

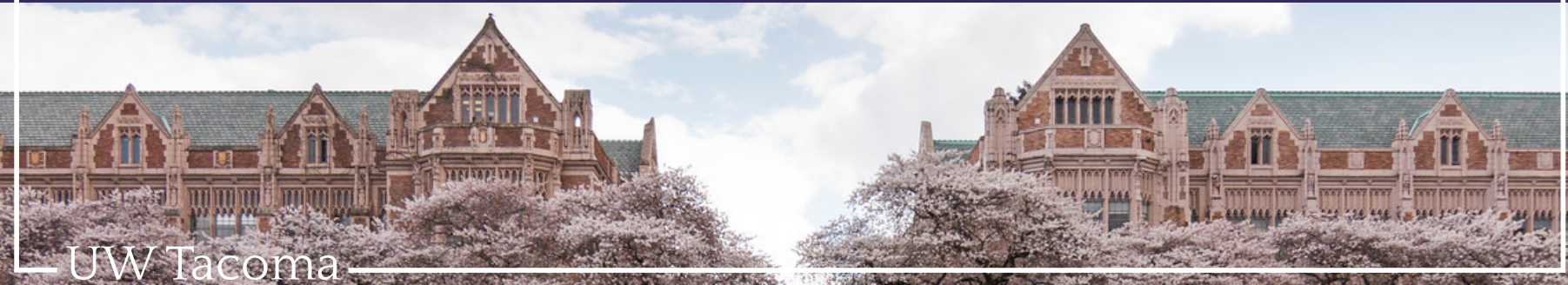
# R Exposure 4

## Part 2: Mapping

Chuck Lanfear

Apr 19, 2020

Updated: Apr 18, 2020



# This Unit

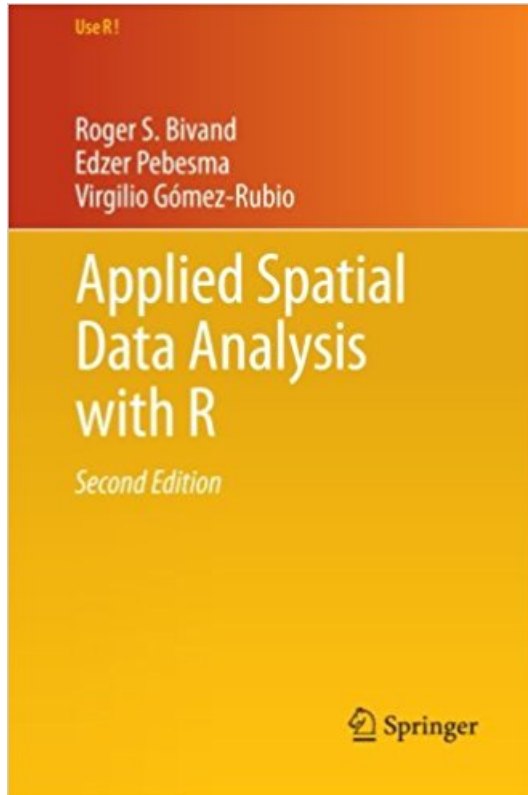
## Basic Mapping in `ggplot2`

- Mapping with raw `ggplot2` using coordinates
- `ggmap` for mashing up maps with `ggplot2`
- Labeling points and using `ggrepel` to avoid overlaps

## Advanced Mapping

- `sf`: Simple Features geometry for R
- `tidycensus` for obtaining Census Bureau data

# Mapping in R: A quick plug



If you are interested in mapping, GIS, and geospatial analysis in R, *acquire this book*.

[RSpatial.org](http://RSpatial.org) is a great resource as well.

# Basic Mapping

`ggplot2` and `ggmap`

# One Day of SPD Incidents

Let's look at types and locations of incidents the Seattle Police Department responded to in a single day.

```
library(tidyverse)
```

```
spd_raw <- read_csv("https://clanfear.github.io/CSS508/Seattle_Police")
```

# Taking a glimpse()

```
glimpse(spd_raw)
```

```
## Rows: 706
## Columns: 19
## $ `CAD CDW ID`      <dbl> 1701856, 1701857, 1701853, ...
## $ `CAD Event Number` <dbl> 1.6e+10, 1.6e+10, 1.6e+10, ...
## $ `General Offense Number` <dbl> 2.02e+09, 2.02e+09, 2.02e+0...
## $ `Event Clearance Code` <chr> "063", "064", "161", "245",...
## $ `Event Clearance Description` <chr> "THEFT - CAR PROWL", "SHOPL...
## $ `Event Clearance SubGroup` <chr> "CAR PROWL", "THEFT", "TRES...
## $ `Event Clearance Group` <chr> "CAR PROWL", "SHOPLIFTING",...
## $ `Event Clearance Date` <chr> "03/25/2016 11:58:30 PM", "...
## $ `Hundred Block Location` <chr> "S KING ST / 8 AV S", "92XX...
## $ `District/Sector` <chr> "K", "S", "D", "M", "M", "B...
## $ `Zone/Beat` <chr> "K3", "S3", "D2", "M1", "M3...
## $ `Census Tract` <dbl> 9100, 11801, 7200, 8002, 81...
## $ Longitude <dbl> -122, -122, -122, -122, -12...
## $ Latitude <dbl> 47.6, 47.5, 47.6, 47.6, 47....
## $ `Incident Location` <chr> "(47.598347, -122.32245)", ...
## $ `Initial Type Description` <chr> "THEFT (DOES NOT INCLUDE SH...
## $ `Initial Type Subgroup` <chr> "OTHER PROPERTY", "SHOPLIFT...
## $ `Initial Type Group` <chr> "THEFT", "THEFT", "TRESPASS...
## $ `At Scene Time` <chr> "03/25/2016 10:25:51 PM", "...
```

