Intro to R Extended

Stacey Borrego

4/11/2018

Purpose

This document is a continuation of Intro-to-R.html and will address more topics in R such as filtering/subsetting your data and saving a file. These concepts require further explanations and understanding than will be provided here. However, I hope this document will demonstrate the power of R and introduce you to essential concepts.

The text file (combined_out.txt) used here and complemenatry documents can be found on the Bioinformatics Support Group GitHub - https://github.com/bioinformaticssg/Introduction-to-R.

To get the files:

- Go to https://github.com/bioinformaticssg/Introduction-to-R
- Click on the green **Clone or download** button in the upper right corner
- Click on **Download ZIP**

Reading Data into R

Just as a precaution, let's load our data again. Since our data is tabular, it will be stored in a dataframe which I will name **my_data**. I have included the full path to the file on my computer, you need to replace everything in quotes with the path on your computer.

Note: I will use the functions head() and tail() to keep looking at the data to make sure the changes I expect to have occurred actually did.

```
my_data <- read.table("/Users/stacey/Data/GitHub/Introduction-to-R/combined_out.txt")
head(my_data)
```

```
##                   V1 V2 V3 V4
## 1 ENSG00000277248.1  0  0  0
## 2 ENSG00000274237.1  0  0  0
## 3 ENSG00000280363.1  0  0  0
## 4 ENSG00000279973.1  0  0  0
## 5 ENSG00000226444.2  0  0  0
## 6 ENSG00000276871.1  0  0  0
```

```
tail(my_data)
```

```
##               V1  V2   V3   V4
## 1458 ERCC-00163   5    7   13
## 1459 ERCC-00164   0    1    0
## 1460 ERCC-00165  33   64   41
## 1461 ERCC-00168   2    1    1
## 1462 ERCC-00170   1    2    3
## 1463 ERCC-00171 916 1253 1056
```

Changing Column and Row Names

I want to make our data more visually friendly so I will repeat the modifications we performed in Intro-to-R.html. First, change the column names.

```
# Make a vector with new names for each column
new_names <- c("GeneID", "Sample_1", "Sample_2", "Sample_3")
# Assign the column names of the dataframe to the list of new column names
colnames(my_data) <- new_names
# Look at the top of the dataframe. You should see new column names at the top.
head(my_data)
```

```
##              GeneID Sample_1 Sample_2 Sample_3
## 1 ENSG00000277248.1        0        0        0
## 2 ENSG00000274237.1        0        0        0
## 3 ENSG00000280363.1        0        0        0
## 4 ENSG00000279973.1        0        0        0
## 5 ENSG00000226444.2        0        0        0
## 6 ENSG00000276871.1        0        0        0
```

Second, rename your rows to your gene ID's and remove the gene name column.

```
# Assign the row names of the dataframe to be the ID's found in the column GeneID
row.names(my_data) <- my_data$GeneID
# Remove the GeneID column by setting it to NULL
my_data$GeneID <- NULL
# Look at the top of the dataframe. The numbers on the far left should be replaced with the gene ID's a
head(my_data)
```

```
##                    Sample_1 Sample_2 Sample_3
## ENSG00000277248.1        0        0        0
## ENSG00000274237.1        0        0        0
## ENSG00000280363.1        0        0        0
## ENSG00000279973.1        0        0        0
## ENSG00000226444.2        0        0        0
## ENSG00000276871.1        0        0        0
```

Subsetting

There are so many ways to extract specific data from objects like dataframes in R. Most of them are useful but some are better in certain situations than others. Here I will demonstrate a few ways to subset in the context of cleaning our data for further analysis. No matter the method you choose, the general idea remains the same.

1. Identify the elements in your object (list, dataframe, etc.) that meets the condtions you require. Your condition requirements may be:
    - **Pattern matching**. *Example: looking for a group of genes with the same name*
    - **Setting a Threshold**. *Example: accepting read values greater than or equal to 1*
    - **Observations that meet multiple conditions**. *Example: accepting read values greater than or equal to 1 in at least 2 of 3 replicate samples*
2. Extract the elements that meet your conditions from the dataset
3. Store the extracted/subsetted data into a new object (list, object, etc.)

Subsetting with grep()

When we want to extract or subset information from the data that matches a pattern we can use the function grep(). This is an extremely helpful tool and you can read more on how to use it by typing ?grep into the console.

Here is a shortened form of grep() usage:

grep(pattern, x)

We are going to use grep() to extract our spike-in reads from the data. Where **pattern** will be replaced with "ERCC" (the spike-in name) and **x** will be replaced with what we want to search - in this case the row names of the **my_data** dataframe - row.names(my_data).

Note: The ^ before "ERCC" is a regular expression and indicates that we only want instances of "ERCC" when it is at the beginning of the line. If there was a gene named "brERCC", this would show up in our list if the ^ regular expression was not included.

