# Thinking about map data the way R does

Many GIS programs (Tableau, Qlik, etc.) make it extraordinarily easy for users to create maps by loading a file with a geographic identifier and data to plot. They are designed to simply drag and drop a field calle "zipcode" or "county" onto a map and automatically draw the appropriate shapes. Then, you can drag anc drop a field called "population density" or "GDP per capita" onto the map, and the shapes automatically color appropriately. These "drag and drop" GIS programs do a lot of work behind the scenes to translate your geographic idenifier into a geometric shape and your fields into colorful metrics.

In R, we have to do this work ourselves. R has no innate knowledge of what we want to graph; we have t provide every detail. This means we need to pass R the information it needs in order to, say, draw the shape of Pennsylvania or draw a line representing I-95. R needs to be told the 4 coordinates defining a rectangle; R needs to be told the hundreds of points defining a rectangle-ish shape like Pennsylvania. If we want to fill Pennsylvania with a color, we need to explicitly tell R how to do so.

The manual nature of GIS in R can cause some headaches, as we need to hunt down all of the information in order to provide it to R to graph. Once you have the desired information, however, you will find that the manual nature of R's graphing allows significantly more flexibility than "drag and drop" programs allow. We are not constrained by the type of information pre-loaded into the program, by the number of shapes we can draw at once, by the color palletes provided, or by any other factor. We have complete flexibility.

If you want to draw state borders (polygons), county borders (more polygons), major highways (lines), an highway rest stops (points), add each of them as an individual layer to the same plot, and color them as you please. There are no constraints when visualizing geospatial data in R.

This post will focus on how to find, import, and clean geospatial data. The actual graphing will come in Part 2 (static maps with `ggplot2`) and Part 3 (interactive maps with `leaflet`).

# A brief introduction to simple features data in R

Out in the wild, map data most frequntly comes as either geoJSON files (.geojson) or Shapefiles (.shp). These files will, at the very minimum, contain information about the geometry of each object to be drawn, such as instructions to draw a point in a certain location or to draw a polygon with certain dimensions. Th raw file may, however, also contain any amount of additional information, such as a name for the object ("Pennsylvania"), or summary statistics (GDP per capita, total population, etc.). Regardless of whether th data is geoJSON or a Shapefile, and regardless of how much additional data the file has, you can use on convenient function from the `sf` package to import the raw data into R as a simple features object. Simpl use either `sf::read_sf(my_json_file)` or `sf::read_sf(my_shp_file)`.

```
library(sf)

# Data from OpenDataPhilly
# Source: https://www.opendataphilly.org/dataset/zip-codes

zip_geojson <- "http://data.phl.opendata.arcgis.com/datasets/b54ec5210cee41c3a884c9086f7af1
be_0.geojson"
phl_zip_raw <- sf::read_sf(zip_geojson)

# If you want to save / load a local copy
# sf::write_sf(phl_zip_raw, "phl_zip_raw.shp")
```

```
# phl_zip_raw <- sf::read_sf("phl_zip_raw.shp")
```

Let's take a look at the simple features data we imported above.

```
head(phl_zip_raw)
#> Simple feature collection with 6 features and 5 fields
#> geometry type:  POLYGON
#> dimension:      XY
#> bbox:           xmin: -75.20435 ymin: 39.95577 xmax: -75.06099 ymax:
40.05317
#> geographic CRS: WGS 84
#> # A tibble: 6 x 6
#>   OBJECTID CODE    COD Shape__Area Shape__Length
geometry
#>
#> 1        1 19120    20   91779697.        49922. ((-75.11107 40.04682,
-75.1094~
#> 2        2 19121    21   69598787.        39535. ((-75.19227 39.99463,
-75.1920~
#> 3        3 19122    22   35916319.        24125. ((-75.15406 39.98601,
-75.1532~
#> 4        4 19123    23   35851751.        26422. ((-75.1519 39.97056,
-75.1515 ~
#> 5        5 19124    24  144808025.        63659. ((-75.0966 40.04249,
-75.09281~
#> 6        6 19125    25   48226254.        30114. ((-75.10849 39.9703,
-75.11051~
```

