DSCoV Workshop: Intro to R

Introduction to R and RStudio

This workshop introduces you to R as a programming language. To start, you need to download R and RStudio.

Why R?

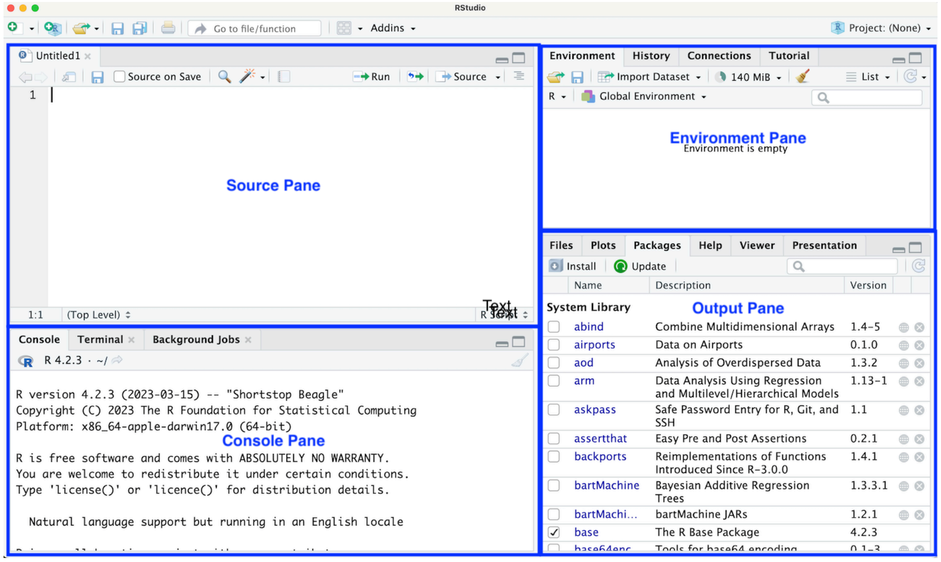
What are some of the benefits of using R?

- R is built for statisticians and data analysts.
- R is open source.
- R has most of the latest statistical methods available.
- R is flexible.

Since R is designed for statisticians, it is built with data in mind. This comes in handy when we want to streamline how we process and analyze data. It also means that many statisticians working on new methods are publishing user-created packages in R, so R users have access to most methods of interest. R is also an interpreted language, which means that we do not have to compile our code into machine language first; this allows for simpler syntax and more flexibility when writing our code, which also makes it a great first programming language to learn.

RStudio and Quarto

This document is a **Quarto document**. Quarto documents allow us to integrate text and code together. You can open Quarto files in RStudio to run the code as you read. When you open RStudio, there are multiple windows. You should see several windows as outlined in Figure 1.



```
R version 4.2.3 (2023-03-15) -- "Shortstop Beagle"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale
```

System Library

	Name	Description	Version
□	abind	Combine Multidimensional Arrays	1.4-5
□	airports	Data on Airports	0.1.0
□	aod	Analysis of Overdispersed Data	1.3.2
□	arm	Data Analysis Using Regression and Multilevel/Hierarchical Models	1.13-1
□	askpass	Safe Password Entry for R, Git, and SSH	1.1
□	assertthat	Easy Pre and Post Assertions	0.2.1
□	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.4.1
□	bartMachine	Bayesian Additive Regression Trees	1.3.3.1
□	bartMachi...	bartMachine JARs	1.2.1
☑	base	The R Base Package	4.2.3
□	base64enc	Tools for base64 encoding	0.1-3

Figure 1: RStudio Layout and Panes.

Panes

There are four panes shown by default:

- **Source Pane** - used for editing code files such as R Scripts or Quarto documents.
- **Console Pane** - used to show the live R session.
- **Environment Pane** - contains the Environment and History tabs, used to keep track of the current state.
- **Output Pane** - contains the Plots and Packages tabs.

```
# Calculate primary care physician to specialist ratio
pcp_phys <- c(6300, 1080, 9297, 16433)
spec_phys <- c(6750, 837, 10517, 22984)
pcp_spec_ratio <- 1000 * pcp_phys / spec_phys
```

The first line in the code chunk above starts with # and does not contain any code. This is a **comment** line, which allows us to add context, intent, or extra information to help the reader understand our code. A good rule of thumb is that we want to write enough comments so that we could open our code in 6 months and be able to understand what we were doing. As we develop longer chunks of code, this will become more important.

