Workshop Day 1

Day 1

Today, we will introduce RStudio and jupyter notebooks, the basic data structures in R (lists, data frames, matrices, vectors) and how each is used, and then a start on exploratory analysis such as looking at the distribution of a single column or finding the percentage of missing values for all columns. To start, we need to check that the following libraries are installed and loaded. I will demonstrate installing these libraries and opening RStudio. This notebook will be available as an Jupyter notebook, R markdown file, and an exported pdf and will be updated at the end of the session.

```r
library(HDSinRdata)
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
## -- 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 become error
```

Basic Computations

Let's start with some basic calculations using the operators below:

- Addition: `5+6`
- Subtraction: `7-2`
- Multiplication: `2*3`
- Division: `6/3`
- Exponentiation: `4^2`
- Modulo: `100 %% 4`

For example, use the modulo operator to find what 100 mod 4 is. It should return 0 since 100 is divisible by 4.

```r
100 %% 4
```

```
## [1] 0
```

If we want to save the result of any computation, we need to create an object to store our value of interest. An **object** is simply a named data structure that allows us to reference that data structure. Objects are also commonly called **variables**. In the code below, we create an object `x` which stores the value `5` using the assignment operator `<-`. The assignment operator assigns whatever is on the right hand side of the operator to the name on the left hand side. We can now reference `x` by calling its name. Additionally, we can update its value by adding 1. In the second line of code, the computer first finds the value of the right hand side by finding the current value of `x` before adding 1 and assigning it back to `x`.

```r
x <- 2+3
x <- x+1
x
```

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

We can create and store multiple objects by using different names. The code below creates a new object `y` that is one more than the value of `x`. We can see that the value of `x` is still `5` after running this code.

```r
x <- 2+3
y <- x
y <- y + 1
x
```

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

Data Structures

Data structures are how information is stored in R, and the data structures that we use inform R how to interpret our code. Any **object** is a named instance of a data structure. For example, the object `ex_num` below is a vector of numeric type.

```r
ex_num <- 4
```

The main data structures in R are **vectors**, **factors**, **matrices**, **arrays**, **lists**, and **data frames**. These structures are distinguished by their dimensions and by the type of data they store. For example, we might have a 1-dimensional vector that contains all numeric values, or we could have a 2-dimensional data frame with rows and columns where we might have one numeric column and one character column.

Data Types

Each individual value in R has a type: logical, integer, double, or character. We can think of these as the building blocks of all data structures. Below, we can use the `typeof` function to find the type of our vector from above, which shows that the value of `ex_num` is a **double**. A double is a numeric value with a stored decimal.

```r
typeof(ex_num)
```

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

We can also work with two other types of objects: characters (e.g. "php", "stats") and booleans (e.g. TRUE, FALSE), also known as logicals.

```r
ex_bool <- TRUE
ex_char <- "Alice"

typeof(ex_bool)
```

```
## [1] "logical"
```

```r
typeof(ex_char)
```

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

One important characteristic of logical objects is that R will also interpret them as 0/1. This means they can be added as in the example below: each `TRUE` has a value of 1 and each `FALSE` has a value of 0.

```r
TRUE+FALSE+TRUE
```

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

To create all of the above objects, we used the assignment operator `<-`. You may see code elsewhere that uses an `=` instead. While `=` can also be used for assignment, it is more standard practice to use `<-`.

Vectors

In the examples above, we created objects with a single value. R actually uses a vector of length 1 to store this information. **Vectors** are 1-dimensional data structures that can store multiple data values of the same type (e.g. character, boolean, or numeric). We can confirm this by using the `is.vector()` function, which returns whether or not the inputted argument is a vector.

```r
is.vector(ex_bool)
```

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

One way to create a vector with multiple values is to use the combine function `c()`. Below we create two vectors: one with the days of the week and one with the amount of rain on each day. The first vector has all character values, and the second one has all numeric values.

```r
days <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
rain <- c(5, 0.1, 0, 0, 0.4)
```

Remember, vectors cannot store objects of different types. Because of this, in the code below, R automatically converts the numeric value to be a character in order to store these values in a vector together.

```r
c("Monday", 5)
```

```
## [1] "Monday" "5"
```

The `class()` function returns the data structure of an object. If we check the classes of these two objects using the `class()` function, we will see that R tells us that the first is a character vector and the second is a numeric vector. This matches the data type in this case.

```r
class(days)
```

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

```r
class(rain)
```

```
## [1] "numeric"
```

Indexing a Vector

Once we have a vector, we may want to access certain values stored in that vector. To do so, we index the vector using the position of each value: the first value in the vector has index 1, the second value has index 2, etc. When we say a vector is 1-dimensional, we mean that we can define the position of each value by a single index. To index the vector, we then use square brackets `[]` after the vector name and provide the position. Below, we use these indices to find the value at index 1 and the value at index 4.

```r
days[1]
```

```
## [1] "Monday"
```

```r
days[4]
```

```
## [1] "Thursday"
```

