to zoom in and out, and click on the map to pan around the map.

There is a lot you can do to make the map more interesting. For example, you can add another layer to the map with the track of Hurricane Irma. You can read that track in from a shapefile using `st_read`, as described in the "Map" section, transforming the projection to map the projection of the accident data using `st_transform`.

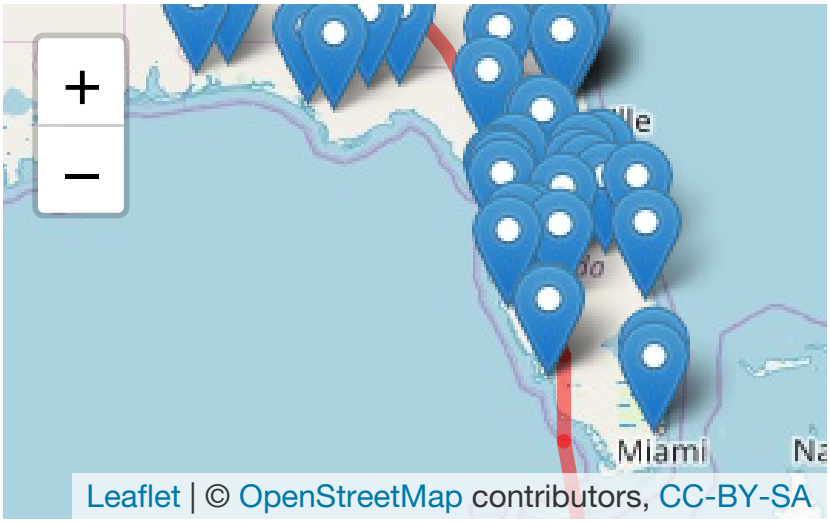
```
irma_track <- st_read("data/al112017_best_track",
    layer = "al112017_lin") %>% st_transform(crs = 4326)
```

```
## 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
```

This spatial object is a type known as a "polyline", so you can add it to the `leaflet` map with a layer called with `addPolylines`. In this example, I've made the line red with the option `color = "red"`. The `leaflet` plot will automatically zoom to fit the data you're plotting—since the hurricane started in the tropics and went past Florida, its range is much larger than Florida. To have the `leaflet` plot still open zoomed in to Florida, you can use the `fitBounds` call to specify the opening view of the map. Finally, with the call `popup = ~ date`, we're specifying that the each marker should show the date of the accident when you click on it.

```
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")
```



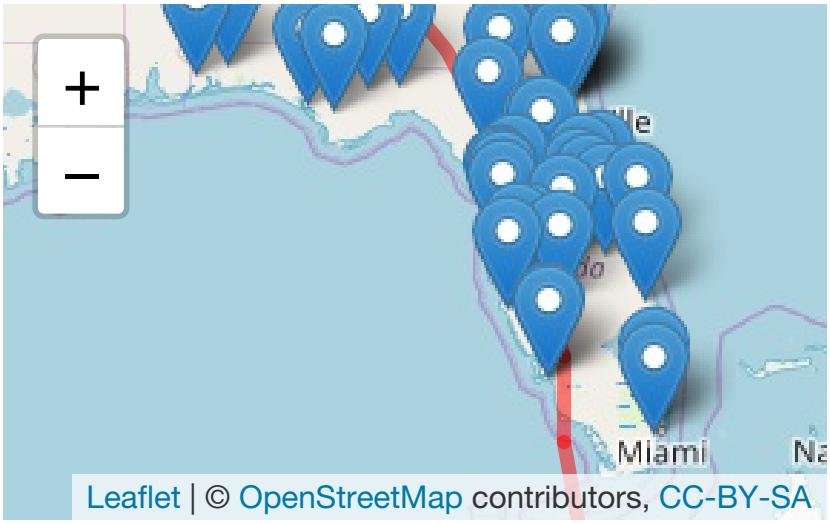
The "pop-ups" for the markers can be developed to be pretty fancy. You can use `paste` to paste together elements from the data with other words to create nice labels. You can also format these, using HTML formating.[4]

Try this out in the following code. After you run this, the `leaflet` map should have pop-ups that give both the date and the number of the fatalities, on separate lines (the `<br>` creates a line break). First, add a column to `fl_accidents` called `popup`, with the full (HTML formatted) text you want to include in the labels. Then, in the `addMarkers` call, specify that this column should be used for the pop-up with `popup = ~ popup`.

