ACS Data & Mapping

Susan E. Chen

Contents

```r
knitr::opts_chunk$set(echo = TRUE) # setting so all your code chunks displayed.
#Load all packages you wish to use in this RMD file here
```

Introduction

In 2020 we introduced the VT Data Science for the Public Good (DSPG) program in the agricultural and applied economics program at Virginia Tech. The experience centers around a capstone-type project the students would work on for ten weeks during the summer. The students work on projects from university partners like Virginia Cooperative Extension (VCE). Over the last four years, data analytics and the creation of a ShinyApp have been the medium that excites and brings the summer capstone projects to life for the undergraduate intern. Data analytics at this level includes exploratory data analysis and visualization tools, including maps, plots, and other creative ways to present data. However, the icing on the cake that seems to excite the students is the publication of a shiny app that is easily accessible and a valuable artifact to list on resumes.

While our theoretical models guide how we frame a problem, inevitably, we must find an efficient way to apply our theories to the project at hand. Our economic thinking impacts the variables we choose to curate and the construction of measures we can use to glean actionable insight; however, finding approachable and easy-to-use tools that our (non-data intensive) stakeholders can use is equally important. This paper presents the first independent data analysis exercise we teach our students in the VT-DSPG program. In this exercise, we teach the student how to download Census data using the `R` software. We show them how to use this raw data to construct meaningful statistics and to create interesting visualizations such as bar charts. We go a step further and show them how to manage these data and add them as layers to maps at varying levels of geography. We then make the maps interactive and create a simple Shiny App, which we publish online. This exercise gets the students up and running in `R` quickly, and publishing the shiny app online provides the students with an immediate sense of accomplishment and excitement. All the materials used in this example are available in the following GIT repository@@.

Preliminairies: Installing R, R Studio and R Markdown

We assume that all the students have installed `R` and `RStudio` which are all freely available. Instructions for installing the software are available at https://rstudio-education.github.io/hopr/starting.html for both Mac and Windows operating systems. We also use `rmarkdown` which can be installed with these instructions https://bookdown.org/yihui/rmarkdown/installation.html. At this time you may want to also install `tinytex` if you think you would like to write your files to pdf. We will not write to pdf's in this exercise.

We are going to use RStudio as our integrated development environment. This will allow the student to see the instructions and the R code chunks and run them all within the same environment. At this time you can share the file R-exercise1.Rmd with your students. They can double click on the file in the file manager and the file will open in the Rstudio IDE. We recommend that the student always start with a template like this first rather than a blank page.

Part 1: extracting American Community Survey Data (30 mins)

Our first exercise uses data obtained from the Census Bureau. We use the `R` package to interface with the US Census Bureau's API to efficiently download data that we then use for the rest of the exercise. By using `R` to obtain the data from Census this entire exercise is self contained to just coding alone. No extraneous data files are needed. For teaching purposes, this makes the teaching process compact and self-contained. The `tidycensus` package will be used to interact with the Census website.

Downloading Census Data With `R` and the `tidyCensus` Package

The `tidycensus` package was developed by the Census Bureau and allows the user to interface with the Census website and to easily pull Census data. Programming is an efficient way to manage data. You have a record of the variables you pulled, where you pulled them from and can easily amend the code to include more variables or to make other changes – as you will see me do in this exercise. In other words, programming and pulling data in this way allows for replicability which is an important lesson to teach students early on in the research process.

You should install the `tidycensus` package if you do not already have it. Recall to install a package you type `install.packages("tidycensus")` or you can click `Tools` in the above menu then `Install` and then type in tidycensus in the dialog box. Once you have installed the `tidycensus` package you can use the package using library(tidycensus) as I have done in the code chunk below. @@Note to execute a code chunk in R studio click on the green arrow that you see in the right hand side of the R Script Window as you see in the diagram below.

```
library(tidycensus)
```

Getting a Census API key And Saving It On Your Hard Drive

For the public to interact with the Census Website you have to get a Census API key. You can get a Census API key here. Please fill in your email address and organization then submit to get your key. A key will be emailed to you. Note your key is like a password so you should make sure to secure it. I recommend that you save this API key in a `.Renviron` on your hard drive. If you do this then you do not need to remember it or type it in every time you want to use it in `R`. Below I provide a command that shows you how to save your Census API key on your hard drive in a .Renviron file. {footnote: An `.Renviron` file is a way for R and RStudio to look for a file that you can use to control the behavior of your R session, for example by setting options or environment variables.}

The easiest way to create a .Renviron file is to use the `tidycensus` package. The code below uses the tidycensus package and the `census_api_key()` command to read in your Census API key and creates a `.Renviron` where it saves the key for perpetuity on your hard drive. Since this is the first time you are installing an API key then you should set the option `install = TRUE`. In the code chunk below please insert your key where you see "XXXXXXX".

The `census_api_key()` command only needs to be run once. After this you should be able to use this key in the future with the readRenviron line as I have done below. I recommend you delete or comment out the code chunk once you have read in yoru key.

```r
#census_api_key("XXXX",  install = TRUE) #only do this once then delete this code chun
```

You then use the the `readRenviron("~/.Renviron")` to read in your Census API key for this `R` session. Note your Census key is not written in the script and remains confidential using this method.

```r
readRenviron(".Renviron") #read in your Census API key
```

If you want to check to see if your key has been read in properly use the following `Sys.getenv("CENSUS_API_KEY")` command. Your key will display in your console confirming it was read in properly.

```r
Sys.getenv("CENSUS_API_KEY") # You can check your key
```

Using Census Data

Identifying US counties, states, and census tracts

Much of the work that we do for our projects center on studying rural counties with populations that are too small to be reported in annual Census data products (@reference). The Census product that we use will then be the American Community Survey (ACS). The ACS combines data over multiple years for counties with low population, this is where you can find county level information on many socioeconomic variables outside of the Decennial Census. We will use the combined 5 year ACS data (ACS5) in this exercise. It is important to note that other Census products can be pulled using the `tidycensus` package. Please read here is you are interested in the annual ACS, or the Decennial Census.

Here is the exercise we give to the student:

Aim: Choose a rural county in Virginia. Then use the `tidycensus` package to download census track level information from the ACS5.

Geographic Region: The objective of this exercise is to identify a county. It is important that we think about the Census geographic boundary that we are interested in studying and the time period. As previously mentioned, for many smaller counties, there is no annual data so the only option is a sliding 5 year windows (ACS5). Once you have identified the geographic boundary it is best if you know the abbreviation for the geographical area and the FIPS code. Here is a list of FIPS codes for all US counties database here. As you will see below, knowing the FIPS for counties and states will help you to easily pull data from Census using the `tidycensus` package in `R`.