1. **We have 6 features.** Each row is a feature that we could plot; since we called `head()` we have only see the first 6 even though the full dataset has more
2. **We have 5 fields.** Each column is a field with (potentially) useful information about the feature. Note that the geometry column is not considered a field
3. We are told this is a collection of **polygons,** as opposed to points, lines, etc.
4. We are told the **bounding box** for our data (the most western/eastern longitudes and northern/southern latitudes)
5. We are told the **Coordinate Reference System (CRS),** which in this case is "WGS 84." CRSs are cartographers' ways of telling each other what system they used for describing points on the earth. Cartographers need to pick an equation for an ellipsoid to approximate earth's shape since it's slightly pear-shaped. Cartographers also need to determine a set of reference markers--known as datum--to use to set coordinates, as earth's tectonic plates shift ever so slightly over time. Togehether, the ellipsoid and datum become a CRS.

   WGS 84 is one of the most common CRSs and is the standard used for GPS applications. In the US, you may see data provided using NAD 83. WGS 84 and NAD 83 were originally identical (bacl in the 1980s), but both have been modified over time as the earth changes and scientific knowledg progresses. WGS 84 seeks to keep the global average of points as similar as possible while NAD 83 tries to keep the North American plate as constant as possible. The net result is that the two different CRSs may vary by about a meter in different places. This is not a big difference for most purposes, but sometimes you may need to adjust.

If we wanted to transform our data between CRSs, we would call `sf::st_transform(map_raw, crs = 4326)`, where 4362 is the EPSG code of the CRS into which we would like to transform our geometry. EPSGs are a standard, shorthand way to refer to various CRSs. 4326 is the EPSG code for WGS 84 and 4269 is the EPSG code for NAD 83.

6. Finally, we are provided a column called **"geometry."** This column contains everything that R will need to draw each of the ZIP Codes in Philadelphia, with one row per ZIP Code

# Finding data

Simple features data in R will always look similar to the example above. You will have some metadata describing the type of geometry, the CRS, and so on; a "geometry" column; and optionally some fields of additional data. The trouble comes in trying to find the data you need--both the geometry and the proper additional fields--and getting them together into the same object in R.

### Finding geospatial data

One of the most common sources of geospatial files in R is the `tigris` package. This package allows users to directly download and use TIGER/Line shapefiles--the shapefiles describing the U.S. Census Buerau's census areas. The package includes, among other files, data for national boundaries, state boundaries, county boundaries, ZIP Code Tabulation Areas (very similar to ZIP Codes), census tracts, congressional districts, metro areas, roads, and many other useful US geographic features.

`tigris` allows you to import directly as a simple features object. Let's take a quick look at how to import county data.

