Lecture_1_Introduction_to_R

2025-02-21

What is R Programming?

R is a functional programming language appropriate for data wrangling, cleaning, and visualization, statistical analysis, modeling, and other computational tasks. It was developed at the University of Auckland in 1993 by Ross Ihaka and Robert Gentleman to teach introductory statistics and is widely used in bioinformatics, epidemiology, finance, and social sciences due to its extensive statistical modeling capabilities and rich ecosystem of packages. R is also an excellent choice of language for publication ready figure production.

R supports built in base functions such as `print()` as well as higher level functions such as `lapply()` and `Reduce()`. R also supports functions provided by specific packages and allows for users to define their own functions. Whether a function is built into R, called from a specialized package, or generated by the user, functions can be assigned to a variable, passed to arguments, or returned from other functions. You can write R code in many text editors, but throughout this class we will be using R Studio to write and execute our code.

Coding in R Studio

A Brief Anatomy of R Studio

Prior to class you should have received an email with instructions for installing R and R Studio. Navigate to your R Studio install on your computer if you have not already done so. When you open R studio, you will see several windows: code editor, console, workspace and history, and plots and files.

Code Editor: The window in the upper left is referring to as the code editor window where you will write and execute your code.

Console: The console in the bottom left window shows the pieces of code that have been run. Importantly, error messages and warning will appear in this window after you have executed code.

Workspace and History: The workspace and history window in the upper right will display the names of objects as well as sizes of objects loaded into the environment.

Plots and Files: The bottom right window will display plots generated by your code. This window will also display any documentation related to functions if you run `?name_of_your_function()`. For example, try typing `print()` in your console and hitting enter. A description of the print function should appear in the plots and files window.

For comfort of use, R Studio also has an "Appearance" option under Tools > Global Options > Appearance. This option allows the user to specify a background theme and color. You might find it easier on your eyes to have a dark background.

File Generation in R Studio

Using R Studio we can generate several types of documents. Here we will discuss the file types with which you will be interacting over the course of our class: R Script: An R Script is a plain text file that contains lines of R codem usually written in sequetial order, It is the most basic file type for writing and running R code. R Scripts are great for writing and executing scripts, developing functions, running analyses, and automating tasks.

R Markdown: An R Markdown file combines R code with formatted text using Markdown syntax. It allows users to create dynamic documents, reports, presentation, and even websites that include both text and executable code. These sorts of files are great for creating reproducible research documents, technical reports, and presentations that combine code, analysis, and visualizations. In fact, this very file is an R Markdown file and we will be using them as a learning tool throughout the course of this class.

R Notebook: An R Notebook is a type of R Markdown file that provides an interactive interface for executing code chunks and immediately viewing results inline. R Notebooks are great for interactive data exploration and analysis with a mix of code and narrative, where results are displayed immediately without needing to knit the document.

R for Basic Calculations

To get our feet wet with R, let's execute some simple arithmetic operations. R leverages the following operators to perform simple calculations:

Function	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	ˆ

Let's implement a few of these operators here:

```r
# Simple Addition
6 + 7
```

```
## [1] 13
```

```r
# Simple Substraction
3 - 2
```

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

```r
# Simple Multiplication
8 * 1
```

```
## [1] 8
```

```r
# Simple Division
16 / 2
```

```
## [1] 8
```

```r
# Simple Exponentiation
5^2
```

```
## [1] 25
```

As you can see, we can use R as a calculator, but this exercise does not leverage the full extent of what R can do with these sorts of operations. In R, we can assign variables using the assignment operators `<-`, `=`, or `->`, though `<-` is the most commonly used. Assigning variables allows us to store values, making it easier to reuse data, perform calculations, and structure code efficiently. For example, `x <- 10` assigns the value 10 to x, allowing us to reference x later in computations. Variable assignment improves readability, reduces redundancy, and enables modular programming by storing results of functions or operations for further use. Additionally, variables in R can hold different data types, including vectors, data frames, and lists, making them essential for data analysis and statistical modeling.

Side Note: In the code chunk above, you will notice that I have added some lines that begin with `#`. The text following this symbol are referred to as comments and are not read by the computer, rather they are for the human reading your code. Comments are a method by which we can communicate what each line of code is doing to coworkers, managers, and anyone who comes across our code on a GitHub repo. It is incredibly important that your code be well commented.