Unlike when we type code into the console, we can write multiple lines of code in our R Script without running them. In order to run the code in the script, we need to tell RStudio we are ready to run it. To run a single line of code, we can either hit Ctrl+Enter when on that line or we can hit the Run button at the top right of the source pane. This copies the code to the R Console. Try this out to run the first line of code that defines `pcp_phys`. You can see that the line of code has been run in the console pane. Now check your environment pane. You should see that you have a new object representing the one we just created. This pane keeps track of all current objects. Run the second line of code and see how the environment updates. If you look at the History tab within this pane, you see the history of R commands run.

If we want to run all lines of code in our script, we can use the Source button. Before we do this, we will clear our environment. You can do this by clicking the broom in the environment pane, which deletes all objects in the environment. Alternatively, you can go to Session -> Restart R in the main menu, which restarts your whole R session. After clearing your environment, click the Source button. You will see that in the R Console it shows that it sourced this file. This means that it runs through all lines of code in this file. You can see that our objects have been added back into our environment.

Basic Operations and Functions

Try out some other basic calculations using the following operators:

- 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.

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 following code, 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`.

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

```
[1] 6
```

We can create and store multiple objects by using different names. The following code 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.

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

```
[1] 5
```

When we use R, we have access to all the functions available in base R. A **function** takes in one or more inputs and returns a single output object. Let's first use the simple function `exp()`. This exponential function takes in one (or more) numeric values and exponentiates them. The following code computes e^3.

```
exp(3)
```

```
[1] 20.08554
```

Some other simple functions are shown that all convert a numeric input to an integer value. The `ceiling()` and `floor()` functions return the ceiling and floor of your input, and the `round()` function rounds your input to the closest integer. Note that the `round()` function rounds a number ending in 0.5 to the closest even integer.

```
ceiling(3.7)
```

```
[1] 4
```

```
floor(3.7)
```

```
[1] 3
```

```
round(2.5)
```

```
[1] 2
```

```r
round(3.5)
```

```
[1] 4
```

If we want to learn about a function, we can use the help operator `?` by typing it in front of the function we are interested in: this brings up the documentation for that particular function. This documentation often tells you the usage of the function, the **arguments** (the object inputs) and the **value** (information about the returned object), and it gives some examples of how to use the function. For example, if we want to understand the difference between `floor()` and `ceiling()`, we can call `?floor` and `?ceiling`. This should bring up the documentation in your help window. We can then read that the floor function rounds a numeric input down to the nearest integer, whereas the ceiling function rounds a numeric input up to the nearest integer.

Working Directories and Paths

Let's try using another example function: `read.csv()`. This function reads in a comma-delimited file and returns the information as a data frame (try typing `?read.csv` in the console to read more about this function). The first argument to this function is a file, which can be expressed as either a file name or a path to a file. By default, R looks for the file in your current working directory To find the working directory, you can run `getwd()`. You can see in the following output that my current working directory is where the book content is on my computer.

```r
getwd()
```

```
[1] "/Users/Alice/Brown Dropbox/Alice Paul/DSCoV-Workshop-Intro-to-R"
```

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.

```r
setwd('/Users/Alice/Dropbox/DSCoV-Workshop-Intro-to-R/')
```

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. In the following code, I chose to specify the path to the downloaded .csv file, saved this file to an object called `pain`, and then printed that `pain` object.

```
# update this with the path to your file
pain <- read.csv("pain.csv")
```

We can see that `df` contains the information from the .csv file, and that R has printed the first few observations of the data.

Data Frames

A data frame is a two-dimensional data structure. 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 is character type. We can 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.

```
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.

```
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
colnames(weather_data)
```

```
[1] "day_of_week" "temp"        "rain"
```

```r
rownames(weather_data)
```

```
[1] "1" "2" "3" "4"
```

```r
names(weather_data)
```

```
[1] "day_of_week" "temp"        "rain"
```

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

```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
```

Indexing a Data Frame

We can select elements of the data frame using the indices or names. In the subsequent code, 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 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
```

The column `day_of_week` is a categorical column, but it can only take on a limited number of values. For this kind of column, it is often useful to convert that column to a factor as we did before.

```
weather_data$day_of_week <- factor(weather_data$day_of_week)
levels(weather_data$day_of_week)
```

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

Summarizing Data Columns

We 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 not all 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 column 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. 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)
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. 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
```

Column Summaries

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. We can see that the pain intensity values range from 0 to 10 with a median value of 7 and that there is a NA value.

