R Basics Part 1

Introduction to data analysis with R on Sapelo2.

Hello! If you have come across this document, you are likely a user of the Sapelo2 supercomputer at the University of Georgia. If not, that's fine too, though this document may be confusing to you because it was specifically designed to be used as supplemental material for training offered through the Georgia Advanced Computational Resource Center. This document was designed to be used in conjunction with training offered through zoom. Most of you reading this document will have already gone through this training. If you have not gone through the R training pertaining to this document, feel free to sign up here:

https://wiki.gacrc.uga.edu/wiki/Training#How_to_Register

This training is given as a presentation. The slide number which corresponds to the section will be printed in the left margins of this document. The slides are sent out before training but can also be found here:

https://wiki.gacrc.uga.edu/images/b/ba/R_Language_Basics_PowerPoint_v1.0.pdf

(Slide 2)

By the end of this training you should be able to:

- Have R ready to use on your machine either through Sapelo2 or locally
- Recognize and create the basic R objects such as dataframes,vectors and list
- Interact with your file system and load data into your environment
- Get quick info on the data through built in functions
- Manipulate dataframes and create new dataframes from old
- Use other's code, I.e packages, to expand your options
- save/export data
- Run a batch job on Sapelo2 using R code

Getting Started 2
Feature or the R language 2
Downloading R or Running on Sapelo2 3
Packages 5
A quick note on Functions 6

Using R 7
Data types 7
Dataframes 9
Interacting with the file system 11
Importing data 11
Quick Analysis 12
Subsetting and Checking Data Distribution 12
dplyr 14
Exporting Figures and Data 14

Batch Job Example 15

Up next and helpful links 15
Vectorization and Subsetting 15

Getting Started

Features of the R language

(slide 3)

R is a high-level language, which means it's closer to human language than computer language and therefore more intuitive. R is easier to read and write than lower level languages.

R is vectorized. This means most functions can operate on all elements of a vector without needing to loop through and act on each element one at a time. This makes writing code more concise, easy to read, and faster.

R packages are collections of functions and data sets developed by the community. There are thousands of packages that cover many areas including visualization, genetics,machine learning,and even a package that simulates Gucci Mane telling you when your script is done running.

The Bioconductor project provides access to thousands of packages developed specifically for the management and exploration of high-throughput genomic data. static graphics, which can produce publication-quality graphs

R is comparable to popular commercial statistical packages such as SAS, SPSS, and Stata, but R is FREE, can do a lot more, and is much more customizable.

R provides the intuitive data frame structure that simplifies the manipulation of data by end users with little programming experience.

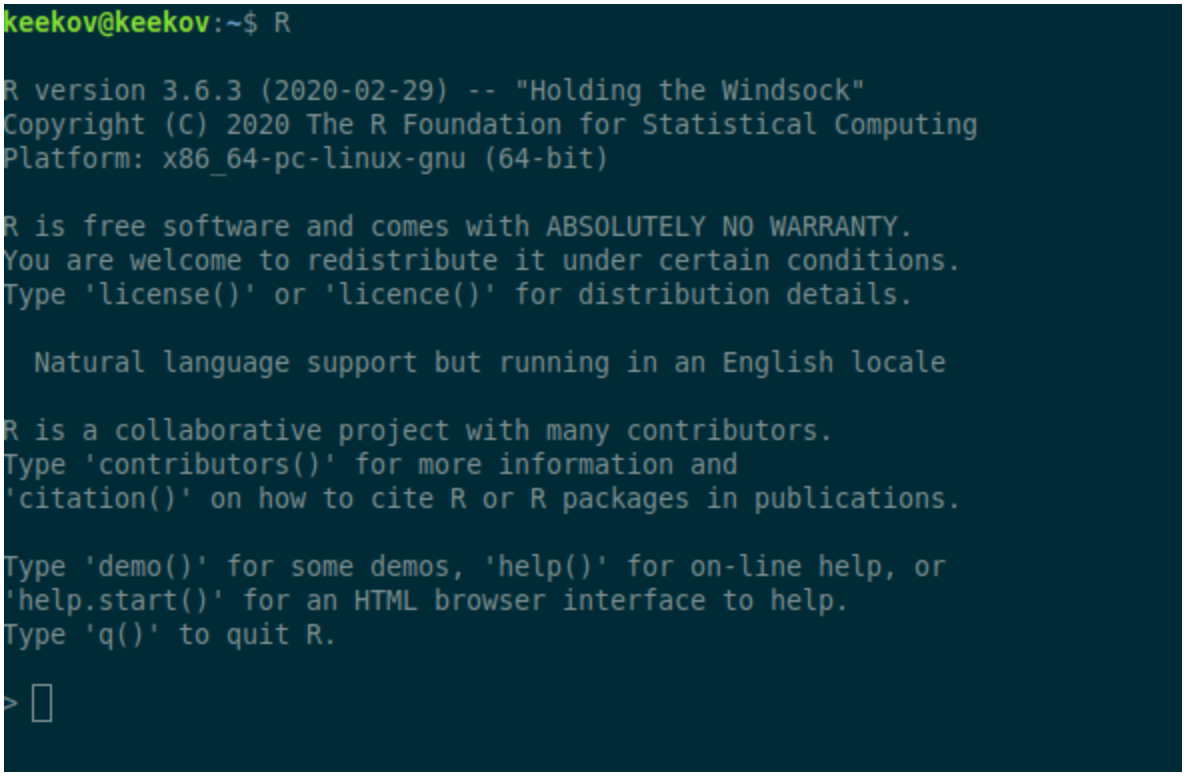
R allows for the easy creation of documents with R-Markdown. These include static and dynamic output such as html, PDF, MS-Word, shiny applications, websites and more.