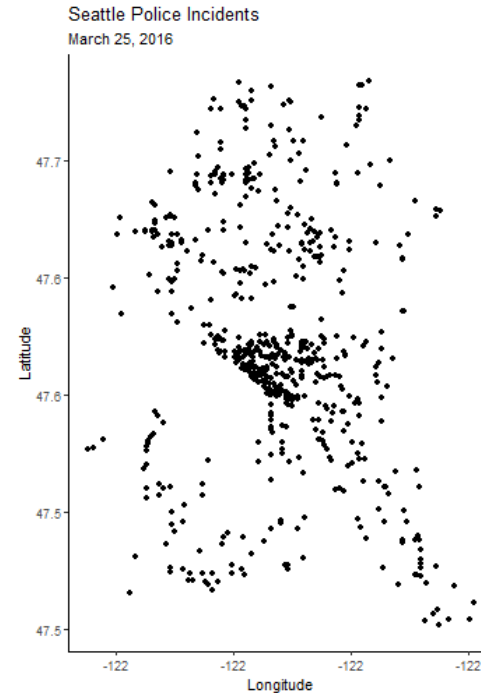
# x, y as Coordinates

Coordinates, such as longitude and latitude, can be provided in `aes()` as `x` and `y` values.

This is ideal when you don't need to place points over some map for reference.

```
ggplot(spd_raw,  
       aes(Longitude, Latitude)) +  
  geom_point() +  
  coord_fixed() +  
  ggtitle("Seattle Police Incidents",  
          subtitle="March 25, 2016") +  
  theme_classic()
```

Sometimes, however, we want to plot these points over existing maps.



ggmap



# ggmap

`ggmap` is a package that works with `ggplot2` to plot spatial data directly on map images downloaded from Google Maps<sup>1</sup>, OpenStreetMap, and Stamen Maps (good artistic/minimal options).

What this package does for you:

1. Queries servers for a map (`get_map()`) at the location and scale you want
2. Plots the **raster** (bitmap) image as a `ggplot` object
3. Lets you add more `ggplot` layers like points, 2D density plots, text annotations
4. Additional functions for interacting with Google Maps (e.g. getting distances by bike)

[1] [Requires an API Key now.](#)

# Installation

As of Apr 18, 2020, the current version of `ggmap` must be downloaded from GitHub.

This can be done using the `devtools` package.

```
if(!requireNamespace("devtools")) install.packages("devtools")  
devtools::install_github("dkahle/ggmap", ref = "tidyup")
```

```
library(ggmap)
```

This may require compilation on your computer. If you get errors during install, check with me after class.

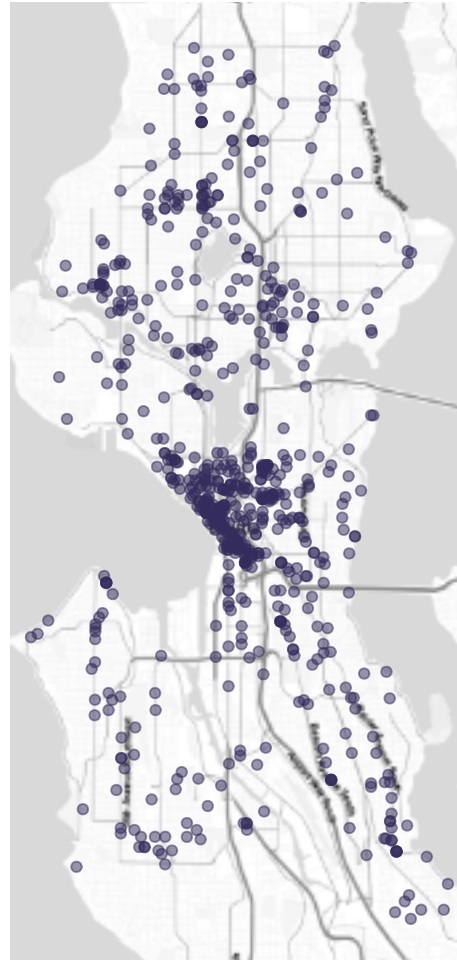
# Quick Maps with `qmpLOT()`

`qmpLOT` will automatically set the map region based on your data:

```
qmpLOT(data = spd_raw,  
        x = Longitude,  
        y = Latitude,  
        color = I("#342c5c"),  
        alpha = I(0.5))
```

All I provided was numeric latitude and longitude, and it placed the data points correctly on a raster map of Seattle.

