The Other Data Types:

Factors, Date/Time, Spatial Data

Outline

Factors with forcats

Dates and Times with lubridate

Spatial Data

- Spatial geometries with sf (simple features)
- Base maps with ggmap

Factors

A *factor* is one of R's semi-basic data types that are used to represent categorical variables.

At a basic level, a factor is a vector of integers with associated levels which are characters.

```
eye_color <- factor(c("Blue", "Green", "Brown", "Blue",
"Brown"))

str(eye_color)
```

```
Factor w/ 3 levels "Blue","Brown",..: 1 3 2 1 2
```

Factors are useful when:

- You want to use categorical variables in modeling
- You want to display character vectors in non-alphabetical order

forcats

forcats is the tidyverse package for working with factors.

forcats functions are split up into 4 different types:

1. creating/inspecting/combining factors
2. change level order
3. change level value
4. add/drop levels

1: `fct_count()`, `fct_unique()`, `fct_c()`, `fct_unify()`

2: `fct_relevel()`, `fct_infreq()`, `fct_inorder()`, `fct_shift()`, `fct_reorder()`, etc.

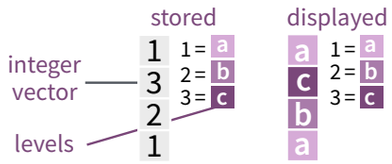
3: `fct_recode()`, `fct_anon()`, `fct_lump()`, `fct_collapse()`, `fct_other()`

4: `fct_drop()`, `fct_expand()`, `fct_explicit_na()`

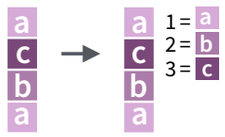
Factors with forcats : : **CHEAT SHEET**

The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors

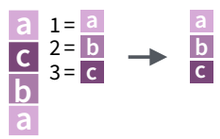


R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the values associated with them.



Create a factor with factor()

factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA) Convert a vector to a factor. Also **as_factor**.
f <- factor(c("a", "c", "b", "a"), levels = c("a", "b", "c"))

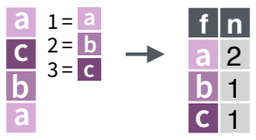


Return its levels with levels()

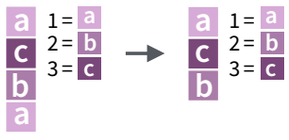
levels(x) Return/set the levels of a factor. *levels(f); levels(f) <- c("x","y","z")*

Use unclass() to see its structure

Inspect Factors

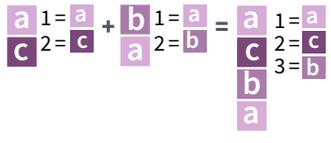


fct_count(f, sort = FALSE) Count the number of values with each level. *fct_count(f)*

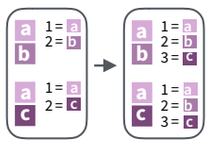


fct_unique(f) Return the unique values, removing duplicates. *fct_unique(f)*

Combine Factors

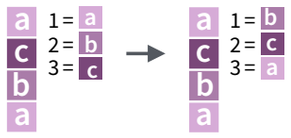


fct_c(…) Combine factors with different levels.
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
fct_c(f1, f2)

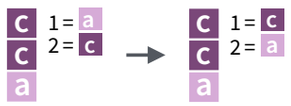


fct_unify(fs, levels = lvls_union(fs)) Standardize levels across a list of factors.
fct_unify(list(f2, f1))

Change the order of levels

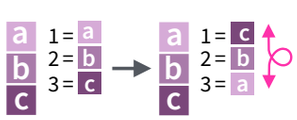


fct_relevel(.f, ..., after = 0L) Manually reorder factor levels.
fct_relevel(f, c("b", "c", "a"))

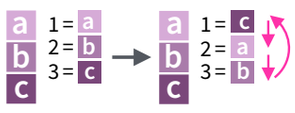


fct_infreq(f, ordered = NA) Reorder levels by the frequency in which they appear in the data (highest frequency first).
f3 <- factor(c("c", "c", "a"))
fct_infreq(f3)

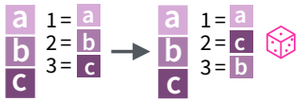
fct_inorder(f, ordered = NA) Reorder levels by order in which they appear in the data.
fct_inorder(f2)



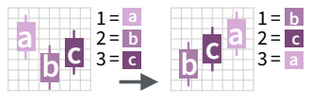
fct_rev(f) Reverse level order.
f4 <- factor(c("a","b","c"))
fct_rev(f4)



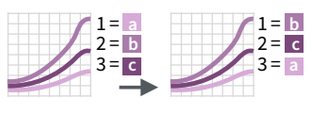
fct_shift(f) Shift levels to left or right, wrapping around end.
fct_shift(f4)



fct_shuffle(f, n = 1L) Randomly permute order of factor levels.
fct_shuffle(f4)

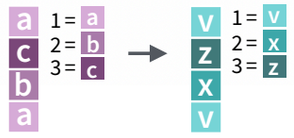


fct_reorder(.f, .x, .fun=median, ..., .desc = FALSE) Reorder levels by their relationship with another variable.
boxplot(data = iris, Sepal.Width ~ fct_reorder(Species, Sepal.Width))

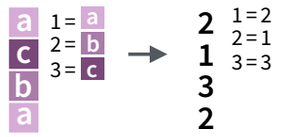


fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE) Reorder levels by their final values when plotted with two other variables.
ggplot(data = iris, aes(Sepal.Width, Sepal.Length, color = fct_reorder2(Species, Sepal.Width, Sepal.Length))) + geom_smooth()

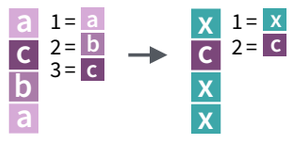
Change the value of levels



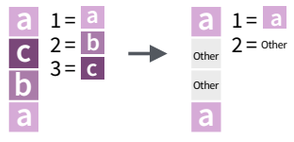
fct_recode(.f, ...) Manually change levels. Also **fct_relabel** which obeys purrr::map syntax to apply a function or expression to each level.
fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))



