

ANALYZING SEX NOW DATA IN R-STUDIO

Kiffer G. Card, PhD
Kirk J. Hepburn, MPP

Analyzing *Sex Now* Data in R Studio:

An Introduction to Basic Descriptive Analyses

Table of Contents

Before you Begin	3
1. Getting Started with R	4
1.1 R Basics	4
1.1.1 About R	4
1.1.2 Install R and RStudio	4
To Install R	4
To Install RStudio	4
1.1.3 A quick tour of RStudio	4
1.1.4 Getting help	5
1.1.5 R Packages	5
1.1.6 Entering Data into R	6
1.1.7 R Syntax	7
1.1.8 Annotating Your R Code	7
1.2 Exploring and Summarizing Data	8
1.2.1 Understanding Your Data	8
1.2.2 Data Dictionaries & Skip Patterns	9
1.2.3 Using R to Look at Your Data	9
1.3 Subsetting Your Data	10
1.4 Recoding Variables	10
1.4.1 Recoding Categorical Variables	10
1.4.2 Categorizing Continuous Data	12
1.5 Descriptive Statistics	13
1.5.1 Frequencies	13
1.5.2 Cross Tabulations	13
1.5.3 Summaries of Numeric Variables	15
1.6 Quick Overview	17

Before you Begin

This manual will provide a rapid overview of the content you will need to conduct basic analyses in R. I strongly encourage you to also explore a general introductory course in R. Below are two tutorials which offer free trials for new users:

- Coursera's R Programming - <https://www.coursera.org/learn/r-programming>
- Paul Barton's R Statistics Essential Training - <https://www.lynda.com/R-tutorials/R-Statistics-Essential-Training/142447-2.html>

1. Getting Started with R

In this manual, we will introduce R and explore how it can be used to summarize and data with a focus on the Sex Now 2018 dataset. In this guide, we often provide the simplest methods for meeting each objective, however, in R there are numerous ways to complete given tasks and you are welcome to explore and identify strategies that work better for you.

1.1 R Basics

1.1.1 About R

R is a statistical language commonly accessed through the integrated development environment called RStudio. Base R performs a vast number of useful statistical operations, but it can be enhanced with packages. These packages are open source and created by the community of R users, making these resources open for public use free of charge.

1.1.2 Install R and RStudio

Install R and then install RStudio using the following steps:

To Install R

- Open an internet browser and go to www.r-project.org.
- Click the "download R" link in the middle of the page under "Getting Started."
- Select a CRAN location (a mirror site) and click the corresponding link.
- Click on the "Download R for (Mac) OS X" link at the top of the page.
- Click on the file containing the latest version of R under "Files."
- Save the pkg file, double-click it to open, and follow the installation instructions.
- Now that R is installed, you need to download and install RStudio.

