Week 2 - Introduction to Mathematical Modelling

R Data Types

Let's now explore what R can do. R is really just a big fancy calculator. For example, type in the following mathematical expression in the R console (left window)

```
1+1
```

```
[1] 2
```

Note that spacing does not matter: 1+1 will generate the same answer as 1 + 1. Can you say hello to the world?

```
hello world
```

```
Error: <text>:1:7: unexpected symbol
1: hello world
          ^
```

Nope. What is the problem here? We need to put quotes around it.

```
"hello world"
```

```
[1] "hello world"
```

"hello world" is a character and R recognizes characters only if there are quotes around it. This brings us to the topic of basic data types in R. There are four basic data types in R: character, logical, numeric, and factors (there are two others - complex and raw - but we won't cover them because they are rarely used in practice).

Characters

Characters are used to represent words or letters in R. We saw this above with "hello world". Character values are also known as strings. You might think that the value "1" is a number. Well, with quotes around, it isn't! Anything with quotes will be interpreted as a character. No ifs, ands or buts about it.

Logicals

A logical takes on two values: FALSE or TRUE. Logicals are usually constructed with comparison operators, which we'll go through more carefully in Lab 2. Think of a logical as the answer to a question like "Is this value greater than (lower than/equal to) this other value?" The answer will be either TRUE or FALSE. TRUE and FALSE are logical values in R. For example, typing in the following

```
3 > 2
```

```
[1] TRUE
```

Gives you a true. What about the following?

```
"jacob" == "catherine"
```

```
[1] FALSE
```

Numeric

Numerics are separated into two types: integer and double. The distinction between integers and doubles is usually not important. R treats numerics as doubles by default because it is a less restrictive data type. You can do any mathematical operation on numeric values. We added one and one above. We can also multiply using the * operator.

```
2*3
```

```
[1] 6
```

Divide

```
4/2
```

```
[1] 2
```

And even take the logarithm!

```
log(1)
```

```
[1] 0
```

```
log(0)
```

```
[1] -Inf
```

Uh oh. What is -Inf? Well, you can't take the logarithm of 0, so R is telling you that you're getting a non numeric value in return. The value -Inf is another type of value type that you can get in R.

Factors

Think of a factor as a categorical variable. It is sort of like a character, but not really. It is actually a numeric code with character-valued levels. Think of a character as a true string and a factor as a set of categories represented as characters. We won't use factors too much in this course.

R Data Structures

You just learned that R has four basic data types. Now, let's go through how we can store data in R. That is, you type in the character "hello world" or the number 3, and you want to store these values. You do this by using R's various data structures.

Vectors

A vector is the most common and basic R data structure and is pretty much the workhorse of the language. A vector is simply a sequence of values which can be of any data type but all of the same type. There are a number of ways to create a vector depending on the data type, but the most common is to insert the data you want to save in a vector into the command c(). For example, to represent the values 4, 16 and 9 in a vector type in

```
c(4, 16, 9)
```

```
[1]  4 16  9
```

You can also have a vector of character values

```
c("jacob", "anne", "gwen")
```

```
[1] "jacob" "anne"  "gwen"
```

The above code does not actually "save" the values 4, 16, and 9 - it just presents it on the screen in a vector. If you want to use these values again without having to type out c(4, 16, 9), you can save it in a data object. At the heart of almost everything you will do (or ever likely to do) in R is the concept that everything in R is an object. These objects can be almost anything, from a single number or character string (like a word) to highly complex structures like the output of a plot, a map, a summary of your statistical analysis or a set of R commands that perform a specific task.

You assign data to an object using the arrow sign <-. This will create an object in R's memory that can be called back into the command window at any time. For example, you can save "hello world" to a vector called b by typing in

```
b <- "hello world"
b
```

```
[1] "hello world"
```

You can pronounce the above as "b becomes 'hello world' ".

Note that R is case sensitive, if you type in B instead of b, you will get an error.

Similarly, you can save the numbers 4, 16 and 9 into a vector called v1

```
v1 <- c(4, 16, 9)
v1
```

```
[1]  4 16  9
```

You should see the objects b and v1 pop up in the Environment tab on the top right window of your RStudio interface.

Environment window

Note that the name v1 is nothing special here. You could have named the object x or crd230 or your pet's name (mine was charlie). You can't, however, name objects using special characters (e.g. !, @, $) or only numbers (although you can combine numbers and letters, but a number cannot be at the beginning e.g. 2d2). For example, you'll get an error if you save the vector c(4,16,9) to an object with the following names

```
123 <- c(4, 16, 9)
!!! <- c(4, 16, 9)
```

```
Error: <text>:2:5: unexpected assignment
1: 123 <- c(4, 16, 9)
2: !!! <-
       ^
```

Also note that to distinguish a character value from a variable name, it needs to be quoted. "v1" is a character value whereas v1 is a variable. One of the most common mistakes for beginners is to forget the quotes.

```
brazil
```

```
Error in eval(expr, envir, enclos): object 'brazil' not found
```

The error occurs because R tries to print the value of object brazil, but there is no such variable. So remember that any time you get the error message object 'something' not found, the most likely reason is that you forgot to quote a character value. If not, it probably means that you have misspelled, or not yet created, the object that you are referring to. I've included the common pitfalls and R tips in this class resource.

