Contenido

[Installing R 1](#_Toc47649281)

[What is R? What is CRAN? 2](#_Toc47649282)

[Why should you use R? 2](#_Toc47649283)

[Installing R 3](#_Toc47649284)

[Installation - for Windows 3](#_Toc47649285)

[Installation - for Macs 3](#_Toc47649286)

[**Installing RStudio** 4](#_Toc47649287)

[**What is RStudio?** 4](#_Toc47649288)

[**Installing RStudio** 4](#_Toc47649289)

[**Installing RStudio - Windows** 4](#_Toc47649290)

[**Installing RStudio - Macs** 5](#_Toc47649291)

[**Summary** 5](#_Toc47649292)

[**RStudio Tour** 5](#_Toc47649293)

[**The various quadrants** 5](#_Toc47649294)

[**The menu bar** 6](#_Toc47649295)

[**The console** 7](#_Toc47649296)

[**The environment/history** 7](#_Toc47649297)

[**The source/The script editor panel** 8](#_Toc47649298)

[**Files/help/plots/packages panel** 8](#_Toc47649299)

[**Summary** 8](#_Toc47649300)

[**R packages** 9](#_Toc47649301)

[**What is an R package?** 9](#_Toc47649302)

[**What are repositories?** 9](#_Toc47649303)

[**How do you know what package is right for you?** 10](#_Toc47649304)

[**How do you install packages?** 10](#_Toc47649305)

[**Loading packages** 11](#_Toc47649306)

[**Updating, removing, unloading packages** 12](#_Toc47649307)

[**Using the commands in a function** 13](#_Toc47649308)

[**Summary** 14](#_Toc47649309)

# Installing R

Now that we’ve got a handle on what a data scientist is, how to find answers, and then spent some time going over a data science example, it’s time to get you set up to start exploring on your own. And the first step of that is installing R.

## What is R? What is CRAN?

First, let’s remind ourselves exactly what R is and why we might want to use it.

[R](https://www.r-project.org/) is both a programming language and an environment, focused mainly on statistical analysis and graphics. It will be one of the main tools you use in this and following courses.

R is downloaded from the [Comprehensive R Archive Network](https://cran.r-project.org/), or CRAN, and while this might be your first brush with it, we will be returning to CRAN time and time again, when we install packages - so keep an eye out!

Why should you use R?

Outside of this course, you may be asking yourself - [why should I use R?](https://www.r-bloggers.com/why-use-r-five-reasons/)

The reasons for using R are myriad, but some big ones are:

**1) Its popularity**

R is quickly becoming the standard language for statistical analysis. This makes R a great language to learn as the more popular a software is, the quicker new functionality is developed, the more powerful it becomes, and the better the support there is! Additionally, as you can see in the [graph below](http://r4stats.com/articles/popularity/), knowing R is one of the top five languages asked for in data scientist job postings!

**R’s popularity among data scientists from r4stats.com**

**2) Its cost**

FREE!

This one is pretty self-explanatory - every aspect of R is free to use, unlike some other stats packages you may have heard of (eg: SAS, SPSS), so there is no cost barrier to using R!

**3) Its extensive functionality**

R is a very versatile language - we’ve talked about its use in stats and in graphing, but its use can be expanded to many different functions - from making [websites](http://rmarkdown.rstudio.com/rmarkdown_websites.html), making maps using [GIS data](http://www.nickeubank.com/gis-in-r/), analysing [language](https://cran.r-project.org/web/views/NaturalLanguageProcessing.html)… and even making these [lectures and videos](https://cran.r-project.org/web/packages/ari/index.html)! For whatever task you have in mind, there is often a package available for download that does exactly that!

**4) Its community**

And the reason that the functionality of R is so extensive is the community that has been built around R. Individuals have come together to make “packages” that add to the functionality of R - and more are being developed every day!

Particularly for people just getting started out with R, its community is a huge benefit - due to its popularity, there are multiple forums that have pages and pages dedicated to solving R problems. We talked about this in the Getting Help lesson; these forums are great both for finding other people who have had the same problem as you, and posting your own new problems.

Installing R

Now that we’ve spent some time looking at the benefits of R, it is time to install it! We’ll go over installation for both Windows and Mac below, but know that these are general guidelines and small details are likely to change subsequent to the making of this lecture - use this as a scaffold.

For both Windows and Mac machines, we start at the CRAN homepage: <https://cran.r-project.org/>

Installation - for Windows

If you are on a Windows computer, follow the link [Download R for Windows](https://cran.r-project.org/bin/windows/), and follow the directions there - if this is your first time installing R, go to the [base distribution](https://cran.r-project.org/bin/windows/base/) and click on the link at the top of the page that should say something like “Download R [version number] for Windows.” This will download an executable file for installation.

The download page for R on Windows

Open the executable, and if prompted by a security warning, allow it to run. Select the language you prefer during installation and agree to the licensing information. You will next be prompted for a destination location - this will likely be defaulted to Program Files, in a subfolder called R, followed by another directory of the version number. Unless you have any issues with this, the default location is perfect.

**The install wizard for installing R**

You will then be prompted to select which components should be installed. Unless you are running short on memory, installing all of the components is desireable. Next you’ll be asked about startup options, and again, the defaults are fine for this. You will then be asked where Setup should place shortcuts - this is completely up to you, you can allow it to add the program to the start menu, or you can click the box at the bottom that says to not create a start menu link. Finally, you will be asked whether you want a desktop or Quick Launch icon - up to you! I do not recommend changing the defaults for the registry entries though.

After this window, the installation should begin. Test that the installation worked by opening R for the first time!

**The R terminal running!**

Installation - for Macs

If you are on a Mac computer, follow the link [Download R for (Mac) OS X](https://cran.r-project.org/bin/macosx/). There you can find the various R versions for download. Note: If your Mac is older than OS X 10.6 (“Snow Leopard”), you will need to follow the directions on this page for downloading older versions of R that are compatible with those operating systems.

Click on the link to the most recent version of R, which will download a .pkg file.

# **Installing RStudio**

We’ve installed R and can open the R interface to input code, but there are other ways to interface with R - and one of those ways is using RStudio. In this lesson, we’ll get RStudio installed on your computer.

### **What is RStudio?**

[RStudio](https://www.rstudio.com/products/RStudio/) is a graphical user interface for R, that allows you to write, edit and store code, generate, view and store plots, manage files, objects and dataframes, and integrate with version control systems – to name a few of its functions. We will be exploring exactly what RStudio can do for you in future lessons, but for anybody just starting out with R coding, the visual nature of this program as an interface for R is a huge benefit.

### **Installing RStudio**

Thankfully, installation of RStudio is fairly straightforward. First, you go to the [RStudio download page](https://www.rstudio.com/products/rstudio/download/https:/www.rstudio.com/products/rstudio/download/). We want to download the RStudio Desktop version of the software, so click on the appropriate “Download” , under that heading and you will see a list of [“Installers for supported platforms”](https://www.rstudio.com/products/rstudio/download/#download).

**The RStudio download page. Select RStudio Desktop**

**The various versions of RStudio available for different operating systems**

At this point the installation process diverges for Macs and Windows, so follow the instructions for the appropriate OS.

### **Installing RStudio - Windows**

For Windows, select the RStudio installer for the various Windows editions (Vista, 7, 8, 10). This will initiate the download process. When the download is complete, open this executable file to access the installation wizard. You may be presented with a security warning at this time - allow it to make changes to your computer.

Following this, the installation wizard will open. Following the defaults on each of the windows of the wizard is appropriate for installation. In brief, on the welcome screen, click next. If you want RStudio installed elsewhere, “Browse” through your file system. Otherwise, it will likely default to the “Program Files” folder - this is appropriate. Click next. On this final page, allow RStudio to create a Start menu shortcut. Click Install. RStudio is now being installed. Wait for this process to finish; RStudio is now installed on your computer. Click Finish.

Check that RStudio is working appropriately by opening it from your Start menu.

**The installation wizard for RStudio installation**

**Completing installation of RStudio**

**RStudio is running!**

### **Installing RStudio - Macs**

For Macs, select the Macs OS X RStudio installer (Mac OS X 10.6+ (64-bit)). This will initiate the download process. When the download is complete, click on the downloaded file and it will begin to install. When this is finished, the applications window will open. Drag the RStudio icon into the Applications directory. Test the installation by opening your applications folder and opening the RStudio software.

**Download the Mac appropiate version of RStudio**

**Drag the RStudio file into your Applications folder to complete installation for RStudio**

**RStudio is running!**

### **Summary**

In this lesson we installed RStudio, both for Macs and for Windows computers. Before moving on to the next lecture, click through the available menus and explore the software a bit. We will have an entire lesson dedicated to exploring RStudio, but having some familiarity beforehand will be helpful!

# **RStudio Tour**

Now that we have RStudio installed, we should familiarize ourselves with the various components and functionality of it! RStudio provides a cheatsheet of the [RStudio environment](https://github.com/rstudio/cheatsheets/raw/master/rstudio-ide.pdf) - warning: this link initiates a download of a PDF from the RStudio GitHub.

### **The various quadrants**

Rstudio can be roughly divided into four quadrants, each with specific and varied functions, plus a main menu bar. When you first open RStudio, you should see a window that looks roughly like this:

**RStudio’s main interface**

You may be missing the upper left quadrant and instead have the left side of the screen with just one region, “Console” - if this is the case, go to File > New File > R Script and now it should more closely resemble the image. You can change the sizes of each of the various quadrants by hovering your mouse over the spaces between quadrants and click-dragging the divider to resize the sections.

We will go through each of the regions and describe some of their main functions. It would be impossible to cover everything that RStudio can do, so we urge you to explore RStudio on your own too!

