Project 1

Paula McCree-Bailey

2022-09-14

Contents

Data Processing

First Steps

Our project focuses on manipulating data with the R programming language. It is completed in 4 phases :

```
1.  Data Processing,
2.  Writing a Generic Function for Data Processing,
3.  Writing a Generic Function for Summarizing,and
4.  Put it Together
```

The csv data set is from the U.S. Census Bureau. It contains the public school enrollment from 1986 to 2009 which is provided by county, state, and the entire country.

First step, we read in the data set with `read_csv()` from https://www4.stat.ncsu.edu/~online/datasets/EDU01a.csv

```
library(readr)
library(tidyverse)
library(tidyr)

sheet1 <- read_csv("https://www4.stat.ncsu.edu/~online/datasets/EDU01a.csv")
```

```
head(sheet1, n=5)
```

```
## # A tibble: 5 x 42
##   Area_n~1 STCOU EDU01~2 EDU01~3 EDU01~4 EDU01~5 EDU01~6 EDU01~7 EDU01~8 EDU01~9
##   <chr>    <chr>   <dbl>   <dbl> <chr>   <chr>     <dbl>   <dbl> <chr>   <chr>
## 1 UNITED ~ 00000       0  4.00e7 0000    0000          0  4.00e7 0000    0000
## 2 ALABAMA  01000       0  7.34e5 0000    0000          0  7.28e5 0000    0000
## 3 Autauga~ 01001       0  6.83e3 0000    0000          0  6.9 e3 0000    0000
## 4 Baldwin~ 01003       0  1.64e4 0000    0000          0  1.65e4 0000    0000
## 5 Barbour~ 01005       0  5.07e3 0000    0000          0  5.10e3 0000    0000
## # ... with 32 more variables: EDU010189F <dbl>, EDU010189D <dbl>,
## #   EDU010189N1 <chr>, EDU010189N2 <chr>, EDU010190F <dbl>, EDU010190D <dbl>,
## #   EDU010190N1 <chr>, EDU010190N2 <chr>, EDU010191F <dbl>, EDU010191D <dbl>,
## #   EDU010191N1 <chr>, EDU010191N2 <chr>, EDU010192F <dbl>, EDU010192D <dbl>,
## #   EDU010192N1 <chr>, EDU010192N2 <chr>, EDU010193F <dbl>, EDU010193D <dbl>,
## #   EDU010193N1 <chr>, EDU010193N2 <chr>, EDU010194F <dbl>, EDU010194D <dbl>,
## #   EDU010194N1 <chr>, EDU010194N2 <chr>, EDU010195F <dbl>, ...
```

In order to protect the original data, we create a new data set called sheet2 which is used to complete the project. In addition to protecting the original data set, if we need to start over we can rerun this step rather than downloading the original data again. We use `select()` function to select the following columns: Area_name, STCOU, and any columns that ended in "D". We use `rename()` to rename the variable Area_name to "area_name". We use `end_with("D")` to parse off all columns that end in D.

```
sheet2 <- sheet1 %>%
          select(Area_name, STCOU, ends_with('D')) %>%
          rename('area_name' = Area_name)
```

We only need 12 of the 42 columns, but retained all row.

```
head(sheet2, n=5)
```

```
## # A tibble: 5 x 12
##   area_n~1 STCOU EDU01~2 EDU01~3 EDU01~4 EDU01~5 EDU01~6 EDU01~7 EDU01~8 EDU01~9
##   <chr>    <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 UNITED ~ 00000  4.00e7  4.00e7  4.03e7  4.07e7  4.14e7  4.21e7  4.27e7  4.34e7
## 2 ALABAMA  01000  7.34e5  7.28e5  7.30e5  7.28e5  7.26e5  7.26e5  7.28e5  7.31e5
## 3 Autauga~ 01001  6.83e3  6.9 e3  6.92e3  6.85e3  7.01e3  7.14e3  7.15e3  7.38e3
## 4 Baldwin~ 01003  1.64e4  1.65e4  1.68e4  1.71e4  1.75e4  1.80e4  1.87e4  1.94e4
## 5 Barbour~ 01005  5.07e3  5.10e3  5.07e3  5.16e3  5.17e3  5.25e3  5.14e3  5.11e3
## # ... with 2 more variables: EDU010195D <dbl>, EDU010196D <dbl>, and
## #   abbreviated variable names 1: area_name, 2: EDU010187D, 3: EDU010188D,
## #   4: EDU010189D, 5: EDU010190D, 6: EDU010191D, 7: EDU010192D, 8: EDU010193D,
## #   9: EDU010194D
```

Step 2. Our data is currently in a wide format with 12 rows and 3198 columns. Wide format is where a subject's information is repeated in a single row. To work with this data effectively, we need to transform it into a long format. Long format is where each row is one point per subject. In this case, each subject (state or county) has data in multiple rows. We use `pivot_long()` and piping `%>%` to convert the data set to long format by leaving columns area_name and STCOU as is, but we want the columns 3:12 to pivot with a new column name of "enrollment" and the data named "Values".

```
sheet2 <- sheet2 %>% pivot_longer(cols = 3:12,
            names_to = 'enrollment',
            values_to = 'value')
```

After the pivot (transformation) from wide to long format, we can see each observation (location) has it's columns(information) listed on the same row. When data is in this type of format it is mush easier to complete analysis on it.

```
head(sheet2, n=5)
```

```
## # A tibble: 5 x 4
##   area_name     STCOU enrollment    value
##   <chr>         <chr> <chr>         <dbl>
## 1 UNITED STATES 00000 EDU010187D 40024299
## 2 UNITED STATES 00000 EDU010188D 39967624
## 3 UNITED STATES 00000 EDU010189D 40317775
## 4 UNITED STATES 00000 EDU010190D 40737600
## 5 UNITED STATES 00000 EDU010191D 41385442
```

Step 3. Within the enrollment variable contains important information. The first three characters represent the survey with the next four representing the type of value you have from that survey. The last two digits prior to the "D" represent the year of the measurement. For example, first observation, we have the following:

- EDU survey
- 40024229 value
- 0101 type of value
- 87 year of survey

We use `mutate()`, `as.numeric()`, `substr()`, `format()`, and `%>%` to separate the enrollment variable into the additional variables we need. `mutate()` is used to create a new column within the data. `as.numeric` as is sounds converts a character type into a numeric type. `substr()` allows us to extract or replace sections of a vector. `format()` encodes objects into a common format. `%>%` also called chaining provides effective script writing and ability to chain functions together.

We first need to parse off the characters for the year and measurement and create separate columns. The year value temporarily called yearT is a character and we need to convert it into a numeric date. This is achieved by telling `as.Date()` to tell R we have a character value with the structure of YY. This is denoted by %y. Then with the use of `format()`, we tell R change format to YYYY. Finally, `as.numeric` converts it from a character to numeric value. Below we have an example using character 88. We know 88 is a character because of ' ' single quotes.

```
as.numeric(format(as.Date('88', '%y'),'%Y'))
```

```
## [1] 1988
```

```
sheet2 <-sheet2 %>% mutate(yearT = (substr(sheet2$enrollment, 8,9)),
                    measure = substr(sheet2$enrollment,1,7))

sheet2 <- sheet2 %>% mutate(year = as.numeric(format(as.Date(yearT, '%y'),'%Y')))
```

Step 4. We want to create two data sets one for county and another for state (non-county) data; however, our data provided does not provide a breakdown. We just have our 7 columns of data and metadata, which is embedded. We know all county measurements have the format "County Name, DD" where "DD" represents the state. We can use this information, `grepl()` and a pattern to subset the data. `grepl(pattern = ", \\w\\w", Area_name)`. \w\w represents the DD. grepl() searches for matches of a string and returns TRUE or FALSE. Rather than allowing it to use the default True or FALSE, we substituted "county" and "state" into the function.

`mutate(class = ifelse(grepl(pattern = ", \\w\\w", sheet2$area_name),"county", "state"))`

We use `ifelse` to help check for the pattern in the area_name column. If grepl() is true, we found a pattern. So, we create a column named "class" and write "county in it's column, otherwise it writes"state".

```
sheet2 <- sheet2%>% mutate(class = ifelse(grepl(pattern = ", \\w\\w", sheet2$area_name),'county', 'state
```

```
head(sheet2, n=5)
```

```
## # A tibble: 5 x 8
##   area_name     STCOU enrollment    value yearT measure   year class
##   <chr>         <chr> <chr>         <dbl> <chr> <chr>    <dbl> <chr>
## 1 UNITED STATES 00000 EDU010187D 40024299 87    EDU0101   1987 state
## 2 UNITED STATES 00000 EDU010188D 39967624 88    EDU0101   1988 state
## 3 UNITED STATES 00000 EDU010189D 40317775 89    EDU0101   1989 state
## 4 UNITED STATES 00000 EDU010190D 40737600 90    EDU0101   1990 state
## 5 UNITED STATES 00000 EDU010191D 41385442 91    EDU0101   1991 state
```

We use `subset()` to divide the sheet2 data into county and state data sets. The function uses our new class column to parse the data.

```
state <- subset(sheet2, sheet2$class == 'state',
                select = c('area_name','STCOU','enrollment', 'value','class', 'year', 'measure'))

county <- subset(sheet2, sheet2$class == 'county',
                select = c('area_name','STCOU','enrollment', 'value','class', 'year', 'measure'))
```

For both state and county data, we need to relate the data to a class. This allows us to create a reusable plotting function that only needs to know which class argument to perform. `class()` is a blueprint for a object by setting the class an object inherits from.

```
class(county) <- c('county', class(county))
class(state)  <- c('state', class(state))
```

Step 5. We use `mutate()` to add a new variable to our data sets. For the county level, we create state which corresponds to the state that the county resides in. `mutate(division = substr(county$area_name,(nchar(county$area_r` In order to complete step, we need to know the length of each area_name. nchar is a good choice. `nchar()` provides the length of a string. `substr( )` takes each area_name and uses it's length and length -1 to parse off the required information which is saved in division.

```
county <- county %>%
  mutate(division = substr(county$area_name,(nchar(county$area_name)-1),nchar(county$area_name)))
```

Step 6. For the state (non-county) level, we create a new variable named division which corresponds to the state division classification based on the information here. https://en.wikipedia.org/wiki/List_of_regions_ of__the__United__States If are_name corresponds to UNITED STATES, we return ERROR for the division. We use `mutate()`, `if_else()`, and `%in%` to create and update the division column. We need a list of states that correspond to a specific division. `C()` is a helpful function. It allow us to create a character vector containing states.