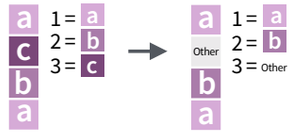
fct_anon(f, prefix = "")) Anonymize levels with random integers. *fct_anon(f)*



fct_collapse(.f, ...) Collapse levels into manually defined groups.
fct_collapse(f, x = c("a", "b"))

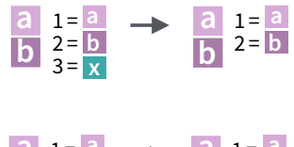


fct_lump(f, n, prop, w = NULL, other_level = "Other", ties.method = c("min", "average", "first", "last", "random", "max")) Lump together least/most common levels into a single level. Also **fct_lump_min**.
fct_lump(f, n = 1)

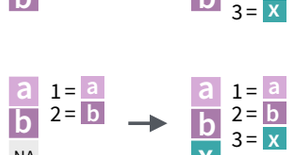


fct_other(f, keep, drop, other_level = "Other") Replace levels with "other."
fct_other(f, keep = c("a", "b"))

Add or drop levels



fct_drop(f, only) Drop unused levels.
f5 <- factor(c("a","b"),c("a","b","x"))
f6 <- fct_drop(f5)



fct_expand(f, …) Add levels to a factor. *fct_expand(f6, "x")*

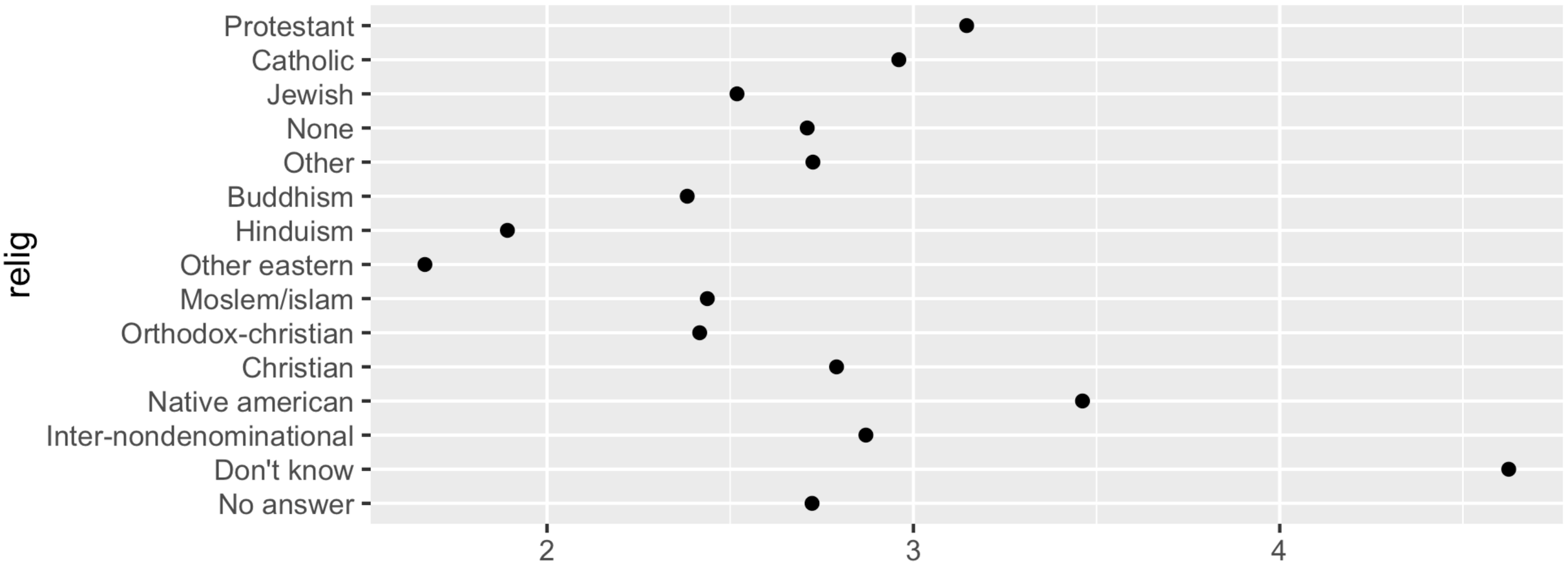
fct_explicit_na(f, na_level="(Missing)") Assigns a level to NAs to ensure they appear in plots, etc.
fct_explicit_na(factor(c("a", "b", NA)))

Use Cases

Factor are helpful in reordering characters on plots.

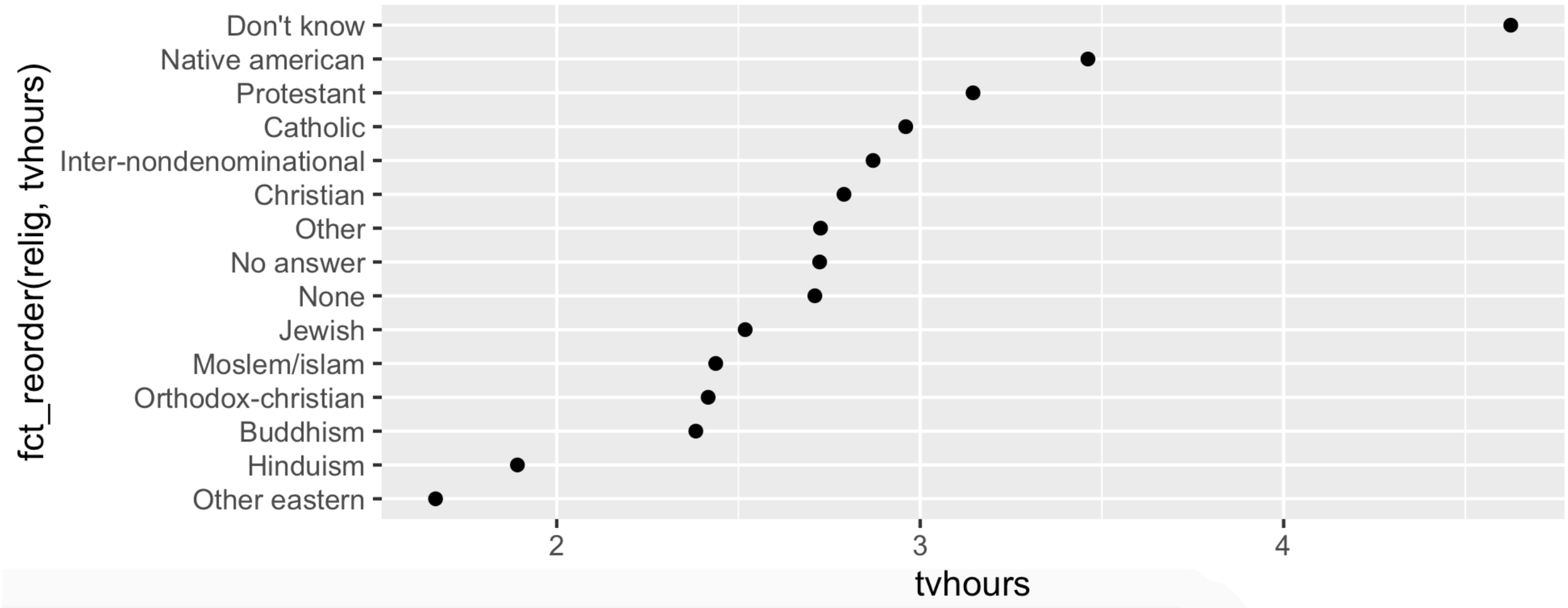
```
relig_summary <- gss_cat %>%
    group_by(relig) %>%
    summarise(
        age = mean(age, na.rm = TRUE),
        tvhours = mean(tvhours, na.rm = TRUE),
        n = n()
    )

ggplot(relig_summary, aes(tvhours, relig)) + geom_point()
```



