

## Reporting data results #2

---

## Example data

---

## Example data

This week, we'll be using some example data from NOAA's Storm Events Database.

Go to <https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/csvfiles/> and download the bulk storm details data for 2017, in the file that starts "StormEvents\_details" and includes "d2017".

Move this into a good directory for your current working directory and read it in using `read_csv` from the `readr` package.

## Example data

This data lists major weather-related storm events during 2017.

For each event, it includes information like the start and end dates, where it happened, associated deaths, injuries, and property damage, and some other characteristics.

## Example data

```
colnames(storms_2017)
```

##	[1]	"BEGIN_YEARMONTH"	"BEGIN_DAY"	"BEGIN_TIME"
##	[4]	"END_YEARMONTH"	"END_DAY"	"END_TIME"
##	[7]	"EPISODE_ID"	"EVENT_ID"	"STATE"
##	[10]	"STATE_FIPS"	"YEAR"	"MONTH_NAME"
##	[13]	"EVENT_TYPE"	"CZ_TYPE"	"CZ_FIPS"
##	[16]	"CZ_NAME"	"WFO"	"BEGIN_DATE_TIME"
##	[19]	"CZ_TIMEZONE"	"END_DATE_TIME"	"INJURIES_DIRECT"
##	[22]	"INJURIES_INDIRECT"	"DEATHS_DIRECT"	"DEATHS_INDIRECT"
##	[25]	"DAMAGE_PROPERTY"	"DAMAGE_CROPS"	"SOURCE"
##	[28]	"MAGNITUDE"	"MAGNITUDE_TYPE"	"FLOOD_CAUSE"
##	[31]	"CATEGORY"	"TOR_F_SCALE"	"TOR_LENGTH"
##	[34]	"TOR_WIDTH"	"TOR_OTHER_WFO"	"TOR_OTHER_CZ_STATE"
##	[37]	"TOR_OTHER_CZ_FIPS"	"TOR_OTHER_CZ_NAME"	"BEGIN_RANGE"
##	[40]	"BEGIN_AZIMUTH"	"BEGIN_LOCATION"	"END_RANGE"
##	[43]	"END_AZIMUTH"	"END_LOCATION"	"BEGIN_LAT"
##	[46]	"BEGIN_LON"	"END_LAT"	"END_LON"
##	[49]	"EPISODE_NARRATIVE"	"EVENT_NARRATIVE"	"DATA_SOURCE"

## Example data

Each row is a separate **event**. However, often several events are grouped together within the same **episode**.

```
storms_2017 %>% nrow()
```

```
## [1] 56989
```

```
storms_2017 %>%  
  select(EVENT_ID) %>%  
  distinct() %>%  
  nrow()
```

```
## [1] 56989
```

```
storms_2017 %>%  
  select(EPISODE_ID) %>%  
  distinct() %>%  
  nrow()
```

```
## [1] 8964
```

## Example data

Some of the event types are listed by their county ID (FIPS code) ("C"), but some are listed by a forecast zone ID ("Z"). Which ID is used is given in the column CZ\_TYPE:

```
storms_2017 %>%  
  group_by(CZ_TYPE) %>%  
  count()
```

```
## # A tibble: 3 x 2  
## # Groups:   CZ_TYPE [3]  
##   CZ_TYPE      n  
##   <chr>    <int>  
## 1 2          17  
## 2 C        36933  
## 3 Z        20039
```

## Example data

For the first in-course exercise, you will clean up this data a bit for us to use in the rest of the class.

- Download and read in the data
- Limit the dataframe to: the beginning and ending dates and times, the episode ID, the event ID, the state name and FIPS, the “CZ” name, type, and FIPS, the event type, the source, and the beginning latitude and longitude and ending latitude and longitude
- Convert the beginning and ending dates to a “date-time” class (there should be one column for the beginning date-time and one for the ending date-time)
- Change state and county names to title case (e.g., “New Jersey” instead of “NEW JERSEY”)
- Limit to the events listed by county FIPS (CZ\_TYPE of “C”) and then remove the CZ\_TYPE column
- Pad the state and county FIPS with a “0” at the beginning (hint: there’s a function in `stringr` to do this) and then unite the two columns to make one `fips` column with the 5-digit county FIPS code
- Change all the column names to lower case (you may want to try the `rename_all` function for this)



# Example data

```
storms_2017 %>%  
  slice(1:3)
```

```
## # A tibble: 3 x 13
```

```
##   begin_date_time      end_date_time      episode_id event_id state fips  
##   <dtm>             <dtm>             <dbl>    <dbl> <chr> <chr>  
## 1 2017-04-06 15:09:00 2017-04-06 15:09:00    113355    678791 New ~ 34015  
## 2 2017-04-06 09:30:00 2017-04-06 09:40:00    113459    679228 Flor~ 12071  
## 3 2017-04-05 17:49:00 2017-04-05 17:53:00    113448    679268 Ohio  39057  
## # ... with 7 more variables: event_type <chr>, cz_name <chr>,  
## #   source <chr>, begin_lat <dbl>, begin_lon <dbl>, end_lat <dbl>,  
## #   end_lon <dbl>
```

## State data

There is data that comes with R on U.S. states. Use that to create a dataframe with the state name, area, and region:

```
data("state")  
us_state_info <- tibble(state = state.name,  
                        area = state.area,  
                        region = state.region)
```

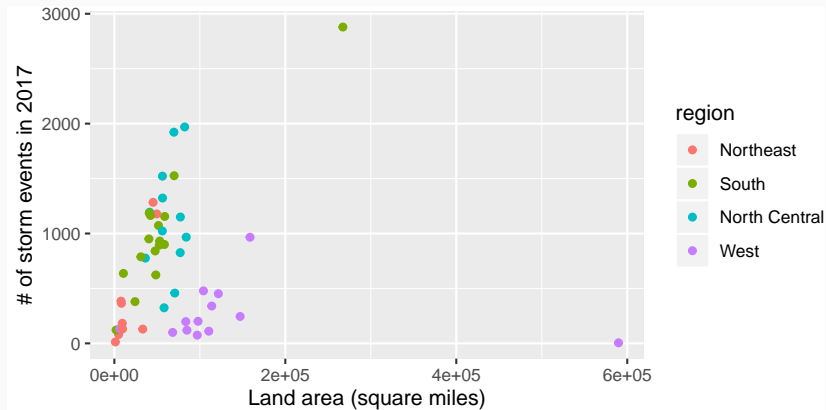
## Example data

Create a dataframe with the number of events per state in 2017. Merge in the state information dataframe you just created. Remove any states that are not in the state information dataframe.

```
state_storms <- storms_2017 %>%  
  group_by(state) %>%  
  count() %>%  
  ungroup() %>%  
  right_join(us_state_info, by = "state")
```

## Example data

Ultimately, in this group exercise, you will create a plot of state land area versus the number of storm events in the state:



We'll now take a break to do the first part of the in-course exercise.

