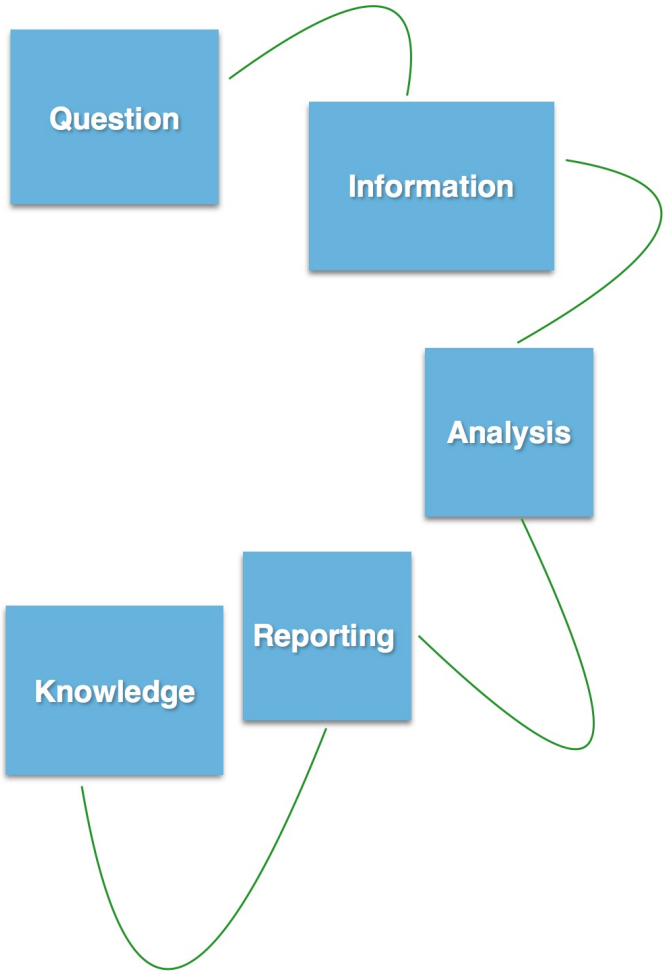
Lecture 3: Reproducible Research

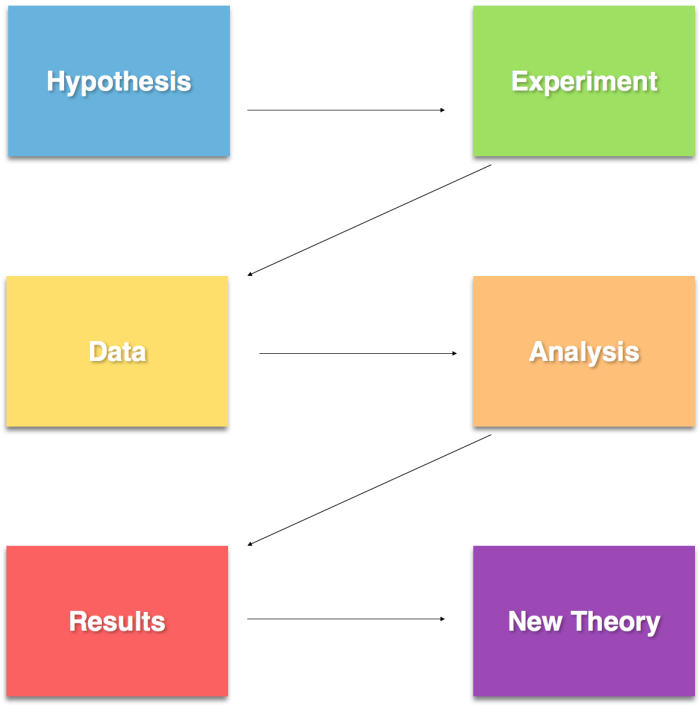
David Shilane

2019-02-07

Analytical Reporting



The Scientific Method



Reproducibility is the Gold Standard of Science

- Scientific theories are only considered sound if they make accurate and reliable predictions about the results of experiments.
- The full description of the experimental design, observed data, and methods of analysis are presented in detail.
- Anyone with the appropriate training should be able to **repeat the experiment** to independently evaluate the results.