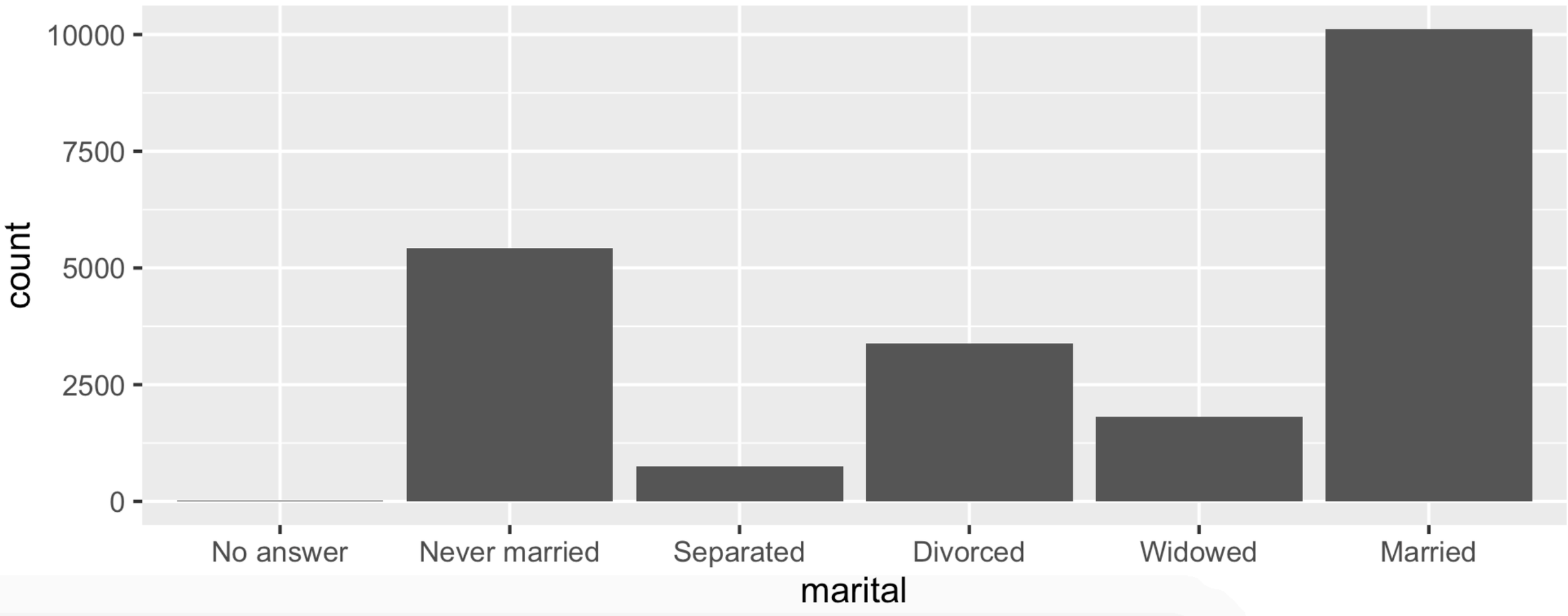
Use Cases

```
ggplot(relig_summary, aes(tvhours, fct_reorder(relig, tvhours)))
+ geom_point()
```