```
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 is NA.

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

```
[1] 0
```

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

```
[1] 10
```

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

```
[1] 6.485271
```

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

```
[1] 7
```

Additionally, the following functions are helpful for summarizing quantitative columns.

- `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.

We can also use the `summary()` function for categorical variables. In this case, R finds the counts for each level.

```
summary(pain$PAT_SEX)
```

```
   Length     Class      Mode
    21659 character character
```

For categorical columns, it is also useful to use the `table()` function, which returns the counts for each possible value, instead of the `summary()` function. By default, `table()` ignores NA values. However, we can set `useNA="always"` if we also want to display the number of NA values in the table output. Additionally, we can use the `prop.table()` function to convert the counts to proportions. Using these functions, we can see that the column `PAT_SEX` column, which corresponds to the reported patient sex, has a single missing value, and we can also see that around 60% of patients are female.

```r
table(pain$PAT_SEX, useNA="always")
```

```

female   male   <NA>
 13102   8556      1
```

```r
prop.table(table(pain$PAT_SEX))
```

```

   female      male
0.6049497 0.3950503
```

Note that this column is not actually a factor column yet, which we can check using the `is.factor()` function. We can convert it to one using `as.factor()`.

```r
is.factor(pain$PAT_SEX)
```

```
[1] FALSE
```

```r
pain$PAT_SEX <- as.factor(pain$PAT_SEX)
is.factor(pain$PAT_SEX)
```

```
[1] TRUE
```

Missing, Infinite, and NaN Values

As we have seen, 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 following example, we can see that R figures out that 1 plus an unknown number is also unknown!

```
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. We observe that there is a single NA value for the column `PATIENT_NUM`, which is the patient ID number.

```
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. We find that row 11749 has all NA values.

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

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.

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

We create two new columns: first, we create a column for the change in pain at follow-up, and second, we create a column for the percent change in pain at follow-up.

```r
pain$PAIN_CHANGE <- pain$PAIN_INTENSITY_AVERAGE.FOLLOW_UP -
  pain$PAIN_INTENSITY_AVERAGE
summary(pain$PAIN_CHANGE)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
-10.000  -1.000   0.000  -0.338   1.000   8.000   14520
```

```r
pain$PERC_PAIN_CHANGE <- pain$PAIN_CHANGE /
  pain$PAIN_INTENSITY_AVERAGE
summary(pain$PERC_PAIN_CHANGE)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 -1.000  -0.200   0.000     Inf   0.143     Inf   14520
```

In the summary of the percent change, we can see that the maximum value is `Inf`. This is R's representation of infinity. This occurred because some patients have an initial pain score of 0, which creates infinite values when we divide through by this value to find the percent change. We can test whether something is infinite using the `is.infinite()` or `is.finite()` functions. This shows that there were three patients with infinite values. The value `-Inf` is used to represent negative infinity.

```r
sum(is.infinite(pain$PERC_PAIN_CHANGE))
```

```
[1] 3
```

Another special value in R is `NaN`, which stands for "Not a Number". For example, `0/0` results in a NaN value. We can test for `NaN` values using the `is.nan()` function.

```r
0/0
```

```
[1] NaN
```

Looking back at the missing values, there are two useful functions for selecting the complete cases in a data frame. The `na.omit()` function returns the data frame with incomplete cases removed, whereas `complete.cases()` returns TRUE/FALSE values for each row indicating whether each row is complete, which we can then use to select the rows with TRUE values. In the following code, we see that both approaches select the same number of rows.