We can either access a single value or a subset of values using a vector of indices. Let's see what happens when we use a vector of indices `c(1,4)` and then try using `-c(1,4)` and see what happens then. In the first case, we get the values at index 1 *and* at index 4. In the second case, we get all values *except* at those indices. The `-` indicates that we want to remove rather than select these indices.

```r
days[c(1,4)]
```

```
## [1] "Monday"   "Thursday"
```

```r
days[-c(1,4)]
```

```
## [1] "Tuesday"   "Wednesday" "Friday"
```

However, always indexing by the index value can sometimes be difficult or inefficient. One extra feature of vectors is that we can associate a name with each value. Below, we update the names of the vector `rain` to be the days of the week and then find Friday's rain count by indexing with the name.

```r
names(rain) <- days
print(rain)
```

```
##    Monday   Tuesday Wednesday  Thursday    Friday 
##       5.0       0.1       0.0       0.0       0.4
```

```r
rain["Friday"]
```

```
## Friday 
##    0.4
```

The last way to index a vector is to use TRUE and FALSE values. If we have a vector of booleans that is the same length as our original vector, then this will return all the values that correspond to a TRUE value. For example, indexing the `days` vector by the logical vector `ind_bools` below will return its first and fourth values. We will see more about using logic to access certain values later on.

```r
ind_bools <- c(TRUE, FALSE, FALSE, TRUE, FALSE)
days[ind_bools]
```

```
## [1] "Monday"   "Thursday"
```

Editing a Vector and Calculations

The mathematical operators we saw in the last chapter (`+`, `-`, `*`, `/`, `^`, `%%`) can all be applied to numeric vectors and will be applied element-wise. That is, in the code below, the two vectors are added together by index. This holds true for some of the built-in math functions as well:

- `exp()` - expoential
- `log()` - log
- `sqrt()` - square root
- `abs()` - absolute value
- `round()` - round to nearest integer value
- `ceiling()` - round up to the nearest integer value
- `floor()` - round down to the nearest integer value

```r
c(1,2,3) + c(1,1,1)
```

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

```r
c(1,2,3) + 1 # equivalent to the code above
```

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

```r
sqrt(c(1,4,16))
```

```
## [1] 1 2 4
```

After we create a vector, we may need to update its values. For example, we may want to change a specific value. We can do so using indexing. Below, we update the rain value for Friday using the assignment operator.

```r
rain["Friday"] <- 0.5
rain
```

```
##    Monday   Tuesday Wednesday  Thursday    Friday 
##       5.0       0.1       0.0       0.0       0.5
```

Further, we may need to add extra entries. We can do so using the `c()` function again but this time passing in the vector we want to add to as our first argument. This will create a single vector with all previous and new values. Below, we add two days to both vectors and then check the length of the updated vector `rain`. The `length()` function returns the length of a vector.

```r
length(rain)
```

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

```r
days <- c(days,"Saturday","Sunday") # add the weekend with no rain
rain <- c(rain,0,0)
length(rain)
```

```
## [1] 7
```

We can also call some useful functions on vectors. For example, the `sum()`, `max()`, and `min()` functions will return the sum, maximum value, and minimum value of a vector, respectively.

Factors

A **factor** is a special kind of vector that behaves exactly like a regular vector except that it represents values from a category. In particular, a factor keeps track of all possible values of that category, which are called the **levels** of the factor. Factors will be especially helpful when we start getting into data analysis and have categorical columns. The `as.factor()` function will convert a vector to a factor.

```r
days <- c("Monday", "Tuesday", "Wednesday", "Monday", "Thursday", "Wednesday")
days_fct <- as.factor(days)

class(days_fct)
```

```
## [1] "factor"
```

```r
levels(days_fct)
```

```
## [1] "Monday"    "Thursday"  "Tuesday"   "Wednesday"
```

Data Frames

A data frame is two-dimensional data structure. Data frames allow for each column to be a different type. In this case, each row corresponds to a single data entry (or observation) and each column corresponds to a different variable.

For example, suppose that, for every day in a study, we want to record the temperature, rainfall, and day of the week. Temperature and rainfall can be numeric values, but day of the week will be character type. We create a data frame using the `data.frame()` function. Note that I am providing column names for each vector (column).