This All Used to be Easy

- Simple experiments with small sample sizes.
- Widespread agreement on the methods of analysis.
- Low barriers to entry for reproducing the work yourself.

Complex Data, Complex Analyses

- Today's data are massive in scale, complex in dimension.
- Not all experimental designs include randomized assignments of treatments and controls. Many data sets are observational, incomplete, or potentially subject to biases.
- The methods of analysis can now include numerous choices in the selection of techniques, inclusion of features, form of the data, and interpretation of the results.

Greater Barriers to Entry, Too

- Many analyses now take place for private purposes within corporate environments.
- The relevant data may not be publicly available.
- Even a technical research report may not fully provide the details on how the work was done.

We now have greater opportunities than ever to work with large and diverse data sets. We also have greater reasons for skepticism in viewing the results described in reports.

Reproducible Research, Reproducible Reports

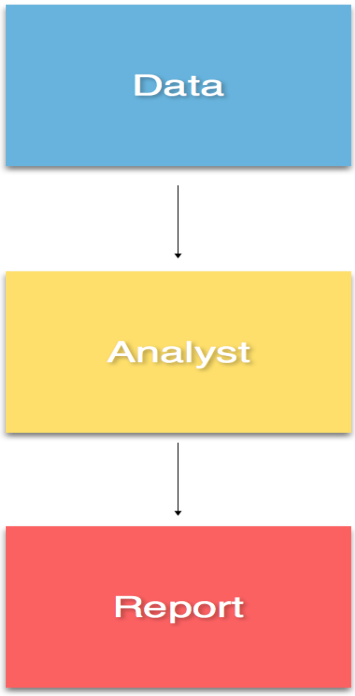
- It is no longer sufficient to create a reproducible experiment in complex circumstances.
- Reproducibility must extend into the methods of analysis.
- Reproducible analyses must also lead to reproducible reports.

What is a Reproducible Report?

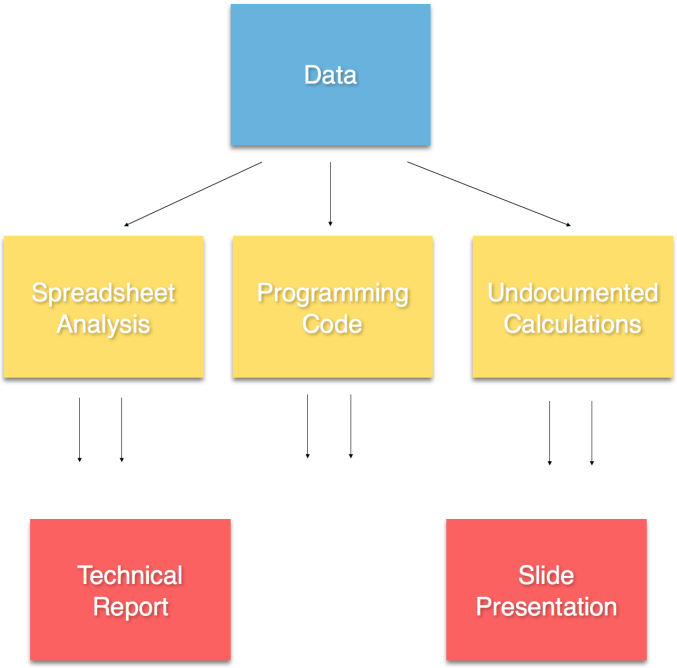
- All methods and calculations are thoroughly explained.
- Every fact and figure has a paper trail.
- An independent analyst could fully reproduce the results.
- The best way to meet this standard is to provide full access to the data, analysis code, and the report. A reproducible report could generate all of the results with the press of a button, **even if you supplied a new version of the data**.

A Simple Project

Example: A homework assignment or small research report.

