# NICAR 2020 - R1: Intro to R and RStudio

Meghan Hoyer & Ryan Thornburg

3/7/2020

## Welcome to R

R is a powerful open-source programming and statistical language. Scientists, statisticians and, increasingly, journalists are using it to find answers in mountains of data. People turn to R for everything from analyzing multi-year medical studies to preparing charts and drawing maps. The language itself, known as "base R", can do quite a bit. But computer scientists have vastly expanded its capabilities with more than 12,000 add-on tools or packages.

R is going to help you do more complex analyses in an easier way. But there is something of a learning curve. For one, R is a strict copy-editor - capitalization and spelling matter, so make sure you stick to some naming conventions and spell things correctly. Beyond that, and the biggest issue is there are approximately 1,543 ways to solve any problem in R. I'd recommend hewing as closely as you can to the principles of "tidy" data as you work.

The most popular of R packages is a suite of about 20 tools known as the tidyverse. It was developed largely by Hadley Wickham, chief scientist at RStudio, the graphical user interface for R. The tidyverse includes tools for importing, manipulating, editing and visualizing data and is based around a basic principle: that data be tidy, with one observation, or data point, in each row. "When your data is tidy, each column is a variable, and each row is an observation."

If you're looking for more resources or next steps to help you learn, here is a pretty comprehensive list of RStudio + R resources, including books, tipsheets and webinars compiled by Ron Campbell and me.

### First up: The basics of RStudio

RStudio is pretty nifty because it lets you see all the pieces of your project, and it even saves you from mistakes or problems later on by allowing you to run all or part of your code, and by keeping your history and showing you your loaded dataframes, functions and lists.

Here's a primer:

- The CONSOLE (bottom left) is where you can type in commands
- The SOURCE (top left) is where you'll see/open your scripts and also View() any datatable. Datatables opened in source (either by double-clicking from Environment or calling View() in your script or console) can also be explored through sorting and filtering
- ENVIRONMENT (top right) is your loaded data
- HISTORY (top right) is every command you've run, starting with your most recent command
- FILES (bottom left) allows you to navigate through your directory
- PLOTS (bottom left) shows you any output from ggplot

- PACKAGES (bottom left) shows you your system library, your package versions, and allows you to install new packages if needed
- HELP (bottom left) includes documentation for packages

### Setting Up Your Work

Since R is open-source, it is free. You can download R and most packages at CRAN, the Comprehensive R Archive Network. Installing and loading packages is a two-step process. To install the tidyverse, for example, you would enter the following commands at the console - note the quotes around the package name:

```
#install.packages("tidyverse")
```

IRE-NICAR staff has already installed R, R Studio and the tidyverse on the computers we'll be using in this class. But whenever you start R, you must do step 2: loading the packages you need in memory. All library calls should be included at the top of your scripts; that way, anyone using your code can see and load all the necessary packages to run your code.

So - let's open a new script (File -> New File -> R Script) and load the tidyverse package so we can get to work. Note for library calls there are no quotes around the library name.

```
library(tidyverse)
```

Generally, when you work in R, it's best to create a 'Project' in RStudio to make sure your installed packages, files and all other pieces are in one place. While we aren't creating a project for this class, we will go partway – we'll set a working directory. The directory we'll use is wherever the IRE-NICAR staff placed our data. To set the working directory go to the Session menu and click on Set Working Directory. We'll then look for the correct directory. Best practice: never code a hard-path to setwd() into your script - it'll work on your machine, but won't on anyone else's.

```
getwd()
```

```
## [1] "/Users/thornbur/OneDrive - University of North Carolina at Chapel Hill/Default Working Directory
```

```
#setwd()
```

### R Syntax

There are many parts of R that look a lot like other data tools – functions such as filtering, summing, and the like. But there are two main tools in writing R code that may be less familiar to you. First, to assign a value to a variable, we use an arrow-like thing called an assignment operator.

Here it is: <-

We'll also be writing multi-line formulas, joining each line with a device called a pipe. It looks like this: %>%. There's a shortcut for this: In Windows, type Shift-Control-M; in Mac, type Shift-Command-M. The pipe always goes at the end of a line, followed by a return.