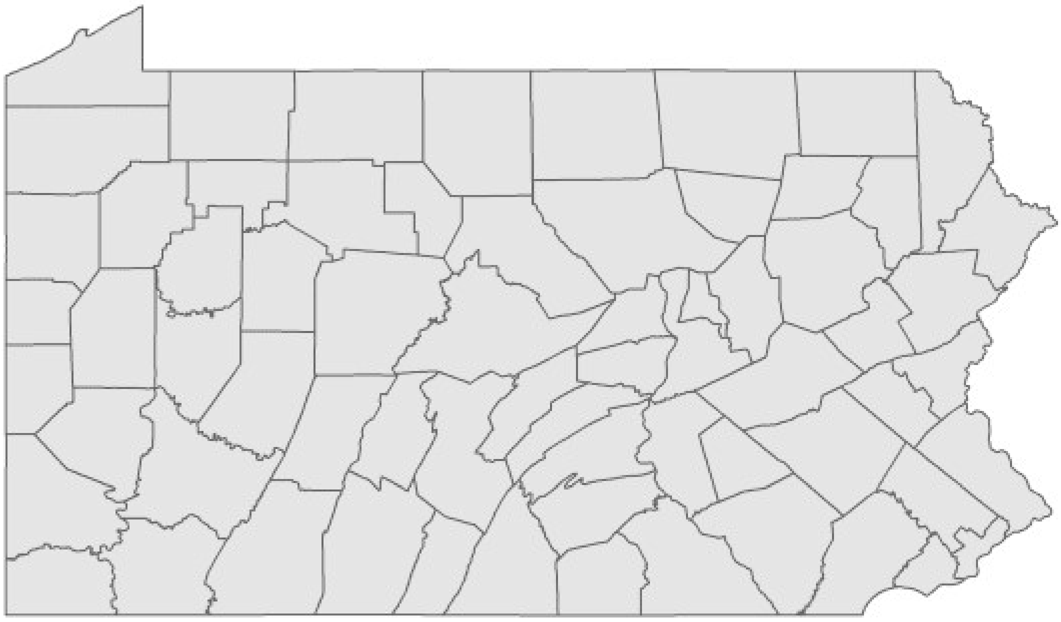
```
library(tigris)
library(ggplot2)

pa_counties_raw <- tigris::counties(
  state = "PA",
  cb = TRUE,
  resolution = "500k",
  year = 2018,
  class = "sf"
)

head(pa_counties_raw)
#> Simple feature collection with 6 features and 9 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -80.51942 ymin: 39.72089 xmax: -75.01507 ymax:
41.47858
#> geographic CRS: NAD83
#>     STATEFP COUNTYFP COUNTYNS       AFFGEOID GEOID       NAME LSAD
ALAND
#> 239      42      005 01213658 0500000US42005 42005  Armstrong   06
1691724751
#> 240      42      029 01209174 0500000US42029 42029    Chester   06
1943848979
#> 241      42      035 01214721 0500000US42035 42035    Clinton   06
2299868396
#> 242      42      059 01214033 0500000US42059 42059     Greene   06
```

```
1491700989
#> 243      42      067 01209180 0500000US42067 42067    Juniata   06
1013592882
#> 244      42      091 01213680 0500000US42091 42091 Montgomery   06
1250855248
#>       AWATER                       geometry
#> 239 27619089 MULTIPOLYGON (((-79.69293 4...
#> 240 22559478 MULTIPOLYGON (((-75.59129 3...
#> 241 23178635 MULTIPOLYGON (((-78.09338 4...
#> 242  5253865 MULTIPOLYGON (((-80.51942 3...
#> 243  5606077 MULTIPOLYGON (((-77.74677 4...
#> 244 11016762 MULTIPOLYGON (((-75.69595 4...
```

```
ggplot2::ggplot(pa_counties_raw) +
  ggplot2::geom_sf() +
  ggplot2::theme_void()
```



Basic map of PA counties. Source: U.S. Census Bureau TIGER/Line Shapefiles.

For non-US applications, the package `rnaturalearth`, which is a well-supported part of the rOpenSci project, provides easy access to global data. Like `tigris`, we can import directly as a simple features object. Here's a quick look at how to import all the countries in Asia.

```
library(rnaturalearth)
library(ggplot2)

asia <- rnaturalearth::ne_countries(
  continent = "Asia",
  returnclass = "sf"
)

head(asia, 0)
#> Simple feature collection with 0 features and 63 fields
#> bbox:           xmin: NA ymin: NA xmax: NA ymax: NA
```

```
#> CRS:            +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84
+towgs84=0,0,0
#>  [1] scalerank  featurecla labelrank  sovereignt sov_a3     adm0_dif
#>  [7] level      type       admin      adm0_a3    geou_dif   geounit
#> [13] gu_a3      su_dif     subunit    su_a3      brk_diff   name
#> [19] name_long  brk_a3     brk_name   brk_group  abbrev     postal
#> [25] formal_en  formal_fr  note_adm0  note_brk   name_sort  name_alt
#> [31] mapcolor7  mapcolor8  mapcolor9  mapcolor13 pop_est    gdp_md_est
#> [37] pop_year   lastcensus gdp_year   economy    income_grp wikipedia
#> [43] fips_10    iso_a2     iso_a3     iso_n3     un_a3      wb_a2
#> [49] wb_a3      woe_id     adm0_a3_is adm0_a3_us adm0_a3_un adm0_a3_wb
#> [55] continent  region_un  subregion  region_wb  name_len   long_len
#> [61] abbrev_len tiny       homepart   geometry
#> <0 rows> (or 0-length row.names)
```

```
ggplot2::ggplot(asia) +
  ggplot2::geom_sf() +
  ggplot2::theme_void()
```

Basic map of countries in Asia. Source: rnaturalearth R package.