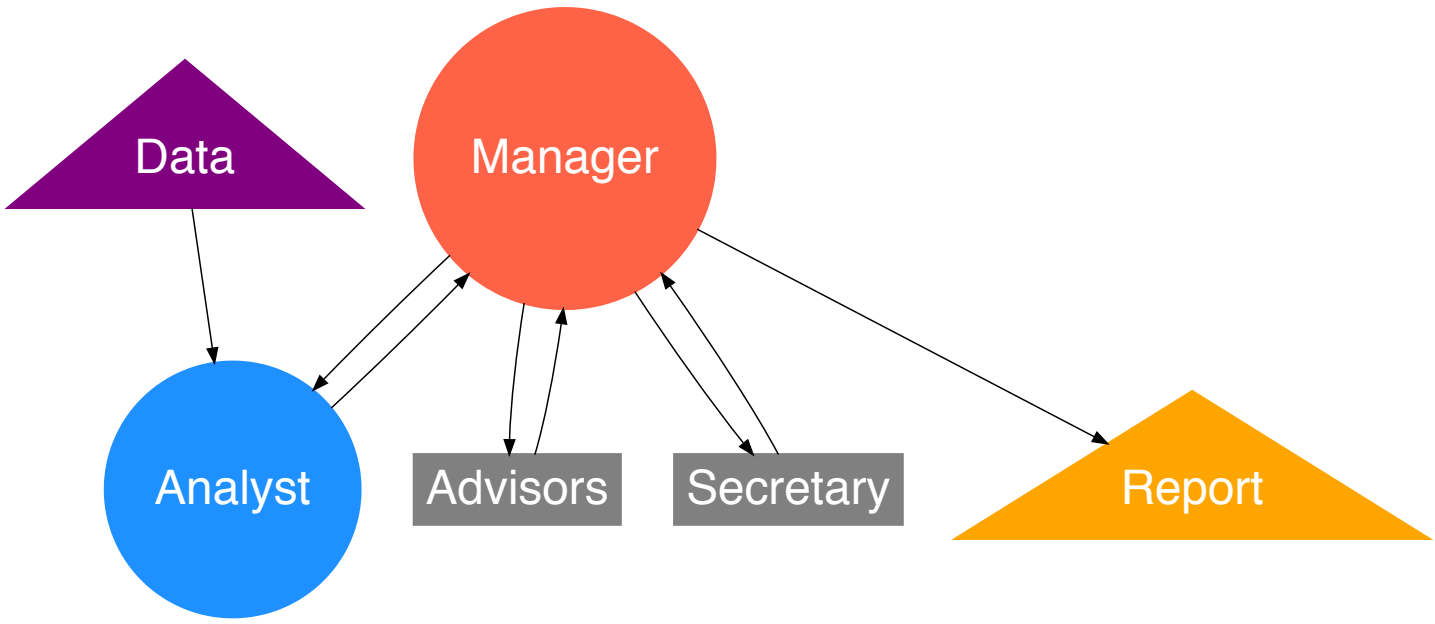


But It Probably Looks Like This



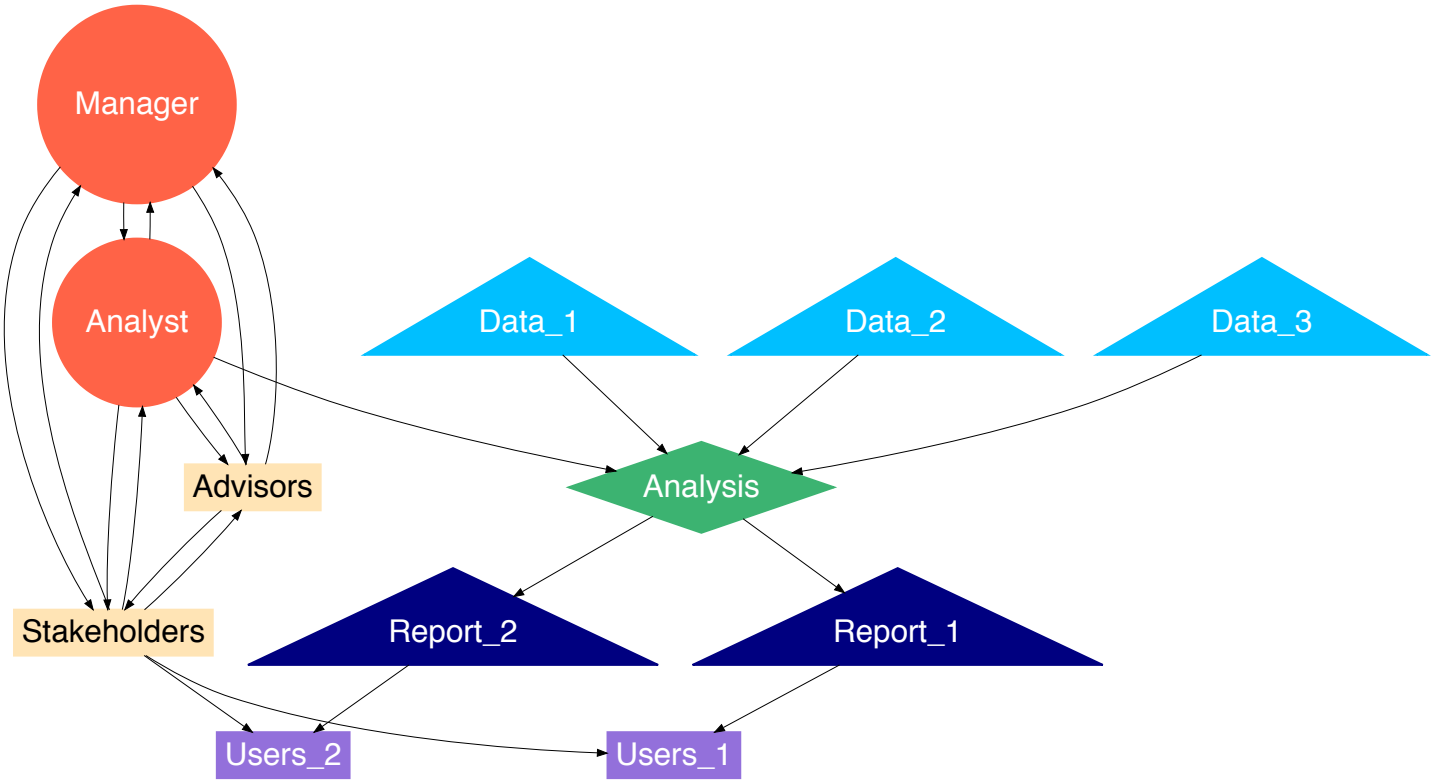
Moderate Complexity

- Now consider adding some additional people to the group, each of whom contribute in different ways: producing data, performing analyses, reviewing the work, writing reports, etc.
- Example: An analytics group with doctors, statisticians, and *rogue actions*.



Complex Analyses and Organizations

- Large teams don't just write a report; they build entire processes and systems.



A Million Excuses

Creating reproducible processes, reports, and systems can be very challenging. There are always many reasons why it might be difficult to implement.

- A member of the team needed a quick, ad-hoc analysis on a very tight deadline.
- It was easier to copy and paste a few items than to re-calculate everything.
- A few of the tables needed a small change that could be handled in a word processing program.
- Someone who is delivering the presentation in 30 minutes trusts the calculator on a phone or the back of an envelope.

The Possible Results

However, a lack of accountability in the process can lead to major problems:

- Reports that name the wrong person, get a few numbers wrong.
- No clear paper trail for how an important number – one that your organization is now known for – was ultimately derived.
- No ability to repeat the work when small changes are requested.

More Severe Results

- The wrong data were used to calculate an important figure.
- Your conclusions – and the products that you signed off on – have fundamental flaws.
- Your reputation can suffer.

We All Make Mistakes

- I have made more than my share of sloppy errors, oversights, and mistakes.
- In some cases, my errors have required considerable effort – and support from other team members – to mitigate and correct.
- However, the process can be greatly improved by implementing simple yet effective practices to improve the manner in which the results are generated.

Fixing the Process

- Involvement of multiple people in the process of content creation and review.
- Taking responsibility for inaccuracies, but also recognizing that mistakes happen and ensuring that there are low-stress opportunities to correct the work.
- Not rushing the process of development and review. Working quickly is great, but only if the cost of mistakes is not too great.

Changing the Process Starts with You

- As a data scientist, you have a great opportunity to introduce or improve an organization's approach to accountability and reproducibility.
- The methods you use can go a long way toward preventing the issues before they arise.
- Your example will have an impact on others, too.