Beyond that – give future you and anyone else who will use or read your code a break: Good R practice is to put spaces around assignment operators, to do a hard return after a pipe, and stick to a naming convention for dataframes, functions and variables. There are some styleguides for R here and here that will help you think about best practices.

Also don't forget to comment out your code! You can add a comment by putting in an R Script by putting a '#' before your words.

##Reading in Data

Finally – let's dig in! Today we're going to be creating several tables, known in R as data frames, and naming them as we go.

Here's the file you'll need. Download it and put it in the folder for this class that you've set as your directory above.

Now let's use read_csv to pull data from the Census into R.

There are other options for reading in files that are tab or pipe-delimited, in .txt, .xlsx or other formats – including SAS and Stata files, tables from a SQL database, Google sheets or the like. Note that for some of these, you'll need to download additional libraries.

```
census <- read_csv("data/grandparentsR1.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_character(),
##   id2 = col_double(),
##   Geography = col_character(),
##   TotalPop = col_double(),
##   PopOver30 = col_double(),
##   LivingWGrandChildren = col_double(),
##   Resp4GrandChildren = col_double(),
##   Resp4GrandChildren_30to59 = col_double(),
##   Resp4GrandChildren_Over60 = col_double()
## )
```

R does a pretty decent job of guessing your data types, but every once in awhile it messes them up and you'll have to rework them. For instance, check that id2 value. You can declare specific data types for each column while using read_csv, or you can change types on individual columns. For this, let's just fix the column individually by doing:

```
census$id2 <- as.character(census$id2)
```

### Looking at your Data

Now, let's open this dataframe up and explore as we might explore an Excel spreadsheet.

In your console, type:

```
View(census)
```

To limit the View on a large file you can also call head with a certain number of rows, similar to LIMIT in SQL:

```
head(census, 5) %>% View()
```

You can sort and filter this View, which allows you to get a basic familiarity with your dataframe.

But now let's start really digging into this to get an idea of what might be in our data. Summary – which is part of base R – isn't a bad place to start looking at things, particularly if you have a lot of numbers, because it returns minimum and maximum values for each column, as well as averages and medians.

```
summary(census)
```

```
##       id                 id2             Geography            TotalPop
##  Length:64          Length:64          Length:64          Min.   :  4771
##  Class :character   Class :character   Class :character   1st Qu.: 20046
##  Mode  :character   Mode  :character   Mode  :character   Median : 33436
##                                                           Mean   : 72867
##                                                           3rd Qu.: 75904
##                                                           Max.   :446167
##    PopOver30      LivingWGrandChildren Resp4GrandChildren
##  Min.   :  2924   Min.   :  102.0      Min.   :  58.0
##  1st Qu.: 11528   1st Qu.:  546.0      1st Qu.: 322.5
##  Median : 19102   Median :  847.5      Median : 452.0
##  Mean   : 41174   Mean   : 1821.7      Mean   : 902.1
##  3rd Qu.: 41816   3rd Qu.: 2088.5      3rd Qu.:1105.5
##  Max.   :261300   Max.   :11062.0      Max.   :4640.0
##  Resp4GrandChildren_30to59 Resp4GrandChildren_Over60
##  Min.   :  45.0            Min.   :   7.0
##  1st Qu.: 175.2            1st Qu.: 133.5
##  Median : 263.5            Median : 194.0
##  Mean   : 535.4            Mean   : 366.7
##  3rd Qu.: 673.0            3rd Qu.: 424.0
##  Max.   :2867.0            Max.   :1925.0
```

### Tidying your data

OK, that's great and all, but it's still really hard to understand what we're looking at. This data isn't tidy! There are lots of different demographic group breakdowns per row – we don't have just one observation in each row.

Let's tidy things up using a function called "pivot_longer" (which has recently replaced a similar function called "gather" that you will still see in tutorials that are more than about a year old.)

This is a somewhat complex operation, so you don't need to totally understand how it works right now, but basically we're reshaping our data, so that the basic information (FIPS codes, city name and total population) are attached to each individual observation (the breakdowns of the type of population and the count of that group). We're creating an individual row based on columns 5 - 9 of our dataframe.