**The four main quadrants of RStudio, plus the main menu bar**

### **The menu bar**

The menu bar runs across the top of your screen and should have two rows. The first row should be a fairly standard menu, starting with “File” and “Edit.” Below that, there is a row of icons that are shortcuts for functions that you’ll frequently use.

**The commonly used options of the main menu bar**

To start, let’s explore the main sections of the menu bar that you will use. The first being the File menu. Here we can open new or saved files, open new or saved projects (we’ll have an entire lesson in the future about R Projects, so stay tuned!), save our current document or close RStudio. If you mouse over “New File”, a new menu will appear that suggests the various file formats available to you. R Script and R Markdown files are the most common file types for use, but you can also generate R notebooks, web apps, websites, or slide presentations. If you click on any one of these, a new tab in the “Source” quadrant will open. We’ll spend more time in a future lesson on R Markdown files and their use.

**The File menu**

The Session menu has some R specific functions, in which you can restart, interrupt or terminate R - these can be helpful if R isn’t behaving or is stuck and you want to stop what it is doing and start from scratch.

**The Session menu**

The Tools menu is a treasure trove of functions for you to explore. For now, you should know that this is where you can go to install new packages (see next lecture), set up your version control software (see future lesson: Linking GitHub and RStudio), and set your options and preferences for how RStudio looks and functions. For now, we will leave this alone, but be sure to explore these menus on your own once you have a bit more experience with RStudio and see what you can change to best suit your preferences!

**The Tools menu**

### **The console**

This region should look familiar to you - when you opened R, you were presented with the console. This is where you type and execute commands, and where the output of said command is displayed.

**The console**

To execute your first command, try typing 1 + 1 then enter at the > prompt. You should see the output [1] 2 below your command.

**Typing into the console and getting an output**

Now copy and paste the following into your console and hit enter.

example <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8), nrow = 4, ncol = 2)

This creates a matrix with four rows and two columns, with the numbers 1 through 8.

### **The environment/history**

To view this matrix, first look to the Environment quadrant, where you should see the following:

**The environment quadrant**

Click anywhere on the “example” line, and a new tab on the Source quadrant should appear, showing the matrix you created. Any dataframe or matrix that you create in R can be viewed this way in RStudio.

**Your newly made matrix, opened in a new tab of the source panel**

RStudio also tells you some information about the object in the environment, like whether it is a list or a dataframe or if it contains numbers, integers or characters. This is very helpful information to have as some functions only work with certain classes of data. And knowing what kind of data you have is the first step to that.

The quadrant has two other tabs running across the top of it. We’ll just look at the History tab now. Your history tab should look something like this:

**The history tab**

Here you will see the commands that we have run in this session of R. If you click on any one of them, you can click “To Console” or “To Source” and this will either rerun the command in the console, or will move the command to the source, respectively. Do so now for your example matrix and send it to Source.

**The history tab**

### **The source/The script editor panel**

The Source panel is where you will be spending most of your time in RStudio. This is where you store the R commands that you want to save for later, either as a record of what you did or as a way to rerun code. We’ll spend a lot of time in this quadrant when we discuss R Markdown, but for now, click the save icon along the top of this quadrant and save the script as my\_first\_R\_script.R Now you will always have a record of creating this matrix.

**Your first R script!**

### **Files/help/plots/packages panel**

The final region we’ll look at occupies the bottom right of the RStudio window. In this quadrant, five tabs run across the top: Files, Plots, Packages, Help, and Viewer.

In Files, you can see all of the files in your current working directory. If this isn’t where you want to save or retrieve files from, you can also change the current working directory in this tab using the ellipsis at the far right, finding the desired folder, and then under the “More” cogwheel, setting this new folder as the working directory.

**The files tab**

In the Plots tab, if you generate a plot with your code, it will appear here. You can use the arrows to navigate to previously generated plots. The Zoom function will open the plot in a new window, that is much larger than the quadrant. Export is how you save the plot. You can either save it as an image or as a PDF. The broom icon clears all plots from memory.

**The plots tab**

The Packages tab will be explored more in depth in the next lesson on R packages. Here you can see all the packages you have installed, load and unload these packages, and update them.

**The packages tab**

The Help tab is where you find the documentation for your R packages and various functions. In the upper right of this panel there is a search function for when you have a specific function or package in question.

**The help tab**

### **Summary**

In this lesson we took a tour of the RStudio software. We became familiar with the main menu and its various menus. We looked at the Console, where R code is input and run. We then moved on to the Environment panel that lists all of the objects that have been created within an R session and allows you to view these objects in a new tab in Source. In this same quadrant, there is a History tab, that keeps a record of all commands that have been run. It also presents the option to either rerun the command in the Console, or send the command to Source, to be saved. Source is where you save your R commands. And the bottom right quadrant contains a listing of all the files in your working directory, displays generated plots, lists your installed packages, and supplies help files for when you need some assistance! Take some time to explore RStudio on your own!

# **R packages**

Now that we’ve installed R and RStudio and have a basic understanding of how they work together, we can get at what makes R so special: packages.

### **What is an R package?**

So far, anything we’ve played around with in R uses the “base” R system. Base R, or everything included in R when you download it, has rather basic functionality for statistics and plotting but it can sometimes be limiting. To expand upon R’s basic functionality, people have developed **packages.** A package is a collection of functions, data, and code conveniently provided in a nice, complete format for you. At the time of writing, there are just over 14,300 packages available to download - each with their own specialized functions and code, all for some different purpose. For a really in depth look at R Packages (what they are, how to develop them), check out Hadley Wickham’s book from O’Reilly, [“R Packages.”](http://r-pkgs.had.co.nz/)

Side note: A package is not to be confused with a **library** (these two terms are often conflated in colloquial speech about R). A library is the place where the package is located on your computer. To think of an analogy, a library is, well, a library… and a package is a book within the library. The library is where the books/packages are located.

Packages are what make R so unique. Not only does base R have some great functionality but these packages greatly expand its functionality. And perhaps most special of all, each package is developed and published by the R community at large and deposited in **repositories.**

### **What are repositories?**

A repository is a central location where many developed packages are located and available for download.

There are three big repositories:  
1. [**CRAN (Comprehensive R Archive Network):**](https://cran.r-project.org/web/packages/) R’s main repository (>12,100 packages available!)  
2. [**BioConductor:**](https://bioconductor.org/packages/release/BiocViews.html#___Software) A repository mainly for bioinformatic-focused packages  
3. [**GitHub:**](https://github.com/collections) A very popular, open source repository (not R specific!)

Take a second to explore the links above and check out the various packages that are out there!

**The big three repositories for R packages**

### **How do you know what package is right for you?**

So you know where to find packages… but there are so many of them, how can you find a package that will do what you are trying to do in R? There are a few different avenues for exploring packages.

First, CRAN groups all of its packages by their functionality/topic into 35 “themes.” It calls this its [“Task view.”](https://cran.r-project.org/web/views/) This at least allows you to narrow the packages you can look through to a topic relevant to your interests.

**CRAN’s “Task View” that groups packages into 35 topics**

Second, there is a great website, [**RDocumentation,**](https://www.rdocumentation.org/) which is a search engine for packages and functions from CRAN, BioConductor, and GitHub (ie: the big three repositories). If you have a task in mind, this is a great way to search for specific packages to help you accomplish that task! It also has a [“task” view](https://www.rdocumentation.org/taskviews) like CRAN, that allows you to browse themes.

More often, if you have a specific task in mind, Googling that task followed by “R package” is a great place to start! From there, looking at tutorials, vignettes, and forums for people already doing what you want to do is a great way to find relevant packages.

### **How do you install packages?**

Great! You’ve found a package you want… How do you install it?

**Installing from CRAN**  
If you are installing from the CRAN repository, use the install.packages() function, with the name of the package you want to install in quotes between the parentheses (note: you can use either single or double quotes). For example, if you want to install the package “ggplot2”, you would use: install.packages("ggplot2")

Try doing so in your R console! This command downloads the “ggplot2” package from CRAN and installs it onto your computer.

If you want to install multiple packages at once, you can do so by using a character vector, like: install.packages(c("ggplot2", "devtools", "lme4"))

If you want to use RStudio’s graphical interface to install packages, go to the Tools menu, and the first option should be “Install packages…” If installing from CRAN, select it as the repository and type the desired packages in the appropriate box.

**Various methods to install packages within R/RStudio**

**Installing packages from CRAN through R/RStudio**

**Installing from Bioconductor**  
The BioConductor repository uses their own method to [install packages](https://www.bioconductor.org/install/). First, to get the basic functions required to install through BioConductor, use: source("https://bioconductor.org/biocLite.R")

This makes the main install function of BioConductor, biocLite(), available to you. Following this, you call the package you want to install in quotes, between the parentheses of the biocLite command, like so: biocLite("GenomicFeatures")

**Installing packages with BioConductor**

**Installing from GitHub**  
This is a more specific case that you probably won’t run into too often. In the event you want to do this, you first must find the package you want on GitHub and take note of both the package name AND the author of the package. Check out [this guide](http://kbroman.org/pkg_primer/pages/github.html) for installing from GitHub, but the general workflow is:

1. install.packages("devtools") - only run this if you don’t already have devtools installed. If you’ve been following along with this lesson, you may have installed it when we were practicing installations using the R console
2. library(devtools) - more on what this command is doing immediately below this
3. install\_github("author/package") replacing “author” and “package” with their GitHub username and the name of the package.

**Installing packages from GitHub**

### **Loading packages**

Installing a package does not make its functions immediately available to you. First you must **load** the package into R; to do so, use the library() function. Think of this like any other software you install on your computer. Just because you’ve installed a program, doesn’t mean it’s automatically running - you have to open the program. Same with R. You’ve installed it, but now you have to “open” it. For example, to “open” the “ggplot2” package, you would run:library(ggplot2)

**NOTE:** Do **not** put the package name in quotes! Unlike when you are installing the packages, the library() command does not accept package names in quotes!