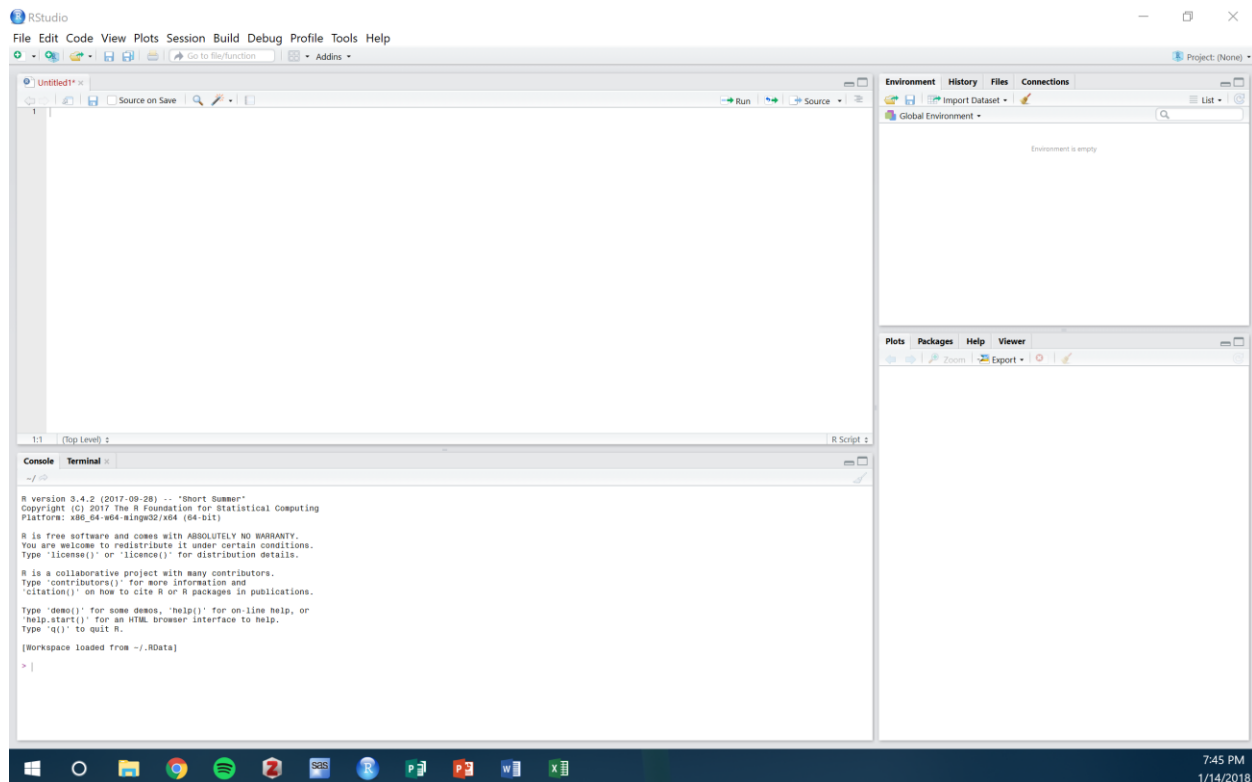
To Install RStudio

- Go to www.rstudio.com and click on the "Download RStudio" button.
- Click on "Download RStudio Desktop."
- Click on the version recommended for your system, or for the latest Mac version, save the .dmg file on your computer, double-click it to open, and then drag and drop it to your applications folder.

1.1.3 A quick tour of RStudio

There are several important windows to become familiar with when working in RStudio:

- The top left window is a script editor. Here you can edit and execute R code (by highlighting and hitting ctrl+Enter).
- The bottom left is the R console, where the R engine does its work. You're your code is executed, with results and error notifications. You can also enter ad hoc commands in this window, but these commands will not be saved.



- In the top right, R objects are displayed and summarized. Objects are named sets of data, whether a set of data from a survey, a list of names, a set of randomly-generated numbers, or even functions you have defined (created) yourself.
- The bottom right contains several useful tools: a file explorer, a graphical device for displaying any plots you create (histograms, bar charts, and the like), a list of the packages you have installed, a viewer for function help documentation, and a viewer for web content.

1.1.4 Getting help

Specific functions are described fully in R. Getting help on the function `mean`, for instance, is as simple as entering `?mean` into the console. The documentation for the `mean` function will appear in the “Help” window in Rstudio. You can also click on a function that you have typed in your script editor and press F1, which will also open the “Help” window.

Furthermore, if you have questions about a specific function or error, the R community is extremely active. Your question has likely already been asked on StackExchange or similar websites (e.g., <https://www.r-bloggers.com/>, <https://www.statmethods.net/>).

1.1.5 R Packages

As mentioned, R comes ready with a large set of useful functions, but the R community has created many new functions which perform more complex or specialized tasks or which simplify the language around basic R tasks. For the latter purpose, I highly recommend installing and familiarizing yourself with the Tidyverse family of packages, particularly `plyr`, `dplyr`, `ggplot2`, and `readr`. These simplify and extend data reading, cleaning, and

plotting. You will need these packages to perform the tasks I demonstrate below. Install them from RStudio under Tools > Install Packages... Input the list of desired packages: `plyr` `tidyverse`. Hit OK and RStudio will download and install the packages from CRAN. You can also download the packages from the R console:

```
install.packages(pkgs = c("plyr", "tidyverse"))
```

Finally, to activate all of the functions in each package, enter the following in the script editor or R console:

```
library(plyr) library(tidyverse)
```

This will load all of the functions contained in these packages into your R session; without doing this, R will not have access to many of the functions used below.