## ggplot2 extras and extensions

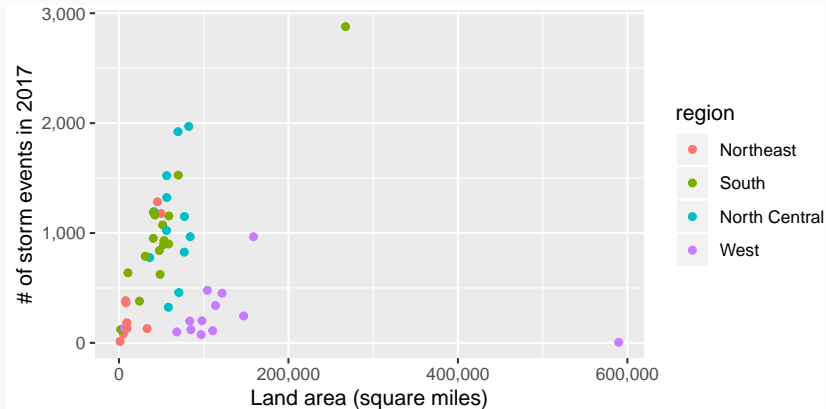
---

The `scales` package gives you a few more options for labeling with your `ggplot` scales. For example, if you wanted to change the notation for the axes in the plot of state area versus number of storm events, you could use the `scales` package to add commas to the numeric axis values.

For the rest of these slides, I've saved the `ggplot` object with out plot to the object named `storm_plot`, so we don't have to repeat that code every time.

# scales package

```
library(scales)
storm_plot +
  scale_x_continuous(labels = comma) +
  scale_y_continuous(labels = comma)
```





The scales package also includes labeling functions for:

- `dollars(labels = dollar)`
- `percent(labels = percent)`

The ggplot2 framework is set up so that others can create packages that “extend” the system, creating functions that can be added on as layers to a ggplot object.

Some of the types of extensions available include:

- More themes
- Useful additions (things that you may be able to do without the package, but that the package makes easier)
- Tools for plotting different types of data

## Where to find ggplot2 extensions

There is a gallery with links to ggplot2 extensions at <https://www.ggplot2-exts.org>.

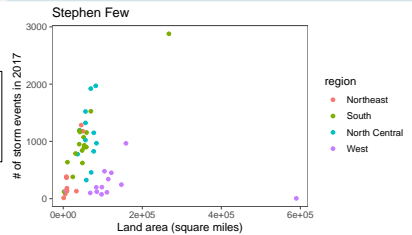
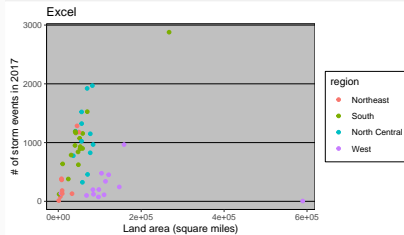
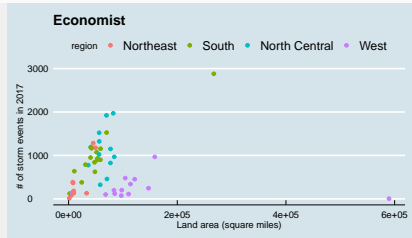
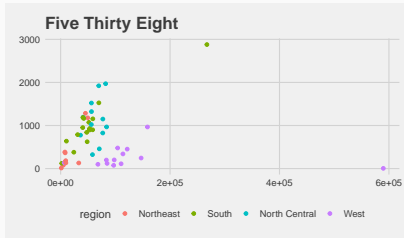
This list may not be exhaustive—there may be other extensions on CRAN or on GitHub that the package maintainer did not submit for this gallery.

You have already played around a lot with using ggplot themes to change how your graphs look.

Several people have created packages with additional themes:

- `ggthemes`
- `ggthemr`
- `ggtech`
- `ggsci`

# Some ggthemes themes

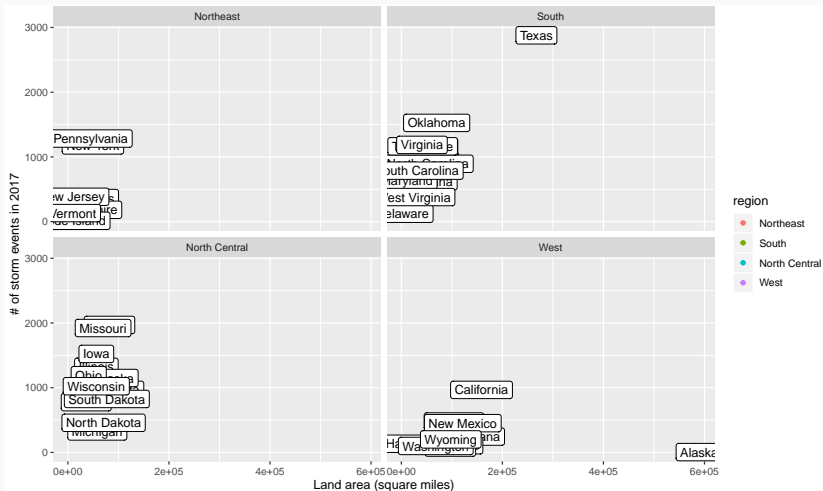


- Highlighting interesting points
- “Repelling” text labels
- Arranging plots

# ggplot2 extensions: repelling text labels

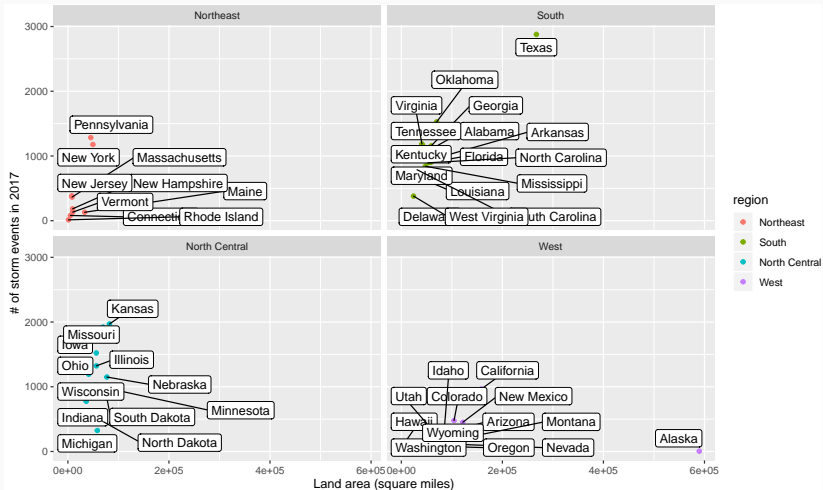
When you add labels to points on a plot, they often overlap:

```
storm_plot + facet_wrap(~ region) +  
  geom_label(aes(label = state))
```



# ggplot2 extensions: repelling text labels

```
library(ggrepel)
storm_plot + facet_wrap(~ region) +
  geom_label_repel(aes(label = state))
```



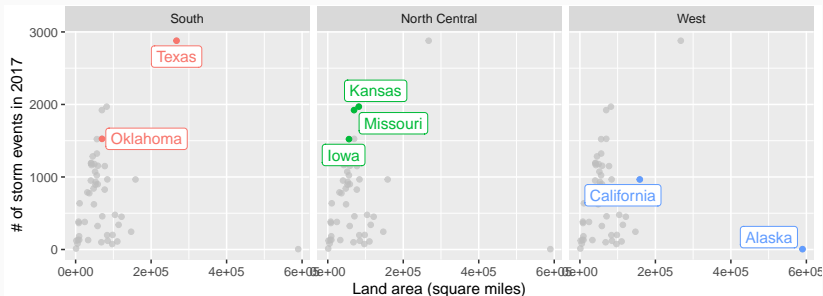


# gghighlight package

It may be too much to label every point. Instead, you may just want to highlight notable point.