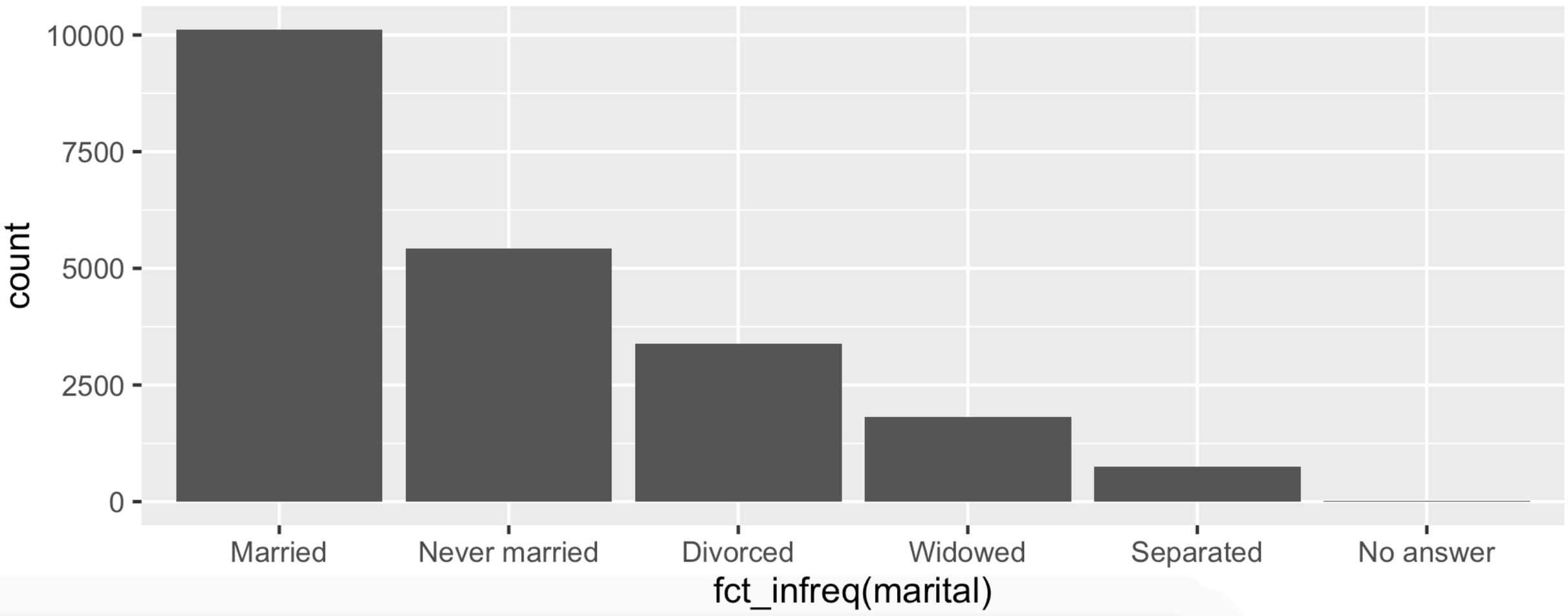
Use Cases

```
ggplot(gss_cat, aes(marital)) + geom_bar()
```



Use Cases

```
ggplot(gss_cat, aes(fct_infreq(marital))) + geom_bar()
```



Dates and Times

Dates and times in R may seem simple at first, but they can quite confusing and hard to work with.

It is challenging because you have to reconcile physical phenomena like the rotation of the earth and its orbit around the sun. In addition, you also have to consider other geopolitical issues like months, time zones, and daylight savings time, etc.

R has many base functions to deal with dates/times and so generally has good support for these types. However base functions can be odd at times, and lubridate offers a unifying structure (as tidyverse packages generally do)

Dates and Times

Support for dates/times in base R

```
as.Date("1993-6-3")
```

```
[1] "1993-06-03"
```

```
as.Date("1993-6-3") - as.Date("1993-3-14")
```

```
Time difference of 81 days
```

```
as.POSIXlt(Sys.time(), "EST")
```

```
[1] "2020-03-10 12:30:06 EST"
```

Dates and times with lubridate : : CHEAT SHEET

Date-times

2017-11-28 12:00:00
A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

dt <- ***as_datetime****(1511870400)*
"2017-11-28 12:00:00 UTC"

2017-11-28
A **date** is a day stored as the number of days since 1970-01-01

d <- ***as_date****(17498)*
"2017-11-28"

12:00:00
An hms is a **time** stored as the number of seconds since 00:00:00

*t <- hms::****as.hms****(85)*
00:01:25

PARSE DATE-TIMES (Convert strings or numbers to date-times)

1. Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
2. Use the function below whose name replicates the order. Each accepts a wide variety of input formats.

2017-11-28T14:02:00	**ymd_hms**(), **ymd_hm**(), **ymd_h**(). *ymd_hms("2017-11-28T14:02:00")*
2017-22-12 10:00:00	**ydm_hms**(), **ydm_hm**(), **ydm_h**(). *ydm_hms("2017-22-12 10:00:00")*
11/28/2017 1:02:03	**mdy_hms**(), **mdy_hm**(), **mdy_h**(). *mdy_hms("11/28/2017 1:02:03")*
1 Jan 2017 23:59:59	**dmy_hms**(), **dmy_hm**(), **dmy_h**(). *dmy_hms("1 Jan 2017 23:59:59")*
20170131	**ymd**(), **ydm**(). *ymd(20170131)*
July 4th, 2000	**mdy**(), **myd**(). *mdy("July 4th, 2000")*
4th of July '99	**dmy**(), **dym**(). *dmy("4th of July '99")*
2001: Q3	**yq**() Q for quarter. *yq("2001: Q3")*
2:01	hms::**hms**() Also lubridate::**hms**(), **hm**() and **ms**(), which return periods.* *hms::hms(sec = 0, min= 1, hours = 2)*
2017.5	**date_decimal**(decimal, tz = "UTC") *date_decimal(2017.5)*
	now(tzone = "") Current time in tz (defaults to system tz). *now()*
	today(tzone = "") Current date in a tz (defaults to system tz). *today()*
	fast_strptime() Faster strptime. *fast_strptime('9/1/01', '%y/%m/%d')*
	parse_date_time() Easier strptime. *parse_date_time("9/1/01", "ymd")*

GET AND SET COMPONENTS

Use an accessor function to get a component.

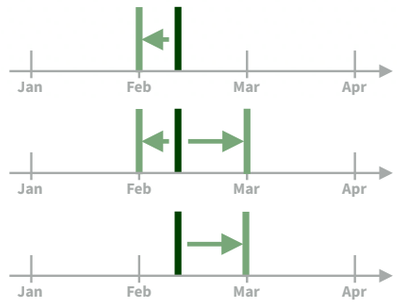
d ## "2017-11-28"
day(d) ## 28

Assign into an accessor function to change a component in place.

day(d) <- 1
d ## "2017-11-01"