`I()` is used here to specify *set* (constant) rather than *mapped* values.



# get\_map()

Both `qplot()` and `qmap()` are wrappers for a function called `get_map()` that retrieves a base map layer. Some options:

- `location=` search query or numeric vector of longitude and latitude
- `zoom=` a zoom level (3 = continent, 10 = city, 21 = building)
- `source=`
  - `"google"`: Google Maps for general purpose maps<sup>1</sup>
  - `"osm"`: OpenStreetMaps, general purpose but open access
  - `"stamen"`: Aesthetically pleasing alternatives based on OpenStreetMaps
- `maptype=`
  - Google types: `"terrain"`, `"terrain-background"`, `"satellite"`, `"roadmap"`, `"hybrid"`
  - Stamen types: `"watercolor"`, `"toner"`, `"toner-background"`, `"toner-lite"`
- `color=` `"color"` or `"bw"`

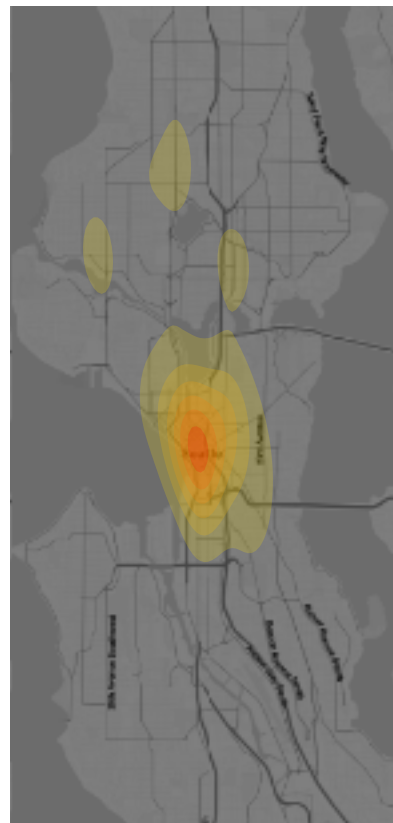
[1] Requires API key!

# Adding Density Layers


Call `qplot()` with no `geom()`, and then add density layers:

```
qplot(data = spd_raw, geom = "blank",  
      x = Longitude, y = Latitude,  
      maptype = "toner-lite",  
      darken = 0.5) +  
  stat_density_2d(  
    aes(fill = stat(level)),  
    geom = "polygon",  
    alpha = .2, color = NA) +  
  scale_fill_gradient2(  
    "Incident\nConcentration",  
    low = "white",  
    mid = "yellow",  
    high = "red") +  
  theme(legend.position = "bottom")
```

`stat(level)` indicates we want  
`fill=` to be based on `level` values  
calculated by the layer.



Incident  
Concentration



50 100 150 200 250 300

# Labeling Points

Let's label the assaults and robberies specifically in downtown:

First filter to downtown based on values "eyeballed" from our earlier map:

```
downtown <- spd_raw %>%  
  filter(Latitude > 47.58, Latitude < 47.64,  
         Longitude > -122.36, Longitude < -122.31)
```

Then make a dataframe of just assaults and robberies:

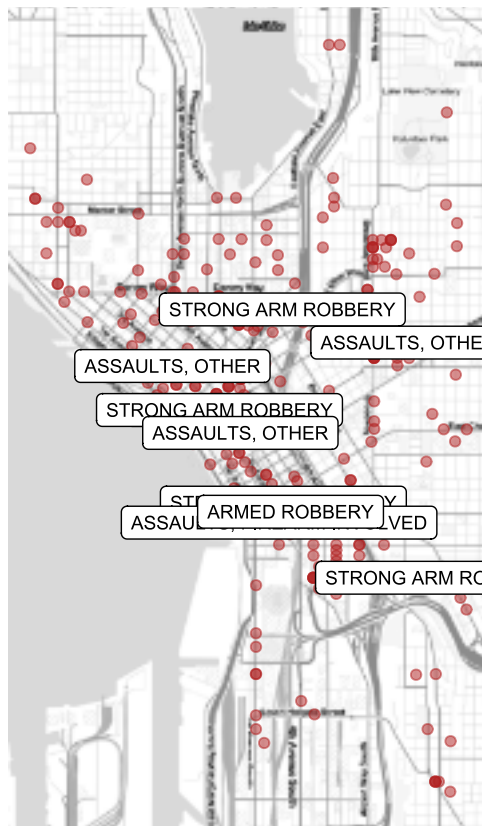
```
assaults <- downtown %>%  
  mutate(assault_label =  
    ifelse(`Event Clearance Group` %in%  
           c("ASSAULTS", "ROBBERY"),  
           `Event Clearance Description`, "") %>%  
  filter(assault_label != "")
```

# Plotting with Labels

Now let's plot the events and label them with `geom_label()` (`geom_text()` without background or border):

```
qplot(data = downtown,
      x = Longitude,
      y = Latitude,
      matype = "toner-lite",
      color = I("firebrick"),
      alpha = I(0.5)) +
  geom_label(data = assaults,
            aes(label = assault_label),
            size=2.5)
```

Placing the arguments for `color=` and `alpha=` inside `I()` prevents them from also applying to the labels. We would get transparent red labels otherwise!

