

STA130 - Class #3: How R You?

Nathan Taback

2018-01-22

Today's Class

- RStudio user interface
- R Objects
- R Functions
- R Scripts
- R Packages
- R Lists
- R Notation
- R Missing Data
- dplyr

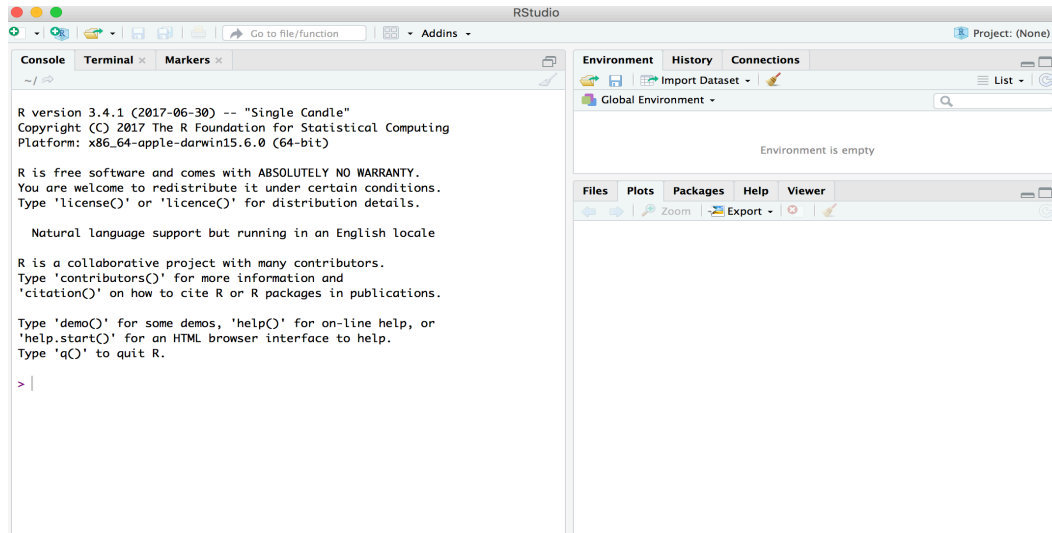
Announcements

- Tutorial grades will be assigned according to the following marking scheme.

	Mark
Attendance for the entire tutorial	1
Assigned homework completion ^a	1
In-class exercises	4
Total	6

- You will learn about the mentorship program in this week's tutorial (3% of final grade).

RStudio User Interface



R Objects

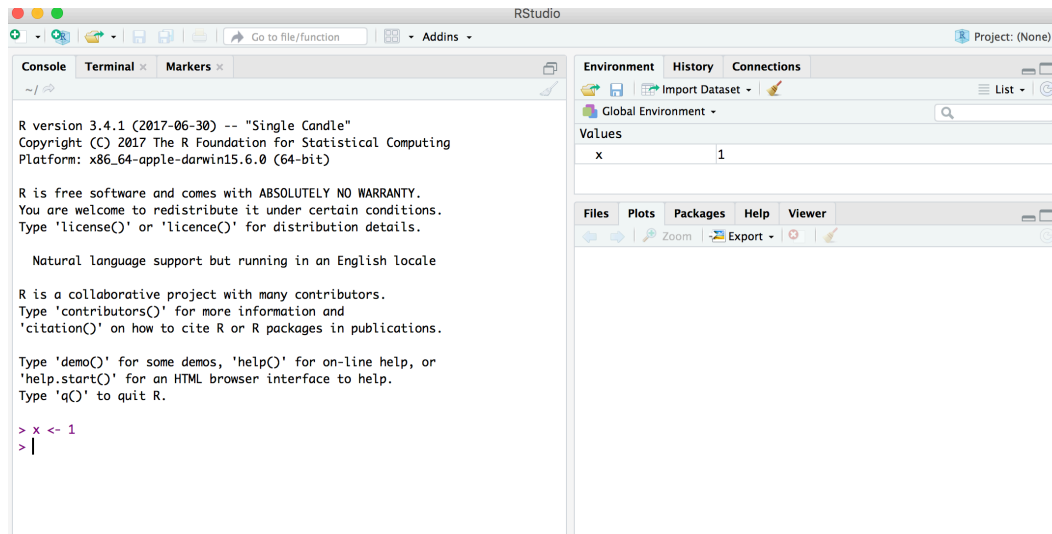
- R lets you save data by storing it inside an R object.
- What's an object? Just a name that you can use to call up stored data.

```
x <- 1  
x
```

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

Environment Pane in RStudio

- When you create an object, the object will appear in the environment pane of RStudio.



