R Basics

(slide 1 and 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: dataframes,vectors,list
Load data into your environment
Get quick info on the data through built in functions and graphics
Manipulate dataframes and create new dataframes from old
Use other's code, I.e packages, to expand your options
Save publication quality graphics
save/export data

Uses for 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.

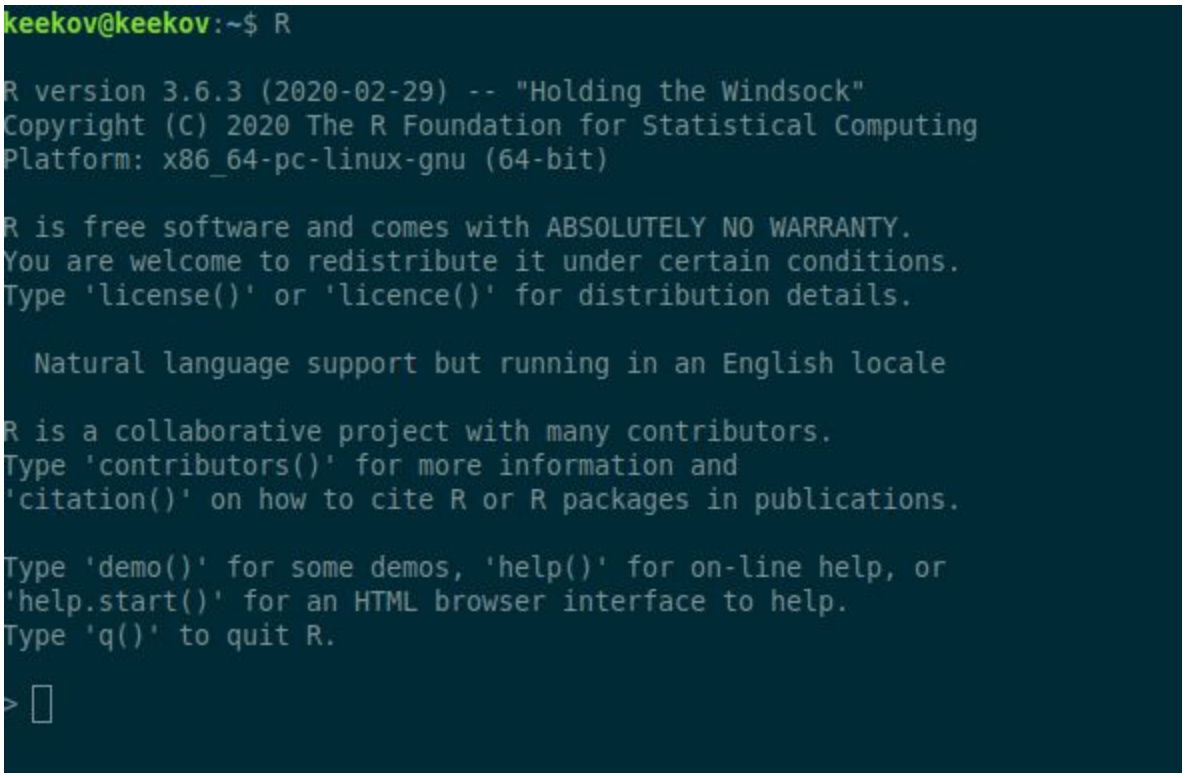
R allows for the easy creation of documents with R-Markdown. These include static and dynamic output such as html, PDF, MS-Word, shiny applications, websites and more.