Every vector has two key properties: type and length. The type property indicates the data type that the vector is holding. Use the command typeof() to determine the type

```r
typeof(b)
```

```
[1] "character"
```

```r
typeof(v1)
```

```
[1] "double"
```

Note that a vector cannot hold values of different types. If different data types exist, R will coerce the values into the highest type based on its internal hierarchy: logical < integer < double < character. Type in test <- c("r", 6, TRUE) in your R console. What is the vector type of test?

The command length() determines the number of data values that the vector is storing

```r
length(b)
```

```
[1] 1
```

```r
length(v1)
```

```
[1] 3
```

You can also directly determine if a vector is of a specific data type by using the command is.X() where you replace X with the data type. For example, to find out if v1 is numeric, type in

```r
is.numeric(b)
```

```
[1] FALSE
```

```r
is.numeric(v1)
```

```
[1] TRUE
```

There is also is.logical(), is.character(), and is.factor(). You can also coerce a vector of one data type to another. For example, save the value "1" and "2" (both in quotes) into a vector named x1

```r
x1 <- c("1", "2")
typeof(x1)
```

```
[1] "character"
```

To convert x1 into a numeric, use the command as.numeric()

```r
x2 <- as.numeric(x1)
typeof(x2)
```

```
[1] "double"
```

There is also as.logical(), as.character(), and as.factor().

An important practice you should adopt early is to keep only necessary objects in your current R Environment. For example, we will not be using x2 any longer in this guide. To remove this object from R forever, use the command rm()

```r
rm(x2)
```

The data frame object x2 should have disappeared from the Environment tab. Bye bye!

Also note that when you close down R Studio, the objects you created above will disappear for good. Unless you save them onto your hard drive (we'll touch on saving data in Lab 2), all data objects you create in your current R session will go bye bye when you exit the program.

Data Frames

We learned that data values can be stored in data structures known as vectors. The next step is to learn how to store vectors into an even higher level data structure. The data frame can do this. Data frames store vectors of the same length. Create a vector called v2 storing the values 5, 12, and 25

```r
v2 <- c(5,12,25)
```

We can create a data frame using the command data.frame() storing the vectors v1 and v2 as columns

```r
data.frame(v1, v2)
```

```
  v1 v2
1  4  5
2 16 12
3  9 25
```

Store this data frame in an object called df1

```r
df1<-data.frame(v1, v2)
```

df1 should pop up in your Environment window. You'll notice a next to df1. This tells you that df1 possesses or holds more than one object. Click on and you'll see the two vectors we saved into df1. Another neat thing you can do is directly click on df1 from the Environment window to bring up an Excel style worksheet on the top left of your RStudio interface. You can also type in

```r
View(df1)
```

```
Error in check_for_XQuartz(file.path(R.home("modules"), "R_de.so")): X11 library is missing:
```

to bring the worksheet up. You can't edit this worksheet directly, but it allows you to see the values that a higher level R data object contains.

We can store different types of vectors in a data frame. For example, we can store one character vector and one numeric vector in a single data frame.

```r
v3 <- c("jacob", "anne", "gwen")
df2 <- data.frame(v1, v3)
df2
```

```
  v1    v3
1  4 jacob
2 16  anne
3  9  gwen
```

For higher level data structures like a data frame, use the function class() to figure out what kind of object you're working with.

```r
class(df2)
```

```
[1] "data.frame"
```

We can't use length() on a data frame because it has more than one vector. Instead, it has dimensions - the number of rows and columns. You can find the number of rows and columns that a data frame has by using the command dim()

```r
dim(df1)
```

```
[1] 3 2
```

Here, the data frame df1 has 3 rows and 2 columns. Data frames also have column names, which are characters.

```r
colnames(df1)
```

```
[1] "v1" "v2"
```

In this case, the data frame used the vector names for the column names.

We can extract columns from data frames by referring to their names using the $ sign.

```r
df1$v1
```

```
[1]  4 16  9
```

We can also extra data from data frames using brackets [,]

```r
df1[,1]
```

```
[1]  4 16  9
```

The value before the comma indicates the row, which you leave empty if you are not selecting by row, which we did above. The value after the comma indicates the column, which you leave empty if you are not selecting by column. The above line of code selected the first column. Let's select the 2nd row.

```r
df1[2,]
```

```
  v1 v2
2 16 12
```

What is the value in the 2nd row and 1st column?

```r
df1[2,1]
```

```
[1] 16
```

Functions

Let's take a step back and talk about functions (also known as commands). An R function is a packaged recipe that converts one or more inputs (called arguments) into a single output. You execute all of your tasks in R using functions. We have already used a couple of functions above including typeof() and colnames(). Every function in R will have the following basic format

```
functionName(arg1 = val1, arg2 = val2, ...)
```

In R, you type in the function's name and set a number of options or parameters within parentheses that are separated by commas. Some options need to be set by the user - i.e. the function will spit out an error because a required option is blank - whereas others can be set but are not required because there is a default value established.