## Finding non-geospatial data

Chances are that you are coming to a geospatial mapping project with a particular dataset in mind. Perhaps you want to explore the *New York Times'* Covid-19 data. Or perhaps you are interested in FiveThirtyEight's hate crimes by state data). Your data likely has the statistics you want but not the geometry you need for graphing. Hopefully, your data has an ID that you can use to identify each geospatial region. In the example hospital data below, the PA Department of Health provides a ZIP Code and a County name. We also have a longitude and latitude that could be coerced into a simple features geometry (it isn't one yet, though...just a column with a numeric value).

```
library(readr)

# Hospitals by county
```

```
# Data from PASDA
# Source: https://www.pasda.psu.edu/uci/DataSummary.aspx?dataset=909

pa_hospitals_url <- "https://www.pasda.psu.edu/spreadsheet/DOH_Hospitals201912.csv"
pa_hospitals_raw <- readr::read_csv(url(pa_hospitals_url))

head(pa_hospitals_raw)
#> # A tibble: 6 x 19
#>   SURVEY_ID_ FACILITY_I LONGITUDE LATITUDE FACILITY_U GEOCODING_ FACILITY_
#>
#> 1 1357       135701          -80.3     40.7 http://ww~ 00         HERITAGE ~
#> 2 0040       13570101        -80.3     40.7 http://cu~ 00         CURAHEALT~
#> 3 1370       137001          -79.9     40.2 http://ww~ 00         MONONGAHE~
#> 4 0047       53010101        -79.7     40.6 https://w~ 00         NEW LIFEC~
#> 5 7901       790101          -79.7     40.6 http://ww~ 00         ALLEGHENY~
#> 6 0023       490601          -80.2     40.4 http://cu~ 00         CURAHEALT~
#> # ... with 12 more variables: STREET , CITY , ZIP_CODE ,
#> #   ZIP_CODE_E , CITY_BORO_ , COUNTY , AREA_CODE ,
#> #   TELEPHONE_ , CHIEF_EXEC , CHIEF_EX_1 , LAT , LNG
```

Let's think for a moment, though about geospatial analysis. Having the number of hospitals in a county is useful, but what we really want to know is the number of hospitals per capita. Often times with geospatial visualizations, we want to know penetration rates per capita. To do this, we will need to find census data.

U.S. Census data is available at census.gov. The data.census.gov website is not always the most intuitive to navigate, as the data live in many different tables from different governemtn surveys. In addition to the 10-year census survey, there are over 130 intermediate-year surveys including the American Community Survey (ACS). You can browse surveys and available data to your heart's content. Compounding this difficulty is the Census Bureau's naming convention. If you want median household income, for example, you need to look for variable "B19013_001." Once you manage to struggle through all that, you can download a CSV with your desired census data.

```
library(readr)

county_pop_url <- "https://www2.census.gov/programs-surveys/popest/datasets/2010-2019
/counties/totals/co-est2019-alldata.csv"
county_pop_raw <- readr::read_csv(url(county_pop_url))

head(county_pop_raw)
#> # A tibble: 6 x 164
#>   SUMLEV REGION DIVISION STATE COUNTY STNAME CTYNAME CENSUS2010POP
#>
#> 1 050         3        6 01    000    Alaba~ Alabama       4779736
#> 2 050         3        6 01    001    Alaba~ Autaug~         54571
#> 3 050         3        6 01    003    Alaba~ Baldwi~        182265
#> 4 050         3        6 01    005    Alaba~ Barbou~         27457
#> 5 050         3        6 01    007    Alaba~ Bibb C~         22915
#> 6 050         3        6 01    009    Alaba~ Blount~         57322
#> # ... with 156 more variables: ESTIMATESBASE2010 , POPESTIMATE2010 ,
#> #   POPESTIMATE2011 , POPESTIMATE2012 , POPESTIMATE2013 ,
#> #   POPESTIMATE2014 , POPESTIMATE2015 , POPESTIMATE2016 ,
#> #   POPESTIMATE2017 , POPESTIMATE2018 , POPESTIMATE2019 ,
```

```
#> #   ...
```

# Combining spatial data with non-spatial data

Now we have our county geospatial data, our original dataset with GDP data, and another datset with census data. How in the world do we plot this?

Four simple steps to prepare your data for graphing.