2018-01-31 11:59:59	**date**(x) Date component. *date(dt)*
2018-01-31 11:59:59	**year**(x) Year. *year(dt)* **isoyear**(x) The ISO 8601 year. **epiyear**(x) Epidemiological year.
2018-01-31 11:59:59	**month**(x, label, abbr) Month. *month(dt)*
2018-01-31 11:59:59	**day**(x) Day of month. *day(dt)* **wday**(x,label,abbr) Day of week. **qday**(x) Day of quarter.
2018-01-31 11:59:59	**hour**(x) Hour. *hour(dt)*
2018-01-31 11:59:59	**minute**(x) Minutes. *minute(dt)*
2018-01-31 11:59:59	**second**(x) Seconds. *second(dt)*
	week(x) Week of the year. *week(dt)* **isoweek**() ISO 8601 week. **epiweek**() Epidemiological week.
	quarter(x, with_year = FALSE) Quarter. *quarter(dt)*
	semester(x, with_year = FALSE) Semester. *semester(dt)*
	am(x) Is it in the am? *am(dt)* **pm**(x) Is it in the pm? *pm(dt)*
	dst(x) Is it daylight savings? *dst(d)*
	leap_year(x) Is it a leap year? *leap_year(d)*
	update(object, ..., simple = FALSE) *update(dt, mday = 2, hour = 1)*

Round Date-times



floor_date(x, unit = "second")
Round down to nearest unit.
floor_date(dt, unit = "month")

round_date(x, unit = "second")
Round to nearest unit.
round_date(dt, unit = "month")

ceiling_date(x, unit = "second", change_on_boundary = NULL)
Round up to nearest unit.
ceiling_date(dt, unit = "month")

rollback(dates, roll_to_first = FALSE, preserve_hms = TRUE)
Roll back to last day of previous month. *rollback(dt)*

Stamp Date-times

stamp() Derive a template from an example string and return a new function that will apply the template to date-times. Also **stamp_date**() and **stamp_time**().

1. Derive a template, create a function
   sf <- stamp("Created Sunday, Jan 17, 1999 3:34")
2. Apply the template to dates
   sf(ymd("2010-04-05"))
   ## [1] "Created Monday, Apr 05, 2010 00:00"

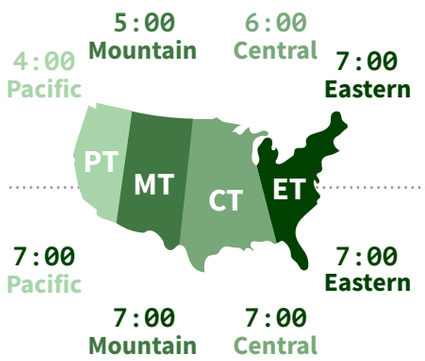
Tip: use a date with day > 12

Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns *one* time zone per vector.

Use the **UTC** time zone to avoid Daylight Savings.

OlsonNames() Returns a list of valid time zone names. *OlsonNames()*



with_tz(time, tzone = "") Get the **same date-time** in a new time zone (a new clock time).
with_tz(dt, "US/Pacific")

force_tz(time, tzone = "") Get the **same clock time** in a new time zone (a new date-time).
force_tz(dt, "US/Pacific")

lubridate

lubridate is the tidyverse package for dealing with dates/times

lubridate has a bunch of functions! Most of them deal with parsing dates and extracting important info like the duration btw two date/time objects, etc.

```
now()
```

```
[1] "2020-03-10 13:38:44 EDT"
```

```
mdy("March 16th, 2020")
```

```
[1] "2020-03-16"
```

```
mdy_hms("March 16th, 2020, 3:30:00 PM")
```

```
[1] "2020-03-16 15:30:00 UTC"
```

Spatial Data in R

A popular way to view or model data is over time or space! Let's focus on space:

Spatial data is information about a physical object that can be represented by numerical values in a geographic coordinate system. This leads to complex data structures because you have to hold the location, size, and shape of an object. These are called geometries (point, line, polygon)

Although there are many file types that house this info, the traditional standard has been a shapefile(.shp)

These types of files are commonly used in geographic information systems (GIS) software like ArcGIS

In R, these objects were typically held in a dataframe-like objects that hold the geometries in column (slots).

sf

Historically, the main package for spatial data in R was sp, but we are going to focus on a newer, lighter package called sf (simple features)

Simple features refers to a formal standard on how objects in the real world can be held on computers

An sf data object is a basically a data frame / tibble with extra geometry info tagging along

Although not a tidyverse pacakge, the naming conventions of functions follows similarly. Many of the functions begin with st_* which standard for spatio-temporal

Spatial manipulation with sf: : CHEAT SHEET

The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.

Geometric confirmation

st_contains(x, y, ...) Identifies if x is within y (i.e. point within polygon)

st_covered_by(x, y, ...) Identifies if x is completely within y (i.e. polygon completely within polygon)

st_covers(x, y, ...) Identifies if any point from x is outside of y (i.e. polygon outside polygon)

st_crosses(x, y, ...) Identifies if any geometry of x have commonalities with y

st_disjoint(x, y, ...) Identifies when geometries from x do not share space with y

st_equals(x, y, ...) Identifies if x and y share the same geometry

st_intersects(x, y, ...) Identifies if x and y geometry share any space

st_overlaps(x, y, ...) Identifies if geometries of x and y share space, are of the same dimension, but are not completely contained by each other

st_touches(x, y, ...) Identifies if geometries of x and y share a common point but their interiors do not intersect

st_within(x, y, ...) Identifies if x is in a specified distance to y

Geometric operations

st_boundary(x) Creates a polygon that encompasses the full extent of the geometry

st_buffer(x, dist, nQuadSegs) Creates a polygon covering all points of the geometry within a given distance

st_centroid(x, ..., of_largest_polygon) Creates a point at the geometric centre of the geometry

st_convex_hull(x) Creates geometry that represents the minimum convex geometry of x

st_line_merge(x) Creates linestring geometry from sewing multi linestring geometry together

st_node(x) Creates nodes on overlapping geometry where nodes do not exist

st_point_on_surface(x) Creates a point that is guarenteed to fall on the surface of the geometry

st_polygonize(x) Creates polygon geometry from linestring geometry

st_segmentize(x, dfMaxLength, ...) Creates linesting geometry from x based on a specified length

st_simplify(x, preserveTopology, dTolerance) Creates a simplified version of the geometry based on a specified tolerance

Geometry creation

st_triangulate(x, dTolerance, bOnlyEdges) Creates polygon geometry as triangles from point geometry

st_voronoi(x, envelope, dTolerance, bOnlyEdges) Creates polygon geometry covering the envolope of x, with x at the centre of the geometry

st_point(x, c(numeric vector), dim = "XYZ") Creating point geometry from numeric values