`mutate(division = if_else(area_name %in% c("CONNECTICUT", "MAINE", "MASSACHUSETTS",....,"1",`
If area_name is contained in `C()`, we will place a character value in that field. Since we need to take into consideration the ERROR message for the United State, it is necessary to make this a character (char) field.

```
state<- state %>%
mutate(division = if_else(area_name %in% c('CONNECTICUT', 'MAINE', 'MASSACHUSETTS',
                                           'NEW HAMPSHIRE', 'RHODE ISLAND', 'VERMONT'),'1',
                    if_else(area_name %in% c('NEW JERSEY', 'NEW YORK','PENNSYLVANIA'),'2',
                       if_else(area_name %in% c('ILLINOIS', 'INDIANA', 'MICHIGAN', 'OHIO',
                                           'WISCONSIN'),'3',
                        if_else(area_name %in% c('IOWA', 'KANSAS', 'MINNESOTA', 'MISSOURI',
                                           'NEBRASKA', 'NORTH DAKOTA','SOUTH DAKOTA'),'4',
                           if_else(area_name %in% c('DELAWARE', 'FLORIDA', 'GEORGIA', 'MARYLAND',
                                           'NORTH CAROLINA', 'SOUTH CAROLINA', 'VIRGINIA',
                                           'WASHINGTON, D.C.','WEST VIRGINIA',
                                           'Washington, D.C.', 'DISTRICT OF COLUMBIA',
                                           'District of Columbia'),'5',
                              if_else(area_name %in% c('ALABAMA', 'KENTUCKY', 'MISSISSIPPI',
                                             'TENNESSEE'),'6',
                                 if_else(area_name %in% c('ARKANSAS', 'LOUISIANA', 'OKLAHOMA',
                                           'TEXAS'),'7',
                                    if_else(area_name %in% c('ARIZONA', 'COLORADO', 'IDAHO',
                                           'MONTANA', 'NEVADA', 'NEW MEXICO', 'UTAH', 'WYOMING')
                                           ,'8',
                                      if_else(area_name %in% c('ALASKA', 'CALIFORNIA', 'HAWAII',
                                           'OREGON', 'WASHINGTON'),'9', 'ERROR'))))))))))
```

Writing a Generic Function for Data Processing

Now that we have figured out the basic information for the project, we are repeating the process above on `EDU01b.csv` data set and create user defined functions for each of the steps above. Why user defined functions? These functions allow us to be more efficient. We can call our functions rather than copying and pasting section we need to reference. When we are writing functions, R will return the output automatically for us; however, it is best practice to always include return at the end of the function.

We read in our new data set `EDU01b.csv` and save it to another variable.

```
EDU01b <- read_csv("https://www4.stat.ncsu.edu/~online/datasets/EDU01b.csv")
```

We combine steps 1 and 2 to create `translong()` function. This function takes any data sets that we call using `transLong(DF)` to select columns - area_name, STCOU, and any columns ending in D and transforms the data from wide to long. The function also has an optional argument named colName to allow the user to specify the name of the column representing the value. We can call the function as either `transLong(DF)` or `transLong(DF, "Your_Name")`

```
transLong <- function(DF, colName = 'Public_School'){
  DF <- DF %>%
            select(Area_name, STCOU, ends_with('D')) %>%
            rename('area_name' = Area_name) %>%
            pivot_longer(cols = 3:12,
            names_to = 'enrollment',
            values_to = 'value')

  invisible(DF)
  return(DF)
}
```

We create `parseMe()` function from step 3. This function takes any data sets that we call using `parseMe(DF)` to create class, measure, and year columns.

```
parseMe <- function(DF){
    DF <- DF %>% mutate(class = ifelse(grepl(pattern = ', \\w\\w',
                DF$area_name),'state' ,'county'))

    DF <-DF %>% mutate(yearT = (substr(DF$enrollment, 8,9)),
                    measure = substr(DF$enrollment,1,7))

    DF <- DF %>% mutate(year = as.numeric(format(as.Date(yearT, '%y'),'%Y')))

invisible(DF)
return(DF)

}
```

We create `divisionCounty()` function from step 5. This function takes any county level data sets that we call using `divisionCounty(DF)` to create division column by subsetting the state's initials from the area_name.

```
divisionCounty <- function(DF){
  DF <- DF %>%
  mutate(division = substr(DF$area_name,(nchar(DF$area_name)-2),
  nchar(DF$area_name)))

  return(DF)
}
```

We create `divisionState()` function from step 6. This function takes any state level data sets that we call using `divisionState(DF)` to create division column by determining its region of the country.

```
divisionState <- function(DF){
DF <- DF %>%
mutate(division = if_else(area_name %in% c('CONNECTICUT', 'MAINE', 'MASSACHUSETTS',
                                            'NEW HAMPSHIRE', 'RHODE ISLAND', 'VERMONT'),'1',
                    if_else(area_name %in% c('NEW JERSEY', 'NEW YORK','PENNSYLVANIA'),'2',
                       if_else(area_name %in% c('ILLINOIS', 'INDIANA', 'MICHIGAN', 'OHIO',
                                            'WISCONSIN'),'3',
                        if_else(area_name %in% c('IOWA', 'KANSAS', 'MINNESOTA', 'MISSOURI',
                                            'NEBRASKA', 'NORTH DAKOTA','SOUTH DAKOTA'),'4',
                           if_else(area_name %in% c('DELAWARE', 'FLORIDA', 'GEORGIA', 'MARYLAND',
                                            'NORTH CAROLINA', 'SOUTH CAROLINA', 'VIRGINIA',
                                            'WASHINGTON, D.C.','WEST VIRGINIA',
                                            'Washington, D.C.', 'DISTRICT OF COLUMBIA',
                                            'District of Columbia'),'5',
                              if_else(area_name %in% c('ALABAMA', 'KENTUCKY', 'MISSISSIPPI',
                                              'TENNESSEE'),'6',
                                 if_else(area_name %in% c('ARKANSAS', 'LOUISIANA', 'OKLAHOMA',
                                            'TEXAS'),'7',
                                    if_else(area_name %in% c('ARIZONA', 'COLORADO', 'IDAHO',
                                            'MONTANA', 'NEVADA', 'NEW MEXICO', 'UTAH', 'WYOMING')
                                            ,'8',
                                      if_else(area_name %in% c('ALASKA', 'CALIFORNIA', 'HAWAII',
                                            'OREGON', 'WASHINGTON'),'9', 'ERROR'))))))))))

return(DF)
}
```

This function completes step 4 by creating our two data sets - county and state. First, We pass the result (DF) from `parseMe()` to `createDFs()` function. Afterwards, we call `divisionState(DF)` and `divisionCounty()`to complete the division columns separately for both data sets. Normally throughout this process, we are returning one DF. This function is returning two DFs (county and state) in the form of a list. `return(list(state, county, DF))` We can see later how to view DFs in a list.

```
createDFs <- function(DF){

  T1 <- subset(DF, DF$class == 'state',
            select = c('area_name','STCOU','enrollment',  'value','class', 'year', 'measure'))

  T2 <- subset(DF, DF$class == 'county',
            select = c('area_name','STCOU','enrollment', 'value','class', 'year', 'measure'))

  class(county) <- c('county', class(county))
  class(state)  <- c('state', class(state))

  county_sheet <- divisionCounty(T2)
  state_sheet <- divisionState(T1)

  invisible(list(state, county, DF))

  return(list(state, county, DF))

}
```

Now, we are at the final step of this process. We create the final subroutine or wrapper function where we call all four functions to complete our task of data processing. Keep in mind, as long as we are using a data set with the same format, we can call this function `my_wrapper()`. We only need to call the function and include the DF (dataframe or dataset). Note this function has three ellipses . . . Recall when we call `transLong(DF, colName ="Public_School")` we have the option of providing a name for the enrollment value.

```
my_wrapper <- function(url, ...){
  a <- read_csv(url, ...)
  b <- transLong(a, ...)
  c <- parseMe(b, ...)
  d <- createDFs(c, ...)

  invisible()

  return(list(firstelement = state, secondelement =county))
}
```

Call It and Combine It

Finally, we have created our functions. As a check, we can pass our csv files - EDU01a and EDU01b into our new function `my_wrapper()`. Since we assigned the output to a variable, we can not see the data without requesting for it.

```
EDU01a <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/EDU01a.csv")
```

Let's look at EDU01b, since we looked at EDU01a previously.

```
EDU01b <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/EDU01b.csv")
```