Where data in Excel is usually wide, this makes our data long. The opposite of gather is spread, and it can work to make tables wide, with multiple observations per row if you so desire.

```
grandparents <- census %>% pivot_longer(cols = (5:9), names_to ="type", values_to = "pop_in_category")
```

Open that file up with View(grandparents). You'll see that now your datafram has a LOT more rows – 320 of them to be exact – but a fewer columns (only five). Now that things are tidy, we can really start to explore.

### Sanity-checking

You want to make sure the data is OK and you don't have duplicate parishes.

```
census %>% distinct(Geography) %>% count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1    64
```

You can also use count to see how many of a certain thing meets your filter criteria. Here, let's see how many parishes have more than 100,000 people:

```
census %>% filter(TotalPop>100000) %>% distinct(Geography) %>% count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1    14
```

### Filtering and arranging your data

You're probably familiar with filtering from Excel, or from using "WHERE" statements in SQL. This is similar.

Note that filtering on strings and characters requires a double equal sign.

```
census %>%
  filter(Geography == "West Feliciana Parish, Louisiana")
```

```
## # A tibble: 1 x 9
##   id    id2   Geography TotalPop PopOver30 LivingWGrandChi... Resp4GrandChild...
##   <chr> <chr> <chr>        <dbl>     <dbl>            <dbl>            <dbl>
## 1 0500... 22125 West Fel...    15376      9527              576              132
## # ... with 2 more variables: Resp4GrandChildren_30to59 <dbl>,
## #   Resp4GrandChildren_Over60 <dbl>
```

### Removing and adding columns: Select & Mutate functions

While the filter() function in R allows you to see only certain rows, the select() function lets you see only certain columns.

```
census %>%
  filter(Geography == "West Feliciana Parish, Louisiana")  %>%
  select(Geography, TotalPop, PopOver30, Resp4GrandChildren)
```

```
## # A tibble: 1 x 4
##   Geography                          TotalPop PopOver30 Resp4GrandChildren
##   <chr>                                 <dbl>     <dbl>              <dbl>
## 1 West Feliciana Parish, Louisiana      15376      9527                132
```

In R, creating a new column in your data is called *Mutating* – basically, you're changing your dataframe by adding a new column without changing anything in your existing dataframe.

```
census %>%
  mutate(pct = Resp4GrandChildren/PopOver30)
```

```
## # A tibble: 64 x 10
##    id    id2   Geography TotalPop PopOver30 LivingWGrandChi... Resp4GrandChild...
##    <chr> <chr> <chr>        <dbl>     <dbl>            <dbl>             <dbl>
##  1 0500... 22005 Ascensio...   119129     66493             2716              1085
##  2 0500... 22055 Lafayett...   238230    131202             3572              1237
##  3 0500... 22125 West Fel...    15376      9527              576               132
##  4 0500... 22051 Jefferso...   437038    261300            11062              4157
##  5 0500... 22011 Beaurega...    36598     21078             1139               764
##  6 0500... 22013 Bienvill...    13806      8265              502               335
##  7 0500... 22015 Bossier ...   125698     69026             3353              1905
##  8 0500... 22023 Cameron ...     6806      4140              102                58
##  9 0500... 22037 East Fel...    19553     12239              534               323
## 10 0500... 22039 Evangeli...    33750     18606              558               342
## # ... with 54 more rows, and 3 more variables: Resp4GrandChildren_30to59 <dbl>,
## #   Resp4GrandChildren_Over60 <dbl>, pct <dbl>
```

Sometimes its useful to create a new dataframe from once you've added and removed the columns you want for your final analysis. Here we'll create a dataframe called census_focused.

```
census_focused <- census %>%
  mutate(pct = Resp4GrandChildren/PopOver30) %>%
  select(Geography, TotalPop, PopOver30, Resp4GrandChildren, pct)
```