Reproducibility is a Key to Productivity

- Documentation and lining up multiple processes is often thought to require additional time over pure development.
- However, working accountably from the start can greatly reduce the time that is required for resolving issues later on.
- Reproducible research and reporting enable me to do higher quality work at a faster pace. In many ways, it is an important conduit to greater productivity and success.

So Let's Create Reproducible Reports

- All of the work you do in R can be performed in reproducible and accountable ways.
- You can produce all of the forms of content – tables, figures, full written reports, slideshows, and dynamic applications – using R alone.
- Every document you produce will give you additional opportunities to improve the process by which you generate results.

Reproducible Reporting in R

- **RMarkdown** is a package and a reporting format.
- It builds on earlier efforts (**Sweave**, **knitr**) to make user-friendly, reproducible reports.
- Analyses and reporting are **interwoven** in one document. There will be no unaccountable means of porting the results from one program into the written report in another program. Your code includes your exposition, and the final result of your program is a fully written, polished report.

Reproducibility Basics

Example: A simulated file of asthma data with 1 million patients.

```
dat <- fread("simulated asthma data.csv")
geo.group <- "Urban"
mean.ER.1yr.Urban <- round(x = dat[geog == geo.group, mean(ER_1yr)], digits = 2)
sd.ER.1yr.Urban <- round(x = dat[geog == geo.group, sd(ER_1yr)], digits = 2)
n <- dat[geog == geo.group, .N]
```

For patients who live in urban areas, the mean number of visits to the ER in the first year was 0.82 with a standard deviation of 1.06 on a sample size of 499311.

How I Actually Wrote This

````
## Reproducibility Basics

Example:  The asthma data.

```{r asthma_read_in, echo=TRUE}
dat <- fread("simulated asthma data.csv")
geo.group <- "Urban"
mean.ER.1yr.Urban <- round(x = dat[geog == geo.group, mean(ER_1yr)], digits = 2)
sd.ER.1yr.Urban <- round(x = dat[geog == geo.group, sd(ER_1yr)], digits = 2)
n <- dat[geog == geo.group, .N]
```

For patients who live in urban areas, the mean number of visits to the ER in the first
year was `r mean.ER.1yr.Urban` with a standard deviation of `r sd.ER.1yr.Urban` on a
sample size of `r n`.

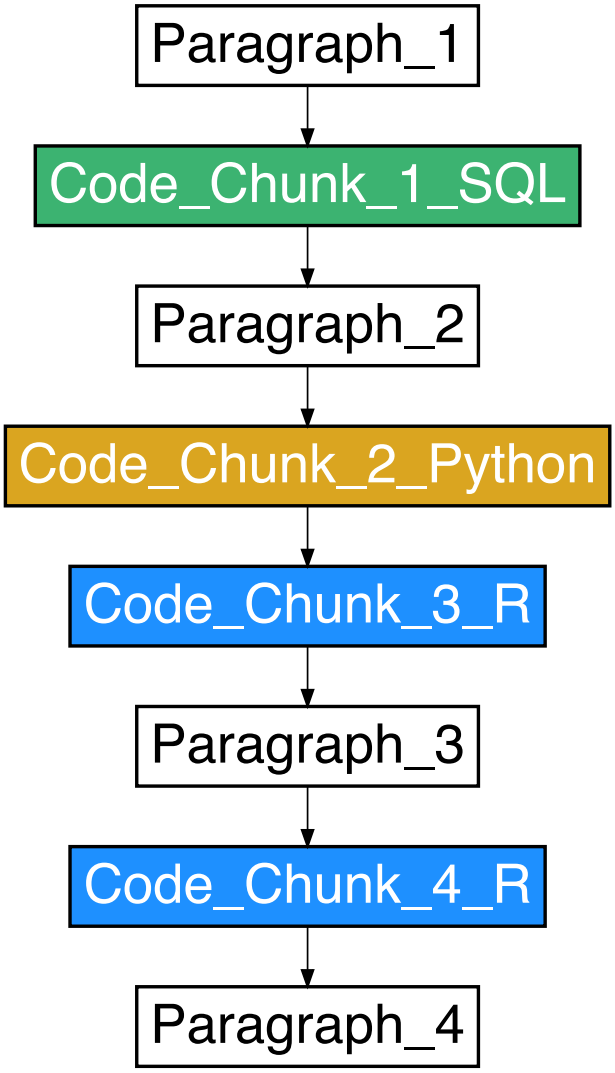
## How I Actually Wrote This
````

- The screenshot above shows a snippet of the actual RMarkdown code used to generate the previous slide.