Functions

- R comes with many functions that you can use to do sophisticated tasks like random sampling.
- For example, you can round a number with the round function `round()`, or calculate its absolute value with `abs()`.
- Write the name of the function and then the data you want the function to operate on in parentheses:

```
round(-2.718282, 2)
```

```
## [1] -2.72
```

```
abs(-5)
```

```
## [1] 5
```

```
abs(round(-2.718282, 2))
```

```
## [1] 2.72
```

Function Constructor

- Every function in R has three basic parts: a name, a body of code, and a set of arguments.
- To make your own function, you need to replicate these parts and store them in an R object, which you can do with the `function` function.
- To do this, call `function()` and follow it with a pair of braces, `{}`: `my_function <- function() {}`

Function Constructor

- We can simulate rolling a pair of dice and adding the result with the code:

```
die <- 1:6  
dice <- sample(die, size = 2, replace = TRUE)  
sum(dice)
```

```
## [1] 9
```

Function Constructor

- We can create our own function with

```
roll <- function() {  
  die <- 1:6  
  dice <- sample(die, size = 2, replace = TRUE)  
  sum(dice)  
}
```

Call the function `roll()`

```
roll() # call the function. NB: result will differ with every call
```

```
## [1] 4
```

Function Arguments

Instead of rolling one die consider rolling four or ten dice then adding the results of all the rolls together.

```
roll2 <- function(numrolls) { # x is the argument of the function roll2
  die <- 1:6
  dice <- sample(die, size = numrolls, replace = TRUE) # the size of the sample
  sum(dice) # add up the roll results
}
```

`numrolls` is called an *argument* of the function `roll2()`.

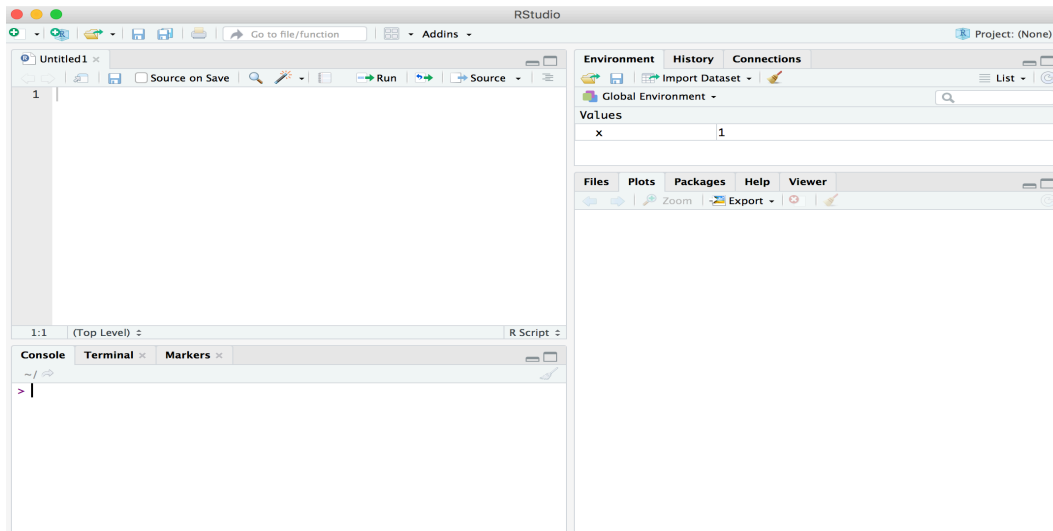
Let's simulate rolling ten dice and adding the results together.

```
roll2(10)
```

```
## [1] 40
```

Scripts

- If we want to edit the function `roll2()` then we will want to save it in a script.
- To do this in RStudio File > New File > R script in the menu bar.