Let's use the function seq() which makes regular sequences of numbers. You can find out what the options are for a function by calling up its help documentation by typing ? and the function name

```
? seq
```

The help documentation should have popped up in the bottom right window of your RStudio interface. The documentation should also provide some examples of the function at the bottom of the page. Type the arguments from = 1, to = 10 inside the parentheses

```r
seq(from = 1, to = 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

You should get the same result if you type in

```r
seq(1, 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

The code above demonstrates something about how R resolves function arguments. When you use a function, you can always specify all the arguments in arg = value form. But if you do not, R attempts to resolve by position. So in the code above, it is assumed that we want a sequence from = 1 that goes to = 10 because we typed 1 before 10. Type in 10 before 1 and see what happens. Since we didn't specify step size, the default value of by in the function definition is used, which ends up being 1 in this case.

Each argument requires a certain type of data type. For example, you'll get an error when you use character values in seq()

```r
seq("p", "w")
```

```
Warning in seq.default("p", "w"): NAs introduced by coercion
```

```
Error in seq.default("p", "w"): 'from' must be a finite number
```

Packages

Functions do not exist in a vacuum, but exist within R packages. Packages are the fundamental units of reproducible R code. They include reusable R functions, the documentation that describes how to use them, and sample data. At the top left of a function's help documentation, you'll find in curly brackets the R package that the function is housed in. For example, type in your console ? seq. At the top right of the help documentation, you'll find that seq() is in the package base. All the functions we have used so far are part of packages that have been pre-installed and pre-loaded into R.

In order to use functions in a new package, you first need to install the package using the install.packages() command. For example, we will be using commands from the package tidyverse in this lab. If you are working on a campus lab computer, you will likely not need to install this package.

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

You should see a bunch of gobbledygook roll through your console screen. Don't worry, that's just R downloading all of the other packages and applications that tidyverse relies on. These are known as dependencies. Unless you get a message in red that indicates there is an error (like we saw when we typed in "hello world" without quotes), you should be fine.

Next, you will need to load packages in your working environment (every time you start RStudio). We do this with the library() function. Notice there are no quotes around tidyverse this time.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.4.4     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

The Packages window at the lower-right of your RStudio shows you all the packages you currently have installed. If you don't have a package listed in this window, you'll need to use the install.packages() function to install it. If the package is checked, that means it is loaded into your current R session

To uninstall a package, use the function remove.packages().

Note that you only need to install packages once (install.pacakges()), but you need to load them each time you relaunch RStudio (library()). Repeat after me: Install once, library every time. If you need to reinstall R or update to a new version of R, you will need to reinstall all packages. And as noted earlier, R has several packages already preloaded into your working environment. These are known as base packages and a list of their functions can be found here.

Tidyverse

In most labs, we will be using commands from the tidyverse package. Tidyverse is a collection of high-powered, consistent, and easy-to-use packages developed by a number of thoughtful and talented R developers.The consistency of the tidyverse, together with the goal of increasing productivity, mean that the syntax of tidy functions is typically straightforward to learn.

Tibbles

Although the tidyverse works with all data objects, its fundamental object type is the tibble. Tibbles are data frames, but they tweak some older behaviors to make life a little easier. There are two main differences in the usage of a data frame vs a tibble: printing and subsetting. Let's be clear here - tibbles are just a special kind of data frame. They just makes things a little "tidier." Let's bring in some data to illustrate the differences and similarities between data frames and tibbles. Install the package nycflights13

```
install.packages("nycflights13")
```

Make sure you also load the package.

```
library(nycflights13)
```

There is a dataset called flights included in this package. It includes information on all 336,776 flights that departed from New York City in 2013. Let's save this file in the local R environment

```
nyctib <- flights
class(nyctib)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

This dataset is a tibble. Let's also save it as a regular data frame by using the as.data.frame() function

```
nycdf <- as.data.frame(flights)
class(nycdf)
```

```
[1] "data.frame"
```

The first difference between data frames and tibbles is how the dataset looks. Tibbles have a refined print method that shows only the first 10 rows, and only the columns that fit on the screen. In addition, each column reports its name and type.

```
nyctib
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Tibbles are designed so that you don't overwhelm your console when you print large data frames. Compare the print output above to what you get with a data frame

```
nycdf
```

Ugly, right? You can bring up the Excel like worksheet of the tibble (or data frame) using the View() function

```r
View(nyctib)
```

```
Error in check_for_XQuartz(file.path(R.home("modules"), "R_de.so")): X11 library is missing:
```

You can identify the names of the columns (and hence the variables in the dataset) by using the function names()

```r
names(nyctib)
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
 [9] "arr_delay"      "carrier"        "flight"         "tailnum"
[13] "origin"         "dest"           "air_time"       "distance"
[17] "hour"           "minute"         "time_hour"
```