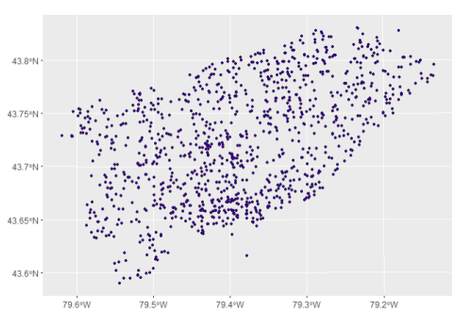
st_multipoint(x = matrix(numeric values in rows), dim = "XYZ") Creating multi point geometry from numeric values

st_linestring(x = matrix(numeric values in rows), dim = "XYZ") Creating linestring geometry from numeric values

st_multilinestring(x = list(numeric matricesin rows), dim = "XYZ") Creating multi linestring geometry from numeric values

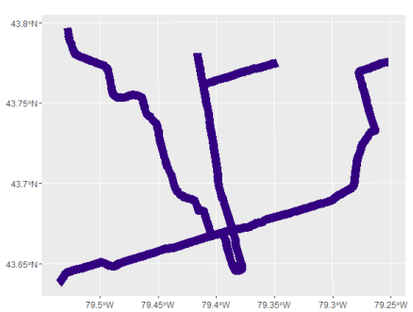
st_polygon(x = list(numeric matrices in rows), dim = "XYZ") Creating polygon geometry from numeric values

st_multipolygon(x = list(numeric matrices in rows), dim = "XYZ") Creating multi polygon geometry from numeric values



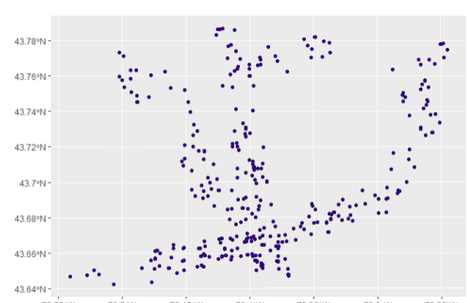

```
ggplot() +
  geom_sf(data = schools)
```

+




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

=>




```
ggplot() +
  geom_sf(data = st_intersection(schools, st_buffer(subway, 1000)))
```

This cheatsheet presents the sf package [Edzer Pebesma 2018] in version 0.6.3. See https://github.com/r-spatial/sf for more details.

sf

```
nc <- st_read(system.file("shape/nc.shp", package="sf"))
```

```
Simple feature collection with 100 features and 14 fields
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
epsg (SRID):    4267
proj4string:    +proj=longlat +datum=NAD27 +no_defs
First 10 features:
    AREA PERIMETER CNTY_ CNTY_ID        NAME  FIPS FIPSNO CRESS_ID BIR74 SID74
1  0.114     1.442  1825    1825        Ashe 37009  37009         5  1091     1
2  0.061     1.231  1827    1827   Alleghany 37005  37005         3   487     0
3  0.143     1.630  1828    1828       Surry 37171  37171        86  3188     5
4  0.070     2.968  1831    1831   Currituck 37053  37053        27   508     1
5  0.153     2.206  1832    1832 Northampton 37131  37131        66  1421     9
```

```
class(nc)
```

```
[1] “sf”   "data.frame"
```

sf

```
nc <- st_read(system.file("shape/nc.shp", package="sf"))
```

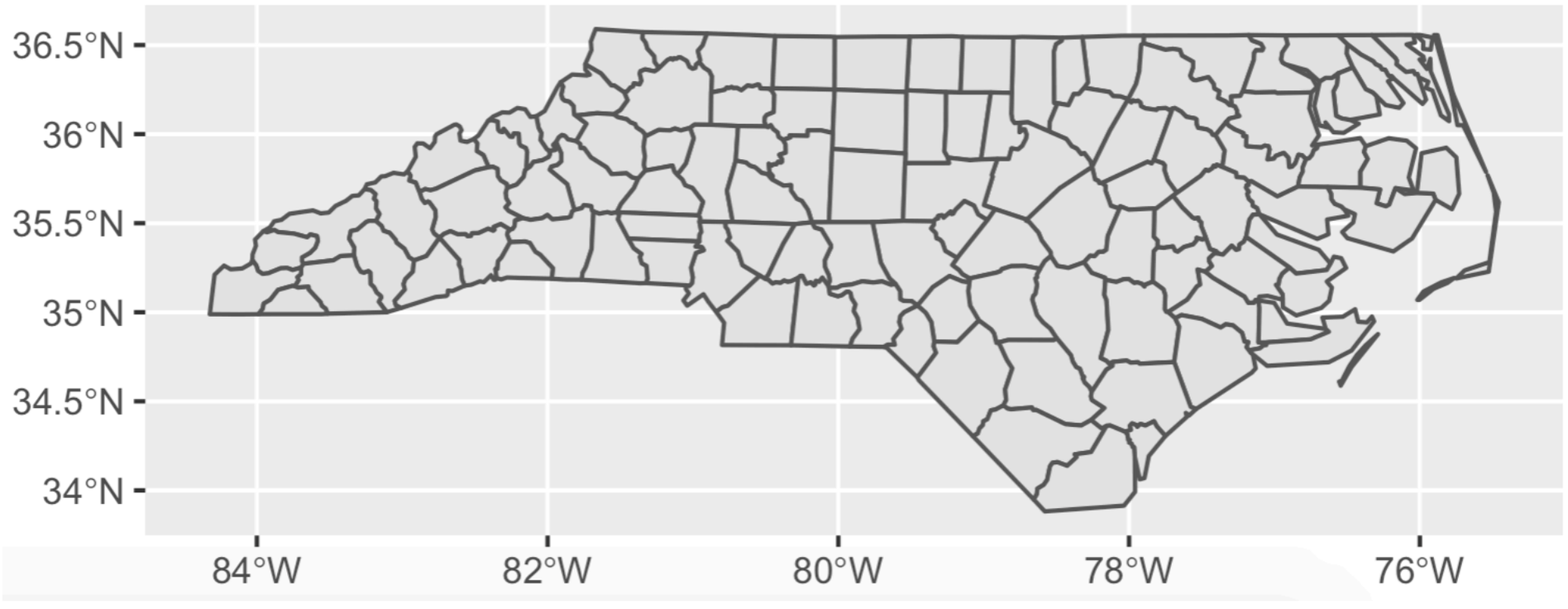
```
nc$geometry
```

```
Geometry set for 100 features
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
epsg (SRID):    4267
proj4string:    +proj=longlat +datum=NAD27 +no_defs
First 5 geometries:
MULTIPOLYGON (((-81.47276 36.23436, -81.54084 3...
MULTIPOLYGON (((-81.23989 36.36536, -81.24069 3...
MULTIPOLYGON (((-80.45634 36.24256, -80.47639 3...
MULTIPOLYGON (((-76.00897 36.3196, -76.01735 36...
MULTIPOLYGON (((-77.21767 36.24098, -77.23461 3...
```