Reproducibility

- Every number, table, figure, and extracted piece of text can be directly traced to your analysis.
- Source code chunks can be displayed in the report by setting a parameter.
- Notice that I produced the report on the mean, standard deviation, and sample size without ever writing the numbers myself. They were transcribed directly from the calculations on the data.
- **Better yet**, what would happen if the data changed? Perhaps we collected more samples and need to update the report. So long as the structure of the data remains constant, one can **feed in the new data** and **automatically re-calculate all of the reported figures**. There is no need to manually do this work.
- Both **your calculations and your report** can be simultaneously updated.

The Anatomy of a Report



Advantages of Reproducible Reporting

- Your code serves as the report's paper trail.
- There is no disconnection between your calculations and your report.
- The risk of transcription and versioning errors is greatly reduced.
- You have automatic updates when your calculations or data change.

Reports are Complex

- There are many calculated figures, tables, graphics, etc.
- Making a change in one place – e.g. by modifying the inclusion criteria for the study – can mean updating **every single piece of content** downstream.
- The last thing you want to do is many rounds of recalculation and manual review.

RMarkdown: Not Just for R

Program in any combination of these languages:

- R, SQL, Python
- Javascript, CSS
- Bash, Rcpp, Stan

Output to any of these formats:

- Word
- PDF
- HTML
- This slideshow: in PDF, HTML, or Power Point formats.

Run it all in RStudio with R in the background.

Creating Applications

Produce a full range of content:

- Websites
- GUIs
- Dashboards
- Dynamic, reactive content
- Animations

RMarkdown builds on earlier designs (**Shiny**) in a more user-friendly way. It is designed for reproducible reporting.

GUIs

Produce a full range of web elements:

- Sliders
- Dropdowns
- Checkboxes
- Radio Buttons
- Inputs for text, numbers, files, and passwords

RMarkdown provides excellent opportunities to create customized, interactive reports.

Examples


US Ski Team
Data
Gender
Follow-Up
Attrition
Development
Performance
Model
Select Data:
Slalom
Select Color:
black
Legend: X coordinate
25
Legend: Y coordinate
300
Show sample size
Select Minimum Sample Size
Performance Percentile
Text Magnifier: Sample Size
Slalom Points
Average for all Racers
90th Percentile
Points
Sample Sizes below
206 2199 4984 5368 4805 4187 2672 1802 1373 1088 667 356 214 121 75 52 45 47 30 24
Age

Getting Started with RMarkdown

- Great templates are available in RStudio.
- RMarkdown's cheat sheet is available at: https://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf
- Then, just start writing your report.

Code Chunks

````
## Reproducibility Basics

Example:  The asthma data.

```{r asthma_read_in, echo=TRUE}
dat <- fread("simulated asthma data.csv")
geo.group <- "Urban"
mean.ER.1yr.Urban <- round(x = dat[geog == geo.group, mean(ER_1yr)], digits = 2)
sd.ER.1yr.Urban <- round(x = dat[geog == geo.group, sd(ER_1yr)], digits = 2)
n <- dat[geog == geo.group, .N]
```

For patients who live in urban areas, the mean number of visits to the ER in the first
year was `r mean.ER.1yr.Urban` with a standard deviation of `r sd.ER.1yr.Urban` on a
sample size of `r n`.

## How I Actually Wrote This
````

- All code is within the accent grave characters.
- Within a code chunk, the program operates as if it were a typical piece of R code.
- Code chunks can be arbitrarily long. You could choose to write all of your code together or break it into small pieces.

Delineating Code Chunks

```
## Reproducibility Basics

Example:  The asthma data.