**Step one of getting a package is installing it, but to use it, you must load it using library(); similar to installing R and then loading it by opening the .exe file**

There is an order to loading packages - some packages require other packages to be loaded first (**dependencies**). That package’s manual/help pages will help you out in finding that order, if they are picky.

If you want to load a package using the RStudio interface, in the lower right quadrant there is a tab called “Packages” that lists out all of the packages and a brief description, as well as the version number, of all of the packages you have installed. To load a package just click on the checkbox beside the package name

**Using the RStudio interface to load a package**

### **Updating, removing, unloading packages**

Once you’ve got a package, there are a few things you might need to know how to do:

**Checking what packages you have installed**  
If you aren’t sure if you’ve already installed a package, or want to check what packages are installed, you can use either of: installed.packages() or library() with nothing between the parentheses to check!

In RStudio, that package tab introduced earlier is another way to look at all of the packages you have installed.

**Updating packages**

You can check what packages need an update with a call to the function old.packages() This will identify all packages that have been updated since you installed them/last updated them.

To update all packages, use update.packages(). If you only want to update a specific package, just use once again install.packages("packagename")

**Functions used to see what packages are installed and update them**

Within the RStudio interface, still in that Packages tab, you can click “Update,” which will list all of the packages that are not up to date. It gives you the option to update all of your packages, or allows you to select specific packages.

**Using the RStudio interface to update your packages**

You will want to periodically check in on your packages and check if you’ve fallen out of date - be careful though! Sometimes an update can change the functionality of certain functions, so if you re-run some old code, the command may be changed or perhaps even outright gone and you will need to update your code too!

**Unloading packages**

Sometimes you want to unload a package in the middle of a script - the package you have loaded may not play nicely with another package you want to use.

To unload a given package you can use the detach() function. For example, detach("package:ggplot2", unload=TRUE) would unload the ggplot2 package (that we loaded earlier). Within the RStudio interface, in the Packages tab, you can simply unload a package by unchecking the box beside the package name.

**Unloading or “detaching” a package**

**Uninstalling packages**

If you no longer want to have a package installed, you can simply uninstall it using the function remove.packages(). For example, remove.packages("ggplot2")

(Try that, but then actually re-install the ggplot2 package - it’s a super useful plotting package!)

Within RStudio, in the Packages tab, clicking on the “X” at the end of a package’s row will uninstall that package.

**Uninstalling packages**

**Sidenote: How do you know what version of R you have?**

Sometimes, when you are looking at a package that you might want to install, you will see that it requires a certain version of R to run. To know if you can use that package, you need to know what version of R you are running!

One way to know your R version is to check when you first open R/RStudio - the first thing it outputs in the console tells you what version of R is currently running. If you didn’t pay attention at the beginning, you can type version into the console and it will output information on the R version you are running. Another helpful command is sessionInfo() - it will tell you what version of R you are running along with a listing of all of the packages you have loaded. The output of this command is a great detail to include when posting a question to forums - it tells potential helpers a lot of information about your OS, R, and the packages (plus their version numbers!) that you are using.

**Ways to see what version of R you are running**

### **Using the commands in a function**

In all of this information about packages, we haven’t actually discussed how to use a package’s functions!

First, you need to know what functions are included within a package. To do this, you can look at the man/help pages included in all (well-made) packages. In the console, you can use the help() function to access a package’s help files. Try help(package = "ggplot2") and you will see all of the many functions that ggplot2 provides. Within the RStudio interface, you can access the help files through the Packages tab (again) - clicking on any package name should open up the associated help files in the “Help” tab, found in that same quadrant, beside the Packages tab. Clicking on any one of these help pages will take you to that functions help page, that tells you what that function is for and how to use it.

**The help functions available to you**

Once you know what function within a package you want to use, you simply call it in the console like any other function we’ve been using throughout this lesson. Once a package has been loaded, it is as if it were a part of the base R functionality.

If you still have questions about what functions within a package are right for you or how to use them, many packages include **“vignettes.”** These are extended help files, that include an overview of the package and its functions, but often they go the extra mile and include detailed examples of how to use the functions in plain words that you can follow along with to see how to use the package. To see the vignettes included in a package, you can use the browseVignettes() function. For example, let’s look at the vignettes included in ggplot2:browseVignettes("ggplot2") . You should see that there are two included vignettes: “Extending ggplot2” and “Aesthetic specifications.” Exploring the Aesthetic specifications vignette is a great example of how vignettes can be helpful, clear instructions on how to use the included functions.

**How to browse vignettes for packages**

### **Summary**

In this lesson, we’ve explored R packages in depth. We examined what a packages is (and how it differs from a library), what repositories are, and how to find a package relevant to your interests. We investigated all aspects of how packages work: how to install them (from the various repositories), how to load them, how to check which packages are installed, and how to update, uninstall, and unload packages. We took a small detour and looked at how to check what version of R you have, which is often an important detail to know when installing packages. And finally, we spent some time learning how to explore help files and vignettes, which often give you a good idea of how to use a package and all of its functions.

If you still want to learn more about R packages, here are two great resources! [R Packages: A Beginner’s Guide](https://www.datacamp.com/community/tutorials/r-packages-guide) from Adolfo Álvarez on DataCamp and a lesson from the University of Washington, on an [Introduction to R Packages](http://faculty.washington.edu/kenrice/rintro/sess08.pdf) from Ken Rice and Timothy Thornton.

# **R Projects**

One of the ways people organize their work in R is through the use of R Projects, a built in functionality of RStudio that helps to keep all your related files together. RStudio provides a [great guide](https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects) on how to use Projects so definitely check that out!

### **What is an R Project?**

When you make a Project, it creates a folder where all files will be kept, which is helpful for organizing yourself and keeping multiple projects separate from each other. When you re-open a project, RStudio remembers what files were open and will restore the work environment as if you had never left - which is very helpful when you are starting back up on a project after some time off! Functionally, creating a Project in R will create a new folder and assign that as the working directory so that all files generated will be assigned to the same directory.

### **What are the benefits to using Projects?**

The main benefit of using Projects is that it starts the organization process off right! It creates a folder for you and now you have a place to store all of your input data, your code, and the output of your code. Everything you are working on within a Project is self-contained; which often means finding things is much easier - there’s only one place to look!

Also, since everything related to one project is all in the same place, it is much easier to share your work with others - either by directly sharing the folder/files, or by associating it with version control software. We’ll talk more about linking Projects in R with version control systems in a future lesson entirely dedicated to the topic!

Finally, since RStudio remembers what documents you had open when you closed the session, it is easier to pick a project up after a break - everything is set-up just as you left it!

### **Creating a Project**

There are three ways to make a Project:  
1) From scratch - this will create a new directory for all your files to go in  
2) From an existing folder - this will link an existing directory with RStudio  
3) From version control - this will “clone” an existing project onto your computer (Don’t worry too much about this one, you’ll get more familiar with it in the next few lessons)

Let’s create a Project from scratch, which is often what you will be doing!

Open RStudio, and under File, select “New Project”. You can also create a new Project by using the Projects toolbar and selecting “New Project” in the drop down menu, or there is a “New Project” shortcut in the toolbar.

**Ways to initiate a new project**

Since we are starting from scratch, select “New Project” and a window will appear. Select “New Directory” and when prompted about the Project type, select “New Project”

**New project options**

Pick a name for your project and for this time, save it to your Desktop. This will create a folder on your Desktop where all of the files associated with this Project will be kept. Click “Create Project.”

**Creating a new project**

A blank RStudio session should open.

**Your new project**

A few things to note:  
1) In the “Files” quadrant of the screen, you can see that RStudio has made this new directory your working directory and generated a single file with the extension “.Rproj”  
2) In the upper-right of the window, there is a Projects toolbar that states the name of your current Project and has a drop down menu with a few different options that we’ll talk about in a second.

**Note the new project file in the Files quadrant and the Project toolbar**

### **Opening a project**

Opening an existing Project is as simple as double clicking the .Rproj file on your computer. You can accomplish the same from within RStudio by opening RStudio and going to File > Open Project. You can also use the Project toolbar and open the drop down menu and select “Open Project…”

**Ways to open a project**

### **Quitting a project or switching to another**

Quitting a project is as simple as closing your RStudio window. You can also go to File > Close Project, and this will do the same. Finally, you can use the Project toolbar by clicking on the drop down menu and choosing “Close Project”.

**Ways to quit a project**

All of these options will quit a Project and doing so will cause RStudio to write which documents are currently open (so they can be restored when you start back up again) and it then closes the R session. When you set up your Project, you can tell it to save environment (so, for example, all of your variables and data tables will be preloaded when you reopen the project), but this is not the default behaviour.

The Projects toolbar is also an easy way to switch between Projects - click on the drop down menu and choose “Open Project” and find your new Project you want to open - this will save the current Project, close it, and then open the new Project within the same window. If you want multiple Projects open at the same time, do the same but instead select “Open Project in New Session”. This can also be accomplished through the File menu, where those same options are available.

**Ways to switch between projects**

### **Best practices**

When you are setting up a project, it can be helpful to start out creating a few directories. Try a few strategies and see what works best for you, but most file structures are set-up around having a directory containing the raw data, a directory that you keep scripts/R files in, and a directory for the output of your code.

For example:

**An example of a possible folder structure to organize your project**

If you set up these folders before you start, it can save you organizational headaches later on in a project when you can’t quite remember where something is!

### **Summary**

In this lesson we’ve covered what Projects in R are, why you might want to use them, how to open, close, or switch between projects, and some best practices to best set you up for organizing yourself!

# **Version Control**

Now that we’ve got a handle on R, RStudio, and projects, there are a few more things we want to set you up with before moving on to the other courses - understanding version control, installing Git, and linking Git with RStudio. In this lesson, we’ll give you a basic understanding of version control.