R can be integrated with the git for easy code development and collaboration.

Downloading R or Running on Sapelo2

(slide 4)

The R project website has installers for Microsoft Windows, Apple OSX, and Linux.
https://www.r-project.org/
Many of you may have already gone through this process to download R onto your local computer.
On Linux and Mac, once R is installed just type “R” into your command line and the following should appear:



On Windows and Mac, you can open the console and navigate to R.exe, wherever it was installed. Then type R.exe to run R. You can also double click the R icon in your applications.
Press the Windows key in Windows or open Finder in Mac to access your applications. Once R.exe is launched it should open a new window. This is running R in "console-mode".

(slide 5)

On Sapelo2, login to the interactive node first by running the qlogin command. To see which R versions are available you can enter :

```
ml spider R
```

There are multiple versions or R available on Sapelo2. This is because some software may require older versions. R 4.0 is still relatively new. For this tutorial we will use R 4.0.0. To use the module enter the command:

```
ml R/4.0.0-foss-2019b
```

Then type **R** and press enter.

```
[keekov@c1-7 ~]$ ml R/4.0.0-foss-2019b
[keekov@c1-7 ~]$ R

R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

The > symbol indicates R is ready for a new command. After typing a command, press enter to run it. The output will be displayed and R will be ready for a new command.
The # symbol signifies a **comment**. Lines of code with # in front will not be evaluated.
Tab can be used to auto complete.
The up and down arrow keys toggle among the last lines of code which were run.
Entere **quit()** to quit R.
Use CTRL+C to end a process, such as an infinite loop, without having to quit R.

If you would like to see output graphics you either need to be running R from your local machine or to enable X-forwarding when you ssh into the login node of Sapelo2. This can be done in Linux by simply adding the -X option when ssh-ing into Sapelo2. To enable X-forwarding on mac or Windows see question 10 and 11 here: https://wiki.gacrc.uga.edu/wiki/Frequently_Asked_Questions

You would also need to use the xqlogin command as opposed to qlogin. For this training you will ***not*** need X-forwarding enabled. I will have it enabled while presenting in case I'd like to display figures.

R scripts are typically saved with the .R extension. For example, testcode.R adds 2 and 3 together and displays the result. testcode.R is simply a text file with the line "print(2+3)". To run testcode.R in the command line use **Rscript**

```
(base) keekov@keekov:~/R$ Rscript testcode.R
[1] 5
```

To run testcode.R while R is running use **source**. Don't forget the quotes!