sf

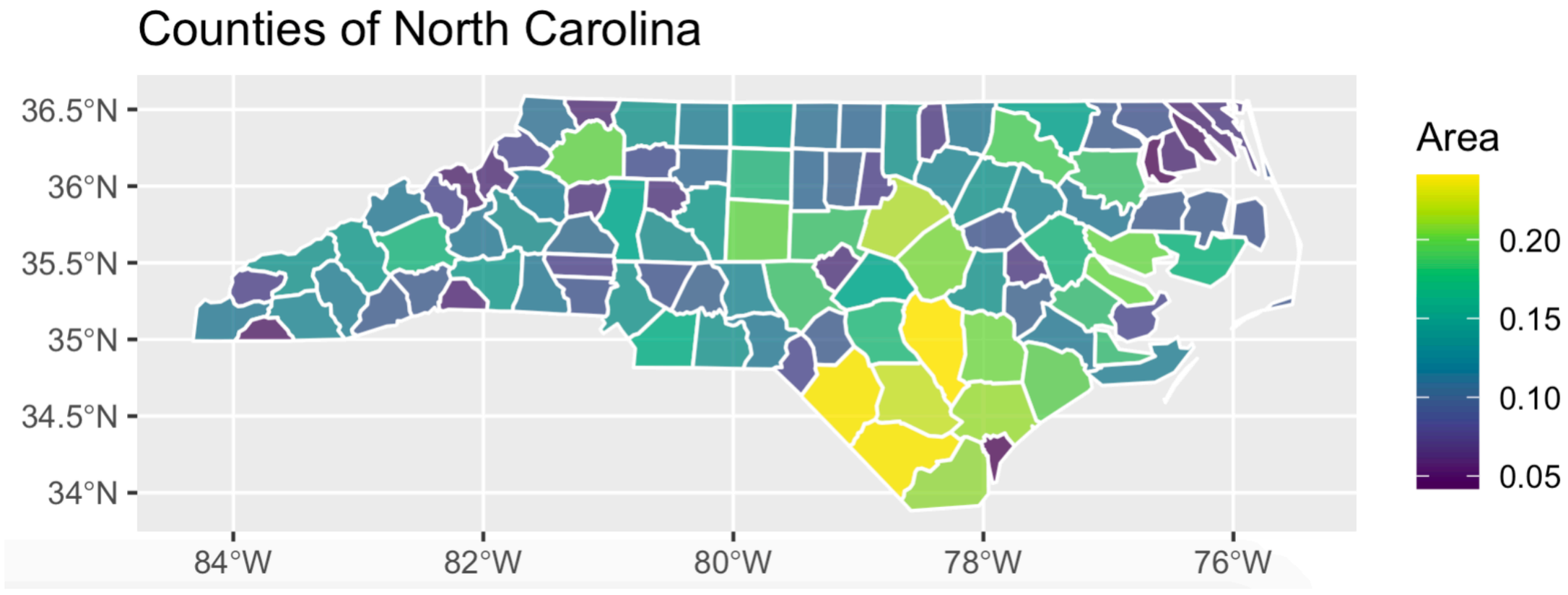
Plotting sf objects is super easy thanks to ggplot2's support of sf objects with `geom_sf()`

```
nc %>% ggplot() + geom_sf()
```



sf

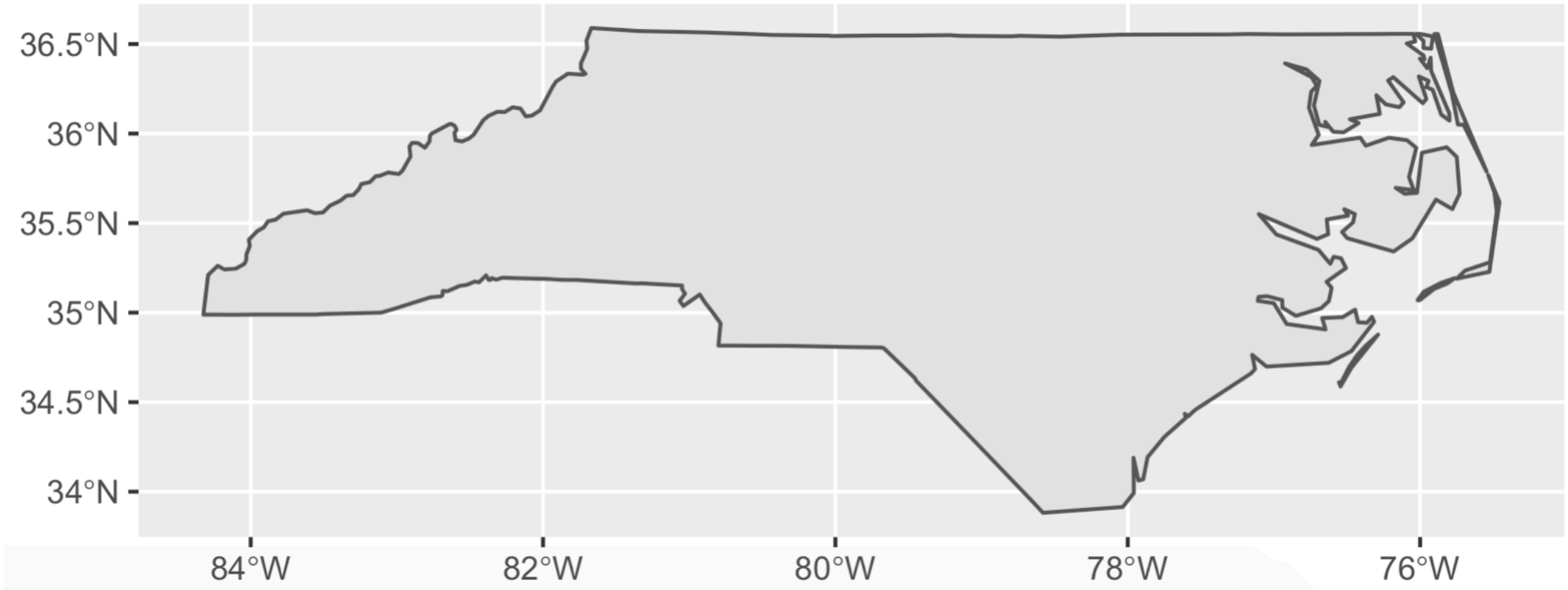
```
ggplot(nc) +
    geom_sf(aes(fill = AREA), alpha=0.8, col="white") +
    scale_fill_viridis_c(name = "Area") +
    ggtitle("Counties of North Carolina")
```