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

```
[1] 2413   94
```

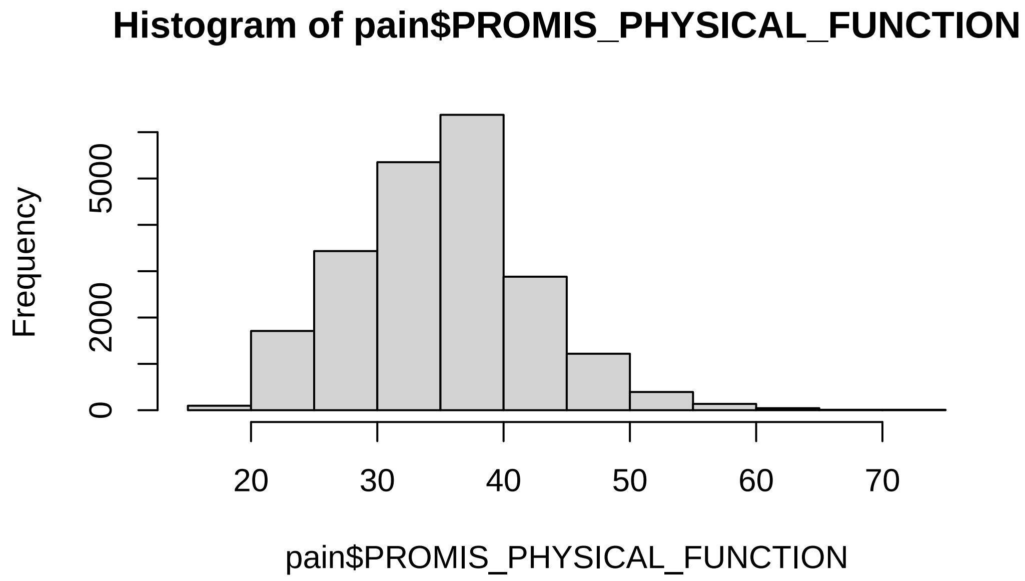
```r
dim(pain_sub2)
```

```
[1] 2413   94
```

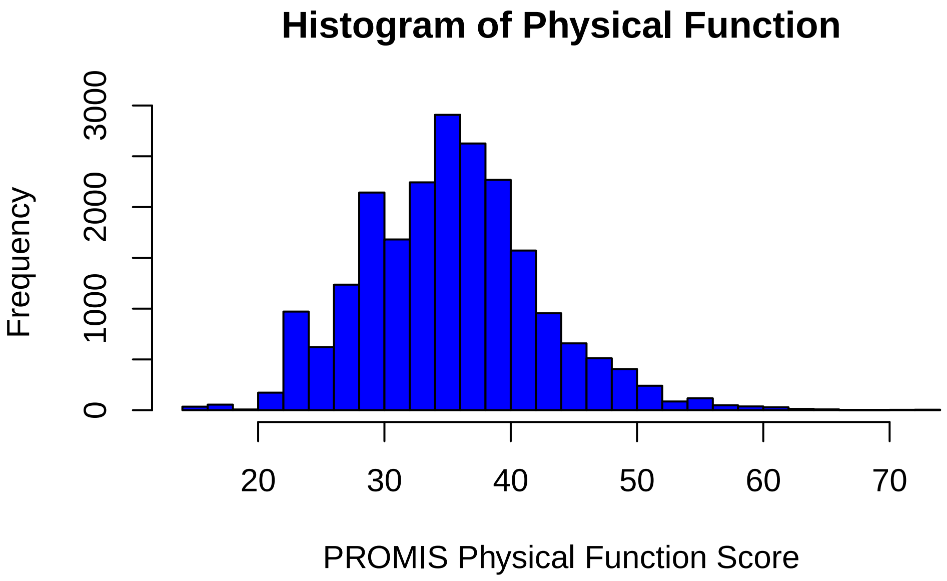
Simple Plots

If we want to create a histogram, we can use the `hist()` function. We can also include some of the other optional arguments to the `hist()` function to help polish the figure. For example, we may want to update the text in the title and x-axis. In the following code, we update the color, labels, and number of bins for the plot. The function `colors()` returns all recognized colors in R. The argument `breaks` specifies the number of bins to use to create the histogram, `col` specifies the color, `main` specifies the title of the plot, and `xlab` specifies the x-axis label (using `ylab` would specify the y-axis label). Read the documentation `?hist` for the full list of arguments available.

```r
hist(pain$PROMIS_PHYSICAL_FUNCTION)
```