Variables: We are now going to pull ACS data. To have R pull data for you quickly requires that you know the name of the variable you want to pull as it is listed on the Census website. This requires you browse the Census website and identify the data table you are interested in using. In the example below, my variable of interest is median population income. To find this variable I look at the ACS data tables online to make sure what I need is there and note how it is displayed. *You must always check the data table, you cannot just pull data.*

Once I am sure that the information I want is in the Census ACS5 data tables, I will then identify the "names" of the variables I want to pull from the Census archive. You can teach yourself how to figure out the names by using this Census tutorial link. There is also a pdf of the webinar at the same link. I prefer the pdf because I can scan it quickly but the webinar video is really good for first timers. The video explains how to identify the name of the variable you want to pull. After a while, finding the tables you want will get easier. An expert level step for getting the variable names, and the one I will use here, is to read in a data table from Census and ask for the variable names. The variable names are then written to a list called `ACS5_var_list` as in the code chunk below. I use the `load_variables()` command from `tidycensus` with the options year=2019 and dataset="acs5". This pulls 5 year ACS data for 2015-2019. I also set `cache = TRUE` which allows you to cache the data for future access.

You can initialize a variable this.year to pull the year of interest. In the code chunk below this.year=2019 means that the data will be pulled for 2015-2019. This is a easy easy way in the future to incorporate changes without having to search around in the code to change or add in another year of data.

```
# Set a year of interest
this.year = 2019

# This looks at the 5 year estimates
# You can also do 1 year estimates with the option: dataset = "acs1"
ACS5_var_list <- load_variables(year = this.year,
                      dataset = "acs5",
                      cache = TRUE)
```

To check how many variables are in the ACS5 you can use the dim command on the ACS5_var_list to obtain the dimensions of the list.

```
# Get dimension of the vars list
Numvars <- dim(ACS5_var_list)
```

As you can see, there are 27040, 4 variables described in `varlist`. The `head()` function provides a printout of the first 6 variables in the list along with a description of the variable in the ACS5.

```
## # A tibble: 6 x 4
##   name       label                                    concept    geography
##   <chr>      <chr>                                    <chr>      <chr>
## 1 B01001_001 Estimate!!Total:                         SEX BY AGE block group
## 2 B01001_002 Estimate!!Total:!!Male:                  SEX BY AGE block group
## 3 B01001_003 Estimate!!Total:!!Male:!!Under 5 years   SEX BY AGE block group
## 4 B01001_004 Estimate!!Total:!!Male:!!5 to 9 years    SEX BY AGE block group
## 5 B01001_005 Estimate!!Total:!!Male:!!10 to 14 years SEX BY AGE block group
## 6 B01001_006 Estimate!!Total:!!Male:!!15 to 17 years SEX BY AGE block group
```

Please be cautious and make sure you understand the naming convention and check with published Census tables to make sure you are pulling the correct variables.

Variable Names in Census

Once I have these variables, you must decide on the geographical unit i.e. the state, the county, or the tract, etc. In the example below I would like to pull the tract-level variables for Powhatan Virginia.

Pulling one variable from American Community Survey

To start, you can pull one variable at the Census tract level for your county and saves this variable to an `R` dataframe. The variable I will pull is median.household.income which we know from the table is `B19013_001` (see ACS5_var_list that we created above to confirm this). You can specify a vector of variables that you want to pull and save. The Pow_vars serves two purposes: (1) to specify the variables I want to pull from the Census website and (2) to rename the variables to more user friendly names.. You can also take this opportunity in this vector to rename the variable to something more user friendly such as `median.household.income`.

The `get_acs` function that you use comes from the `tidycensus` package in `R`. You can use the following options `survey = "acs5"` and `year= this.year`. Other options include the geographical area, `state="VA"`, `county = "Powhatan"`, and `geography = tract`. Finally, I put the variable I want to pull but I also change it to a more user friendly name with `variables = (median.household.income ="B19013_001")`.

I save the information I have pulled in a list called `Pow_income`.

```
Pow_vars <- c(median.household.income = "B19013_001") #vector of variables
my_county <- c("Powhatan") #vector of counties

Pow_income <- get_acs(survey = "acs5", geography = "tract", year=this.year,
                state = "VA", county = my_county, geometry = FALSE,
                variables = Pow_vars)
```

```
## Getting data from the 2015-2019 5-year ACS
```

```
Pow_income #displays the variables in the pulled list
```

```
## # A tibble: 5 x 5
##   GEOID       NAME                                     variable estimate   moe
##   <chr>       <chr>                                    <chr>       <dbl> <dbl>
## 1 51145500101 Census Tract 5001.01, Powhatan County, Vi~ median.~    98906  7471
## 2 51145500102 Census Tract 5001.02, Powhatan County, Vi~ median.~    97596  9668
## 3 51145500200 Census Tract 5002, Powhatan County, Virgi~ median.~    86469  9294
## 4 51145500300 Census Tract 5003, Powhatan County, Virgi~ median.~    93438 38203
## 5 51145500400 Census Tract 5004, Powhatan County, Virgi~ median.~    71004  9647
```

The above output shows the structure of the dataset and the variables that were pulled from the Census API. These are:

```
GEOID - an ID for each tract in Powhatan county
name - the Census name of the tract (FIPS tract id)
variable - the variable name (median.household.income)
estimate - the estimate of median.household.income from the ACS5
moe - the margin of error or degree of uncertainty of the estimate.
```

Pulling multiple variables from American Community Survey

To pull multiple variables from the ACS5 you can specify more variables in the `Pow_vars` vectors. As you can see, adding more variables to pull from the Census API site is easy with this method. The final option `output="wide"` is very helpful to use as it puts the data into one row per Census tract.

```
Pow_vars <- c(white = "B03002_003", #declare variables to pull and rename them
              af.am = "B03002_004",
              hispanic = "B03002_012",
              am.ind = "B03002_005",
              asian = "B03002_006",
              nh.pi = "B03002_007",
              multiple = "B03002_009",
              other = "B03002_008",
              population = "B02001_001")

MorePow <- get_acs(survey = "acs5", geography = "tract", year=this.year,
                   state = "VA", county = "Powhatan", geometry = FALSE,
                   variables = Pow_vars, output = "wide")
```

```
## Getting data from the 2015-2019 5-year ACS
```

```
MorePow[1:5,1:7] #view some subset of list
```