sf

Specialized geometric operations:

```
st_union(nc) %>% ggplot() + geom_sf()
```



ggmap

When you are working with spatial data, it might be helpful to add a map (like Google) as the background

ggmap interacts with Google Maps API and other mapping services like Stamen to pull down map tiles

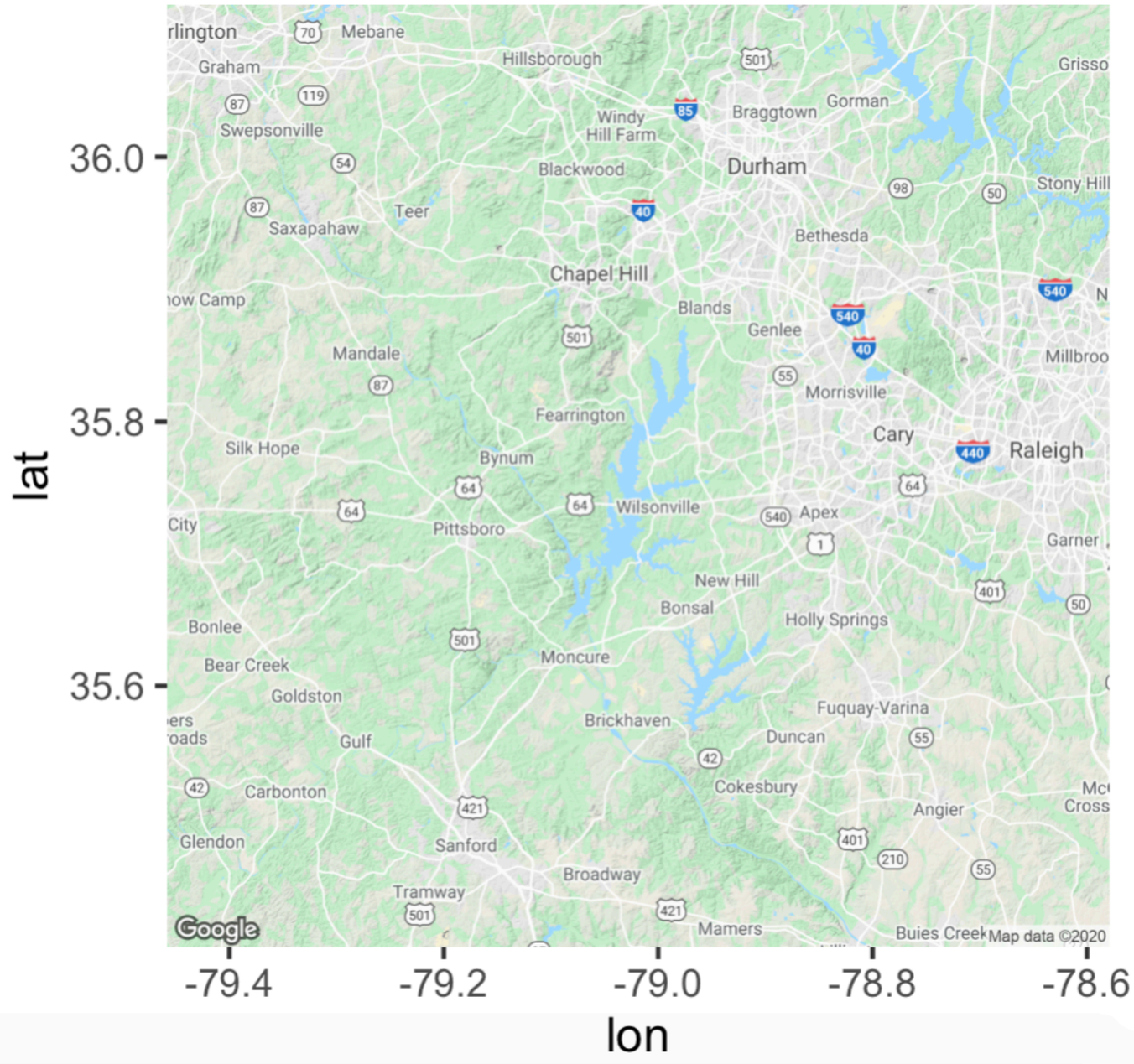
ggmap was made by my graduate school advisor, David Kahle, and so I can't not talk about it

Google Maps API recently changed its settings (requires registering an API key, entering a credit card for possible payments, etc.) which make it more inconvenient to use…

ggmap

```
map <- get_googlemap("north carolina")

ggmap(map)
```



ggmap

Special utility functions:

```
geocode("Shippensburg University")
```

```
# A tibble: 1 x 2
    lon   lat
  <dbl> <dbl>
1 -77.5  40.1
```

```
revgeocode(c(-77.03653, 38.89768))
```

```
[1] "1600 Pennsylvania Ave NW, Washington, DC 20500, USA"
```

```
route_df <- route("shippensburg university", "new york city",
structure = "route")
```

ggmap

```
get_map("allentown pa", zoom = 7) %>%
    ggmap(extent = "device") +
    geom_path(
        aes(lon, lat), color = "blue",
        size = 1.5, alpha = .5,
        data = route_df, lineend = "round"
    )
```