The `head()` function prints the first six rows of a data frame (to avoid printing very large datasets). In our case, all the data is shown because we only created four rows. The column names are displayed as well as their type. By contrast, the `tail()` function prints the last six rows of a data frame.

```r
weather_data <- data.frame(day_of_week = c("Monday","Tuesday","Wednesday","Monday"),
                           temp = c(70,62,75,50), rain = c(5,0.1,0.0,0.5))
head(weather_data)
```

```
##   day_of_week temp rain
## 1      Monday   70  5.0
## 2     Tuesday   62  0.1
## 3   Wednesday   75  0.0
## 4      Monday   50  0.5
```

The `dim()`, `nrow()`, and `ncol()` functions return the dimensions, number of rows, and number of columns of a data frame, respectively.

```r
dim(weather_data)
```

```
## [1] 4 3
```

```r
nrow(weather_data)
```

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

```r
ncol(weather_data)
```

```
## [1] 3
```

The column names can be found (or assigned) using the `colnames()` or `names()` function. These were specified when I created the data. On the other hand, the row names are currently the indices.

```r
rownames(weather_data) <- c("6/1", "6/2", "6/3", "6/8")
head(weather_data)
```

```
##     day_of_week temp rain
## 6/1      Monday   70  5.0
## 6/2     Tuesday   62  0.1
## 6/3   Wednesday   75  0.0
## 6/8      Monday   50  0.5
```

We update the row names to be more informative as with a matrix using the `rownames()` function.

```
rownames(weather_data) <- c("6/1", "6/2", "6/3", "6/8")
head(weather_data)
```

```
##     day_of_week temp rain
## 6/1      Monday   70  5.0
## 6/2     Tuesday   62  0.1
## 6/3   Wednesday   75  0.0
## 6/8      Monday   50  0.5
```

Indexing a Data Frame

We can select elements of the data frame using the indices for both the rows and columns we would like to select. Below, we access a single value and then a subset of our data frame. The subset returned is itself a data frame. Note that the second line below returns a data frame.

```
weather_data[1,2]
```

```
## [1] 70
```

```
weather_data[1,c("day_of_week","temp")]
```

```
##     day_of_week temp
## 6/1      Monday   70
```

Another useful way to access the columns of a data frame is by using the $ accessor and the column name.

```
weather_data$day_of_week
```

```
## [1] "Monday"    "Tuesday"   "Wednesday" "Monday"
```

```
weather_data$temp
```

```
## [1] 70 62 75 50
```

As with vectors, we can change values in a data frame by indexing those entries.

```
weather_data[1, "rain"] <- 2.2
weather_data
```

```
##     day_of_week temp rain
## 6/1      Monday   70  2.2
## 6/2     Tuesday   62  0.1
## 6/3   Wednesday   75  0.0
## 6/8      Monday   50  0.5
```

Lists

A data frame is actually a special kind of another data structure called a **list**, which is a collection of objects under the same name. These objects can be vectors, matrices, data frames, or even other lists! There does not have to be any relation in size, type, or other attribute between different members of the list. Below, we create an example list using the `list()` function, which takes in a series of objects. What are the types of each element of the list? We can access each element using the index, but here we need to use double brackets.

```
ex_list <- list(name="John", medications = c("ibuprofen", "metformin"),
                past_weights = c(136, 142, 159))
print(ex_list)
```

```
## $name
## [1] "John"
##
## $medications
## [1] "ibuprofen" "metformin"
##
## $past_weights
## [1] 136 142 159
```

```
ex_list$medications
```

```
## [1] "ibuprofen" "metformin"
```

Exploratory Analysis in R

For our exploratory analysis, we will use some pain available in the `HDSinRdata` package. However, when we have data stored in a local file, we can load that data into R using the corresponding function. We demonstrate this with the `read.csv()` function to read in a comma-delimited file. This function returns the information as a data frame (try typing `?read.csv` in the console to read more about this function).By default, R will look for the file in your current working directory. To find the working directory, you can run `getwd()`. You can see below that my current working directory is where the book content is on my computer.

```
getwd()
```

```
## [1] "/Users/alice/Dropbox/health-data-science-in-r/workshop"
```

You can either move the .csv file to your current working directory and load it in, or you can specify the path to the .csv file. Another option is to update your working directory by using the `setwd()` function (e.g. `setwd('/Users/Alice/Dropbox/r-for-health-data-science/book/data')`). If you receive an error that a file cannot be found, you most likely have the wrong path to the file or the wrong file name. Below, I chose to specify the path to the downloaded .csv file, saved this file to an object called `df`, and then printed that `df` object.

```
df <- read.csv("../book/data/fake_names.csv") # you will need to update this with the path to your file
df
```

```
##                   Name Age     DOB           City State
## 1            Ken Irwin  37 6/28/85      Providence    RI
## 2 Delores Whittington  56 4/28/67      Smithfield    RI
## 3       Daniel Hughes  41 5/22/82      Providence    RI
## 4         Carlos Fain  83  2/2/40          Warren    RI
## 5        James Alford  67 2/23/56 East Providence    RI
## 6        Ruth Alvarez  34 9/22/88      Providence    RI
```

We will use the `pain` data available in our package. We load this in using the `data()` function.

```r
data(pain)
```