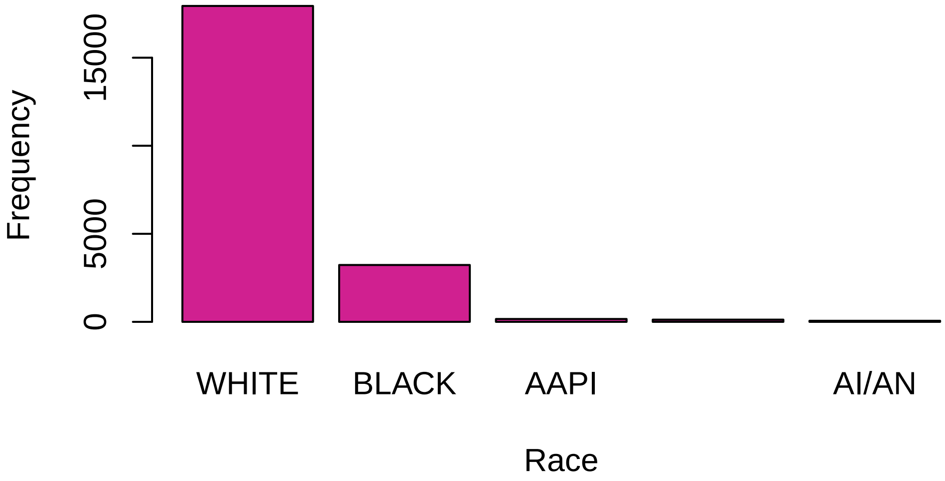


```
# with more options
hist(pain$PROMIS_PHYSICAL_FUNCTION, breaks = 30, col = "blue",
     main = "Histogram of Physical Function",
     xlab = "PROMIS Physical Function Score")
```



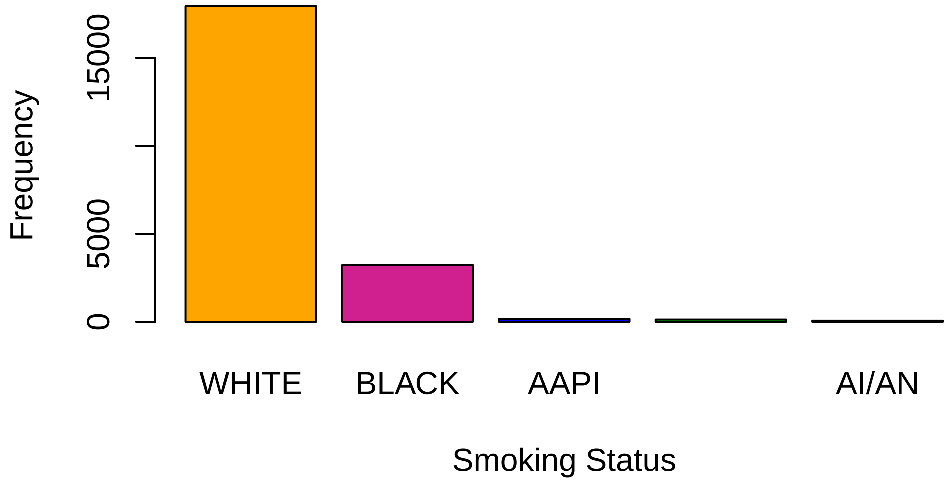
For categorical columns, we may want to plot the counts in each category using a barplot. The function `barplot()` asks us to specify the `names` and `heights` of the bars. To do so, we need to store the counts for each category. Again, we update the color and labels.

```
race_counts <- sort(table(pain$PAT_RACE), decreasing = TRUE)
barplot(height = race_counts, names = names(race_counts),
        col = "violetred", xlab="Race", ylab="Frequency")
```



With a barplot, we can even specify a different color for each bar. To do so, `col` must be a vector of specified colors with the same length as the number of categories.

```
barplot(height = race_counts, names = names(race_counts),
        col = c("orange", "violetred", "blue", "green", "yellow"),
        xlab = "Smoking Status", ylab = "Frequency")
```



Bivariate Distributions

We now turn our attention to relationships among multiple columns. When we have two categorical columns, we can use the `table()` function to find the counts across all combinations. For example, we look at the distribution of race by sex.

```
table(pain$PAT_SEX, pain$PAT_RACE)
```

```
        AAPI AI/AN BLACK NOT SPECIFIED WHITE
female    95    38  2178            83 10644
male      67    22  1051            43  7296
```

To look at the sample distribution of a continuous column stratified by a categorical column, we can call the `summary()` function for each subset of the data. In the subsequent code, we look at the distribution of blood lead level by sex and observe higher blood lead levels in male observations.

```
summary(pain$PROMIS_DEPRESSION[pain$PAT_SEX == "female"])
```