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

Downloading R or Running on Sapelo2

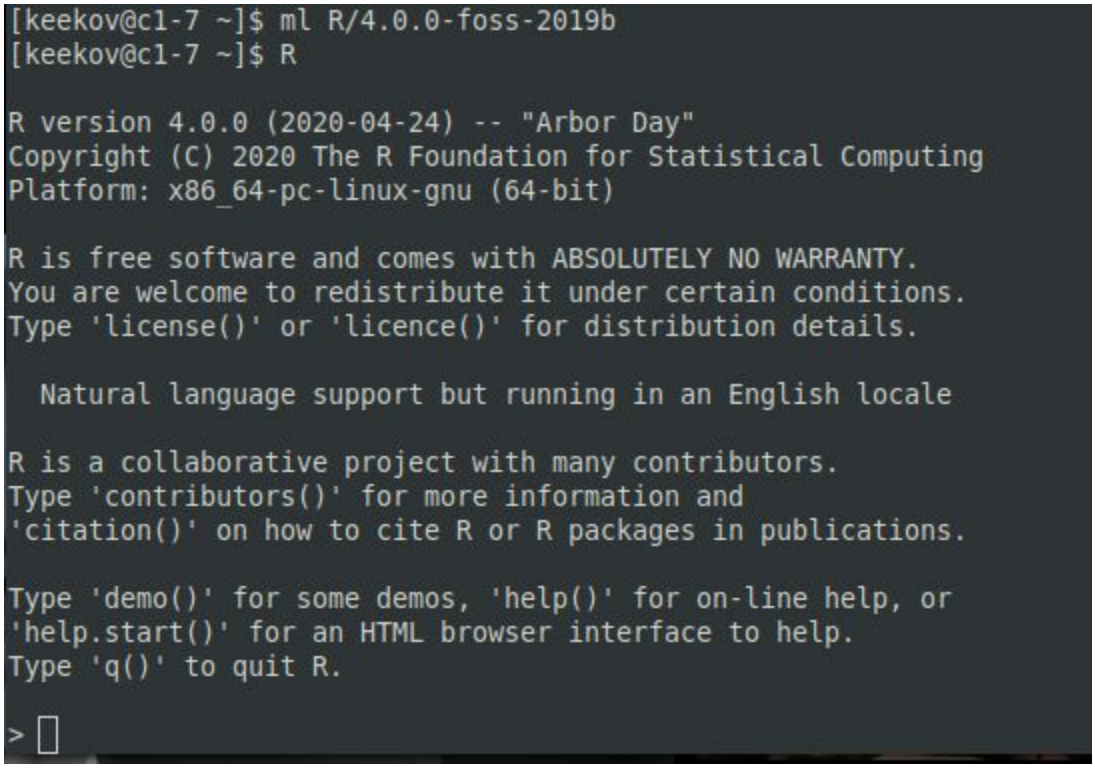
(slide 4)

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.
On Linux and mac, once R is installed just type “R” into your command line and the following should appear:



On Windows and Mac-OS 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 xqlogin command. Then enter the command
 ml R/4.0.0-foss-2019b
 to load the R module. Then type R and press enter.



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.

For part of this tutorial we will need to see the output of some graphics. To do this you either need to be running R from your local machine or to enable X-forwarding when we 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

For the example BASH script below, which we will submit to Sapelo2, you will not need X-forwarding enabled.

R scripts are typically saved with the .R extension. For example testcode.R adds 2 and 3 together and displays the result. 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.

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

Data types

(slide 6 and 7)

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:

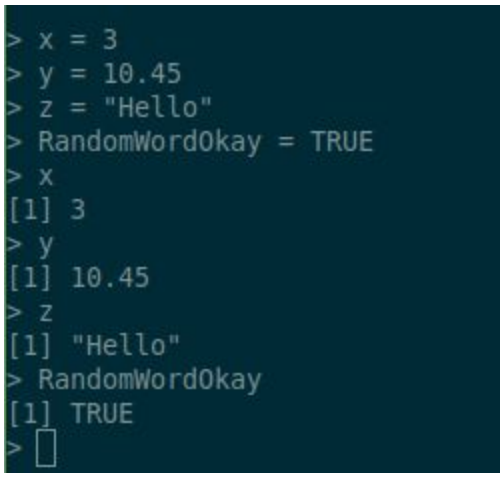
Characters	“a”, “Hello”
Numeric	2, 12.4
Logical	TRUE,FALSE
Vector	(1,22,3,15,20)
List	(1,3,”green”,FALSE)