Data Wrangling - removing unwanted columns

The list `MorePow` include both the estimate (variable ending with an E) and the margin of error (variable ending with an M). To remove the columns that end with “M” from the MorePow list use the `dplyr` package and the `select()` function with the minus operator (`-`) to delete the unwanted columns. Below the `-otherE` variable is also removed because it contains just a vector of 0’s. To specify the list of variables with an M in the title use the shorthand command `grep`. The grep function searches the variables names with an M and saves them in the vector `columns_to_remove`. The `select(-columns_to_remove, -otherE)` removes the `otherE` column and all the columns with an M.

```
#Quick way to save variable names with an M in a vector
columns_to_remove <- grep("M", names(MorePow)) #vector of MOE variables to delete
```

```
#Turn a list into a dataframe to use piping
data.frame(MorePow) #turn list into a data frame
```

```
##         GEOID                                             NAME whiteE whiteM
## 1 51145500101 Census Tract 5001.01, Powhatan County, Virginia   6199    341
## 2 51145500102 Census Tract 5001.02, Powhatan County, Virginia   4889    365
## 3 51145500200    Census Tract 5002, Powhatan County, Virginia   8288    424
## 4 51145500300    Census Tract 5003, Powhatan County, Virginia   1129    166
## 5 51145500400    Census Tract 5004, Powhatan County, Virginia   4199    379
##   af.amE af.amM hispanicE hispanicM am.indE am.indM asianE asianM nh.piE nh.piM
## 1    219    157       113       131       0      17      5     11     15     27
## 2    187     74        58        62      14      27      0     17     31     51
## 3    317    157       114       102      12      20     29     29      0     17
## 4   1555    190       156        65      16      19     39     29      0     12
## 5    617    218       190       134       0      17      0     17      0     17
##   multipleE multipleM otherE otherM populationE populationM
## 1        76        61      0     17        6627         286
## 2       118        99      0     17        5297         350
## 3       173       140      0     17        8933         405
## 4        38        27      0     12        2933         256
## 5        19        31      0     17        5025         306
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
#Remove columns with select() function by piping
#creates a new data frame with no MOE variables
MorePow2 <- MorePow %>% select(-columns_to_remove, -otherE)
head(MorePow2)
```

```
## # A tibble: 5 x 9
##   GEOID      whiteE af.amE hispanicE am.indE asianE nh.piE multipleE populationE
##   <chr>       <dbl>  <dbl>     <dbl>   <dbl>  <dbl>  <dbl>     <dbl>       <dbl>
## 1 511455001~   6199    219       113       0      5     15        76        6627
## 2 511455001~   4889    187        58      14      0     31       118        5297
## 3 511455002~   8288    317       114      12     29      0       173        8933
## 4 511455003~    1129   1555         156       16      39       0          38         2933
## 5 511455004~    4199    617         190        0       0       0          19         5025
```

The list `MorePow2` has only the estimates of race for the 5 census tracts in Powhatan county as you see below. The estimates are in columns 2 to 9.

```r
head(MorePow2)
```

Data Wrangling - constructing new variables

For categorical variables like race, it is useful to construct percentages. You can use the `mutate` function in the `dplyr` package to construct many percentages at one time. Here are the steps to do this:

1. Change the MorePow2 list to another `R` object type called a data frame using the data.frame() function
2. use the mutate(across(c(2:9), .fns= ~.*100/populationE)) to use the data in columns 2:9 to create percentage

3. Save to a new dataframe called `Percent_Race`.

Data Wrangling - constructing variables

```r
library(dplyr)
Percent_Race <- MorePow2 %>%
  mutate(across(c(2:8), .fns = ~.*100/populationE)) %>%
  select(-populationE) #remove the populationE variable
```

Data Wrangling - changing variable names to nice names for labels

To change the variable names you can use the `stringr` package and the `rename` function.

```r
library(stringr)
#rename for nice variable names
 Percent_Race <- rename(Percent_Race, White=whiteE,
                  African_Am=af.amE,
                  American_In = am.indE,
                  Hispanic=hispanicE,
                  Asian=asianE,
                  NH_Pacific_Is=nh.piE,
                  Multiple=multipleE)
```

Data Wrangling - reshaping a dataframe

To create a side-by-side plot of race for all counties in Powhatan County the dataset has to conform to a format where the race information is in one column. This is done using the `tidyr` package and the `gather` function. As you can see in the output, all the information is still there, it is just in a different format and the race variable is stacked.

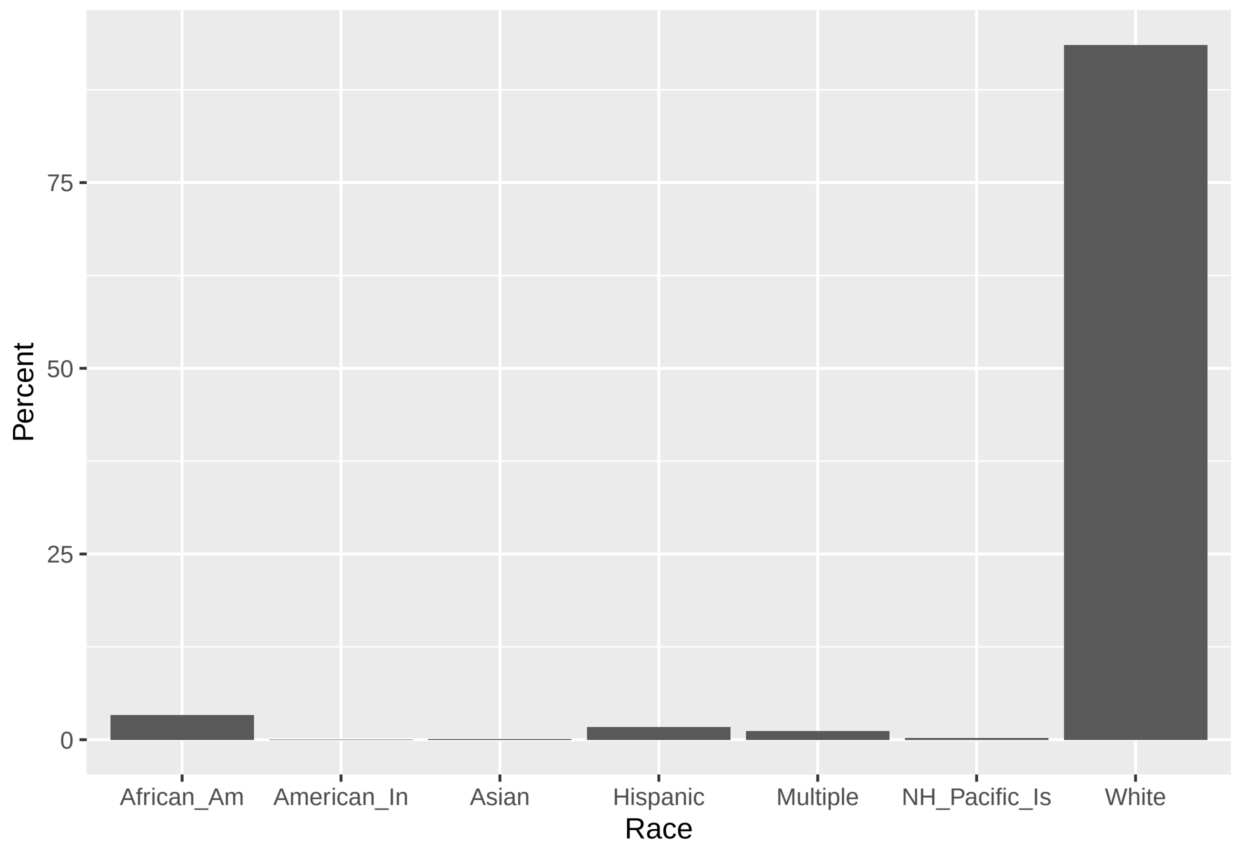
```r
library(tidyr)
long_DF <- Percent_Race %>% gather(Race, Percent, White:Multiple)
long_DF
```

```
## # A tibble: 35 x 3
##    GEOID       Race       Percent
##    <chr>       <chr>        <dbl>
##  1 51145500101 White        93.5
##  2 51145500102 White        92.3
##  3 51145500200 White        92.8
##  4 51145500300 White        38.5
##  5 51145500400 White        83.6
##  6 51145500101 African_Am    3.30
##  7 51145500102 African_Am    3.53
##  8 51145500200 African_Am    3.55
##  9 51145500300 African_Am   53.0
## 10 51145500400 African_Am   12.3
## # i 25 more rows
```