[4] With HTML formatting, you use special tags to specify formatting like bold font or weblinks. For example, to write "**bold** is in bold" with HTML, you would write "`<b>bold<b> is in bold`".

```
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")
```



The `leaflet` package has extensive online documentation. Browse through the sections of this online documentation to get a lot more ideas of how you can create and customize interesting interactive maps with R.

4.4 Learn more

Many of the packages for these types of interactive graphics were developed by people at R Studio, and they have an excellent website with more information about the `htmlwidgets` family. This website links through to tutorials for each package, including `DT`, `plotly`, and `leaflet`, with lots of examples and the code behind them. This website also has a gallery, with great examples.

Lots of people are using `htmlwidgets` to create interesting visualizations through R, and they will often blog about what they did. If you want more examples of how to create visualizations with a certain package (e.g., `leaflet`), it's worth googling something like "blog post R leaflet example".[5]

Once you've worked some with the `htmlwidgets` packages, you should also check out Shiny. Shiny allows you to create much more complex interactive graphics in R—actual web applications, where the R code to create the visualization is re-run when the user interacts with elements of the visualization. To get an idea of the things you can create with Shiny, you should check out the Shiny gallery and Show Me Shiny.

[5] Some of the `htmlwidgets` R packages have the same name as the Javascript library they're associated with, so be sure to include "R" in your Google search, so you'll find examples in the right programming language.

5
Report

Literate programming, an idea developed by Donald Knuth, mixes code that can be executed with regular text. The files you create can then be rendered, to run any embedded code. The final output will have results from your code and the regular text.

R has a collection of packages that extend these ideas of literate programming, allowing you to create reports that interweave R code, the results from that code, and text. This collection include the packages `knitr` and `rmarkdown`. I believe both are automatically installed to your computer when you install RStudio.

This section will walk you through some of the basics of creating two types of reports using RMarkdown: a basic RMarkdown report and a dashboard. The R project directory you downloaded in the "Prerequisites" section includes example files for both in its "reports" subdirectory.

5.1 RMarkdown

The RMarkdown framework makes it very easy to create nice reports from R code in RStudio.

Here are the basics of opening and rendering an R Markdown file in RStudio:

- To open a new R Markdown file, go to "File" -> "New File" -> "RMarkdown…". In the box that pops up, chose a "Document" in "HTML" format.
- This will open a new R Markdown file in RStudio. The file extension for R Markdown files is ".Rmd".
- The new file comes with some example code and text. You can run the file as-is to try out the example. You will ultimately delete this example code and text and replace it with your own.
- Once you "knit" the R Markdown file, R will render an HTML file with the output. This is automatically saved in the same directory where you saved your .Rmd file.
- Write everything besides R code using Markdown syntax (explained below).

To include R code in an RMarkdown document, you need to separate off the

code chunk using the following syntax (here, the code assigns the numbers one to 10 to the object `my_vec`):

```
```{r}
my_vec <- 1:10
```
```

This syntax might look unusual, but it tells R how to find the start and end of pieces of R code when the file is processed by R. R will walk through, find each piece of R code inside these special sections, run it and create output (printed output or figures, for example). R will ignore anything not in one of these sections, including the text of the report. The output from this R processing will then pass into another program to complete rendering (e.g., a LaTeX engine for pdf files).

R Markdown files are mostly written using Markdown. To write R Markdown files, you need to understand what markup languages like Markdown are and how they work.