Finally, you can convert a regular data frame to a tibble using the as_tibble() function

```
as_tibble(nycdf)
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Not all functions work with tibbles, particularly those that are specific to spatial data. As such, we'll be using a combination of tibbles and regular data frames throughout the class, with a preference towards tibbles where possible. Note that when you search on Google for how to do something in R, you will likely get non tidy ways of doing things. Most of these suggestions are fine, but some are not and may screw you up down the road. My advice is to try to stick with tidy functions to do things in R.

Data Wrangling

It is rare that the data work on are in exactly the right form for analysis. For example, you might want to discard certain variables from the dataset to reduce clutter. Or you need to create new variables from existing ones. Or you encounter missing data. The process of gathering data in its raw form and molding it into a form that is suitable for its end use is known as data wrangling. What's great about the tidyverse package is its suite of functions make data wrangling relatively easy, straight forward, and transparent.

In this lab, we won't have time to go through all of the methods and functions in R that are associated with the data wrangling process. We will cover more in later labs and many methods you will have to learn on your own given the specific tasks you will need to accomplish.

In the rest of this guide, we'll go through some of the basic data wrangling techniques using the functions found in the package dplyr, which was automatically installed and loaded when you brought in the tidyverse package. These functions can be used for either tibbles or regular data frames.

Reading in data

The dataset nycflights13 was included in an R package. In most cases, you'll have to read it in. Most data files you will encounter are comma-delimited (or comma-separated) files, which have .csv extensions. Comma-delimited means that columns are separated by commas. We're going to bring in two csv files lab1dataset1.csv and lab1dataset2.csv. The first file is a county-level dataset containing median household income. The second file is also a county-level dataset containing Non-Hispanic white, Non-Hispanic black, non-Hispanic Asian, and Hispanic population counts. Both data sets come from the 2014-2018 American Community Survey (ACS). We'll cover the Census, and how to download Census data, in the next lab.

To read in a csv file, use the function read_csv(), which is a part of the tidyverse package, and plug in the name of the file in quotes inside the parentheses. Make sure you include the .csv extension. I uploaded the two files on GitHub, so you can read them in directly from there. We'll name these objects ca1 and ca2

```
ca1 <- read_csv("https://raw.githubusercontent.com/crd230/data/master/lab1dataset1.csv")
```

```
Rows: 58 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): County, Formatted FIPS
dbl (2): FIPS Code, Estimated median income of a household, between 2014-2018.

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
ca2 <- read_csv("https://raw.githubusercontent.com/crd230/data/master/lab1dataset2.csv")
```

```
Rows: 58 Columns: 12
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (2): GEOID, NAME
dbl (10): tpoprE, tpoprM, nhwhiteE, nhwhiteM, nhblkE, nhblkM, nhasnE, nhasnM...
```

```
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

You should see two tibbles ca1 and ca2 pop up in your Environment window (top right). Every time you bring a dataset into R for the first time, look at it to make sure you understand its structure. You can do this a number of ways. One is to use the function glimpse(), which gives you a succinct summary of your data.

```
glimpse(ca1)
```

```
Rows: 58
Columns: 4
$ `FIPS Code`                                             <dbl> 6071, 602~
$ County                                                  <chr> "San Bern~
$ `Formatted FIPS`                                        <chr> "06071", ~
$ `Estimated median income of a household, between 2014-2018.` <dbl> 60164, 52~
```

```
glimpse(ca2)
```

```
Rows: 58
Columns: 12
$ GEOID    <chr> "06033", "06047", "06043", "06049", "06013", "06027", "06099"~
$ NAME     <chr> "Lake County, California", "Merced County, California", "Mari~
$ tpoprE   <dbl> 64148, 269075, 17540, 8938, 1133247, 18085, 539301, 443738, 1~
$ tpoprM   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ nhwhiteE <dbl> 45623, 76008, 14125, 6962, 502951, 11389, 229796, 199356, 923~
$ nhwhiteM <dbl> 30, 200, 31, 6, 607, 26, 445, 221, 121, 38, 980, 166, 201, 48~
$ nhblkE   <dbl> 1426, 8038, 166, 149, 93683, 160, 14338, 7881, 40, 434, 14400~
$ nhblkM   <dbl> 112, 371, 111, 97, 1433, 37, 584, 449, 47, 88, 2016, 209, 438~
$ nhasnE   <dbl> 642, 19487, 243, 130, 182135, 289, 28599, 22996, 336, 1705, 2~
$ nhasnM   <dbl> 187, 630, 95, 118, 1993, 62, 876, 507, 223, 125, 1893, 402, 6~
$ hispE    <dbl> 12830, 158494, 1909, 1292, 288101, 3927, 245973, 200060, 3866~
$ hispM    <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
```

If you like viewing your data through an Excel style worksheet, type in View(ca1), and ca1 should pop up in the top left window of your R Studio interface. Scroll up and down, left and right.

We'll learn how to summarize your data using descriptive statistics and graphs in the next lab.