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.
To use the function called str(), put what you want the function to act on inside the ()

```
> 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 8)
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 9)
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
>
```

A quick note on Functions

(slide 10)

str() is an example of a function

str works on many different classes. The output for a numeric and a Dataframe are very different so how does str() know what to do?

```
> 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!

Vectorization and Subsetting

(slide 11)

Vectorization and subsetting are features of R that allow efficient calculations to occur. Vectorization is often built into functions.

Instead of looping an action is done to each member of a vector in one command

```
> testnumbers
 [1]  1  2  3  4  5  6  7  8  9 10
> testnumbers + 3
 [1]  4  5  6  7  8  9 10 11 12 13
```

Comparisons using vectors are also very fast

```
> testnumbers < 5
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

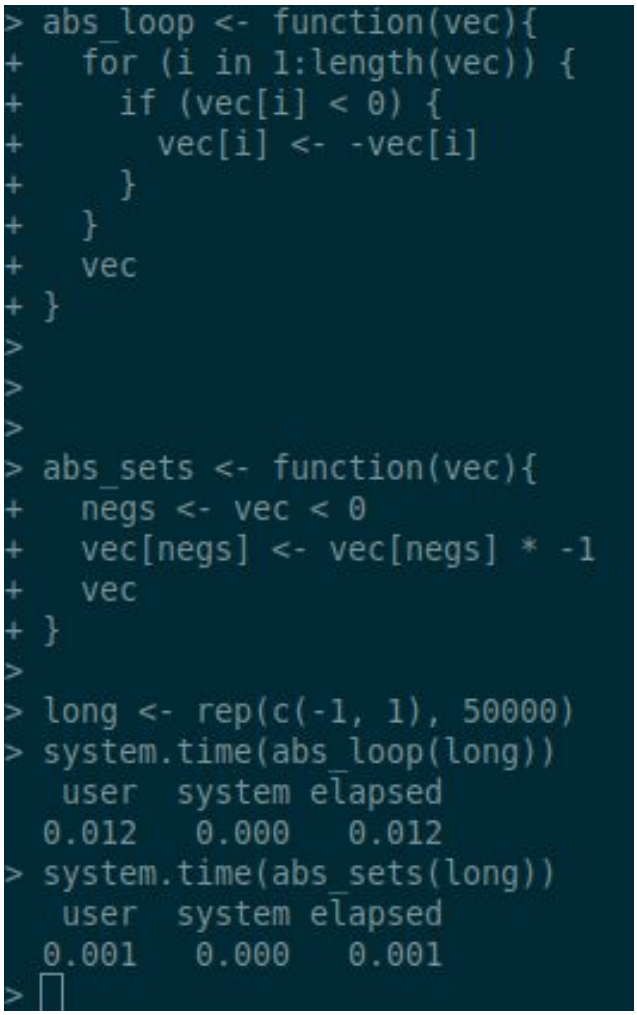
As well as subsetting a vector

```
> testnumbers[testnumbers<5]
[1] 1 2 3 4
>
```

Use vectors and subsetting whenever you can to speed up processes.

The following example shows two functions which take the absolute value of each number in a vector. The first, abs_loop, does not use vectorization or subsetting.
The second, abs_stes does.
rep(c(-1,1),50000) creates a vector of length 50000 with values of either -1 or 1.

The function system.time() shows us that the vectorized version is twelve times as fast!



```
> abs_loop <- function(vec){
+   for (i in 1:length(vec)) {
+     if (vec[i] < 0) {
+       vec[i] <- -vec[i]
+     }
+   }
+   vec
+ }
>
>
>
> abs_sets <- function(vec){
+   negs <- vec < 0
+   vec[negs] <- vec[negs] * -1
+   vec
+ }
>
> long <- rep(c(-1, 1), 50000)
> system.time(abs_loop(long))
   user  system elapsed
  0.012   0.000   0.012
> system.time(abs_sets(long))
   user  system elapsed
  0.001   0.000   0.001
>
```

Interacting with the file system

(slide 12)

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

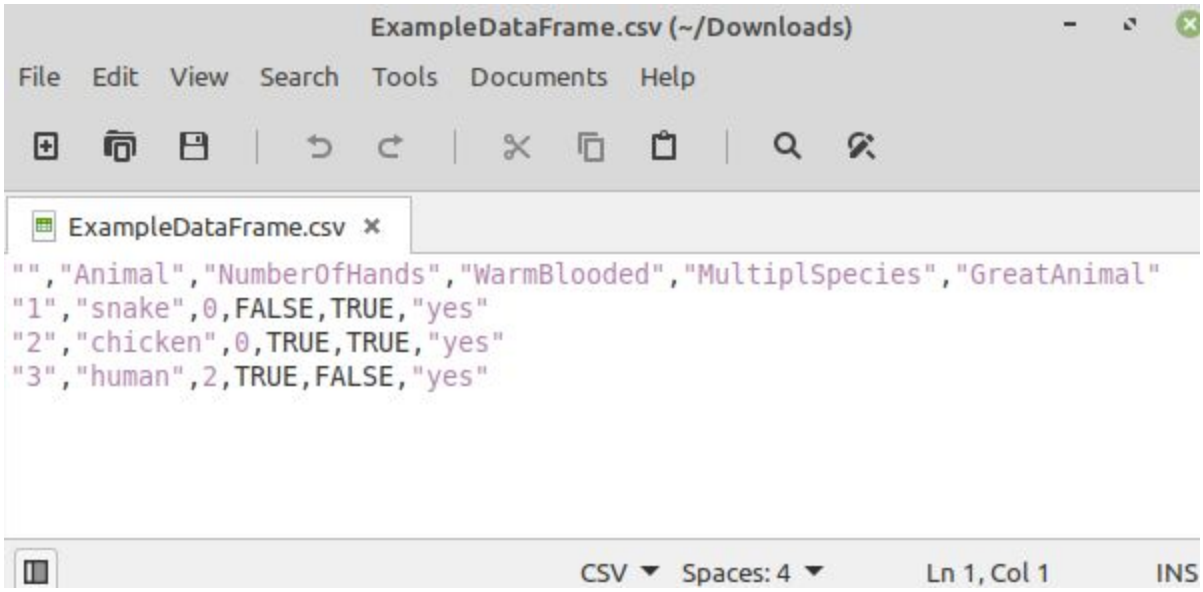
list.files() - List the files in the current directory or a specified location.

Example: list.files(“/scratch/keekov/testdir”)

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:

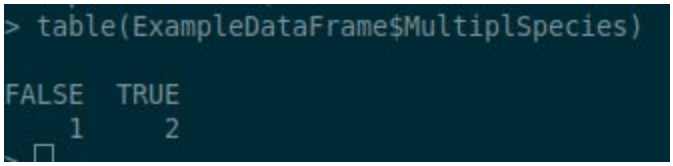


A variety of data types can be loaded. Txt Data can be loaded with read.delim() and read.csv() for csv files. Read.excel() reads excel files.
For the following examples we will use data called yf_Data.csv
On Sapelo2 it is located at /usr/local/training/RTraining/yf_Data.csv
It was also emailed to you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

This data was an experiment of infecting immune cells by yellow fever virus (YFV). Three experimental groups are baseline (0 hour), mock infection (6 hours of culture but no virus), and YFV infection (6 hours). Each group included three biological samples, and each biological sample was run as triplicates on a mass spectrometer coupled with liquid chromatography. During Mass spectrometry, the sample is run through a filter so that each particle is filtered by mass. Mz stands for the mass and time is when the particle passed the sensor. The other columns are the intensity. If we know the weight of a certain compound, we can check that row and see the intensity of the compound in each sample.

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



Subsetting and Checking Data Distribution

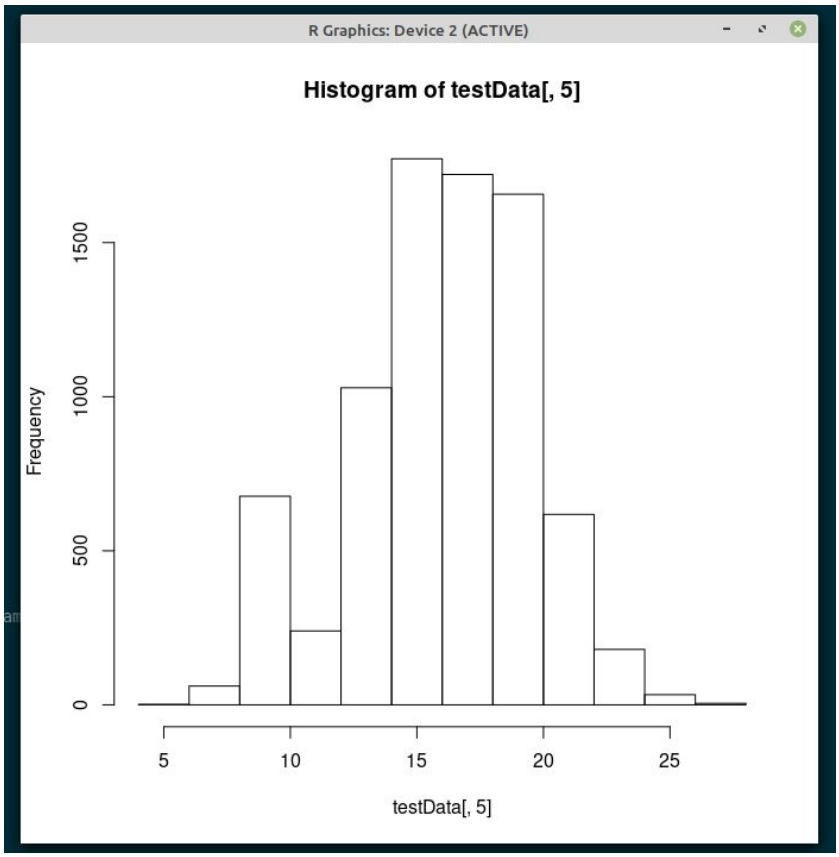
(slide 15)

We plot here the intensity distribution (histogram). The data after log transformation should approximate a normal distribution (this is why we do it). In the histogram below, there is a spike on the left, indicating many values close to 10. They were from imputation, i.e. missing values replaced by an arbitrary number, the common detection limit in the lab in this case.

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 make a histogram we will use the hist() function and the column 5 as input.
hist(yfData[,5])



Subsetting and Check data quality by average ion intensity

(slide 16)

Bad data can be from bad samples, or malfunction of an instrument. We check data quality here by checking the average ion intensity in each sample. We would like to make a new dataframe which contains the mean of each column.

We can get the mean of a column using the mean() function. If we want to apply a function to every row or column we can use the apply() function. The input to the apply function includes the dataframe, whether the function should be applied row or column wise and the name of the function.

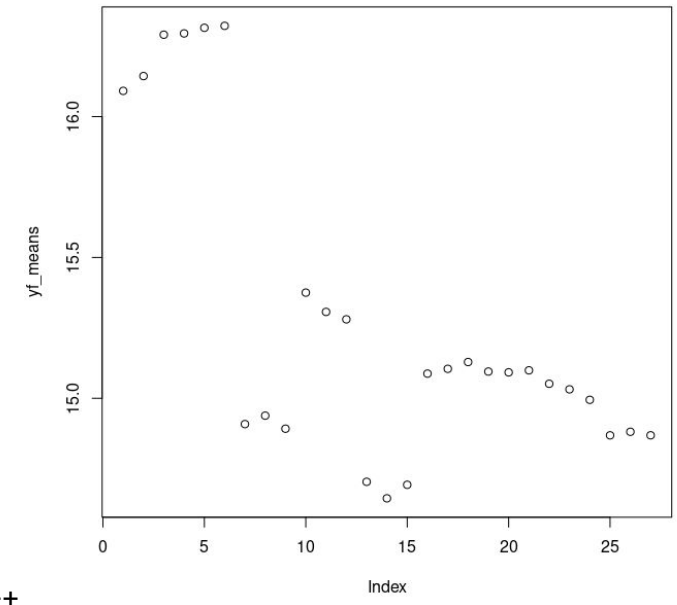
yf_colmeans = apply(yf_Data2,2,mean) 2 is used for column wise
yf_rowmeans = apply(yf_Data2,1,mean) 1 is used for row wise

In this example we want the mean for every column.

```
> yf_means = apply(yf_Data2,2,mean)
> yf_means
   p_0hr_01_1    p_0hr_01_3    p_0hr_01_5    p_0hr_02_1    p_0hr_02_3
     16.09141      16.14401      16.29092      16.29543      16.31543
   p_0hr_02_5    p_0hr_03_1    p_0hr_03_3    p_0hr_03_5 mock_6hr_01_1
     16.32235      14.90833      14.93833      14.89200      15.37507
mock_6hr_01_3 mock_6hr_01_5 mock_6hr_02_1 mock_6hr_02_3 mock_6hr_02_5
     15.30669      15.28028      14.70343      14.64476      14.69285
mock_6hr_03_1 mock_6hr_03_3 mock_6hr_03_5   yf_6hr_01_1   yf_6hr_01_3
     15.08756      15.10459      15.12874      15.09484      15.09203
  yf_6hr_01_5   yf_6hr_02_1   yf_6hr_02_3   yf_6hr_02_5   yf_6hr_03_1
     15.09919      15.05152      15.03172      14.99455      14.86840
  yf_6hr_03_3   yf_6hr_03_5
     14.88100      14.86852
>
```

yf_Data2=yf_Data[,5:ncol(yf_Data)]
yf_means = apply(yf_Data2,2,mean)

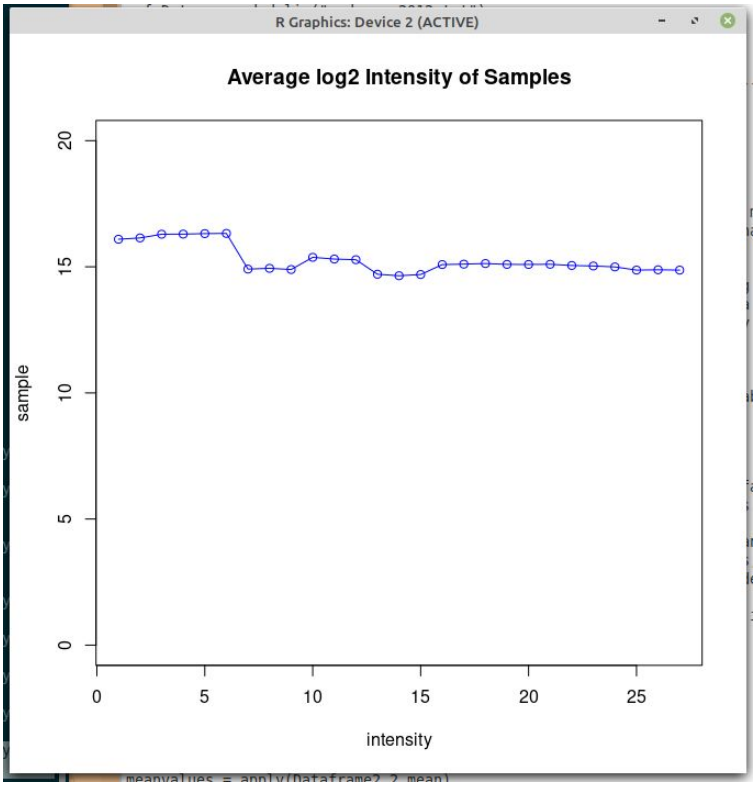
We can use the plot() function to plot this data.
plot(yf_means)



++++++++++++++++++++

The following adds some customization arguments for clarity.

plot(yf_means,ylim=c(0,20),main="Average log2 Intensity of Samples",xlab="intensity",ylab = "sample",type="o",col="blue")



****

Packages

(slide 17)

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. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.
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.