You can use the `gghighlight` package to do that.

```
library(gghighlight)
storm_plot + facet_wrap(~ region) +
  gghighlight(area > 150000 | n > 1500, label_key = state)
```



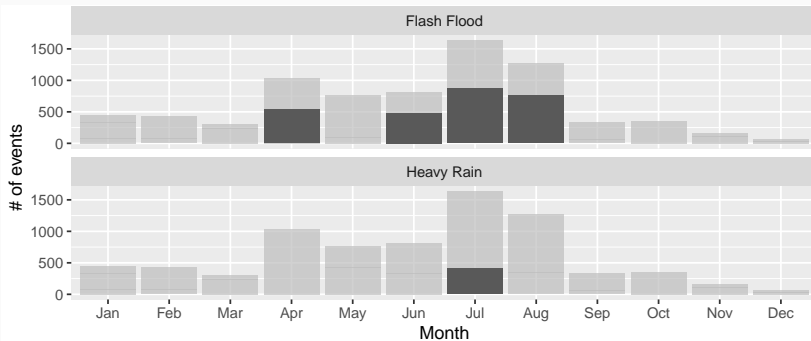
## gghighlight package

The gghighlight package also works for things like histograms. For example, you could create a dataset with the count by day-of-year of certain types of events:

```
storms_by_month <- storms_2017 %>%  
  filter(event_type %in% c("Flood", "Flash Flood", "Heavy Rain")) %>%  
  mutate(month = month(begin_date_time, label = TRUE)) %>%  
  group_by(month, event_type) %>%  
  count() %>%  
  ungroup()  
storms_by_month %>%  
  slice(1:4)  
  
## # A tibble: 4 x 3  
##   month event_type      n  
##   <ord> <chr>      <int>  
## 1 Jan   Flash Flood   113  
## 2 Jan   Flood           255  
## 3 Jan   Heavy Rain       80  
## 4 Feb   Flash Flood       65
```

# gghighlight package

```
ggplot(storms_by_month, aes(x = month, y = n, group = event_type)) +  
  geom_bar(stat = "identity") +  
  labs(x = "Month", y = "# of events") +  
  gghighlight(max(n) > 400, label_key = event_type) +  
  facet_wrap(~ event_type, ncol = 1)
```



## ggplot2 extensions: Arranging plots

You may have multiple related plots you want to have as multiple panels of a single figure.

There are a few packages that help with this. One very good one is `patchwork`.

You need to install this from GitHub:

```
devtools::install_github("thomasp85/patchwork")
```

Find out more: <https://github.com/thomasp85/patchwork#patchwork>

## ggplot2 extensions: Arranging plots

Say we want to plot seasonal patterns in events in the five counties with the highest number of events in 2017. We can use `dplyr` to figure out these counties:

```
top_counties <- storms_2017 %>%  
  group_by(fips, state, cz_name) %>%  
  count() %>%  
  ungroup() %>%  
  top_n(5, wt = n)
```

## ggplot2 extensions: Arranging plots

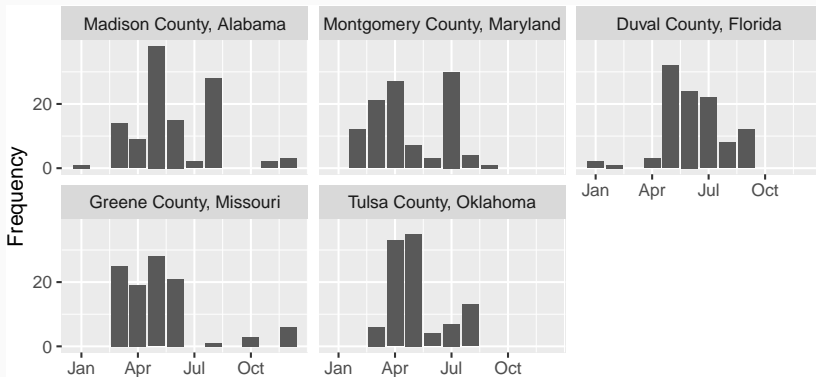
Then create a plot with the time patterns:

```
library(forcats)
top_counties_month <- storms_2017 %>%
  semi_join(top_counties, by = "fips") %>%
  mutate(month = month(begin_date_time),
         county = paste(cz_name, " County", state, sep = "")) %>%
  count(county, month) %>%
  ggplot(aes(x = month, y = n)) +
  geom_bar(stat = "identity") +
  facet_wrap(~ fct_reorder(county, n, .fun = sum, .desc = TRUE), nrow = 2) +
  scale_x_continuous(name = "", breaks = c(1, 4, 7, 10),
                    labels = c("Jan", "Apr", "Jul", "Oct")) +
  scale_y_continuous(name = "Frequency", breaks = c(0, 20))
```

# ggplot2 extensions: Arranging plots

Here's this plot:

```
top_counties_month
```



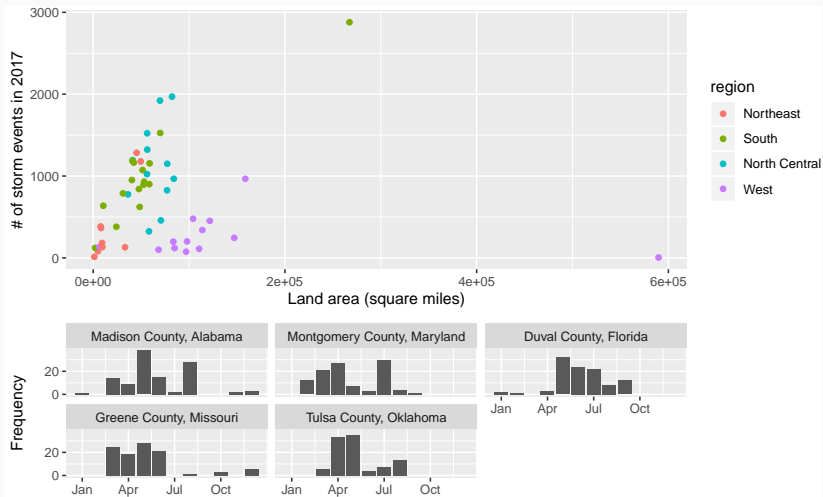
## ggplot2 extensions: Arranging plots

Now that you have two ggplot objects (`storm_plot` and `top_counties_month`), you can use `patchwork` to put them together:

```
library(patchwork)
storm_plot +
  top_counties_month +
  plot_layout(ncol = 1, heights = c(2, 1))
```