In Word and other word processing programs you have used, you can add formatting using buttons and keyboard shortcuts (e.g., "Ctrl-B" for bold). The file saves the words you type. It also saves the formatting, but you see the final output, rather than the formatting markup, when you edit the file (WYSIWYG – what you see is what you get).

In markup languages,[1] on the other hand, you markup the document directly to show what formatting the final version should have (e.g., you type `**bold**` in the file to end up with a document with **bold**).

[1] Examples include HTML, LaTeX, and Markdown

To write a file in Markdown, you'll need to learn the conventions for creating formatting. This table shows the markup for some common formatting choices:

Code	Rendering	Explanation
'**text**'	**text**	boldface
'*text*'	*text*	italicized
'[text](www.google.com)'	[text](www.google.com)	hyperlink
'# text'		first-level header
'## text'		second-level header

Some other simple things you can do in Markdown include:

- Lists (ordered or bulleted)
- Equations
- Tables
- Figures from file
- Block quotes
- Superscripts

See the resources listed in "Learn More" for links to some helpful resources for learning RMarkdown, including more of these Markdown formatting markups.

5.2 Dashboards

In RStudio, you can also now create a "dashboard" using RMarkdown. A dashboard might be a good choice if you need to summarize or show a lot of information on a single webpage in a way that everything can be viewed in one window. Figure 5.1 gives an example dashboard (although this is just a screenshot—if posted online, the leaflet map in this would be interactive).

To create a dashboard, you'll need to have the `flexdashboard` package (Iannone et al., 2018) installed. If you have this package installed, you can create a dashboard by opening your RStudio session and selecting "File" -> "New File" -> "RMarkdown". In the box that pops up, go to the "From Template" tab and select "Flexdashboard".

The R Project with examples for this workshop includes an example file for creating a dashboard in the "reports" subdirectory's "flexdashboard_example.Rmd" file.[2] Open this file and try creating the dashboard using the "Knit" button. You can get an idea for how the code converts to elements of the dashboard by changing some of the code in the RMarkdown file.

The `flexdashboard` package is extensively documented through its website, where you can find many examples of how to create dashboards with RMarkdown.

[2] You might notice that there are some other files in this subdirectory that start with "flexdashboard_example". These are the files that are written out when you knit the RMarkdown file. You should never change these by hand, but these include the files you'll want to share or post online as the output of the RMarkdown file (e.g., "flexdashboard_example.html").

5.3 Learn more

For more Markdown conventions, see RStudio's R Markdown Reference Guide (link also available through "Help" in RStudio). There are several great books on `knitr` and RMarkdown now; R Markdown: The Definitive Guide is particularly good, and includes the creater of `knitr` and `rmarkdown` as a coauthor.

For more on creating dashboards using RMarkdown, see `flexdashboard` package's website, which includes many examples of code and output for these dashboards.

You can post the output of an RMarkdown file for free (note that it will be public) through RPubs. This is also a great site for exploring reports others have created using RMarkdown. For more on posting your own files to RPubs, see here.

A collection of books published with R using RMarkdown is available through bookdown.org. You can also use RMarkdown to create your own website with a blog using the `blogdown` package, which is explained in blogdown: Creating Websites with R Markdown.