```{r asthma_read_in, echo=TRUE}
dat <- fread("simulated asthma data.csv")
geo.group <- "Urban"
mean.ER.1yr.Urban <- round(x = dat[geog == geo.group, mean(ER_1yr)], digits = 2)
sd.ER.1yr.Urban <- round(x = dat[geog == geo.group, sd(ER_1yr)], digits = 2)
n <- dat[geog == geo.group, .N]
```

For patients who live in urban areas, the mean number of visits to the ER in the first
year was `r mean.ER.1yr.Urban` with a standard deviation of `r sd.ER.1yr.Urban` on a
sample size of `r n`.

How I Actually Wrote This
```

- Every code chunk **begins and ends** with triple accent grave marks.
- The beginning and ending marks must be on separate lines.
- All R code within a chunk must be on lines in between the beginning and end. You cannot put code on the same line as the markings for the beginning or ending of a chunk!

Code Chunks: Language and Identifier

```
## Reproducibility Basics

Example:  The asthma data.

```{r asthma_read_in  echo=TRUE}
dat <- fread("simulated asthma data.csv")
geo.group <- "Urban"
mean.ER.1yr.Urban <- round(x = dat[geog == geo.group, mean(ER_1yr)], digits = 2)
sd.ER.1yr.Urban <- round(x = dat[geog == geo.group, sd(ER_1yr)], digits = 2)
n <- dat[geog == geo.group, .N]
```

For patients who live in urban areas, the mean number of visits to the ER in the first
year was `r mean.ER.1yr.Urban` with a standard deviation of `r sd.ER.1yr.Urban` on a
sample size of `r n`.

How I Actually Wrote This
```

- After the **beginning** with the triple accent grave marks, the code chunk **must include curly braces and specify a language**
- **{r }** would be a minimally sufficient specification for code that will run in R. This portion is marked with the red rectangle above.
- An optional **identifier** (like **asthma_read_in**, marked with the blue rectangle in the image above) may be specified to provide a name for the code chunk. This identifier must be unique relate to the identifiers in all of the code chunks in the RMarkdown file. If unspecified, the code chunk will run just fine. However, for the

organization of your code and the ease of debugging, we recommend including an identifier.

Code Chunks – Parameters

- Then there are many parameters that can be specified in the top line of a code chunk.
- Each parameter specifies a certain functionality of the chunk: how it runs, what kind of output is displayed, etc.
- Each parameter has a **default** value so that code chunks can be specified as minimally as possible when desired.

Code Chunks – The echo Parameter

````
## Reproducibility Basics

Example:  The asthma data.

```{r asthma_read_in, echo=TRUE}
dat <- fread("simulated asthma data.csv")
geo.group <- "Urban"
mean.ER.1yr.Urban <- round(x = dat[geog == geo.group, mean(ER_1yr)], digits = 2)
sd.ER.1yr.Urban <- round(x = dat[geog == geo.group, sd(ER_1yr)], digits = 2)
n <- dat[geog == geo.group, .N]
```

For patients who live in urban areas, the mean number of visits to the ER in the first
year was `r mean.ER.1yr.Urban` with a standard deviation of `r sd.ER.1yr.Urban` on a
sample size of `r n`.

## How I Actually Wrote This
````

- **echo** specifies whether the chunk's **source code** (the code you write within the chunk) should be displayed in the output that is produced when an RMarkdown file is compiled.
- The possible values are **echo = TRUE** or **echo = FALSE**
- Essentially all of the code you've seen in the lecture notes was produced using **echo = TRUE**. However, when a report does not call for showing the code that produced the results (e.g. a traditional report), it can be quite useful to specify **echo = FALSE** for all or most of the code chunks.

Examples of the echo Parameter:

- The following code and output was produced from a code chunk with **echo = TRUE**:

```
x <- 3
y <- 5 * x
this.dat <- data.table(x, y)
datatable(data = this.dat, rownames = FALSE)
```

Show 10 entries Search:

x	y
3	15

Showing 1 to 1 of 1 entries Previous 1 Next

- Then, the following table was produced **from exactly the same code** as the chunk above, but this time in a chunk with **echo = FALSE**:

Show 10 entries Search:

x	y
3	15

Showing 1 to 1 of 1 entries Previous 1 Next

Code Chunks – The eval Parameter

- The **eval** parameter indicates whether or not a code chunk should be **evaluated**.
- The possible values are **eval = TRUE** and **eval = FALSE**.
- When **eval = FALSE**, the code within the chunk **will not run at all**. The included statements will not be run in the console. Any variables defined therein will not be created, no values will be changed, and no output will be produced.

