Lesson 1 - Intro to R and R-Studio: Becoming Friends with the R Environment

Contents

A quick intro to the Introduction to R for Data Science

This 'Introduction to R for Data Science' is brought to you by the Centre for the Analysis of Genome Evolution & Function's (CAGEF) bioinformatics training initiative. This CSB1020 was developed based on feedback on the needs and interests of the Department of Cell & Systems Biology and the Department of Ecology and Evolutionary Biology.

This lesson is the first in a 6-part series. The idea is that at the end of the series, you will be able to import and manipulate your data, make exploratory plots, perform some basic statistical tests, test a regression model, and make some even prettier plots and documents to share your results.

How do we get there? Today we are going to be learning about the basic data structures in R, get cozy with the R environment, and learn how to get help when you are stuck. Because everyone gets stuck. A lot. Then you will learn how to get your data in and out of R. Next week we will learn how to tidy our data, subset and merge data and generate descriptive statistics. The next lesson will be data cleaning and string manipulation; this is really the battleground of coding - getting your data into the format where you can analyse it. After

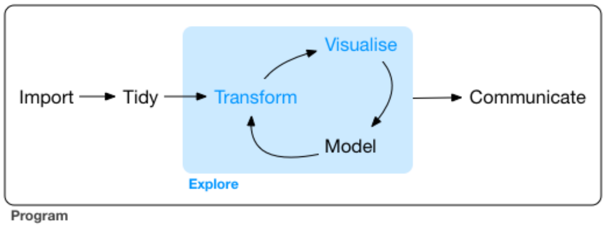


Figure 1: R for Data Science (r4ds.had.co.nz)

that, we will make all sorts of plots - from simple data exploration to interactive plots - this is always a fun lesson. And then lastly, we will learn to write some functions, which really can save you time and help scale up your analyses.

The structure of the class is a code-along style. It is hands on. The lecture AND code we are going through are available on GitHub for download at https://github.com/eacton/CAGEF, so you can spend the time coding and not taking notes. As we go along, there will be some challenge questions and multiple choice questions on Socrative. At the end of the class if you could please fill out a post-lesson survey (https://www.surveymonkey.com/r/GD3KJB9), it will help me further develop this course and would be greatly appreciated.

Highlighting

`grey background` - a package, function(), code, command or directory
italics - an important term or concept or an individual file or folder
bold - heading or a term that is being defined
blue text - named or unnamed hyperlink

Data Files Used in This Lesson

Lesson files can be downloaded at https://github.com/eacton/CAGEF/tree/master/Lesson_1/data. Right-click on the filename and select 'Save Link As…' to save the file locally. The files should be saved in the same folder you plan on using for your R script for this lesson.

Or click on the blue hyperlink at the start of the README.md at https://github.com/eacton/CAGEF/tree/master/Lesson_1 to download the entire folder at DownGit.

Objective: At the end of this session you will be familiar with the R environment, setting your working directory, know about basic data structures in R and how to create them. You will be able to install and load packages.

Installing R and RStudio

Installing R:
Windows:
- Go to http://cran.utstat.utoronto.ca/
- Click on 'Download R for Windows'
- Click on 'install R for the first time'

- Click on 'Download R 3.4.4 for Windows' (if there is a newer version that is a-okay)
- Double-click on the .exe file once it has downloaded (in your Downloads folder if you haven't specified elsewhere) and follow the instructions.

(Mac) OS X:
- Go to http://cran.utstat.utoronto.ca/
- Click on 'Download R for (Mac) OS X'
- Click on R-3.4.4.pkg (if there is a newer version that is a-okay)
- Open the .pkg file once it has downloaded (in your Downloads folder if you haven't specified elsewhere) and follow the instructions.

Linux:
- sudo apt-get update
- sudo apt-get install r-base
- sudo apt-get install r-base-dev (so you can compile packages from source)

Installing RStudio:
Windows:
- Go to https://www.rstudio.com/products/rstudio/download/#download
- Click on 'RStudio 1.1.442 - Windows Vista/7/8/10' to download the installer (if there is a newer version that is a-okay)
- Double-click on the .exe file once it has downloaded (in your Downloads folder if you haven't specified elsewhere) and follow the instructions.

(Mac) OS X:
- Go to https://www.rstudio.com/products/rstudio/download/#download
- Click on 'RStudio 1.1.442 - Mac OS X 10.6+ (64-bit)' to download the installer (if there is a newer version that is a-okay)
- Double-click on the .dmg file once it has downloaded (in your Downloads folder if you haven't specified elsewhere) and follow the instructions.

Linux:
- Go to https://www.rstudio.com/products/rstudio/download/#download
- Click on the installer that describes your system ie. 'RStudio 1.1.442 - Ubuntu 16.04+/Debian 9+ (64-bit)' (if there is a newer version that is a-okay)
- Double-click on the .deb file once it has downloaded and follow the instructions.
- If double-clicking on your .deb file did not open the software manager open the terminal (this can be found by searching for Terminal) and type sudo dpkg -i Downloads/rstudio-xenial-1.1.442-amd64.deb
Note: You have 3 things that could change in this last command.
1. This assumes you have just opened the terminal and are in your home directory. (If not, you have to modify your path. You can get to your home directory by typing cd ~.)
2. This assumes you have downloaded the .deb file to Downloads. (If you downloaded the file somewhere else, you have to change the path to the file, or download the .deb file to Downloads).
3. This assumes your filename for .deb is the same as above. (Put the name matching the .deb file you downloaded).