### **What is version control?**

First things first: What is version control? Version control is a system that records changes that are made to a file or a set of files over time. As you make edits, the version control system takes snapshots of your files and the changes, and then saves those snapshots so you can refer or revert back to previous versions later if need be! If you’ve ever used the “Track changes” feature in Microsoft Word, you have seen a rudimentary type of version control, in which the changes to a file are tracked, and you can either choose to keep those edits or revert to the original format. Version control systems, like [Git](https://git-scm.com/), are like a more sophisticated “Track changes” - in that they are far more powerful and are capable of meticulously tracking successive changes on many files, with potentially many people working simultaneously on the same groups of files.

If you’ve ever worked collaboratively on a document before, [this comic](http://phdcomics.com/comics/archive.php?comicid=1531) from PHD Comics might resonate with you.

Hopefully, once you’ve mastered version control software, Paper\_Final\_FINAL2\_actually\_FINAL.docx will be a thing of the past for you!

### **What are the benefits of using version control?**

As we’ve seen in the example, without version control, you might be keeping multiple, very similar copies of a file. And this could be dangerous - you might start editing the wrong version, not recognizing that the document labelled “FINAL” has been further edited to “FINAL2” - and now all your new changes have been applied to the wrong file! Version control systems help to solve this problem by keeping a single, updated version of each file, with a record of all previous versions AND a record of exactly what changed between the versions.

Which brings us to the next major benefit of version control: It keeps a record of all changes made to the files. This can be of great help when you are collaborating with many people on the same files - the version control software keeps track of who, when, and why those specific changes were made. It’s like “Track changes” to the extreme!

**An example of the version control history for the development of this course!**

This record is also helpful when developing code, if you realize after some time that you made a mistake and introduced an error. You can find the last time you edited that particular bit of code, see the changes you made, and revert back to that original, unbroken code, leaving everything else you’ve done in the meanwhile untouched!

Finally, when working with a group of people on the same set of files, version control is helpful for ensuring that you aren’t making changes to files that conflict with other changes. If you’ve ever shared a document with another person for editing, you know the frustration of integrating their edits with a document that has changed since you sent the original file - now you have two versions of that same original document. Version control allows multiple people to work on the same file and then helps merge all of the versions of the file and all of their edits into one cohesive file.

### **What is Git? Why should you use it?**

Git is a free and open source version control system. It was developed in 2005 and has since become the most commonly used version control system around! StackOverflow, which should sound familiar from our Getting Help lesson, surveyed over 60,000 respondents on which version control system they use, and as you can tell from the chart below, [Git is by far the winner](https://insights.stackoverflow.com/survey/2017#work-version-control).

**Results of a StackOverflow survey asking which version control software their respondents use**

And as you become more familiar with Git and how it works and interfaces with your projects, you’ll begin to see why is has risen to the height of popularity. One of the main benefits of Git is that it keeps a local copy of your work and revisions, which you can then edit offline, and then once you return to internet service, you can sync your copy of the work, with all of your new edits and tracked changes to the main repository online. Additionally, since all collaborators on a project have their own local copy of the code, everybody can simultaneously work on their own parts of the code, without disturbing that common repository.

Another big benefit that we’ll definitely be taking advantage of is the ease with which RStudio and Git interface with each other. In the next lesson, we’ll work on getting Git installed and linked with RStudio and making a GitHub account.

### **What is GitHub?**

GitHub is an online interface for Git. Git is software used locally on your computer to record changes. GitHub is a host for your files and the records of the changes made. You can sort of think of it as being similar to DropBox - the files are on your computer, but they are also hosted online and are accessible from any computer. GitHub has the added benefit of interfacing with Git to keep track of all of your file versions and changes.

### **Version control vocabulary**

There is a lot of vocabulary involved in working with Git, and often the understanding of one word relies on your understanding of a different Git concept. Take some time to familiarize yourself with the words below and read over it a few times to see how the concepts relate.

**Repository:** Equivalent to the project’s folder/directory - all of your version controlled files (and the recorded changes) are located in a repository. This is often shortened to **repo**. Repositories are what are hosted on GitHub and through this interface you can either keep your repositories private and share them with select collaborators, or you can make them public - anybody can see your files and their history.

**Commit:** To commit is to save your edits and the changes made. A commit is like a snapshot of your files: Git compares the previous version of all of your files in the repo to the current version and identifies those that have changed since then. Those that have not changed, it maintains that previously stored file, untouched. Those that have changed, it compares the files, logs the changes and uploads the new version of your file. We’ll touch on this in the next section, but when you commit a file, typically you accompany that file change with a little note about what you changed and why.

When we talk about version control systems, commits are at the heart of them. If you find a mistake, you revert your files to a previous commit. If you want to see what has changed in a file over time, you compare the commits and look at the messages to see why and who.

**Push:** Updating the repository with your edits. Since Git involves making changes locally, you need to be able to share your changes with the common, online repository. Pushing is sending those committed changes to that repository, so now everybody has access to your edits.

**Pull:** Updating your local version of the repository to the current version, since others may have edited in the meanwhile. Because the shared repository is hosted online and any of your collaborators (or even yourself on a different computer!) could have made changes to the files and then pushed them to the shared repository, you are behind the times! The files you have locally on your computer may be outdated, so you pull to check if you are up to date with the main repository.

**Analogies to these concepts**

**Staging:** The act of preparing a file for a commit. For example, if since your last commit you have edited three files for completely different reasons, you don’t want to commit all of the changes in one go; your message on why you are making the commit and what has changed will be complicated since three files have been changed for different reasons. So instead, you can stage just one of the files and prepare it for committing. Once you’ve committed that file, you can stage the second file and commit it. And so on. Staging allows you to separate out file changes into separate commits. Very helpful!

To summarize these commonly used terms so far and to test whether you’ve got the hang of this, files are hosted in a **repository** that is shared online with collaborators. You **pull** the repository’s contents so that you have a local copy of the files that you can edit. Once you are happy with your changes to a file, you **stage** the file and then **commit** it. You **push** this commit to the shared repository. This uploads your new file and all of the changes and is accompanied by a message explaining what changed, why and by whom.

**Branch:** When the same file has two simultaneous copies. When you are working locally and editing a file, you have created a branch where your edits are not shared with the main repository (yet) - so there are two versions of the file: the version that everybody has access to on the repository and your local edited version of the file. Until you push your changes and merge them back into the main repository, you are working on a branch. Following a branch point, the version history splits into two and tracks the independent changes made to both the original file in the repository that others may be editing, and tracking your changes on your branch, and then merges the files together.

**Merge:** Independent edits of the same file are incorporated into a single, unified file. Independent edits are identified by Git and are brought together into a single file, with both sets of edits incorporated. But, you can see a potential problem here - if both people made an edit to the same sentence that precludes one of the edits from being possible, we have a problem! Git recognizes this disparity (**conflict**) and asks for user assistance in picking which edit to keep.

**Conflict:** When multiple people make changes to the same file and Git is unable to merge the edits. You are presented with the option to manually try and merge the edits or to keep one edit over the other.

\*\*A visual representation of these concepts, from [https://www.atlassian.com/git/tutorials/using-branches/git-merge\*\*](https://www.atlassian.com/git/tutorials/using-branches/git-merge**)

**Clone:** Making a copy of an existing Git repository. If you have just been brought on to a project that has been tracked with version control, you would clone the repository to get access to and create a local version of all of the repository’s files and all of the tracked changes.

**Fork:** A personal copy of a repository that you have taken from another person. If somebody is working on a cool project and you want to play around with it, you can fork their repository and then when you make changes, the edits are logged on your repository, not theirs.

### **Best practices**

It can take some time to get used to working with version control software like Git, but there are a few things to keep in mind to help establish good habits that will help you out in the future.

One of those things is to make purposeful commits. Each commit should only address a single issue. This way if you need to identify when you changed a certain line of code, there is only one place to look to identify the change and you can easily see how to revert the code.

Similarly, making sure you write informative messages on each commit is a helpful habit to get into. If each message is precise in what was being changed, anybody can examine the committed file and identify the purpose for your change. Additionally, if you are looking for a specific edit you made in the past, you can easily scan through all of your commits to identify those changes related to the desired edit.

You don’t want to get in the same habit that [XKCD](https://xkcd.com/1296/) has!

Finally, be cognizant of the version of files you are working on. Frequently check that you are up to date with the current repo by frequently pulling. Additionally, don’t horde your edited files - once you have committed your files (and written that helpful message!), you should push those changes to the common repository. If you are done editing a section of code and are planning on moving on to an unrelated problem, you need to share that edit with your collaborators!

**A summary of the main best practices to keep in mind as you work with version control**

### **Summary**

Now that we’ve covered what version control is and some of the benefits, you should be able to understand why we have three whole lessons dedicated to version control and installing it. We looked at what Git and GitHub are, and then covered much of the commonly used (and sometimes confusing!) vocabulary inherent to version control work. We then quickly went over some best practices to using Git – but the best way to get a hang of this all is to use it! Hopefully you feel like you have a better handle on how Git works than the people in [this XKCD comic](https://xkcd.com/1597/)! So let’s move on to the next lesson and get it installed!

# **GitHub and Git**

Now that we’ve got a handle on what version control is, in this lesson, you will sign-up for a GitHub account, navigate around the GitHub website to become familiar with some of its features, and install and configure Git; all in preparation for linking both with your RStudio!

### **What is GitHub?**

As we previously learned, [GitHub](https://github.com/) is a cloud-based management system for your version controlled files. Like DropBox, your files are both locally on your computer and hosted online and easily accessible. Its interface allows you to manage version control and provides users with a web-based interface for creating projects, sharing them, updating code, etc.

### **Signing up for GitHub**

To get a GitHub account, first go to <https://github.com/>. You will be brought to their homepage, where you should fill in your information - make a username, put in your email, choose a secure password, and click “Sign up for GitHub.”