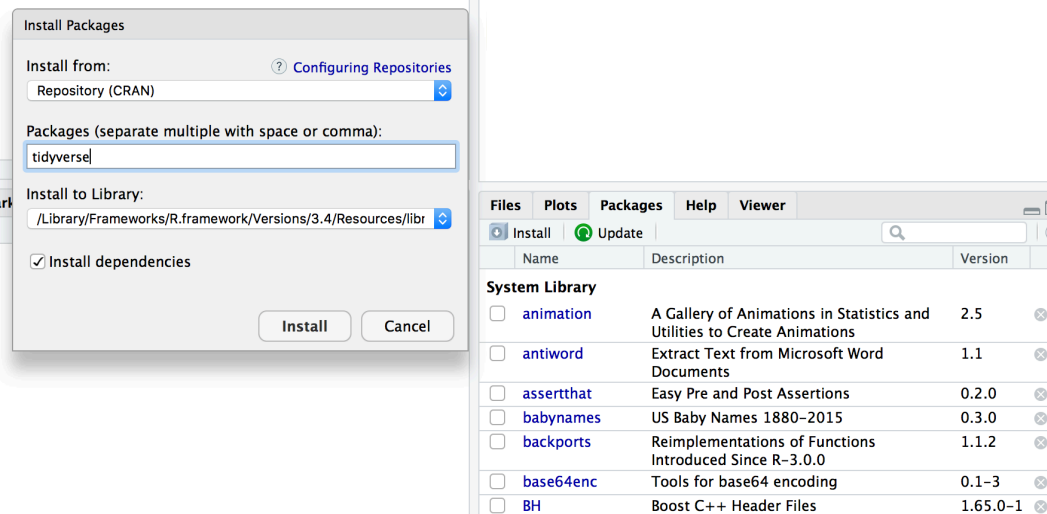


Packages

- You're not the only person writing your own functions with R.
- Many professors, programmers, and statisticians use R to design tools that can help people analyze data.
- They then make these tools free for anyone to use.
- To use these tools, you just have to download them. They come as preassembled collections of functions and objects called packages.
- We have already used two packages `ggplot2` and `dplyr`.

Packages

To install the package `tidyverse` in RStudio go to the Packages tab in RStudio and click Install.

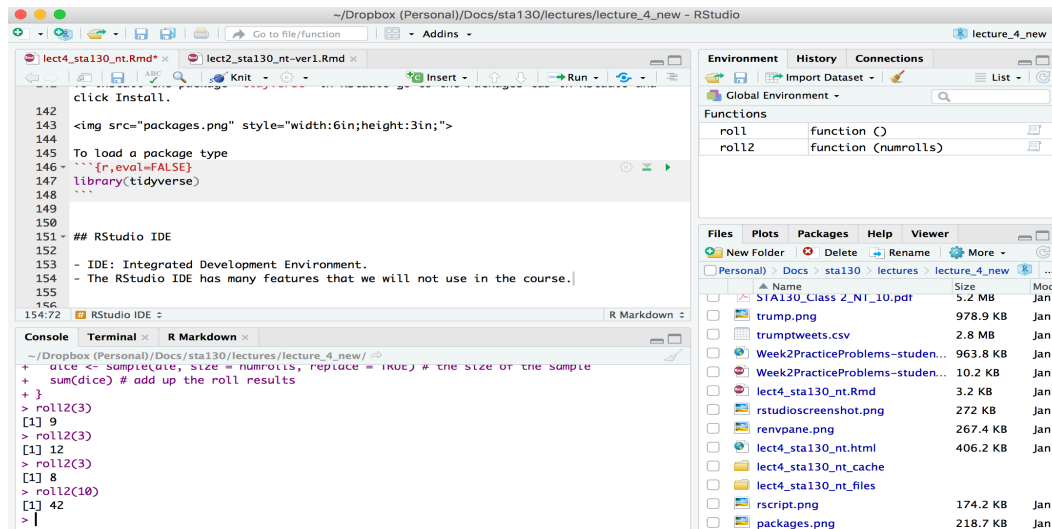


To load a package type

```
library(tidyverse)
```

RStudio IDE

- IDE: Integrated Development Environment.
- The RStudio IDE has many features that we will not use in the course.



- The **console** is where you can type an R command at the prompt and the result is returned.
- Write code in an R script, R Markdown document, or R Notebook.
- Run a script or R chunks from an R Markdown or R Notebook by pushing the run button in the chunk.

R Objects

- R stores data in objects such as vectors, arrays, and matrices.
- In most applications we will usually load data from an external file.