```
> source("testcode.R")
[1] 5
```

Packages

(slide 6)

R packages are collections of functions and data sets developed by the community. They increase the power of R by improving existing base R functionalities, or by adding new ones.
A default installation of R consists of a small set of foundational or core packages which offer basic functionality in terms of data manipulation, statistical tests, analysis, and visualization. More packages can be added later, when they are needed for some specific purpose.

A simple analogy would be borrowing books from your local library. Your library comes with a set of books just as your download of R comes with basic R packages. If you want a book which your library doesn't have, you can ask the librarian if that book is in the library's network. If so, the librarian can get that book sent to your library for you to use. This is similar to how installing a package works in R. If the package is in the network, specifically the Comprehensive R Archive Network(CRAN), you can use the "install.packages" command to install the package in your library. If the book you want is outside of the network, you may have to get a copy yourself for your library. This is more complicated and won't be covered in this training.

To install packages locally(not on Sapelo) use
`install.packages("Package Name")`
Even if you don't see the package in CRAN, just try this out and it may work anyway.

Once the book is available at your library you have to go check it out. Similarly, when we launch R, we have access to basic packages but we must explicitly tell R we want to load others from our library.

To load a package so that you can use it during an R session use:
`library("package Name")`

To install packages on Sapelo2, reach out to GACRC and we can install the packages for you to avoid causing issues with software which may depend on the current packages. You can also download packages into your home directory.
Instructions on how to install packages in your home directory can be found here:
https://wiki.gacrc.uga.edu/wiki/Installing_Applications_on_Sapelo2#How_to_install_R_packages

Bioconductor
Bioconductor is managed separately from CRAN so the installation of packages involves a different process due to the way Bioconductor packages are developed.

A quick note on Functions

(slide 7)

str() is an example of a function
str works on many different objects in R. The output for a numeric and a Dataframe are very different so how does str() know what to do?
Below is an example of running the string function on a dataframe(ExampleDataFrame) and running str on the number 3.

```
> str(ExampleDataFrame)
'data.frame':   3 obs. of  5 variables:
$ Animal         : Factor w/ 3 levels "chicken","human",..: 3 1 2
$ NumberOfHands  : num  0 0 2
$ WarmBlooded    : logi  FALSE TRUE TRUE
$ MultipleSpecies: logi  TRUE TRUE FALSE
$ GreatAnimal    : chr  "yes" "yes" "yes"
> str(3)
num 3
```

Functions will act differently based on what class they are acting on
str() is a generic function. Actually, it has a collection of a number of methods. You can check all these methods with methods(str).

```
> methods(str)
[1] str.data.frame* str.Date*       str.default*    str.dendrogram*
[5] str.logLik*     str.POSIXt*
see '?methods' for accessing help and source code
>
```

When str() is run it first checks the data type of the object it's running on. Then it picks the method to run.
The input to a function could be one object or many, even another function. To check the documentation of a function you can run ? followed by the functions name
?str
?mean
Etc..
The documentation should include the default **arguments**(inputs) to functions. If you don't explicitly set the variables the function assumes the default values. When **calling** a function you can specify arguments by position, by complete name, or by partial name.
Here is the documentation for the mean() function:

[mean function | R Documentation]

Usage

```
mean(x, …)

# S3 method for default
mean(x, trim = 0, na.rm = FALSE, …)
```

The arguments include x(the data) and trim, which is how to round before calculating the mean. As well as na.rm, which is to ignore NA values.
Running mean(exampledata,0.5,FALSE) is the same as mean(x=exampledata,trim=0.5,na.rm=FALSE) and mean(trim=0.5,x=exampledata,na.rm=FALSE) The arguments can only be out of order if they are named!

Basic Data Analysis Using R .

Data types

(slide 8 / 9)

Object-Oriented Programming is a programming model in which different methods are used to design software around data or objects rather than programming around functions. It is object-oriented because all the processing revolves around the **objects.** Every object has different attributes and behavior.
In R, everything is an object, which has a type and belongs to a class.
The class is the blueprint that helps to create an object and contains the object's attributes.
There are various kinds of R-objects or data structures. Some of the basic data-types, on which other R-objects are built, include Numeric, Integer, Character, Factor, and Logical.

Individual values are either **character** values (text), **numeric** values (numbers), or **logical** values (TRUE or FALSE). *R* also supports complex values with an imaginary component.

There is a distinction within numeric values between integers and real values, but integer values tend to be coerced to real values if anything is done to them. The function integer() can turn a numeric back into an integer.

The simplest data structure in *R* is a **vector**. All elements of a vector must have the same basic type. Most operators and many functions accept vector arguments and return a vector result. This method of coding in R is much more efficient than using a loop to run through each value in the vector. Depending on the task, vectorization can be up to hundreds of times faster than using loops.

Matrices are multidimensional analogues of the vector. All elements must have the same type.

Data frames are collections of vectors where each vector must have the same length, but different vectors can have different types. Data frames are the standard way to represent a data set in *R*.

Lists are like dataframes but each component can be a vector of a different size. More generally, the components of a list can be more complex data structures, such as matrices, data frames, or even other lists. Lists can be used to efficiently represent hierarchical data in *R*.

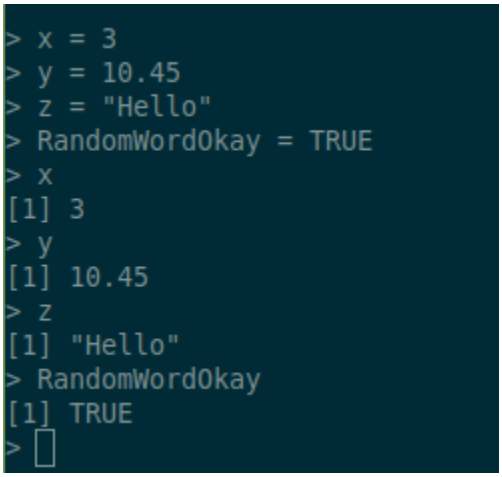
Examples of some basic data-types:
Characters “a”, “Hello”
Numeric 2, 12.4
Logical TRUE,FALSE
Vector (1,22,3,15,20)
List (1,3,”green”,FALSE)

Another word for an object is an **instance**. Although synonyms, usually we think of an instance as a concrete occurrence of an object existing during the runtime of the program.

To assign a value to an object, use <- or =
They are technically different and some people are picky about this but there are no practical reasons to use one over the other. Don’t use both. This has been the source of many bitter arguments for me so be warned. Here is my biased source comparing them:

[Assignment operators in R: '=' vs. '<-']

An object like a numeric can vary in it’s value, so often it’s called a **variable**. Any named object that contains a value could be called a variable. So often creating an object is called “declaring a variable”
After creating an object in R we can type it’s name and hit enter to see it printed out.



There are helper functions (which are themselves objects) that will help determine the structure of a given object. The str() function does a good job of telling you what the type and structure of an object is.

