Introduction to R

Using basic functions in R

Inspecting function documentation

There are many functions in base R that you can use (in a later tutorial, we will discuss how to create your own functions!). In order to view documentation for a function in R, you can type ?function_name or search for the function name in the help tab. As practice, let's explore the matrix() and mean() functions in R by typing ?matrix and ?mean into the console. Uncomment (by deleting the #s) the code below and run it in the console to inspect the documentation.

```
#?matrix
#?mean
```

Under usage for the matrix() function, you should see the following:
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
The matrix function has 5 distinct parameters. All of them have default values. For example, if you don't put any data into the function, the resulting matrix will be made up of NA values (this is how missing data is generally coded in R). Additionally, the function will have 1 row and 1 column by default and data will be filled by columns because byrow=FALSE by default. The matrix will not have any row or column names because dimnames=NULL by default.

Because all parameters of the matrix() function have defaults, we could call matrix() with no inputs and we would get a 1x1 matrix with NA values and no dimension names. We can also pick and choose whatever parameters we do want to fill in and ignore anything that we want to leave as defaults.

```
# Call matrix() with no inputs
matrix()
```

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

```
# Make a 2x3 matrix of NAs
matrix(nrow=2, ncol=3)
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

In contrast, the mean() function has some required parameters. When you type ?mean into the console, you should see the following:
mean(x, trim = 0, na.rm = FALSE, ...)
The parameter x has no default value; you must specify the values that you want to take the mean of. The other paramters are given defaults: trim=0 and na.rm=FALSE. If you read the descriptions for these parameters, you will see that trim allows you to calculate a trimmed mean (i.e., eliminate some proportion of extreme values before calculating the mean) and na.rm allows you to remove missing (NA) values before calculating the mean. By default, R will not do any trimming and NAs will not be removed. This can cause issues (see below):

```
# Create some data and save it as data1 and data2
data1 <- c(1,2,3,4,7,NA)
data2 <- c(1,2,3,4,7)

# Calculate the mean of data1 and data2
# Note that data1 has a mean of NA because there was an NA value that was not removed
mean(data1)
```

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

```
mean(data2)
```

```
## [1] 3.4
```

```
# Now explicitly set na.rm=TRUE and recalculate the mean of data1. Now we get the same as data2
mean(data1, na.rm=TRUE)
```

```
## [1] 3.4
```

Calling functions in R

When using functions in R, you can make it clear which inputs refer to which parameters by either a) using the order of the parameters specified in the usage or b) naming them explicitly.

```
# Note that these two calls to the mean() function will return the same output
mean(data1, .2, TRUE)
```

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

```
mean(x=data1, trim=.2, na.rm=TRUE)
```

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

```
# However, the following code will give an error because "TRUE" is not a valid input to trim,
# Which is the second parameter listed in the usage
# Mean(data1,TRUE)

# To specify na.rm=TRUE but leave the trim=0 default as is, we simply do the following:
mean(data1, na.rm=TRUE)
```

```
## [1] 3.4
```

```
# (Note that, since data1 still matches to x, which is the first parameter in usage,
#  we do not need to specify x=data1)
```

Data types and structures

This section covers:
-Four basic data types in R: characters, numerics, integers, and logicals
-Four basic ways to store data in R: vectors, matrices, data frames, and lists

You can learn about these data types below

Numerics and integers

```
# To store data in some variable name, use either = or <-
# To save the number 5 as "number":
number <- 5
number = 5 #does the same thing

# Print number:
print(number)
```

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

```r
# Find out the class of number:
class(number)
```

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

```r
# Change number to an integer and re-save it as number2:
number2 <- as.integer(number)

# Another way to save a value as an integer:
number3 <- 5L
class(number3)
```

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

```r
# Inspect the class of number2 to see that it is an integer:
class(number2)
```

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

Characters

```r
# Save a character string in a variable named message
message <- "welcome"

# Print message:
print(message)
```

```
## [1] "welcome"
```

```r
# Inspect the class of message:
class(message)
```

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

Logicals

Logical values are either TRUE or FALSE
In R, TRUE=1 and FALSE=0

```r
# Save the logical TRUE as a variable called outcome:
outcome <- TRUE

# Print outcome
print(outcome)
```

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

```r
# Inspect class of outcome
class(outcome)
```

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