This one is the state level data for EDU01b and

```
EDU01a['firstelement']
```

```
## $firstelement
## # A tibble: 530 x 8
##    area_name     STCOU enrollment    value class  year measure division
##    <chr>         <chr> <chr>         <dbl> <chr> <dbl> <chr>   <chr>
##  1 UNITED STATES 00000 EDU010187D 40024299 state  1987 EDU0101 ERROR
##  2 UNITED STATES 00000 EDU010188D 39967624 state  1988 EDU0101 ERROR
##  3 UNITED STATES 00000 EDU010189D 40317775 state  1989 EDU0101 ERROR
##  4 UNITED STATES 00000 EDU010190D 40737600 state  1990 EDU0101 ERROR
##  5 UNITED STATES 00000 EDU010191D 41385442 state  1991 EDU0101 ERROR
##  6 UNITED STATES 00000 EDU010192D 42088151 state  1992 EDU0101 ERROR
##  7 UNITED STATES 00000 EDU010193D 42724710 state  1993 EDU0101 ERROR
##  8 UNITED STATES 00000 EDU010194D 43369917 state  1994 EDU0101 ERROR
##  9 UNITED STATES 00000 EDU010195D 43993459 state  1995 EDU0101 ERROR
## 10 UNITED STATES 00000 EDU010196D 44715737 state  1996 EDU0101 ERROR
## # ... with 520 more rows
```

This is the county level data for EDU01b. Recall, when the wrapper returned the two data set it's in a form of a list. We need to use [] to drill down into the list.

```
EDU01a['secondelement']
```

```
## $secondelement
## # A tibble: 31,450 x 8
##    area_name   STCOU enrollment value class   year measure division
##    <chr>       <chr> <chr>      <dbl> <chr>  <dbl> <chr>   <chr>
##  1 Autauga, AL 01001 EDU010187D  6829 county  1987 EDU0101 AL
##  2 Autauga, AL 01001 EDU010188D  6900 county  1988 EDU0101 AL
##  3 Autauga, AL 01001 EDU010189D  6920 county  1989 EDU0101 AL
##  4 Autauga, AL 01001 EDU010190D  6847 county  1990 EDU0101 AL
##  5 Autauga, AL 01001 EDU010191D  7008 county  1991 EDU0101 AL
##  6 Autauga, AL 01001 EDU010192D  7137 county  1992 EDU0101 AL
##  7 Autauga, AL 01001 EDU010193D  7152 county  1993 EDU0101 AL
##  8 Autauga, AL 01001 EDU010194D  7381 county  1994 EDU0101 AL
##  9 Autauga, AL 01001 EDU010195D  7568 county  1995 EDU0101 AL
## 10 Autauga, AL 01001 EDU010196D  7834 county  1996 EDU0101 AL
## # ... with 31,440 more rows
```

We can use a small function that takes in the results from `my_wrapper()` to combine two data sets into one. This function is`dplyr::bind_rows`. Below we combine county level data sets twice and the non-county level twice to create two larger data sets.

```
bindState  <- dplyr::bind_rows(EDU01a['firstelement'], EDU01b['firstelement'])

bindCounty <- dplyr::bind_rows(EDU01a['secondelement'], EDU01b['secondelement'])
```

Writing a Generic Function for Summarizing

First Steps

We use the concept of classes to determine what type of plot R creates when either state or county function is called. For this to work, we create functions using "plot.county" or "plot.state". Recall, earlier we created the following `class(state) <- c("state", class(state))` to order to create a inherited relationship between objects. We are creating a relationship between a plot and the type of data being pass to the plotting function.

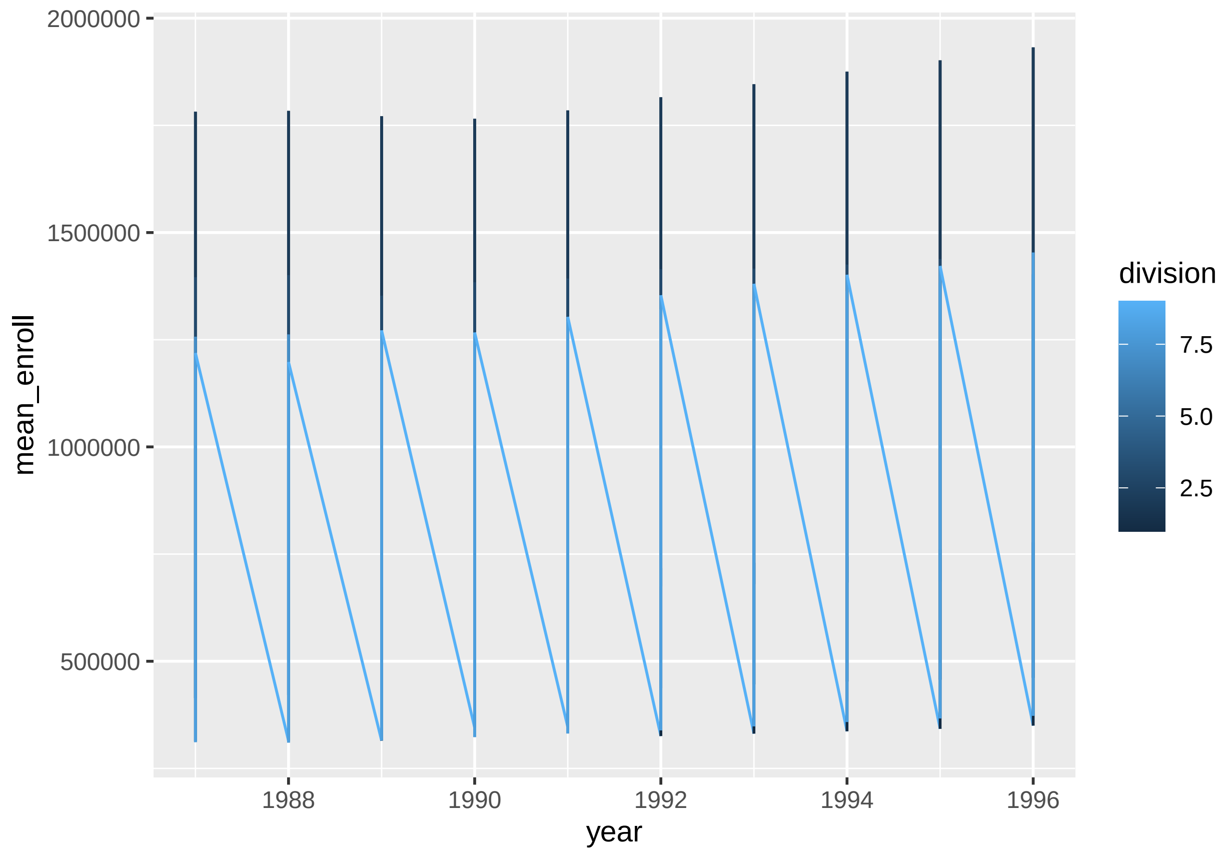
For the state plotting function `plot(DF, var_name = "value")`, it plots the mean value of the statistic (value) across the years for each Division. Recall when we created `divisionState()`, it created the division column as a character, because we needed to account for the Error message. First we need to remove Error from the data set. So, for each Division, we have mean from 1988 to 1996.

Based on the output of the state plotting function, we questioned the variation in the data between years and division. We noticed the final plot.state() graph is not smooth. It has larger variation.

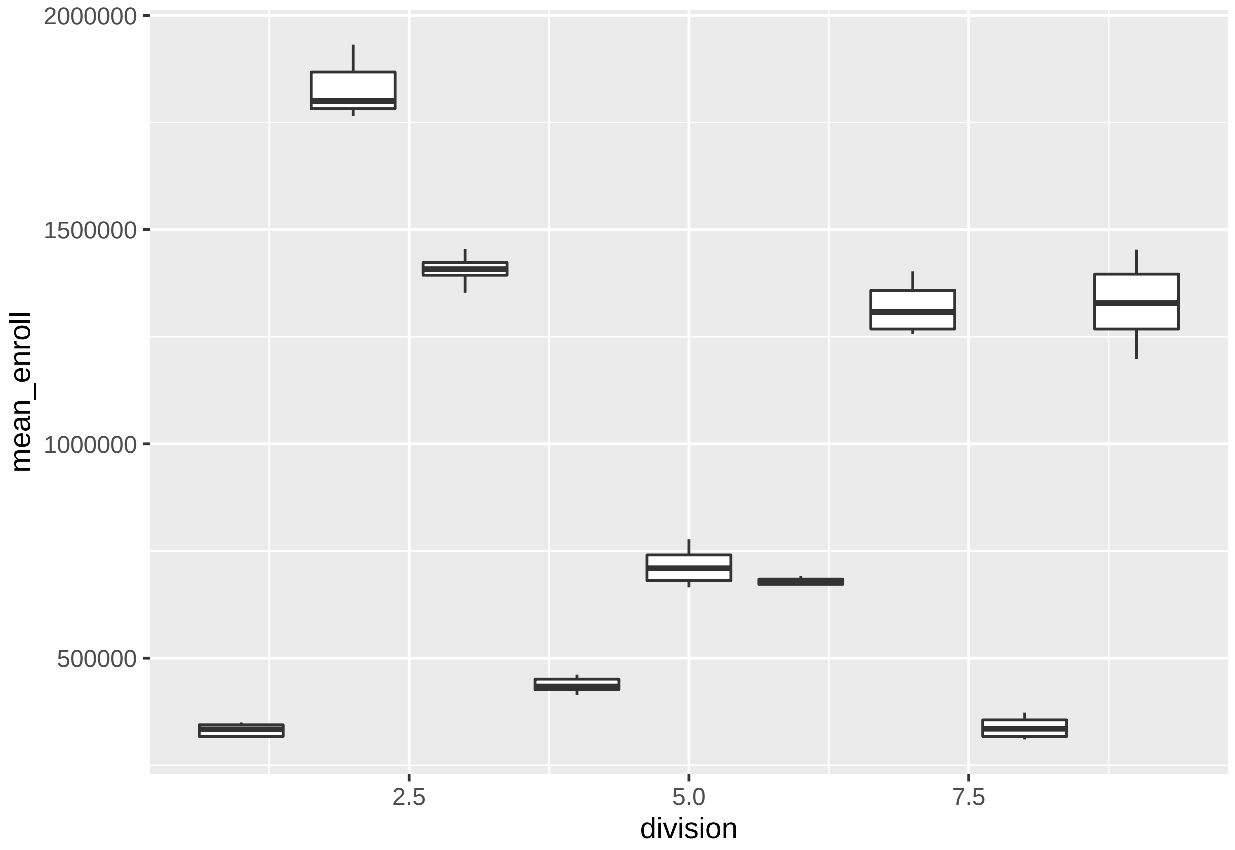
Below we create a box plot of the state data summarized by the mean. The first box plot indicates a lot of variation between the divisions between the computed means.

```
plot.state <- function(df = state, var_name = value){
  temp <- filter(state, division != 'ERROR')%>% mutate(division = as.numeric(division))

  newDF <- temp%>% select(year,area_name, value, division) %>% group_by(year, division) %>%
           summarise(mean_enroll = mean(get(var_name)))

    ggplot(newDF, aes(x = year, y = mean_enroll, color = division)) +geom_line()
}
```

```
plot(state, var_name = 'value')
```