# ggplot2 extensions: Arranging plots

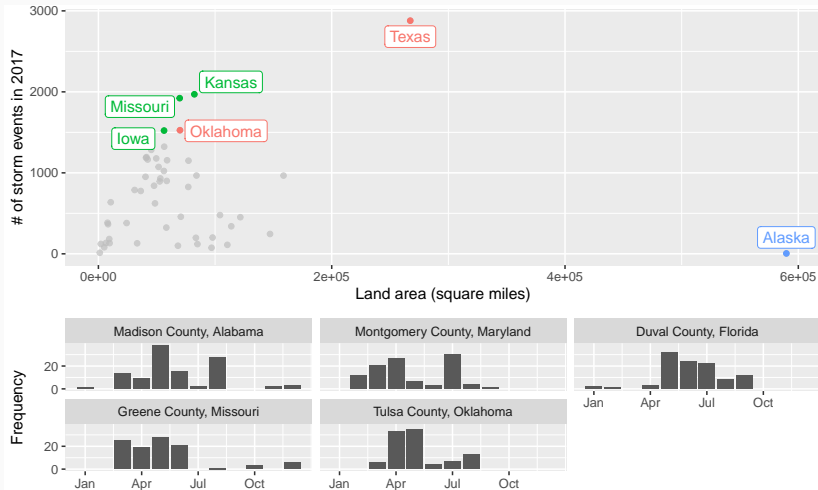


## ggplot2 extensions: Arranging plots

A slightly fancier version:

```
(storm_plot + theme(legend.position = "top") +  
  gghighlight(n > 1500 | area > 200000,  
    label_key = state)) +  
top_counties_month +  
plot_layout(ncol = 1, heights = c(2, 1))
```

# ggplot2 extensions: Arranging plots



## ggplot2 extensions: Arranging plots

Other packages for arranging ggplot objects:

- `gridExtra`
- `cowplot`

We'll take a break now to do the second part of the in-course exercise.

# Heat maps

**Heat maps** are helpful for exploring the measurements for many variables, including how similar they are, by showing small boxes of color.

In `ggplot2`, there is a `geom_tile` that is useful for creating heatmaps.

# Heat maps

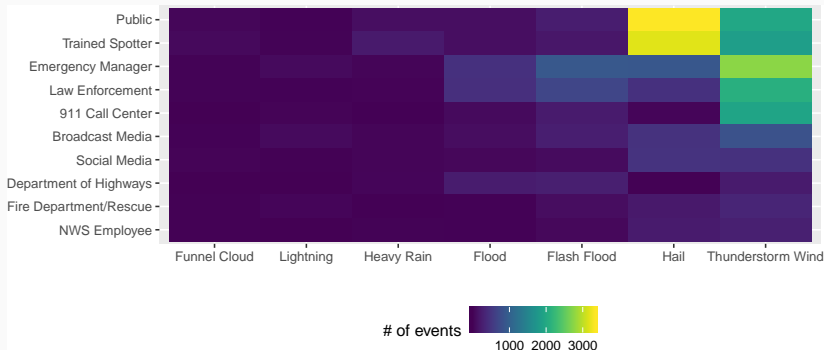
For example, we might want to look at how many events of each type were reported by each type of source. First, we can summarize the data to get this data:

```
type_by_source <- storms_2017 %>%  
  filter(event_type %in% c("Thunderstorm Wind", "Heavy Rain",  
                           "Funnel Cloud", "Flash Flood",  
                           "Flood", "Hail", "Lightning")) %>%  
  filter(source %in% c("911 Call Center", "Broadcast Media",  
                       "Department of Highways",  
                       "Emergency Manager", "Fire Department/Rescue",  
                       "Law Enforcement", "NWS Employee", "Public",  
                       "Social Media", "Trained Spotter")) %>%  
  group_by(event_type, source) %>% count() %>% ungroup()  
type_by_source %>% slice(1:3)
```

```
## # A tibble: 3 x 3  
##   event_type source          n  
##   <chr>      <chr>      <int>  
## 1 Flash Flood 911 Call Center    238  
## 2 Flash Flood Broadcast Media  268  
## 3 Flash Flood Department of Highways 276
```

# Heat maps

```
library(viridis); library(forcats)
ggplot(type_by_source,
       aes(x = fct_reorder(event_type, n, .fun = sum),
           y = fct_reorder(source, n, .fun = sum))) +
  geom_tile(aes(fill = n)) +
  scale_fill_viridis() +
  labs(x = "", y = "", fill = "# of events") +
  theme(legend.position = "bottom")
```

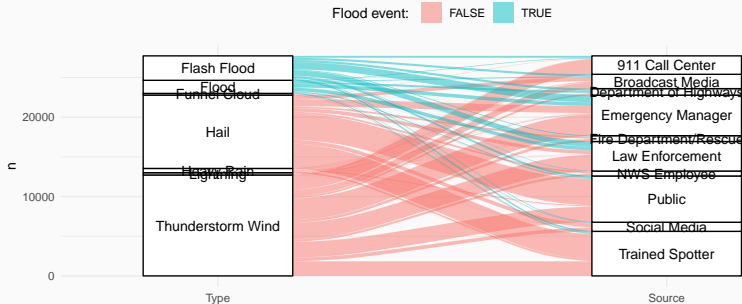




# Alluvial plots

The package `ggalluvial` allows you to make **alluvial plots**.

```
library(ggalluvial)
ggplot(data = type_by_source,
       aes(axis1 = event_type, axis2 = source, y = n)) +
  scale_x_discrete(limits = c("Type", "Source"), expand = c(.1, .05)) +
  geom_alluvium(aes(fill = event_type %in% c("Flash Flood", "Flood"))) +
  theme_minimal() + geom_stratum() +
  geom_text(stat = "stratum", label.strata = TRUE) +
  theme(legend.position = "top") + labs(fill = "Flood event: ")
```



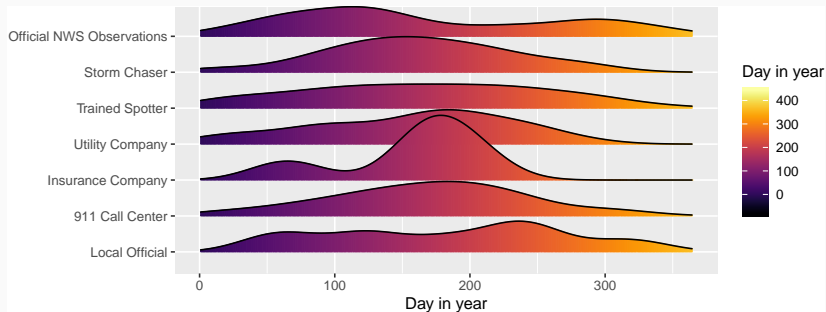
The ggribes package lets you plot the distribution of several levels of a factor across another variable.

Create a dataframe with the timing versus the source of the storm:

```
library(ggribges)
library(forcats)
storms_by_yday <- storms_2017 %>%
  filter(source %in% c("River/Stream Gages", "Insurance Company",
                      "Airline Pilot", "Trained Spotter",
                      "Storm Chaser", "Local Official",
                      "Official NWS Observations", "Utility Company",
                      "PostOffice", "911 Call Center")) %>%
  mutate(yday = yday(begin_date_time)) %>%
  group_by(yday, source) %>%
  count() %>%
  ungroup()
```

# ggridges package

```
library(viridis)
ggplot(storms_by_yday, aes(x = yday,
                           y = fct_reorder(source, yday,
                                             .fun = median, .desc = TRUE),
                           fill = ..x..)) +
  geom_density_ridges_gradient() +
  scale_fill_viridis(name = "Day in year", option = "B") +
  xlim(c(0, 365)) +
  labs(x = "Day in year", y = "")
```



Before next week's class, be sure to do the following:

- Register for GitHub: <https://github.com>
- Download and install git: <https://git-scm.com>

# Mapping in the tidyverse

---

# Simple features

sf objects: “Simple features”

- R framework that is in active development
- There will likely be changes in the near future
- Plays very well with tidyverse functions, including dplyr and ggplot2 tools

```
library(sf)
```

## Simple features

To show simple features, we'll pull in the Colorado county boundaries from the U.S. Census.