```r
# Note that, weirdly, outcome+outcome=2
outcome+outcome
```

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

In R, you can test a statement to see if it is TRUE or FALSE. Note that R allows you to make comparisons accross variable types: integers may be compared to numerics and logicals may be compared to integers/numerics. For characters, comparatives are assessed using alphabetical order (letters earlier in the

alphabet are “smaller”):
1) == means “is equal to”
2) != means “is not equal to”
3) > means “greater than”; >= means “greater than or equal to”
4) < means “less than; <= means”less than or equal to"
5) & means “and”; | means “or”

```
# Is 5 equal to 3?
5==3
```

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

```
# Is 5 not equal to 3?
5!=3
```

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

```
# Is 5 less than 3?
5<3
```

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

```
# Is 5 greater than 3?
5>3
```

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

```
# Is 5 greater than 3 AND less than 7?
5>3 & 5<7
```

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

```
# Is 5 less than 3 OR less than 7?
5<3 | 5<7
```

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

```
# Is 5 greater than 5?
5>5
```

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

```
# Is 5 greater than or equal to 5?
5>=5
```

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

```
# Is 5 equal to 5?
5==5
```

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

```
# Is "hello" equal to "hello"?
"hello" == "hello"
```

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

```
# Is "hello" equal to "goodbye"?
"hello" == "goodbye"
```

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

```
# Is "hello" greater than "goodbye"? (in other words is "hello" after "goodbye" alphabetically?)
"hello">"goodbye"
```

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

```
# Is TRUE == 1?
TRUE==1
```

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

```
# Is FALSE==0?
FALSE==0
```

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

Vectors

```
# The easiest way to create a vector is by using the c() function
vec1 <- c(2,3,4,5)
print(vec1)
```

```
## [1] 2 3 4 5
```

```
# Note that, if you include multiple data types in a vector, R will change all values to the same type
vec2 <- c(7,3)
vec2
```

```
## [1] 7 3
```

```
class(vec2)
```

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

```
vec3 <- c(7,3,"hello")
vec3
```

```
## [1] "7"     "3"     "hello"
```

```
class(vec3)
```

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

```
# For values in a row, we can also use a colon:
vec3 <- 2:5
print(vec3)
```

```
## [1] 2 3 4 5
```

```
# We can use the c() function to combine pre-saved vectors:
vec4 <- c(vec1,vec3)
print(vec4)
```

```
## [1] 2 3 4 5 2 3 4 5
```

```
# Use the length function to find the length of a vector
length(vec4)
```

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

Here are some other useful shortcuts for creating vectors in R:

```
# Use the rep() function to repeat values. Inspect the following to see how it works!
rep(x=0, times=5) #create a vector of 5 zeros
```

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

```r
rep(x=c(1,2,3), times=5) #create a vector with five repeats of 1,2,3
```

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

```r
rep(c(1,2,3), each=2, times=5) #repeat each value in 1,2,3 twice, then repeat that 5 times
```

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

```r
# Use the seq function to create a sequence of values
seq(from=1, to=5, by=.5) #create a vector with values from 1-5, incementing by .5
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
seq(from=1, to=5, length.out=17) #create a vector with 17 equally spaced values going from 1 to 5
```

```
##  [1] 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25
## [15] 4.50 4.75 5.00
```

Matrices

As shown above, an m by n matrix can be created in R using the matrix() function
Here are some examples

```r
A <- matrix(2, nrow=3, ncol=3)
print(A)
```

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

```r
B <- matrix(c(1,2,5,3,4,0,2,1,5), nrow=3, ncol=3, byrow=TRUE)
print(B)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    5
## [2,]    3    4    0
## [3,]    2    1    5
```

```r
# Print the dimensions of a matrix (number of rows followed by number of columns)
dim(A)
```

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

```r
# Matrix multiplication
A %*% B
```

```
##      [,1] [,2] [,3]
## [1,]   12   14   20
## [2,]   12   14   20
## [3,]   12   14   20
```

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

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

```r
# Element-wise addition
A+B
```

```
##      [,1] [,2] [,3]
## [1,]    3    4    7
## [2,]    5    6    2
## [3,]    4    3    7
```

```r
# Transpose of a matrix
t(B)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    2
## [2,]    2    4    1
## [3,]    5    0    5
```