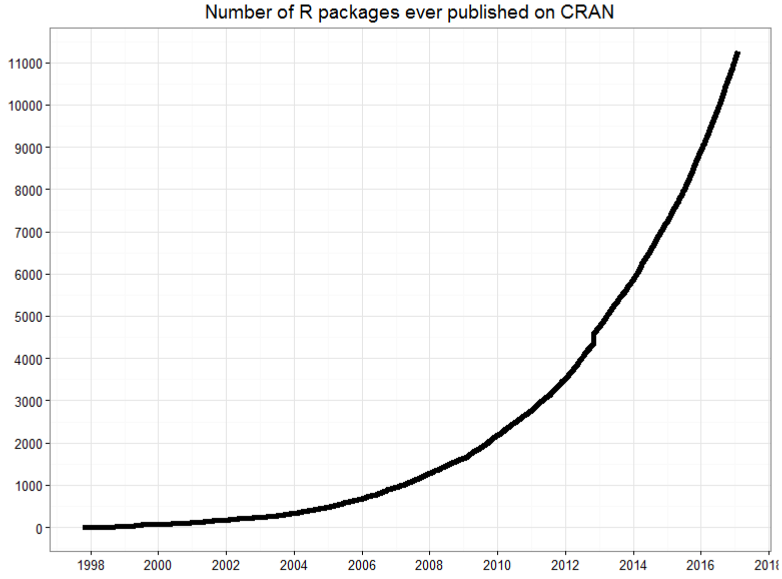
If you have a problem with installing R or RStudio, please come 15 minutes before the lesson starts and I will help you troubleshoot your installation. You can also try to solve the problem yourself by Googling any error messages you get.

A quick intro to the R environment

R is a language and an **environment** because it has the tools and software for the storage, manipulation, statistical analysis, and graphical display for of data. It comes with about 25 built-in 'packages' and uses a simple programming language ("S"). https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

So… what are in these packages? A **package** is a collection of functions, sometimes data, and compiled code. **Functions** are the basic workhorses of R; they are the tools we use to analyze our data. Each function can be thought of as a unit that has a specific task. A function takes an input, evaluates an expression (ie. a calculation, plot, merge, etc.), and returns a value. (We will have an entire separate lesson to explore functions.)

Users have been encouraged to make there own packages… there are now over 12,000 packages on CRAN and about 1,500 on Bioconductor.



The "Comprehensive R Archive Network" (CRAN) is a collection of sites that have the same R and related R material (ie. new and previous versions of R software, documentation, packages and collections of R packages to download that might be useful in a particular field (CRAN Task Views), links to the R journal and R search sites, bug reports and fixes). Different sites (for example, we used http://cran.utstat.utoronto.ca/), are called a *mirrors* because they reflect the content from the master site in Austria. There are mirrors worldwide to reduce the burden on the network. CRAN will be referred to here as a main repository for obtaining R packages.

Bioconductor is another repository for R packages (https://www.bioconductor.org/), but it specializes in tools for high-throughput genomics data. One nice thing about Bioconductor is that it has decent vignettes; a **vignette** is the set of documentation for a package explaining its functions and usages in a tutorial-like format.

RStudio is an IDE (Integrated Development Environment) for R that provides a more user-friendly experience than using R in a terminal setting. It has 4 main areas or panes, which you can customize to some extent under `Tools > Global Options > Pane Layout`:

1. **Source** - The code you are annotating and keeping in your script.
2. **Console** - Where your code is executed.
3. **Environment** - What global objects you have created and functions you have written/sourced.
   History - A record of all the code you have executed in the console.
   Connections - Which data sources you are connecting to. (Not being used in this course.)
4. **Files, Plots, Packages, Help, Viewer** - self-explanatoryish if you click on their tabs.

All of the panes can be minimized or maximized using the large and small box outlines in the top right of each pane.

Source

The **Source** is where you are keeping the code and annotation that you want to be saved as your script. The tab at the top left of the pane has your script name (ie. 'Untitled.R'), and you can switch between scripts by toggling the tabs. You can save, search or publish your source code using the buttons along the pane header. Code in the Source pane is *run* or *executed* automatically.

To run your current line of code or a highlighted segment of code from the Source pane you can:
a) click the button `'Run' -> 'Run Selected Line(s)'`,
b) click `'Code' -> 'Run Selected Line(s)'` from the menu bar,
c) use the keyboard shortcut `CTRL + ENTER` (Windows & Linux) `Command + ENTER` (Mac) (recommended),
d) copy and paste your code into the Console and hit `Enter` (not recommended).

There are always many ways to do things in R, but the fastest way will always be the option that keeps your hands on the keyboard.

Console

You can also type and execute your code (by hitting `ENTER`) in the **Console** when the `>` prompt is visible. If you enter code and you see a `+` instead of a prompt, R doesn't think you are finished entering code (ie. you might be missing a bracket). If this isn't immediately fixable, you can hit `Esc` twice to get back to your prompt. Using the up and down arrow keys, you can find previous commands in the Console if you want to rerun code or fix an error resulting from a typo.

On the Console tab in the top left of that pane is your current working directory. Pressing the arrow next to your working directory will open your current folder in the Files pane. If you find your Console is getting too cluttered, selecting the broom icon in that pane will clear it for you. The Console also shows information: upon startup about R (such as version number), during the installation of packages, when there are warnings, and when there are errors.

Environment

In the **Global Environment** you can see all of the stored objects you have created or sourced (imported from another script). The Global Environment can become cluttered, so it also has a broom button to clear its workspace.

Objects are made by using the assignment operator `<-`. On the left side of the arrow, you have the name of your object. On the right side you have what you are assigning to that object. In this sense, you can think of an object as a container. The container holds the values given as well as information about 'class' and 'methods' (which we will come back to).

Type `x <- c(2,4)` in the Console followed by `Enter`. 1D objects' data types can be seen immediately as well as their first few values. Now type `y <- data.frame(numbers = c(1,2,3), letters = c("a","b","c"))` in the Console followed by `Enter`. You can immediately see the dimension of 2D objects, and you can check the structure of data frames and lists (more later) by clicking on the object's arrow. Clicking on the object name will open the object to view in a new tab. Custom functions created in session or sourced will also appear in this pane.

The Environment pane dropdown displays all of the currently loaded packages in addition to the Global Environment. *Loaded* means that all of the tools/functions in the package are available for use. R comes with a number of packages pre-loaded (ie. base, grDevices).

In the History tab are all of the commands you have executed in the Console during your session. You can select a line of code and send it to the Source or Console.

The Connections tab is to connect to data sources such as Spark and will not be used in this lesson.

Files, Plots, Packages, Help, Viewer

The Files tab allows you to search through directories; you can go to or set your working directory by making the appropriate selection under the `More` (blue gear) drop-down menu. The `...` to the top left of the pane allows you to search for a folder in a more traditional manner.