Summarizing Data Columns

We will now look at the data we have loaded into the data frame called `pain`. We use the `head()` function to print the first six rows. However, note that we have so many columns that all not of the columns are displayed! For those that are displayed, we can see the data type for each column under the column name. For example, we can see that the column `PATIENT_NUM` is a numeric column of type `dbl`. Because patients identification numbers are technically nominal in nature, we might consider whether we should make convert this variable to a factor or a character representation later on. We can use the `names()` function to print all the column names. Note that columns `X101` to `X238` correspond to numbers on a body pain map (see the data documentation for the image of this map). Each of these columns has a 1 if the patient indicated that they have pain in that corresponding body part and a 0 otherwise.

```r
head(pain)
```

```
## # A tibble: 6 x 92
##   PATIENT_NUM  X101  X102  X103  X104  X105  X106  X107  X108  X109  X110  X111
##         <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1       13118     0     0     0     0     0     0     0     0     0     0     0
## 2       21384     0     0     0     0     0     0     0     0     0     0     0
## 3        6240     0     0     0     0     0     0     0     0     0     0     0
## 4        1827     0     0     0     0     0     0     0     0     0     0     0
## 5       11309     0     0     0     0     0     0     0     0     0     0     0
## 6       11093     0     0     0     0     0     0     0     0     0     1     0
## # i 80 more variables: X112 <dbl>, X113 <dbl>, X114 <dbl>, X115 <dbl>,
## #   X116 <dbl>, X117 <dbl>, X118 <dbl>, X119 <dbl>, X120 <dbl>, X121 <dbl>,
## #   X122 <dbl>, X123 <dbl>, X124 <dbl>, X125 <dbl>, X126 <dbl>, X127 <dbl>,
## #   X128 <dbl>, X129 <dbl>, X130 <dbl>, X131 <dbl>, X132 <dbl>, X133 <dbl>,
## #   X134 <dbl>, X135 <dbl>, X136 <dbl>, X201 <dbl>, X202 <dbl>, X203 <dbl>,
## #   X204 <dbl>, X205 <dbl>, X206 <dbl>, X207 <dbl>, X208 <dbl>, X209 <dbl>,
## #   X210 <dbl>, X211 <dbl>, X212 <dbl>, X213 <dbl>, X214 <dbl>, X215 <dbl>, ...
```

```r
names(pain)
```

```
##  [1] "PATIENT_NUM"                        "X101"
##  [3] "X102"                               "X103"
##  [5] "X104"                               "X105"
##  [7] "X106"                               "X107"
##  [9] "X108"                               "X109"
## [11] "X110"                               "X111"
## [13] "X112"                            "X113"
## [15] "X114"                            "X115"
## [17] "X116"                            "X117"
## [19] "X118"                            "X119"
## [21] "X120"                            "X121"
## [23] "X122"                            "X123"
## [25] "X124"                            "X125"
## [27] "X126"                            "X127"
## [29] "X128"                            "X129"
## [31] "X130"                            "X131"
## [33] "X132"                            "X133"
## [35] "X134"                            "X135"
## [37] "X136"                            "X201"
## [39] "X202"                            "X203"
## [41] "X204"                            "X205"
## [43] "X206"                            "X207"
## [45] "X208"                            "X209"
## [47] "X210"                            "X211"
## [49] "X212"                            "X213"
## [51] "X214"                            "X215"
## [53] "X216"                            "X217"
## [55] "X218"                            "X219"
## [57] "X220"                            "X221"
## [59] "X222"                            "X223"
## [61] "X224"                            "X225"
## [63] "X226"                            "X227"
## [65] "X228"                            "X229"
## [67] "X230"                            "X231"
## [69] "X232"                            "X233"
## [71] "X234"                            "X235"
## [73] "X236"                            "X237"
## [75] "X238"                            "PAIN_INTENSITY_AVERAGE"
## [77] "PROMIS_PHYSICAL_FUNCTION"        "PROMIS_PAIN_BEHAVIOR"
## [79] "PROMIS_DEPRESSION"               "PROMIS_ANXIETY"
## [81] "PROMIS_SLEEP_DISTURB_V1_0"       "PROMIS_PAIN_INTERFERENCE"
## [83] "GH_MENTAL_SCORE"                 "GH_PHYSICAL_SCORE"
## [85] "AGE_AT_CONTACT"                  "BMI"
## [87] "CCI_TOTAL_SCORE"                 "PAIN_INTENSITY_AVERAGE.FOLLOW_UP"
## [89] "PAT_SEX"                         "PAT_RACE"
## [91] "CCI_BIN"                         "MEDICAID_BIN"
```