```r
# Inverse of a matrix
solve(B)
```

```
##            [,1]        [,2]        [,3]
## [1,] -0.5714286  0.14285714  0.57142857
## [2,]  0.4285714  0.14285714 -0.42857143
## [3,]  0.1428571 -0.08571429  0.05714286
```

Data Frames

Data frames are generally used to store tabular data and are composed of same-length vectors; these vectors can be of differing data types. In general, when you read a .csv data file into R, it will be saved as a data frame.
We can create a data frame in R as follows:

```r
# Create a fake dataset called example_data
example_data <- data.frame(ID_Num = c(1:10),
                           Age = rep(24:28, each=2),
                           State = c(rep("New Jersey", 5), rep("New York", 5)))

# Change row names of the data frame (some made up names)
rownames(example_data) <- c("Sarah", "Mike", "Drew", "Eric", "Maria",
                            "Lindsey", "Mark", "Jenny", "Sophie", "Paul")

# Print the data frame
example_data
```

```
##         ID_Num Age      State
## Sarah        1  24 New Jersey
## Mike         2  24 New Jersey
## Drew         3  25 New Jersey
## Eric         4  25 New Jersey
## Maria        5  26 New Jersey
## Lindsey      6  26   New York
## Mark         7  27   New York
## Jenny        8  27   New York
## Sophie       9  28   New York
## Paul        10  28   New York
```

The following R code outlines a few ways to inspect data in a data frame.

```r
# Get dimensions (same as matrices)
dim(example_data)
```

```
## [1] 10  3
```

```r
# Get number of columns
ncol(example_data)
```

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

```r
# Get number of rows
nrow(example_data)
```

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

```r
# Get summaries of the columns
summary(example_data)
```

```
##      ID_Num           Age            State
##  Min.   : 1.00   Min.   :24   New Jersey:5
##  1st Qu.: 3.25   1st Qu.:25   New York  :5
##  Median : 5.50   Median :26
##  Mean   : 5.50   Mean   :26
##  3rd Qu.: 7.75   3rd Qu.:27
##  Max.   :10.00   Max.   :28
```

```r
# Access a single column of the data frame using $
example_data$Age
```

```
##  [1] 24 24 25 25 26 26 27 27 28 28
```

```r
# Inspect row names
rownames(example_data)
```

```
##  [1] "Sarah"   "Mike"    "Drew"    "Eric"    "Maria"   "Lindsey" "Mark"
##  [8] "Jenny"   "Sophie"  "Paul"
```

```r
# Inspect column names
colnames(example_data)
```

```
## [1] "ID_Num" "Age"    "State"
```

Lists

Lists enable multiple data types or data sets to be stored in a simple object. For example, a list could have a data frame as its first element, a vector as its second element, and a character string as its third element.

```r
# Save vector vec1, matrix A, and vector vec2 in a list called example list
example_list <- list(vec1, A, vec2)

# Print example_list
example_list
```

```
## [[1]]
## [1] 2 3 4 5
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    2    2    2
```

```
## [2,]    2    2    2
## [3,]    2    2    2
##
## [[3]]
## [1] 7 3
```

Indexing

In R, indices start at 1, not 0 as in other languages. For example, the index of the 3rd element in a vector is 3.

Using indices to extract elements in a vector

We can use indices enclosed in square brackets in order to extract data from a vector as follows:

```
# This R chunk uses vector vec4 from above
# Re-print vec4
vec4
```

```
## [1] 2 3 4 5 2 3 4 5
```

```
# Extract the 3rd element in vec4
vec4[3]
```

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

```
# Extract the 3rd through 5th elements in vec4
vec4[3:5]
```

```
## [1] 4 5 2
```

```
# Extract the 1st, 3rd, and 7th elements in vec4
vec4[c(1,3,7)]
```

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

```
# Remove the 2nd element from vec4
vec4[-2]
```

```
## [1] 2 4 5 2 3 4 5
```

```
# Remove the 2nd, 4th, and 5th elements from vec4
vec4[-c(2,4,5)]
```

```
## [1] 2 4 3 4 5
```

We can also use the following functions to either a) get a logical vector indicating which values in the vector meet some criterion or b) get indices of values in a vector that meet some criterion.