In the following code we assign some of these numbers and their associated outputs to variables.

```r
# Assign 6 to a
a <- 6

# Assign 7 to b
b <- 7

# Assign output of a + b to c
c <- a + b

# Display c
c
```

```
## [1] 13
```

After running this code, check out your environment window. You should see "Values" come up displaying a as 6, b as 7, and c as 13. We have assigned values to our variables and used those variables to conduct calculations. While this is a simple example, we will work with more complex data structures later in this class and the power to implement the use of variable is instrumental in expedient analysis.

Additional Operators in R

In addition to arithmetic operators, R also allows the use of relational and logical operators. Here is a summary of relational operators:

Operator	Effect
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

Here are a few examples of how to implement these logicals in R:

```r
# Less than (<)
5 < 10
```

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

```r
# Greater than (>)
10 > 5
```

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

```r
# Less than or equal to (<=)
5 <= 5
```

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

```r
3 <= 5
```

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

```r
# Greater than or equal to (>=)
10 >= 10
```

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

```r
15 >= 10
```

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

```r
# Equal to (==)
5 == 5
```

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

```r
5 == 10
```

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

```r
# Not equal to (!=)
5 != 10
```

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

```r
5 != 5
```

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

Some additional operators you might find useful for filtering your data can be found the the table below:

Operator	Effect
!	NOT
&	AND

```r
# NOT (!)
x <- 5
!(x == 5)  # FALSE (because x is 5, and NOT TRUE is FALSE)
```

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

```r
!(x > 10)  # TRUE (because x > 10 is FALSE, and NOT FALSE is TRUE)
```

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

```r
# AND (&)
x <- 5
y <- 10

(x > 3) & (y < 15)   # TRUE (both conditions are TRUE: 5 > 3 AND 10 < 15)
```

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

```r
(x > 3) & (y > 15)   # FALSE (one condition is FALSE: 10 is not > 15)
```

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

```r
(x < 3) & (y < 15)   # FALSE (first condition is FALSE: 5 is not < 3)
```

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

```
# OR (|)
x <- 5
y <- 10

(x > 3) | (y > 15)   # TRUE (one condition is TRUE: 5 > 3, even though 10 is not > 15)
```

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

```
(x < 3) | (y > 15)   # FALSE (both conditions are FALSE: 5 is not < 3 AND 10 is not > 15)
```

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

```
(x == 5) | (y == 20) # TRUE (first condition is TRUE: x is 5, even though y is not 20)
```

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

Data Structures in R

During this lesson we have worked almost entirely with low complexity data, meaning there really is not much structure or organization to the information with which we are working. A data structure is a way of organizing and storing data so that it can be accessed and modified efficiently. In R, common data structures include vectors, list, matrices, and data frames, each serving different purposes based on how the data is structured and how it will be used. Understanding the data structures you're working with is crucial because it affects how you can manipulate the data, the operations you can conduct, and the performance of those operations. Let's take a look at our main data structures in R, starting with vectors.

Vectors

A vector is the simplest data structure in R, consisting of elements that are the same type. The data can be numeric, logical, character, or complex data types, but all elements in a vector must be the same type!

Vectors can be created using the `c()` function (standing for combine), which concatenates its arguments together into a single vector. `c()` can be used in conjunction with the assignment operator `<-` which tells R you want to assign the vector to a specific variable. Below, we display how you can generate vectors containing different types of data:

```
# numeric
x <- c(1.63, 2.25, 3.83, 4.99)

# integer
x <- as.integer(c(1, 2, 3, 4))

# character
x <- as.character(c("a", "b", "c", "d"))

# logical
x <- c(TRUE, FALSE, TRUE, TRUE)
```

Let's say you are given a vector like the one below. How can you tell what type of data it holds?

```
x <- c(2, "a", TRUE, "87", "apple")
```

We can implement the `class()` to determine the class or type of an object. When used on a vector, the `class()` function tells you whether the vector is of a specific type, such as numeric, character, logical, or integer, depending on the elements contained in the vector.

```
# Numeric vector
num_vector <- c(1, 2, 3)
class(num_vector)  # Returns "numeric"
```

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

```
# Character vector
char_vector <- c("apple", "banana", "cherry")
class(char_vector)  # Returns "character"
```

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

```
# Logical vector
log_vector <- c(TRUE, FALSE, TRUE)
class(log_vector)  # Returns "logical"
```