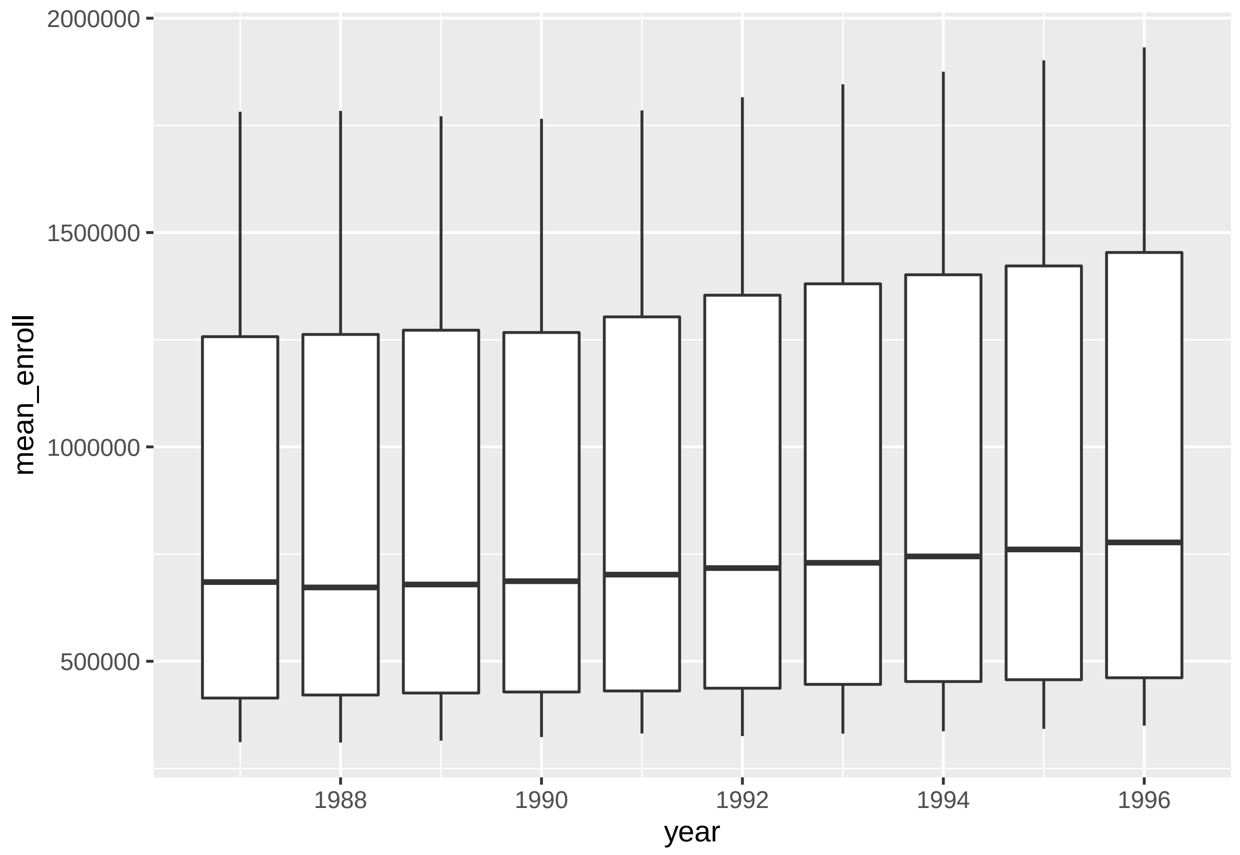


```
temp <- filter(state, division != 'ERROR')%>% mutate(division = as.numeric(division))
newDF1 <- temp%>% select(year,area_name, value, division, enrollment) %>%
          group_by(year,division) %>%
          summarise(mean_enroll = mean(get('value')))

    ggplot(newDF1, aes( x = division, y = mean_enroll, group = division)) +geom_boxplot()
```



However, when the data is grouped by year with the computed mean, there is much less variation.

```
  temp <- filter(state, division != 'ERROR')%>% mutate(division = as.numeric(division))
  newDF1 <- temp%>% select(year,area_name, value, division, enrollment) %>%
            group_by(year,division) %>%
            summarise(mean_enroll = mean(get('value')))
   ggplot(newDF1, aes( x = year, y = mean_enroll, group = year)) +geom_boxplot()
```



The county plotting function plots the mean value of the statistic (value) for each area_name. The user can specify the state of interest, determine whether the ‘top’ or ‘bottom’ most counties should be looked at, and instruct how many of the ‘top’ or ‘bottom’ counties to be presented.

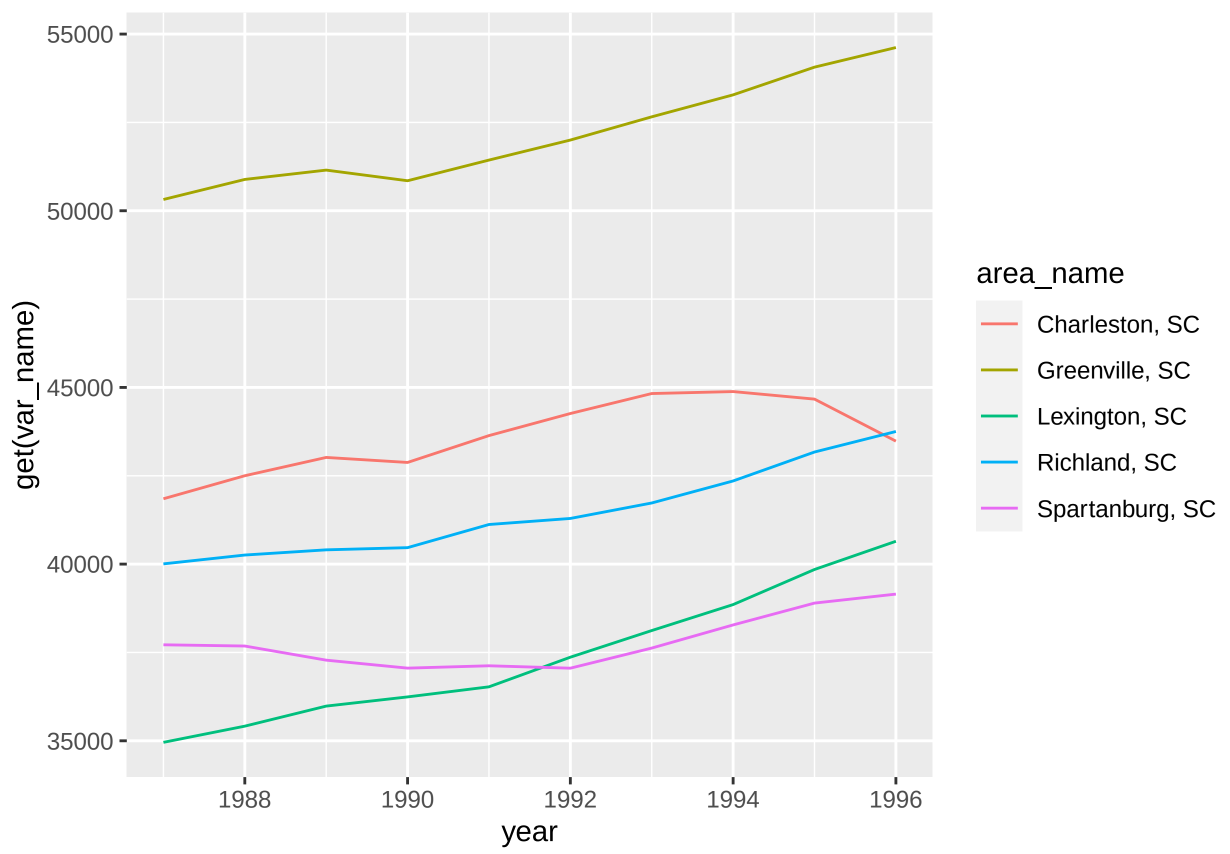
This function offers several default arguments `plot(DF, var_name = 'value', location = 'top', ST= 'SC', number = 5)` and the user has the option to change them all when calling the function. The user must provide at least the name of the data file. Any missing argument are handled by the defaults.