Graphing - simple barplots

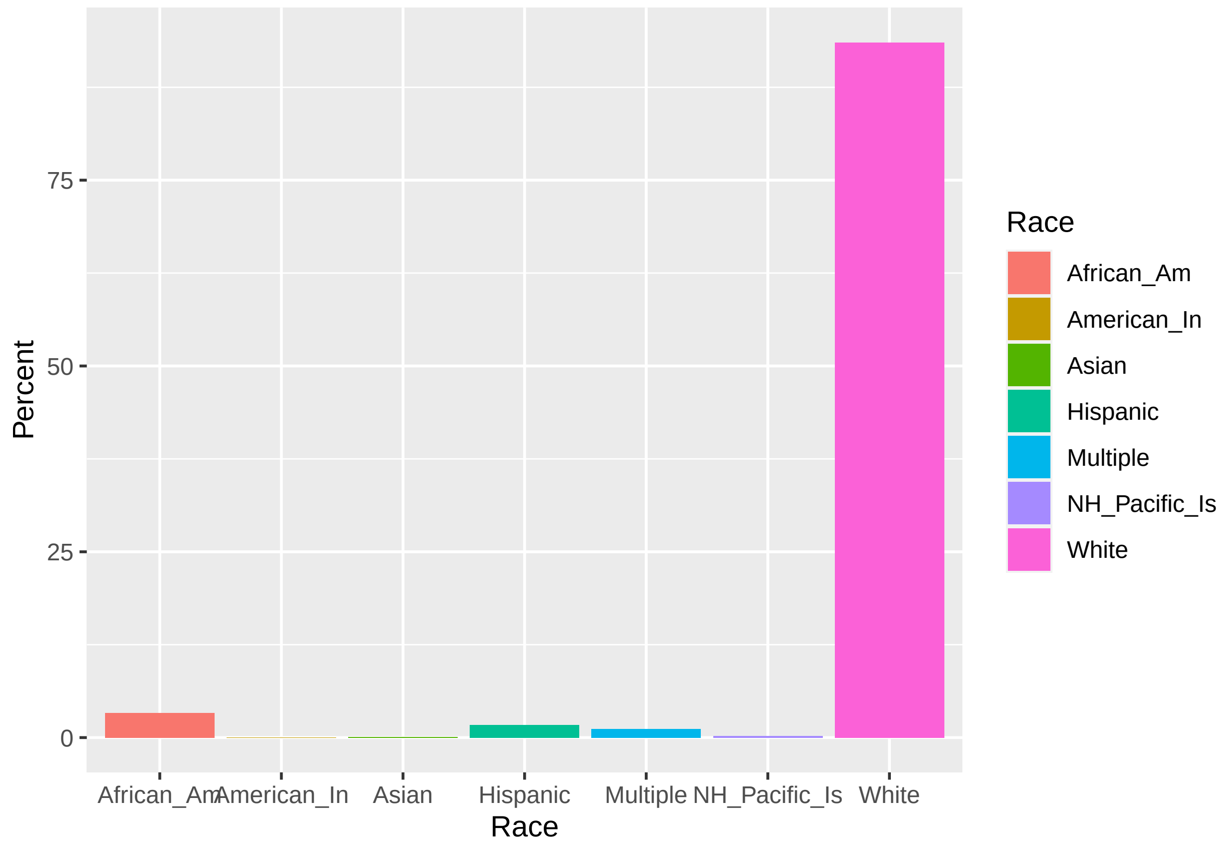
To generate a standard barplot with one variable we can use the `ggplot2` package. For just a simple barplot of one Census tract with GEOID==51145500101 you can select only the data for that tract with the `data=long_DF[long_DF$GEOID==51145500101,]`. Then the other options you set are: aes which specifies your x and y variable, geom_bar(stat="identity") is the option for a bar chart where the height is the sum of the y variable, grouped by the x variable. PowBar1 is the name you give the barchart you create for Powhatan county.

```
library(ggplot2)
PowBar1<-ggplot(data=long_DF[long_DF$GEOID==51145500101,], aes(x=Race, y=Percent)) +
  geom_bar(stat="identity")
PowBar1
```



To change the color of the race bars, you can use the fill=Race option in the aes() function