Recall that the `$` operator can be used to access a single column. Alternatively, we can use double brackets `[[]]` to select a column. Below, we demonstrate both ways to print the first five values in the column with the patient's average pain intensity.

```
pain$PAIN_INTENSITY_AVERAGE[1:5]
```

```
## [1] 7 5 4 7 8
```

```
pain[["PAIN_INTENSITY_AVERAGE"]][1:5]
```

```
## [1] 7 5 4 7 8
```

To explore the range and distribution of a column's values, we can use some of the base R functions. For example, the `summary()` function is a useful way to summarize a numeric column's values. Below, we can see that the pain intensity values range from 0 to 10 with a median value of 7 and that there is 1 NA value.

```r
summary(pain$PAIN_INTENSITY_AVERAGE)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##   0.000   5.000   7.000   6.485   8.000  10.000       1
```

We have already seen the `max()`, `min()`, `mean()`, and `median()` functions that could have computed some of these values for us separately. Since we do have an NA value, we add the `na.rm=TRUE` argument to these functions. Without this argument, the returned value for all of the functions will be NA.

```r
min(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
```

```
## [1] 0
```

```r
max(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
```

```
## [1] 10
```

```r
mean(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
```

```
## [1] 6.485271
```

```r
median(pain$PAIN_INTENSITY_AVERAGE, na.rm=TRUE)
```

```
## [1] 7
```

Additionally, the functions below are helpful for summarizing quantitative variables.

- `range()` - returns the minimum and maximum values for a numeric vector x
- `quantile()` - returns the sample quantiles for a numeric vector
- `IQR()` - returns the interquartile range for a numeric vector

By default, the `quantile()` function returns the sample quartiles.

```r
quantile(pain$PAIN_INTENSITY_AVERAGE, na.rm = TRUE)
```

```
##   0%  25%  50%  75% 100% 
##    0    5    7    8   10
```

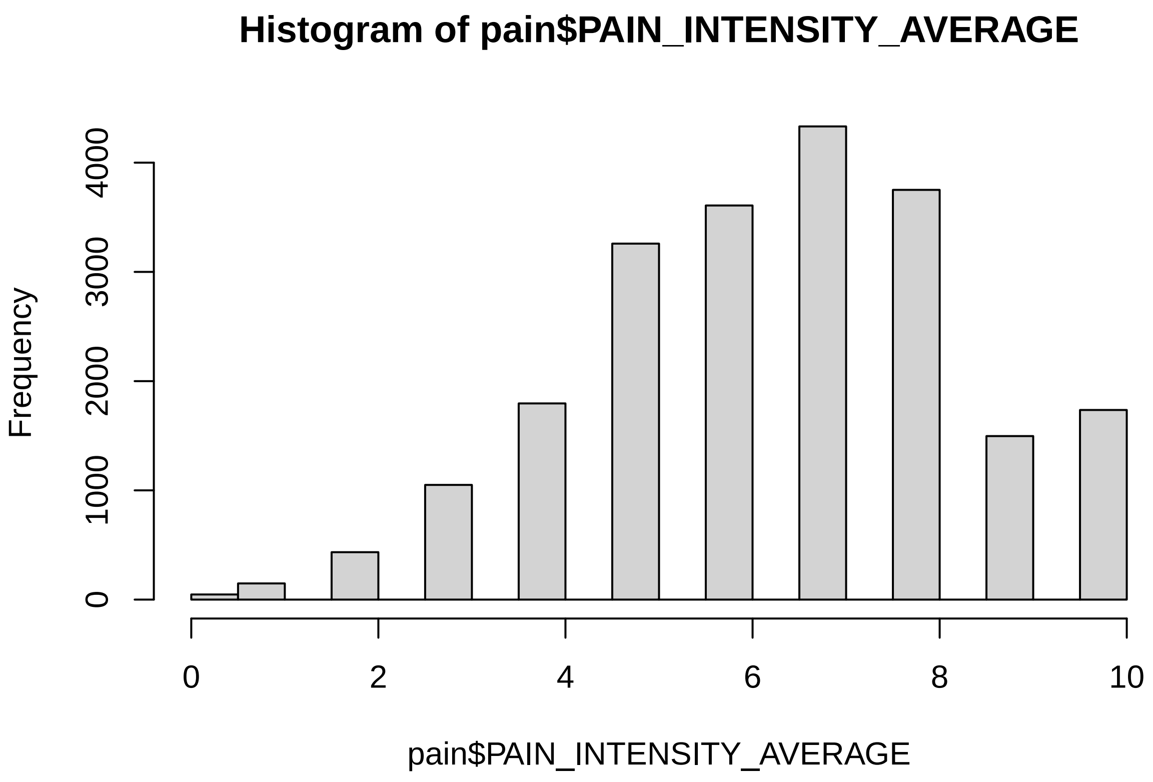
However, we can pass in a list of probabilities to use instead. For example, below we find the 0.1 and 0.9 quantiles. Again, we add the `na.rm=TRUE` argument.

```r
quantile(pain$PAIN_INTENSITY_AVERAGE, probs = c(0.1, 0.9), na.rm=TRUE)
```

```
## 10% 90% 
##   4   9
```