1.1.6 Entering Data into R

Before we begin using R to explore data, we must read that data into the R environment. Anything stored in the R environment's memory is called an *object*. To assign data to an object, we use the `<-` syntax.

The data we will be using is from the 2018 Sex Now Survey, which is collected and managed by the Community-based Research Centre. Much of the data that you encounter, including those from the 2018 Sex Now Survey, will be stored in comma-separated values or other text-based formats. As such, you can use the `read.csv` command to load the data into R.

Note: There are a few things you need to know about loading data into R. First, the text in green must be replaced by the path to where the file is stored on your computer. To find this you can navigate to the data file using your system browser, press shift, right click on the file, and select "copy as path." You can also manually type in the file name. In my case, the data is stored on a network drive called the Z drive. Second, you must add a backslash to each \ character. This is important both when reading and writing data in R.

```
sexnow2018 <- read.csv("Z:\\Kiffer\\SexNow2018\\SN2018.csv")
```

"sexnow2018" is the name we have given to our new object; the `<-` symbol is like an arrow which tells R "Assign the results of this function to this object." Therefore, whatever the `read.csv` function manages to spit out (but only if it manages to spit something out) will be stored in an object named "sexnow2018". After you run this line of code, the following should appear in your object explorer.



You now have the Sex Now 2018 data loaded into R! Now, when you want to recall an object within the `sexnow2018` object, you will need to understand a bit about how syntax works in R.

1.1.7 R Syntax

For an object that is a dataset, R syntax allows you to reference parts (or variables) of that object using the `$` syntax. The `$` separates out the dataset from the variable in the dataset.

R syntax also utilizes functions which provide the means to analyze an object. R syntax for functions almost universally follows this form: `function(arguments)`, where the function is the command and arguments are the details of that command. For instance, the `mean` function can take the following arguments, each of which is described in the functions help file (which can be accessed as described earlier):

<code>x</code>	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for <code>trim = 0</code> , only.
<code>trim</code>	the fraction (0 to 0.5) of observations to be trimmed from each end of <code>x</code> before the mean is computed. Values of <code>trim</code> outside that range are taken as the nearest endpoint.
<code>na.rm</code>	a logical value indicating whether NA values should be stripped before the computation proceeds.
<code>...</code>	further arguments passed to or from other methods.

R takes the arguments in order, meaning that you can provide it with the arguments without naming them and R will simply assume what each argument represents. I find it less confusing if I go ahead and name each argument. For instance:

```
mean(x = sexnow2018$Q11_age, trim = 0, na.rm = TRUE)
```

Here I have told R to calculate the average age of a men in the SexNow2018 dataset. I have not trimmed any observations, nor have I removed missing values. Because the `trim` and `na.rm` arguments have defaults, I do not need to provide those arguments. I can achieve the same result by inputting

```
mean(x = sexnow2018$Q11_age)
```

1.1.8 Annotating Your R Code

R has a comment flag, the hashtag (`#`). Any text on a line of code beginning with a `#` will not be read by the R engine. This allows you to explain what your code is trying to accomplish for your future self or for other readers of your code. It is vital that you comment your code, and it is best that you comment as you go. You will thank yourself time and again for developing this habit.

```
#Calculate Mean for Age for participants  
mean(x = sexnow2018$Q11_age, trim = 0, na.rm = TRUE)
```