```
# Logical vector of the same length as vec4
# TRUE wherever elements equal 2; FALSE elsewhere
vec4==2
```

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

```
# Get indices of all values in vec4 that are equal to 2
which(vec4==2)
```

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

```
# Get index of the maximum value in vec4
# If the maximum occurs more than once, this returns the first location by default
which.max(vec4)
```

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

```
# Return logical vector indicating which elements of vec4 are either equal to 2 or 4
vec4 %in% c(2,4)
```

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

```
# Another way to do the same. Note that | means "or" and & means "and"
vec4==2 | vec4==4
```

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

By enclosing the output from the above functions in square brackets, we can extract elements meeting particular criteria from a vector. For example:

```
# Extract elements of vec4 that are equal to 2
vec4[vec4==2]
```

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

```
# Extract elements of vec4 that are equal to 2 or 4
vec4[vec4==2|vec4==4]
```

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

```
#or:
vec4[vec4 %in% c(2,4)]
```

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

Using indices to extract elements in a matrix

In a similar way, we can use square brackets to extract elements from a matrix. However, we now need both row and column indices to specify a particular element. See examples below:

```
# Re-print matrix B for reference
B
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    5
## [2,]    3    4    0
## [3,]    2    1    5
```

```
# Extract the element of matrix B that is located in row 3, column 2
B[3,2]
```

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

```
# Extract all elements of B that are greater than 2 and less than 5
B[B>2 & B<5]
```

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

We can also use indexing to extract particular rows or columns of a matrix. Note that, in general, we extract elements from a matrix by using [row_index,column_index]. If we just want to specify row indices, but not column indicies, we can leave the column index blank; similarly, if we just want to specify column indices, we can leave the row index blank. For example:

```r
# Extract the 2nd row of matrix B
B[2,]
```

```
## [1] 3 4 0
```

```r
# Extract the 1st and 3rd rows of matrix B
B[c(1,3),]
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    5
## [2,]    2    1    5
```

```r
# Extract the 2nd column of matrix B
# Note that, because these values are in the same column, they are returned as a vector, not matrix
B[,2]
```

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

```r
# Extract the values that are in the 1st and 3rd rows and 2nd and 3rd columns of matrix B
B[c(1,3),2:3]
```

```
##      [,1] [,2]
## [1,]    2    5
## [2,]    1    5
```

Extracting elements of a data frame

Extracting values from a data frame works in much the same way as above; however, it is also possible to specify rows and columns of a data frame by name (or by using a dollar sign for columns). See below:

```r
# Print the 3rd row of example_data
example_data[3,]
```

```
##      ID_Num Age      State
## Drew      3  25 New Jersey
```

```r
# Print the 2nd and 3rd row of example_data
example_data[2:3,]
```

```
##      ID_Num Age      State
## Mike      2  24 New Jersey
## Drew      3  25 New Jersey
```

```r
# Print Sarah's Age
example_data["Sarah","Age"]
```

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

```r
### THREE DIFFERENT WAYS TO GET THE 2ND COLUMN (Age)
# Using the column index
example_data[,2]
```

```
##  [1] 24 24 25 25 26 26 27 27 28 28
```

```r
# Using a $
example_data$Age
```

```
##  [1] 24 24 25 25 26 26 27 27 28 28
```

```r
# Using square brackets and the column name
example_data[,"Age"]
```

```
##  [1] 24 24 25 25 26 26 27 27 28 28
```

Some more examples of using logicals to extract specific data from a data frame:

```
# Extract the ages of people who are from New York
example_data$Age[example_data$State=="New York"]
```

```
## [1] 26 27 27 28 28
```

```
# Extract only the rows of example_data where Age is equal to 24
example_data[example_data$Age==24,]
```

```
##       ID_Num Age      State
## Sarah      1  24 New Jersey
## Mike       2  24 New Jersey
```

```
# Extract only the rows of example_data where Age is 24 and State is New York
example_data[example_data$Age==24 & example_data$State=="New York",]
```

```
## [1] ID_Num Age    State
## <0 rows> (or 0-length row.names)
```

```
# Extract the row names (i.e., names) of those who are 24 from New York
rownames(example_data)[example_data$Age==24 & example_data$State=="New York"]
```

```
## character(0)
```

Extracting elements of a list

Extracting values from a list requires two steps: first you'll need to extract the element of the list you are interested in using double square brackets: [[]]. Then, you can use regular square brackets to extract values from each element as described in the above sections. See below:

```
# Re-print example_list
example_list
```

```
## [[1]]
## [1] 2 3 4 5
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    2    2    2
## [3,]    2    2    2
##
## [[3]]
## [1] 7 3
```

```
# Extract the first element in the list (which is a vector)
example_list[[1]]
```

```
## [1] 2 3 4 5
```

```
# Extract the 2nd value in that vector
example_list[[1]][2]
```

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

```
# Extract the value in the 1st row, 3rd column of the second element of the list
example_list[[2]][1,3]
```

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

Factor variables and levels

One other data structure that you'll see is a factor variable with levels. This is often used for categorical data. For example, suppose you collect some data where you ask people to rate their agreement with the statement, "I like coffee." Each person responds on a scale from 1-5 where 1=strongly disagree, 2=disagree, 3=no opinion, 4=agree, and 5=strongly agree. When you collect this data, you might want the numerical responses to be linked with their descriptions; but you don't want R to treat this as a continuous variable (for example, a value of 2.3 is not possible). (note: factor variables can also be used for non-ordered categorical variables). The code below shows an example:

```
# Create some fake data
fake_data <- sample(c("strongly disagree", "disagree", "no opinon", "agree", "strongly agree"),
                    size=100, replace=TRUE, prob=c(.2,.3,.1,.3,.2))

# Make fake_data into a factor variable
fake_data <- as.factor(fake_data)

# Look at the first 3 elements of the fake data
fake_data[1:3]
```

```
## [1] strongly disagree no opinon         no opinon
## Levels: agree disagree no opinon strongly agree strongly disagree
```

```
# Inspect class of fake_data
class(fake_data)
```

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

```
# Inspect the structure of fake_data
str(fake_data)
```

```
##  Factor w/ 5 levels "agree","disagree",..: 5 3 3 2 1 2 4 2 3 1 ...
```

```
# Inspect the levels of fake_data
levels(fake_data)
```

```
## [1] "agree"             "disagree"          "no opinon"
## [4] "strongly agree"    "strongly disagree"
```

Note that we have created a factor variable with 5 levels. Even though each element in the data is a character string, the class is "factor", not "character". This tells R that the variable is categorical.

Reading in data

There are a number of ways to read data into R, but the most common is to use the read.csv function, which can read in a .csv file and convert it to a data frame in your working environment. In order for R to read a file, you must either include the full path to the file or you must set your working directory to the location of the file. If you are planning to knit to PDF, the call to setwd() or the full file name must be included in an R chunk (instead of just running it in the Console). Another option is to use an Rproj file to automatically set your working directory to a particular location every time you open it.

Note that an easy way to get a file path is to right click on the file in your finder window and then click "get info". The info page should include a file path that you can copy and paste into R. Note that file names and path names need to be in quotes.

```r
# Use setwd() to set your working directory to a particular location
setwd("/Users/sophiesommer/Desktop/Grad School/A3SR-Welcome-Package/Assignments")

# Now that I've set my working directory to my "Assignments" folder,
# I can read any .csv file in that location by simply writing the name in quotes:
data <- read.csv("height_sex.csv")
```

Some commonly used data sets are already available to you in base R (and others are available once you download particular packages). For example, I can load the iris dataset by simply using the data() function:

```r
#load iris data set into my working environment
data(iris)

#view first five rows of the iris dataset
iris[1:5,]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
```

Finally, if you have data in other formats, you might need some other functions in R. Probably the most relevant is the read_dta function (in the "haven" package) or read.dta (in the "foreign" package). Both functions read .dta files, which is how Stata files are generally saved.

Downloading new packages and functions

To download new packages in R, you can use the install.packages() function and then use library() or require() to load the package in your workspace. Note that you will only have to use install.packages() once (unless you update R and have to delete packages for some reason); whereas you will need to re-load packages with library() or require() every time you close and re-open R Studio (assuming that you cleared your working environment). You will also must include the library() or require() call within Rmd files if you are planning to knit to PDF (but you can write {r, echo=FALSE, message=FALSE} at the beginning of the chunk to surpress the output if you don't want it to show). Package names must always be in quotes.

```r
# Install the psych package
# Note: I have commented out the code so that it doesnt re-install on my computer
# install.packages("psych")

# Load psych package
library(psych)
#require() does the same thing but will first check if the package is already loaded,
#and only loads it if it is not already there:
require(psych)
```

Using R as a calculator

Top 10 most useful functions to look up