R Objects - Atomic Vectors

You can make an atomic vector by grouping some values of data together with c:

```
die<-c(1,2,3,4,5,6)
```

```
die
```

```
## [1] 1 2 3 4 5 6
```

```
is.vector(die)
```

```
## [1] TRUE
```

```
length(die)
```

```
## [1] 6
```

R Objects - Atomic Vectors

You can also make an atomic vector with just one value. R saves single values as an atomic vector of length 1:

```
two <- 2  
two
```

```
## [1] 2
```

R Objects - Atomic Vectors: Integer and Character

- Each atomic vector can only store one type of data. You can save different types of data in R by using different types of atomic vectors.
- R recognizes six basic types of atomic vectors: doubles, integers, characters, logicals, complex, and raw.
- We will not be using complex or raw types in STA130.
- Integer vectors included a capital L with input, and character vectors have input surrounded by quotation marks.

R Objects - Atomic Vectors: Integer and Character

```
mynums <- c(2L, 3L)
courses <- "STA130"
courses <- c("STA130", "MAT137")
sum(mynums)
```

```
## [1] 5
```

```
sum(courses)
```

```
## Error in sum(courses): invalid 'type' (character) of argument
```

```
sum(courses == "STA130")
```

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

R Objects - Double Vectors

- A double vector stores real numbers. Doubles are often called numerics.

```
die <- c(1,2,3,4,5,6)  
typeof(die)
```

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

R Objects - Logical Vectors

- Logical vectors store TRUES and FALSEs, R's form of Boolean data. Logicals are very helpful for doing things like comparisons:

```
3 > 4
```

```
## [1] FALSE
```

- TRUE or FALSE in capital letters (without quotation marks) will be treated as logical data. R also assumes that T and F are shorthand for TRUE and FALSE.

```
logic <- c(TRUE, FALSE, TRUE)  
logic
```

```
## [1] TRUE FALSE TRUE
```

R Objects - Atomic Vectors: `dim()`

You can transform an atomic vector into an n-dimensional array by giving it a dimensions attribute with `dim`.

```
die <- c(1,2,3,4,5,6)
dim(die) <- c(2,3) # a 2x3 matrix
die
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
die <- c(1,2,3,4,5,6)
dim(die) <- c(3,2) # a 3x2 matrix
die
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

R always fills up each matrix by columns, instead of by rows unless you use `matrix()` or `array()`.

Factors

- Factors are R's way of storing categorical information, like ethnicity or eye color.
- A factor as something like sex since it can only have certain values.
- Factors very useful for recording the treatment levels of a categorical variable.

```
sex <- factor(c("male", "female", "female", "male"))
typeof(sex)
```

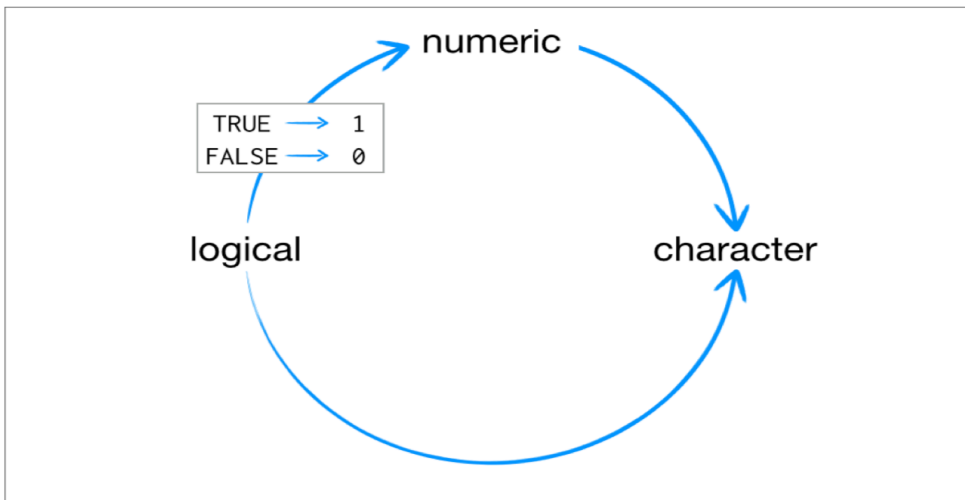
```
## [1] "integer"
```

```
unclass(sex) # shows how R is storing the factor vector
```

```
## [1] 2 1 1 2
## attr(,"levels")
## [1] "female" "male"
```


Coercion

R always follows the same rules when it coerces data types. Once you are familiar with these rules, you can use R's coercion behavior to do surprisingly useful things.



For example `sum(c(TRUE, TRUE, FALSE, FALSE))` will become `sum(c(1, 1, 0, 0))`.