Based on the arguments presented, the function summarizes the mean value for the area_name. Again based on the user, it determines whether to sort the results in descending when ‘top’ is provided or ascending for another value. These results are saved in meanCounty. Lastly, we need to filter state data to only include the counties in meanCounty. This line of code helps us complete this task. `df[df$area_name %in% meanCounty$area_name,]` The `%in%` compares area_name in state and meanCounty data sets and only keeps the area_name information that’s in both data sets.

```
plot.county <- function(df, var_name = 'value',location = 'top', ST='SC', number=5){

meanTemp  <- df %>% group_by(area_name, division) %>% summarise(area_mean = mean(value))

  if(location == 'TOP' | location =='top' |location =='Top'){
    temp <- filter(meanTemp, division == ST) %>%
            arrange(desc(area_mean))
    temp2 <- temp[c(1:number),]
  }else{
    temp <- filter(meanTemp, division == ST) %>%
            arrange(area_mean)
    temp2 <- temp[c((nrow(temp)-number+1) :nrow(temp)),]
  }

meanCounty <- temp2

filtered_df <- df[df$area_name %in% meanCounty$area_name,]

ggplot(filtered_df, aes(x = year, y = get(var_name), color = area_name)) +geom_line()

}
```

plot(county) uses default arguments to complete the graph

```
plot(county)
```



Put it Together

Practice with EDU01a and EDU01b

As the title says "we are ready to put it together". We read in EDU01a and EDU01b and save the output to another object.

```
EDU01a <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/EDU01a.csv")
```

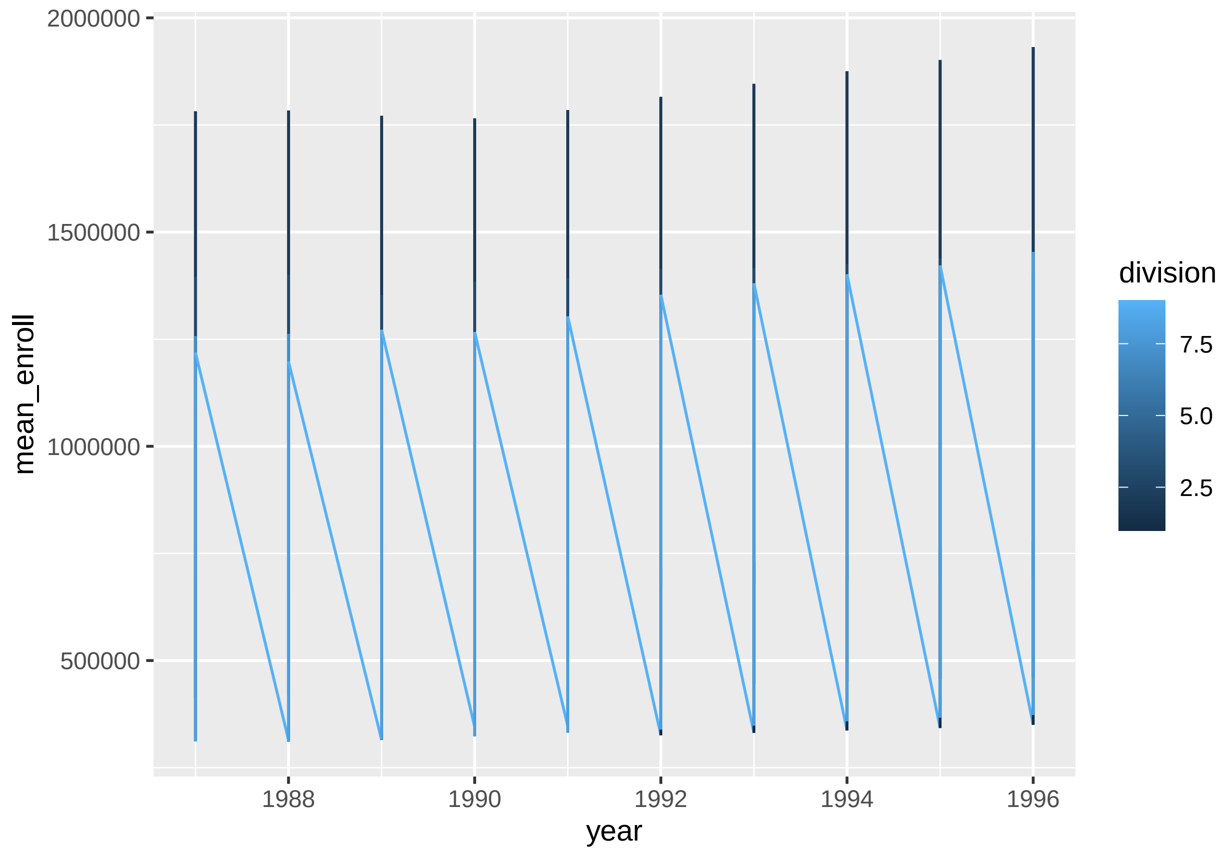
```
EDU01b <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/EDU01b.csv")
```

We run our data combining function to put these into one object (with two data frames)

```
bindState  <- dplyr::bind_rows(EDU01a['firstelement'], EDU01b['firstelement'])

bindCounty <- dplyr::bind_rows(EDU01a['secondelement'], EDU01b['secondelement'])
```

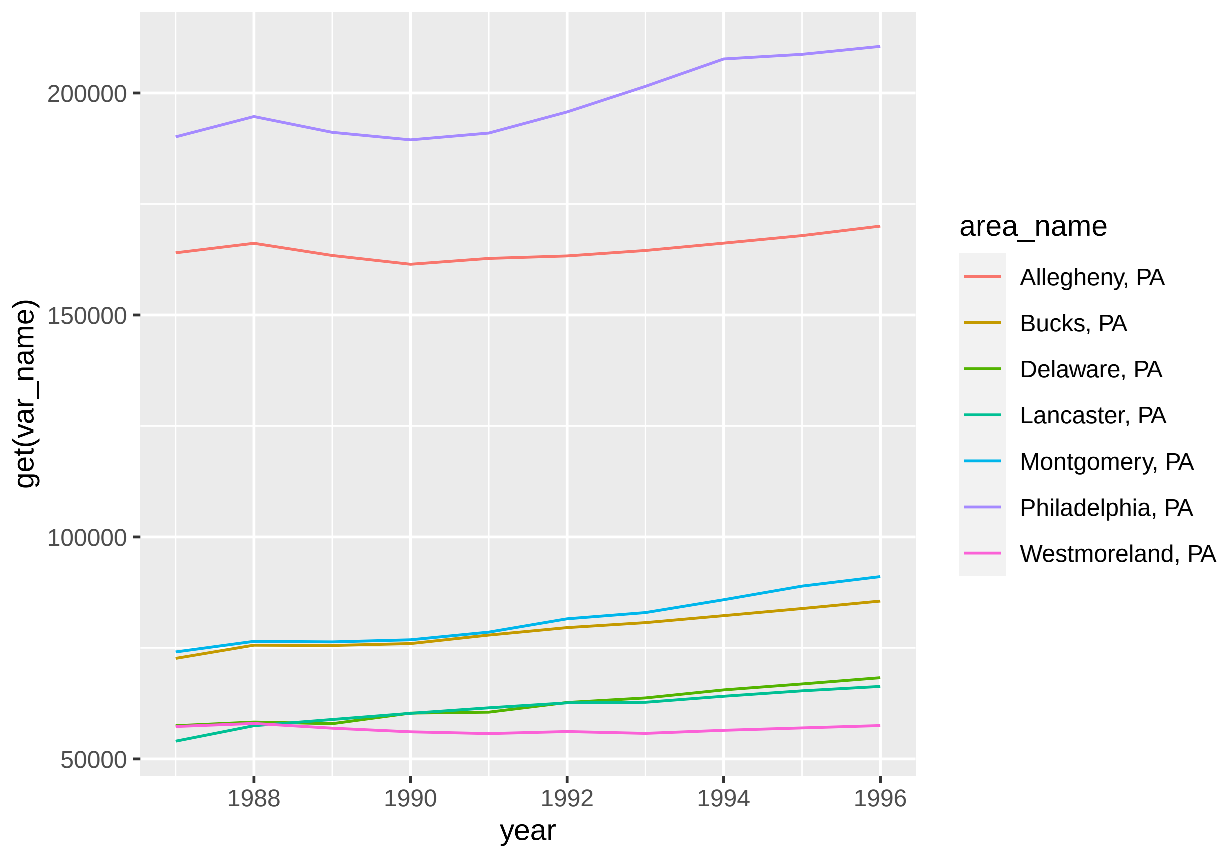
We use the plot function on the binState data frame

```
plot(bindState, var_name = 'value')
```