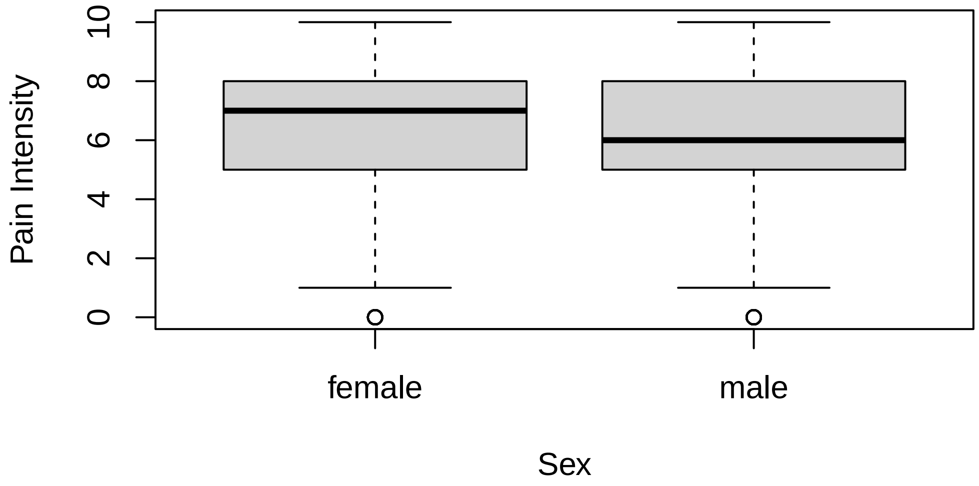
```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
  34.17   49.42   56.07   55.84   62.17   84.36      55 
```

```
summary(pain$PROMIS_DEPRESSION[pain$PAT_SEX == "male"])
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
  34.17   47.63   54.75   54.32   62.17   84.36      32 
```

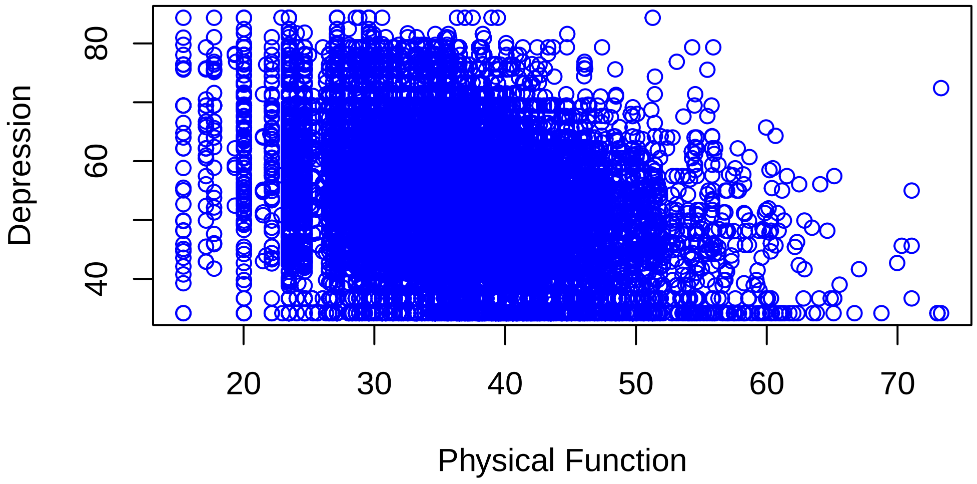
We can also observe this visually through a boxplot. When given one categorical column and one continuous column, the `plot()` function creates a boxplot. By default, the first argument is the x-axis and the second argument is the y-axis.

```
plot(pain$PAT_SEX, pain$PAIN_INTENSITY_AVERAGE, ylab = "Pain Intensity",
     xlab = "Sex")
```



To visualize the bivariate distributions between two continuous columns, we can use scatter plots. To create a scatter plot, we use the `plot()` function again. We use this function to show the relationship between systolic and diastolic blood pressure.

```
plot(pain$PROMIS_PHYSICAL_FUNCTION, pain$PROMIS_DEPRESSION, col = "blue",
    xlab = "Physical Function",
    ylab = "Depression")
```



The two measures look correlated. We can calculate their Pearson and Spearman correlation using the `cor()` function. The default method is the Pearson correlation, but we can also calculate the Kendall or Spearman correlation by specifying the method.

```
cor(pain$PROMIS_PHYSICAL_FUNCTION, pain$PROMIS_DEPRESSION, use = "complete.obs")
```

```
[1] -0.3729094
```