To do this, we'll use the `tigris` package, which accesses the U.S. Census API. It allows you to pull geographic data for U.S. counties, states, tracts, voting districts, roads, rails, and a number of other geographies.

To learn more about the `tigris` package, check out this article:

<https://journal.r-project.org/archive/2016/RJ-2016-043/index.html>



## Simple features

With `tigris`, you can read in data for county boundaries using the `counties` function.

We'll use the option `class = "sf"` to read these spatial dataframes in as `sf` objects.

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

class(co_counties)

## [1] "sf"          "data.frame"
```

# Simple features

You can think of an sf object as a dataframe, but with one special column called geometry.

```
co_counties %>%  
  slice(1:3)
```

```
## Simple feature collection with 3 features and 9 fields  
## geometry type:  MULTIPOLYGON  
## dimension:      XY  
## bbox:           xmin: -108.379 ymin: 38.66805 xmax: -105.0528 ymax: 40.48625  
## epsg (SRID):    4269  
## proj4string:    +proj=longlat +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +no_defs  
##   STATEFP COUNTYFP COUNTYNS   AFFGEOID GEOID   NAME LSAD   ALAND  
## 1      08      013 00198122 05000000US08013 08013 Boulder  06 1881281634  
## 2      08      029 00198130 05000000US08029 08029   Delta  06 2958099789  
## 3      08      049 00198140 05000000US08049 08049   Grand  06 4782260517  
##      AWATER      geometry  
## 1 36521204 MULTIPOLYGON (((-105.0826 3...  
## 2 16777978 MULTIPOLYGON (((-108.379 38...  
## 3 59902382 MULTIPOLYGON (((-106.6527 4...
```

## Simple features

The geometry column has a special class (sfc):

```
class(co_counties$geometry)
```

```
## [1] "sfc_MULTIPOLYGON" "sfc"
```

# Simple features

You'll notice there's some extra stuff up at the top, too:

- **Geometry type:** Points, polygons, lines
- **Dimension:** Often two-dimensional, but can go up to four (if you have, for example, time for each measurement and some measure of measurement error / uncertainty)
- **Bounding box (bbox):** The x- and y-range of the data included
- **EPSG:** The EPSG Geodetic Parameter Dataset code for the Coordinate Reference Systems
- **Projection (proj4string):** How the data is currently projected, includes projection (“+proj”) and datum (“+datum”)

## Simple features

You can pull some of this information out of the geometry column. For example, you can pull out the coordinates of the bounding box:

```
st_bbox(co_counties$geometry)           # For all counties
```

```
##           xmin           ymin           xmax           ymax
## -109.06025    36.99243 -102.04152    41.00344
```

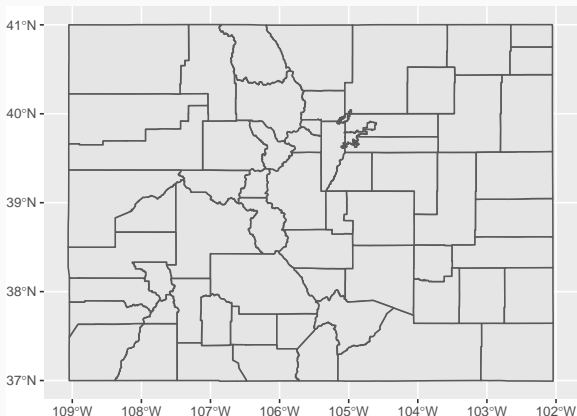
```
st_bbox(co_counties$geometry[1]) # Just for first county
```

```
##           xmin           ymin           xmax           ymax
## -105.69436    39.91289 -105.05277    40.26278
```

# Simple features

You can add sf objects to ggplot objects using `geom_sf`:

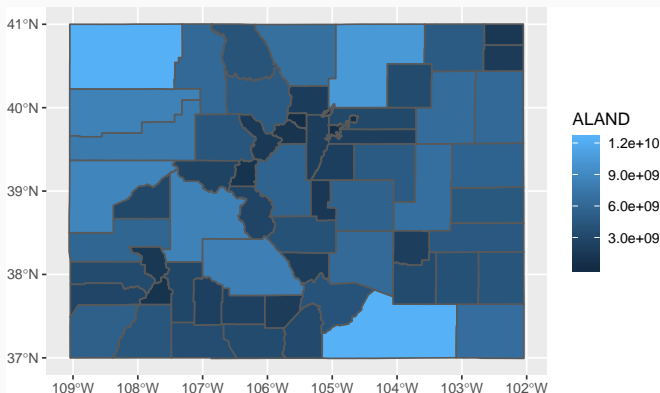
```
library(ggplot2)
ggplot() +
  geom_sf(data = co_counties)
```



# Simple features

You can map one of the columns in the `sf` object to the fill aesthetic to make a **choropleth**:

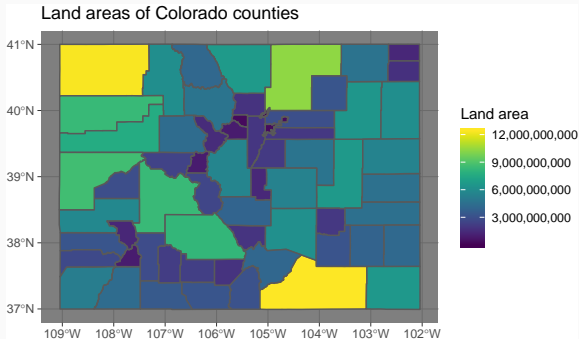
```
ggplot() +  
  geom_sf(data = co_counties, aes(fill = ALAND))
```



# Simple features

You can use all your usual ggplot tricks with this:

```
ggplot() +  
  geom_sf(data = co_counties, aes(fill = ALAND)) +  
  scale_fill_viridis(name = "Land area", label = comma) +  
  ggtitle("Land areas of Colorado counties") +  
  theme_dark()
```





# Simple features

Because simple features are a special type of dataframe, you can also use a lot of dplyr tricks.

For example, you could pull out just Larimer County, CO:

```
larimer <- co_counties %>%  
  filter(NAME == "Larimer")  
larimer
```

```
## Simple feature collection with 1 feature and 9 fields
```

```
## geometry type:  MULTIPOLYGON
```

```
## dimension:      XY
```

```
## bbox:           xmin: -106.1954 ymin: 40.25788 xmax: -104.9431 ymax: 40.9982
```

```
## epsg (SRID):    4269
```

```
## proj4string:     +proj=longlat +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +no_defs
```

```
##   STATEFP COUNTYFP COUNTYNS   AFFGEOID GEOID   NAME LSAD   ALAND
```

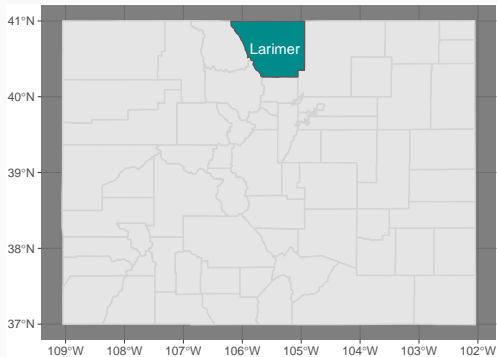
```
## 1      08      069 00198150 05000000US08069 08069 Larimer  06 6723022936
```