```
> str(x)
 num 3
> str(z)
 chr "Hello"
> str(RandomWordOkay)
 logi TRUE
>
```

The function **object.size()** returns the approximate number of bytes used by an *R* data structure in memory.

To see what objects you have in your **environment** use ls()

```
> ls()
[1] "RandomWordOkay" "x"              "y"              "z"
>
```

To remove an object from your environment use rm()

```
> rm(x)
> ls()
[1] "RandomWordOkay" "y"              "z"
>
```

Dataframes

(slide 10)

The most common data structure for tabular data is the **Dataframe**
It is like a matrix but can hold different data types.
Vectors are made using the notation c(1,2,"red","blue",...etc)
You can make a dataframe from three vectors using the function data.frame()
Then you can use a function like str() or attributes() to get a summary of the data-frame.

(Slide 11)

The $ character is used to reference and create new columns.

```
> Animal = c("snake","chicken","human")
> NumberOfHands = c(0,0,2)
> WarmBlooded = c(FALSE,TRUE,TRUE)
> ExampleDataFrame = data.frame(Animal,NumberOfHands,WarmBlooded)
> ExampleDataFrame
   Animal NumberOfHands WarmBlooded
1   snake             0       FALSE
2 chicken             0        TRUE
3   human             2        TRUE
> attributes(ExampleDataFrame)
$names
[1] "Animal"        "NumberOfHands" "WarmBlooded"

$class
[1] "data.frame"

$row.names
[1] 1 2 3

>
```

```
> ExampleDataFrame$Animal
[1] snake   chicken human
Levels: chicken human snake
> ExampleDataFrame$MultipleSpecies = c(TRUE,TRUE,FALSE)
> ExampleDataFrame
   Animal NumberOfHands WarmBlooded MultipleSpecies
1   snake             0       FALSE            TRUE
2 chicken             0        TRUE            TRUE
3   human             2        TRUE           FALSE
>
```

Trying to add a vector as a column which doesn't have the same number of rows as the data frame will give an error. You can also add a column with identical values.

```
> ExampleDataFrame$GreatAnimal = "yes"
> ExampleDataFrame
   Animal NumberOfHands WarmBlooded MultipleSpecies GreatAnimal
1   snake             0       FALSE            TRUE         yes
2 chicken             0        TRUE            TRUE         yes
3   human             2        TRUE           FALSE         yes
>
```

Interacting with the file system

(slide 12)

getwd() , setwd() - get and set the current directory

Be sure that you use quotes around your path or R will think you are looking for an

object you have declared as opposed to a file.

list.files() - List the files in the current directory or a specified location.

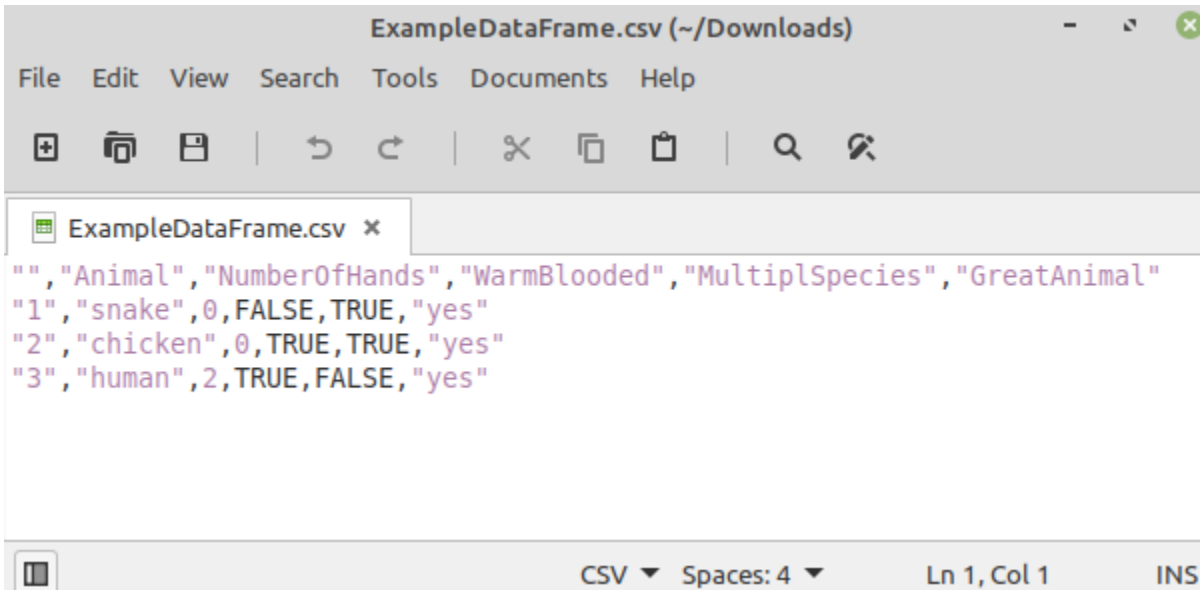
```
> list.files()
[1] "Desktop"   "Documents" "Downloads" "Music"     "Pictures"  "Public"
[7] "R"         "Templates" "Videos"
> getwd()
[1] "/home/keeko"
> list.files()
[1] "Desktop"   "Documents" "Downloads" "Music"     "Pictures"  "Public"
[7] "R"         "Templates" "Videos"
> setwd("Downloads/")
> getwd()
[1] "/home/keeko/Downloads"
>
```

file.remove(), file.rename(), file.copy(), dir.create() - file manipulation

Importing data

(slide 13)

The most common type of data is csv, or “comma separated values”. This is just a formatted text file where the values are separated by commas. There is also a bit more formatting to let R know what the rows and columns are. Csv can also be opened by Excel easily. This is a csv file opened in a text editor:



A variety of data types can be loaded. Txt Data can be loaded with read.delim() and read.csv() for csv files. Read.excel() reads excel files.
For the following examples we will use data called ML_Data.csv
On Sapelo2 it is located at /usr/local/training/RTraining/ML_Data.csv
It was also emailed to you before training as an attachment.

For this analysis, we will use data from the Kaggle Machine Learning & Data Science Survey. This data was collected from surveys which were run by Kaggle. Kaggle is an extremely useful online community which focuses on data science and machine learning. It hosts data-sets,tutorials,competitions and much more. The page for the data we will be using can be found here:

https://www.kaggle.com/c/kaggle-survey-2019/overview

The surveys had 23,859 (2018), 19,717 (2019), and 20,036 (2020) valid responses from participants in 171 different countries and territories. The data was randomized column wise so one row doesn't necessarily correspond to one person.

Quick Analysis

(slide 14)

Useful Data Frame Functions
head() - shows first 6 rows
tail() - shows last 6 rows
dim() - returns the dimensions of data frame
ncol() - number of columns
str() - structure of data frame - name, type and preview of data in each column
names() or colnames() - both show the names attribute for a data frame
Table() - builds a contingency table of a column

```
> table(responses$Q1)

                    18-21                     22-24
                     2502                      3610
                    25-29                     30-34
                     4458                      3120
                    35-39                     40-44
                     2087                      1439
                    45-49                     50-54
                      949                       692
                    55-59                     60-69
                      422                       338
                      70+ What is your age (# years)?
                      100                         1
```

Subsetting and Checking Data Distribution

(slide 15)

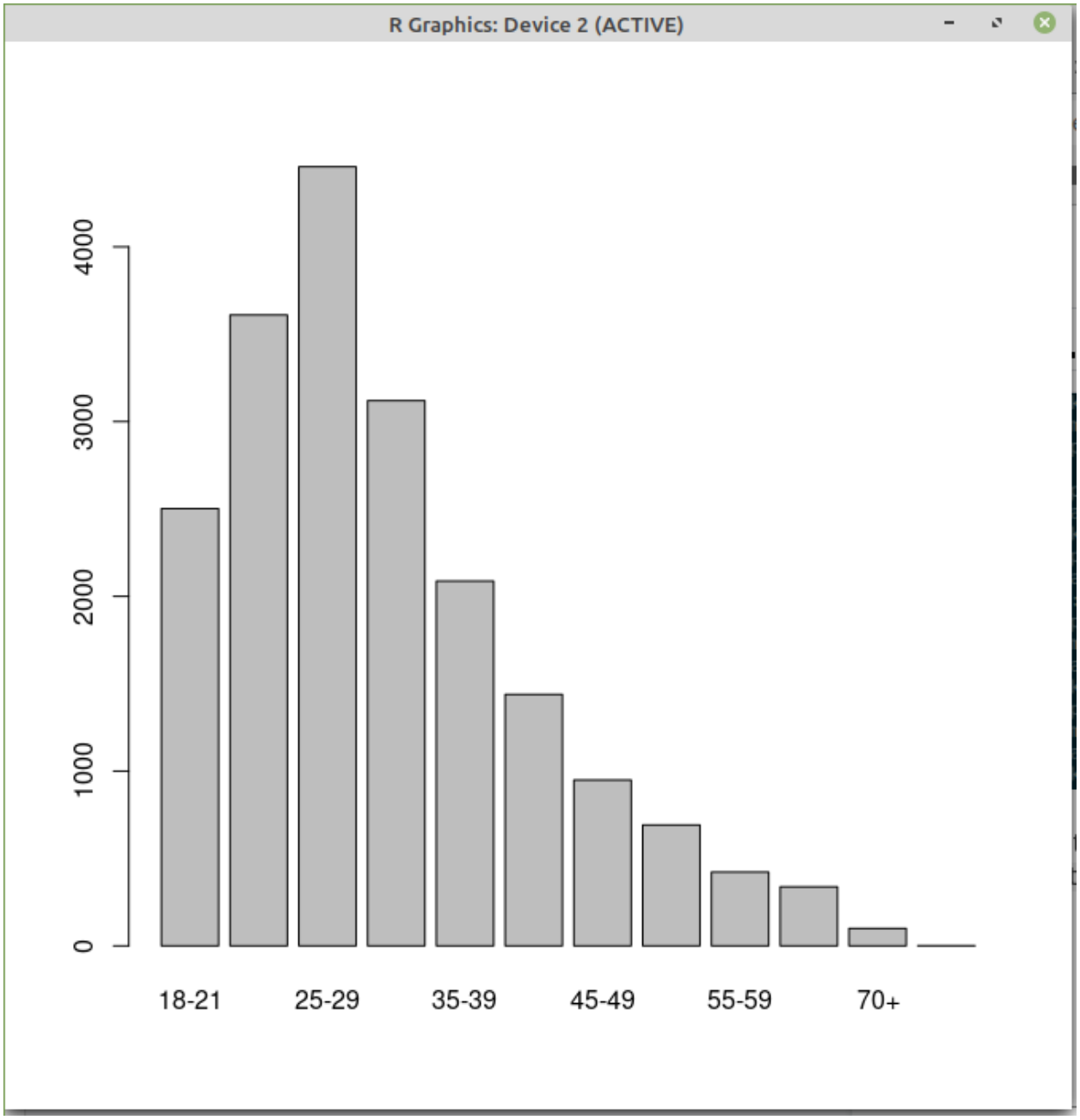
We would like to take a quick look at the ages of survey participants. This corresponds to column 2 of the data.

In order to select one column we need to learn how to subset a dataframe. One way is through bracket notation. First you put the name of the dataframe, followed by brackets containing the row number a comma and then the column number. The left number could be one row or many. The right number could be one column or many. If the number is left blank then it's assumed we want all of either the rows or columns. Use : to specify a range. Use c() to pick specific values.

```
> ExampleDataFrame
   Animal NumberOfHands WarmBlooded MultiplSpecies GreatAnimal
1   snake             0       FALSE           TRUE         yes
2 chicken             0        TRUE           TRUE         yes
3   human             2        TRUE          FALSE         yes
> ExampleDataFrame[1,2]
[1] 0
> ExampleDataFrame[1,]
  Animal NumberOfHands WarmBlooded MultiplSpecies GreatAnimal
1  snake             0       FALSE           TRUE         yes
> ExampleDataFrame[,1]
[1] snake   chicken human
Levels: chicken human snake
> ExampleDataFrame[1:2,1:3]
   Animal NumberOfHands WarmBlooded
1   snake             0       FALSE
2 chicken             0        TRUE
> ExampleDataFrame[1:2,c(1,3)]
   Animal WarmBlooded
1   snake       FALSE
2 chicken        TRUE
```

To plot the ages of survey takers we will use the plot() function.

```
plot(responses[,2])
```



---

dplyr

(slide 17)

The package dplyr, written by Hadley Wickham, is an optimized set of functions designed to work efficiently with data frames. This package is available through CRAN to add to your library and is also available on all versions of R on Sapelo2.
The dplyr package provides a series of “verbs”(functions) which you can do to dataframes.The `dplyr` functions are very fast, as many key operations are coded in C++.

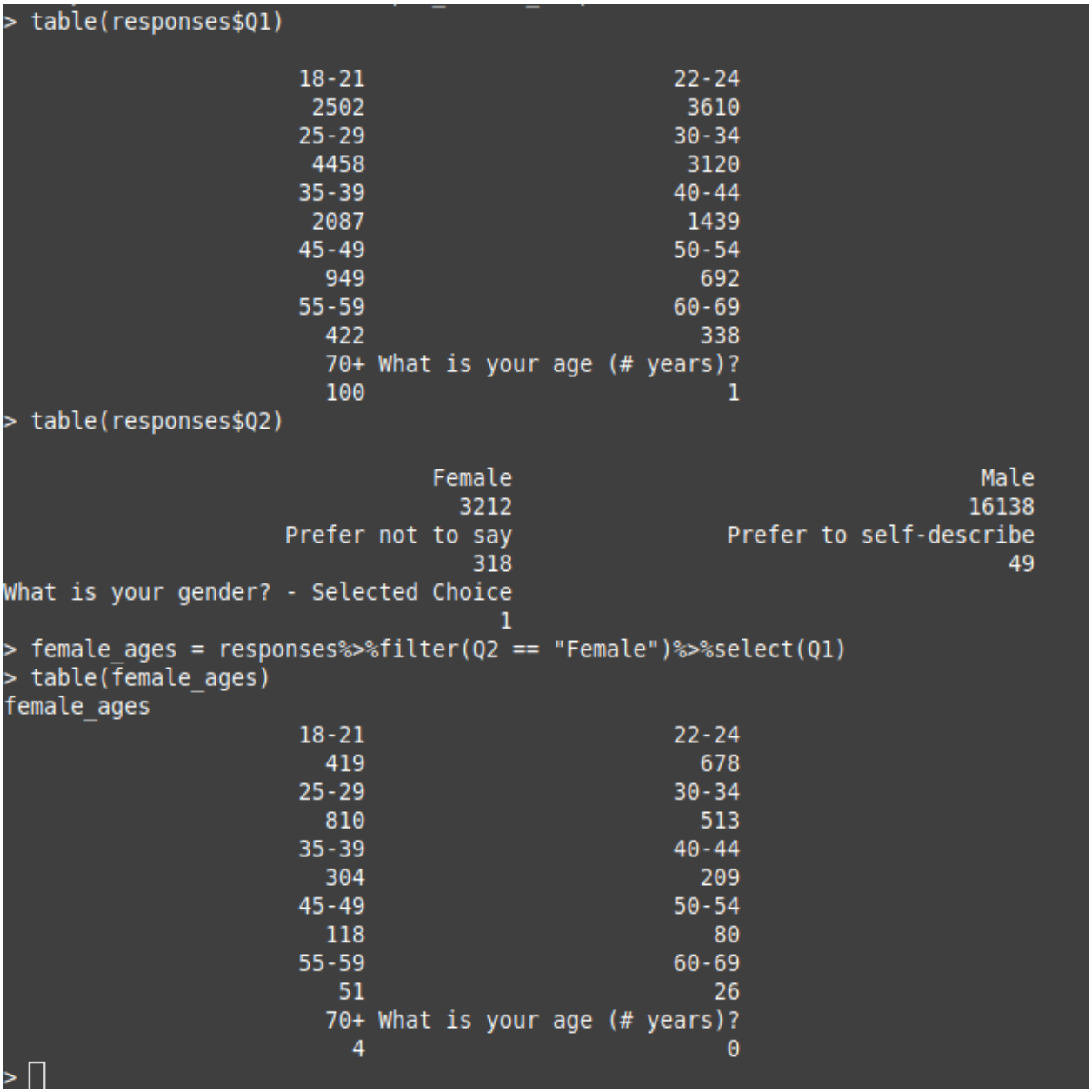
The following are some functions you can use on a dataframe:

- `select`: return a subset of the columns of a data frame, using a flexible notation
- `filter`: extract a subset of rows from a data frame based on logical conditions
- `arrange`: reorder rows of a data frame
- `rename`: rename variables in a data frame
- `mutate`: add new variables/columns or transform existing variables
- `summarise` / `summarize`: generate summary statistics of different variables in the data frame
- `%>%`: the “pipe” operator is used to connect multiple verb actions together into a pipeline

For all of these functions, the first argument is a data frame, the subsequent arguments describe what to do with the data frame. You can refer to columns by name without using the $ operator. The output of the functions is a new data frame.

The pipe operator allows for the input of one function to be the output of the next. Functions can be strung together so that we can perform a series of actions on a dataframe. For example, if we’d like to **select** the age column but have it be **filtered** by only female participants first then we can run:

```
female_ages = responses%>%filter(Q2 == "Female")%>%select(Q1)
```



Why does:

```
female_ages = responses%>%select(Q1)%>%filter(Q2 == "Female")
```

Not work? Take a look at the data set:

```
female_ages = responses%>%select(Q1)
```

To see why.

The groupby() and summarise() functions are often used together. First the data is grouped using group_by. This creates a key of groups which can be accessed with group_keys()

```
> responses%>%group_by(Q2)%>%group_keys()
# A tibble: 5 x 1
  Q2
  <fct>
1 Female
2 Male
3 Prefer not to say
4 Prefer to self-describe
5 What is your gender? - Selected Choice
>
```

The summarise function creates a dataframe where the group keys are the first column. The next columns are created as arguments to the function. These new columns are summarised versions of the original data frame columns per group.

In this example we will group by age range(Q1) and then create a column for the count of how many rows were in that group, a column for how many females were in each group, and a column for how many males were in each group.

```
> gendersummary = responses%>%group_by(Q1)%>%summarise(count = n(),Females = sum(Q2=="Female"), Males = sum(Q2=="Male"))
> gendersummary
# A tibble: 12 x 4
   Q1                          count Females Males
   <fct>                       <int>   <int> <int>
 1 18-21                        2502     419  2049
 2 22-24                        3610     678  2880
 3 25-29                        4458     810  3562
 4 30-34                        3120     513  2552
 5 35-39                        2087     304  1743
 6 40-44                        1439     209  1201
 7 45-49                         949     118   810
 8 50-54                         692      80   598
 9 55-59                         422      51   357
10 60-69                         338      26   306
11 70+                           100       4    80
12 What is your age (# years)?     1       0     0
>
```

Exporting Figures and Data

(slide 18)

There are a number of useful output types (PDF, JPEG, PNG, SVG) in addition to the default high resolution on-screen graphics capability.

There are three primary user-focused graphics systems: Base, Lattice, and ggplot2. It is useful to consider that the first graphics package, Base, was the original default display package for R with the other three being added over time as the R language evolved and matured.

Saving pdf Example:

```
pdf("rplot.pdf")
 plot(responses[,2])
dev.off()
```

Batch Job Example

Example Batch job at
/usr/local/training/RTraining/Rsub.sh

Up next and helpful links

R studio is software that includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.
UI for connecting to git, managing packages and project management.

https://rstudio.com/

R markdown supports outputs such as HTML, PDFs, MS-Word documents, applications, websites and more.

https://rmarkdown.rstudio.com/

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data.
https://www.bioconductor.org/

R cheat sheets:
https://rstudio.com/resources/cheatsheets/

Including ones for importing data, applying functions, and base R→
http://github.com/rstudio/cheatsheets/raw/master/base-r.pdf