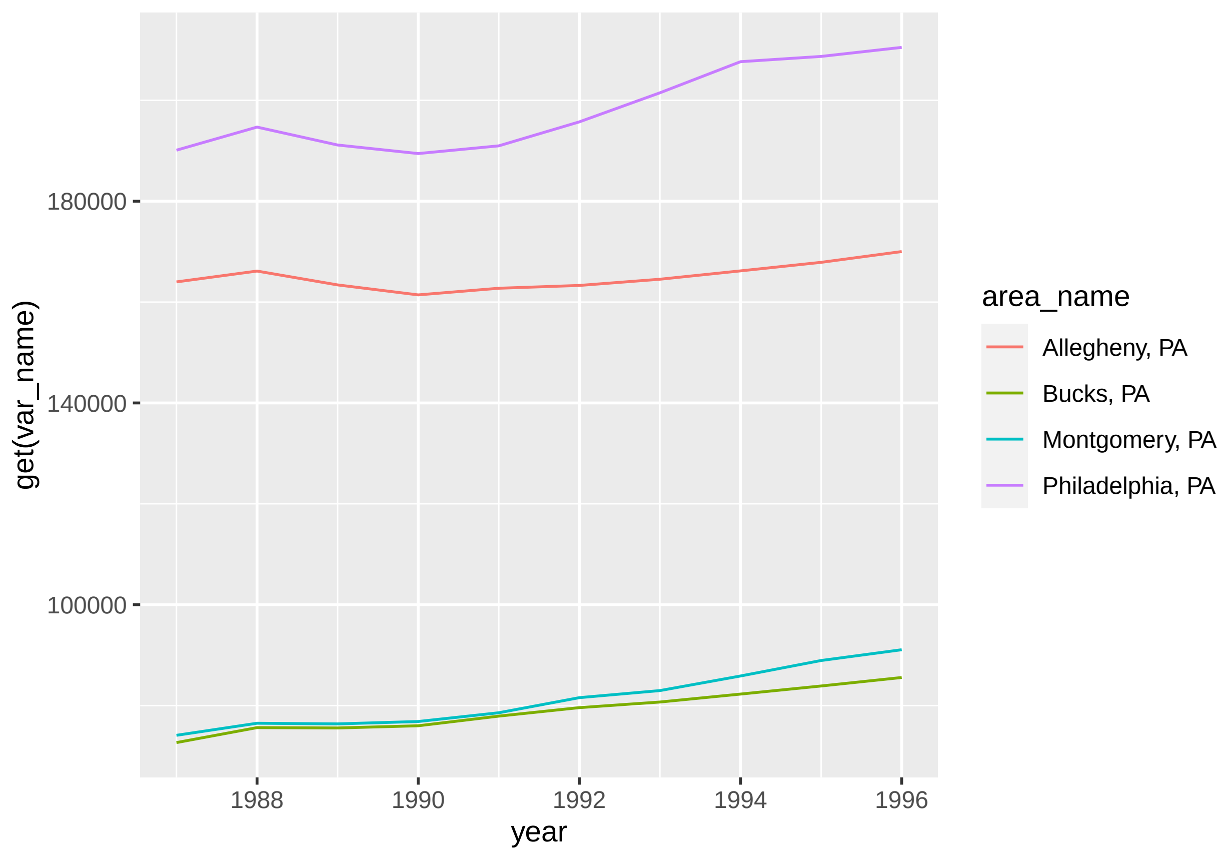
We run plot(bindCounty) specifying the state to be "PA", the group being the top, the number looked at being 7.

```
plot(bindCounty, ST = 'PA', location  = 'top', number = 7)
```



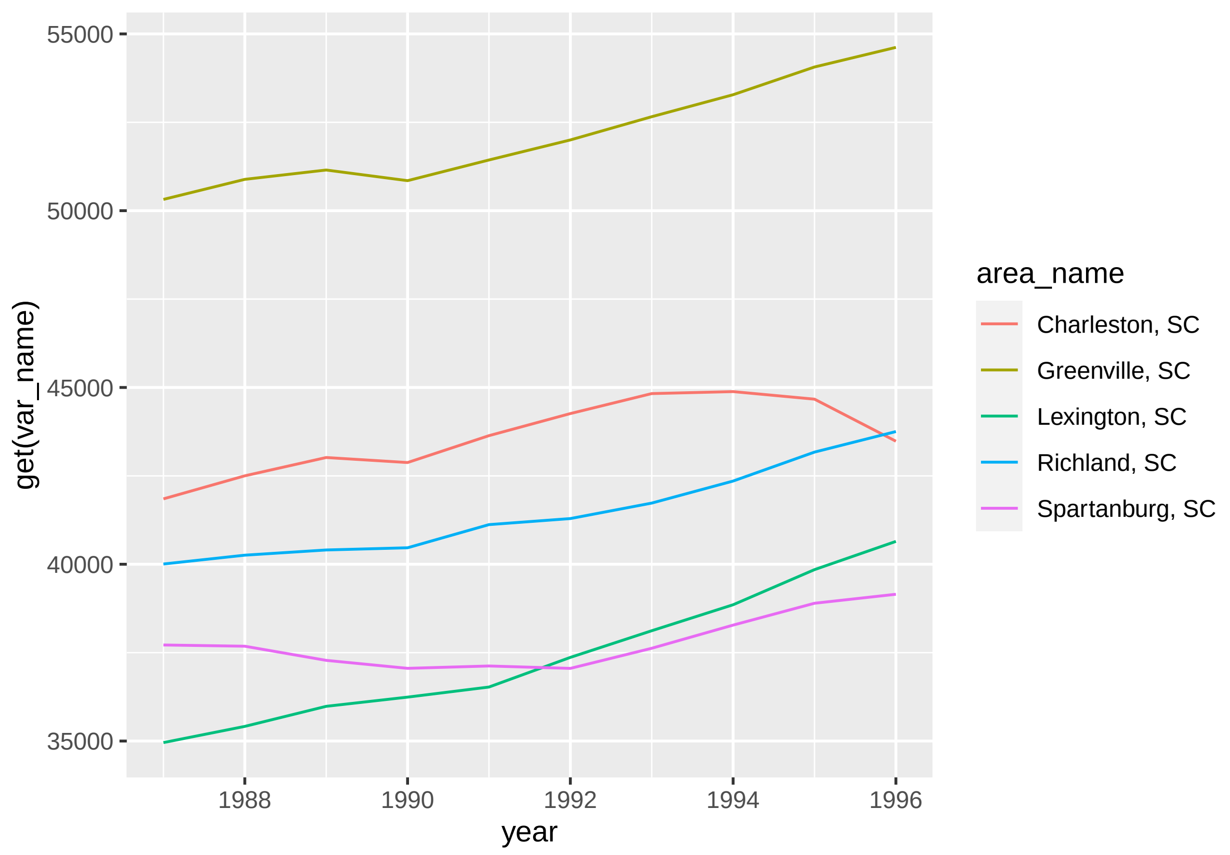
We run plot(bindCounty) specifying the state to be “PA”, the group being the bottom, the number looked at being 4.

```
plot(bindCounty, ST = 'PA', location = 'btm', number = 4)
```



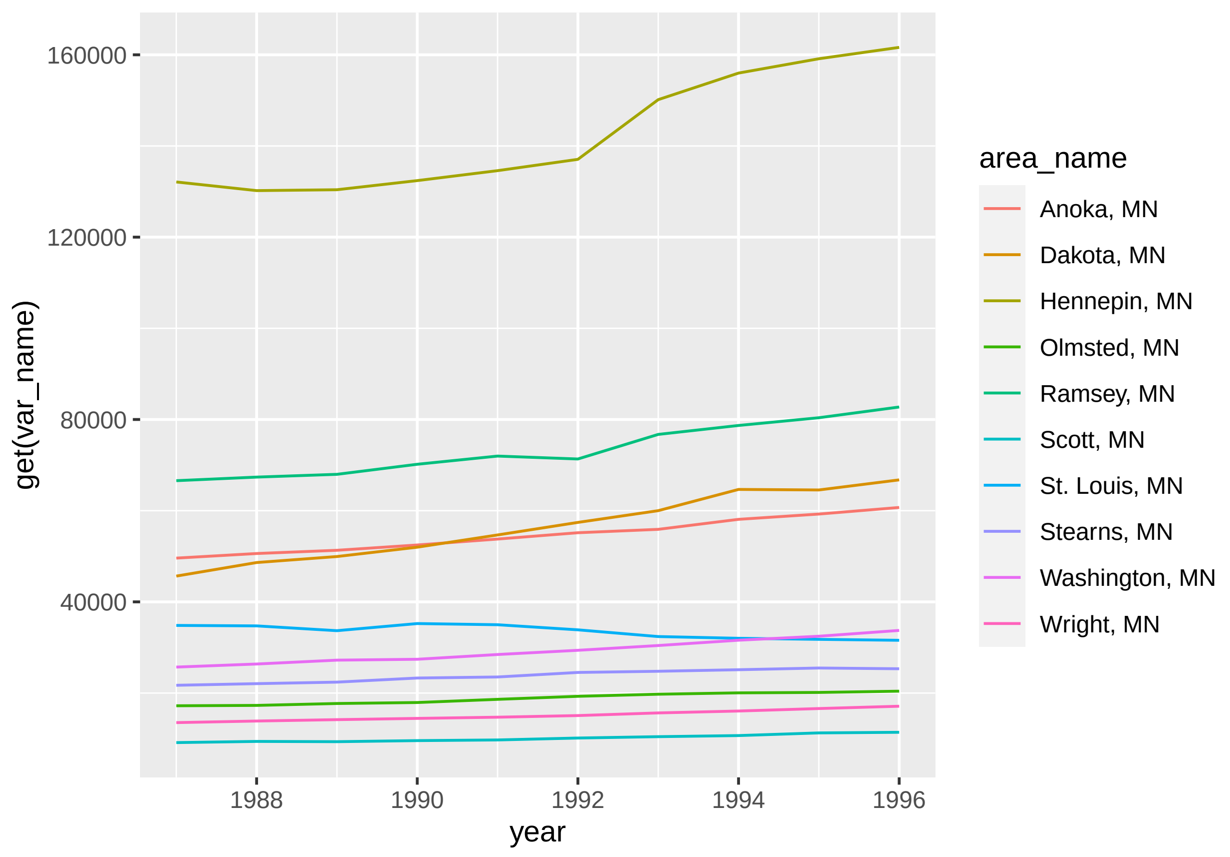
We run plot(bindCounty) without specifying anything (defaults used).

```
plot(bindCounty)
```



We run plot(bindCounty) specifying the state to be “MN”, the group being the top, the number looked at being 10.

```
plot(bindCounty, ST = 'MN', location = 'top', number = 10)
```



Read in similar data sets and apply our functions.