The Plots tab is where plots you make in a .R script will appear (notebooks and markdown plots will be shown in the Source pane). There is the option to Export and save these plots manually.

The Packages tab has all of the packages that are installed and their versions, and buttons to Install or Update packages. A checkmark in the box next to the package means that the package is loaded. You can load a package by adding a checkmark next to a package, however it is good practice to instead load the package in your script to aid in reproducibility.

The Help menu has the documentation for all packages and functions. For each function you will find a description of what the function does, the arguments it takes, what the function does to the inputs (details), what it outputs, and an example. Some of the help documentation is difficult to read or less than comprehensive, in which case googling the function is a good idea.

The Viewer will display vignettes, or local web content such as a Shiny app, interactive graphs, or a rendered html document.

Global Options

I suggest you take a look at `Tools -> Global Options` to customize your experience.

For example, under `Code -> Editing` I have selected `Soft-wrap R source files` followed by `Apply` so that my text will wrap by itself when I am typing and not create a long line of text.

You may also want to change the `Appearance` of your code. I like the `RStudio theme: Modern` and `Editor font: Ubuntu Mono`, but pick whatever you like! Again, you need to hit `Apply` to make changes.

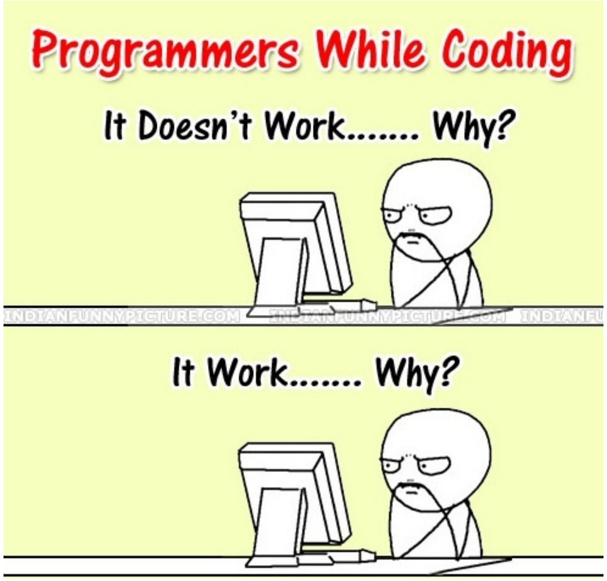
That whirlwind tour isn't everything the IDE can do, but it is enough to get started.

Making Life Easier

Annotate your code

Why???

-Can you rerun this analysis and but change X parameter? (said every PI ever)
-Can you make this plot, but with dashed lines, a different axis, with error bars? (said every PI ever)
-Can I borrow your code? (a collaborator or officemate)
-Your worst collaborator is actually you in 6-months. Do you remember what you had for breakfast last Tuesday?



You can annotate your code for selfish reasons, or altruistic reasons, but annotate your code.

How do I start?

- A hash-tag `#` will comment your text. A hash-tag in front of code means that code will not be evaluated.
- Put a description of what you are doing near your code (I prefer above) - at every process, decision point, or non-default argument in a function. For example, why you selected k=6, or the Spearman over Pearson option for your correlation matrix, or quantile over median normalization, or why you made the decision to filter out certain samples.
- Break your code into sections to make it readable.
- Give your objects informative object names **that are not the same as function names**.

Keyboard shortcuts:

Comment/Uncomment lines `CTRL + SHIFT + C` (Windows, Linux) / `Command + SHIFT + C` (Mac)
Reflow Comment (Wrap comments) `CTRL + SHIFT + /` (Windows, Linux) / `Command + SHIFT + /` (Mac)

Best Practices for Writing Scripts

```
Start each script with a description of what it does.

Then load all required packages.

Consider what working directory you are in when sourcing a script.

Use comments to mark off sections of code.

Put function definitions at the top of your file, or in a separate file if there are many.

Name and style code consistently.

Break code into small, discrete pieces.

Factor out common operations rather than repeating them.
```

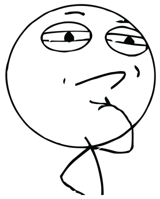


Figure 2:

```
Keep all of the source files for a project in one directory and use relative paths to access them.

Keep track of the memory used by your program.

Always start with a clean environment instead of saving the workspace.

Keep track of session information in your project folder.

Have someone else review your code.

Use version control.
```

https://swcarpentry.github.io/r-novice-inflammation/06-best-practices-R/

Trouble-shooting basics

- *file does not exist* - working directory errors - use `getwd()` to check where you are working, the Files window to check that your file exists there, and `setwd()` to change your directory - work inside an *R project* with your files in the same folder - your working directory will be set automatically when you open the project (this can be done by using `File -> New Project` and following prompts)
- *typos* - R is case sensitive, check that you've spelled everything right - use *tab-completion* when possible
- *open quotes, parentheses, brackets* - R Studio highlights matching brackets, and gives an 'x' on your left sidebar if your final bracket is not closed
- *data type* - you can't perform a calculation on those 1, and 0's because they are actually characters and not numeric
- *unexpected answers* - access the *help menu* by typing `help("function")` or `?function` or `help(package = "package")` in the Source or Console to double-check how the function works and its output. The result will be in the lower-right pane under the Help tab (which is also searchable).
- *function is not found* - make sure the package is installed AND loaded
- *weird error* you are not sure the meaning of - find answers online (see below)
- sometimes weird things happen - clear your environment and restart your R session
- *the R bomb*!! - congrats, you have completed an R rite of passage.

Beginner Advice

Try to solve a problem yourself, but set a 30 minute cutoff on being stuck. At this level, many people have had and solved your problem. Beginners get frustrated because they get stuck and take hours to solve a problem themselves. Set your limit, stay within it, then go online and get help.

Finding answers online

- 99% of the time, someone has already asked your question

Figure 3:

- Google, Stackoverflow, SEQanswers, quora, researchgate, RSeek, twitter, even reddit

- Including the program, version, error, package and function helps, be specific

- http://stat545.com/help-general.html

Asking a question

- Summarize your question in the title.
- Introduce your question, how you ran into the problem, **and how you tried to solve it yourself**. If you haven't done the bolded thing, do the bolded thing.
- Show enough of your code to reproduce the problem.
- Add tags that match your problem.
- Respond to the feedback. People put in their free time to answer you.
- https://stackoverflow.com/help/how-to-ask

- https://www.r-project.org/posting-guide.html

Remember - everyone looks for help online ALL THE TIME. It is completely normal. Also, with programming there are multiple ways to come up with an answer. There are different packages that let you do the same thing, but with shortcuts. There are different levels of 'efficiency' and 'redundancy' in coding. You will work on refining these aspects of your code as you go along in this course, and in your coding career.

Last but not least, to make life easier: under Help there is a *Cheatsheet of Keyboard Shortcuts* or a browser list here.

A quick note on directory structure

This is an image of a possible directory.

In this hierarchy we will pretend to be *benedict*, and we are hanging out in our *Tables* folder. R looks to read in your files from your **working directory**, which in this case would be *Tables*, so at this moment, R would have access to *proof.tsv* and *genes.csv*. If I tried to open *paper.txt* under *benedict*, R would tell me there is no such file in my current working directory.

To get your working directory in R you would type in your Source pane:

```
getwd()
```

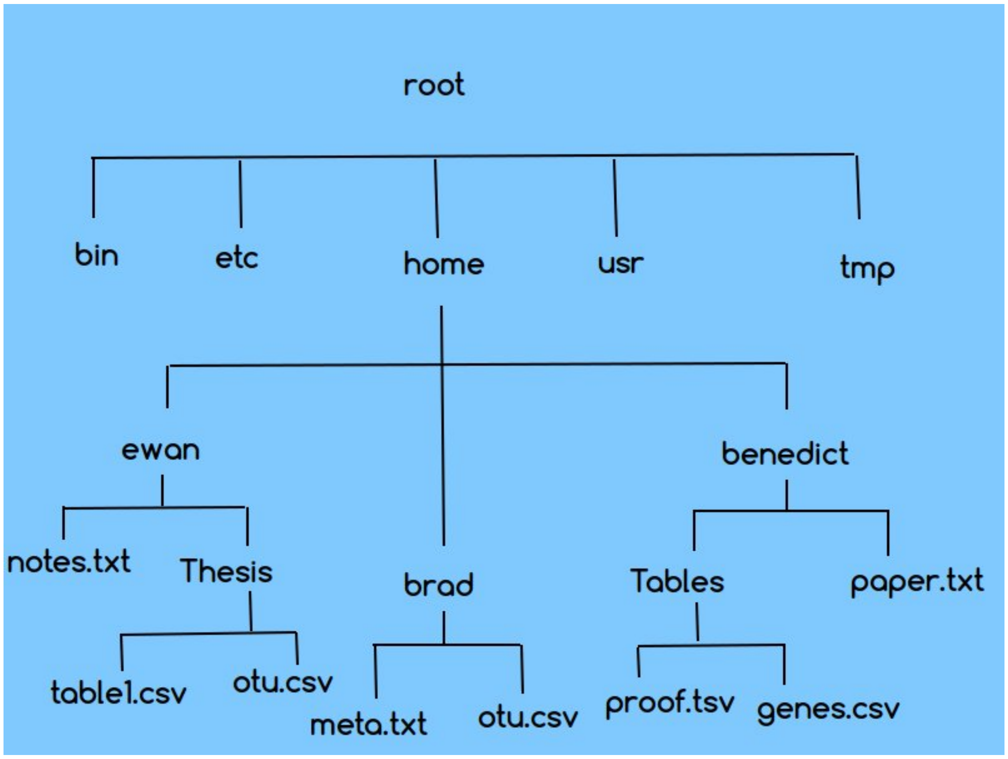


Figure 4:

You would then press `Enter` to execute the code in the Console. Technically, there is no reason to save this code and it could be typed and executed in the Console. However, to save some mental effort choosing which code to save or not (especially when starting out), let's type everything in the Source. You can always go back and choose what to delete, and harder to recreate what you didn't save.

The output in your Console would be:

```
[1] "/home/benedict/Tables"
```

R will always tell you your **absolute directory**. An absolute directory is a *path* starting from your root `"/"`. The absolute directory can vary from computer to computer. My home directory and your home directory are not the same; our names differ in the path.

To move directories, it is good to know a couple shortcuts. `'.'` is your current directory. `'..'` is up one directory level. `'~'` is your home directory (a shortcut for `"/home/benedict"`). Therefore, our current location could also be denoted as `"~/Tables"`.

To move to the directory *ewan* we use a function that will set the working directory:

```
setwd("/home/ewan")
#or
setwd("../../ewan")
```

A **relative directory** is a path starting from whereever your currently are (your working directory). This path could be the same on your computer and my computer if and only if we have the same directory structure.

If I wanted to move back to *Tables* using the absolute *path*, I would set a new working directory:

```
setwd("/home/benedict/Tables")
#or
```

```
setwd("~/Tables")
```

And the *relative path* would be:

```
setwd("../benedict/Tables")
```

There is some talk over setting working directories within scripts. Obviously, not everyone has the same absolute path, so if must set a directory in your script, *it is best to have a relative path starting from the folder your script is in.* Keep in mind that others you share your script with might not have the same directory structure if you refer to subfolders.

You can set your working directory by:

1. `setwd()`

2. `Session -> Set Working Directory` (3 Options)

3. `Files Window -> More (Gear Symbol) -> Set As Working Directory`

Challenge Question: Socrative

A quick intro to R data types and data structures

What do I mean by 'types' (flavours) of data?

- character - a#c$E
- numeric - 7.5
- integer - 1
- logical - TRUE, FALSE

There are 5 types of data structures in R.

1. vectors - 1D - holds one type of data
2. lists - 1D - holds multiple data types
3. matrices - 2D - holds one type of data
4. data frames - 2D - holds multiple data types
5. arrays - nD - holds one type of data

Let's make some data!!

The assignment operator `"<-"` assigns a value to an object. The equals sign `'='` is also an assignment operator. However, since equals signs are used in logical statements (==, >=, <=, !=) with a different meaning, I prefer to use `<-`. You will see both outside this course.

```
object_name <- value
#or equivalently
object_name = value
```

(This is for illustrative purposes only and is not code.)

A quick word on functions.

We are going to use a simple function to create a vector. Functions take arguments. To find out which arguments they take, we can look at the help menu. The help menus in R can be intimidating and hard to read. Don't think it is just because you are new. If you really don't understand the help menu, you are probably not the only one, so look online and there will likely be a simpler explanation.

Let's look up the function `c()`, which combines values into a vector or list. It takes in the argument `'...'`, which is defined as the objects to be concatenated. In the description it says 'all arguments are coerced to a common type, which is the type of the returned value' and in the details it tells you the hierarchy of coercion: *NULL < raw < logical < integer < double < complex < character < list < expression*. It also says 'factors are treated only via their internal integer codes'. Now we don't know what all this means yet, but we come back to this as we go along. The output of the function is 'NULL or an expression or a vector'. At the bottom of the help menu, there are usually examples of how to use the function, which are sometimes easier to understand than the documentation.

Vectors

Here are what vectors of each data 'type' would look like. Note that character items must be in quotations. 'L' is placed next to a number to specify an integer.

```
vec_char <- c("bacteria", "virus", "archaea")
#this is equivalent to
vec_char <- c('bacteria', 'virus', 'archaea')
vec_char
```

```
## [1] "bacteria" "virus"    "archaea"
```

```
vec_num <- c(1:10)
vec_num
```

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

```
vec_log <- c(TRUE, FALSE, TRUE)
vec_log
```

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

```
vec_int <- c(1L, 8L)
vec_int
```

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

What happens if we try to include more than one type of data?

```
vec_mixed <- c("bacteria", 1, TRUE)
vec_mixed
```

```
## [1] "bacteria" "1"        "TRUE"
```

R will *coerce* your vector to be of one data type, in this case the type that is most inclusive is a character vector.

Challenge Question: Socrative x 2

I am highlighting this for a couple of reasons. Keep your data types in mind. It is good practice to look at your object or the global environment to make sure the object that you just made is what you think it is. Secondly, it can be useful for data analysis to be able to switch from TRUE/FALSE to 1/0, and it is pretty easy, as we have just seen.

You can name the contents of your vectors or specify them upon vector creation.

```
names(vec_log) <- c("male", "elderly", "heart attack")
#is equivalent to
```

```
vec_log <- c("male" = TRUE, "elderly" = FALSE, "heart attack" = TRUE)
vec_log
```

```
##         male      elderly heart attack
##         TRUE        FALSE         TRUE
```

The number of elements in a vector is its length.

```
length(vec_char)
```

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

```
length(vec_num)
```

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

```
length(vec_log)
```

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

You can grab a specific element by its index, or by its name.

```
vec_char[3]
```

```
## [1] "archaea"
```

```
vec_char[2:3]
```

```
## [1] "virus"   "archaea"
```

```
#second and third element in the vector inclusive
#(this is not the same for all programming languages)

#you can use negative indexing to select 'everything but'
vec_char[-1]
```

```
## [1] "virus"   "archaea"
```

```
vec_log["male"]
```

```
## male
## TRUE
```

Lists

Lists can hold mixed data types of different lengths.

```
list_mix <- list(character = c('bacteria', 'virus', 'archaea'),
                 num = c(1:10),
                 log = c(TRUE, FALSE, TRUE))

#formatting - equivalent, but less reader friendly for longer lists
list_mix = list(character = c('bacteria', 'virus', 'archaea'), num = c(1:10), log = c(TRUE, FALSE, TRUE
list_mix
```

```
## $character
## [1] "bacteria" "virus"    "archaea"
##
## $num
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
##
## $log
## [1]  TRUE FALSE  TRUE
```

Lists can get complicated. If you forget what is in your list, use the `str()` function to check out its structure. It will tell you the number of items in your list and their data types. You can (and should) call `str()` on any R object. You can also try it on one of our vectors.

```
str(list_mix)
```

```
## List of 3
##  $ character: chr [1:3] "bacteria" "virus" "archaea"
##  $ num      : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ log      : logi [1:3] TRUE FALSE TRUE
```

```
str(vec_mixed)
```

```
##  chr [1:3] "bacteria" "1" "TRUE"
```

To subset for 'virus', I first have to subset for the character element of the list. Kind of like a Russian nested doll or a present, where you have to open the outer layer to get to the next.

```
list_mix[[1]]
```

```
## [1] "bacteria" "virus"    "archaea"
```

```
## list_mix$character

## list_mix[[1]][2]
## list_mix$character[2]
```

Matrices

Create a demo matrix.

```
mat <- matrix(c(rep(0, 10), rep(1,10)), nrow = 5, ncol = 5)
#equivalent to
mat <- matrix(c(rep(0:1, each = 10, times = 2)), nrow = 5, ncol = 5)
mat
```

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

What has happened here? Look up the `rep()` function. Why has R not thrown an error? How would I make this same matrix without vector recycling? Can you think of 2 ways?

Challenge

Make a 4 x 4 matrix that looks like this, using the `seq()` function at least once.

A matrix is a 2D object. We can now check out a couple more properties - like the number of rows and columns.

```
str(mat)
```

```
##  num [1:5, 1:5] 0 0 0 0 0 0 0 0 0 0 ...
```

```
nrow(mat)
```

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

```
ncol(mat)
```

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

```
dim(mat)
```

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

```
length(mat)
```

```
## [1] 25
```

To access a specific row or column we can still use indexing.

```
mat[3:5,]
```

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

```
mat[, 4]
```

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

Note that when we are subsetting a single row or column, we end up with a vector.

```
is.vector(mat[,4])
```

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

It is common to transform matrices. Note that the set of ones will now be in rows rather than columns.

```
t(mat)
```

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

Now that we have had the opportunity to create a few different objects, let's talk about what an object *class* is. An object class can be thought of as how an object will behave in a function. Because of this data frames, lists and matrices have their own classes, while vectors inherit from their data type (vectors of characters behave like characters, vectors of numbers behave like numbers).

```
class(vec_char)
```

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

```
class(vec_num)
```

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

```
class(mat)
```

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

```
class(list_mix)
```

```
## [1] "list"
```

Some R package developers have created their own object classes. We won't deal with this today, but it is good to be aware of from a trouble-shooting perspective that your data may need to be formatted to fit a certain class of object when using different packages.

Data Frames

Data frames are lists to the extent that they can hold different types of data. However, they must be of equal length.

```
dat <- data.frame(character = c('bacteria', 'virus', 'archaea'),
                  num = c(1:10),
                  log = c(TRUE, FALSE, TRUE))
```

```
## Error in data.frame(character = c("bacteria", "virus", "archaea"), num = c(1:10), : arguments imply
```

```
dat <- data.frame(character = c('bacteria', 'virus', 'archaea'),
                  num = c(1:3),
                  log = c(TRUE, FALSE, TRUE))
dat
```

```
##   character num   log
## 1  bacteria   1  TRUE
## 2     virus   2 FALSE
## 3   archaea   3  TRUE
```

Many R packages have been made to work with data in data frames, and this is the class of object where we will spend most of our time.

Let's use some of the functions we have learned for finding out about the structure of our data frame.

```
str(dat)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ character: Factor w/ 3 levels "archaea","bacteria",..: 2 3 1
##  $ num      : int  1 2 3
##  $ log      : logi  TRUE FALSE TRUE
```

What is a *factor*?

A factor is a *class* of object used to encode a character vector into categories. This will become clear with a bit more data, so lets make our data frame larger by adding rows. We can only do this if the data we want to add has the same number of columns. How many rows and columns does this new data frame have?

```r
dat_large <-  rbind(dat, dat, dat)

nrow(dat_large)
```

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

```r
ncol(dat_large)
```

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

```r
dim(dat_large)
```

```
## [1] 9 3
```

If we look at the structure again, we still have 3 levels. This is because each unique character element has been encoded as a number. (Note that a column can be subset by index or by its name using the `'$'` operator.)

```r
dat_large$character
```

```
## [1] bacteria virus    archaea  bacteria virus    archaea  bacteria virus   
## [9] archaea 
## Levels: archaea bacteria virus
```

```r
#equivalent to
dat_large[ ,1]
```

```
## [1] bacteria virus    archaea  bacteria virus    archaea  bacteria virus   
## [9] archaea 
## Levels: archaea bacteria virus
```

```r
levels(dat_large$character)
```

```
## [1] "archaea"  "bacteria" "virus"
```

Note that the first character object in the data frame is 'bacteria', however, the first factor level is archaea. R by default puts factor levels in *alphabetical order*. This can cause problems if we aren't aware of it. Always check to make sure your factor levels are what you expect. With factors, we can deal with our character levels directly, or their numeric equivalents. Factors are extremely useful for performing group calculations as we will see later in the course.

```r
as.numeric(dat_large$character)
```

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

Challenge

Look up the factor function. Use it to make 'bacteria' the first level, 'virus' the second level, and 'archaea' the third level for the data frame 'dat'. Bonus if you can make the level numbers match (1,2,3 instead of 2,3,1). Use functions from the lesson to make sure your answer is correct.

We can also convert between data types if they are similar enough. For example, I can convert my matrix into a data frame. Since a data frame can hold any type of data, it can hold all of the numeric data in a matrix.

```r
new_dat <- as.data.frame(mat)
new_dat
```

```
##   V1 V2 V3 V4 V5
## 1  0  0  1  1  0
## 2  0  0  1  1  0
## 3  0  0  1  1  0
## 4  0  0  1  1  0
## 5  0  0  1  1  0
```

Note that R just made up column names for us. We can provide our own vector of column names.

```
colnames(new_dat) <- c("col1", "col2", "col3", "col4", "col5")
#equivalent to
colnames(new_dat) <- c(paste0(rep("col",5), 1:5))
new_dat
```

```
##   col1 col2 col3 col4 col5
## 1    0    0    1    1    0
## 2    0    0    1    1    0
## 3    0    0    1    1    0
## 4    0    0    1    1    0
## 5    0    0    1    1    0
```

In contrast, our data frame with multiple data types can not be converted into a matrix, as a matrix can only hold one data type. We could however, transform our new_dat back into a matrix. The matrix will retain our column heading.

```
new_mat <- as.matrix(new_dat)
new_mat
```

```
##      col1 col2 col3 col4 col5
## [1,]    0    0    1    1    0
## [2,]    0    0    1    1    0
## [3,]    0    0    1    1    0
## [4,]    0    0    1    1    0
## [5,]    0    0    1    1    0
```

Arrays

Arrays are n dimensional objects that hold numeric data. To create an array, we give a vector of data to fill the array, and then the dimensions of the array. This code will recycle the vector 1:10 and fill 5 arrays that have 2 x 3 dimensions. To visualize the array, we will print it afterwards.

```
arr <- array(data = 1:10, dim = c(2,3,5))
arr
```

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

This arrangement makes it more clear how we would subset the number 7 out of array 5.

```
arr[1, 2, 5]
```

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

A 2D array is just a matrix. Unless you specify a 3rd dimension.

```
mat_arr <- array(data = 1:10, dim = c(2,3))
mat_arr
```

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

```
#equivalent to
arr_mat <- array(data = 1:10, dim = c(2,3,1))
arr_mat
```

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

```
all.equal(mat_arr, arr_mat)
```

```
## [1] "Attributes: < Component \"dim\": Numeric: lengths (2, 3) differ >"
## [2] "target is matrix, current is array"
```

```
class(mat_arr)
```

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

```
class(arr_mat)
```

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

Personally, I don't use arrays in my daily genomics life, so if you find this confusing, I wouldn't worry about it too much.

Using R for a Calculator

So you can do math…

Addition

```
3 + 4
```

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

Subtraction

```
3 - 4
```

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

Multiplication

```
3 * 4
```

```
## [1] 12
```

Division

```
3 / 4
```

```
## [1] 0.75
```

Exponents

```
3^4
```

```
## [1] 81
```

A logic test ensues.

```
x <- 7
y <- x +3
y
```

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

If x gets updated, what happens to y? This is something you need to be aware of when running code - variables dependent on other variables and where in your program they are created and updated.

```
x <- 8
y
```

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

```
y <- x +3
y
```

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

So you can do math... on a vector.

```
vec_num * 4
```

```
##  [1]  4  8 12 16 20 24 28 32 36 40
```

So you can do math... on a vector.

```
vec_log * 4 #uh oh - that is probably not what you want...
```

```
##         male      elderly heart attack
##            4            0            4
```

So you can do math... on a list.

```
list_mix * 4
```

```
## Error in list_mix * 4: non-numeric argument to binary operator
```

```
list_mix[[2]] * 4
```

```
##  [1]  4  8 12 16 20 24 28 32 36 40
```

```
list_mix[[2]][2:4] * 4
```

```
## [1]  8 12 16
```

So you can do math… on a matrix.

```
mat * 4
```

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

So you can do math… on a data frame. Non-numeric (character) data will throw an error, while factors will give a warning message and populate the offending column with NAs. Logical data will be converted to numeric values and multiplied. Therefore be careful to specify your numeric data for mathematical operations.

```
dat * 4
```

```
## Warning in Ops.factor(left, right): '*' not meaningful for factors
```

```
##   character num log
## 1        NA   4   4
## 2        NA   8   0
## 3        NA  12   4
```

```
dat$num * 4
```

```
## [1]  4  8 12
```

```
dat[ , 2] * 4
```

```
## [1]  4  8 12
```

So you can do math… on an array.

```
arr * 4
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    4   12   20
## [2,]    8   16   24
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   28   36    4
## [2,]   32   40    8
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   12   20   28
## [2,]   16   24   32
```

```
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   36    4   12
## [2,]   40    8   16
##
## , , 5
##
##      [,1] [,2] [,3]
## [1,]   20   28   36
## [2,]   24   32   40
```

```
arr[1, 2, 5] * 4
```

```
## [1] 28
```

These are illustrative examples to see how our different data structures behave. In reality, you will want to do calculations across rows and columns, and not on your entire matrix or data frame. For example, we might have a count table where rows are genes, columns are samples, and we want to know the sum of all the counts for a gene. To do this, we can use the `apply()` function. `apply()` takes an array, matrix (or something that can be coerced to such, like a numeric data frame), and applies a function over row (`MARGIN = 1`) or columns (`MARGIN = 2`). Here we can invoke the `sum` function.

```
counts <- data.frame(Site1 = c(geneA = 2, geneB = 4, geneC = 12, geneD = 8),
                     Site2 = c(geneA = 15, geneB = 18, geneC = 27, geneD = 28),
                     Site3 = c(geneA = 10, geneB = 7, geneC = 13, geneD = 15))

counts
```

```
##       Site1 Site2 Site3
## geneA     2    15    10
## geneB     4    18     7
## geneC    12    27    13
## geneD     8    28    15
```

```
apply(counts, MARGIN = 1, sum)
```

```
## geneA geneB geneC geneD
##    27    29    52    51
```

```
str(apply(counts, MARGIN = 1, sum))
```

```
##  Named num [1:4] 27 29 52 51
##  - attr(*, "names")= chr [1:4] "geneA" "geneB" "geneC" "geneD"
```

```
class(apply(counts, MARGIN = 1, sum))
```

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

Note that the output is no longer a data frame. Since the resulting sums would have the dimensions of a 1x4 matrix, the results are instead coerced to a named numeric vector. The apply function will recognize basic functions.

```
apply(counts, MARGIN = 1, mean)
```

```
##     geneA     geneB     geneC     geneD
##  9.000000  9.666667 17.333333 17.000000
```

```
apply(counts, MARGIN = 1, sd)
```

```
##     geneA     geneB     geneC     geneD
##  6.557439  7.371115  8.386497 10.148892
```

```
apply(counts, MARGIN = 1, median)
```

```
## geneA geneB geneC geneD
##    10     7    13    15
```

```
apply(counts, MARGIN = 1, quantile)
```

```
##      geneA geneB geneC geneD
## 0%     2.0   4.0  12.0   8.0
## 25%    6.0   5.5  12.5  11.5
## 50%   10.0   7.0  13.0  15.0
## 75%   12.5  12.5  20.0  21.5
## 100%  15.0  18.0  27.0  28.0
```

When all data values are transformed, the output is a numeric matrix.

What if I want to know something else? We can create a function. The sum function we called before can also be written as a function taking in x (in this case the vector of values from our coerced data frame row by row) and summing them. Other functions can be passed to `apply()` in this way.

```
apply(counts, MARGIN = 1, sum)
```

```
## geneA geneB geneC geneD
##    27    29    52    51
```

```
#equivalent to
apply(counts, MARGIN = 1, function(x) sum(x))
```

```
## geneA geneB geneC geneD
##    27    29    52    51
```

Challenge

Use the `apply()` function to multiply the counts for each gene by 3.

A primer on missing data

Sometimes there is missing data in a dataset. For an example, I am going to take the earlier counts table and add a few NAs. If I now try to calculate the mean number of counts, I will get NA as an answer for the rows that had NAs.

```
counts <- data.frame(Site1 = c(geneA = 2, geneB = 4, geneC = 12, geneD = 8),
                     Site2 = c(geneA = 15, geneB = NA, geneC = 27, geneD = 28),
                     Site3 = c(geneA = 10, geneB = 7, geneC = 13, geneD = NA))

counts
```

```
##       Site1 Site2 Site3
## geneA     2    15    10
## geneB     4    NA     7
## geneC    12    27    13
## geneD     8    28    NA
```

```
apply(counts, MARGIN = 1, mean)
```

```
##    geneA    geneB    geneC    geneD 
##  9.00000       NA 17.33333       NA
```

How do we find out ahead of time that we are missing data? Knowing is half the battle. With a vector we can easily see how some basic functions work.

```
na_vec <- c(5, 6, NA, 7, 7, NA)

is.na(na_vec)
```

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

We are returned a logical vector of whether or not a value was NA. We can get the positional index and remove the NAs.

```
which(is.na(na_vec))
```

```
## [1] 3 6
```

```
idx <- which(is.na(na_vec))
remove_na_vec <- na_vec[-idx]
#equivalentish to
remove_na_vec <- na.omit(na_vec)
remove_na_vec
```

```
## [1] 5 6 7 7
## attr(,"na.action")
## [1] 3 6
## attr(,"class")
## [1] "omit"
```

With a large data frame, it may be hard to look at every cell to tell if there are NAs. The function `complete.cases()` looks by row to see whether any row contains an NA. You can then subset out the rows with the NAs.

```
is.na(counts)
```

```
##       Site1 Site2 Site3
## geneA FALSE FALSE FALSE
## geneB FALSE  TRUE FALSE
## geneC FALSE FALSE FALSE
## geneD FALSE FALSE  TRUE
```

```
any(is.na(counts))
```

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

```
complete.cases(counts)
```

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

```
counts[complete.cases(counts),]
```

```
##       Site1 Site2 Site3
## geneA     2    15    10
## geneC    12    27    13
```

If you want to keep all of the observations in your data frame and do your calculations anyways, now that you are aware of what is going on in your dataset, some functions specifically allow for this. Let's look up the documentation for the `mean()` function.

```
apply(counts, MARGIN = 1, mean, na.rm = TRUE)
```

```
##    geneA    geneB    geneC    geneD
##  9.00000  5.50000 17.33333 18.00000
```

In this case na.omit can be useful.

```
apply(counts, MARGIN = 1, na.omit(log))
```

```
##           geneA    geneB    geneC    geneD
## Site1 0.6931472 1.386294 2.484907 2.079442
## Site2 2.7080502       NA 3.295837 3.332205
## Site3 2.3025851 1.945910 2.564949       NA
```

You can similarly deal with NaNs in R. NaNs (not a number) are NAs (not available), but NAs are not NaNs. NaNs appear for imaginary or complex numbers or unusual numeric values. Some packages may output NAs, NaNs, or Inf/-Inf (can be found with `is.finite()`).

```
na_vec <- c(5, 6, NA, 7, 7, NA)
nan_vec <- c(5, 6, NaN, 7, 7, 0/0)

is.na(na_vec)
```

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

```
is.na(nan_vec)
```

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

```
is.nan(na_vec)
```

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

```
is.nan(nan_vec)
```

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

Basically, if you come across NaNs, you can use the same functions such as `complete.cases()` that you use with NAs.

Depending on your purpose, you may replace NAs with a sample average, or the mode of the data, or a value that is below a threshold.

```
counts[is.na(counts)] <- 0

counts
```

```
##       Site1 Site2 Site3
## geneA     2    15    10
## geneB     4     0     7
## geneC    12    27    13
## geneD     8    28     0
```

Installing and importing libraries

There are a few different places you can install packages from R. Listed in order of decreasing sketchiness:

- Bioconductor (Bioinformatics/Genomics focus)
  - Guidelines for submission, reviewed, and must have a vignette.
- CRAN (The Comprehensive R Archive Network)
  - Guidelines for submission, reviewed. Where the majority of packages are.
- GitHub
  - No formal review process, but peers can opens issues to highlight problems or suggest fixes.
  - The is an increasing number of publication-related packages.
  - Check to see the last time updates or comments were made to see if it is maintained by the developer.

- Joe's website
  - No review process. Not sure I trust that guy.

`devtools` is a package that is used for making R packages, but it also helps us to install packages from Github. It is downloaded from CRAN.

```
install.packages('devtools')
```

R may give you package installation warnings. Don't panic. In general, your package will either be installed and R will test if the installed package can be loaded, or R will give you a *non-zero exit status* - which means your package did not install. If you read the entire error message, it will give you a hint as to why the package did not install.

Some packages *depend* on previously developed packages and can only be installed after another package is installed in your library. Similarly, that previous package may depend on another package… here is the solution to install the package and all of the prior packages it relies on.

```
install.packages('devtools', dependencies = TRUE)
```

A package only has to be installed once. It is now in your *library.* To use a package, you must *load* the package into memory. Unless this is one of the packages R loads automatically, you choose which packages to load every session.

```
library(devtools)
# or library('devtools')
```

`library()` takes a single argument. We are going to write a function later in the course to load a character vector of packages. `library()` will throw an *error* if you try to load a package that is not installed. You may see `require()` on help pages, which also loads packages. It is usually used inside functions (it gives a *warning* instead of an error if a package is not installed). Errors will stop code from running while warnings allow code to run until an error is reached.

To install from Bioconductor you can either always use source to use biocLite…

```
source("https://bioconductor.org/biocLite.R")
biocLite("BiocInstaller")
```

Or you can use the BiocInstaller package to install other packages instead of typing in that url every time.

```
library(BiocInstaller)
biocLite('limma')
```

As forementioned `devtools` is required to install from GitHub. We don't actually need to load the entire package for `devtools` if we are only going to use one function. We select a function using this syntax `package::function()`.

```
devtools::install_github("jennybc/googlesheets")
```

All packages are loaded the same regardless of their origin, using `library()`.

```
library(googlesheets)
```

Challenge

Install and load the packages `tidyverse` and `readxl`.

Resources

http://archive.ics.uci.edu/ml/datasets/Adult
https://github.com/patrickwalls/R-examples/blob/master/LinearAlgebraInR.Rmd
http://stat545.com/block002__hello-r-workspace-wd-project.html
http://stat545.com/block026__file-out-in.html
http://sisbid.github.io/Module1/
https://github.com/tidyverse/readxl
https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
https://swcarpentry.github.io/r-novice-inflammation/06-best-practices-R/
http://stat545.com/help-general.html
https://stackoverflow.com/help/how-to-ask
https://www.r-project.org/posting-guide.html
http://www.quantide.com/ramarro-chapter-07/
file:///usr/lib/rstudio/www/docs/keyboard.htm

Post-Lesson Assessment

Your feedback is essential to help the next cohort of trainees. Please take a minute to complete the following short survey: https://www.surveymonkey.com/r/GD3KJB9

Thanks for coming!!!