Note: When using gene symbols in mammals, there is a family of genes called "ERCC ..." so it helps to use the dash at the end of the spike-in name to help clarify what we want to extract (e.g. "^ERCC-"). However, care needs to be taken when using symbols for pattern matching since many like ^ and . are regular expressions. Here is a link to a regular expression cheat sheet.

Using grep() on **my_data** results in a print out of all the elements containing our pattern of interest but not the element itself.

```
# Look at the bottom of the dataframe. It is organized in a way that the spike-in names are at the end.
tail(my_data)
```

```
##            Sample_1 Sample_2 Sample_3
## ERCC-00163        5        7       13
## ERCC-00164        0        1        0
## ERCC-00165       33       64       41
## ERCC-00168        2        1        1
## ERCC-00170        1        2        3
## ERCC-00171      916     1253     1056
```

```
# Identify the elements that have a row name starting with ERCC-
grep("^ERCC-", row.names(my_data))
```

```
##  [1] 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385
## [15] 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399
## [29] 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413
## [43] 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427
## [57] 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441
## [71] 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455
## [85] 1456 1457 1458 1459 1460 1461 1462 1463
```

We can save our results into a new object, I'll name it **ercc_results**.

```
# Store the results in a new object
ercc_results <- grep("^ERCC-", row.names(my_data))
```

Now using our indexing skills, we will extract all the rows that match to the elements in our **ercc_results** list. The column value will be omitted so that we can retrieve data from all of the columns. These results will be saved in a new object that I have named **spikes**.

```
# Subset all columns from the data using the list of indices provided in ercc_results
spikes <- my_data[ercc_results, ]
```

You can look at the top of the file using head().

```
# Print out the top 6 lines of the object.
# Note: tables are stored as an object called a data frame. So head() will print out the top lines of t
head(spikes)
```

```
##            Sample_1 Sample_2 Sample_3
## ERCC-00002    18892    23545    21054
## ERCC-00003     1447     1579     1511
## ERCC-00004      455      536      498
## ERCC-00009      318      385      354
## ERCC-00012        0        0        0
## ERCC-00013        0        0        0
```

```
# Check what type of object "spikes" is.
class(spikes)
```

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

Based on the documentation for the ERCC spike-in, we know that 92 transcripts should be identified. Using dim() to look at the dimensions of our **spikes** data, it indicates that there are indeed 92 rows/transcripts in all three of our samples.

Note: If there are more or less rows in your **spikes** dataframe, your subsetting may have included or excluded something that was not intended.

```
dim(spikes)
```

```
## [1] 92  3
```

It is important to note that the data stored in **my_data** is untouched. When we subsetted the "ERCC" spike-in data from the **my-data** dataframe, we made a copy of those rows and stored them in a new dataframe called **spikes**. If you look in the **Environment** tab in RStudio, you should see two objects listed: my_data (1463 obs. of 3 variables) and spikes (92 obs. of 3 variables).

Filtering Reads with Zeros

In many downstream analysis tools it can be problematic if zeros are present in the data. Some programs can adjust for this and filter the data for you. Alternatively, you can perform this process yourself by subsetting.

Subsetting with apply()

The function apply() is a powerful way to perform functions on multiple elements of an object like a dataframe. We can indicate if the function should be applied to each row or each column of the dataframe.

If you type ?apply in your console you will see the documentation in the **Help** tab. It tells us that we can use apply() in the following way.

Usage apply(X, MARGIN, FUN, ...)

X An array, such as the dataframe named **my_data**

MARGIN This determines what to filter, either rows or columns. Rows are indicated by 1 and columns are indicated by 2.

FUN The function to be applied to each row, column, or both as indicated by MARGIN. Do not include the parenthesis after the function name!

Using the function sum(), we can see how sum() can be applied to the data.

```
# Using the function sum() to take the sum of all the values in the dataframe
sum(my_data)
```

```
## [1] 313831
```

Here we apply sum() to the data set, indicating that this function should be performed on each row by setting the MARGIN argument to 1.

```
# Using apply to take the sum of each row of the data (indicated by 1).
# The result is a long list with each element and row sum listed. I don't want it to print out to the c
rows <- apply(my_data, 1, sum)
head(rows)
```

```
## ENSG00000277248.1 ENSG00000274237.1 ENSG00000280363.1 ENSG00000279973.1
##                 0                 0                 0                 0
## ENSG00000226444.2 ENSG00000276871.1
##                 0                 0
```

Here we apply sum() to the data set, indicating that this function should be performed on each column by setting the MARGIN argument to 2.

```
# Using apply to take the sum of each column of the data (indicated by 2)
apply(my_data, 2, sum)
```

```
## Sample_1 Sample_2 Sample_3
##    94489   116118   103224
```

To filter our reads, I will use the function all(). In a nutshell, all() will determine if all the values are non-zero. If they are all non-zero, the result will be TRUE. If one or more values is zero, the result will be FALSE.