1. Import all data (already completed above)
2. Clean your geospatial data frame
3. Combine non-spatial data into a single, clean data frame
4. Merge your two data frames together

## Step 1: Import all data

This was completed above, but to refresh your memory, we have `pa_counties_raw` (spatial), `pa_hospitals_raw` (non-spatial), and `county_pop_raw` (non-spatial).

## Step 2: Clean your geospatial data frame

Not much work to do here, but just to demonstrate how it's done, let's drop and rename some columns.

```
library(dplyr)

# Even though we don't select "geometry,", the sf object will keep it
# To drop a geometry, use sf::st_drop_geometry()
pa_counties <- pa_counties_raw %>%
  dplyr::transmute(
    GEOID,
    MAP_NAME = NAME, # Adams
    COUNTY = toupper(NAME) # ADAMS
  )
```

## Step 3: Combine non-spatial data into a single, clean data frame

```
library(dplyr)
library(tidyr)

pa_hospitals <- pa_hospitals_raw %>%
  dplyr::group_by(COUNTY) %>%
  dplyr::summarise(N_HOSP = n()) %>%
  dplyr::ungroup()

head(pa_hospitals)
#> # A tibble: 6 x 2
#>   COUNTY    N_HOSP
#>
#> 1 ADAMS          1
#> 2 ALLEGHENY     28
#> 3 ARMSTRONG      1
#> 4 BEAVER         2
#> 5 BEDFORD        1
#> 6 BERKS          6
```

```
pa_pop <- county_pop_raw %>%
  dplyr::filter(SUMLEV == "050") %>% # County level
  dplyr::filter(STNAME == "Pennsylvania") %>%
  dplyr::select(
    COUNTY = CTYNAME,
    POPESTIMATE2019
  ) %>%
  dplyr::mutate(
    COUNTY = toupper(COUNTY),
    COUNTY = gsub("(.*)( COUNTY)", "\\1", COUNTY)
  ) # "Adams County" --> "ADAMS COUNTY" --> "ADAMS"

head(pa_pop)
#> # A tibble: 6 x 2
#>   COUNTY    POPESTIMATE2019
#>
#> 1 ADAMS              103009
#> 2 ALLEGHENY         1216045
#> 3 ARMSTRONG           64735
#> 4 BEAVER             163929
#> 5 BEDFORD             47888
#> 6 BERKS              421164

combined_pa_data <-
  dplyr::full_join(pa_hospitals, pa_pop, by = "COUNTY") %>%
  tidyr::replace_na(list(N_HOSP = 0)) %>%
  dplyr::mutate(HOSP_PER_1M = N_HOSP / (POPESTIMATE2019/1000000))

head(combined_pa_data)
#> # A tibble: 6 x 4
#>   COUNTY    N_HOSP POPESTIMATE2019 HOSP_PER_1M
#>
#> 1 ADAMS          1          103009        9.71
#> 2 ALLEGHENY     28         1216045       23.0
#> 3 ARMSTRONG      1           64735       15.4
#> 4 BEAVER         2          163929       12.2
#> 5 BEDFORD        1           47888       20.9
#> 6 BERKS          6          421164       14.2
```

### Step 4: Merge your two data frames together

The `tigris` package mentioned above has a function for combining geospatial data with a standard data frame. We need to provide `tigris::geo_join` with our datasets and three instructions.

- `by_sp`: Column name from my spatial data to identify unique fields (*e.g.*, COUNTY, ZIP_CODE, FIPS)
- `by_df`: Column name from my non-spatial data to identify unique fields
- `how`: "inner" to keep rows that are present in both datasets or "left" to keep all rows from the spatial dataset and fill in NA for missing non-spatial rows