```
sum(c(TRUE, TRUE, FALSE, FALSE))
```

```
## [1] 2
```

Lists

- Lists are like atomic vectors because they group data into a one-dimensional set.
- Lists do not group together individual values.
- Lists group together R objects, such as atomic vectors and other lists.
- For example, you can make a list that contains a numeric vector of length 31 in its first element, a character vector of length 1 in its second element, and a new list of length 2 in its third element.

```
list1 <- list(1:31, "Prof. Taback", list(TRUE, FALSE))
list1

## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31
##
## [[2]]
## [1] "Prof. Taback"
##
## [[3]]
## [[3]][[1]]
## [1] TRUE
##
## [[3]][[2]]
## [1] FALSE
```

Data Frames

- Data frames are the two-dimensional version of a list.
- They are the most useful storage structure for data analysis
- A data frame is R's equivalent to the Excel spreadsheet because it stores data in a similar format.

Data Frames

- Data frames group vectors together into a two-dimensional table.
- Each vector becomes a column in the table.
- As a result, each column of a data frame can contain a different type of data; but within a column, every cell must be the same type of data.

data frame	1	"R"	TRUE
	2	"S"	FALSE
	3	"T"	TRUE
	numeric	character	logical

Data Frames

```
student_num <- c(1, 2, 3, 4)
name <- c("Nadia", "Shiyi", "Yizhe", "Wei")
mydat <- data.frame(obsnum = student_num, student_name = name)
mydat
```

```
##  obsnum student_name
## 1      1      Nadia
## 2      2      Shiyi
## 3      3      Yizhe
## 4      4       Wei
```

- Creating a data frame by hand takes a lot of typing, but you can do it with the `data.frame()` function.
- Give `data.frame()` any number of vectors, each separated with a comma.
- Each vector should be set equal to a name that describes the vector.
- `data.frame()` will turn each vector into a column of the new data frame.

Data Frames

You can view a data frame in RStudio by clicking on the data frame name in the Environment tab

The screenshot shows the RStudio interface. The Environment tab on the right displays the data frame 'mydat' with 4 observations and 2 variables. The Console on the left shows the R code used to create the data frame.

obsnum	student_name
1	Nadia
2	Shiyi
3	Yizhe
4	Wei

```
~/Dropbox (Personal)/Docs/sta130/lectures/lecture_4_new/ > view(mydat)
> student_num <- c(1, 2, 3, 4)
> name <- c("Nadia", "Shiyi", "Yizhe", "Wei")
> mydat <- data.frame(student_num, name)
> mydat
> View(mydat)
> student_num <- c(1, 2, 3, 4)
> name <- c("Nadia", "Shiyi", "Yizhe", "Wei")
> mydat <- data.frame(obsnum = student_num, student_name = name)
> mydat
> View(mydat)
> |
```

R Notation - [,]

- To extract a value or set of values from a data frame, write the data frame's name followed by a pair of square brackets with a comma [,].

```
mydat[ , ]
```

R Notation - [,]

```
mydat
```

```
##   obsnum student_name  
## 1     1         Nadia  
## 2     2         Shiyi  
## 3     3         Yizhe  
## 4     4           Wei
```

```
mydat[1,2] # the value in row 1 and column 2
```

```
## [1] Nadia  
## Levels: Nadia Shiyi Wei Yizhe
```

```
mydat[c(1,2),2] # all values in rows 1 and 2 in second column
```

```
## [1] Nadia Shiyi  
## Levels: Nadia Shiyi Wei Yizhe
```


R Notation - \$

The `$` tells R to return all of the values in a column as a vector.