We can also plot a histogram of the sample distribution using the `hist()` function.

```r
hist(pain$PAIN_INTENSITY_AVERAGE)
```



Missing values

As we saw above, this data contains some missing values, which are represented as `NA` in R. R treats these values as if they were unknown, which is why we have to add the `na.rm=TRUE` argument to functions like `sum()` and `max()`. In the example below, we can see that R figures out that 1 plus an unknown number is also unknown!

```r
NA+1
```

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

We can determine whether a value is missing using the function `is.na()`. This function returns `TRUE` if the value is NA and `FALSE` otherwise. We can then sum up these values for a single column, since each `TRUE` value corresponds to a value of 1 and each `FALSE` corresponds to a value of 0. Below we can see that there is a single NA value for the column `PATIENT_NUM`, which is the patient ID number.

```r
sum(is.na(pain$PATIENT_NUM))
```

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

If we want to calculate the sum of NA values for each column instead of just a single column, we can use the `apply` function. Since we want to apply this computation over the columns, the second argument has value 2. Recall that the last argument is the function we want to call for each column. In this case, we want to apply the combination of the `sum()` and `is.na()` function. To do so, we have to specify this function ourselves. This is called an **anonymous function** since it doesn't have a name.

```
num_missing_col <- apply(pain, 2, function(x) sum(is.na(x)))
min(num_missing_col)
```

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

Interestingly, we can see that there is at least one missing value in each column. It might be the case that there is a row with all NA values. Let's apply the same function by row. Taking the maximum, we can see that row 11749 has all NA values.

```
num_missing_row <- apply(pain, 1, function(x) sum(is.na(x)))
max(num_missing_row)
```

```
## [1] 92
```

```
which.max(num_missing_row)
```

```
## [1] 11749
```

We remove that row and then find the percentage of missing values by column. We can see that the column with the highest percentage of missing values is the pain intensity at follow-up. In fact, only 33% of patients have a recorded follow-up visit.

```
pain <- pain[-11749,]
num_missing_col <- apply(pain, 2, function(x) sum(is.na(x))/nrow(pain))
num_missing_col
```

```
##                     PATIENT_NUM                           X101
##                     0.000000000                    0.000000000
##                            X102                           X103
##                     0.000000000                    0.000000000
##                            X104                           X105
##                     0.000000000                    0.000000000
##                            X106                           X107
##                     0.000000000                    0.000000000
##                            X108                           X109
##                     0.000000000                    0.000000000
##                            X110                           X111
##                     0.000000000                    0.000000000
##                            X112                           X113
##                     0.000000000                    0.000000000
##                            X114                           X115
##                     0.000000000                    0.000000000
##                            X116                           X117
##                     0.000000000                    0.000000000
##                            X118                           X119
##                     0.000000000                    0.000000000
##                            X120                           X121
##                     0.000000000                    0.000000000
##                            X122                           X123
##                     0.000000000                    0.000000000
##                             X124                          X125
##                      0.000000000                   0.000000000
##                             X126                          X127
##                      0.000000000                   0.000000000
##                             X128                          X129
##                      0.000000000                   0.000000000
##                             X130                          X131
##                      0.000000000                   0.000000000
##                             X132                          X133
##                      0.000000000                   0.000000000
##                             X134                          X135
##                      0.000000000                   0.000000000
##                             X136                          X201
##                      0.000000000                   0.000000000
##                             X202                          X203
##                      0.000000000                   0.000000000
##                             X204                          X205
##                      0.000000000                   0.000000000
##                             X206                          X207
##                      0.000000000                   0.000000000
##                             X208                          X209
##                      0.000000000                   0.000000000
##                             X210                          X211
##                      0.000000000                   0.000000000
##                             X212                          X213
##                      0.000000000                   0.000000000
##                             X214                          X215
##                      0.000000000                   0.000000000
##                             X216                          X217
##                      0.000000000                   0.000000000
##                             X218                          X219
##                      0.000000000                   0.000000000
##                             X220                          X221
##                      0.000000000                   0.000000000
##                             X222                          X223
##                      0.000000000                   0.000000000
##                             X224                          X225
##                      0.000000000                   0.000000000
##                             X226                          X227
##                      0.000000000                   0.000000000
##                             X228                          X229
##                      0.000000000                   0.000000000
##                             X230                          X231
##                      0.000000000                   0.000000000
##                             X232                          X233
##                      0.000000000                   0.000000000
##                             X234                          X235
##                      0.000000000                   0.000000000
##                             X236                          X237
##                      0.000000000                   0.000000000
##                             X238        PAIN_INTENSITY_AVERAGE
##                      0.000000000                   0.000000000
##          PROMIS_PHYSICAL_FUNCTION          PROMIS_PAIN_BEHAVIOR
##                      0.000000000                   0.294117647
##              PROMIS_DEPRESSION                     PROMIS_ANXIETY
##                    0.004016991                        0.004016991
##        PROMIS_SLEEP_DISTURB_V1_0         PROMIS_PAIN_INTERFERENCE
##                    0.004016991                        0.006972020
##                GH_MENTAL_SCORE                  GH_PHYSICAL_SCORE
##                    0.136023640                        0.136023640
##                 AGE_AT_CONTACT                                BMI
##                    0.000000000                        0.260042479
##                CCI_TOTAL_SCORE PAIN_INTENSITY_AVERAGE.FOLLOW_UP
##                    0.000000000                        0.670422015
##                        PAT_SEX                           PAT_RACE
##                    0.000000000                        0.006510296
##                        CCI_BIN                       MEDICAID_BIN
##                    0.000000000                        0.013851695
```