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

Still, vectors can be coerced from one class to another using function written to modify their attributes.

```
x <- c(1, 2, 3, 4)
as.character(x)
```

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

```
x <- c(TRUE, FALSE, TRUE, TRUE)
as.numeric(x)
```

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

```
x <- c(1.63, 2.25, 3.83, 4.99)
as.integer(x) # Note that this is different from the round() function!
```

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

You may also find it helpful to explor how many items are in a vector using the `length()` function.

```
x <- c("f","o","u","n","d","a","t","i","o","n","s","o","f","d","a","t","a","s","c","i","e","n","c","e")
length(x)
```

```
## [1] 24
```

Vectors can be combined or nested to create a single vector, or evaluated against each other:

```
# combine a vector and a nested vector
x <- c(1, 2, 3, 4, c(1, 2, 3, 4))
x
```

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

```
# multiply two integer vectors
y <- c(2, 2, 2, 2)
x * y
```

```
## [1] 2 4 6 8 2 4 6 8
```

It is also important to know how to access data in a vector. In R, accessing data in a vector is simple and done using indexing, R uses 1-bsed indexing, meaning the first element of a vector has an index of 1. Here is an example of how to access elements in a vector in R:

```
# Define a numeric vector
x <- c(5,10, 15, 20, 25, 30)

```

```r
# Returns 5
x[1]
```

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

```r
# Returns 15
x[3]
```

```
## [1] 15
```

```r
# You also have the power to access multiple elements
# Returns 5 and 15
x[c(1,3)]
```

```
## [1]  5 15
```

```r
# You can access sequences of data
# Returns 10, 15, 20, and 25
x[2:4]
```

```
## [1] 10 15 20
```

Using negative indices excludes specific elements. This means the element at the specified index will not be included in the result:

```r
# Returns 5, 15, 20, 25, 30
x[-2]
```

```
## [1]  5 15 20 25 30
```

```r
# Returns 10, 15, 20, 30
x[c(-1,-5)]
```

```
## [1] 10 15 20 30
```

Up front, many data structures can contain `NA` values. You will even find `NA` values in data you download from public repositories, so make sure you explore your data when you get it and check for `NA` values! You can do so using the command here:

```r
x <- c(1,2, NA, 3, 4)

y <- is.na(x)
y
```

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

```r
# What does "y" contain and why?
class(y)
```

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

A factor in R is a data structure used to represent categorical data, which consists of a limited number of unique values (also known as levels). Factors are particularly useful when dealing with qualitative data, such as grouping variables or categorical responses, because they store both the values and the corresponding levels in an efficient way.

You might store subject disease disease_status for all the subjects in your study as a factor, with the levels "Covid+" and "Covid-".

```r
# Vector with disease disease_status
x <- c("Covid+", "Covid+", "Covid-", "Covid+", "Covid-")

```

```r
# factor() forces a factorization of the data in your vector
x <- factor(x, levels = c("Covid+", "Covid-"))

# Check the class and factor levels
class(x)
```

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

```r
levels(x)
```

```
## [1] "Covid+" "Covid-"
```

```r
# The table() function is very helpful to count instances of the levels
table(x)
```

```
## x
## Covid+ Covid-
##      3      2
```

Lists

A list in R is a flexible data structure that can store multiple types of data, including vectors, matrices, data frames, other lists, and even functions. Unlike vectors, which just contain elements of the same data type, lists can hold a mix of different types, making then a powerful tool for organizing complex data sets.

Much like with vectors, the data in lists are stored in sequence and can be accessed by index or name. Let's create a list of student names, ages, and heights.

```r
students <- list(name = c("Josh", "Rebecca", "Lamar"), age = c(24, 21, 32), height = c("5'11","5'1","5'

print(students)
```

```
## $name
## [1] "Josh"    "Rebecca" "Lamar"
##
## $age
## [1] 24 21 32
##
## $height
## [1] "5'11" "5'1"  "5'9"
```

```r
# We can access the heights of the students by directly calling that element

students[["height"]]
```

```
## [1] "5'11" "5'1"  "5'9"
```

```r
# Alternatively, we can use the $ symbol to access elements
students$height
```

```
## [1] "5'11" "5'1"  "5'9"
```

```r
# The same can be achieved by indicating the index of height
students[3]
```

```
## $height
## [1] "5'11" "5'1"  "5'9"
```

```r
# Let's say we want to access the first entry in height, we can do that using the command below
students[[3]][1]
```

```
## [1] "5'11"
```

```
# We can also change the names of the items in our lists

names(students) <- c("Name","Age","Height")
```

Now we have previously discussed how important it is to understand the structure of our data. One really handy function that you can use to quickly assess the structure of your data is `str()` along with our previously described `length()`.

```
str(students)
```

```
## List of 3
##  $ Name  : chr [1:3] "Josh" "Rebecca" "Lamar"
##  $ Age   : num [1:3] 24 21 32
##  $ Height: chr [1:3] "5'11" "5'1" "5'9"
```

```
length(students)
```

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

Matrices

In my line of work, I process a lot of RNA-seq data. After completing an RNA-sequencing run, you will be provided a count matrix with sample names across the top of your text file and gene names down the left hand side. This file is used for downstream processing. A matrix in R is a two-dimensional data structure that contains elements of the same type (numeric, character, or logical). It is similar to a data frame, but more restricted, as all elements in a matrix must be of the same type. Matrices are particularly useful for mathematical operations and manipulation.

Today, we will be creating a count matrix of our own to familiarize ourselves with vectors and data frames. We will begin by generating a matrix with 10 columns and 10 rows of random numbers between 0 and 10.

First we create a vector of numbers from 0 to 10:

```
num.vector <- c(0:10)
```

We have our computer randomly select 100 numbers from our num.vector and put it in a vector of size 100. You might notice the argument replace=TRUE. This tell the computer to sample from our num.vector with replacement, meaning each number can be chosen to be put in the count.vector more than once.

```
count.vector <- sample(num.vector, size = 100, replace = TRUE)
count.vector
```

```
##   [1]  8  1  9  6  4  9  7 10  4  1 10  6  0  1  9  5  4  3  5  9  4 10  5  3  5
##  [26]  3 10  8  4  7  5 10  7  0  0  9  8  5  7  5  6  6  2  4  4  5  9  9  9  3
##  [51]  0  4  7  7  5  7  7 10  2  8  2  6  4  1  3 10  5  0  0  3  4  1 10  7  2
##  [76]  1  9  6  6  1  2  0  3  7  3  4  2  6  6 10 10  5  1  0  5  2  8  9  5  5
```

We now create a matrix using our count.vector. We tell R that we want a matrix with 10 rows and 10 columns with the data in count.vector. byrow means that we are arranging the data row-wise instead of column-wise, which is the default in R.

```
count.matrix <- matrix(count.vector, ncol=10, nrow=10, byrow = TRUE)
count.matrix
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    8    1    9    6    4    9    7   10    4     1
##  [2,]   10    6    0    1    9    5    4    3    5     9
##  [3,]    4   10    5    3    5    3   10    8    4     7
##  [4,]    5   10    7    0    0    9    8    5    7     5
##  [5,]    6    6    2    4    4    5    9    9    9     3
##  [6,]    0    4    7    7    5    7    7   10    2     8
##  [7,]    2    6    4    1    3   10    5    0    0     3
##  [8,]    4    1   10    7    2    1    9    6    6     1
##  [9,]    2    0    3    7    3    4    2    6    6    10
## [10,]   10    5    1    0    5    2    8    9    5     5
```

Now that we have created a matrix of random whole numbers for our count matrix, we need to add sample names and genes. I mentioned previously that our sample names will be the column headers and the row names will be the gene names. Hence, we will be needing 10 sample names and 10 gene names for our dataset.

```
rownames(count.matrix) <- c("gene_1", "gene_2", "gene_3","gene_4","gene_5","gene_6","gene_7","gene_8","
colnames(count.matrix) <- c("subject_1", "subject_2", "subject_3", "subject_4","subject_5","subject_6",
count.matrix
```

```
##         subject_1 subject_2 subject_3 subject_4 subject_5 subject_6 subject_7
## gene_1          8         1         9         6         4         9         7
## gene_2         10         6         0         1         9         5         4
## gene_3          4        10         5         3         5         3        10
## gene_4          5        10         7         0         0         9         8
## gene_5          6         6         2         4         4         5         9
## gene_6          0         4         7         7         5         7         7
## gene_7          2         6         4         1         3        10         5
## gene_8          4         1        10         7         2         1         9
## gene_9          2         0         3         7         3         4         2
## gene_10        10         5         1         0         5         2         8
##         subject_8 subject_9 subject_10
## gene_1         10         4          1
## gene_2          3         5          9
## gene_3          8         4          7
## gene_4          5         7          5
## gene_5          9         9          3
## gene_6         10         2          8
## gene_7          0         0          3
## gene_8          6         6          1
## gene_9          6         6         10
## gene_10         9         5          5
```

Challenge: We can use a coding shortcut here! It's easy to make typos while writing out all the gene and sample names. Let's use the paste function to make things easier for us. Here we are telling R to make the first part of our name 'gene' and 'sample' respectively. Then, we are telling R to add the numbers 1 through 10 to the end of each sample or gene name.

```
rownames(count.matrix) <- paste('gene',1:10,sep='_')
colnames(count.matrix) <- paste('subject',1:10,sep='_')
count.matrix
```

```
##         subject_1 subject_2 subject_3 subject_4 subject_5 subject_6 subject_7
## gene_1          8         1         9         6         4         9         7
## gene_2         10         6         0         1         9         5         4
## gene_3          4        10         5         3         5         3        10
## gene_4          5        10         7         0         0         9         8
## gene_5          6         6         2         4         4         5         9
## gene_6          0         4         7         7         5         7         7
## gene_7          2         6         4         1         3        10         5
## gene_8          4         1        10         7         2         1         9
## gene_9          2         0         3         7         3         4         2
## gene_10        10         5         1         0         5         2         8
##          subject_8 subject_9 subject_10
## gene_1         10         4          1
## gene_2          3         5          9
## gene_3          8         4          7
## gene_4          5         7          5
## gene_5          9         9          3
## gene_6         10         2          8
## gene_7          0         0          3
## gene_8          6         6          1
## gene_9          6         6         10
## gene_10         9         5          5
```

You can access data in the matrix using the commands below.

```
# Let's use str() as well as dim() to get a better understanding of the structure and number of rows an
str(count.matrix)
```

```
##  int [1:10, 1:10] 8 10 4 5 6 0 2 4 2 10 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:10] "gene_1" "gene_2" "gene_3" "gene_4" ...
##   ..$ : chr [1:10] "subject_1" "subject_2" "subject_3" "subject_4" ...
```

```
dim(count.matrix)
```

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

```
# Here is how you can access elements in your count matrix

# Rows
count.matrix[1,]
```

```
##  subject_1  subject_2  subject_3  subject_4  subject_5  subject_6  subject_7
##          8          1          9          6          4          9          7
##  subject_8  subject_9 subject_10
##         10          4          1
```

```
# Columns
count.matrix[,2]
```

```
##  gene_1  gene_2  gene_3  gene_4  gene_5  gene_6  gene_7  gene_8  gene_9 gene_10
##       1       6      10      10       6       4       6       1       0       5
```

```
# Rows and Columns
count.matrix[3,2]
```

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

```
# You can also use class() to describe your data
class(count.matrix)
```

```
## [1] "matrix" "array"
```

You can modify matrices, changing entire rows/ columns or individual elements using the above accesstion techniques:

```
# Let's generate a matrix
my_matrix <- matrix(1:9, nrow = 3, byrow = TRUE)

# Change a single element
my_matrix[1, 2] <- 100
```

```r
# Change an entire row
my_matrix[2, ] <- c(10, 20, 30)

# Change an entire column
my_matrix[, 3] <- c(5, 10, 15)
```

You can add rows and columns to matrices using `rbind()` and `cbind()`.

```r
new_col <- c(10, 20, 30)
my_matrix <- cbind(my_matrix, new_col)

new_row <- c(100, 200, 300, 400)
my_matrix <- rbind(my_matrix, new_row)
```

One last handy function is `t()`. This function give you the power to transpose your matrix, making the row columns and the columns rows.

```r
t(my_matrix)
```

```
##                   new_row
##           1 10  7     100
##         100 20  8     200
##           5 10 15     300
## new_col  10 20 30     400
```

Of course, you can use matrices to conduct more involved calculations like addition, subtraction, and multiplication, as displayed below:

```r
# Create two matrices
A <- matrix(1:4, nrow = 2)
B <- matrix(5:8, nrow = 2)

# Perform element-wise addition and subtraction
A + B
```

```
##      [,1] [,2]
## [1,]    6   10
## [2,]    8   12
```

```r
A - B
```

```
##      [,1] [,2]
## [1,]   -4   -4
## [2,]   -4   -4
```

```r
# Element-wise multiplication
A * B
```

```
##      [,1] [,2]
## [1,]    5   21
## [2,]   12   32
```

```r
# True matrix multiplication
A %*% B
```

```
##      [,1] [,2]
## [1,]   23   31
## [2,]   34   46
```

Data Frames

A data frame in R is a tow-dimensional data structure that is similar to a matrix, but allows different types of data in different columns. It is one of the most commonly used structures for handling tabular data, making it an essential tool for data analysis and manipulation.

In the previous section we generated a count matrix called `count.matrix` that contained pseudo-RNA-seq count data. Let's build a data frame that contains associated mate data about each of the subjects in our study.

```r
df <- data.frame(subject_id = c("subject_1", "subject_2", "subject_3", "subject_4","subject_5","subject
                 age = c(45, 83, 38, 23, 65, 40, 32, 89, 77, 53),
                 sex = c("female", "female", "male", "female", "female", "male", "female","male","male"
                 disease_status = c("case", "case", "control", "control","case","case","case","control"

str(df)
```

```
## 'data.frame':    10 obs. of  4 variables:
##  $ subject_id    : chr  "subject_1" "subject_2" "subject_3" "subject_4" ...
##  $ age           : num  45 83 38 23 65 40 32 89 77 53
##  $ sex           : chr  "female" "female" "male" "female" ...
##  $ disease_status: chr  "case" "case" "control" "control" ...
```

Here we created a data frame containing subject id, age, sex, and disease disease_status information. These will be the columns of our data frame. Let's access the data in our data frame. Data frames can be subset in similar ways to matrices using brackets or the $ subsetting operator. Columns/variables can also be added using the $ operator.

```r
# get first row
df[1,]
```

```
##   subject_id age    sex disease_status
## 1  subject_1  45 female           case
```

```r
# get first column
df[,1]
```

```
##  [1] "subject_1"  "subject_2"  "subject_3"  "subject_4"  "subject_5"
##  [6] "subject_6"  "subject_7"  "subject_8"  "subject_9"  "subject_10"
```

```r
# get sex variable/column
df[, c("sex")]
```

```
##  [1] "female" "female" "male"   "female" "female" "male"   "female" "male"
##  [9] "male"   "male"
```

```r
# # get sex and disease_status
df[, c("sex", "disease_status")]
```

```
##       sex disease_status
## 1  female           case
## 2  female           case
## 3    male        control
## 4  female        control
## 5  female           case
## 6    male           case
## 7  female           case
## 8    male        control
## 9    male        control
## 10   male           case
```

```
# get the sex variable with $
df$sex
```

```
##  [1] "female" "female" "male"   "female" "female" "male"   "female" "male"
##  [9] "male"   "male"
```

```
# add a column for smoking disease_status
df$smoking_status <- c("former", "none", "heavy", "none","none","heavy","heavy","heavy","former","forme
```

Remember those relational operator we discussed at the beginning of this lesson? We can put them to good use here to filter our data.

```
# obtain a logical indicating which subjects are female
df$sex == "female"
```

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

```
# use logical to subset the data frame for only female subjects (rows)
df2 <- df[df$sex == "female", ]

# check dimensions of the new data frame
dim(df2)
```

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

```
# use the LOGICAL NOT operator ! to obtain only male subjects
df[!df$sex == "female", ]
```

```
##    subject_id age  sex disease_status smoking_status
## 3   subject_3  38 male        control          heavy
## 6   subject_6  40 male           case          heavy
## 8   subject_8  89 male        control          heavy
## 9   subject_9  77 male        control         former
## 10 subject_10  53 male           case         former
```

```
# this could obviously also be achieved with..
df[df$sex == "male", ]
```

```
##    subject_id age  sex disease_status smoking_status
## 3   subject_3  38 male        control          heavy
## 6   subject_6  40 male           case          heavy
## 8   subject_8  89 male        control          heavy
## 9   subject_9  77 male        control         former
## 10 subject_10  53 male           case         former
```

Loops in R

If/Else Statements in R

Functions

Saving Objects and Other Files in R