**Signing up for GitHub**

### **Logging in to GitHub**

You should now be logged in to GitHub! In the future, to log on to GitHub, go to <https://github.com/>, where you will be presented with the homepage. If you aren’t already logged in, click on the “Sign in” link at the top.

Once you’ve done that, you will see the log in page where you will enter in your username and password that you created earlier.

**GitHub’s log in page**

Once logged in, you will be back at <https://github.com/>, but this time the screen should look like this:

\*\*GitHub’s homepage at [https://github.com/\*\*](https://github.com/**)

### **The homepage**

We’re going to take a quick tour of the GitHub website, and we’ll particularly focus on these sections of the interface:

1. User settings
2. Notifications
3. Help files
4. The GitHub guide

Following this tour, we’ll make your very first repository using the GitHub guide!

**Some major features of GitHub**

### **User settings**

Now that you’ve logged on to GitHub, we should fill out some of your profile information and get acquainted with the account settings. In the upper right corner, there is an icon with an arrow beside it, click this and go to “Your profile”

**Where to find user settings**

This is where you control your account from and can view your contribution histories and repositories.

**Your profile**

Since you are just starting out, you aren’t going to have any repositories or contributions yet - but hopefully we’ll change that soon enough! What we can do right now is edit your profile.

Go to “Edit profile” along the lefthand edge of the page. Here, take some time and fill out your name and a little description of yourself in the “Bio” box, and if you like, upload a picture of yourself! When you are done, click “Update profile”

**Editing your profile page**

Along the lefthand side of this page, there are many options for you to explore. Click through each of these menus to get familiar with the options available to you. To get you started, go to the account page.

**Your account page**

Here, you can edit your password or if you are unhappy with your username, change it. Be careful though, there can be [unintended consequences](https://help.github.com/articles/what-happens-when-i-change-my-username/) when you change your username - if you are just starting out and don’t have any content yet, you’ll probably be safe though.

Continue looking through the personal setting options on your own. When you are done, go back to your profile.

Once you’ve had a bit more experience with GitHub, you’ll eventually end up with some repositories to your name. To find those, click on the “Repositories” link on your profile. For now, it will probably look like this:

**Your repositories page**

By the end of the lecture though, check back to this page to find your newly created repository!

### **Notifications**

Next, we’ll check out the [notifications menu](https://github.com/notifications). Along the menu bar across the top of your window, there is a bell icon, representing your notifications. Click on the bell.

**Location of the bell icon**

**Your notifications**

Once you become more active on GitHub and are collaborating with others, here is where you can find messages and notifications for all the repositories, teams, and conversations you are a part of.

### **Help files**

Along the bottom of every. single. page. there is the [“Help” button](https://help.github.com/). GitHub has a great help system in place - if you ever have a question about GitHub, this should be your first point to search! Take some time now and look through the various help files, and see if any catch your eye.

**At the bottom of every page, you can find the Help page**

**GitHub’s help files**

### **The GitHub guide**

GitHub recognizes that this can be an overwhelming process for new users, and as such have developed a mini tutorial to get you started with GitHub. Go through [this guide](https://guides.github.com/activities/hello-world/) now and create your first repository! When you are done, you should have a repository that looks something like this:

**Your first repository**

Take some time to explore around the repository - Check out your commit history so far. Here you can find all of the changes that have been made to the repository, and you can see **who** made the change, **when** they made the change, and provided you wrote an appropriate commit message, you can see **why** they made the change! It should look like similar to this:

**Your first repository’s commit history**

Once you’ve explored all of the options in the repository, go back to your user profile. It should look a little different from before:

**Your profile now shows your first repository**

Now when you are on your profile you can see your latest repository created and for a complete listing of your repositories, click on the “Repositories” tab. Here you can see all of your repositories, a brief description, the time of the last edit, and along the right hand side, there is an activity graph, showing when and how many edits have been made on the repository.

**Your shiny new repository page!**

### **Git**

As you may remember from our last lecture, Git is the free and open source version control system which GitHub is built on.

One of the main benefits of using the Git system is its compatibility with RStudio; however, in order to link the two software together, we first need to download and install Git on your computer.

### **Downloading and installing Git**

To download Git, go to <https://git-scm.com/download>. You should arrive at a webpage like this:

**Downloading Git from git-scm.com/download**

Click on the appropriate download link for your operating system. This should initiate the download process.

### **For Windows**

Once the download is finished, open the .exe file to initiate the installation wizard. If you receive a security warning, click “Run” and/or “Allow.” Following this, click through the installation wizard, generally accepting the default options unless you have a compelling reason not to.

**Installation wizard for Git on Windows**

Click “Install” and allow the wizard to complete the installation process. Following this, check the “Launch Git Bash” option, and unless you are curious, deselect the “View Release Notes” box, as you are probably not interested in this right now.

**Finishing the install process**

Doing so, a command line environment will open. Provided you accepted the default options during the installation process, there will now be a Start menu shortcut to launch Git Bash in the future. You have now installed Git.

**Git Bash is the command line interface you will use to configure Git**

### **For Mac**

We will walk you through the most common installation process however, there are multiple ways to get Git onto your Mac. You can follow the tutorials at <https://www.atlassian.com/git/tutorials/install-git> for alternative installation routes.

After downloading the appropriate Git version for Macs, you should have downloaded a DMG file for installation on your Mac. Open this file. This will install Git on your computer. A new window will open.

**Installation wizard for Git on Mac**

Double click on the .pkg file and an installation wizard will open. Click through the options, accepting the defaults. Click Install. When prompted, close the installation wizard. You have successfully installed Git!

**Steps to a successful installation of Git!**

### **Configuring Git**

Now that Git is installed, we need to configure it for use with GitHub, in preparation for linking it with RStudio.

We need to tell Git what your username and email are, so that it knows how to name each commit as coming from you. To do so, in the command prompt (either Git Bash for Windows or Terminal for Mac), type: git config --global user.name "Jane Doe" with your desired username in place of “Jane Doe.” This is the name each commit will be tagged with.

Following this, in the command prompt, type: git config --global user.email janedoe@gmail.com **MAKING SURE TO USE THE SAME EMAIL ADDRESS YOU SIGNED UP FOR GITHUB WITH!**

**Configuring Git to tag each commit with your name and interface with GitHub**

### **Confirming your configuration**

At this point, you should be set for the next step, but just to check, confirm your changes by typing: git config --list

**Confirming your user name and user email**

Doing so, you should see the username and email you selected above. If you notice any problems or want to change these values, just retype the original config commands from earlier with your desired changes.

Once you are satisfied that your username and email is correct, exit the command line by typing exit and hit Enter. At this point, you are all set up for the next lecture!

# **Linking Git/GitHub and RStudio**

Now that we have both RStudio and Git set-up on your computer and a GitHub account, it’s time to link them together so that you can maximize the benefits of using RStudio in your version control pipelines.

### **Linking RStudio and Git**

In RStudio, go to Tools > Global Options > Git/SVN

**Use the Global Options menu to tell RStudio you are using Git as your version control system**

Sometimes the default path to the Git executable is not correct. Confirm that git.exe resides in the directory that RStudio has specified; if not, change the directory to the correct path. Otherwise, click OK or Apply.

**Confirm that the directory RStudio points to for the Git executable is correct**

RStudio and Git are now linked.

### **Linking RStudio and GitHub**

In that same RStudio option window, click “Create RSA Key” and when this completes, click “Close.”

Following this, in that same window again, click “View public key” and copy the string of numbers and letters. Close this window.

**Generate an RSA key and copy the public key to your clipboard**

You have now created a key that is specific to you which we will provide to GitHub, so that it knows who you are when you commit a change from within RStudio.

To do so, go to [github.com/](https://github.com/), log-in if you are not already, and go to your account settings. There, go to “SSH and GPG keys” and click “New SSH key”. Paste in the public key you have copied from RStudio into the Key box and give it a Title related to RStudio. Confirm the addition of the key with your GitHub password.

**Location of “SSH and GPG keys” on your profile settings**

**Telling GitHub the public SSH key generated in RStudio**

GitHub and RStudio are now linked. From here, we can create a repository on GitHub and link to RStudio.

### **Create a new repository and edit it in RStudio**

On GitHub, create a new repository (github.com > Your Profile > Repositories > New). Name your new test repository and give it a short description. Click Create repository. Copy the URL for your new repository.

**Location of the “Repositories” link on your profile**

**Creating a new repository on GitHub**

In RStudio, go to File > New Project. Select Version Control. Select Git as your version control software. Paste in the repository URL from before, select the location where you would like the project stored. When done, click on “Create Project”. Doing so will initialize a new project, linked to the GitHub repository, and open a new session of RStudio.

**Creating a version controlled project on RStudio**

**Cloning your Git repository to RStudio**

Create a new R script (File > New File > R Script) and copy and paste the following code:

print("This file was created within RStudio")

print("And now it lives on GitHub")

Save the file. Note that when you do so, the default location for the file is within the new Project directory you created earlier.

**Saving your first script for this project**

Once that is done, looking back at RStudio, in the Git tab of the environment quadrant, you should see your file you just created! Click the checkbox under “Staged” to stage your file.

**All files that have been modified since your last pull appear in the Git tab**

Click “Commit”. A new window should open, that lists all of the changed files from earlier, and below that shows the differences in the staged files from previous versions. In the upper quadrant, in the “Commit message” box, write yourself a commit message. Click Commit. Close the window.

**Commiting your R Script to the repository!**

So far, you have created a file, saved it, staged it, and committed it. If you remember your version control lecture, the next step is to push your changes to your online repository. Push your changes to the GitHub repository.

**How to push your commit to the GitHub repository**

Go to your GitHub repository and see that the commit has been recorded.

You’ve just successfully pushed your first commit from within RStudio to GitHub!

### **Summary**

In this lesson, we linked Git and RStudio, so that RStudio recognizes you are using Git as your version control software. Following that, we linked RStudio to GitHub, so that you can push and pull repositories from within RStudio. To test this, we created a repository on GitHub, linked it with a new project within RStudio, created a new file, and then staged, committed, and pushed the file to your GitHub

# **Projects under version control**

In the previous lesson, we linked RStudio with Git and GitHub. In doing this, we created a repository on GitHub and linked it to RStudio. Sometimes, however, you may already have an R Project that isn’t yet under version control or linked with GitHub. Let’s fix that!

### **Linking an existing Project with Git**

So what if you already have an R Project that you’ve been working on, but don’t have it linked up to any version control software (tut tut!)?

Thankfully, RStudio and GitHub recognize this can happen and have steps in place to help you (admittedly, this is slightly more troublesome to do than just creating a repository on GitHub and linking it with RStudio before starting the project…).

So first, let’s set up a situation where we have a local project that isn’t under version control. Go to File > New Project > New Directory > New Project and name your project. Since we are trying to emulate a time where you have a project not currently under version control, do **NOT** click “Create a git repository”. Click Create Project.

**Creating a project that is not under version control**

We’ve now created an R Project that is not currently under version control. Let’s fix that. First, let’s set it up to interact with Git. Open Git Bash or Terminal and navigate to the directory containing your project files. Move around directories by typing cd ~/dir/name/of/path/to/file

When the command prompt in the line before the dollar sign says the correct directory location of your project, you are in the correct location. Once here, type git init followed by git add . - this initializes (init) this directory as a git repository and adds all of the files in the directory (.) to your local repository. Commit these changes to the git repository using git commit -m "Initial commit"

# **R Markdown**

We’ve spent a lot of time getting R and RStudio working, learning about projects and version control - you are practically an expert at this! There is one last major functionality of R/RStudio that we would be remiss to not include in your introduction to R - [Markdown!](http://rmarkdown.rstudio.com/)

### **What is R Markdown?**

R Markdown is a way of creating fully reproducible documents, in which both text and code can be combined. In fact, these lessons are written using R Markdown! That’s how we make things:

* bullets
* **bold**
* italics
* [links](https://en.wikipedia.org/wiki/Rickrolling)
* or run inline r code

And by the end of this lesson, you should be able to do each of those things too, and more!

Despite these documents all starting as plain text, you can render them into HTML pages, or PDFs, or Word documents, or slides! The symbols you use to signal, for example, **bold** or italics is compatible with all of those formats.

### **Why use R Markdown?**

One of the main benefits is the reproducibility of using R Markdown. Since you can easily combine text and code chunks in one document, you can easily integrate introductions, hypotheses, your code that you are running, the results of that code and your conclusions all in one document. Sharing what you did, why you did it and how it turned out becomes so simple - and that person you share it with can re-run your code and get the exact same answers you got. That’s what we mean about reproducibility. But also, sometimes you will be working on a project that takes many weeks to complete; you want to be able to see what you did a long time ago (and perhaps be reminded exactly why you were doing this) and you can see exactly what you ran AND the results of that code - and R Markdown documents allow you to do that.

Another major benefit to R Markdown is that since it is plain text, it works very well with version control systems. It is easy to track what character changes occur between commits; unlike other formats that aren’t plain text. For example, in one version of this lesson, I may have forgotten to bold **this** word. When I catch my mistake, I can make the plain text changes to signal I would like that word bolded, and in the commit, you can see the exact character changes that occurred to now make the word bold.

Check out [this video](https://vimeo.com/178485416) that the RStudio developers have released about R Markdown and what it is!

### **Installation**

Another (selfish) benefit of R Markdown is how easy it is to use! Like everything in R, this extended functionality comes from an R package - “rmarkdown.” All you need to do to install it is run install.packages("rmarkdown")

And that’s it, you are ready to go.

### **Getting started with R Markdown**

To create an R Markdown document, in R Studio, go to File > New File > R Markdown. You will be presented with the following window:

**Initiating an R Markdown file**

I’ve filled in a title and an author and switched the output format to a PDF. Explore around this window and the tabs along the left to see all the different formats that you can output to. When you are done, click OK, and a new window should open with a little explanation on R Markdown files.

**The default template for R Markdown files**

There are three main sections of an R Markdown document. The first is the **header** at the top, bounded by the three dashes. This is where you can specify details like the title, your name, the date, and what kind of document you want output. If you filled in the blanks in the window earlier, these should be filled out for you.

Also on this page, you can see **text sections**, for example, one section starts with “## R Markdown” - We’ll talk more about what this means in a second, but this section will render as text when you produce the PDF of this file - and all of the formatting you will learn generally applies to this section.

And finally, you will see **code chunks**. These are bounded by the triple backticks. These are pieces of R code (“chunks”) that you can run right from within your document - and the output of this code will be included in the PDF when you create it.

The easiest way to see how each of these sections behave is to produce the PDF!

### **“Knitting” documents**

When you are done with a document, in R Markdown, you are said to **“knit”** your plain text and code into your final document. To do so, click on the “Knit” button along the top of the source panel. When you do so, it will prompt you to save the document as an RMD file. Do so.

You should see a document like this:

**The rendered PDF you created by knitting your markdown file**

So here you can see that the content of the header was rendered into a title, followed by your name and the date. The text chunks produced a section header called “R Markdown” which is followed by two paragraphs of text. Following this, you can see the R code summary(cars), importantly, followed by the output of running that code. And further down you will see code that ran to produce a plot, and then that plot. This is one of the huge benefits of R Markdown - rendering the results to code inline.

Go back to the R Markdown file that produced this PDF and see if you can see how you signify you want text bolded. (Hint: Look at the word “Knit” and see what it is surrounded by).

### **What are some easy Markdown commands?**

At this point, I hope we’ve convinced you that R Markdown is a useful way to keep your code/data and have set you up to be able to play around with it. To get you started, we’ll practice some of the formatting that is inherent to R Markdown documents.

To start, let’s look at bolding and italicising text. To bold text, you surround it by two asterisks on either side. Similarly, to italicise text, you surround the word with a single asterisk on either side. \*\*bold\*\* and \*italics\* respectively.

We’ve also seen from the default document that you can make section headers. To do this, you put a series of hash marks (#). The number of hash marks determines what level of heading it is. One hash is the highest level and will make the largest text (see the first line of this lecture), two hashes is the next highest level and so on. Play around with this formatting and make a series of headers, like so:

# Header level 1  
## Header level 2  
### Header level 3...

The other thing we’ve seen so far is code chunks. To make an R code chunk, you can type the three backticks, followed by the curly brackets surrounding a lower case R, put your code on a new line and end the chunk with three more backticks. Thankfully, RStudio recognized you’d be doing this a lot and there are short cuts, namely Ctrl+Alt+I (Windows) or Cmd + Option + I (Mac). Additionally, along the top of the source quadrant, there is the “Insert” button, that will also produce an empty code chunk. Try making an empty code chunk. Inside it, type the code print("Hello world"). When you knit your document, you will see this code chunk and the (admittedly simplistic) output of that chunk.

If you aren’t ready to knit your document yet, but want to see the output of your code, select the line of code you want to run and use Ctrl+Enter or hit the “Run” button along the top of your source window. The text “Hello world” should be output in your console window. If you have multiple lines of code in a chunk and you want to run them all in one go, you can run the entire chunk by using Ctrl+Shift+Enter OR hitting the green arrow button on the right side of the chunk OR going to the Run menu and selecting Run current chunk.

One final thing we will go into detail on is making bulleted lists, like the one at the top of this lesson. Lists are easily created by preceding each prospective bullet point by a single dash, followed by a space. Importantly, at the end of each bullet’s line, end with TWO spaces. This is a quirk of R Markdown that will cause spacing problems if not included.

* Try
* Making
* Your
* Own
* Bullet
* List!

This is a great starting point and there is so much more you can do with R Markdown. Thankfully, RStudio developers have produced an [“R Markdown cheatsheet”](http://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf) that we urge you to go check out and see everything you can do with R Markdown! The sky is the limit!

### **Summary**

In this lesson we’ve delved into R Markdown, starting with what it is and why you might want to use it. We hopefully got you started with R Markdown, first by installing it, and then by generating and knitting our first R Markdown document. We then looked at some of the various formatting options available to you and practiced generating code and running it within the R Studio interface.

# **Types of data science questions**

In this lesson, we’re going to be a little more conceptual and look at some of the types of analyses data scientists employ to answer questions in data science.

### **The main divisions of data science questions**

There are, broadly speaking, six categories in which data analyses fall. In the approximate order of difficulty, they are:

1. Descriptive
2. Exploratory
3. Inferential
4. Predictive
5. Causal
6. Mechanistic

Let’s explore the goals of each of these types and look at some examples of each analysis!

### **1. Descriptive analysis**

The goal of descriptive analysis is to **describe** or **summarize** a set of data. Whenever you get a new dataset to examine, this is usually the first kind of analysis you will perform. Descriptive analysis will generate simple summaries about the samples and their measurements. You may be familiar with common descriptive statistics: measures of central tendency (eg: mean, median, mode) or measures of variability (eg: range, standard deviations or variance).

This type of analysis is aimed at summarizing your sample – not for generalizing the results of the analysis to a larger population or trying to make conclusions. Description of data is separated from making interpretations; generalizations and interpretations require additional statistical steps.

Some examples of purely descriptive analysis can be seen in censuses. Here, the government collects a series of measurements on all of the country’s citizens, which can then be summarized. Here, you are being shown the age distribution in the US, stratified by sex. The goal of this is just to describe the distribution. There is no inferences about what this means or predictions on how the data might trend in the future. It is just to show you a summary of the data collected.

**A population pyramid describing the population distribution in the US**

### **2. Exploratory analysis**

The goal of exploratory analysis is to examine or **explore** the data and find **relationships** that weren’t previously known. Exploratory analyses explore how different measures might be related to each other but do not confirm that relationship as causitive. You’ve probably heard the phrase “Correlation does not imply causation” and exploratory analyses lie at the root of this saying. Just because you observe a relationship between two variables during exploratory analysis, it does not mean that one necessarily causes the other.

Because of this, exploratory analyses, while useful for discovering new connections, should not be the final say in answering a question! It can allow you to formulate hypotheses and drive the design of future studies and data collection, but exploratory analysis alone should never be used as the final say on why or how data might be related to each other.

Going back to the census example from above, rather than just summarizing the data points within a single variable, we can look at how two or more variables might be related to each other. In the plot below, we can see the percent of the workforce that is made up of women in various sectors and how that has changed between 2000 and 2016. Exploring this data, we can see quite a few relationships. Looking just at the top row of the data, we can see that women make up a vast majority of nurses and that it has slightly decreased in 16 years. While these are interesting relationships to note, the causes of these relationships is not apparent from this analysis. All exploratory analysis can tell us is that a relationship exists, not the cause.

**Exploring the relationships between the percentage of women in the workforce in various sectors between 2000 and 2016**

### **3. Inferential analysis**

The goal of inferential analyses is to use a relatively **small sample** of data to **infer** or say something about the **population** at large. Inferential analysis is commonly the goal of statistical modelling, where you have a small amount of information to extrapolate and generalize that information to a larger group.

Inferential analysis typically involves using the data you have to estimate that value in the population and then give a measure of your uncertainty about your estimate. Since you are moving from a small amount of data and trying to generalize to a larger population, your ability to accurately infer information about the larger population depends heavily on your sampling scheme - if the data you collect is not from a representative sample of the population, the generalizations you infer won’t be accurate for the population.

Unlike in our previous examples, we shouldn’t be using census data in inferential analysis - a census already collects information on (functionally) the entire population, there is nobody left to infer to; and inferring data from the US census to another country would not be a good idea because the US isn’t necessarily representative of another country that we are trying to infer knowledge about. Instead, a better example of inferential analysis is [a study](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3521092/) in which a subset of the US population was assayed for their life expectancy given the level of air pollution they experienced. This study uses the data they collected from a sample of the US population to infer how air pollution might be impacting life expectancy in the entire US.

### **4. Predictive analysis**

The goal of predictive analysis is to use **current** data to make **predictions** about **future** data. Essentially, you are using current and historical data to find patterns and predict the likelihood of future outcomes.

Like in inferential analysis, your accuracy in predictions is dependent on measuring the right variables. If you aren’t measuring the right variables to predict an outcome, your predictions aren’t going to be accurate. Additionally, there are many ways to build up prediction models with some being better or worse for specific cases, but in general, having more data and a simple model generally performs well at predicting future outcomes.

All this being said, much like in exploratory analysis, just because one variable may predict another, it does not mean that one causes the other; you are just capitalizing on this observed relationship to predict the second variable.

A common saying is that prediction is hard, especially about the future. There aren’t easy ways to gauge how well you are going to predict an event until that event has come to pass; so evaluating different approaches or models is a challenge.

We spend a lot of time trying to predict things - the upcoming weather, the outcomes of sports events, and in the example we’ll explore here, the outcomes of elections. We’ve previously mentioned Nate Silver of [FiveThirtyEight](http://fivethirtyeight.com/), where they try and predict the outcomes of U.S. elections (and sports matches, too!). Using historical polling data and trends and current polling, FiveThirtyEight builds models to predict the outcomes in the next US Presidential vote - and has been fairly accurate at doing so! FiveThirtyEight’s models accurately predicted the 2008 and 2012 elections and was widely considered an outlier in the 2016 US elections, as it was one of the few models to suggest Donald Trump at having a chance of winning.

**FiveThirtyEight’s predictions over time for the winner of the US 2016 election**

### **5. Causal analysis**

The caveat to a lot of the analyses we’ve looked at so far is that we can only see correlations and can’t get at the cause of the relationships we observe. Causal analysis fills that gap; the goal of causal analysis is to see what happens to one variable when we manipulate another variable - looking at the **cause** and **effect** of a **relationship**.

Generally, causal analyses are fairly complicated to do with observed data alone; there will always be questions as to whether it is correlation driving your conclusions or that the assumptions underlying your analysis are valid. More often, causal analyses are applied to the results of randomized studies that were designed to identify causation. Causal analysis is often considered the gold standard in data analysis, and is seen frequently in scientific studies where scientists are trying to identify the cause of a phenomenon, but often getting appropriate data for doing a causal analysis is a challenge.

One thing to note about causal analysis is that the data is usually analysed in aggregate and observed relationships are usually average effects; so, while on average giving a certain population a drug may alleviate the symptoms of a disease, this causal relationship may not hold true for every single affected individual.

As we’ve said, many scientific studies allow for causal analyses. Randomized control trials for drugs are a prime example of this. For example, [one randomized control trial](http://www.nejm.org/doi/full/10.1056/NEJMoa1702752) examined the effects of a new drug on treating infants with spinal muscular atrophy. Comparing a sample of infants receiving the drug versus a sample receiving a mock control, they measure various clinical outcomes in the babies and look at how the drug affects the outcomes.

### **6. Mechanistic analysis**

Mechanistic analyses are not nearly as commonly used as the previous analyses - the goal of mechanistic analysis is to understand the **exact changes in variables** that lead to **exact changes in other variables**. These analyses are exceedingly hard to use to infer much, except in simple situations or in those that are nicely modeled by deterministic equations. Given this description, it might be clear to see how mechanistic analyses are most commonly applied to physical or engineering sciences; biological sciences, for example, are far too noisy of data sets to use mechanistic analysis. Often, when these analyses are applied, the only noise in the data is measurement error, which can be accounted for.

You can generally find examples of mechanistic analysis in material science experiments. [Here](https://www.sciencedirect.com/science/article/pii/S0142941817303422), we have a study on biocomposites (essentially, making biodegradable plastics) that was examining how biocarbon particle size, functional polymer type and concentration affected mechanical properties of the resulting “plastic.” They are able to do mechanistic analyses through a careful balance of controlling and manipulating variables with very accurate measures of both those variables and the desired outcome.

### **Summary**

In this lesson we’ve covered the various types of data analysis, their goals, and looked at a few examples of each to demonstrate what each analysis is capable of (and importantly, what it is not).

# **Experimental Design**

Now that we’ve looked at the different types of data science questions, we are going to spend some time looking at experimental design concepts. As a data scientist, you are a scientist and as such, need to have the ability to design proper experiments to best answer your data science questions!

### **What does experimental design mean?**

Experimental design is organizing an experiment so that you have the correct data (and enough of it!) to clearly and effectively answer your data science question. This process involves clearly formulating your question in advance of any data collection, designing the best set-up possible to gather the data to answer your question, identifying problems or sources of error in your design, and only then, collecting the appropriate data.

### **Why should you care?**

Going into an analysis, you need to have a plan in advance of what you are going to do and how you are going to analyse the data. If you do the wrong analysis, you can come to the wrong conclusions!

We’ve seen many examples of this exact scenario play out in the scientific community over the years - there’s an entire website, [Retraction Watch](https://retractionwatch.com/), dedicated to identifying papers that have been retracted, or removed from the literature, as a result of poor scientific practices. And sometimes, those poor practices are a result of poor experimental design and analysis.

Occasionally, these erroneous conclusions can have sweeping effects; particularly in the field of human health. For example, [here](https://www.nature.com/articles/nm1491) we have a paper that was trying to predict the effects of a person’s genome on their response to different chemotherapies, to guide which patient receives which drugs to best treat their cancer. As you can see, this paper was retracted, over 4 years after it was initially published. In that time, this data, which was later shown to have numerous problems in their set-up and cleaning, was cited in nearly 450 other papers that may have used these erroneous results to bolster their own research plans. On top of this, this wrongly analysed data was used in clinical trials to determine cancer patient treatment plans. When the stakes are this high, experimental design is paramount.

**A retracted paper and the forensic analysis of what went wrong**

### **Principles of experimental design**

There are a lot of concepts and terms inherent to experimental design. Let’s go over some of these now!

**Independent variable (AKA factor):** The variable that the experimenter manipulates; it does not depend on other variables being measured. Often displayed on the x-axis.

**Dependent variable:** The variable that is expected to change as a result of changes in the independent variable. Often displayed on the y-axis, so that changes in X, the independent variable, effect changes in Y.

So when you are designing an experiment, you have to decide what variables you will measure, and which you will manipulate to effect changes in other measured variables. Additionally, you must develop your **hypothesis**, essentially an educated guess as to the relationship between your variables and the outcome of your experiment.

**How hypotheses, independent, and dependent variables are related to each other**

Let’s do an example experiment now! Let’s say for example that I have a hypothesis that as shoe size increases, literacy also increases. In this case, designing my experiment, I would choose a measure of literacy (eg: reading fluency) as my variable that depends on an individual’s shoe size.

**My experimental set-up: I hypothesize that literacy level depends on shoe size**

To answer this question, I will design an experiment in which I measure the shoe size and literacy level of 100 individuals. **Sample size** is the number of experimental subjects you will include in your experiment. There are ways to pick an optimal sample size, that you will cover in later courses. Before I collect my data though, I need to consider if there are problems with this experiment that might cause an erroneous result. In this case, my experiment may be fatally flawed by a **confounder**.

**Confounder:** An extraneous variable that may affect the relationship between the dependent and independent variables.

In our example, since age affects foot size and literacy is affected by age, if we see any relationship between shoe size and literacy, the relationship may actually be due to age – age is “confounding” our experimental design!