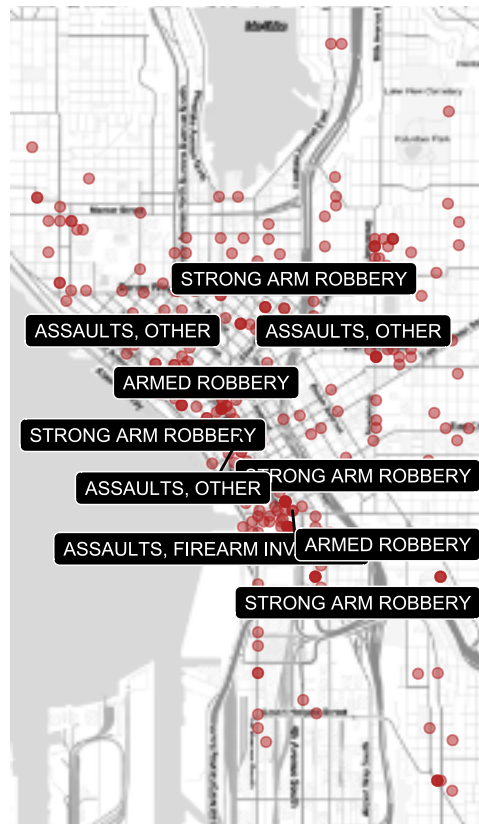


# ggrepel

You can also try

`geom_label_repel()` or  
`geom_text_repel()` in the  
`ggrepel` package to fix or reduce  
overlaps (total space is limited here):

```
library(ggrepel)
qplot(data =
  downtown,
  x = Longitude,
  y = Latitude,
  maptype = "toner-lite",
  color = I("firebrick"),
  alpha = I(0.5)) +
  geom_label_repel(
    data = assaults,
    aes(label = assault_label),
    fill = "black",
    color = "white",
    segment.color = "black",
    size=2.5)
```





# Advanced Mapping

GIS and R with `sf`

# Terminology

- Simple Features (sf)
- Coordinate Reference System (CRS)
- Shapefile

# sf

Until recently, the main way to work with geospatial data in R was through the `sp` package. `sp` works well but does not store data the same way as most GIS packages and can be bulky and complicated.

The more recent `sf` package implements the GIS standard of Simple Features in R.

`sf` is also integrated into the `tidyverse`: e.g. `geom_sf()` in `ggplot2`.

The package is somewhat new but is expected to *replace* `sp` eventually. The principle authors and contributors to `sf` are the same authors as `sp` but with new developers from the `tidyverse` as well.

Because `sf` is the new standard, we will focus on it today.

```
library(sf)
```

# Simple Features

A **Simple Feature** is a single observation with some defined geospatial location(s). Features are stored in special data frames (class `sf`) with two properties:

- **Geometry:** Properties describing a location (usually on Earth).
  - Usually 2 dimensions, but support for up to 4.
  - Stored in a single reserved *list-column* (`geom`, of class `sfc`).<sup>1</sup>
  - Contain a defined coordinate reference system.
- **Attributes:** Characteristics of the location (such as population).
  - These are non-spatial measures that describe a feature.
  - Standard data frame columns.

[1] A list-column is the same length as all other columns in the data, but each element contains *sub-elements* (class `sfg`) with all the geometrical components.

*List-columns* require special functions to manipulate, *including removing them*.

# Coordinate Reference Systems

**Coordinate reference systems (CRS)** specify what location on Earth geometry coordinates are *relative to* (i.e. what location is (0,0) when plotting).

The most commonly used is **WGS84**, the standard for Google Earth, the Department of Defense, and GPS satellites.

There are two common ways to define a CRS:

- **EPSG codes** (`epsg` in R)
  - Numeric codes which *refer to a predefined projection*
  - Ex: WGS84 is `4326`
- **PROJ.4 strings** (`proj4string` in R)
  - Text strings which *define a projection*
  - WGS84 looks like this:

```
+init=epsg:4326 +proj=longlat +ellps=WGS84  
+datum=WGS84 +no_defs +towgs84=0,0,0
```

# Shapefiles

Geospatial data is typically stored in **shapefiles** which store geometric data as **vectors** with associated attributes (variables)

Shapefiles actually consist of multiple individual files. There are usually at least three (but up to 10+):

- `.shp`: The feature geometries
- `.shx`: Shape positional index
- `.dbf`: Attributes describing features<sup>1</sup>

Often there will also be a `.prj` file defining the coordinate system.

[1] This is just a dBase IV file which is an ancient and common database storage file format.

Using 

# Selected `sf` Functions

`sf` is a huge, feature-rich package. Here is a sample of useful functions:

- `st_read()`, `st_write()`: Read and write shapefiles.
- `geom_sf()`: `ggplot()` layer for `sf` objects.
- `st_join()`: Join data by spatial relationship.
- `st_transform()`: Convert between CRS.
- `st_set_geometry(., NULL)`: Remove geometry from a `sf` data frame.
- `st_relate()`: Compute relationships between geometries (like neighbor matrices).
- `st_interpolate_aw()`: Areal-weighted interpolation of polygons.