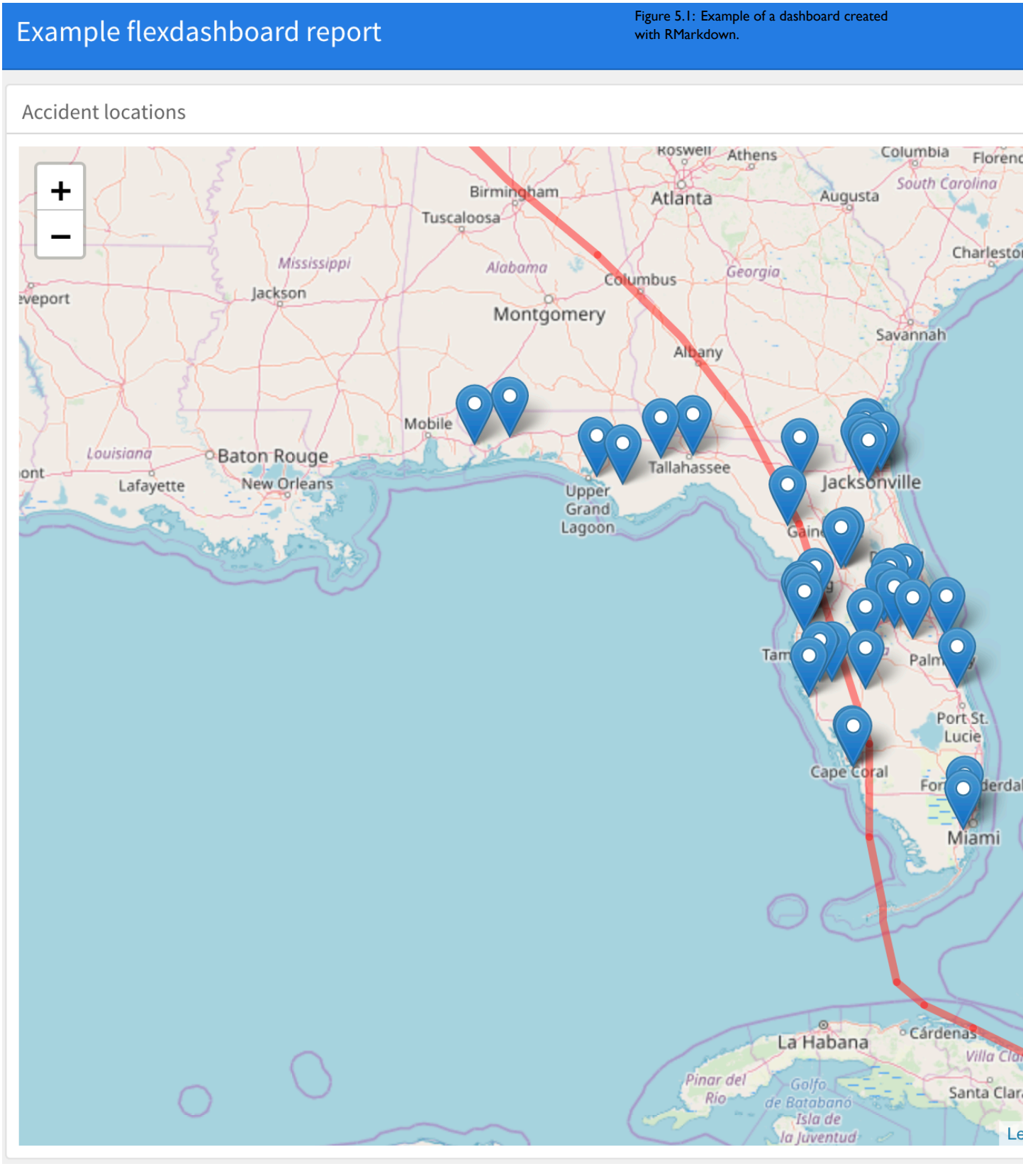


Figure 5.1: Example of a dashboard created with RMarkdown.

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 introduce you to how you can clean real datasets to convert them into this format using tidyverse tools, as well as provide some resources for you to use to develop your skills in "tidying" data.

6.1 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**,[1] you can write clean and compact code to clean even very large and messy datasets.

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).[2] 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.

[1] **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.

[2] There are some clear specifications for this format. I'm not going to go into them here, but several of the references given in the "Learn More" section go into depth in describing and defining this format.

The functions in the tidyverse work this way. For the major data cleaning functions, they all take the same format of input (a tidy tibble) and they all output that same format of input. Just like you can build Legos on top of each other in different orders and patterns to create lots of different structures, this framework of small tools that work on the same type of input and produce the same type of output allow you to string together lots of small, simple calls to do some very complex things.