```
mydat$student_name
```

```
## [1] Nadia Shiyi Yizhe Wei  
## Levels: Nadia Shiyi Wei Yizhe
```

```
vec <- mydat$student_name # assign it to vec  
attributes(vec) # info associated with object vec
```

```
## $levels  
## [1] "Nadia" "Shiyi" "Wei" "Yizhe"  
##  
## $class  
## [1] "factor"
```

```
vec[2] # get second element of vector
```

```
## [1] Shiyi  
## Levels: Nadia Shiyi Wei Yizhe
```

R Notation - combine [,] and \$

```
mydat[mydat$obsnum == 1,] # first row of data frame and all columns
```

```
##  obsnum student_name  
## 1      1      Nadia
```

```
mydat[mydat$obsnum == 1 | mydat$obsnum == 4 ,] # first and fourth rows of data frame and all columns
```

```
##  obsnum student_name  
## 1      1      Nadia  
## 4      4      Wei
```

Missing Data - **NA**

- Missing information problems happen frequently in data science.
- For example a value is missing because the measurement was lost, corrupted, or never recorded.
- The **NA** character is a special symbol in R. It stands for “not available” and can be used as a placeholder for missing information.

```
1 + NA
```

```
## [1] NA
```

Missing Data - `na.rm()`

- Suppose you collected the ages of five students, but you forgot to record the fifth student's age.

```
age <- c(19, 20, 17, 20, NA)
mean(age) # mean will be NA
```

```
## [1] NA
```

```
age <- c(19, 20, 17, 20, NA)
mean(age, na.rm = TRUE) # R will ignore missing values
```

```
## [1] 19
```

Identify and Set Missing Data - `is.na()`

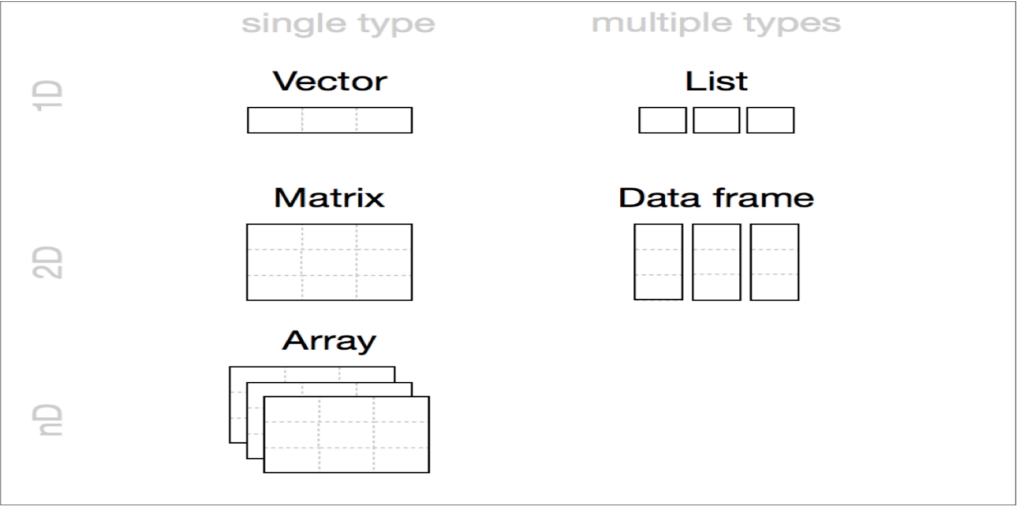
```
age <- c(19, 20, 17, 20, NA)
is.na(age) # check which elements of age are missing
```

```
## [1] FALSE FALSE FALSE FALSE TRUE
```

```
age[1] <- NA # set the first element of age to NA
age
```

```
## [1] NA 20 17 20 NA
```

Summary of R Data Structures



Tidyverse

Tidyverse



R packages for data science

The tidyverse is an opinionated **collection of R packages** designed for data science. All packages share an underlying philosophy and common APIs.

Install the complete tidyverse with:

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

<https://www.tidyverse.org>

Canadian Flu Rates with **dplyr**

The provincial rates for the week ending January 6, 2018 are in the file `fludat_prov.csv` and the the size of the population in each province is in the file `popdat.csv`. The code below reads the files into R data frames.

```
library(tidyverse)
fludat_prov <- read_csv("fludat_prov.csv") # import data from file
popdat <- read_csv("popdat.csv") # import data from file
```


Canadian Flu Rates with `dplyr`

```
head(fludat_prov) # head shows the first six rows of a data frame
```

```
## # A tibble: 6 x 3
##   prov          testpop_size fluA
##   <chr>          <int> <int>
## 1 Newfoundland      96    12
## 2 Prince Edward Island  64    11
## 3 Nova Scotia      144    23
## 4 New Brunswick    347    80
## 5 Province of Québec 6361  1190
## 6 Province of Ontario 2320   344
```

```
head(popdat)
```

```
## # A tibble: 6 x 3
##   prov          prov_pop_size region
##   <chr>          <int> <chr>
## 1 Nunavut      35944 Territories
## 2 Alberta    4067175 <NA>
## 3 Saskatchewan 1098352 West
## 4 Yukon       35874 Territories
## 5 Manitoba    1278365 West
## 6 British Columbia 4648055 West
```

Canadian Flu Rates with **dplyr**

How many Provinces/Territories are in the fludat_prov data frame?

```
fludat_prov %>% summarise(numprov = n()) # n() counts the number of rows in the data frame
```

```
## # A tibble: 1 x 1
##   numprov
##   <int>
## 1      13
```

Canadian Flu Rates with `dplyr`

Do any variables in `fludat` or `popdat` have missing values?

```
fludat_prov %>% filter(is.na(prov) == TRUE | is.na(testpop_size) == TRUE | is.na(fluA) == TRUE)
```

```
## # A tibble: 0 x 3
```

```
## # ... with 3 variables: prov <chr>, testpop_size <int>, fluA <int>
```

```
popdat %>% filter(is.na(prov) == TRUE | is.na(prov_pop_size) == TRUE | is.na(region) == TRUE)
```

```
## # A tibble: 2 x 3
```

```
##   prov      prov_pop_size region
```

```
##   <chr>          <int> <chr>
```

```
## 1 Alberta      4067175 <NA>
```

```
## 2 Quebec       8164361 <NA>
```

Canadian Flu Rates with **dplyr**

Recode specific values using R data frame notation [,] and \$.

```
popdat$region[popdat$prov == "Alberta"] <- "West" #recode only the region value for Alberta
popdat$region[popdat$prov == "Quebec"] <- "East" #recode only the region value for Alberta
popdat$region #print region variable in popdat data
```

```
## [1] "Territories" "West" "West" "Territories" "West"
## [6] "West" "East" "East" "Atlantic" "Atlantic"
## [11] "Territories" "Atlantic" "Atlantic"
```

Canadian Flu Rates with `dplyr` - Joining Two Tables with `inner_join()`

We can join two data frames with `inner_join(x,y)`: return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

```
fludat_prov %>% inner_join(popdat, by = "prov")
```

```
## # A tibble: 9 x 5
##   prov          testpop_size fluA prov_pop_size region
##   <chr>          <int> <int>      <int> <chr>
## 1 Newfoundland      96    12      519716 Atlantic
## 2 Prince Edward Island 64    11      142907 Atlantic
## 3 Nova Scotia      144    23      923598 Atlantic
## 4 New Brunswick    347    80      747101 Atlantic
## 5 Manitoba         849   186     1278365 West
## 6 British Columbia 1078   198     4648055 West
## 7 Yukon            15     1       35874 Territories
## 8 Northwest Territories 28    10       41786 Territories
## 9 Nunavut          18     1       35944 Territories
```

Why are there only 9 observations when there are 13 Provinces/Territories?

Canadian Flu Rates with `dplyr` - Joining Two Tables with `inner_join()`

```
fludat_prov$prov
```

```
## [1] "Newfoundland"           "Prince Edward Island"  
## [3] "Nova Scotia"           "New Brunswick"  
## [5] "Province of Québec"    "Province of Ontario"  
## [7] "Manitoba"              "Province of Saskatchewan"  
## [9] "Province of Alberta"    "British Columbia"  
## [11] "Yukon"                 "Northwest Territories"  
## [13] "Nunavut"
```

```
popdat$prov
```

```
## [1] "Nunavut"                "Alberta"  
## [3] "Saskatchewan"          "Yukon"  
## [5] "Manitoba"              "British Columbia"  
## [7] "Ontario"               "Quebec"  
## [9] "Prince Edward Island"  "Newfoundland"  
## [11] "Northwest Territories" "Nova Scotia"  
## [13] "New Brunswick"
```

Province needs to be recoded. Exercise on this week's practice problems.

Canadian Flu Rates with `dplyr` - Joining Two Tables with `inner_join()`

X			Y		
A	B	C	A	B	D
a	t	1	a	t	3
b	u	2	b	u	2
c	v	3	d	w	1

`inner_join(X, Y)`

A	B	C	D
a	t	1	3
b	u	2	2