To omit those observations with NA values, we can either use the `na.omit()` function, which returns a data frame with all rows with no NA values, or we can use the `complete.cases()` function, which returns the indices of these rows.

```
pain_sub1 <- na.omit(pain)
pain_sub2 <- pain[complete.cases(pain),]
dim(pain_sub1)
```

```
## [1] 2413   92
```

```
dim(pain_sub2)
```

```
## [1] 2413   92
```

Subsetting Data

Above, we used TRUE/FALSE values to select rows in a data frame. The logic operators in R allow us to expand on this capability to write more complex logic. The operators are given below.

- `<` less than
- `<=` less than or equal to
- `>` greater than
- `>=` greater than or equal to
- `==` equal to
- `!=` not equal to
- `a %in% b` a's value is in a vector of values b

The first six operators are a direct comparison between two values and are demonstrated below.

```
2 < 2
```

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

```r
2 <= 2
```

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

```r
3 > 2
```

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

```r
3 >= 2
```

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

```r
"A" == "B"
```

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

```r
"A" != "B"
```

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

The `%in%` operator is slightly different. This operator checks whether a value is in a set of possible values. Below, we can check whether values are in the set `c(4,1,2)`.

```r
1 %in% c(4,1,2)
```

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

```r
c(0,1,5) %in% c(4,1,2)
```

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

Additionally, we can use the following operators, which allow us to negate or combine logical operators.

- `!x` - the **NOT** operator `!` reverses TRUE/FALSE values
- `x | y` - the **OR** operator `|` checks whether *either* x or y is equal to TRUE
- `x & y` - the **AND** operator `&` checks whether *both* x and y are equal to TRUE
- `xor(x,y)` - the **xor** function checks whether exactly one of x or y is equal to TRUE (called exclusive or)
- `any(x)` - the **any** function checks whether any value in x is TRUE (equivalent to using an OR operator `|` between all values)
- `all(x)` - the **all** function checks whether all values in x are TRUE (equivalent to using an AND operator `&` between all values)

Some simple examples for each are given below.

```r
!(2 < 3)
```

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

```
("Alice" < "Bob") | ("Alice" < "Aaron")
```

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

```
("Alice" < "Bob") & ("Alice" < "Aaron")
```

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

```
xor(TRUE, FALSE)
```

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

```
any(c(FALSE, TRUE, TRUE))
```

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

```
all(c(FALSE, TRUE, TRUE))
```

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

Let's demonstrate these operators on the pain data. We first update the Medicaid column by making the character values more informative. The logic on the left hand side selects those that do or do not have Medicaid and then assigns those values to the new ones.

```
pain$MEDICAID_BIN[pain$MEDICAID_BIN == "no"] <- "No Medicaid"
pain$MEDICAID_BIN[pain$MEDICAID_BIN == "yes"] <- "Medicaid"
table(pain$MEDICAID_BIN)
```

```
##
##    Medicaid No Medicaid
##        4601       16757
```

Additionally, we could subset the data to only those who have follow-up. The not operator `!` will reverse the TRUE/FALSE values returned from the `is.na()` function. Therefore, the new value will be TRUE if the follow-up value is *not* NA.

```
pain_follow_up <- pain[!is.na(pain$PAIN_INTENSITY_AVERAGE.FOLLOW_UP),]
```

Earlier, we created a column indicating whether or not a patient has lower back pain. We now use the `any()` function to check whether a patient has general back pain. If at least one of these values is equal to 1, then the function will return TRUE. If we had used the `all()` function instead, this would check whether all values are equal to 1, indicating that a patient has pain on their whole back.

```
pain$BACK <- any(pain$X208==1, pain$X209==1, pain$X212==1, pain$X213==1,
                 pain$X218==1, pain$X219==1)
```