To work with data with tidyverse tools, we'll use two main ideas. The first is that we'll use many small tools that each do one thing well and that can be combined in lots of configurations to achieve complex tasks. The second is that we'll string these small functions togther using a special operator called the "pipe operator".

The main functions in the tidyverse (sometimes called "verbs") all do simple things. For example, there are functions to `select` certain columns, `slice` to specific rows, `filter` to a set of rows that match some criterion, `mutate` existing columns to create new columns or change existing ones in place, and `summarize` a dataframe, possibly one that you `group_by` certain characteristics of the data (e.g., a summary of mean height **grouped by** gender).

The second main idea for this data cleaning approach is that we'll use a **pipe operator** (%>%). This operator lets you input a tidy dataset as the first argument of a function. In practice, this allows you to string together a "pipeline" of data cleaning calls that is very clean and compact.

6.2 Tidying example

We've been using an example dataset with fatal motor vehicle accidents. I downloaded the raw data from the Fatality Analysis Reporting System (FARS), and the original download was a large zipped file, with separate files for variables about the accident, the people involved, etc.

In this short booklet, I won't be able to teach you all the tools you can use to tidy a dataset in R, but I can quickly walk you through an example to show you how powerful these tools can be, and how clean and efficient the final code is. R's power for data analytics comes in part from these wonderful tools for working with data, and in a typical data analysis project, I'll use these tools extensive to create different summaries and views of the data as I work. You will definitely want to learn these tools if you're serious about using R, so I've provided several places to go to master them in the "Learn more" section.

For the rest of the section, we'll look at cleaning up data from the Federal Accident Reporting System, the source of the example data we've used in other sections. If you completed the set-up in the "Prerequisites", you should be able to load this data using:

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

As a reminder, to print out some information about this data, call the object's name:

```
fl_accidents
```

```
## # A tibble: 34,247 x 52
##    STATE ST_CASE VE_TOTAL VE_FORMS PVH_INVL
##    <dbl>   <dbl>    <dbl>    <dbl>    <dbl>
##  1     1   10001        1        1        0
##  2     1   10002        1        1        0
##  3     1   10003        3        3        0
##  4     1   10004        1        1        0
##  5     1   10005        1        1        0
##  6     1   10006        2        2        0
##  7     1   10007        2        2        0
##  8     1   10008        1        1        0
##  9     1   10009        1        1        0
## 10     1   10010        1        1        0
## # ... with 34,237 more rows, and 47 more
## #   variables: PEDS <dbl>, PERNOTMVIT <dbl>,
## #   PERMVIT <dbl>, PERSONS <dbl>,
## #   COUNTY <dbl>, CITY <dbl>, DAY <dbl>,
## #   MONTH <dbl>, YEAR <dbl>, DAY_WEEK <dbl>,
## #   HOUR <dbl>, MINUTE <dbl>, NHS <dbl>,
## #   RUR_URB <dbl>, FUNC_SYS <dbl>,
## #   RD_OWNER <dbl>, ROUTE <dbl>,
## #   TWAY_ID <chr>, TWAY_ID2 <chr>,
## #   MILEPT <dbl>, LATITUDE <dbl>,
## #   LONGITUD <dbl>, SP_JUR <dbl>,
## #   HARM_EV <dbl>, MAN_COLL <dbl>,
## #   RELJCT1 <dbl>, RELJCT2 <dbl>,
## #   TYP_INT <dbl>, WRK_ZONE <dbl>,
## #   REL_ROAD <dbl>, LGT_COND <dbl>,
## #   WEATHER1 <dbl>, WEATHER2 <dbl>,
## #   WEATHER <dbl>, SCH_BUS <dbl>,
## #   RAIL <chr>, NOT_HOUR <dbl>,
## #   NOT_MIN <dbl>, ARR_HOUR <dbl>,
## #   ARR_MIN <dbl>, HOSP_HR <dbl>,
## #   HOSP_MN <dbl>, CF1 <dbl>, CF2 <dbl>,
## #   CF3 <dbl>, FATALS <dbl>, DRUNK_DR <dbl>
```

This is a large dataset, with over 50 columns and over 34,000 rows. It records details about all of the fatal motor vehicle accidents in the US in 2017 (at least, that were reported to this database). Currently it inclues all states, although we're planning to limit it to Florida. In this section, you'll work through

cleaning up this data, as you might if you were starting from this raw data and needed to create summaries and plots similar to those in other sections of this booklet.

The following piece of code is all the code you need to transform this dataset into the `fl_accidents` dataset we've used in earlier examples.

```
library(magrittr)
library(dplyr)
library(tidyr)
library(stringr)
library(lubridate)

fl_accidents %>% rename_all(.funs = str_to_lower) %>%
    select(state, county, day, month, year, latitude,
        longitud, fatals) %>% filter(state ==
    12) %>% mutate(county = str_pad(county, width = 3,
    pad = "0")) %>% unite(col = fips, c(state,
    county), sep = "") %>% unite(col = date, c(month,
    day, year), sep = "-") %>% mutate(date = mdy(date)) %>%
    filter(date >= mdy("9-7-2017") & date <= mdy("9-13-2017"))
```

```
## # A tibble: 37 x 5
##    fips  date       latitude longitud fatals
##    <chr> <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
```

This function takes the large original dataset. It first *renames* all the columns—they were in all capital letters, which are a pain to type in your code, so we're using a function from the `stringr` package to change them all to lowercase. We're then *selecting* just the columns we want to work with for the plot, using their column names to pick them (`state`, `county`, etc.). We're then *filtering* to just the observations in Florida (where the state FIPS code is 12). Counties have five digit FIPS codes that are useful to use when merging different datasets, including merging in geographic data, and the dataset at this stage has the state part of the FIPS code and the county part in different columns. The county part

is currently in a numeric class, which I need to "pad" with 0s at the beginning if it's currently fewer than three digits. We'll *mutate* the county FIPs code to pad it with 0s, using another function from the `stringr` package. We'll then *unite* the state and county FIPS columns to create a single column with the 5-digit FIPS code. Next, we want to convert the date information into a "Date" class, which will let us work with these values more easily. We'll *unite* all the columns with date information (`month`, `day`, `year`) into a single column, and then we'll use a function from the `lubridate` package to *mutate* this column to have a date class. Finally, we'll *filter* to just the observations with a date within a week of Hurricane Irma's landfall on September 10, 2017.

At this stage, don't worry if you don't know which functions you should use to clean up a new dataset, just try to get a feel for how this tidyverse framework is allowing you to clean up the dataframe with lots of small, interoperable tools. In the example code, try highlighting and running different portions of the code and check out the output at each step along the way. This will help you get a better idea for how this process works.

Becoming familiar with these tools so you can use them yourself takes some time, but is well work the effort. In the "Learn more" section, I've got some tips on where you can go to develop those skills.

Finally, the code above is cleaning the data, but not overwriting the original `fl_accidents` object—instead, it's printing out the result, but not saving it for you to use later. To use the cleaned data, you'll need to overwrite the original R object. You can do that in two ways. First, you can use the gets arrow to assign the output to the same R object name:

```
fl_accidents <- fl_accidents %>% rename_all(.funs = str_to_lower) %>%
    select(state, county, day, month, year, latitude,
        longitud, fatals) %>% filter(state ==
    12) %>% mutate(county = str_pad(county, width = 3,
    pad = "0")) %>% unite(col = fips, c(state,
    county), sep = "") %>% unite(col = date, c(month,
    day, year), sep = "-") %>% mutate(date = mdy(date)) %>%
    filter(date >= mdy("9-7-2017") & date <= mdy("9-13-2017"))
```