To install packages locally(not on Sapelo) use
`install.packages("Package Name")`

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.

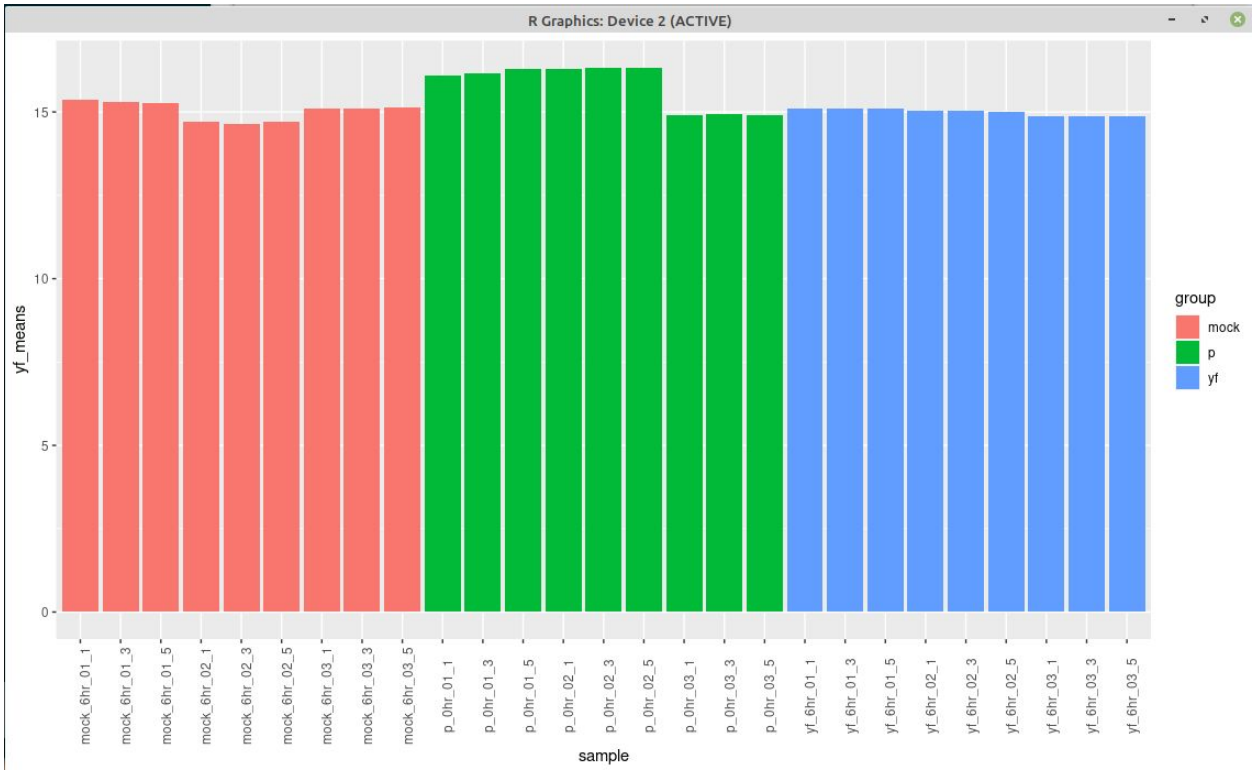
Ggplot2

(slide 18)

```
> install.packages("ggplot2")
> library(ggplot2)
> datayf_means = as.data.frame(yf_means)
```

```
> datayf_means$sample = rownames(datayf_means)
 datayf_means$group = c(rep("p",9),rep("mock",9),rep("yf",9))

> ggplot(data=datayf_means,aes(y=yf_means,x=sample,fill=group))+
+ geom_bar(stat="identity")+
+ theme(axis.text.x = element_text(angle = 90))
```



Exporting Figures and Data

(slide 19)
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”)
hist(yfData[,5])
dev.off()

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