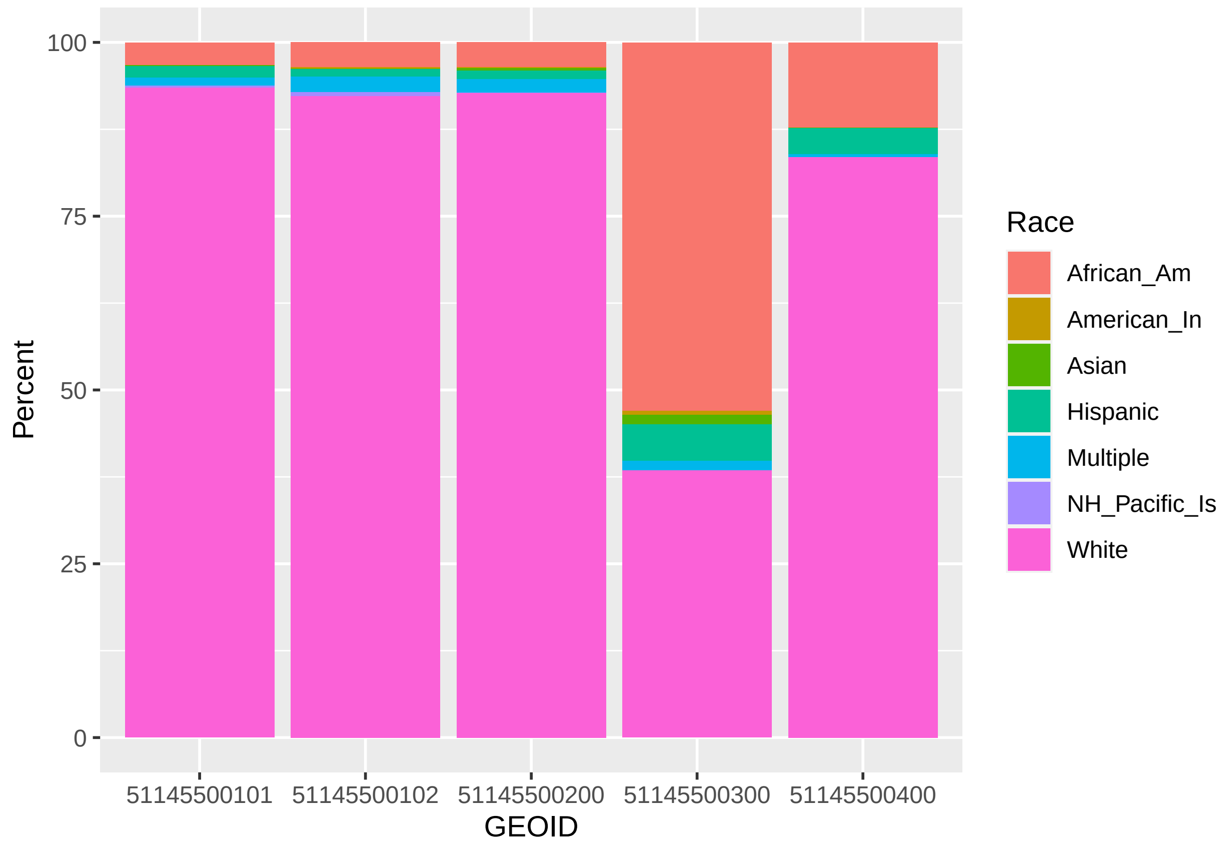
```
PowBarCol<-ggplot(data=long_DF[long_DF$GEOID==51145500101,], aes(x=Race, y=Percent, fill
  geom_bar(stat="identity")
PowBarCol
```



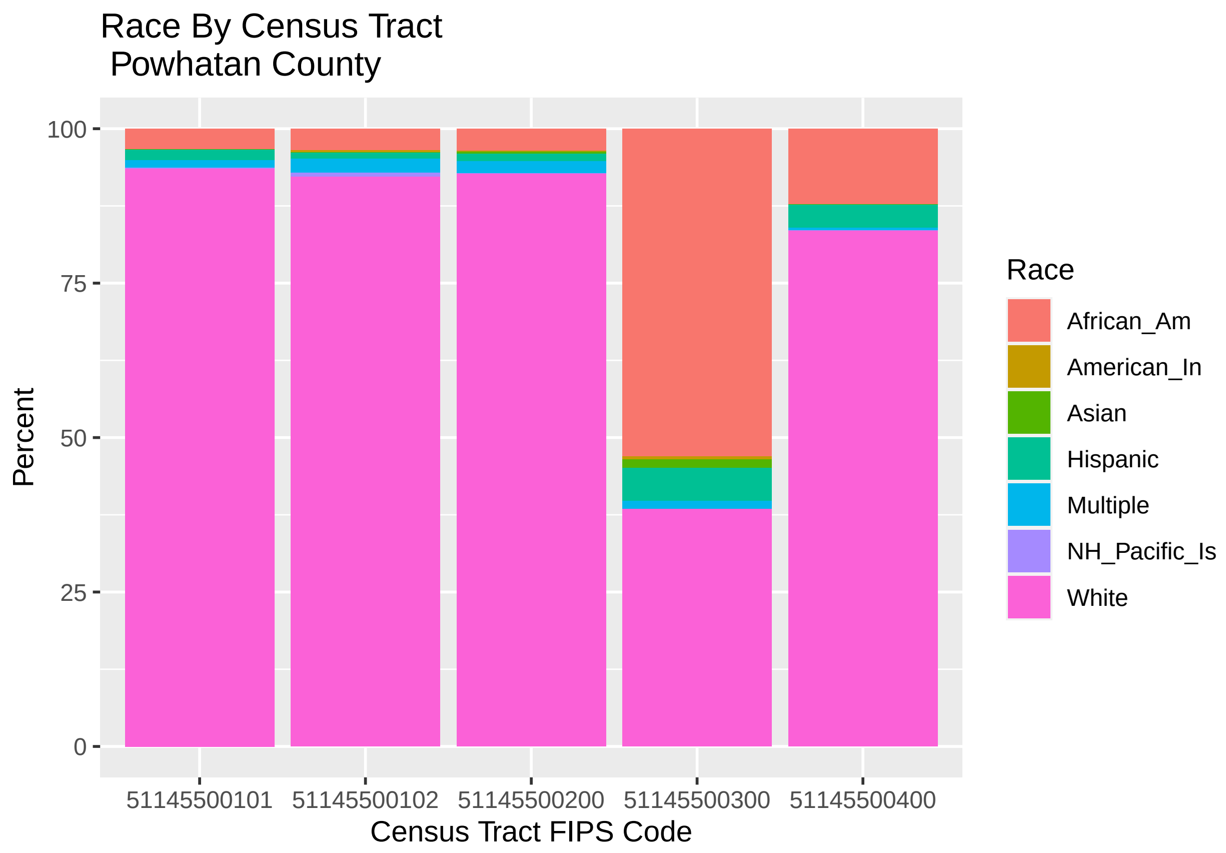
Graphing - stacked barplots

A stacked barplot allows you to extend a simple barplot to show the percentages of more than one categorical variable. In our case, you may be interested in comparing race across the Census tracts in Powhatan County. To do this we use all the data in long_DF.

```r
#library(tidyverse)
options(scipen = 999) #removes scientific notation

# Stacked barplot with multiple groups
#Add nice labels and colorblind friendly palette
pp <- ggplot(data=long_DF, aes(x=GEOID, y=Percent, fill=Race)) +
  geom_bar(stat="identity")
pp
```



```r
#Then add on titles & axis labels
pp + ggtitle("Race By Census Tract \n Powhatan County") +
  xlab("Census Tract FIPS Code") + ylab("Percent")
```



Mapping - Basic

You may also want to put Census information on a map. The `tidycensus` package and the `get_acs()` function also provide the feature geometry you will need to map the Census tracts in Powhatan county. To download the feature geometry use the `geometry=TRUE` option in the `get_acs() function`. Note you can also change the variables you want to pull in the `my_vars` vector.

```
## Set the variables you want to pull
my_vars <- c(median.household.income = "B19013_001")

Pow_income <- get_acs(survey = "acs5", geography = "tract", year=this.year,
                      state = "VA", county = my_county, geometry = TRUE,
                      variables = my_vars)
```