Lastly, we look at the column for patient race `PAT_RACE`. The `table()` function shows that most patients are `WHITE` or `BLACK`. Given how few observations are in the other categories, we may want to combine some of these levels into one.

```r
table(pain$PAT_RACE)
```

```
## 
##           ALASKA NATIVE         AMERICAN INDIAN                   BLACK 
##                       2                      58                    3229 
##                 CHINESE                DECLINED                FILIPINO 
##                      21                     121                       6 
##           GUAM/CHAMORRO                HAWAIIAN         INDIAN (ASIAN) 
##                       1                       1                      49 
##                JAPANESE                  KOREAN           NOT SPECIFIED 
##                       9                      10                       4 
##                   OTHER             OTHER ASIAN OTHER PACIFIC ISLANDER 
##                       1                      47                      12 
##              VIETNAMESE                   WHITE 
##                       6                   17940
```

To combine some of these levels, we can use the `%in%` operator. We first create an Asian, Asian American, or Pacific Islander race category and then create an American Indian or Alaska Native category.

```r
aapi_values <- c("CHINESE", "HAWAIIAN", "INDIAN (ASIAN)", "FILIPINO", "VIETNAMESE",
                 "JAPANESE", "KOREAN", "GUAM/CHAMORRO", "OTHER ASIAN",
                 "OTHER PACIFIC ISLANDER")
pain$PAT_RACE[pain$PAT_RACE %in% aapi_values] <- "AAPI"
pain$PAT_RACE[pain$PAT_RACE %in% c("ALASKA NATIVE", "AMERICAN INDIAN")] <- "AI/AN"
table(pain$PAT_RACE)
```

```
## 
##          AAPI         AI/AN         BLACK      DECLINED NOT SPECIFIED 
##           162            60          3229           121             4 
##         OTHER         WHITE 
##             1         17940
```

Other Selection Functions

Above, we selected rows using TRUE/FALSE boolean values. Instead, we could have also used the `which()` function. This function takes TRUE/FALSE values and returns the index values for all the TRUE values. We use this to treat those with race given as `DECLINED` as not specified.

```r
pain$PAT_RACE[which(pain$PAT_RACE == "DECLINED")] <- "NOT SPECIFIED"
```

Another selection function is the `subset()` function. This function takes in two arguments. The first is the vector, matrix, or data frame to select from and the second is a vector of TRUE/FALSE values to use for row selection. We use this to find the observation with race marked as `OTHER`. We then update this race to also be marked as not specified.

```r
subset(pain, pain$PAT_RACE == "OTHER")
```

```
## # A tibble: 1 x 93
##   PATIENT_NUM  X101  X102  X103  X104  X105  X106  X107  X108  X109  X110  X111
##         <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1         3588     1     1     1     0     1     1     1     0     0     0     0
## # i 81 more variables: X112 <dbl>, X113 <dbl>, X114 <dbl>, X115 <dbl>,
## #   X116 <dbl>, X117 <dbl>, X118 <dbl>, X119 <dbl>, X120 <dbl>, X121 <dbl>,
## #   X122 <dbl>, X123 <dbl>, X124 <dbl>, X125 <dbl>, X126 <dbl>, X127 <dbl>,
## #   X128 <dbl>, X129 <dbl>, X130 <dbl>, X131 <dbl>, X132 <dbl>, X133 <dbl>,
## #   X134 <dbl>, X135 <dbl>, X136 <dbl>, X201 <dbl>, X202 <dbl>, X203 <dbl>,
## #   X204 <dbl>, X205 <dbl>, X206 <dbl>, X207 <dbl>, X208 <dbl>, X209 <dbl>,
## #   X210 <dbl>, X211 <dbl>, X212 <dbl>, X213 <dbl>, X214 <dbl>, X215 <dbl>, ...
```

Practice Exercises

1. Recreate the following data frame in R, where `temperature` and `co2` represent the average $\cdot F$ and CO_2 concentrations in mg/m^3 for the month of January 2008, and name it `city_air_quality`.

2. Use single square brackets to access the precipitation and CO_2 concentration entries for San Francisco and Paris in your data frame. Then, create a list of lists called `city_list` containing the city, precipitation, and CO_2 concentration information for both San Francisco and Paris.
3. Create a new data frame called `pain.new` that doesn't contain patients with NA values for both `GH_MENTAL_SCORE` and `GH_PHYSICAL_SCORE`, which are the PROMIS global mental and physical scores, respectively.
4. Look up the `colMeans()` function using the help operator (`?colMeans`). Use this function to create a vector of the total number of patients who reported pain in each of bodily pain regions. Then, create a data frame containing the minimum, median, mean, maximum, standard deviation, and variance of this vector.
5. Create a histogram to plot the distribution of the `PAIN_INTENSITY_AVERAGE.FOLLOW_UP` variable. Then, create a table summarizing how many patients had missing values in this column. Finally, compare the distributions of the other variables between those with and without missing follow up. What do you notice?