# Loading Data

We will work with King County voting data. You can obtain a shape file of King County voting precincts from the [county GIS data portal](#).

We can load the file using `st_read()`.

```
precinct_shape <- st_read("./data/district/votdst.shp",  
                           stringsAsFactors = F) %>%  
  select(Precinct=NAME, geometry)
```

If following along, click here to download a zip of the shapefile and here for the voting data

We'll pause if folks are following along.

# Voting Data: Processing

```
precincts_votes_sf <-  
  read_csv("../data/king_county_elections_2016.txt") %>%  
  filter(Race=="US President & Vice President",  
         str_detect(Precinct, "SEA ")) %>%  
  select(Precinct, CounterType, SumOfCount) %>%  
  group_by(Precinct) %>%  
  filter(CounterType %in%  
         c("Donald J. Trump & Michael R. Pence",  
           "Hillary Clinton & Tim Kaine",  
           "Registered Voters",  
           "Times Counted")) %>%  
  mutate(CounterType =  
         recode(CounterType,  
                `Donald J. Trump & Michael R. Pence` = "Trump",  
                `Hillary Clinton & Tim Kaine` = "Clinton",  
                `Registered Voters`="RegisteredVoters",  
                `Times Counted` = "TotalVotes")) %>%  
  spread(CounterType, SumOfCount) %>%  
  mutate(P_Dem = Clinton / TotalVotes,  
         P_Rep = Trump / TotalVotes,  
         Turnout = TotalVotes / RegisteredVoters) %>%  
  select(Precinct, P_Dem, P_Rep, Turnout) %>%  
  filter(!is.na(P_Dem)) %>%  
  left_join(precinct_shape) %>%  
  st_as_sf() # Makes sure resulting object is an sf dataframe
```

# Taking a `glimpse()`

```
glimpse(precincts_votes_sf)
```

```
## Rows: 960
## Columns: 5
## Groups: Precinct [960]
## $ Precinct <chr> "SEA 11-1256", "SEA 11-1550", "SEA 11-1552", "SE...
## $ P_Dem      <dbl> 0.771, 0.817, 0.751, 0.838, 0.833, 0.815, 0.828,...
## $ P_Rep      <dbl> 0.1561, 0.0779, 0.1342, 0.0865, 0.0859, 0.0803, ...
## $ Turnout    <dbl> 0.693, 0.727, 0.735, 0.752, 0.758, 0.771, 0.687,...
## $ geometry   <MULTIPOLYGON [US_survey_foot]> MULTIPOLYGON (((127369...
```

Notice the `geometry` column and its unusual class: `MULTIPOLYGON`

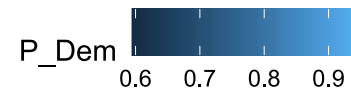
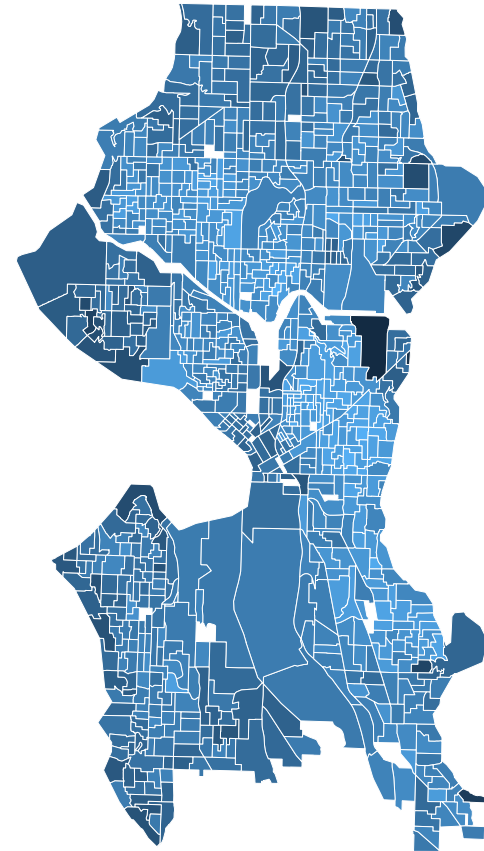
A single observation (row) has a geometry which may consist of multiple polygons.

# Voting Map

We can plot `sf` geometry using `geom_sf()`.

```
ggplot(precincts_votes_sf,  
  aes(fill=P_Dem)) +  
  geom_sf(color="white",  
    size=0.1) +  
  theme_void() +  
  theme(legend.position =  
    "bottom")
```

- `fill=P_Dem` maps color inside precincts to `P_Dem`.
- `color="white"` sets boundaries to white.
- `theme_void()` removes axes and background.



# tidycensus

# tidycensus

`tidycensus` can be used to search the American Community Survey (ACS) and Decennial Census for variables, then download them and automatically format them as tidy dataframes.

**These dataframes include geographical boundaries such as tracts!**

This package utilizes the Census API, so you will need to obtain a [Census API key](#).