Renaming variables

You will likely encounter a variable with a name that is not descriptive. The more descriptive the variable names, the more efficient your analysis will be and the less likely you are going to make a mistake. To see the names of variables in your dataset, use the names() command.

```
names(ca1)
```

```
[1] "FIPS Code"
[2] "County"
[3] "Formatted FIPS"
[4] "Estimated median income of a household, between 2014-2018."
```

The name Estimated median income of a household, between 2014-2018. is super duper long! Use the command rename() to - what else? - rename a variable! Let's rename Estimated median income of a household, between 2014-2018. to medinc.

```
rename(ca1, medinc = "Estimated median income of a household, between 2014-2018.")
```

```
# A tibble: 58 x 4
   `FIPS Code` County         `Formatted FIPS` medinc
         <dbl> <chr>          <chr>             <dbl>
 1        6071 San Bernardino 06071             60164
 2        6027 Inyo           06027             52874
 3        6029 Kern           06029             52479
 4        6093 Siskiyou       06093             44200
 5        6065 Riverside      06065             63948
 6        6019 Fresno         06019             51261
 7        6035 Lassen         06035             56362
 8        6049 Modoc          06049             45149
 9        6107 Tulare         06107             47518
10        6023 Humboldt       06023             45528
# i 48 more rows
```

Note that you can rename multiple variables within the same rename() command. For example, we can also rename Formatted FIPS to GEOID. Make this permanent by assigning it back to ca1 using the arrow operator <-

```
ca1 <- rename(ca1, medinc = "Estimated median income of a household, between 2014-2018.",
                 GEOID = "Formatted FIPS")
names(ca1)
```

```
[1] "FIPS Code" "County"    "GEOID"     "medinc"
```

Selecting variables

In practice, most of the data files you will download will contain variables you don't need. It is easier to work with a smaller dataset as it reduces clutter and clears up memory space, which is important if you are executing complex tasks on a large number of observations. Use the command select() to keep variables by name. Visually, we are doing the following (taken from the RStudio cheatsheet)

Let's take a look at the variables we have in the ca2 dataset

```
names(ca2)
```

```
 [1] "GEOID"    "NAME"     "tpoprE"   "tpoprM"   "nhwhiteE" "nhwhiteM"
 [7] "nhblkE"   "nhblkM"   "nhasnE"   "nhasnM"   "hispE"    "hispM"
```

We'll go into more detail what these variables mean next lab when we cover the U.S. Census, but we only want to keep the variables GEOID, which is the county FIPS code (a unique numeric identifier), and tpoprE, nhwhiteE, nhblkE, nhasnE, and hispE, which are the total, white, black, Asian and Hispanic population counts.

```
ca2 <- select(ca2, GEOID, tpoprE, nhwhiteE, nhblkE, nhasnE, hispE)
```

Here, we provide the data object first, followed by the variables we want to keep separated by commas.

Let's keep County, GEOID, and medinc from the ca1 dataset. Rather than listing all the variables we want to keep like we did above, a shortcut way of doing this is to use the : operator.

```
select(ca1, County:medinc)
```

```
# A tibble: 58 x 3
   County         GEOID medinc
   <chr>          <chr>  <dbl>
 1 San Bernardino 06071  60164
 2 Inyo           06027  52874
 3 Kern           06029  52479
 4 Siskiyou       06093  44200
 5 Riverside      06065  63948
 6 Fresno         06019  51261
 7 Lassen         06035  56362
```

```
 8 Modoc          06049  45149
 9 Tulare         06107  47518
10 Humboldt       06023  45528
# i 48 more rows
```

The : operator tells R to select all the variables from County to medinc. This operator is useful when you've got a lot of variables to keep and they all happen to be ordered sequentially.

You can use also use select() command to keep variables except for the ones you designate. For example, to keep all variables in ca1 except FIPS Code and save this back into ca1, type in

```
ca1 <- select(ca1, -"FIPS Code")
```

The negative sign tells R to exclude the variable. Notice we need to use quotes around FIPS Code because it contains a space. You can delete multiple variables. For example, if you wanted to keep all variables except FIPS Code and County, you would type in select(ca1, -"FIPS Code", -County).

Take a glimpse to see if we got what we wanted.

```
glimpse(ca1)
```

```
Rows: 58
Columns: 3
$ County <chr> "San Bernardino", "Inyo", "Kern", "Siskiyou", "Riverside", "Fre~
$ GEOID  <chr> "06071", "06027", "06029", "06093", "06065", "06019", "06035", ~
$ medinc <dbl> 60164, 52874, 52479, 44200, 63948, 51261, 56362, 45149, 47518, ~
```

Do the same for ca2.

Creating new variables

The mutate() function allows you to create new variables within your dataset. This is important when you need to transform variables in some way - for example, calculating a ratio or adding two variables together. Visually, you are doing this