We put our new functions to work by running the data processing function on the four data sets. First, We read in the new data sets.

```
PST01a <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/PST01a.csv")
```

```
PST01b <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/PST01b.csv")
```

```
PST01c <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/PST01c.csv")
```

```
PST01d <- my_wrapper("https://www4.stat.ncsu.edu/~online/datasets/PST01d.csv")
```

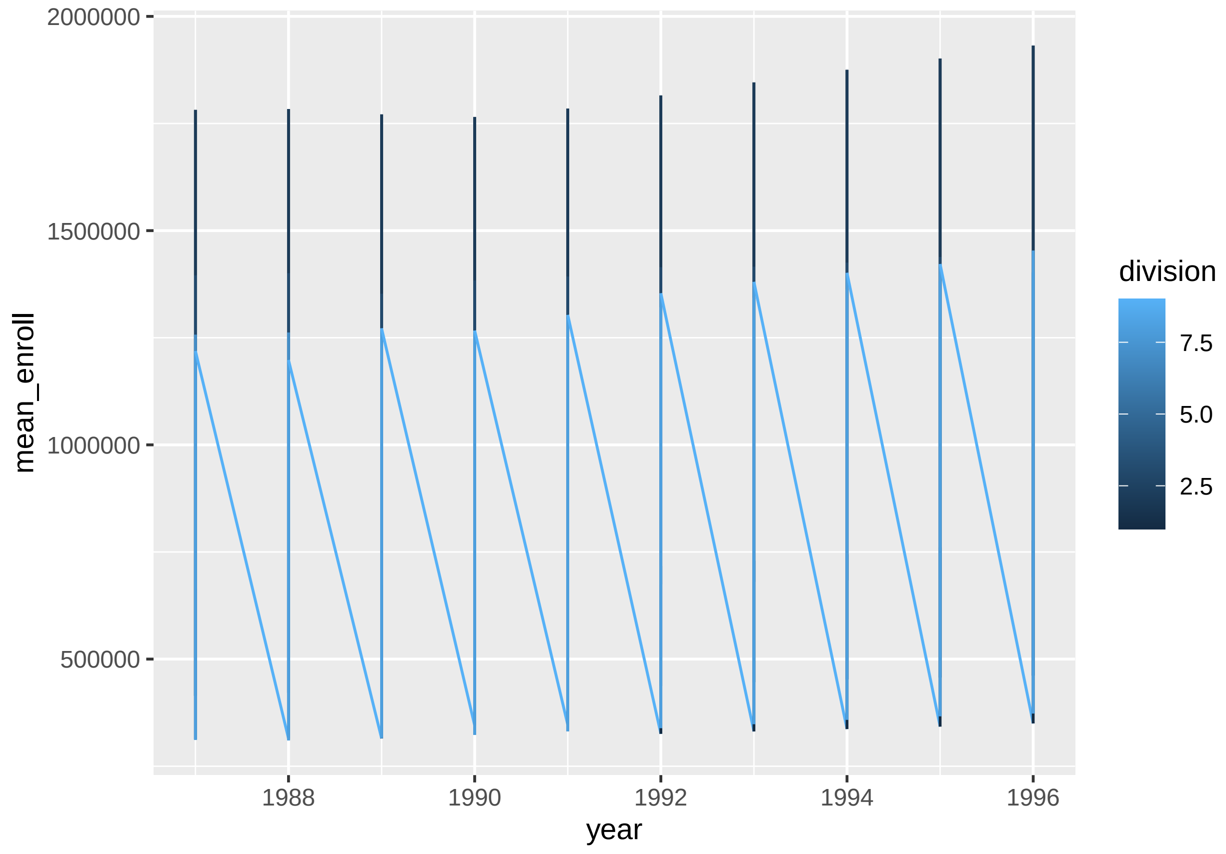
We apply the data combining function to create one object (with two dataframes - state and county). We combine PST01a and PST01b into one set, then combine PST01c and PST01d into another set. The resulting objects combine to create our grpAll_State and grpALL_County.

```
grpAB_state  <- dplyr::bind_rows(PST01a['firstelement'], PST01b['firstelement'])
grpAB_county <- dplyr::bind_rows(PST01a['secondelement'], PST01b['secondelement'])

grpCD_state  <- dplyr::bind_rows(PST01c['firstelement'], PST01d['firstelement'])
grpCD_county <- dplyr::bind_rows(PST01c['secondelement'], PST01d['secondelement'])
```

```r
grpAll_State  <- dplyr::bind_rows(grpAB_state, grpCD_state)
grpAll_county <- dplyr::bind_rows(grpAB_county, grpCD_county)
```

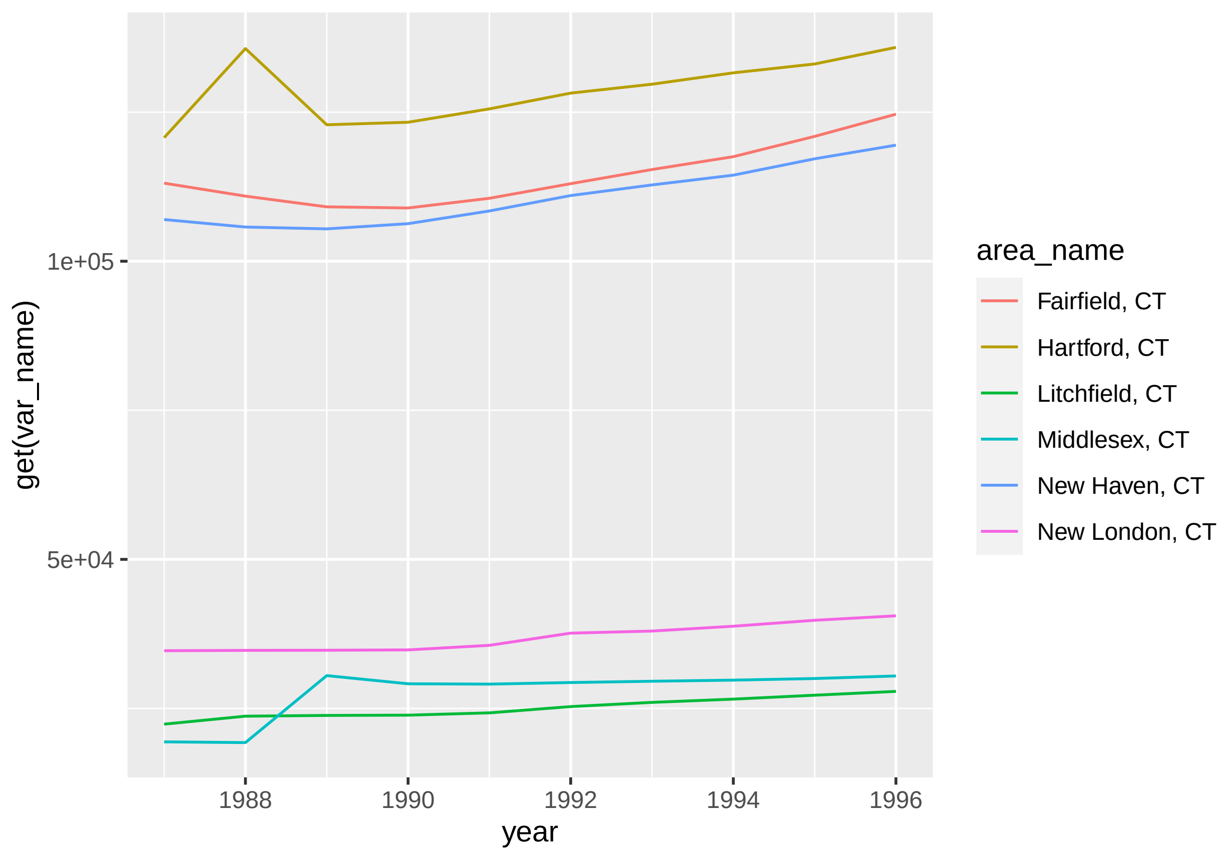
We use our plot.state function to produce a plot for grpAll_state.

```r
plot(grpAll_State, var_name = 'value')
```