And since we don't want to have to go in and sort every time, so let's make it so the greatest share of grandparents caring for grandchildren is on top.

```
census_focused %>%
  arrange(desc(pct))
```

```
## # A tibble: 64 x 5
##    Geography                      TotalPop PopOver30 Resp4GrandChildren    pct
##    <chr>                             <dbl>     <dbl>              <dbl>  <dbl>
##  1 St. Helena Parish, Louisiana      10509      6179                364 0.0589
##  2 Madison Parish, Louisiana         11616      6471                346 0.0535
##  3 Tensas Parish, Louisiana           4771      2924                141 0.0482
##  4 Washington Parish, Louisiana      46449     27156               1167 0.0430
##  5 Iberville Parish, Louisiana       33122     19597                830 0.0424
##  6 Caldwell Parish, Louisiana        10000      5745                243 0.0423
##  7 Bienville Parish, Louisiana       13806      8265                335 0.0405
##  8 Richland Parish, Louisiana        20619     11771                467 0.0397
##  9 Red River Parish, Louisiana        8723      5191                190 0.0366
## 10 Avoyelles Parish, Louisiana       41095     23903                874 0.0366
## # ... with 54 more rows
```

## Summarize

Inside the summarize (or summarise) function there are many basic math operators – including mean, median, minimum, maximum and sum. For all of Louisiana's 64 parishes, let's find out what's the minimum, maximum and median percent of grandparents who are raising their grandchildren.

```
census_focused  %>%
  summarise(Median = median(pct), Minimum = min(pct), Maximum = max(pct))
```

```
## # A tibble: 1 x 3
##   Median Minimum Maximum
##    <dbl>   <dbl>   <dbl>
## 1 0.0257 0.00943  0.0589
```

## Creating and comparing groups

So we're working on a story and only want to look at parishes around New Orleans. And then maybe compare the New Orleans region to the rest of the state.

How do we do that?

First we'll need to filter to isolate the right parishes. Then we'll need to tell R to then add up the total population and the care-giving grandparents for all the parishes in each group.

There's a lot going on here, so let's break it down:

First, the filter. If you want to combine any group of things in R, you put parentheses around them and delineate them with a 'c' – so here, we're saying let's filter by choosing these two types of populations.

Rather than writing *filter(Geography == "Orleans Parish, Louisiana", Geography == "St. Bernard Parish, Louisiana", Geography == "St. Tammany Parish, Louisiana", Geography == "Jefferson Parish, Louisiana", Geography == "Plaquemines Parish, Louisiana")* we're going to create a new variable, which will be the set of parishes in the New Orleans region.

```
NOLA_region <- c("Orleans Parish, Louisiana", "St. Bernard Parish, Louisiana", "St. Tammany Parish, Lou
```

Next: Once we've created this collection of parishes that we're calling "NOLA_region" we need to create a new column and store in that column information about whether the parish is in the New Orleans region or in the rest of the state. We can create these new categorical variables using the case_when() function.

In the case_when function you will see the ~, which can be read as "then".

You will also see a new operator called *%in%* . We use this when we want to ask if a value is equal to any one of many values. And we also see the *!* operator. You may know that this means "not", but in R the *!* operator can sometimes go in places you're not accustomed to seeing it.

So this can be read as:

1. "Start with the census dataframe . . .
2. "and create a new column called region . . .
3. "and insert"NOLA Region" as the value of the new region column anytime the value of the Geography column is in the set of values in NOLA_Region. . . .
4. "and if the value isn't in the set, then set the value of the region column to"Rest of State" . . .
5. "and then take each row and put it into groups based on the values in the region column . . . "
6. "and then, for each group, add the values of PopOver30 and Resp4GrandChildren. Put those summary values into new columns called RegionalPopOver30 and RegionalResp4GrandChildren . . .
7. "and then divide RegionalResp4GrandChildren by RegionalPopOver30 and save that result in yet another new column called regional_pct"

**The code:**

```
regional <- census %>%
    mutate(region =
             case_when(Geography %in% NOLA_region ~ "NOLA Region", !(Geography %in% NOLA_region) ~ "Res
  group_by(region) %>%
  summarize(RegionalPopOver30 = sum(PopOver30), RegionalResp4GrandChildren = sum(Resp4GrandChildren)) %
  mutate(regional_pct = RegionalResp4GrandChildren / RegionalPopOver30)
```