You can use the mutate() command to generate as many new variables as you would like. For example, let's construct four new variables in ca2 - the percent of residents who are non-Hispanic white, non-Hispanic Asian, non-Hispanic black, and Hispanic. Name these variables pwhite, pasian, pblack, and phisp, respectively.

```r
mutate(ca2, pwhite = nhwhiteE/tpoprE, pasian = nhasnE/tpoprE,
              pblack = nhblkE/tpoprE, phisp = hispE/tpoprE)
```

```
# A tibble: 58 x 10
   GEOID   tpoprE nhwhiteE nhblkE nhasnE  hispE pwhite pasian  pblack phisp
   <chr>    <dbl>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>   <dbl> <dbl>
 1 06033    64148    45623   1426    642  12830  0.711 0.0100 0.0222  0.200
 2 06047   269075    76008   8038  19487 158494  0.282 0.0724 0.0299  0.589
 3 06043    17540    14125    166    243   1909  0.805 0.0139 0.00946 0.109
 4 06049     8938     6962    149    130   1292  0.779 0.0145 0.0167  0.145
 5 06013 1133247   502951  93683 182135 288101  0.444 0.161  0.0827  0.254
 6 06027    18085    11389    160    289   3927  0.630 0.0160 0.00885 0.217
 7 06099   539301   229796  14338  28599 245973  0.426 0.0530 0.0266  0.456
 8 06083   443738   199356   7881  22996 200060  0.449 0.0518 0.0178  0.451
 9 06051    14174     9234     40    336   3866  0.651 0.0237 0.00282 0.273
10 06069    59416    20780    434   1705  35248  0.350 0.0287 0.00730 0.593
# i 48 more rows
```

Note that you can create new variables based on the variables you just created in the same line of code. For example, you can create a categorical variable yielding "Majority" if the tract is majority Hispanic and "Not Majority" otherwise after creating the percent Hispanic variable within the same mutate() command. Let's save these changes back into ca2.

```r
ca2 <- mutate(ca2, pwhite = nhwhiteE/tpoprE, pasian = nhasnE/tpoprE,
              pblack = nhblkE/tpoprE, phisp = hispE/tpoprE,
              mhisp = case_when(phisp > 0.5 ~ "Majority",
                                TRUE ~ "Not Majority"))
```

We used the function case_when() to create mhisp - the function tells R that if the condition phisp > 0.5 is met, the tract's value for the variable mhisp will be "Majority", otherwise (designated by TRUE) it will be "Not Majority".

Take a look at our data

```r
glimpse(ca2)
```

```
Rows: 58
Columns: 11
$ GEOID    <chr> "06033", "06047", "06043", "06049", "06013", "06027", "06099"~
$ tpoprE   <dbl> 64148, 269075, 17540, 8938, 1133247, 18085, 539301, 443738, 1~
$ nhwhiteE <dbl> 45623, 76008, 14125, 6962, 502951, 11389, 229796, 199356, 923~
```

```
$ nhblkE   <dbl> 1426, 8038, 166, 149, 93683, 160, 14338, 7881, 40, 434, 14400~
$ nhasnE   <dbl> 642, 19487, 243, 130, 182135, 289, 28599, 22996, 336, 1705, 2~
$ hispE    <dbl> 12830, 158494, 1909, 1292, 288101, 3927, 245973, 200060, 3866~
$ pwhite   <dbl> 0.7112147, 0.2824789, 0.8053022, 0.7789215, 0.4438141, 0.6297~
$ pasian   <dbl> 0.010008106, 0.072422187, 0.013854048, 0.014544641, 0.1607195~
$ pblack   <dbl> 0.022229843, 0.029872712, 0.009464082, 0.016670396, 0.0826677~
$ phisp    <dbl> 0.20000624, 0.58903280, 0.10883694, 0.14455135, 0.25422613, 0~
$ mhisp    <chr> "Not Majority", "Majority", "Not Majority", "Not Majority", "~
```

Joining tables

Rather than working on two separate datasets, we should join the two datasets ca1 and ca2, because we may want to examine the relationship between median household income, which is in ca1, and racial/ethnic composition, which is in ca2. To do this, we need a unique ID that connects the tracts across the two files. The unique Census ID for a county combines the county and state IDs. The Census ID is named GEOID in both files. The IDs should be the same data class, which is the case.

```r
class(ca1$GEOID)
```

```
[1] "character"
```

```r
class(ca2$GEOID)
```

```
[1] "character"
```

If they are not the same class, we can coerce them using the as.numeric() or as.character() function described earlier.

To merge the datasets together, use the function left_join(), which matches pairs of observations whenever their keys or IDs are equal. We match on the variable GEOID and save the merged data set into a new object called cacounty.

```r
cacounty <- left_join(ca1, ca2, by = "GEOID")
```

We want to merge ca2 into ca1, so that's why the sequence is ca1, ca2. The argument by tells R which variable(s) to match rows on, in this case GEOID. You can match on multiple variables and you can also match on a single variable with different variable names (see the left_join() help documentation for how to do this). The number of columns in cacounty equals the number of columns in ca1 plus the number of columns in ca2 minus the ID variable you merged on.

Note that if you have two variables with the same name in both files, R will attach a .x to the variable name in ca1 and a .y to the variable name in ca1. For example, if you have a variable named Robert in both files, cacounty will contain both variables and name it Robert.x (the variable in ca1) and Robert.y (the variable in ca1). Try to avoid having variables with the same names in the two files you want to merge.

Let's use select() to keep the necessary variables.

```
cacounty <- select(cacounty, GEOID, County, pwhite, pasian, pblack, phisp, mhisp, medinc)
```

Filtering

Filtering means selecting rows/observations based on their values. To filter in R, use the command filter(). Visually, filtering rows looks like.