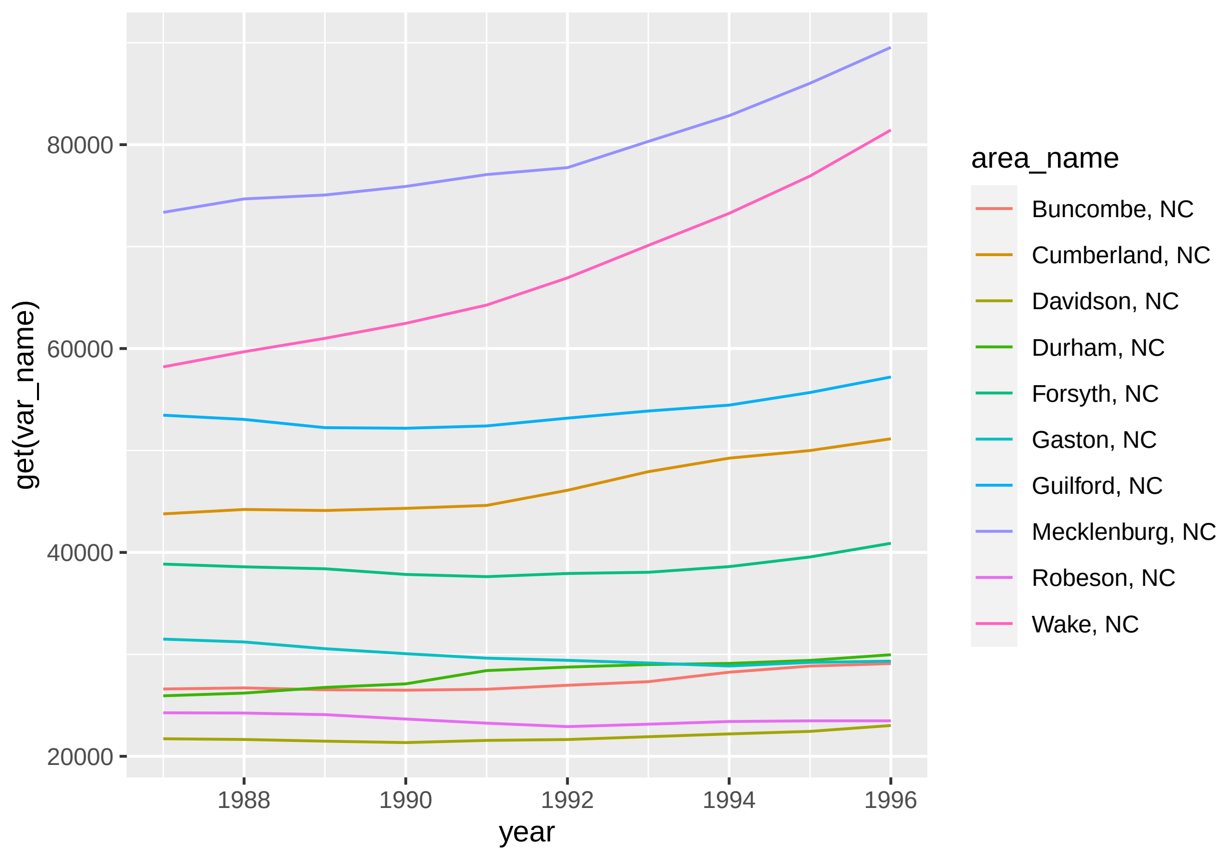
We use plot.county to specify the state to be "CT", the group being the top, the number looked at being 6 using groupAll_county data set.

```r
plot(grpAll_county, ST = 'CT', location = 'top', number = 6)
```



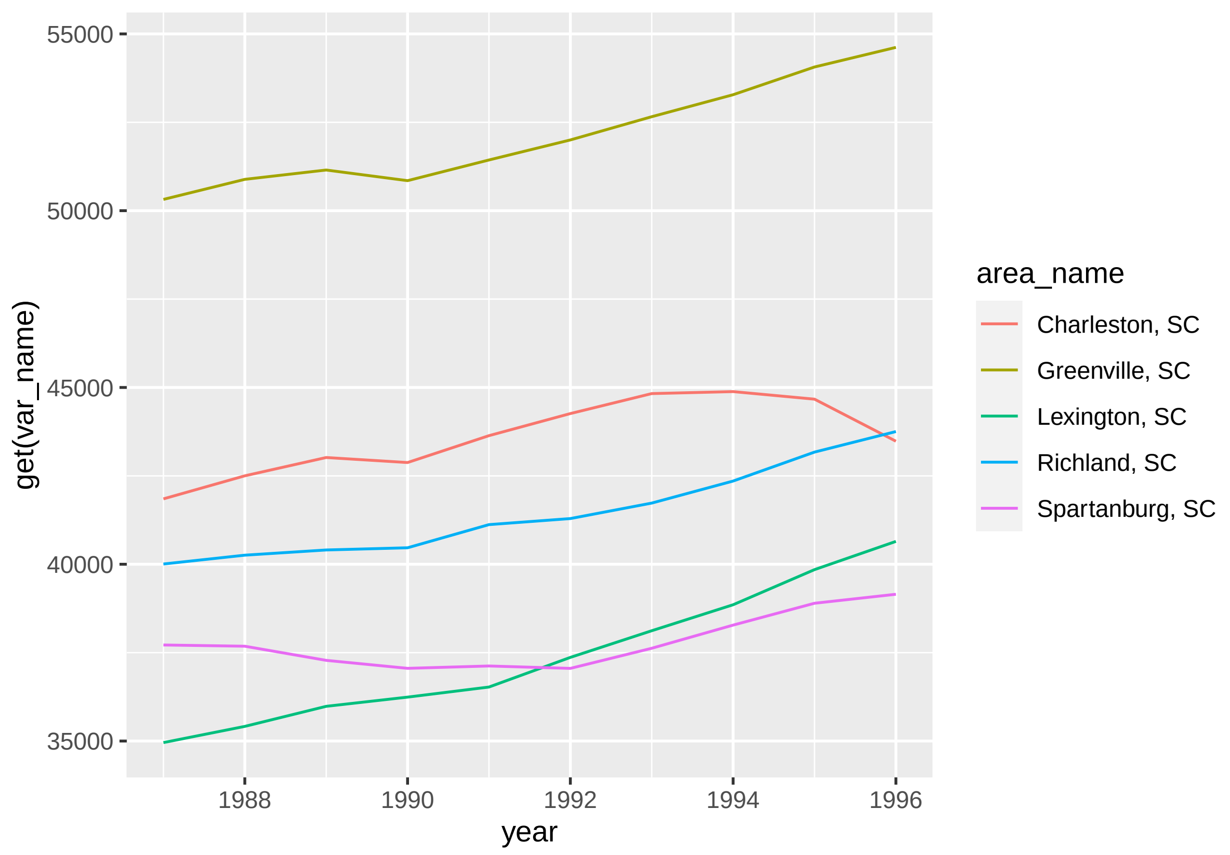
We use plot.county function specifying the state to be "NC", the group being the bottom, the number looked at being 10

```
plot(grpAll_county, ST = 'NC', location = 'btm', number = 10)
```



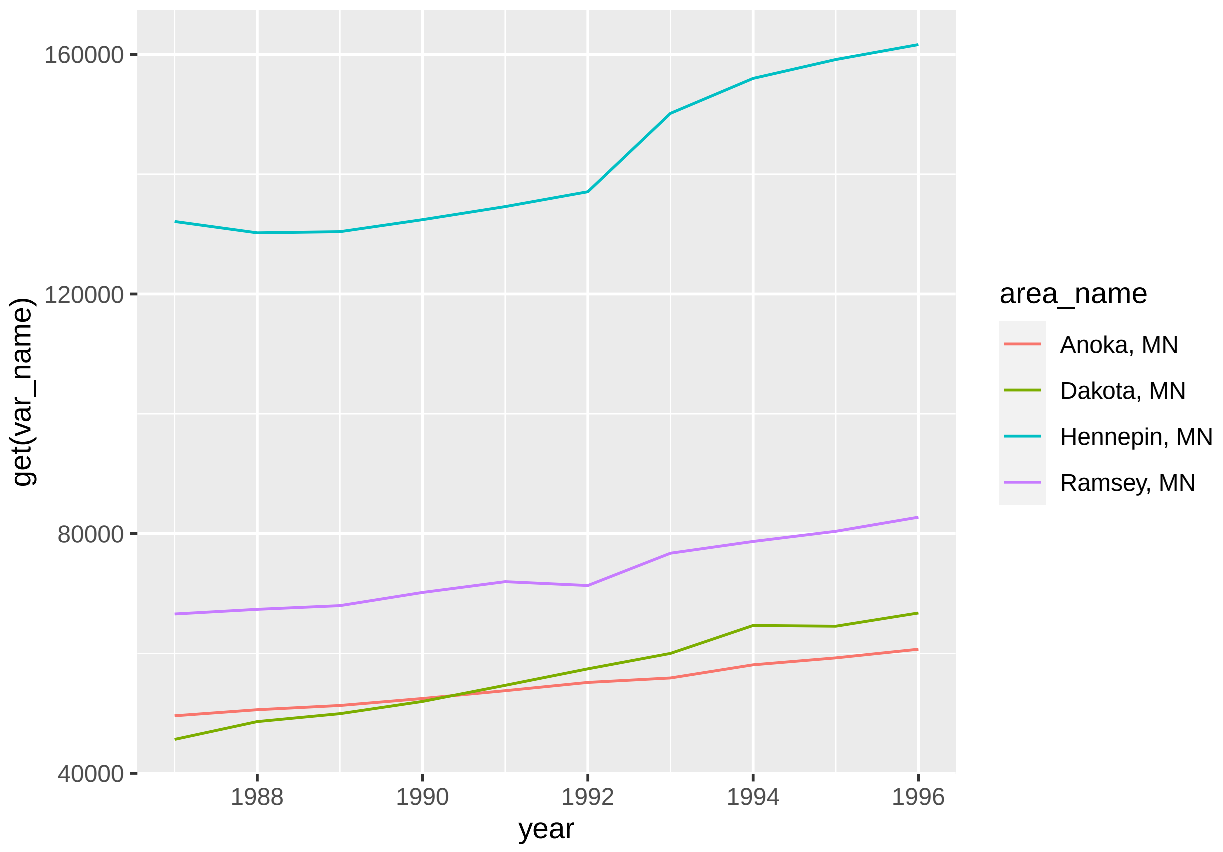
We use plot.county function without specifying anything (defaults used) Recall, the default arguments for any county plots. `plot.county <- function(df, var_name = "value",location = 'top', ST='SC', number=5)`
The graph below provides us with the top 5 counties in South Carolina.

```
plot(grpAll_county)
```



We use plot.county function specifying the state to be “MN”, the group being the top, the number looked at being 4.

```
plot(grpAll_county, ST = 'MN', location = 'top', number = 4)
```



Conclusion

This project demonstrates the importance of creating functions for repetitive tasks.

We have to create the code anyway, so take the additional step to add the wrapper to create a useful tool.