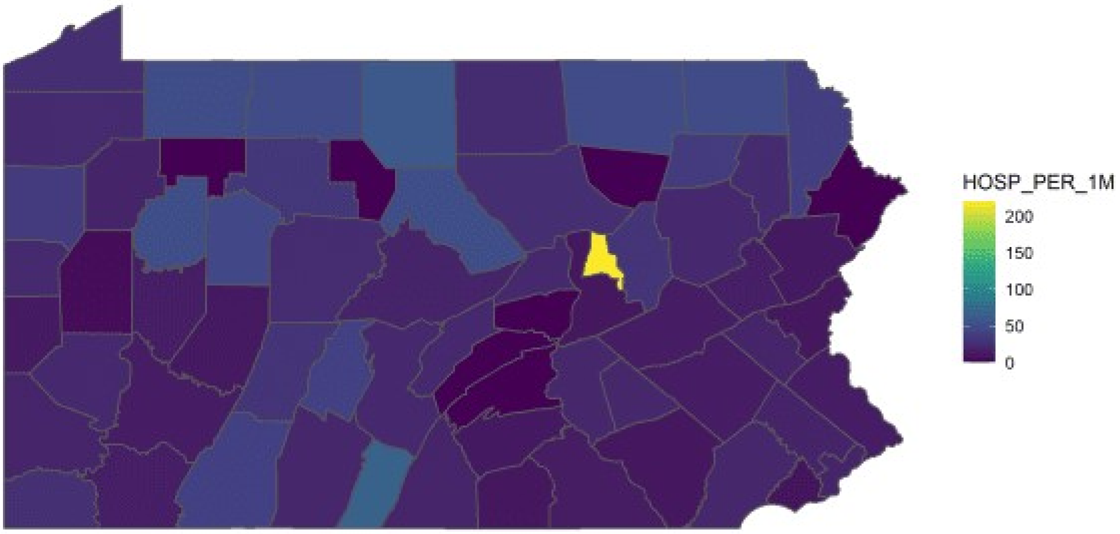
```
library(tigris)
```

```
library(ggplot2)

# Combine spatial and non-spatial data
pa_geospatial_data <- tigris::geo_join(
  spatial_data = pa_counties,
  data_frame = combined_pa_data,
  by_sp = "GEOID",
  by_df = "COUNTY",
  how = "inner"
)

head(pa_geospatial_data)
#> Simple feature collection with 6 features and 6 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -80.51942 ymin: 39.72089 xmax: -75.01507 ymax:
41.47858
#> geographic CRS: NAD83
#>   GEOID   MAP_NAME     COUNTY N_HOSP POPESTIMATE2019 HOSP_PER_1M
#> 1 42005  Armstrong  ARMSTRONG      1           64735    15.44759
#> 2 42029    Chester    CHESTER     11          524989    20.95282
#> 3 42035    Clinton    CLINTON      2           38632    51.77055
#> 4 42059     Greene     GREENE      1           36233    27.59915
#> 5 42067    Juniata    JUNIATA      0           24763     0.00000
#> 6 42091 Montgomery MONTGOMERY     15          830915    18.05239
#>                       geometry
#> 1 MULTIPOLYGON (((-79.69293 4...
#> 2 MULTIPOLYGON (((-75.59129 3...
#> 3 MULTIPOLYGON (((-78.09338 4...
#> 4 MULTIPOLYGON (((-80.51942 3...
#> 5 MULTIPOLYGON (((-77.74677 4...
#> 6 MULTIPOLYGON (((-75.69595 4...

ggplot2::ggplot(pa_geospatial_data, aes(fill = HOSP_PER_1M)) +
  ggplot2::geom_sf() +
  ggplot2::scale_fill_viridis_c() +
  ggplot2::theme_void()
```




Hospitals per million residents. Montour County is apparently the place to be if you need a hospital!

Source: PASDA, U.S. Census Bureau

### An aside on U.S. Census Bureau data

If you are using data from the U.S. Census Bureau, the easist option is to use the `tidycensus` package for it allows you to access census data that comes pre-joined to shapefile data. The package also has helper functions for easy navigation of the U.S. Census Bureau datasets. `tidycensus` uses the U.S. Census API, so you will need to obtain an API key first. The `tidycensus` website has great instruction c how to get an API key and add it to your .Renviron file. The website also has excellent example vignettes to demonstrate the package's robust functionality.

## Conclusion

To conclude, we have now seen how to find geospatial data and import it into R as a simple features object. We have seen how to find U.S. Census data or other non-spatial data and import it into R. We have seen how to tidy both geospatial and non-spatial data and join them as a single dataset in preparation for visualization. Finally, we have had a brief preview of how to plot using `ggplot2`. Visualization was not a focus of this post, but how can we write a whole post on maps without showing you a single map! Consider it a teaser for Part 2, when we discuss visualization in more detail.