If you want to be even more compact, you can use something called the **compound pipe operator** (%<>%). This inputs an R object and then, when it's done running all the code, overwrites that R object with the output. You can think of it as combining the <- and %>% operators. Here's how you would use it to save the cleaned version of the data to the `fl_accidents` object name:

```
fl_accidents %<>% rename_all(.funs = str_to_lower) %>%
    select(state, county, day, month, year, latitude,
        longitud, fatals) %>% filter(state ==
    12) %>% mutate(county = str_pad(county, width = 3,
    pad = "0")) %>% unite(col = fips, c(state,
```

```
    county), sep = "") %>% unite(col = date, c(month,
    day, year), sep = "-") %>% mutate(date = mdy(date)) %>%
    filter(date >= mdy("9-7-2017") & date <= mdy("9-13-2017"))
```

6.3 Learn more

To learn more about working with data using tidyverse tools, one of the best sources is the book R for Data Science mentioned in an earlier "Learn more" section. This book is available for a very reasonable price in paperback as well as free online through bookdown.org. One of its coauthors (Hadley Wickham) is the creator of the tidyverse and many of its packages. It includes a full description of the "tidy data" format, as well as lots of instruction on using tidyverse tools to work with datasets to convert them to, or work with them once they're in, this format.

The tidyverse is a very popular tool now for working with data in R. If you nose around the internet a bit, you should be able to find a lot of example blog posts using tidyverse tools. Try a Google search to find some examples in a topic area that interests you (e.g., your favorite sport, a research topic).

RStudio has several cheatsheets (see "Learn more" in the "Plot" section) related to creating and working with tidy data. These include cheatsheets for working with dates (with the `lubridate` package), factors (the `forcats` package), and strings (the `stringr` package). You can find them all here. As with plotting, you will learn a lot very quickly by working through the examples on those cheatsheets, and then keeping them handy as you try to apply R to explore your own data.

7
Final Words

This booklet, and the associated conference workshop, are only able to give you a taste of R. If you'd like to learn more, try out some of the resources listed in the "Learn more" sections. R is free, all its extension packages are free, and there are lots of excellent free or affordable resources for learning R programming.

Hopefully this booklet has been a helpful example of how you can use R for data visualization. Mastering R coding is something anyone can do, but it does take practice and experimentation. As you learn R, remember that you're not going to break anything by playing around with the code—if something doesn't work, don't be afraid to change things in the code.

I hear concerns every now and then from people who are used to proprietary software about the fact that open source software, since it isn't created and sold by a company, doesn't have the same level of support as proprietary tools. I think this is quite a misperception. One of the best parts of R is its wonderful and welcoming international community. There are very active groups discussing R on Twitter (#rstats) and StackOverflow, a site for posting and answering programming questions. Active mailing lists exist for both R users and R developers. R developers have always provided documentation for their packages, but the development of new reporting tools like RMarkdown have allowed them to create documentation that is even more user-friendly and accessible, and they often provide extensive tutorials to learn how to use their packages. For companies that are interested in using R but require extensive support, there are companies like RStudio that can provide enterprise-level support. None of these concerns are real barriers to using R rather than a proprietary software program.

I hope you will continue exploring R for your own data analysis projects and join the R community!

8

Bibliography

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

Cheng, J., Karambelkar, B., and Xie, Y. (2018). *leaflet: Create Interactive Web Maps with the JavaScript 'Leaflet' Library*. R package version 2.0.2.

Iannone, R., Allaire, J., and Borges, B. (2018). *flexdashboard: R Markdown Format for Flexible Dashboards*. R package version 0.5.1.1.

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

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2018). *plotly: Create Interactive Web Graphics via 'plotly.js'*. R package version 4.8.0.

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

Vaidyanathan, R., Xie, Y., Allaire, J., Cheng, J., and Russell, K. (2018). *htmlwidgets: HTML Widgets for R*. R package version 1.3.

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.

Xie, Y., Cheng, J., and Tan, X. (2018). *DT: A Wrapper of the JavaScript Library 'DataTables'*. R package version 0.5.