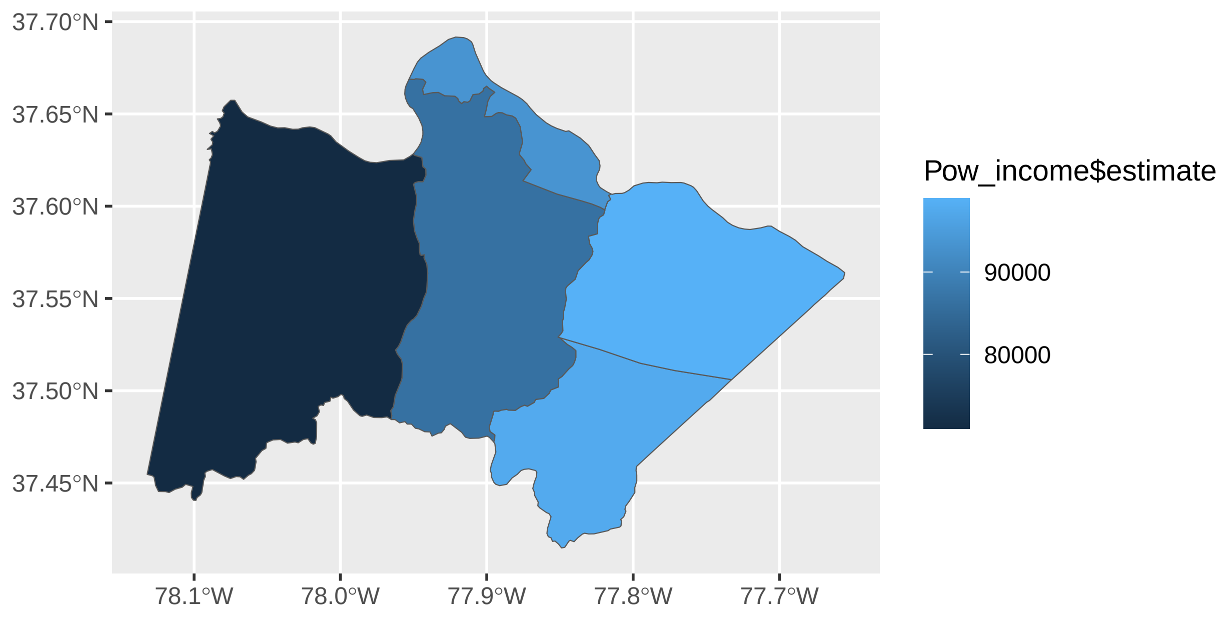
```
## Getting data from the 2015-2019 5-year ACS

## Downloading feature geometry from the Census website.  To cache shapefiles for use in
```

As you can see, the Pow_income list includes a variable called geometry that has the Census tract level boundaries for Powhatan county. We can now use these boundaries to creat map and add the estimates of median household income for each Census tract.

Mapping - A Simple Choropleth Map

You can use the plot function to quickly map your variable of interest. Recall the data has one variable and the value is stored in the variable `estimate` which is median.household.income. You can map this variable using the `ggplot()` function in the `tidyverse` package. You can save this map as `myacs_map` and then use it to add labels, axes, etc as we have done earlier in this exercise.

```
myacs_map <- ggplot(Pow_income$geometry) + #variable that holds the map boundaries
    geom_sf(aes(fill=Pow_income$estimate)) #variable you wish to map
myacs_map #show the map on the screen
```



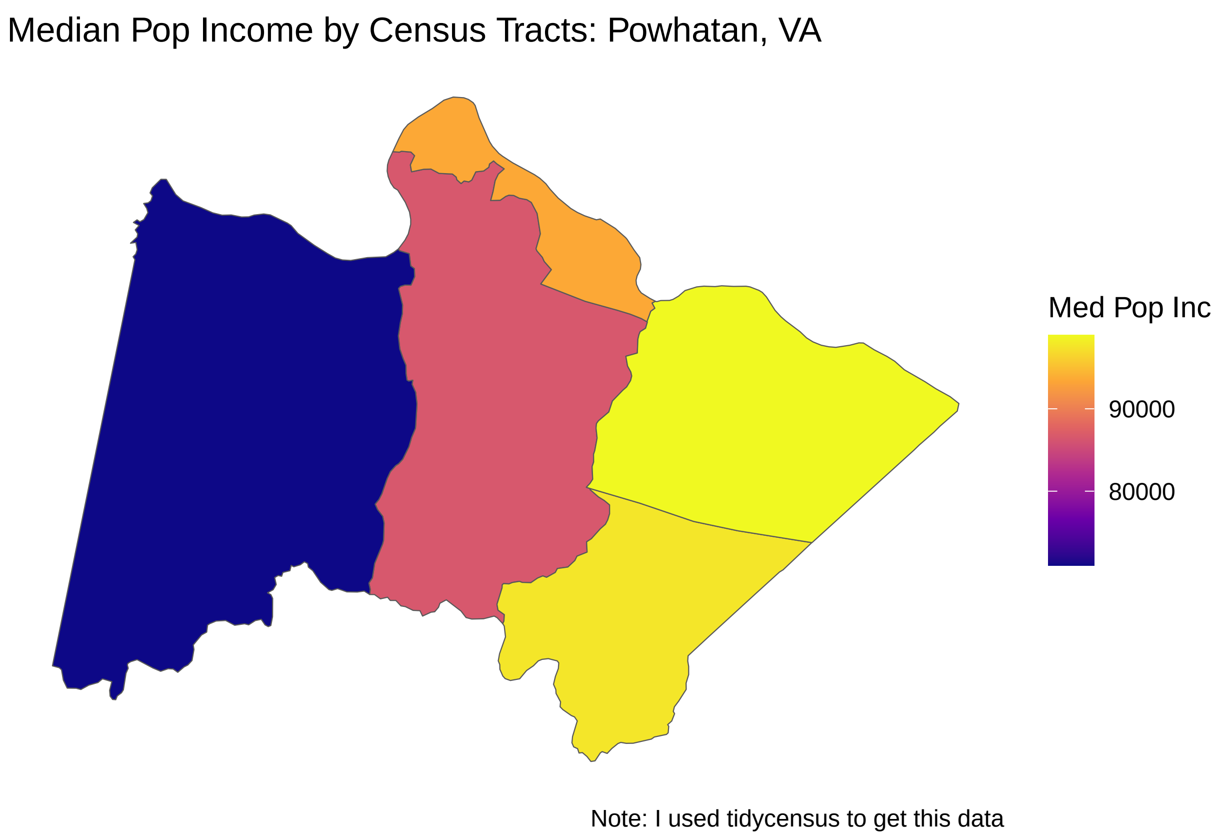
Mapping __ Colorblind Friendly Colors

To make a Chloropleth map with color blind friendly you can use the `viridis` package. The options for this package are" "viridis", "magma", "inferno", or "plasma".

```
# Provides a colorblind friendly palette.
library(viridis)
```

```
## Loading required package: viridisLite
```

```
myacs_map2 <-myacs_map +
    scale_fill_viridis_c(option = "plasma")+ #fill polygons with a color gradient
    labs(fill = "Med Pop Inc",
    title = "Median Pop Income by Census Tracts: Powhatan, VA",
    caption = "Note: I used tidycensus to get this data")
#myacs_map2


myacs_map3 <- myacs_map2 +
    coord_sf(datum = NA) + #remove coordinate system on x-axis
    theme_minimal()  #remove lines and grey background

  myacs_map3
```