The first argument in the parentheses of this command is the name of the data frame. The second and any subsequent arguments (separated by commas) are the expressions that filter the data frame. For example, we can select Sacramento county using its FIPS code

```
filter(cacounty, GEOID == "06067")
```

```
# A tibble: 1 x 8
  GEOID County     pwhite pasian pblack phisp mhisp        medinc
  <chr> <chr>       <dbl>  <dbl>  <dbl> <dbl> <chr>         <dbl>
1 06067 Sacramento  0.452  0.153 0.0954 0.230 Not Majority  63902
```

The double equal operator == means equal to. We can also explicitly exclude cases and keep everything else by using the not equal operator !=. The following code excludes Sacramento county.

```
filter(cacounty, GEOID != "06067")
```

```
# A tibble: 57 x 8
   GEOID County         pwhite pasian  pblack phisp mhisp        medinc
   <chr> <chr>           <dbl>  <dbl>   <dbl> <dbl> <chr>         <dbl>
 1 06071 San Bernardino  0.292 0.0682 0.0791  0.528 Majority      60164
 2 06027 Inyo            0.630 0.0160 0.00885 0.217 Not Majority  52874
 3 06029 Kern            0.348 0.0456 0.0510  0.528 Majority      52479
```

```
 4 06093 Siskiyou        0.767 0.0154 0.0142  0.123 Not Majority  44200
 5 06065 Riverside       0.359 0.0620 0.0606  0.484 Not Majority  63948
 6 06019 Fresno          0.298 0.100  0.0455  0.527 Majority      51261
 7 06035 Lassen          0.658 0.0140 0.0864  0.187 Not Majority  56362
 8 06049 Modoc           0.779 0.0145 0.0167  0.145 Not Majority  45149
 9 06107 Tulare          0.290 0.0321 0.0127  0.641 Majority      47518
10 06023 Humboldt        0.746 0.0298 0.00988 0.113 Not Majority  45528
# i 47 more rows
```

What about filtering if a county has a value greater than a specified value? For example, counties with a percent white greater than 0.5 (50%).

```r
filter(cacounty, pwhite > 0.5)
```

```
# A tibble: 30 x 8
   GEOID County          pwhite pasian  pblack  phisp mhisp        medinc
   <chr> <chr>            <dbl>  <dbl>   <dbl>  <dbl> <chr>         <dbl>
 1 06027 Inyo             0.630 0.0160 0.00885 0.217  Not Majority  52874
 2 06093 Siskiyou         0.767 0.0154 0.0142  0.123  Not Majority  44200
 3 06035 Lassen           0.658 0.0140 0.0864  0.187  Not Majority  56362
 4 06049 Modoc            0.779 0.0145 0.0167  0.145  Not Majority  45149
 5 06023 Humboldt         0.746 0.0298 0.00988 0.113  Not Majority  45528
 6 06089 Shasta           0.802 0.0297 0.0119  0.0983 Not Majority  50905
 7 06045 Mendocino        0.656 0.0191 0.00585 0.248  Not Majority  49233
 8 06105 Trinity          0.825 0.0139 0.00676 0.0723 Not Majority  38497
 9 06079 San Luis Obispo  0.691 0.0353 0.0175  0.224  Not Majority  70699
10 06103 Tehama           0.687 0.0152 0.00663 0.247  Not Majority  42899
# i 20 more rows
```

What about less than 0.5 (50%)?

```r
filter(cacounty, pwhite < 0.5)
```

```
# A tibble: 28 x 8
   GEOID County         pwhite pasian pblack phisp mhisp        medinc
   <chr> <chr>           <dbl>  <dbl>  <dbl> <dbl> <chr>         <dbl>
 1 06071 San Bernardino  0.292 0.0682 0.0791 0.528 Majority      60164
 2 06029 Kern            0.348 0.0456 0.0510 0.528 Majority      52479
 3 06065 Riverside       0.359 0.0620 0.0606 0.484 Not Majority  63948
 4 06019 Fresno          0.298 0.100  0.0455 0.527 Majority      51261
 5 06107 Tulare          0.290 0.0321 0.0127 0.641 Majority      47518
```

```
 6 06037 Los Angeles     0.263 0.144  0.0788 0.485 Not Majority  64251
 7 06073 San Diego       0.459 0.116  0.0471 0.335 Not Majority  74855
 8 06025 Imperial        0.110 0.0132 0.0217 0.838 Majority      45834
 9 06053 Monterey        0.303 0.0546 0.0245 0.583 Majority      66676
10 06083 Santa Barbara   0.449 0.0518 0.0178 0.451 Not Majority  71657
# i 18 more rows
```