To **control** for this, we can make sure we also measure the age of each individual so that we can take into account the effects of age on literacy, as well. Another way we could **control** for age’s effect on literacy would be to **fix** the age of all participants. If everyone we study is the same age, then we have removed the possible effect of age on literacy.

**Age is confounding my experimental design! We need to control for this**

In other experimental design paradigms, a **control group** may be appropriate. This is when you have a group of experimental subjects that are not manipulated. So if you were studying the effect of a drug on survival, you would have a group that received the drug (**treatment**) and a group that did not (**control**). This way, you can compare the effects of the drug in the treatment versus control group.

**A control group is a group of subjects that do not receive the treatment, but still have their dependent variables measured**

In these study designs, there are other strategies we can use to control for confounding effects. One, we can **blind** the subjects to their assigned treatment group. Sometimes, when a subject knows that they are in the treatment group (eg: receiving the experimental drug), they can feel better, not from the drug itself, but from knowing they are receiving treatment. This is known as the **placebo effect**. To combat this, often participants are blinded to the treatment group they are in; this is usually achieved by giving the control group a mock treatment (eg: given a sugar pill they are told is the drug). In this way, if the placebo effect is causing a problem with your experiment, both groups should experience it equally.

**Blinding your study means that your subjects don’t know what group they belong to - all participants receive a “treatment”**

And this strategy is at the heart of many of these studies; spreading any possible confounding effects equally across the groups being compared. For example, if you think age is a possible confounding effect, making sure that both groups have similar ages and age ranges will help to mitigate any effect age may be having on your dependent variable - the effect of age is equal between your two groups.

This “balancing” of confounders is often achieved by **randomization**. Generally, we don’t know what will be a confounder beforehand; to help lessen the risk of accidentally biasing one group to be enriched for a confounder, you can randomly assign individuals to each of your groups. This means that any potential confounding variables should be distributed between each group roughly equally, to help eliminate/reduce systematic errors.

**Randomizing subjects to either the control or treatment group is a great strategy to reduce confounders’ effects**

There is one final concept of experimental design that we need to cover in this lesson, and that is **replication**. Replication is pretty much what it sounds like, repeating an experiment with different experimental subjects. A single experiment’s results may have occured by chance; a confounder was unevenly distributed across your groups, there was a systematic error in the data collection, there were some outliers, etc. However, if you can repeat the experiment and collect a whole new set of data and still come to the same conclusion, your study is much stronger. Also at the heart of replication is that it allows you to measure the **variability** of your data more accurately, which allows you to better assess whether any differences you see in your data are significant.

**Replication studies are a great way to bolster your experimental results and get measures of variability in your data**

### **Sharing data**

Once you’ve collected and analysed your data, one of the next steps of being a good citizen scientist is to share your data and code for analysis. Now that you have a GitHub account and we’ve shown you how to keep your version controlled data and analyses on GitHub, this is a great place to share your code!

In fact, hosted on GitHub, our group, [the Leek group](https://github.com/jtleek/datasharing), has developed a guide that has great advice for how to best share data!

### **Beware p-hacking!**

One of the many things often reported in experiments is a value called the **p-value**. This is a value that tells you the probability that the results of your experiment were observed by chance. This is a very important concept in statistics that we won’t be covering in depth here, if you want to know more, check out [this](https://www.youtube.com/watch?v=UsU-O2Z1rAs) video explaining more about p-values.

What you need to look out for is when you manipulate p-values towards your own end. Often, when your p-value is less than 0.05 (in other words, there is a 5 percent chance that the differences you saw were observed by chance), a result is considered [significant](https://xkcd.com/1478/). But if you do 20 tests, by chance, you would expect one of the twenty (5%) to be significant. In the age of big data, testing twenty hypotheses is a very easy proposition. And this is where the term [p-hacking](https://en.wikipedia.org/wiki/Data_dredging) comes from: This is when you exhaustively search a data set to find patterns and correlations that appear statistically significant by virtue of the sheer number of tests you have performed. These spurious correlations can be reported as significant and if you perform enough tests, you can find a data set and analysis that will show you what you wanted to see.

Check out this [FiveThirtyEight](https://projects.fivethirtyeight.com/p-hacking/) activity where you can manipulate and filter data and perform a series of tests such that you can get the data to find whatever relationship you want!

[XKCD](https://xkcd.com/882/) mocks this concept in a comic testing the link between jelly beans and acne - clearly there is no link there, but if you test enough jelly bean colours, eventually, one of them will be correlated with acne at p-value < 0.05!

### **Summary**

In this lesson we covered what experimental design is and why good experimental design matters. We then looked in depth to the principles of experimental design and defined some of the common terms you need to consider when designing an experiment. Next, we detoured a bit to see how you should share your data and code for analysis. And finally, we looked at the dangers of p-hacking and manipulating data to achieve significance.

# **Big Data**

A term you may have heard of before this course is “Big Data” - there have always been large datasets, but it seems like lately, this has become a buzzword in data science. But what does it mean?

### **What is big data?**

We talked a little about big data in the very first lecture of this course. As the name suggests, big data are very large data sets. We previously discussed three qualities that are commonly attributed to big data sets: Volume, Velocity, Variety. From these three adjectives, we can see that big data involves large data sets of diverse data types that are being generated very rapidly.

**Three qualities of big data**

So none of these qualities seem particulary new - why has the concept of big data been so recently popularized? In part, as technology and data storage has evolved to be able to hold larger and larger data sets, the definition of “big” has evolved too. Also, our ability to collect and record data has improved with time such that the speed with which data is collected is unprecedented. Finally, what is considered “data” has evolved, so that there is now more than ever - companies have recognized the benefits to collecting different sorts of information, and the rise of the internet and technology have allowed different and varied data sets to be more easily collected and available for analysis. One of the main shifts in data science has been moving from structured data sets to tackling unstructured data.

### **What is structured data? What is unstructured data?**

Structured data is what you traditionally might think of data; long tables, spreadsheets, or databases with columns and rows of information that you can sum or average or analyse however you like within those confines. Unfortunately, this is rarely how data is presented to you in this day and age. The data sets we commonly encounter are much messier, and it is our job to extract the information we want and corral it into something tidy and structured.

With the digital age and the advance of the internet, many pieces of information that weren’t traditionally collected were suddenly able to be translated into a format that a computer could record, store, search, and analyse. And once this was appreciated, there was a proliferation of this unstructured data being collected from all of our digital interactions: emails, Facebook and other social media interactions, text messages, shopping habits, smartphones (and their GPS tracking), websites you visit, how long you are on that website and what you look at, CCTV cameras and other video sources, etc. The amount of data and the various sources that can record and transmit data has exploded.

**Some examples of sources of unstructured data sources**

It is because of this explosion in the volume, velocity, and variety of data that “big data” has become so salient a concept; these data sets are now so large and complex that we need new tools and approaches to make the most of them. As you can guess given the variety of data types and sources, very rarely is the data stored in a neat, ordered spreadsheet, that traditional methods for cleaning and analysis can be applied to!

### **Challenges of working with big data**

Given some of the qualities of big data above, you can already start seeing some of the challenges that may be associated with working with big data.

1. It is big: there is a lot of raw data that you need to be able to store and analyse;
2. It is constantly changing and updating: By the time you finish your analysis, there is even more new data you could incorporate into your analysis! Every second you are analysing, is another second of data you haven’t used!
3. The variety can be overwhelming: There are so many sources of information that it can sometimes be difficult to determine what source of data may be best suited to answer your data science question! And finally,
4. It is messy: You don’t have neat data tables to quickly analyse - you have messy data. Before you can start looking for answers, you need to turn your unstructured data into a format that you can analyse!

### **Benefits to working with big data**

So with all of these challenges, why don’t we just stick to analysing smaller, more manageable, curated datasets and arriving at our answers that way?

Sometimes questions are best addressed using these smaller datasets, but many questions benefit from having lots and lots of data, and if there is some messiness or inaccuracies in this data, the sheer volume of it negates the effect of these small errors. So we are able to get closer to the truth even with these messier datasets.

Additionally, when you have data that is constantly updating, while this can be a challenge to analyse, the ability to have real time, up to date information allows you to do analyses that are accurate to the current state and make on the spot, rapid, informed predictions and decisions.

One of the benefits of having all these new sources of information is that questions that weren’t previously able to be answered due to lack of information, suddenly have many more sources to glean information from and new connections and discoveries are now able to be made! Questions that previously were inaccessible now have newer, unconventional data sources that may allow you to answer these formerly unfeasible questions.

Another benefit to using big data is that it can identify hidden correlations. Since we can collect data on a myriad of qualities on any one subject, we can look for qualities that may not be obviously related to our outcome variable, but the big data can identify a correlation there - instead of trying to understand precisely why an engine breaks down or why a drug’s side effect disappears, researchers can instead collect and analyze massive quantities of information about such events and everything that is associated with them, looking for patterns that might help predict future occurrences. Big data helps answer what, not why, and often that’s good enough.

**Comparing the challenges and benefits to working with big data**

### **Will big data solve all our problems?**

Big data has now made it possible to collect vast amounts of data, very rapidly, from a variety of sources (and improvements in technology have made it cheaper to collect, store and analyse) - but the question remains, how much of this data explosion is useful for answering questions you care about?

Regardless of the size of the data, you need the right data to answer a question. A famous statistician, John Tukey, said in 1986, “The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.” Essentially, any given data set may not be suited for your question, even if you really wanted it to; and big data does not fix this. Even the largest data sets around might not be big enough to be able to answer your question if it’s not the right data.

### **Summary**

In this lesson, we went over some qualities that characterize big data: volume, velocity, and variety. We compared structured and unstructured data, and examined some of the new sources of unstructured data. Then we turned to looking at the challenges and benefits of working with these big data sets. And finally, we came back to the idea that data science is question driven science and even the largest of data sets may not be appropriate for your case.