Median Pop Income by Census Tracts: Powhatan, VA
Med Pop Inc
90000
80000
Note: I used tidycensus to get this data

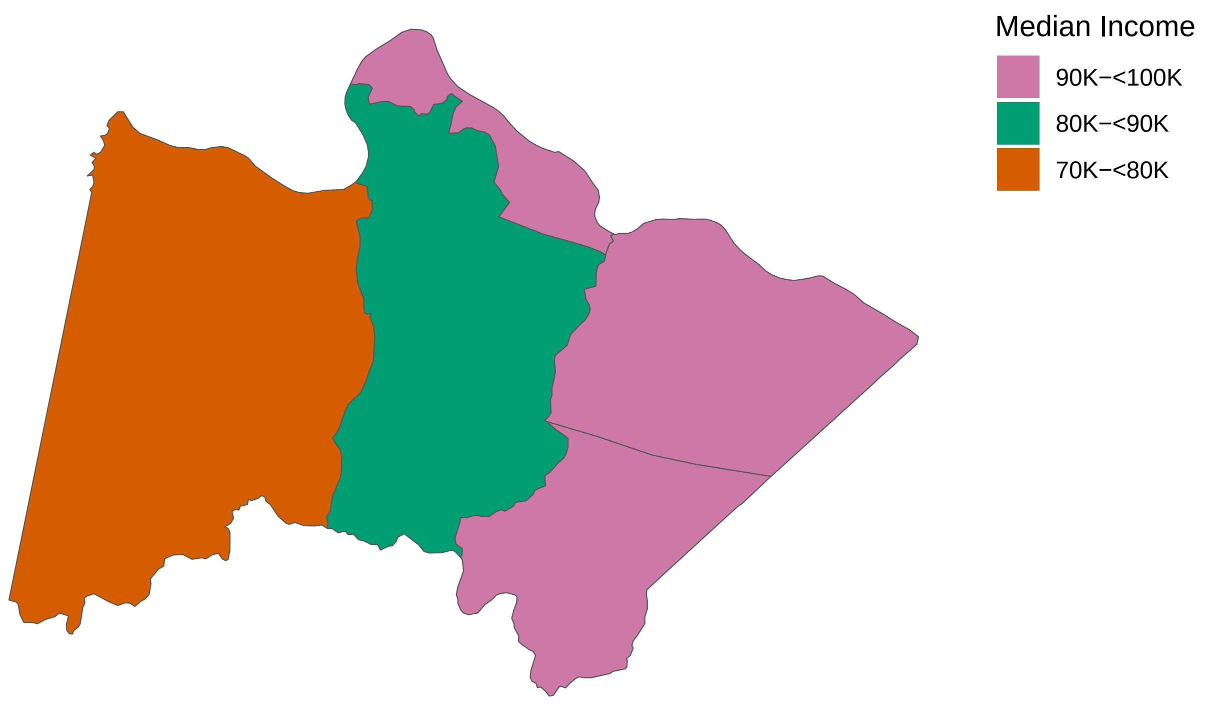
Mapping - Changing Legend Bin Sizes

To control the bin size and legend appearance you should switch the continuous income variable to a discrete scale by binning the income variable manually using the `cut()` function. With this function you can set the breaks to the where you want them. I also set the colors I want to use in the legend for each bin. I save these in my_colours. I can then use this information to plot a map with discrete income categories using ggplot.

```
data.frame(Pow_income) #turn list into a data frame
```

```
##         GEOID                                         NAME
## 1 51145500200    Census Tract 5002, Powhatan County, Virginia
## 2 51145500102 Census Tract 5001.02, Powhatan County, Virginia
## 3 51145500101 Census Tract 5001.01, Powhatan County, Virginia
## 4 51145500300    Census Tract 5003, Powhatan County, Virginia
## 5 51145500400    Census Tract 5004, Powhatan County, Virginia
##                  variable estimate   moe                       geometry
## 1 median.household.income    86469  9294 MULTIPOLYGON (((-77.96564 3...
## 2 median.household.income    97596  9668 MULTIPOLYGON (((-77.89829 3...
## 3 median.household.income    98906  7471 MULTIPOLYGON (((-77.8513 37...
## 4 median.household.income    93438 38203 MULTIPOLYGON (((-77.94981 3...
## 5 median.household.income    71004  9647 MULTIPOLYGON (((-78.13205 3...
```

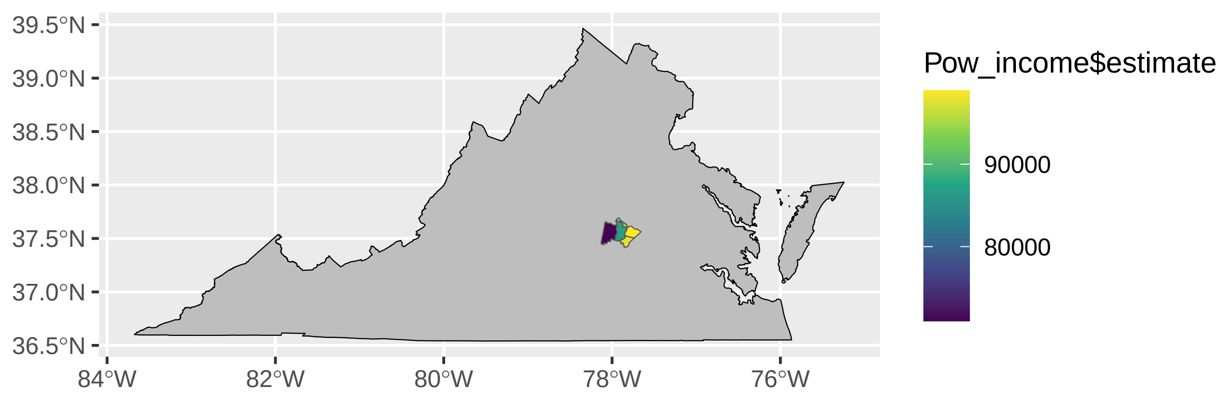
```r
Pow_income$Inc_bins <- cut(Pow_income$estimate,
  breaks = c(70000, 80000, 90000, 100000),
  right = FALSE
)

# Create custom colorblind friendly colors
customvermilion <- rgb(213/255,94/255,0/255)
custombluegreen <- rgb(0/255,158/255,115/255)
customblue <- rgb(0/255,114/255,178/255)
customskyblue <- rgb(86/255,180/255,233/255)
customreddishpurple <- rgb(204/255,121/255,167/255)
customblack <- rgb(0/255,0/255,0/255)
customorange <- rgb(230/255,159/255,0/255)
customyellow <- rgb(240/255,228/255,66/255)

my_colours <- c(customvermilion, custombluegreen, customreddishpurple, customskyblue)

myacs_map4 <- ggplot(data = Pow_income) +
  geom_sf(aes(fill = Inc_bins)) +
  theme_void() +
  theme(
    # legend.position = c(.93, .93),
    legend.justification = c("right", "top"),
    legend.box.just = "right",
    legend.margin = margin(6, 6, 6, 6)
  ) +
  scale_fill_manual(
    values = my_colours,
    limits = levels(Pow_income$Inc_bins),
    labels = c("70K-<80K", "80K-<90K", "90K-<100K"),
    name = "Median Income",
    guide = guide_legend(override.aes = list(color = NA), reverse = TRUE)
  )

myacs_map4
```



State Borders

To put an overlay of the STATE on top of your county you need to used the `get_acs` function to save the STATE geometry maps. You will use this in your maps below. This will change depending on your subsets of states. You can get state borders using the `getacs()` function in the `tidycensus` package with the options `geography = "state"` and `geometry = TRUE`. I put a variable in but I do not use it.) ## One Map

```
## Getting data from the 2015-2019 5-year ACS

## Downloading feature geometry from the Census website.  To cache shapefiles for use in

##   |                                                                              |
```



Leaflet Package for Making Interactive Maps in R

Now we are going to use the `leaflet` package to produce maps. Maps created with leaflet are interactive and allow panning and zooming. In the code chunk below I first declare a color palette that will be defined using the `colorNumeric()` function. This function itself creates a function we are calling `mypal()`, which translates data values to color values for a given color palette. Our chosen color palette in this example is the `viridis magma palette`. You can also try other palettes like for example: `viridis turbo palette`.

Setting up the map:

R comes with a default set of tiles that you can use to change the background look of your map. For example in the code chunk below I set `addProviderTiles(providers$Stamen.TonerLite)`. You should try replacing the line of code with `addProviderTiles(providers$Stamen.Watercolor)` to see the difference in the map background. Here is a list of options to choose from (https://cran.r-project.org/web/packages/leaflet.providers/readme/README.html)

```
# Creating a map with leaflet
library(leaflet)

#Create a function called mypal with a color palette I like.
mypal <- colorNumeric( #maps data values to colors in a given palette.
  palette = "magma", #this is the color palette I will use
  domain = Pow_income$estimate #the possible values of the variable that will be mapped
)

leaflet() %>%
  addProviderTiles(providers$Stamen.TonerLite) %>% # Sets the map that you see in the b
  addPolygons(data = Pow_income, #the options specified here modify the appearance of t
              color = ~mypal(estimate))  %>% #uses the mypal function specified above t
    setView(lat = 37.5643, lng = -77.8825, zoom = 10) %>%
    addLegend(
    position = "bottomright",
    pal = mypal,
    values = Pow_income$estimate,
    title = "Median Population Income Across VA Counties"
  )

#popup
```

Now make it nicer

```
# Creating a map with leaflet
library(leaflet)

#Create a function called mypal with a color palette I like.
mypal <- colorNumeric( #maps data values to colors in a given palette.
  palette = "viridis", #this is the other color palette I will use
  domain = Pow_income$estimate #the possible values of the variable that will be mapped
)

m <- leaflet() %>%
  addProviderTiles(providers$Stamen.TonerLite) %>% # Sets the map that you see in the b
  addPolygons(data = Pow_income, #the options specified here modify the appearance of t
              color = ~mypal(estimate), #uses the mypal function specified above to map
              weight = 0.3, #weight of lines on map
              smoothFactor = 0.2, #smooths the polygon shape, ranges between 0 and 1
              fillOpacity = 0.5, #changes the opacity of the color in the polygon, 0 an
              label = ~paste("Median Income: $",estimate)) %>% #label that shows up whe
  addLegend(
    position = "bottomright",
    pal = mypal,
    values = Pow_income$estimate,
    title = "Median Population Income Across VA Counties"
  )

library(htmlwidgets)
```

```r
saveWidget(m, file="m.html")

m
```

Shiny App with three things with best website to help: https://shiny.posit.co/r/getstarted/shiny-basics/lesson2/

#Put in a slider then you are done.

```r
library(shiny)

ui <- fluidPage(

  titlePanel("A Study Of Powhatan County"),
  br(),

  #leaflet map here
  h4("An Interactive Leaflet Map of Powhatan County"),
  p("p creates a paragraph of text."),
  leafletOutput("mymap"),
  mainPanel(
    p("p creates a paragraph of text."),
    p("Another paragraph here"),
    br(),

    # Bar Chart
    h2("A bar plot of race in Powhatan County"),
    plotOutput("RaceBar"),
    p("Description of the race characteristics by tract."),

  )


)

server <- function(input, output, session) {

  output$mymap <- renderLeaflet({
    leaflet() %>%
      addProviderTiles(providers$Stamen.TonerLite) %>% # background map
      addPolygons(data = Pow_income, #options modify the appearance of polygons
                  color = ~mypal(estimate), #function maps values to colors
                  weight = 0.3, #weight of lines on map
                  smoothFactor = 0.2, #smooths the polygon shape, 0 and 1
                  fillOpacity = 0.5, #opacity of the color in polygon, 0 and 1
                  label = ~paste("Median Income: $",estimate)) %>% #labels
      addLegend(
        position = "bottomright",
        pal = mypal,
        values = Pow_income$estimate,
        title = "Median Population Income Across VA Counties"
      )
  })

  output$RaceBar <- renderPlot({
    #race barplot here
    ggplot(data=long_DF, aes(x=GEOID, y=Percent, fill=Race)) +
      geom_bar(stat="identity")


  })

}
shinyApp(ui, server)
```

Make shiny look nice – any other popups?

At this point I would like to export my data to my hard drive. This is helpful if you plan to use it again. Here is how you save the data as an R dataframe. I challenge you to figure out (by Googling) how to save and export your file in excel or other formats.

```
#save(VA_income,file="ACS5.2019.Rda")
```

In another R script you can load the data that you saved above

```
#load("ACS5.2019.Rda")
```