Both lines of code do not include counties that have a percent white equal to 0.5. We include it by using the less than or equal operator <= or greater than or equal operator >=.

```
filter(cacounty, pwhite <= 0.5)
```

```
# A tibble: 28 x 8
   GEOID County          pwhite pasian pblack phisp mhisp        medinc
   <chr> <chr>            <dbl>  <dbl>  <dbl> <dbl> <chr>         <dbl>
 1 06071 San Bernardino   0.292 0.0682 0.0791 0.528 Majority      60164
 2 06029 Kern             0.348 0.0456 0.0510 0.528 Majority      52479
 3 06065 Riverside        0.359 0.0620 0.0606 0.484 Not Majority  63948
 4 06019 Fresno           0.298 0.100  0.0455 0.527 Majority      51261
 5 06107 Tulare           0.290 0.0321 0.0127 0.641 Majority      47518
 6 06037 Los Angeles      0.263 0.144  0.0788 0.485 Not Majority  64251
 7 06073 San Diego        0.459 0.116  0.0471 0.335 Not Majority  74855
 8 06025 Imperial         0.110 0.0132 0.0217 0.838 Majority      45834
 9 06053 Monterey         0.303 0.0546 0.0245 0.583 Majority      66676
10 06083 Santa Barbara    0.449 0.0518 0.0178 0.451 Not Majority  71657
# i 18 more rows
```

In addition to comparison operators, filtering may also utilize logical operators that make multiple selections. There are three basic logical operators: & (and), | is (or), and ! is (not). We can keep counties with phisp greater than 0.5 and medinc greater than 50000 percent using &.

```
filter(cacounty, phisp > 0.5 & medinc > 50000)
```

```
# A tibble: 9 x 8
  GEOID County          pwhite pasian  pblack phisp mhisp    medinc
  <chr> <chr>            <dbl>  <dbl>   <dbl> <dbl> <chr>     <dbl>
1 06071 San Bernardino   0.292 0.0682 0.0791  0.528 Majority  60164
2 06029 Kern             0.348 0.0456 0.0510  0.528 Majority  52479
3 06019 Fresno           0.298 0.100  0.0455  0.527 Majority  51261
4 06053 Monterey         0.303 0.0546 0.0245  0.583 Majority  66676
```

```
5 06039 Madera          0.345 0.0197 0.0312  0.573 Majority  52884
6 06047 Merced          0.282 0.0724 0.0299  0.589 Majority  50129
7 06069 San Benito      0.350 0.0287 0.00730 0.593 Majority  81977
8 06031 Kings           0.327 0.0382 0.0585  0.541 Majority  53865
9 06011 Colusa          0.357 0.0151 0.0129  0.590 Majority  56704
```

Use | to keep counties with a GEOID of 06067 (Sacramento) or 06113 (Yolo) or 06075 (San Francisco)

```
filter(cacounty, GEOID == "06067" | GEOID == "06113" | GEOID == "06075")
```

```
# A tibble: 3 x 8
  GEOID County        pwhite pasian pblack phisp mhisp        medinc
  <chr> <chr>          <dbl>  <dbl>  <dbl> <dbl> <chr>         <dbl>
1 06113 Yolo           0.471  0.137 0.0243 0.315 Not Majority  65923
2 06067 Sacramento     0.452  0.153 0.0954 0.230 Not Majority  63902
3 06075 San Francisco  0.406  0.339 0.0501 0.152 Not Majority 104552
```

You've gone through some of the basic data wrangling functions offered by tidyverse.

R Markdown

In running the lines of code above, we've asked you to work directly in the R Console and issue commands in an interactive way. That is, you type a command after >, you hit enter/return, R responds, you type the next command, hit enter/return, R responds, and so on. Instead of writing the command directly into the console, you should write it in a script. The process is now: Type your command in the script. Run the code from the script. R responds. You get results. You can write two commands in a script. Run both simultaneously. R responds. You get results. This is the basic flow.

One way to do this is to use the default R Script, which is covered in the assignment guidelines. In your homework assignments, we will be asking you to submit code in another type of script: the R Markdown file. R Markdown allows you to create documents that serve as a neat record of your analysis. Think of it as a word document file, but instead of sentences in an essay, you are writing code for a data analysis.

When going through lab guides, I would recommend not copying and pasting code directly into the R Console, but saving and running it in an R Markdown file. This will give you good practice in the R Markdown environment. Now is a good time to read through the class assignment guidelines as they go through the basics of R Markdown files.

To open an R Markdown file, click on File at the top menu in RStudio, select New File, and then R Markdown. A window should pop up. In that window, for title, put in "Lab 1". For author, put your name. Leave the HTML radio button clicked, and select OK. A new R Markdown file should pop up in the top left window.

Don't change anything inside the YAML (the stuff at the top in between the —). Also keep the grey chunk after the YAML.

Delete everything else. Save this file (File -> Save) in an appropriate folder. It's best to set up a clean and efficient file management structure as described in the assignment guidelines. For example, below is where I would save this file on my Mac laptop (I named the file "Lab 1").

This is what file organization looks like

Follow the directions in the assignment guidelines to add this lab's code in your Lab

1. R Markdown file. Then knit it as an html, word or pdf file. You don't have to turn in the Rmd and its knitted file, but it's good practice to create an Rmd file for each lab.

Although the lab guides and course textbooks should get you through a lot of the functions that are needed to successfully accomplish tasks for this class, there are a number of useful online resources on R and RStudio that you can look into if you get stuck or want to learn more. We outline these resources below