```
##      AWATER      geometry
```

```
## 1 99012625 MULTIPOLYGON (((-106.1954 4...
```

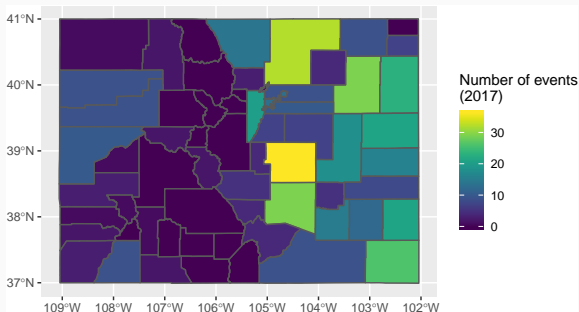
# Simple features

```
ggplot() +  
  geom_sf(data = co_counties, color = "lightgray") +  
  geom_sf(data = larimer, fill = "darkcyan") +  
  geom_sf_text(data = larimer, aes(label = NAME),  
               color = "white") +  
  theme_dark() + labs(x = "", y = "")
```



# Simple features

This operability with tidyverse functions means that you should now be able to figure out how to create a map of the number of events listed in the NOAA Storm Events database (of those listed by county) for each county in Colorado:

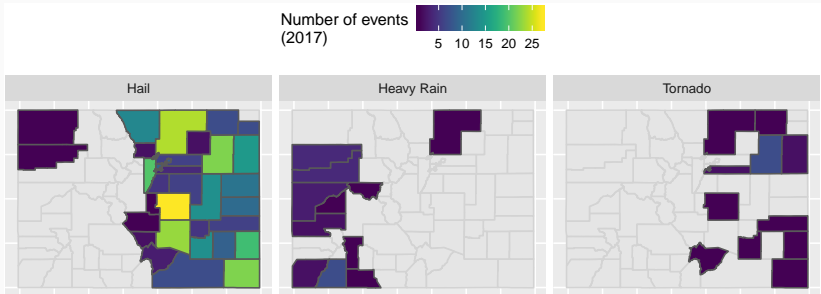


## In-course exercise

We'll now take a break for the next part of the in-course exercise. For this, create the map shown on the previous slide.

## In-course exercise

If you have time, try this one, too. It shows the number of three certain types of events by county. If a county had no events, it's shown in gray (as having a missing value when you count up the events that did happen).



## Simple features

```
co_event_counts <- storms_2017 %>%  
  filter(state == "Colorado") %>%  
  group_by(fips) %>%  
  count() %>%  
  ungroup()  
  
co_county_events <- co_counties %>%  
  mutate(fips = paste(STATEFP, COUNTYFP, sep = "")) %>%  
  full_join(co_event_counts, by = "fips") %>%  
  mutate(n = ifelse(!is.na(n), n, 0))  
  
ggplot() +  
  geom_sf(data = co_county_events, aes(fill = n)) +  
  scale_fill_viridis(name = "Number of events\n(2017)")
```

# Simple features

```
co_event_counts <- storms_2017 %>%  
  filter(state == "Colorado") %>%  
  filter(event_type %in% c("Tornado", "Heavy Rain", "Hail")) %>%  
  group_by(fips, event_type) %>%  
  count() %>%  
  ungroup()
```

```
co_county_events <- co_counties %>%  
  mutate(fips = paste(STATEFP, COUNTYFP, sep = "")) %>%  
  right_join(co_event_counts, by = "fips")
```

```
ggplot() +  
  geom_sf(data = co_counties, color = "lightgray") +  
  geom_sf(data = co_county_events, aes(fill = n)) +  
  scale_fill_viridis(name = "Number of events\n(2017)") +  
  theme(legend.position = "top") +  
  facet_wrap(~ event_type, ncol = 3) +  
  theme(axis.line = element_blank(),  
        axis.text = element_blank(),  
        axis.ticks = element_blank())
```

# State boundaries

The `tigris` package allows you to pull state boundaries, as well, but on some computers mapping these seems to take a really long time.

Instead, for now I recommend that you pull the state boundaries using base R's `maps` package and convert that to an `sf` object:

```
library(maps)
us_states <- map("state", plot = FALSE, fill = TRUE) %>%
  st_as_sf()
```



# State boundaries

You can see these borders include an ID column that you can use to join by state:

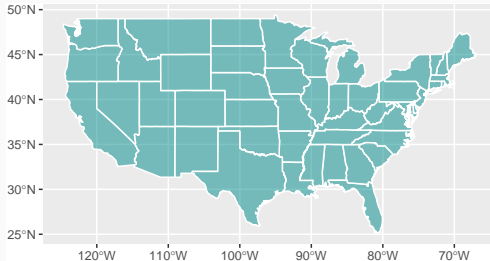
```
head(us_states)
```

```
## Simple feature collection with 6 features and 1 field
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -124.3834 ymin: 30.24071 xmax: -71.78015 ymax: 42.0493
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
##               geometry      ID
## 1 MULTIPOLYGON (((-87.46201 3...  alabama
## 2 MULTIPOLYGON (((-114.6374 3...  arizona
## 3 MULTIPOLYGON (((-94.05103 3...  arkansas
## 4 MULTIPOLYGON (((-120.006 42...  california
## 5 MULTIPOLYGON (((-102.0552 4...  colorado
## 6 MULTIPOLYGON (((-73.49902 4...  connecticut
```

# State boundaries

As with other `sf` objects, you can map these state boundaries using `ggplot`:

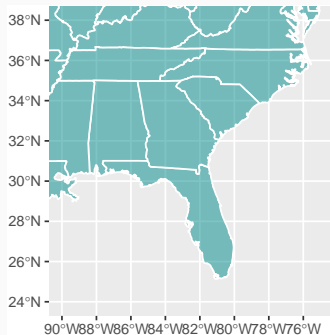
```
ggplot() +  
  geom_sf(data = us_states, color = "white",  
          fill = "darkcyan", alpha = 0.5)
```



# State boundaries

As a note, you can use `xlim` and `ylim` with these plots, but remember that the x-axis is longitude in degrees West, which are negative:

```
ggplot() +  
  geom_sf(data = us_states, color = "white",  
          fill = "darkcyan", alpha = 0.5) +  
  xlim(c(-90, -75)) + ylim(c(24, 38))
```



# Basics of creating an `sf` object

You can create an `sf` object from a regular dataframe.

You just need to specify:

1. The coordinate information (which columns are longitudes and latitudes)
2. The Coordinate Reference System (CRS) (how to translate your coordinates to places in the world)

For the CRS, if you are mapping the new `sf` object with other, existing `sf` objects, make sure that you use the same CRS for all `sf` objects.

# Coordinate reference system

Spatial objects can have different Coordinate Reference Systems (CRSs). CRSs can be *geographic* (e.g., WGS84, for longitude-latitude data) or *projected* (e.g., UTM, NADS83).

There is a website that lists projection strings and can be useful in setting projection information or re-projecting data:

<http://www.spatialreference.org>

Here is an excellent resource on projections and maps in R from Melanie Frazier: <https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/OverviewCoordinateReferenceSystems.pdf>

## Basics of creating an sf object

Let's look at floods in Colorado. First, clean up the data:

```
co_floods <- storms_2017 %>%  
  filter(state == "Colorado" &  
         event_type %in% c("Flood", "Flash Flood")) %>%  
  select(begin_date_time, event_id, begin_lat:end_lon) %>%  
  gather(key = "key", value = "value",  
         -begin_date_time, -event_id) %>%  
  separate(key, c("time", "key")) %>%  
  spread(key = key, value = value)
```

## Basics of creating an sf object

There are now two rows per event, one with the starting location and one with the ending location:

```
co_floods %>%  
  slice(1:5)
```

```
## # A tibble: 5 x 5  
##   begin_date_time      event_id time    lat    lon  
##   <dtm>              <dbl> <chr> <dbl> <dbl>  
## 1 2017-05-08 16:00:00   693374 begin  40.3 -105.  
## 2 2017-05-08 16:00:00   693374 end    40.5 -104.  
## 3 2017-05-10 15:00:00   686479 begin  38.1 -105.  
## 4 2017-05-10 15:00:00   686479 end    38.1 -105.  
## 5 2017-05-10 15:20:00   686480 begin  38.2 -105.
```

# Basics of creating an sf object

Change to an sf object by saying which columns are the coordinates and setting a CRS:

```
co_floods <- st_as_sf(co_floods, coords = c("lon", "lat")) %>%  
  st_set_crs(4269)  
co_floods %>% slice(1:3)
```

```
## Simple feature collection with 3 features and 3 fields
```

```
## geometry type: POINT
```

```
## dimension: XY
```

```
## bbox: xmin: -105.0496 ymin: 38.1167 xmax: -104.39 ymax: 40
```

```
## epsg (SRID): 4269
```

```
## proj4string: +proj=longlat +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +n
```

```
## # A tibble: 3 x 4
```

```
## begin_date_time event_id time geometry
```

```
## <dtm> <dbl> <chr> <POINT [°]>
```

```
## 1 2017-05-08 16:00:00 693374 begin (-104.76 40.32)
```

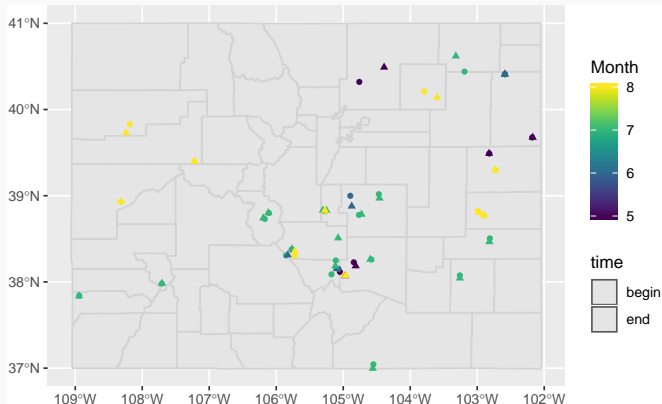
```
## 2 2017-05-08 16:00:00 693374 end (-104.39 40.49)
```

```
## 3 2017-05-10 15:00:00 686479 begin (-105.0496 38.1167)
```



# Basics of creating an sf object

```
ggplot() +  
  geom_sf(data = co_counties, color = "lightgray") +  
  geom_sf(data = co_floods, aes(color = month(begin_date_time),  
                                shape = time)) +  
  scale_color_viridis(name = "Month")
```



## Changing from points to lines

If you want to show lines instead of points, group by the appropriate ID and then summarize within each event to get a line:

```
co_floods <- co_floods %>%  
  group_by(event_id) %>%  
  summarize(month = month(first(begin_date_time)),  
            do_union = FALSE) %>%  
  st_cast("LINESTRING")
```

## Changing from points to lines

```
head(co_floods)
```

```
## Simple feature collection with 6 features and 2 fields
```

```
## geometry type:  LINESTRING
```

```
## dimension:      XY
```

```
## bbox:           xmin: -105.8286 ymin: 38.0708 xmax: -104.39 y
```

```
## epsg (SRID):    4269
```

```
## proj4string:     +proj=longlat +ellps=GRS80 +towgs84=0,0,0,0,0
```

```
## # A tibble: 6 x 3
```

```
##   event_id month                geometry
```

```
##   <dbl> <dbl>                <LINESTRING [°]>
```

```
## 1  686479      5 (-105.0496 38.1167, -104.9687 38.0708)
```

```
## 2  686480      5 (-104.8425 38.2275, -104.8137 38.1854)
```

```
## 3  693306      6 (-104.8947 38.999, -104.8734 38.8783)
```

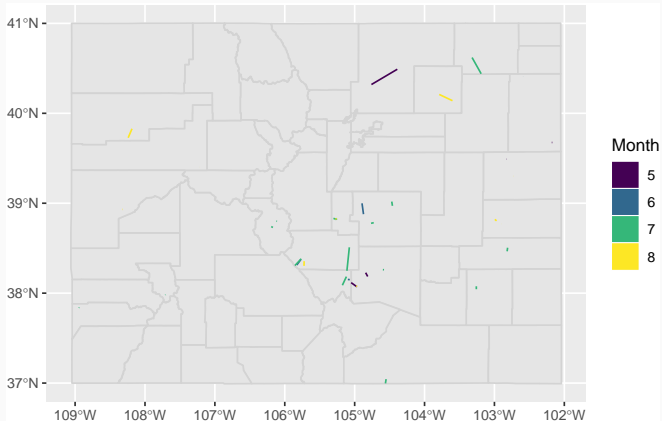
```
## 4  693374      5 (-104.76 40.32, -104.39 40.49)
```

```
## 5  693444      6 (-105.7688 38.3753, -105.8286 38.3127)
```

```
## 6  693449      6 (-105.07 38.15, -105.0973 38.1524)
```

# Changing from points to lines

```
ggplot() +  
  geom_sf(data = co_counties, color = "lightgray") +  
  geom_sf(data = co_floods,  
    aes(color = factor(month), fill = factor(month))) +  
  scale_fill_viridis(name = "Month", discrete = TRUE) +  
  scale_color_viridis(name = "Month", discrete = TRUE)
```



## Reading from GIS files

You can also create `sf` objects by reading in data from files you would normally use for GIS.

For example, you can read in an `sf` object from a shapefile, which is a format often used for GIS in which a collection of several files jointly store geographic data. The files making up a shapefile can include:

- “.shp”: The coordinates defining the shape of each geographic object. For a point, this would be a single coordinate (e.g., latitude and longitude). For lines and polygons, there will be multiple coordinates per geographic object.
- “.prf”: Information on the projection of the data (how to get from the coordinates to a place in the world).
- “.dbf”: Data that goes along with each geographical object. For example, earlier we looked at data on counties, and one thing measured for each county was its land area. Characteristics like that would be included in the “.dbf” file in a shapefile.

Often, with geographic data, you will be given the option to download a compressed file (e.g., a zipped file). When you unzip the folder, it will include a number of files in these types of formats (".shp", ".prf", ".dbf", etc.).

Sometimes, that single folder will include multiple files from each extension. For example, it might have several files that end with ".shp". In this case, you have multiple **layers** of geographic information you can read in.

## Reading in from GIS files

We've been looking at data on storms from NOAA for 2017. As an example, let's try to pair that data up with some from the National Hurricane Center for the same year.

The National Hurricane Center allows you to access a variety of GIS data through the webpage <https://www.nhc.noaa.gov/gis/?text>.

Let's pull some data on Hurricane Harvey in 2017 and map it with information from the NOAA Storm Events database.

## Reading in from GIS files

On <https://www.nhc.noaa.gov/gis/?text>, go to the section called “Preliminary Best Track”. Select the year 2017. Then select “Hurricane Harvey” and download “al092017\_best\_track.zip”.

Depending on your computer, you may then need to unzip this file (many computers will unzip it automatically). Base R has a function called `unzip` that can help with this.

You’ll then have a folder with a number of different files in it. Move this folder somewhere that is convenient for the working directory you use for class. For example, I moved it into the “data” subdirectory of the working directory I use for the class.



# Reading in from GIS files

You can use `list.files` to see all the files in this unzipped folder:

```
list.files("../data/al092017_best_track/")
```

```
## [1] "al092017_lin.dbf"          "al092017_lin.prj"
## [3] "al092017_lin.shp"          "al092017_lin.shp.xml"
## [5] "al092017_lin.shx"          "al092017_pts.dbf"
## [7] "al092017_pts.prj"          "al092017_pts.shp"
## [9] "al092017_pts.shp.xml"      "al092017_pts.shx"
## [11] "al092017_radii.dbf"         "al092017_radii.prj"
## [13] "al092017_radii.shp"         "al092017_radii.shp.xml"
## [15] "al092017_radii.shx"         "al092017_windswath.dbf"
## [17] "al092017_windswath.prj"     "al092017_windswath.shp"
## [19] "al092017_windswath.shp.xml" "al092017_windswath.shx"
```

## Reading from GIS files

You can use `st_layers` to find out the available layers in a shapefile directory:

```
st_layers("../data/al092017_best_track/")
```

```
## Driver: ESRI Shapefile
```

```
## Available layers:
```

##	layer_name	geometry_type	features	fields
## 1	al092017_windswath	Polygon	4	6
## 2	al092017_radii	Polygon	61	9
## 3	al092017_lin	Line String	17	3
## 4	al092017_pts	Point	74	15

## Reading from GIS files

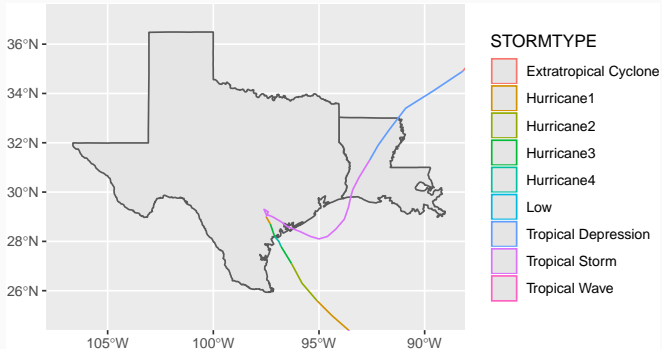
Once you know which layer you want, you can use `read_sf` to read it in as an `sf` object:

```
harvey_track <- read_sf("../data/al092017_best_track/",  
                        layer = "al092017_lin")  
head(harvey_track)
```

```
## Simple feature collection with 6 features and 3 fields  
## geometry type:  LINESTRING  
## dimension:      XY  
## bbox:           xmin: -92.3 ymin: 13 xmax: -45.8 ymax: 21.4  
## epsg (SRID):    NA  
## proj4string:     +proj=longlat +a=6371200 +b=6371200 +no_defs  
## # A tibble: 6 x 4  
##   STORMNUM STORMTYPE      SS  
##   <dbl> <chr>          <int>          <L  
## 1         9 Low              0 (-45.8 13.7, -47.4 13.7, -49 1  
## 2         9 Tropical Dep~    0          (-52 13.4, -53.4
```

# Reading from GIS files

```
ggplot() +  
  geom_sf(data = filter(us_states, ID %in% c("texas", "louisiana"  
  geom_sf(data = harvey_track, aes(color = STORMTYPE)) +  
  xlim(c(-107, -89)) + ylim(c(25, 37))
```



# Reading from GIS files

You can read in other layers:

```
harvey_windswath <- read_sf("../data/al092017_best_track/",  
                             layer = "al092017_windswath")
```

```
head(harvey_windswath)
```

```
## Simple feature collection with 4 features and 6 fields
```

```
## geometry type: POLYGON
```

```
## dimension: XY
```

```
## bbox: xmin: -98.66872 ymin: 12.94564 xmax: -54.58527 ymax: 31.1589
```

```
## epsg (SRID): NA
```

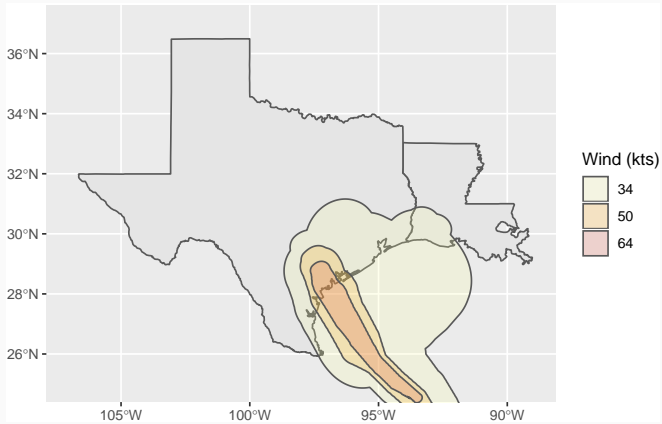
```
## proj4string: +proj=longlat +a=6371200 +b=6371200 +no_defs
```

```
## # A tibble: 4 x 7
```

		RADII	STORMID	BASIN	STORMNUM	STARTDTG	ENDDTG	geometry
	<dbl>	<chr>	<chr>		<dbl>	<chr>	<chr>	<POLYGON [°]>
## 1	34	al092017	AL		9	2017081~	20170~	((-65.68199 14.50528, -65.~
## 2	34	al092017	AL		9	2017082~	20170~	((-96.22456 31.15752, -96.~
## 3	50	al092017	AL		9	2017082~	20170~	((-97.32707 29.60604, -97.~
## 4	64	al092017	AL		9	2017082~	20170~	((-97.20689 29.08475, -97.~

# Reading from GIS files

```
ggplot() +  
  geom_sf(data = filter(us_states, ID %in% c("texas", "louisiana")))) +  
  geom_sf(data = harvey_windswath,  
    aes(fill = factor(RADII)), alpha = 0.2) +  
  xlim(c(-107, -89)) + ylim(c(25, 37)) +  
  scale_fill_viridis(name = "Wind (kts)", discrete = TRUE,  
    option = "B", begin = 0.6, direction = -1)
```



The `read_sf` function is very powerful and can read in data from lots of different formats.

See Section 2 of the `sf` manual

(<https://cran.r-project.org/web/packages/sf/vignettes/sf2.html>) for more on this function.

You can find (much, much) more on working with spatial data in R online:

- **R Spatial:** <http://rspatial.org/index.html>
- **Geocomputation with R:** <https://geocompr.robinlovelace.net>



# In-course exercise

For the in-course exercise, see if you can put everything we've talked about together to create this map:

