Intro to R

QBS Bootcamp 2025

Introduction to R

Learning Objectives

- Download and install R (version 3 or higher)
- Download and install R Studio (version 1.3 or higher)
- Run basic commands in R Console
- Run basic commands in R Studio
- Declare variables in R
- Input data in to R
- Output data from R
- Write a simple R script

Materials

- R version 3.0 or higher https://cran.rstudio.com/
- R Studio 1.3 or higher https://rstudio.com/products/rstudio/download/#download

R Programming

R is a free, open source programming language and statistical software environment, first released in 1993, that is used extensively in bioinformatics. Beyond the basic functionality included in R's standard distribution, an enormous number of packages designed to extend R's functionality for specific applications exist, representing one of R's core strengths.

R is also a very powerful way to create high quality graphics, using both functionality in base R as well as graphics specific packages, such as ggplot2. These packages provide a high level of user control, meaning almost all plotting features can be controlled. Importantly, numerous R packages provide functionality for generating biology and health care science specific visualizations.

Installing R

We will be using R-Studio to explore our example data. The latest versions for R and R-Studio can be found here and here.

Confirm R and R Studio are installed by double clicking the R Studio icon. This icon may be on your Desktop or in your Applications folder. In the Console window copy the following:

```r
R.Version()
```

```
## $platform
## [1] "aarch64-apple-darwin20"
##
## $arch
## [1] "aarch64"
##
## $os
## [1] "darwin20"
##
## $system
## [1] "aarch64, darwin20"
##
## $status
## [1] ""
##
## $major
## [1] "4"
##
## $minor
## [1] "4.2"
##
## $year
## [1] "2024"
##
## $month
## [1] "10"
##
## $day
## [1] "31"
##
## $`svn rev`
## [1] "87279"
##
## $language
## [1] "R"
##
## $version.string
## [1] "R version 4.4.2 (2024-10-31)"
##
## $nickname
## [1] "Pile of Leaves"
```

```
# The "#" hashtag indicates a comment and means that the R interpreter will ignore this line.
```

In the above code block we added a comment. Comments are VERY important when programming to add details about our code or even information about data we are pulling from outside sources. This is useful if we are revisiting a program we wrote months ago or sharing code with a friend.

Now, back to R Studio itself. You may be wondering why do we need R Studio if we already have R installed? RStudio is an IDE (Integrated Development Environment). An IDE is software built to consolidate different aspects of writing, executing, and evaluating computer code. Without an IDE, these aspects of programming would need to be performed in different applications, potentially reducing productivity.

Basic features of the RStudio IDE include:

- console for submitting code to
- syntax-highlighting editor used for writing R-scripts
- windows for environment management, data visualization, and debugging
- facilities for version control & project management

There are options for different types of files in R Studio. Among these are “R Scripts” where you can run all your code with the click of a button and “R Markdowns”, which give you the power to run your code in chunks. R Markdown files also give you the opportunity to knit your code and it’s output into a PDF. Comments in R Markdown files can be written in between code chunks.

R as a Functional Programming Language

R is generally considered a functional programming language. Without going into detail, this essentially means that the way in which R performs tasks and solves problems is centered around functions.

Functions are specific pieces of code that take a defined input, perform an operation or series of operations on the input, and return an output, again in a defined format.

One of the simplest functions is the `print` function. `print` will print the argument(s) provided to it as input to the R console as outputs. In the below code chunk, the input “Hello” is being provided to the print() function as input to its first argument.

```r
print("Hello")
```

```
## [1] "Hello"
```

Manual/help pages for a specific function can be obtained using ?. To bring up the manual page for the print() function:

```r
?print()
```

Orienting Yourself

Now that we understand our programming environment let’s start with a gentle introduction to programming using R as a calculator to orient ourselves. In the Console window of R Studio enter the following code line by line:

```r
4 + 5
```

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

```r
9 - 1
```

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

```r
8 / 2
```

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

```
6^2
```

```
## [1] 36
```

```
7 * 5
```

```
## [1] 35
```

We can expand R's functionality by setting variables for some of these equations. Enter the below code in the Console window line by line:

```
x = 4 + 5

x
```

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

```
y = 9 - 1

y
```

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

```
z = x * y

z
```

```
## [1] 72
```

As you can see from the above code, each variable is representative of the result of it's respective equation. We can even conduct calculations with the variables. This becomes important when our variables have more complex structures as seen in vectors, matrices, and dataframes.

Data Structures

Vectors

Vectors can only hold one type of data (a property referred to as being atomic). In R, five basic object classes exist:

- numeric - real numbers (e.g. 33.3334)
- integer - whole numbers (e.g. 42)
- character - strings of characters (e.g. letters, words, sentences)
- logical - `TRUE` or `FALSE` (commonly called 'Boolean' values elsewhere)
- complex - numbers with real and imaginary parts

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 that vector to a specific variable.

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

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

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

# logical
d <- c(TRUE, FALSE, TRUE, TRUE)

# mixed?
e <- c(1, "a")
```

Each object class has specific attributes, which we can extract using the appropriate accessor functions. For example, the class of an object is itself an attribute that can be obtained using the `class()` function:

```r
class(a)
```

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

```r
class(c)
```

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

```r
class(e)
```

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

Another important attribute is length. For example, if we want to know how many elements are in a character string, we can use the `length()` function.

```r
length(x)
```

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

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

```r
# 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
```

```r
# create another vector
y <- c(2, 2, 2, 2)
y
```

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

```r
# multiply two integer vectors
x * y
```

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

Even though vectors are atomic, they can be coerced from one class to another using functions written to modify their attributes. e.g.

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

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

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

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

```r
x <- c(1.63, 2.25, 3.83, 4.99)
as.integer(x)
```

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

Elements within vectors can be subset or indexed based on their position in that vector. Individual elements can also be assigned names, which can also be used to perform indexing.

```r
# define a character vector
x <- c("a", "b", "c", "d")

# get elements 1 and 3
x[c(1,3)]
```

```
## [1] "a" "c"
```

```r
# get elements 1 to 3 using the ':' operator
x[c(1:3)]
```

```
## [1] "a" "b" "c"
```

```r
# define a numeric vector
x <- c(1.63, 2.25, 3.83, 4.99)

# assign it names
names(x) <- c("gene 1", "gene 2", "gene 3", "gene 4")

# index for specific element
x["gene 1"]
```

```
## gene 1
##   1.63
```

Vectors can contain missing values, defined by `NA` and `NaN`. These elements can be identified with the functions `is.na()` or `is.nan()`:

```
x <- c(1.63, NA, 3.83, 4.99)
x
```

```
## [1] 1.63   NA 3.83 4.99
```

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

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

```
# what object class is returned
class(x.na)
```

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

Operators

We introduced two operators in the examples above, the assignment operator `<-` and the sequence operator `:`. Operators are essentially symbols that tell R how you would like to relate the operands on either side of the symbol. In R, operators can be broadly categorized into assignment, arithmetic, relational, and logical.

We have explored several operators when we used R as a calculator earlier in this class. These are known as arithmetic operators like `+` and `*`. A summary of arithmetic operators can be found below:

- addition: +
- subtraction: -
- multiplication: *
- division: /
- exponentiation: ˆ

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

- less than: <
- greater than: >
- less than or equal to: <=
- greater than or equal to: >=
- equal to: ==
- not equal to: !=

Some example usage of relational operators:

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

# which elements are less than 3
x < 3
```

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

```r
# which elements are less than or equal to 3
x <= 3
```

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

```r
# define a character string
x <- c("a", "b", "c", "d", "a")

# which elements are equal to a
x == "a"
```

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

Logical operators are a great way to filter your data. Here is a list of logical operators you may find useful:

- NOT: !
- AND: &
- OR: |

Some example usage of logical operators:

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

# which elements are NOT equal to 4
x != 4
```

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

```r
# which could also be achieved with
!x == 4
```

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

```r
# which elements are less than 2 or equal to 4
x < 2 | x ==4
```

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

```r
# which elements are less than 2 AND equal to 4
x < 2 & x == 4
```

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

You can also select for indices of elements in a vector using the `which` command.

```r
#This will return the indices of the elements in the vector that are not equal to 4
which(x != 4)
```

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

```r
#It is also useful to use which to find out how many elements meet your parameters
length(which(x !=4))
```

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

Note: When combining operators, operator precedence applies, such that operators with high precedence will be evaluated first. For example, in the above line, `x < 2` will be evaluated before `x == 4` as the `<` has greater precedence than `==`. You can explore operator precedence in R using the main page returned by `?Syntax`.

Relational and logical operators can be used to subset a vector based on the values returned by the operator, and the brackets, as we did above for specific elements.

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

# subset x for values less than 3
x_sub <- x[x < 3]
x_sub
```

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

```r
# define a character string
x <- c("a", "b", "c", "d", "a")

# subset x for elements equal to a
x[x == "a"]
```

```
## [1] "a" "a"
```

Factors

Factors are a special instance of vectors where only predefined values, called levels can be included in the vector. Such vectors are useful when you know that elements of a vector should take on one of those predefined values.

Categorical data is often stored in vectors, making them a very important object class when you start doing any statistical modeling in R. For example, you might store subject sex for all the subjects in your study as a factor, with the levels male and female.

```r
# make a character vector with only male or female as entries
x <- c("female", "female", "male", "female", "male")

# use factor() constructor function to generate the factor
x <- factor(x, levels = c("female", "male"))

# confirm the class and check the levels
class(x)
```

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

```
levels(x)
```

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

```
# use table() to count up all the instances of each level
table(x)
```

```
## x
## female   male
##      3      2
```

Lists

Sometimes, it may be desirable to store multiple vectors, or even vectors of different object classes, in the same R overall object. Lists are a special object class that permits objects with these attributes, making them distinct from atomic vectors.

In the same way that vectors and factors are constructed using `c()` and `factors()` respectively, lists are created using the `list()` constructor function.

```
x <- list(c(1.63, 2.25, 3.83, 4.99),
          c(2.43, 8.31, 3.12, 7.25),
          c(1.29, 3.23, 3.48, 0.23))

# the structure function str() can be useful to examine the composition of a list
str(x)
```

```
## List of 3
##  $ : num [1:4] 1.63 2.25 3.83 4.99
##  $ : num [1:4] 2.43 8.31 3.12 7.25
##  $ : num [1:4] 1.29 3.23 3.48 0.23
```

```
# confirm the length
length(x)
```

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

```
# lists can be subset using brackets
### subset for first element of list
x[[1]]
```

```
## [1] 1.63 2.25 3.83 4.99
```

```
### subset for first element of first vector in list
x[[1]][1]
```

```
## [1] 1.63
```

```
# lists elements can be given names using a character vector equal to list length
names(x) <- c("gene_1", "gene_2", "gene_3")

# names can also be used to subset a list
x[["gene_1"]]
```

```
## [1] 1.63 2.25 3.83 4.99
```

```
# subsetting can also be achieved with the $ subsetting operator
x$gene_1
```

```
## [1] 1.63 2.25 3.83 4.99
```

In our example list, all three vectors stored in the list are numeric, however as mentioned above, lists can store vectors of different classes.

```
x <- list(c(1.63, 2.25, 3.83, 4.99),
          c(TRUE, FALSE, TRUE, TRUE),
          c("a", "b", "c", "d"))

# use the structure function str()to examine the composition of the list
str(x)
```

```
## List of 3
##  $ : num [1:4] 1.63 2.25 3.83 4.99
##  $ : logi [1:4] TRUE FALSE TRUE TRUE
##  $ : chr [1:4] "a" "b" "c" "d"
```

Creating A Count Matrix

If you are producing RNA-seq data, once sequencing is completed on your experiment 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. 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]  9  6  7  4  3  6  1  3  1  9  1  4  4  4  2  8 10  7  6  5  6  5  1  1  8
##  [26]  4  1 10 10  8  2  2  6  4  4  4  3  4  0  3  9  9  4  1  8 10  2 10  1  2
##  [51]  8  9  2  4 10  3  8  8  9  9  7  5  3  5  0  5  2  0  9  9 10  7  8  8  7
##  [76]  2  4  5  2 10  9  9  3 10  6  6  0  2  8  3  8  8  7  7  6  5  2  3  7  8
```

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

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          9         6         7         4         3         6         1
## gene_2          1         4         4         4         2         8        10
## gene_3          6         5         1         1         8         4         1
## gene_4          2         2         6         4         4         4         3
## gene_5          9         9         4         1         8        10         2
## gene_6          8         9         2         4        10         3         8
## gene_7          7         5         3         5         0         5         2
## gene_8         10         7         8         8         7         2         4
## gene_9          9         9         3        10         6         6         0
## gene_10         8         8         7         7         6         5         2
##         subject_8 subject_9 subject_10
## gene_1          3         1          9
## gene_2          7         6          5
## gene_3         10        10          8
## gene_4          4         0          3
## gene_5         10         1          2
## gene_6          8         9          9
## gene_7          0         9          9
## gene_8          5         2         10
## gene_9          2         8          3
## gene_10         3         7          8
```

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          9         6         7         4         3         6         1
## gene_2          1         4         4         4         2         8        10
## gene_3          6         5         1         1         8         4         1
## gene_4          2         2         6         4         4         4         3
## gene_5          9         9         4         1         8        10         2
## gene_6          8         9         2         4        10         3         8
## gene_7          7         5         3         5         0         5         2
## gene_8         10         7         8         8         7         2         4
## gene_9          9         9         3        10         6         6         0
## gene_10         8         8         7         7         6         5         2
##         subject_8 subject_9 subject_10
## gene_1          3         1          9
## gene_2          7         6          5
## gene_3         10        10          8
## gene_4          4         0          3
## gene_5         10         1          2
## gene_6          8         9          9
## gene_7          0         9          9
## gene_8          5         2         10
## gene_9          2         8          3
## gene_10         3         7          8
```

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

```
# check the structure and dimensions with dim()
str(count.matrix)
```

```
##  int [1:10, 1:10] 9 1 6 2 9 8 7 10 9 8 ...
##  - 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
```

```
# specific elements can be obtained through subsetting
### row 1
count.matrix[1,]
```

```
##  subject_1  subject_2  subject_3  subject_4  subject_5  subject_6  subject_7
##          9          6          7          4          3          6          1
##  subject_8  subject_9 subject_10
##          3          1          9
```

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

```
##  gene_1  gene_2  gene_3  gene_4  gene_5  gene_6  gene_7  gene_8  gene_9 gene_10
##       6       4       5       2       9       9       5       7       9       8
```

```
### element 2 of row 3
count.matrix[3,2]
```

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

```
# check class of the object and one row
class(count.matrix)
```

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

```
class(count.matrix[1,])
```

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

Matrices are a very important object class for mathematical and statistical applications in R, so it is certainly worth exploring more complex matrix operations if you will be doing any more complex statistical analysis in R.

Dataframes

Data frames are very efficient ways of storing tabular data in R. Like matrices, data frames have dimensionality and are organized into rows and columns, however data frames can store vectors of different object classes. Let's convert our matrix to a data frame in R.

```
count.df <- as.data.frame(count.matrix)
count.df
```

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

We can also directly create data frames in R. This is useful if we need to add information about our subjects like sex, race, age range, smoking status, etc. Let's create one of these data frames for our subjects here.

```
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),
                 gender = c("female", "female", "male", "female", "female", "male", "female","male","ma
                 status = c("case", "case", "control", "control","case","case","case","control","contro

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
##  $ gender    : chr  "female" "female" "male" "female" ...
##  $ status    : chr  "case" "case" "control" "control" ...
```

Note that the default behavior of data.frame() in R version < 4.0 is to convert character strings to factors. If you want to prevent this behavior, you can set the StringsAsFactors argument as FALSE. In R versions > 4.0, the default behavior is StringsAsFactors==TRUE.

```
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),
                 gender = c("female", "female", "male", "female", "female", "male", "female","male","ma
                 status = c("case", "case", "control", "control","case","case","case","control","contro
                 stringsAsFactors=FALSE)

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
##  $ gender    : chr  "female" "female" "male" "female" ...
##  $ status    : chr  "case" "case" "control" "control" ...
```

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.

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

```
##   subject_id age gender 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 gender variable/column
df[, c("gender")]
```

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

```r
# # get gender and status
df[, c("gender", "status")]
```

```
##    gender  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
```

```r
# get the gender variable with $
df$gender
```

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

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

Relational (e.g. ==) and logical operators (e.g. !) can be used to interrogate specific variables in a data frame. The resulting logical can also be used to subset the data frame.

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

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

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

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

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

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

```
##    subject_id age gender  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
```

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

```
##    subject_id age gender  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
```

If/Else Statements

If/else statements in R allow us to create decision making programs based on a test expression. In a real world example, let's say you are asked to bring a cake or a pie to your friend's house for dessert. If you decide to bring a cake then you need to get flour and eggs from the grocery store. The only remaining option is that you decide to make a pie and you will need to get crust and apples from the grocery store. We can model this decision making with the following code:

```r
decision <- "cake" # "pie"

if (decision == "cake"){
  print("You should buy flour and eggs at the grocery store")
} else{
  print("You should buy crust and apples at the grocery store.")
}
```

```
## [1] "You should buy flour and eggs at the grocery store"
```

Of course, cake will remain, in my opinion, the best option for dessert. Except for ice cream. That's the best dessert.

Functions

Beyond the functions implemented in base R and packages that you install, R allows you to create user defined functions, which can perform any functionality that you can define.

Defining your own functions can be useful when you want to perform a specific set of tasks repeatedly on some input(s) and return a defined output. Furthermore, once defined functions become part of your global environment and are therefore preserved for future use they minimize the need for repetitive code.

Functions are created using `function()` with the assignment operator `<-`. The arguments you use in the `function()` command define the variables that those arguments will be assigned to when you call the function. The last line of the function defines what output is returned.

Let's define a basic function as an example.

```r
# define the function that takes a single argument and returns a single argument
myfun <- function(x){
  y <- x + 1
  return(y)
}

# call the function
myfun(x = 10)
```

```
## [1] 11
```

```r
# assign the output to a new variable
var1 <- myfun(x = 10)
var1
```

```
## [1] 11
```

Functions can have as many arguments as you specify. The names of these arguments are only assigned as variables within the function, however it is good practice to avoid using arguments with the same name as variables already existing in your global environment.

For example, if I already have a variable named x in my environment, I should avoid using `x` to define the name of the arguments to my function.

```r
myfun2 <- function(num1, num2){
  num3 <- num1 + num2
  return(num3)
}

# call the function
myfun2(num1 = 10, num2 = 11)
```

```
## [1] 21
```

Loops

Loops are used to iterate over a piece of code multiple times, and can therefore be used to achieve specific tasks. The most often used type of loop in R is the `for()` loop, which will evaluate the contents of the loop for all of the values provided to the `for()` function.

For example:

```r
x <- c(1,2,3,4,5,6,7,8,9)

# define the loop, using [i] to define the elements of x used in each iteration
for(i in 1:length(x)){
  print(x[i] * 10)
}
```

```
## [1] 10
## [1] 20
## [1] 30
## [1] 40
## [1] 50
## [1] 60
## [1] 70
## [1] 80
## [1] 90
```

We may wish to save the output of each iteration of the loop to a new variable, which can be achieved using the following approach:

```
x <- c(1,2,3,4,5,6,7,8,9)

# create variable to save results to
y <- c()

# define and run the loop
for(i in 1:length(x)){
  y[i] <- x[i] * 10
}

# print the new vector to console
y
```

```
## [1] 10 20 30 40 50 60 70 80 90
```

While loops can often be useful to evaluate whether a statement or set of statements is true and then execute commands so long as that statement(s) is true.

```
x <- 3
while (x < 6){
  print(x)
  x <- x + 1
}
```

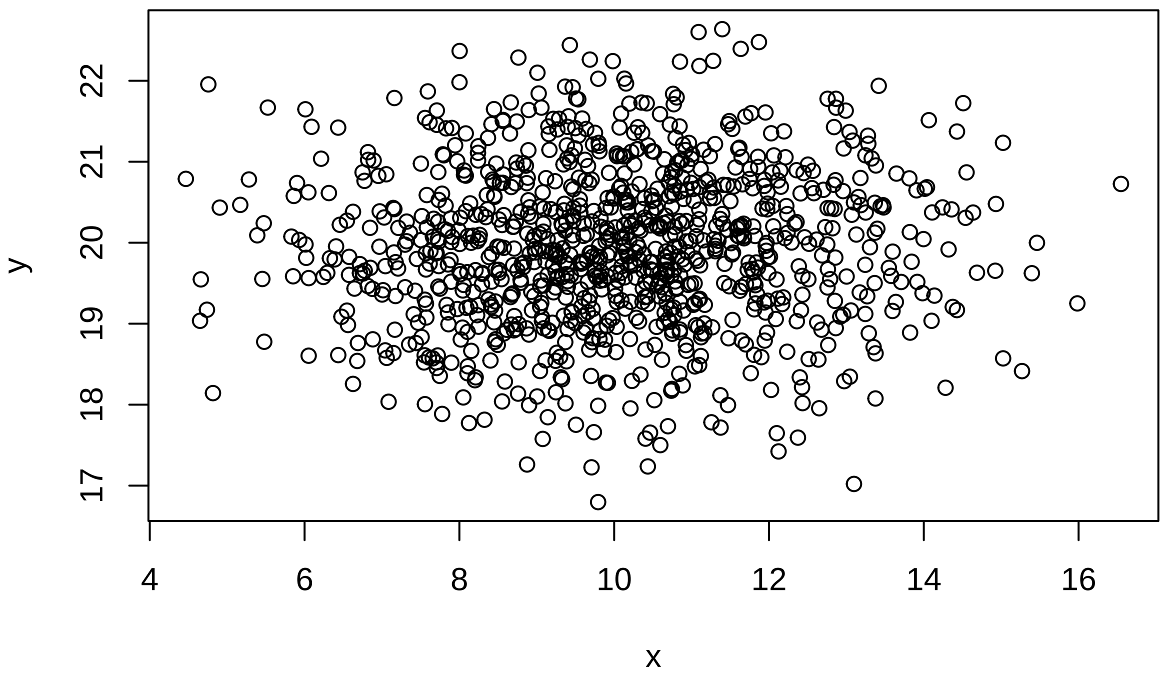
```
## [1] 3
## [1] 4
## [1] 5
```

Basic Data visualization

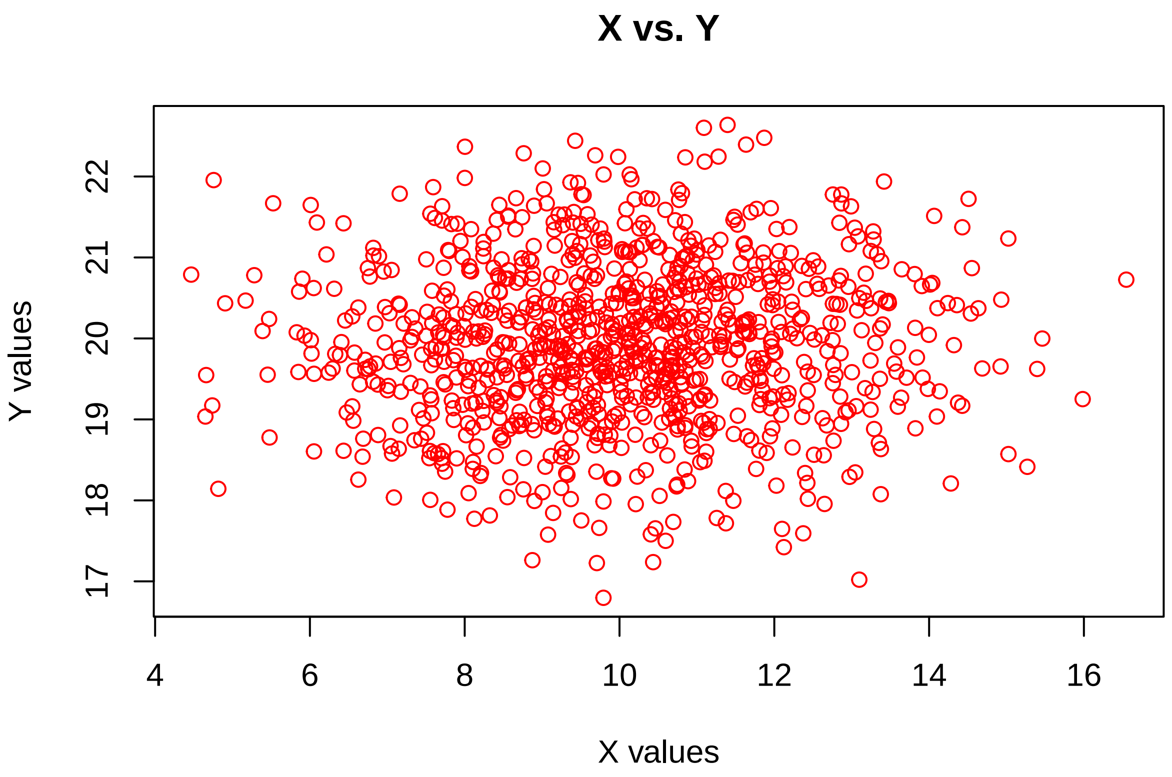
R is a very powerful tool for visualization and provides a large amount of user control over the plotting attributes. Basic visualization in R is achieved using the plot() function.

```
# generate a set of random numbers to plot
x <- rnorm(1000, mean = 10, sd = 2)
y <- rnorm(1000, mean = 20, sd = 1)

# plot x against y to produce a scatter plot
plot(x, y)
```

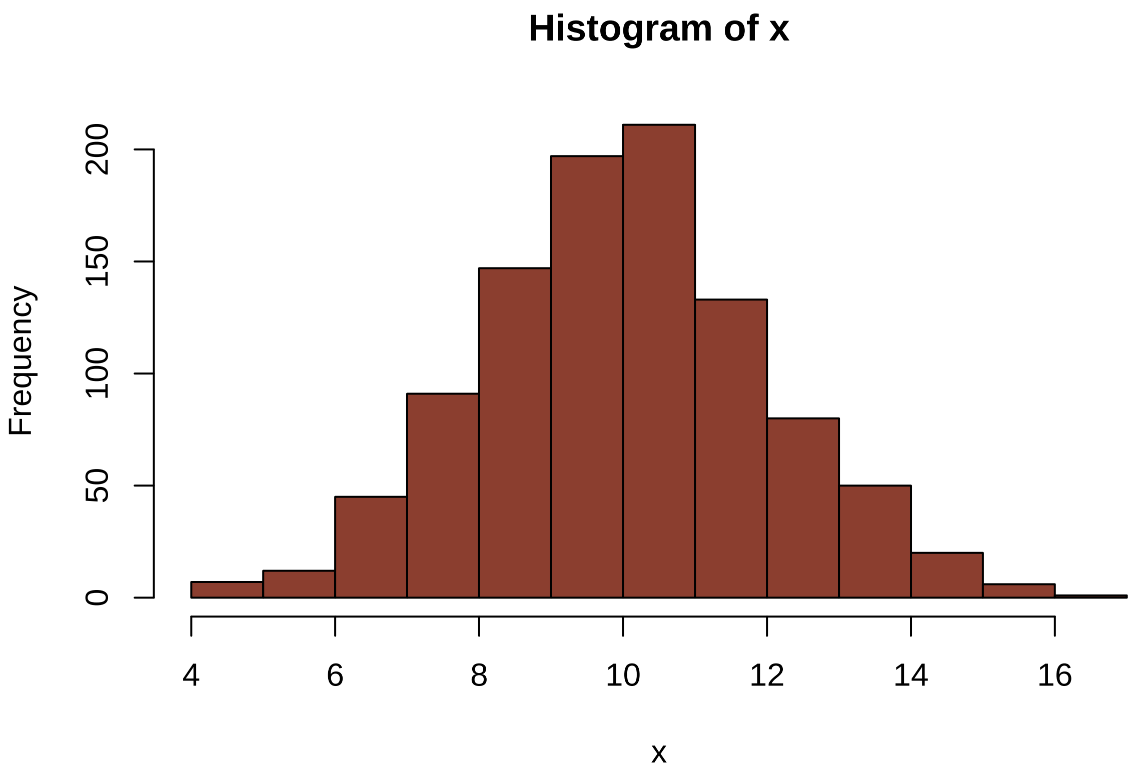


```r
# add labels, title, and color
plot(x, y,
     main = "X vs. Y",
     xlab = "X values",
     ylab = "Y values",
     col = "red")
```

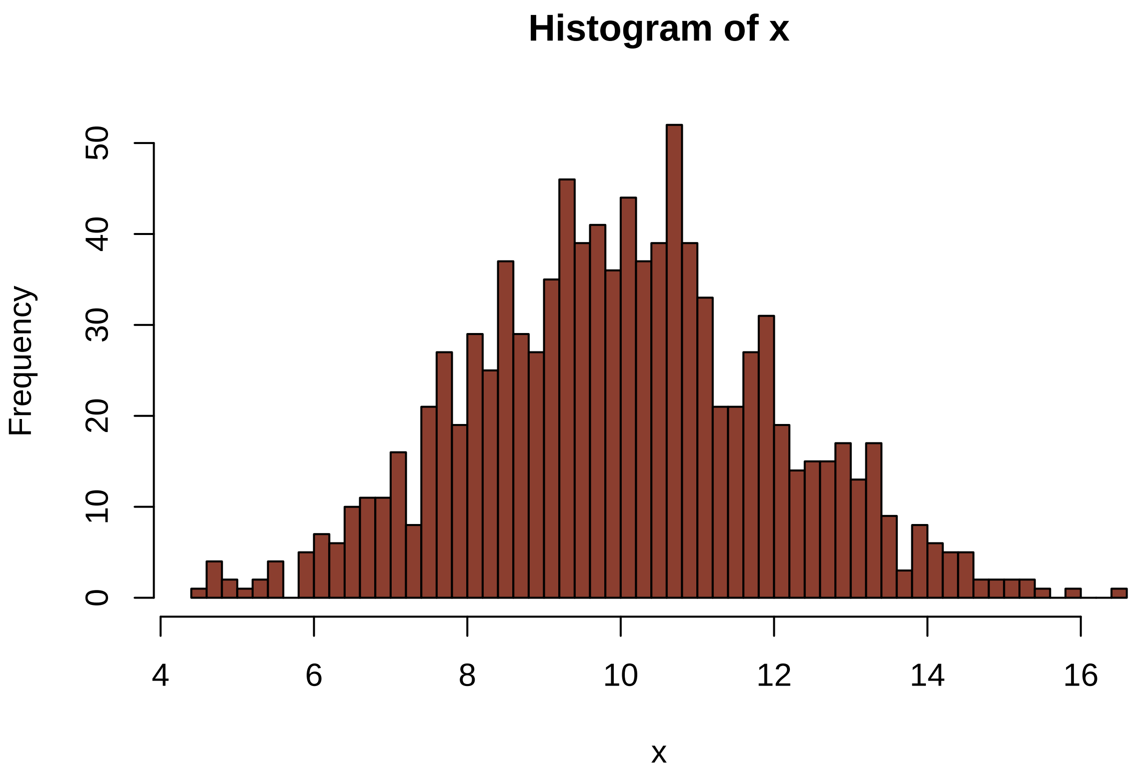


R can also be easily used to generate histograms:

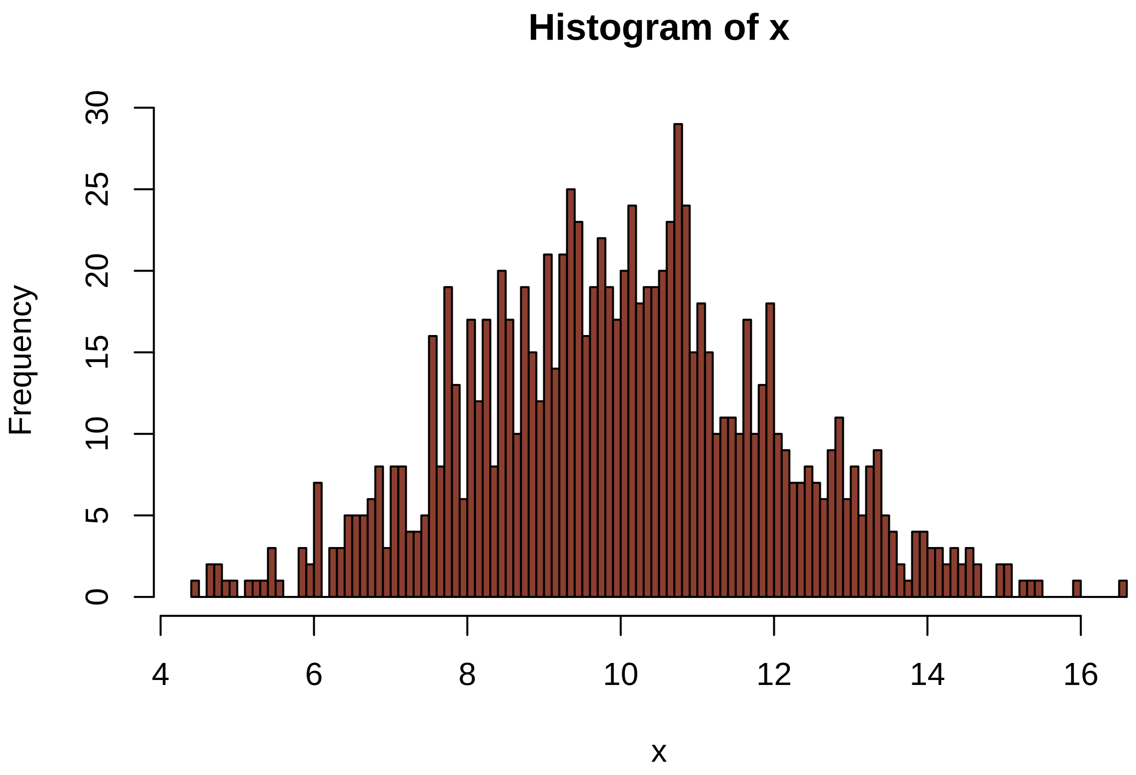
```r
# generate a simple histogram for x
hist(x, col = "coral4")
```



```r
# the breaks argument can be used to change how the intervals are broken up
hist(x, col = "coral4", breaks=10)
hist(x, col = "coral4", breaks=50)
```



```r
hist(x, col = "coral4", breaks=100)
```



There are a large number of packages designed specifically for visualization and are very useful in bioinformatic analyses. We won't cover these here since they are covered extensively elsewhere, but some useful visualization packages to be aware of include:

- ggplot2
- ggpubr
- ploty

Importantly, visualization implemented in these packages form the basis for some bioinformatics specific data visualization packages that we will explore later in the workshop.

Exporting Tabular Data

We may want to save our count matrix and metadata from the Matrices section to files we can later read back into R.

The major functions in base R that exist for writing tabular data to file are write.table() and write.csv(). Similarly to the read functions, write.table() provides a more generalized solution to writing data that requires you to specify the separator value.

In both functions, the first argument specifies the object in your global environment that you wish to write to file. The second argument defines the absolute or relative path to the location you wish to save this file.

```r
# set working directory
setwd("~/Downloads")

# write to tab delimited file using write.table
write.table(count.df, file = "count_df.txt", sep = "\t")
```

In contrast, write.csv() does not require you to set the delimiter value, and by default writes data to comma separated value files (.csv).

```r
# set working directory
setwd("~/Downloads")

# write to csv file
write.csv(df, file = "metadata_df.csv")
```

Importing Tabular Data

Tabular data are often stored as text files where the individual fields containing data points are separated by punctuation points. Three functions exist in base R to facilitate reading in tabular data stored as text files.

read.table() - general function for reading in tabular data with various delimiters. read.csv() - used to read in comma separated values files, where commas are the delimiter. read.delim() - used to read in files in which the delimiters are tabs.

Use read.table() to read in the count_df.txt. Since read.table() is a general function for loading in tabular data, we need to specify the correct separator/delimiter value using the sep argument. Tab delimited data is specified using \t in the sep argument.

```r
# set working directory
setwd("~/Downloads")
# check where we are
getwd()
```

```
## [1] "/Users/ashleekorsberg/Downloads"
```

```r
# using read.table
count.df <- read.table(file = "count_df.txt",
                       sep = "\t", header = TRUE, stringsAsFactors = FALSE)

### Note 1: header accepts logical value indicating if the first row are column names (default FALSE)
### Note 2: we use stringsAsFactors

# check class, dimensions and structure
class(count.df); dim(count.df); str(count.df)
```

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

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

```
## 'data.frame':    10 obs. of  10 variables:
##  $ subject_1 : int  9 1 6 2 9 8 7 10 9 8
##  $ subject_2 : int  6 4 5 2 9 9 5 7 9 8
##  $ subject_3 : int  7 4 1 6 4 2 3 8 3 7
##  $ subject_4 : int  4 4 1 4 1 4 5 8 10 7
##  $ subject_5 : int  3 2 8 4 8 10 0 7 6 6
##  $ subject_6 : int  6 8 4 4 10 3 5 2 6 5
##  $ subject_7 : int  1 10 1 3 2 8 2 4 0 2
##  $ subject_8 : int  3 7 10 4 10 8 0 5 2 3
##  $ subject_9 : int  1 6 10 0 1 9 9 2 8 7
##  $ subject_10: int  9 5 8 3 2 9 9 10 3 8
```

Now use read.delim(). An important difference between read.delim() and read.table() are the default setting for the sep and header arguments. By default in read.delim(), sep is set to and the header argument is set to TRUE, so we do not need to explicitly call those arguments.

```r
# set working directory
setwd("~/Downloads")

# using read.delim
count.df <- read.delim(file = "count_df.txt", stringsAsFactors=FALSE)

# check class, dimensions and structure
class(count.df); dim(count.df); str(count.df)
```

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

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

```
## 'data.frame':    10 obs. of  10 variables:
##  $ subject_1 : int  9 1 6 2 9 8 7 10 9 8
##  $ subject_2 : int  6 4 5 2 9 9 5 7 9 8
##  $ subject_3 : int  7 4 1 6 4 2 3 8 3 7
##  $ subject_4 : int  4 4 1 4 1 4 5 8 10 7
##  $ subject_5 : int  3 2 8 4 8 10 0 7 6 6
##  $ subject_6 : int  6 8 4 4 10 3 5 2 6 5
##  $ subject_7 : int  1 10 1 3 2 8 2 4 0 2
##  $ subject_8 : int  3 7 10 4 10 8 0 5 2 3
##  $ subject_9 : int  1 6 10 0 1 9 9 2 8 7
##  $ subject_10: int  9 5 8 3 2 9 9 10 3 8
```

read.csv() is used in exactly the same way read.delim(), however the file specified must contain data separated by commas and have the extension .csv.

```
# set working directory
setwd("~/Downloads")

# use read.csv()
meta_data <- read.csv(file = "metadata_df.csv",row.names=1)
meta_data
```

```
##    subject_id age gender  status smoking_status
## 1   subject_1  45 female    case         former
## 2   subject_2  83 female    case           none
## 3   subject_3  38   male control          heavy
## 4   subject_4  23 female control           none
## 5   subject_5  65 female    case           none
## 6   subject_6  40   male    case          heavy
## 7   subject_7  32 female    case          heavy
## 8   subject_8  89   male control          heavy
## 9   subject_9  77   male control         former
## 10 subject_10  53   male    case         former
```

Let's read in some publicly available data from a real RNA-seq run. The paper and access to the data can be found at this link. This data was generated from synovial fluid collected from inflamed joints of patients with rheumatoid arthritis before and after treatment with a TNF-a blocker, a common treatment for this disease.

```
# set working directory
setwd("~/Downloads")

# read in data
example.df <- read.table(file = "GSE198520_Raw_gene_count_matrix.txt",
                     sep = "\t", header = TRUE,
                     stringsAsFactors = FALSE)

head(example.df)
```

```
##   GeneSymbol r_006_post_bx r_006_pre_bx r_005_post_bx r_005_pre_bx
## 1    AADACL2             7            2             2            6
## 2       AATF             0            0             0            0
## 3     ABCB11            34           43            11           34
## 4      ABCC1             3            1             0            3
## 5      ABCC6             0            0             0            2
## 6      ABCF1             0            0             0            0
##   r_004_post_bx r_004_pre_bx r_003_post_bx r_003_pre_bx r_002_post_bx
## 1             0            2             7            1             2
## 2             0            0             0            0             0
## 3            15           27            27           41            24
## 4             0            1             2            1             2
## 5             0            0             0            0             2
## 6             0            0             0            0             0
##   r_002_pre_bx r_001_post_bx r_001_pre_bx r_012_post_bx r_012_pre_bx
## 1            2             0            0             5            7
## 2            0             0            1             0            0
## 3            9            16           21            32           28
## 4            0             3            1             0            0
## 5            0             1            2             0            0
## 6            0             0            0             0            0
##   r_011_post_bx r_011_pre_bx r_010_post_bx r_010_pre_bx r_009_post_bx
## 1             1            2             2            1             4
## 2             0            0             0            0             0
## 3            28           27            19           31            24
## 4             0            0             0            0             2
## 5             1            3             0            0             0
## 6             0            0             1            0             0
##   r_009_pre_bx r_008_post_bx r_008_pre_bx r_007_post_bx r_007_pre_bx
## 1            4             1            4             1            0
## 2            0             0            0             0            0
## 3           24            22           16            18           31
## 4            4             0            0             0            0
## 5            1             0            0             0            0
## 6            0             0            0             0            1
##   r_018_post_bx r_018_pre_bx r_017_post_bx r_017_pre_bx r_016_post_bx
## 1             2            3             0            1             1
## 2             0            2             0            0             0
## 3            34           21            34           14            40
## 4             0            4             0            0             0
## 5             0            1             1            0             3
## 6             0            0             0            0             0
##   r_016_pre_bx r_015_post_bx r_015_pre_bx r_014_post_bx r_014_pre_bx
## 1            5             0            0             2            4
## 2            0             0            0             0            0
## 3           24            20           27            16           20
## 4            2             1            2             4            6
## 5            1             3            1             0            4
## 6            0             0            0             0            0
##   r_013_post_bx r_013_pre_bx nr_005_post_bx nr_005_pre_bx nr_004_post_bx
## 1             7            4              1             2              1
## 2             0            0              1             0              0
## 3            26           18             35            16             22
## 4             0            3              2             1              1
## 5             0            0              2             2              0
## 6             0            0              0             0              0
##   nr_004_pre_bx nr_003_post_bx nr_003_pre_bx nr_002_post_bx nr_002_pre_bx
## 1             2              4             7              5             2
## 2             0              0             1              0             1
## 3            20             27            46             59            33
## 4             1              2             3              3             0
## 5             0              0             0              1             2
## 6             0              0             0              0             0
##   nr_001_post_bx nr_001_pre_bx r_019_post_bx r_019_pre_bx nr_012_post_bx
## 1              1             0             0            2              5
## 2              0             0             0            0              0
## 3             18            29            27           19             66
## 4              1             1             1           10              3
## 5              0             1             1            1              0
## 6              0             0             0            0              0
##   nr_012_pre_bx nr_011_post_bx nr_011_pre_bx nr_010_post_bx nr_010_pre_bx
## 1             1              1             6              5             1
## 2             0              0             0              0             0
## 3            33             24            34             15            45
## 4             3              3             1              1             0
## 5             0              0             0              1             0
## 6             0              0             0              0             0
##   nr_009_post_bx nr_009_pre_bx nr_007_post_bx nr_007_pre_bx nr_006_post_bx
## 1              1             1              1             1              2
## 2              0             0              0             0              0
## 3             27            18             11            14             59
## 4              7             4              0             1              0
## 5              1             1              1             1              2
## 6              0             0              0             0              0
##   nr_006_pre_bx mr_003_post_bx mr_003_pre_bx mr_002_post_bx mr_002_pre_bx
## 1             2              1             0              3             2
## 2             0              0             0              0             0
## 3            28             31            22             41            34
## 4             1              0             2              1             1
## 5             1              0             1              1             2
## 6             0              0             0              0             0
##   mr_001_post_bx mr_001_pre_bx nr_015_post_bx nr_015_pre_bx nr_014_post_bx
## 1              4             0              3             3              3
## 2              0             0              0             0              0
## 3             41            10             43            40             37
## 4              1             0              0             2              0
## 5              0             0              0             1              0
## 6              0             0              0             0              0
##   nr_014_pre_bx nr_013_post_bx nr_013_pre_bx mr_008_post_bx mr_008_pre_bx
## 1             1              3             2              0             2
## 2             0              0             0              0             0
## 3            32             40            36             11            21
## 4             1              1             1              0             5
## 5             0              0             0              1             0
## 6             0              0             0              0             0
##   mr_007_post_bx mr_007_pre_bx mr_006_post_bx mr_006_pre_bx mr_005_post_bx
## 1              0             4              1             4              0
## 2              0             0              0             0              0
## 3             33            25             29            55             59
## 4              0             1              5             0              3
## 5              0             0              1             2              1
## 6              0             0              0             0              0
##   mr_005_pre_bx mr_004_post_bx mr_004_pre_bx mr_013_post_bx mr_013_pre_bx
## 1             1              2             3              3             5
## 2             0              0             0              0             0
## 3            32             26            47             21            73
## 4             2              1             3              0             1
## 5             1              0             1              0             2
## 6             0              0             0              0             0
##   mr_012_post_bx mr_012_pre_bx mr_011_post_bx mr_011_pre_bx mr_010_post_bx
## 1              1             2              2             2              0
## 2              0             0              0             0              0
## 3             33            22             59            32             10
## 4              0             2              0             0              6
## 5              0             0              0             0              0
## 6              0             0              1             0              0
##   mr_010_pre_bx mr_009_post_bx mr_009_pre_bx
## 1             1              0             2
## 2             0              1             0
## 3            25             29            40
## 4             1              1             1
## 5             2              0             0
## 6             0              0             0
```

When datasets get very large, these base R functions can be quite slow. Although we do not have time to cover them here, the readr and data.table packages contain functions that introduce extremely fast ways of reading data into an R environment.

Save R Objects to A File

It is possible to save R objects to a file that maintains all of their attributes so that they can be easily loaded back into a new R session. This functionality is achieved using the `save()` and `load()` functions.

`save()` accepts the names of the R objects you wish to write to a file (which will have the extension .rdata) as the first arguments, and the file path where you wish to write this file under the file argument. For example:

```
# create some R objects
x <- c(1.63, 2.25, 3.83, 4.99)
y <- c(TRUE, FALSE, TRUE, TRUE)
z <- c("a", "b", "c", "d")

# check what directory we're in
getwd()
```

```
## [1] "/Users/ashleekorsberg/Ash's Documents/QBS PhD Program_2024/bootcamp_2025/intro_to_R"
```

```
# save all 3 objects to one file
save(x, y, z, file = "my_r_objects.rdata")
```

These objects can then be loaded back into your global environment using the `load()` function with the absolute or relative file path. The objects will appear in your new environment with exactly the same names as in your previous environment.

```
# load file
load(file = "my_r_objects.rdata")
```

Single R objects can be saved and restored in a similar way using the `saveRDS()` and `readRDS()` functions. Files saved using RDS must take on the .rds extension.

```r
# save a single object to a specific file path
saveRDS(x, file = "my_r_object.rds")

# use the assignment operator to assign object to any variable you want
x <- readRDS(file = "my_r_object.rds")

# I changed my mind, I want the object to be assigned to variable `a` in my new env
a <- readRDS(file = "my_r_object.rds")
```