**Application Program Interface (API):** A type of computer interface that exists as the "native" method of communication between computers, often via http (usable via `httr` package).

- R packages that interface with websites and databases typically use APIs.
- APIs make accessing data easy while allowing websites to control access.

See [the developer's GitHub page](#) for detailed instructions.

# Key `tidycensus` Functions

- `census_api_key()` - Install a census api key.
  - Note you will need to run this prior to using any `tidycensus` functions.
- `load_variables()` - Load searchable variable lists.
  - `year =`: Sets census year or endyear of 5-year ACS
  - `dataset =`: Sets dataset (see `?load_variables`)
- `get_decennial()` - Load Census variables and geographical boundaries.
  - `variables =`: Provide vector of variable IDs
  - `geography =`: Sets unit of analysis (e.g. `state`, `tract`, `block`)
  - `year =`: Census year (1990, 2000, or 2010)
  - `geometry = TRUE`: Returns `sf` geometry
- `get_acs()` - Load ACS variables and boundaries.

# Searching for Variables

```
library(tidycensus)
# census_api_key("PUT YOUR KEY HERE", install=TRUE)
acs_2015_vars <- load_variables(2015, "acs5")
acs_2015_vars[10:18,] %>% print()
```

```
## # A tibble: 9 x 3
##   name          label          concept
##   <chr>        <chr>        <chr>
## 1 B01001_008 Estimate!!Total!!Male!!20 years SEX BY AGE
## 2 B01001_009 Estimate!!Total!!Male!!21 years SEX BY AGE
## 3 B01001_010 Estimate!!Total!!Male!!22 to 24 years SEX BY AGE
## 4 B01001_011 Estimate!!Total!!Male!!25 to 29 years SEX BY AGE
## 5 B01001_012 Estimate!!Total!!Male!!30 to 34 years SEX BY AGE
## 6 B01001_013 Estimate!!Total!!Male!!35 to 39 years SEX BY AGE
## 7 B01001_014 Estimate!!Total!!Male!!40 to 44 years SEX BY AGE
## 8 B01001_015 Estimate!!Total!!Male!!45 to 49 years SEX BY AGE
## 9 B01001_016 Estimate!!Total!!Male!!50 to 54 years SEX BY AGE
```



# Getting Data

```
king_county <- get_acs(geography="tract", state="WA",  
                        county="King", geometry = TRUE,  
                        variables=c("B02001_001E",  
                                   "B02009_001E"),  
                        output="wide")
```

What do these look like?

```
glimpse(king_county)
```

```
## Rows: 398  
## Columns: 7  
## $ GEOID      <chr> "53033032309", "53033031704", "53033032313", ...  
## $ NAME       <chr> "Census Tract 323.09, King County, Washington...  
## $ B02001_001E <dbl> 8249, 5728, 6551, 5941, 6365, 5292, 6132, 663...  
## $ B02001_001M <dbl> 457, 299, 365, 322, 275, 417, 602, 580, 311, ...  
## $ B02009_001E <dbl> 375, 633, 294, 128, 26, 80, 53, 2117, 456, 62...  
## $ B02009_001M <dbl> 246, 356, 260, 127, 37, 108, 65, 602, 287, 40...  
## $ geometry   <MULTIPOLYGON [°]> MULTIPOLYGON (((-122 47.7, ..., ...
```

With `output="wide"`, **estimates** end in **E** and *error margins* in **M**.

# Processing Data

We can drop the margins of error, rename the estimates then, `mutate()` into a proportion `Any Black` measure.

```
king_county <- king_county %>%  
  select(-ends_with("M")) %>%  
  rename(`Total Population`=B02001_001E,  
         `Any Black`=B02009_001E) %>%  
  mutate(`Any Black` = `Any Black` / `Total Population`)  
glimpse(king_county)
```

```
## Rows: 398  
## Columns: 5  
## $ GEOID      <chr> "53033032309", "53033031704", "5303303...  
## $ NAME       <chr> "Census Tract 323.09, King County, Was...  
## $ `Total Population` <dbl> 8249, 5728, 6551, 5941, 6365, 5292, 61...  
## $ `Any Black`  <dbl> 0.04546, 0.11051, 0.04488, 0.02155, 0....  
## $ geometry    <MULTIPOLYGON [°]> MULTIPOLYGON (((-122 47.7...
```