Tables

Another useful way to display information about your data is through tables. For example, it is standard practice in articles to have the first table in the paper give information about the study sample, such as the mean and standard deviation for all continuous columns and the proportions for categorical columns. The **gt** package is designed to create polished tables that can include footnotes, titles, column labels, etc. The **gtsummary** package is an extension of this package that can create summary tables.

```
library(gt)
```

```
Warning: package 'gt' was built under R version 4.4.1
```

```
library(gtsummary)
```

```
Warning: package 'gtsummary' was built under R version 4.4.1
```

We use the `tbl_summary()` function in the **gtsummary** package. The first argument to this function is again the data frame. By default, this function summarizes all the columns in the data. Instead, we use the `include` argument to specify a list of columns to include. We then pipe this output to the function `as_gt()`, which creates a gt table from the summary output. Note that the table computes the total number of observations and the proportions for categorical columns and the median and interquartile range for continuous columns.

```
tbl_summary(pain,
            include = c("AGE_AT_CONTACT", "PAT_SEX", "PAT_RACE", "BMI", "MEDICAID_BIN",
                        "PAIN_INTENSITY_AVERAGE"),
            label = list(AGE_AT_CONTACT = "Age",
                         PAT_SEX = "Sex",
                         PAT_RACE = "Race",
                         BMI = "BMI",
                         MEDICAID_BIN = "Medicaid",
                         PAIN_INTENSITY_AVERAGE = "Pain Intensity")) |>
  as_gt()
```

We can update our table by changing some of its arguments. This time, we specify that we want to stratify our table by follow-up status so that the table summarizes the data by this grouping. Additionally, we change how continuous columns are summarized by specifying that we want to report the mean and standard deviation instead of the median and interquartile range. We do this using the `statistic` argument. The documentation for the `tbl_summary()` function can help you format this argument depending on which statistics you would like to display.

```
pain$HAS_FOLLOW_UP <- !is.na(pain$PAIN_INTENSITY_AVERAGE.FOLLOW_UP)
tbl_summary(pain,
            include = c("AGE_AT_CONTACT", "PAT_SEX", "PAT_RACE", "BMI", "MEDICAID_BIN",
                        "PAIN_INTENSITY_AVERAGE"),
            label = list(AGE_AT_CONTACT = "Age",
                         PAT_SEX = "Sex",
                         PAT_RACE = "Race",
                         BMI = "BMI",
                         MEDICAID_BIN = "Medicaid",
                         PAIN_INTENSITY_AVERAGE = "Pain Intensity"),
            by = "HAS_FOLLOW_UP",
```

Characteristic	N = 21,658[1]
Age	57 (46, 68)
Sex	
female	13,102 (60%)
male	8,556 (40%)
Race	
AAPI	162 (0.8%)
AI/AN	60 (0.3%)
BLACK	3,229 (15%)
NOT SPECIFIED	126 (0.6%)
WHITE	17,940 (83%)
Unknown	141
BMI	29 (25, 35)
Unknown	5,632
Medicaid	4,601 (22%)
Unknown	300
Pain Intensity	7 (5, 8)

[1] Median (Q1, Q3); n (%)

```
            statistic = list(all_continuous() ~ "{mean} ({sd})")) |>
  add_p() |>
  as_gt()
```

Characteristic	**FALSE** N = 14,520[1]	**TRUE** N = 7,138[1]	**p-value**[2]
Age	56 (16)	57 (15)	<0.001
Sex			<0.001
female	8,671 (60%)	4,431 (62%)	
male	5,849 (40%)	2,707 (38%)	
Race			<0.001
AAPI	125 (0.9%)	37 (0.5%)	
AI/AN	30 (0.2%)	30 (0.4%)	
BLACK	2,295 (16%)	934 (13%)	
NOT SPECIFIED	96 (0.7%)	30 (0.4%)	
WHITE	11,860 (82%)	6,080 (86%)	
Unknown	114	27	
BMI	30 (8)	31 (8)	<0.001
Unknown	4,012	1,620	
Medicaid	3,263 (23%)	1,338 (19%)	<0.001
Unknown	208	92	
Pain Intensity	7 (2)	6 (2)	<0.001

[1] Mean (SD); n (%)

[2] Wilcoxon rank sum test; Pearson's Chi-squared test