Here are some examples to better understand all().

```
# What does the first line of the dataframe look like?
my_data[1, ]
# The first line is all zeros. Test to make sure we get the correct result using all(). Are all of the
all(my_data[1, ])
# Print row 95 to the console.
my_data[95, ]
# What result do you get when you use all()? Are all of the values non-zero?
all(my_data[95, ])
# What does the last line look like?
my_data[1463, ]
# The last line of the dataframe does not have any zeros in any of the columns. Test to make sure we ge
all(my_data[1463, ])
```

Now we can select all the rows that do not contain zero values by applying all() to the data.

1. Identify all the elements/rows of the data that do not contain zero values
2. Copy all the non-zero elements to a new dataframe
3. Store the dataframe with a new name

```
# Using apply() to identify the elements/rows in the dataframe that have all columns with non-zeros.
# Where X = my_data, MARGIN = 1 (indicating rows), FUN = all
read_filter_all <- apply(my_data,
                         1,
                         all)

# Subsetting the elements that were identified as not having any zeros and storing it in a new datafram
filtered_data_all <- my_data[read_filter_all, ]
```

```r
# Look at the top of the data. Already we can see that the first few rows containing zeros are not incl
head(filtered_data_all)
```

```
##                     Sample_1 Sample_2 Sample_3
## ENSG00000206195.10       10        4        8
## ENSG00000100181.21       19       23       20
## ENSG00000241717.1         2        2        3
## ENSG00000237689.1         1        3        2
## ENSG00000280156.1         1        1        1
## ENSG00000237438.6        22       27       32
```

```r
# The dimensions of the filtered dataset presented in rows (565) and columns (3)
dim(filtered_data_all)
```

```
## [1] 565   3
```

But what if you wanted to set a threshold for a minimum number of read counts greater than zero? We can do this using apply() but making our own function. The function we make will return rows with a minimum number of reads in a minimum number of columns.

For example:

- minimum of 1 read in a minimum of 3 columns
- minimum of 5 reads in a minimum of 3 columns
- minimum of 5 reads in a minimum of 2 columns
  - Since we have 3 columns, this would not exclude zero values in one column. If you didn't want any zeros you would do this on the data after all rows with zeros were excluded.

Don't worry about the function itself right now, creating your own function will be discussed in the Bioinformatics Support Group - Intermediate R meeting.

```r
# The minumum number of reads I want
min_reads <- 1

# The minimum number of samples that must contain the minimum read value (min_reads)
min_samples <- 3

# Using apply() to identify the elements in the dataframe that match our conditions. In this case, the
read_filter <- apply(my_data,
                     1,
                     function(samples) length(which(samples >= min_reads)) >= min_samples)

# Subsetting the elements that were identified using our conditions and storing it in a new object/data
filtered_data <- my_data[read_filter, ]

# Look at the top of the data. Already we can see that the first few rows containing zeros are not incl
head(filtered_data)
```

```
##                     Sample_1 Sample_2 Sample_3
## ENSG00000206195.10       10        4        8
## ENSG00000100181.21       19       23       20
## ENSG00000241717.1         2        2        3
## ENSG00000237689.1         1        3        2
## ENSG00000280156.1         1        1        1
## ENSG00000237438.6        22       27       32
```

```
# Our conditions duplicated the same conditions when we applied all() to the data. Thus, we expect the
dim(filtered_data)
```

```
## [1] 565   3
```

I encourage you to reassign the variables **min_reads** and **min_samples** by replacing the numbers after <- so you can see how it changes the resulting subsetted data.

Write Data to a File

Now that you have cleaned up your file you can save it in a way that is accessible beyond the R environment. These files can be shared and easily opened with Excel if need be.

write.csv()

The easiest way to export your data to a file is by saving it as a comma separated value file - also known as csv. Use the function write.csv() to indicate the object that should be saved and where is should be saved on your computer. Learn more by typing ?write.csv into the console.

Make sure to replace my file path ("/Users/stacey/Desktop/filtered_data.csv") with yours!

```
# Save the dataframe named filltered_data to a .csv file.
# Replace my file name path with yours!!!
write.csv(filtered_data,
          file="/Users/stacey/Desktop/filtered_data.csv")
```

Last but Not Least...

Helpful Hints

- Google is your friend!
  - Copy and paste errors directly in your search bar
  - Always include the programming language that you are talking about in your search (e.g. "how to subset in R", "mean R", "swirl R", "how to delete a dataframe column by name in R")
- Swirl is a great interactive way to learn more about R. You can get started by following this link.
- If you need to clear your environment (remove all variables, loaded data, etc.) you can type this into your console: **rm(list=ls())**

Helpful Concepts and Functions to Keep Mind and Research Further

- Subsetting
  - subset()
  - which()
  - dplyr()
  - Just a few functions to get started with.
- apply()
  - The apply family is your friend, it can look scary but don't avoid it.
  - There are a ton of tutorials on apply(), check them out! I recommend focusing on apply() first, the other apply() family members can wait.
- str()
  - This function displays the internal structure of an R object.
  - Type ?str into the console to learn more about this function.