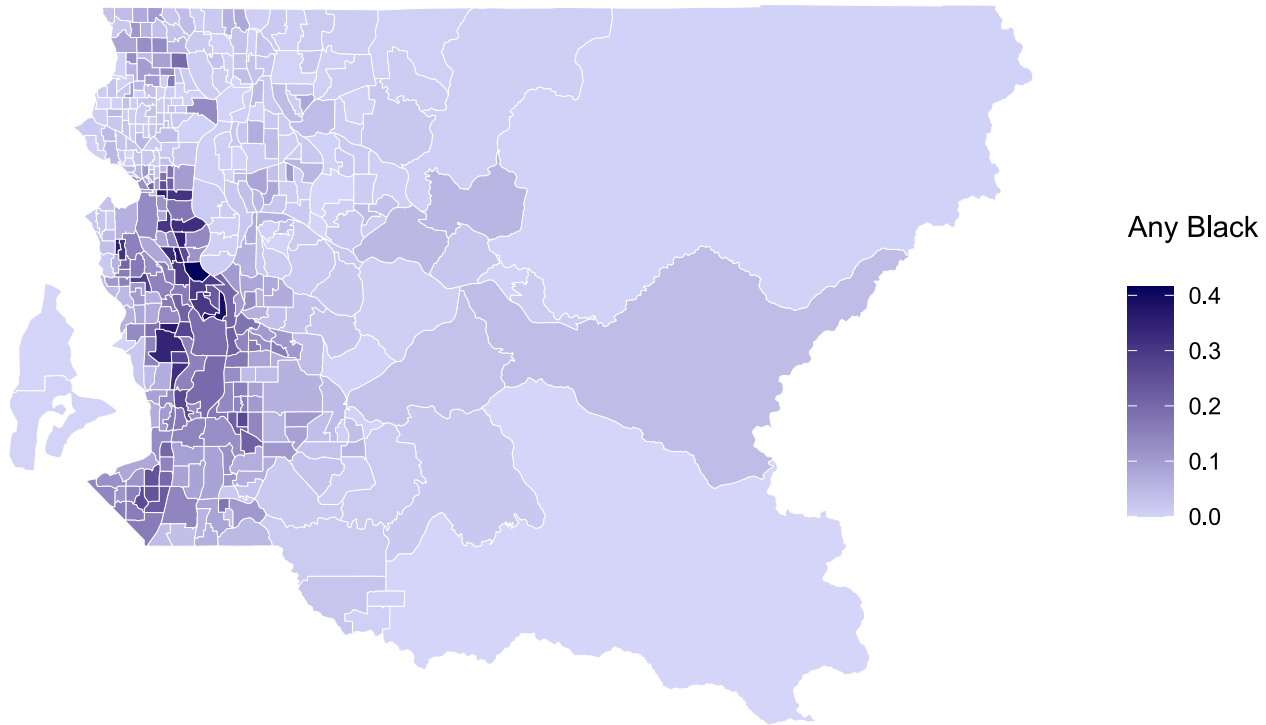
# Mapping Code

```
king_county %>%
  ggplot(aes(fill=`Any Black`)) +
  geom_sf(size=0.1, color="white") +
  coord_sf(crs = "+proj=longlat +datum=WGS84", datum=NA) +
  scale_fill_continuous(name="Any Black\n",
                        low="#d4d5f9",
                        high="#00025b") +
  theme_minimal() + ggtitle("Proportion Any Black")
```

New functions:

- `geom_sf()` draws Simple Features coordinate data.
- `coord_sf()` is used here with these arguments:
  - `crs`: Modifies the coordinate reference system (CRS); WGS84 is possibly the most commonly used CRS.
  - `datum=NA`: Removes graticule lines, which are geographical lines such as meridians and parallels.

## Proportion Any Black



# Removing Water

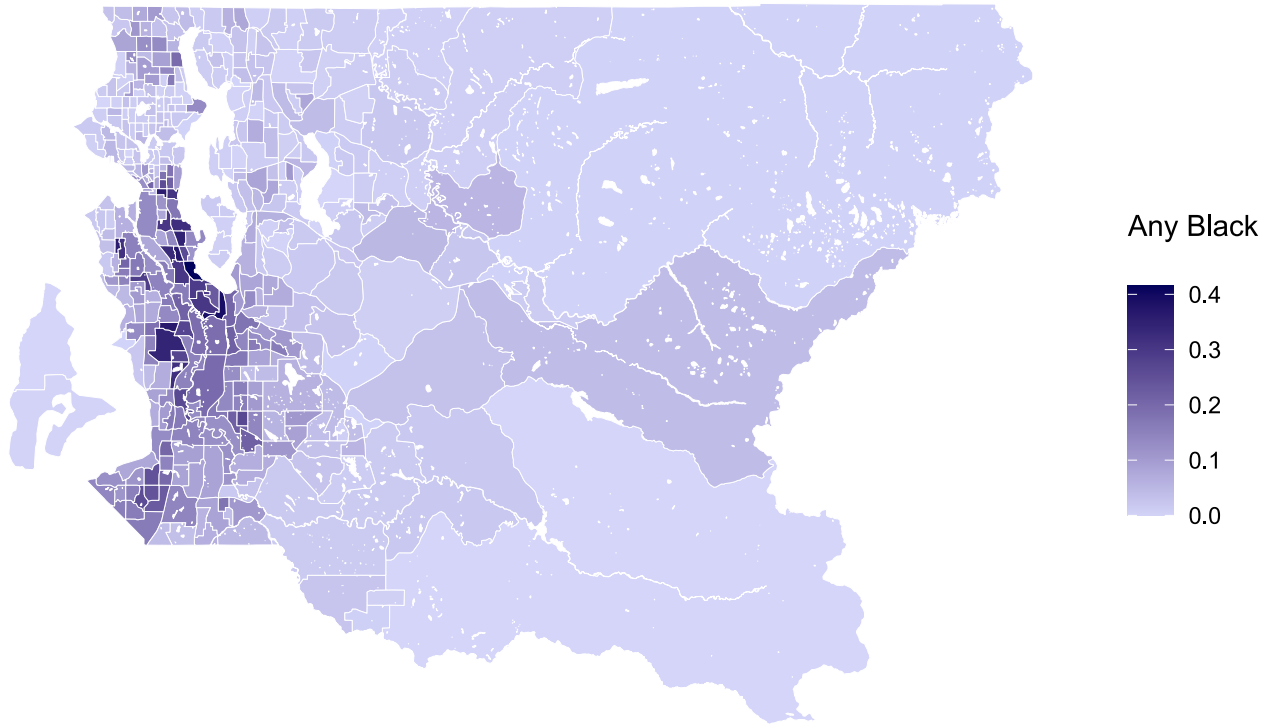
With a simple function and boundaries of water bodies in King County, we can replace water with empty space.

```
st_erase <- function(x, y) {  
  st_difference(x, st_make_valid(st_union(st_combine(y))))  
}  
kc_water <- tigris::area_water("WA", "King", class = "sf")  
kc_nowater <- king_county %>%  
  st_erase(kc_water)
```