1.2 Exploring and Summarizing Data

1.2.1 Understanding Your Data

Before exploring the data for our own purposes, it will help to understand what R thinks of the data. Run this line of code:

```
class(sexnow2018)

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

The code will give you a result with a length of 1, which tells us that the object `sexnow2018` is stored as a data frame.

In R, there are some basic object classes: dataframes, logical, numeric, factor, character, and lists. Logical data can take either TRUE, FALSE, or NA (the latter being the value for missing data in R). Numeric data can take any real number. Character data can take characters or strings of characters or even multiple strings of characters. Factor data behave best for categorical variables, as each factor datum fits into a category to which a number is also assigned. (Much more on this later.) Lists are objects which can hold other objects, i.e. “lists” of objects. These objects can be of any class (including other lists) and be any length. I like to think of these as “folders” similar to your computer’s folder system (though this is not entirely accurate).

Data frames are similar to lists in that they can hold objects of different classes, but all of the objects in a data frame must be the same length. If this sounds at all familiar, it is because data frames a way to look at data like a spreadsheet. Click on the “sexnow2018” name in your object explorer. This calls RStudio’s `View` function on your data, which opens a viewer like this:

The screenshot shows the RStudio 'View' window for the 'sexnow2018' data frame. The window title is 'Analyzing Sex Now Data in R.R' and the file name is 'sexnow2018'. The data is displayed in a table with columns: X, Q2_ethnicity_african, Q2_ethnicity_arab, Q2_ethnicity_asian, Q2_ethnicity_indigenous, Q2_ethnicity_latin, Q2_ethnicity_south_asian, Q2_ethnicity_white, Q7_sex_orientation_gay, Q7_sex_orientation_asexual, and Q7_se. The rows are numbered 1 through 20, with the last row (20) showing '9999: True Missing' for the last two columns.

X	Q2_ethnicity_african	Q2_ethnicity_arab	Q2_ethnicity_asian	Q2_ethnicity_indigenous	Q2_ethnicity_latin	Q2_ethnicity_south_asian	Q2_ethnicity_white	Q7_sex_orientation_gay	Q7_sex_orientation_asexual	Q7_se
1	No	No	No	No	No	No	Yes	Yes	No	Nc
2	No	No	No	No	No	No	Yes	Yes	No	Nc
3	No	No	No	No	No	No	Yes	No	No	Nc
4	No	No	No	No	No	No	Yes	No	No	Nc
5	No	No	No	No	No	No	Yes	Yes	No	Nc
6	No	No	No	No	No	No	Yes	Yes	No	Nc
7	No	No	No	No	Yes	No	Yes	No	No	Nc
8	No	No	No	No	No	No	Yes	Yes	No	Nc
9	No	Yes	No	No	No	No	No	Yes	No	Nc
10	No	No	No	No	No	No	Yes	Yes	No	Nc
11	No	No	No	No	No	No	Yes	No	No	Nc
12	No	No	No	No	No	Yes	No	Yes	Yes	Yes
13	No	No	No	No	No	No	Yes	No	No	Nc
14	No	No	No	No	Yes	No	No	No	No	Nc
15	No	No	No	No	No	No	Yes	Yes	No	Nc
16	No	No	No	No	No	No	Yes	Yes	No	Nc
17	No	No	No	Yes	No	No	No	No	No	Nc
18	No	No	No	No	No	No	Yes	No	No	Nc
19	No	No	No	No	No	No	Yes	No	No	Nc
20	No	No	No	No	No	No	Yes	9999: True Missing	9999: True Missing	99

As in a spreadsheet, each row represents a case or observation, while each column represents a variable (since this is from a survey). Always remember, though, that to R this data frame is basically a list of objects, with each column being an object. They each have their own class. Try the code on the next page:

```
class(sexnow2018$Q11_age)
## [1] "integer"
```

You should get “integer” as the response. While `sexnow2018` has the “data.frame” class, `sexnow2018$Q11_age` has the “integer” class. This is important as not all of these variables should be treated the same. For instance, `Q15_money` is stored as a “factor” because the question is categorical, rather than numeric. Each variable should be assigned the appropriate variable type. If you find that your data has been read in incorrectly, you can reassign the variable to a new type using the functions `as.numeric` (which converts variables to numeric), `as.factor` (which converts variables to categorical), or `as.character` (which converts variables to text). For example, the following code converts age from an integer to a numeric variable.

```
sexnow2018$Q11_age <- as.numeric(sexnow2018$Q11_age)
```

If you re-run the `class` function statement on your `Q11_age` variable, you will find that the class is now “numeric”.

1.2.2 Data Dictionaries & Skip Patterns

Before you begin running analyses, it’s important that we understand what our output means. In order to gain insight on this, we must consult the data dictionary. It provides a summary of the dataset, its purpose, and a variable-by-variable description. You should have been provided a data dictionary with the Sex Now dataset provided to you by the CBRC.

The data dictionary lists the variable name, the text of the question, the response options or recoded categories, and the descriptive frequencies for the overall sample. Take for instance the variable “`Q50_PrEP_ever_used`” The question asked for that question was: “Have you EVER used PrEP?” to which respondents could have chosen “No,” “Yes, Previously” or “Yes, Currently.” One response has also been coded as “7777: Poor Data Quality.” Poor data quality is often the indicated value when a participant reported contradictory responses – either by providing conflicting answers or by selecting two or more answers on the same question. Some participants were also coded as “8888: HIV-Positive.” The 8888 code is used in the survey to indicate when skip logic from a previous question exists. Finally, some responses are coded “9999: True Missing.” These responses are missing. For that question. Generally speaking, you will want to recode 9999 and 8888 responses as “NA” and exclude observations with missing observations from your analysis. We will discuss later how to recode variables and exclude observations later in this text.

1.2.3 Using R to Look at Your Data

While the data dictionary gives us some information on each variable, we can use R to explore it more deeply. For continuous data, the `summary` function is frequently used.

Let's take a look at the age variable, Q11_age.

```
summary(sexnow2018$Q11_age)
```

```
##      Min.      1st Qu.  Median    Mean    3rd Qu.    Max.     NA's
##      16.0      26.0      32.0     35.76    44.0      82.0      47
```

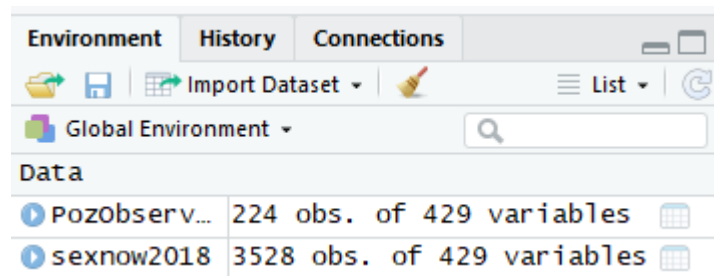
This will give you a few summary statistics about your data: quartiles, the median, mean, minimum, maximum, and the number of missing responses. With that, we have a sense of our age distribution!

For categorical data you can use the `table` function. If you are only interested in identifying the levels in a variable without seeing the response frequencies, you can use the `levels` function. Try using these functions to explore the Q50_PrEP_ever_used variable by examining the help documentation in R (see above for how to open help files). If you can't figure it out, try searching online in google by searching "levels function in R" or "table function in R." We will explore these functions later as well, but it is good to get used to using the help documentation in R and seeking out help online.

1.3 Subsetting Your Data

Manipulating datasets is another essential skill needed for data analysis. For instance, you might want to only select a subset of observations, dropping those which you are not interested in. Often this is done based on participant responses to a specific question. Perhaps the easiest way to do this is to use the 'which' function. For example, in the code below we create a subset dataset which contains records for individuals who said that they had been diagnosed with HIV (i.e., Q43_HIV_ever_diagnosed = "Yes").

```
# Create a subset of the data containing only men who reported having HIV
PozObservations <- sexnow2018[which(sexnow2018$Q43_HIV_ever_diagnosed == 'Yes'), ]
```



Environment		History	Connections
Global Environment			
Data			
PozObserv...	224 obs. of 429 variables		
sexnow2018	3528 obs. of 429 variables		

1.4 Recoding Variables

1.4.1 Recoding Categorical Variables

In addition to merging data sets, you will also find yourself wanting to edit variables. This is often the case when you want to recode variables as missing, when the categories should be collapsed for theoretical reasons, or when sub-groups are not large enough to perform a specific analysis or when your research question involves a specific dichotomy. When

doing this I suggest always creating a new variable, rather than writing over your existing data. In the example below, we collapse the question about lifetime PrEP use variable into “Yes” vs. “No.”

The following code does several things. The first line of code creates a new object in the data frame called `PrEPUse_Ever` and assigns it missing values. Next, we indicate that if “Yes, Currently” was selected, then the new variable value will be set to “Ever”. Likewise, the third line of code sets the “Yes, Previously” response to the original variable as “Ever” in the new variable. The fourth line of code fills in observations with the level “Never” for all observations where the original response was “No.” Finally, any observations that are still blank in the new variable are set to missing (NA).

```
# Create a new variable based on whether respondent reported ever using PrEP
sexnow2018$PrEPUse_Ever <- NA

sexnow2018$PrEPUse_Ever[sexnow2018$Q50_PrEP_ever_used == "Yes, Currently"] <-
"Ever"

sexnow2018$PrEPUse_Ever[sexnow2018$Q50_PrEP_ever_used == "Yes, Previously"] <-
"Ever"

sexnow2018$PrEPUse_Ever[sexnow2018$Q50_PrEP_ever_used == "No"] <- "Never"

sexnow2018$PrEPUse_Ever[is.na(sexnow2018$PrEPUse_Ever)] <- NA
```

You can now use the `table` function to see the distribution of the new variable.

```
# Examine frequency of new variable
table(sexnow2018$PrEPUse_Ever)
```

You should also create a cross tab of the new variable to ensure that it was coded as you would like. To do so, simply put both variables in your table statement, separated by a comma. I also specify the `useNA = "ifany"` argument so that I can see where the newly created missing variables came from.

```
# Examine Cross tabulation between new and old variable versions
table(sexnow2018$Q50_PrEP_ever_used, sexnow2018$PrEPUse_Ever, useNA = "ifany"
)
```

```
##               Ever Never <NA>
## 7777: Poor Data Quality    0    0    1
## 8888: HIV-Positive        0    0  224
## 9999: True Missing        0    0  163
## No                      0 2650    0
## Yes, Currently          395    0    0
## Yes, Previously         95    0    0
```

As you can see the data appears to be coded as I desired, with the last two response options being coded as “Ever”; the “No” option being recorded as “Never”; and the missing observations from 7777, 8888, and 9999, being recorded as NA.

1.4.2 Categorizing Continuous Data

A similar strategy can be used to collapse a continuous variable into a categorical variable. Take for instance the example below where we create a new variable (`age_groups`) in the `sexnow2018` dataset with four age groups representing the age of individuals when they took the survey (`Q11_age`). Here we can take advantage of R’s factor class. Factors are especially useful for categorical data. While they behave a bit like string objects, they are, in fact, objects which can take any of a set of “levels”, or categories. Each category also has a number behind it. Here we create three categories by first assigning characters based on a person’s smoking initiation age, then coercing the object into a factor object.

```
# Create an empty column
sexnow2018$age_groups <- NA

# Collapse Ages 16 to 18 -- 1
sexnow2018$age_groups[sexnow2018$Q11_age <= 18] <- "16-18"
# Collapse Ages 19 to 29 -- 2
sexnow2018$age_groups[sexnow2018$Q11_age >= 19 & sexnow2018$Q11_age <= 29] <-
"19-29"
# Collapse Ages 30 to 58 -- 3
sexnow2018$age_groups[sexnow2018$Q11_age >= 30 & sexnow2018$Q11_age <= 59] <-
"30-59"

# Collapse Ages 31 to 58 -- 3
sexnow2018$age_groups[sexnow2018$SMD030 >= 60] <- "60+"

sexnow2018$age_groups <- as.factor(sexnow2018$age_groups)
summary(sexnow2018$age_groups)

## 16-18  19-29  30-59    NA's
## 93      1333   1827     275
```

The data dictionary tells us that 7777 and 9999 are not valid ages. Since these were not included in the conditions we wrote, they have been left as NA.

Also note that I used the `&` (AND) symbol to specify two conditions in one line of code. Other symbols can be used in a single line of code to allow for more complex logic. For instance, the `!` symbol means “Not” so `!=` would mean “not equal to” while `==` means “equal to”. The `|` symbol means “OR.” You can bracket multiple logical operators together to allow for multilayered logic. You can experiment with these and then check your results with cross tabulations to see how they behave.

Factors can be tricky. It is vital to remember that 1) they don’t like having levels added once created 2) Each variable is really a number hiding behind the category: if you call `as.numeric` on the object, “16-18” would be turned into a “1”, not “17” or an error. 3) The

lowest category is the reference level; in certain types of analysis, all other categories may be compared to that one.

1.5 Descriptive Statistics

1.5.1 Frequencies

Now that you've had some exposure to creating and working with variables, let's examine how we can summarize our data to create descriptive statistics. To begin, let's look at the help file for the `table` function. Its first argument says, "... one or more objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted." "... means "any number of items".

Our province variable is an object which can be interpreted as a factor, i.e. a categorical variable:

```
table(sexnow2018$Q1_province)
```

## 7777: Poor Data Quality	9999: True Missing	Alberta
## 1	26	580
## British Columbia	I don't live in Canada	Manitoba
## 760	7	182
## New Brunswick	Newfoundland & Labrador	Northwest Territories
## 5	1	1
## Nova Scotia	Nunavut	Ontario
## 183	3	1255
## Quebec	Saskatchewan	
## 520	4	

This will give you a quick frequency table, with the same values as the data dictionary. Note that the `$` symbol is used in R to specify that you want to only view a part of the data (i.e., the province variable). The `$` symbol usually only works on data frames and lists.

1.5.2 Cross Tabulations

`table` also allows us to perform cross tabulations, as demonstrated earlier. All we need to do is provide another "object which can be interpreted as a factor". Let's try the newly created PrEP Use variable.

```
table(sexnow2018$Q1_province, sexnow2018$PrEPUse_Ever)
```

	Ever	Never
7777: Poor Data Quality	0	1
9999: True Missing	2	9
Alberta	43	501
British Columbia	125	530
I don't live in Canada	0	6
Manitoba	12	153
New Brunswick	0	4
Newfoundland & Labrador	0	1
Northwest Territories	0	1
Nova Scotia	19	152
Nunavut	1	2
Ontario	184	946
Quebec	104	341
Saskatchewan	0	3

This produces a frequency table where province is cross tabulated by whether the respondent ever used PrEP (vs. Never). This is good, but it's difficult to know which 1 means "Male" and which means "Yes". First, in R rows are always the first dimension in a specification, just as in the subsetting example. Second, it will be easier if we give names to our dimensions. Look again at your help documentation for `table`. There is an argument called "`dnn`" which indicates "the names to be given to the dimensions in the result (the dimnames names)." Let's provide some names, in the same order as the variables were provided:

```
table(sexnow2018$Q1_province, sexnow2018$PrEPUse_Ever, dnn = c("Province", "PrEP Use"))
```

Note that the names have to be "concatenated" into a single object, which we do using the `c` function. This should give us a table where the margins are named. Try it yourself to see what this looks like.

Finally, suppose that we are interested in the proportions, rather than frequencies, of cell. For that, we can use the `prop.table` function. The help documentation says that `prop.table` takes a **table** as its input. It just so happens that the `table` function produces a table object! All that remains is to pass the table we already made to `prop.table`. In other words, you can simply wrap your `table` function in a new `prop.table()` function.

```
prop.table(x = table(sexnow2018$Q1_province, sexnow2018$PrEPUse_Ever, dnn = c(
  "Province", "PrEP Use")))
```

Province	PrEP Use	
	Ever	Never
7777: Poor Data Quality	0.0000000000	0.0003184713
9999: True Missing	0.0006369427	0.0028662420
Alberta	0.0136942675	0.1595541401
British Columbia	0.0398089172	0.1687898089
I don't live in Canada	0.0000000000	0.0019108280
Manitoba	0.0038216561	0.0487261146
New Brunswick	0.0000000000	0.0012738854
Newfoundland & Labrador	0.0000000000	0.0003184713
Northwest Territories	0.0000000000	0.0003184713
Nova Scotia	0.0060509554	0.0484076433
Nunavut	0.0003184713	0.0006369427
Ontario	0.0585987261	0.3012738854
Quebec	0.0331210191	0.1085987261
Saskatchewan	0.0000000000	0.0009554140

Voila! This produces a proportion table out of our frequency table. But the proportions use the entire table as the denominator, meaning that 7.9% of people are men who served in the military, 51.1% are women who did not, etc. This is not always useful, so the `prop.table` function take an argument called “margin”. Margin should be a number indicating the dimension you want used as the denominator for your proportions. R starts with rows, so use “1” if you want the table **row totals** to act as the denominator, and 2 if you want the table **column totals** to be the denominator.

```
prop.table(x = table(sexnow2018$Q1_province, sexnow2018$PrEPUse_Ever, dnn = c(
  "Province", "PrEP Use")), margin = 1)
```

1.5.3 Summaries of Numeric Variables

The `summary` function introduced earlier provides us with many of the descriptive statistics you are likely interested in when working with a continuous variable. You can also use the `mean` and `median` functions to calculate only these statistics. Similarly, you can use the `sd` function to calculate a standard deviation. These are pretty straightforward, however, note that unless you specify `na.rm = TRUE` (short for “remove missing”), an object with any

missing elements will return NA. This is true of many of R's calculation functions; they take missingness quite seriously.

Suppose, though, that we want to know the mean of something in a cross tabulation? For this, we can use the `by` function to get different results for each group. Let's find the mean age for Indigenous and non-Indigenous respondents:

```
by(data = sexnow2018$Q11_age, INDICES = sexnow2018$Q2_ethnicity_indigenous, FUN = mean, na.rm = TRUE)
```

```
sexnow2018$Q2_ethnicity_indigenous: 9999: True Missing  
[1] 28
```

```
-----  
sexnow2018$Q2_ethnicity_indigenous: No  
[1] 36.02431
```

```
-----  
sexnow2018$Q2_ethnicity_indigenous: Yes  
[1] 33.21639
```

If you have trouble, look at the help documentation: “data” can be any object for which our desired function makes sense. It can even be an entire data frame. We used the age variables. “INDICES” must be a factor (or interpretable as one). “FUN” is the function (e.g., mean, median, sd, summary). In this case, we don't use parentheses, just the name of the function. The `by` function splits the age object and applies the `mean` function to it. Turning to our results we see that the Indigenous participants are, on average, slightly younger in our study.

1.6 Quick Overview

Basic descriptive analyses in RStudio are accessible to many users. For the most part, the functions you will use are pretty straightforward. I have tried to summarize them below.

Need	Function
Read in your dataset.	<code>read.csv("<path location of your csv file>")</code>
Subletting your dataset into a smaller dataset	<code><new data> <- [<old data>[which(<old data>\$<old variable> == "<desired group to include in new dataset>"),]</code>
Identify whether a variable is numeric or factor.	<code>class(<data>\$<variable>)</code>
Identify all the levels in a variable.	<code>levels(<data>\$<factor variable>)</code>
Relevel a variable	<code><data>\$<new variable>[<data>\$<old variable> == "<old level>"] <- "<new level 1>"</code> <code><data>\$<new variable>[<data>\$<old variable> == "<old level>"] <- "<new level ...>"</code> <code><data>\$<new variable>[<data>\$<old variable> == "<old level>"] <- "<new level x>"</code>
Convert a categorical factor to a numeric variable.	<code><data>\$<factor variable> <- as.numeric(<data>\$<numeric variable>)</code>
Convert a numeric variable to a categorical factor.	<code><data>\$<numeric variable> <- as.factor(<data>\$<numeric variable>)</code>
Calculate the mean, median, quartile, minimum, or maximum values of a continuous variable.	<code>summary(<data>\$<numeric variable>)</code>
Calculate the standard deviation of a continuous variable.	<code>sd(<data>\$<numeric variable>)</code>
Calculate numeric statistics separately for multiple groups.	<code>by(data = (<data>\$<numeric variable>, INDICES = (<data>\$<grouping factor>, FUN = summary)</code>
Calculate the frequency of responses across a categorical factor variable.	<code>table(<data>\$<factor variable>)</code>
Calculate the proportion of responses for each level of a categorical factor variable	<code>prop.table(table(<data>\$<factor variable>))</code>
Create a cross tabulation of two categorical factor variables	<code>table(<data>\$<factor variable>, <data>\$<factor variable>)</code>
Create cross tabulated proportions of two categorical factor variables	<code>prop.table(table (<data>\$<factor variable>, <data>\$<factor variable>), margin = 1)</code>

If you find yourself in a pinch, remember to consult the help documentation and rely on the active community of R users who can help you out when you encounter errors. Best of luck with your descriptive analyses and enjoy your explorations of the Sex Now dataset!