- `st_combine()` merges all geometries into one
- `st_union()` resolves internal boundaries
- `st_difference()` subtracts `y` geometry from `x`
- `st_make_valid()` fixes geometry errors from subtraction
- `area_water()` obtains `sf` geometry of water bodies

Then we can reproduce the same plot using `kc_nowater`...

## Proportion Any Black



# State Example Data

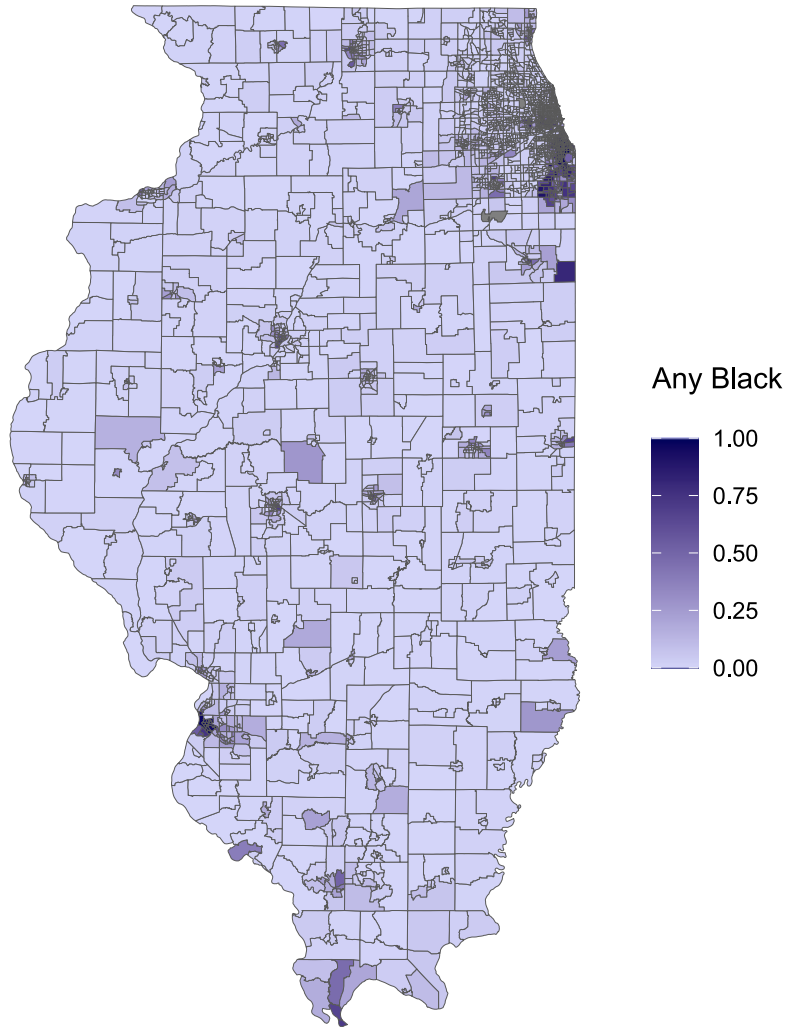
Let's do this again, but for the entire state of Illinois.

```
pb_state <-  
  get_acs(geography = "tract", state = "IL",  
          geometry = TRUE,  
          variables = c("B02001_001E",  
                        "B02009_001E"),  
          output = "wide") %>%  
  select(-ends_with("M")) %>%  
  rename(`Total Population`=B02001_001E,  
         `Any Black`=B02009_001E) %>%  
  mutate(`Any Black` = `Any Black` / `Total Population`)
```

# State Example Plot

```
pb_state %>%  
  ggplot(aes(fill=`Any Black`)) +  
  geom_sf(lwd=0) +  
  coord_sf(crs = "+proj=longlat +datum=WGS84", datum=NA) +  
  scale_fill_continuous(name="Any Black\n",  
                        low="#d4d5f9",  
                        high="#00025b") +  
  theme_minimal()
```





# End of R4, Unit 2