Examples of the eval Parameter

- The following code chunk has **echo = TRUE** and **eval = TRUE**:

```
x <- 3
y <- 5 * x
this.dat <- data.table(x, y)
datatable(data = this.dat, rownames = FALSE)
```

Show 10 entries Search:

x	y
3	15

Showing 1 to 1 of 1 entries Previous 1 Next

- Then, I will include the same code chunk below, with **echo = TRUE** and **eval = FALSE**:

```
x <- 3
y <- 5 * x
this.dat <- data.table(x, y)
datatable(data = this.dat, rownames = FALSE)
```

Think of the Permutations:

Show 10 entries Search:

echo	eval	output
TRUE	TRUE	Code and Results
TRUE	FALSE	Code without Results
FALSE	TRUE	Results without Code
FALSE	FALSE	No Code or Results

Showing 1 to 4 of 4 entries Previous 1 Next

- Which combination did I use in the code chunk for this slide?
- Multiple parameters can be specified with comma separation. Within the curly braces of the first line, you might write: **r the_identifier, echo = TRUE, eval = FALSE, warning = TRUE**.

Code Chunks – More Parameters

- A broader description of the available parameters is included in RMarkdown's cheat sheet: https://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf

A few other notable parameters include:

- **warning**: should coding warnings be included in the output of the report (TRUE) or be suppressed (FALSE)?

- **error**: should the generation of error messages be displayed in the output of the report (TRUE) or abort the rendering process (FALSE)?

- **results**: should the results of the code chunk – the values that would be displayed by R's console – be displayed ('markup'), passed through ('asis'), hidden ('hide'), or only displayed after all of the code of the chunk is displayed ('hold')?

- All parameters are added after the **identifier** of a code chunk in the first line with commas as separators

I also frequently use parameters to specify the formatting of images.

Setting Defaults for the Parameters of Code Chunks

- You may decide that the default values are not your preferred defaults. If so, it would be nice to reset the defaults a single time, ideally toward the top of your document.

- The following code is one example of how to do so:

```
library(knitr)
opts_chunk$set(echo = TRUE, comment = "", warning = FALSE, message = FALSE,
    tidy.opts = list(width.cutoff = 55), tidy = TRUE)
```

- Then, a single change can be used change every code chunk's parameters – or at least those that do not directly specify a value.

Strategies for Code Chunk Parameters

- I like to set an initial default for the parameters, as seen in the previous slide.
- Then, for particularly important chunks, I will directly specify some parameters.
- Quite often, for particularly time-consuming code chunks, I will set eval = FALSE on them while developing other portions of the code. That way, I don't have to wait for the code to run a long period of time to test each new code chunk I subsequently develop. We will return to this point later on.

Code in the Exposition

```
## Reproducibility Basics

Example:  The asthma data.

```{r asthma_read_in, echo=TRUE}
dat <- fread("simulated asthma data.csv")
geo.group <- "Urban"
mean.ER.1yr.Urban <- round(x = dat[geog == geo.group, mean(ER_1yr)], digits = 2)
sd.ER.1yr.Urban <- round(x = dat[geog == geo.group, sd(ER_1yr)], digits = 2)
n <- dat[geog == geo.group, .N]
```

For patients who live in urban areas, the mean number of visits to the ER in the first
year was `r mean.ER.1yr.Urban` with a standard deviation of `r sd.ER.1yr.Urban` on a
sample size of `r n`.

How I Actually Wrote This
```

- Then additional code an be referenced **outside of the code chunks**.
- Each reference is delineated by **single accent grave marks** at the beginning and end.
- After that, the language must be specified, and then a calculation is performed.

How Complex Can Code in the Exposition Be? Should It Be?

- In my experiences, simple calculations like displaying a previously calculated variable always work effectively.
- More complex calculations can sometimes lead to **no output whatsoever**; it is not currently clear to me if there is a limit on what can be computed or if some other bug is generated.
- In either case, **the simplest code in the exposition** is likely also **the most reproducible code**. The exposition of your report is a place to make reference to a specific value. Any complex calculation would be better placed within a code chunk. That will allow for multiple lines of code, proper debugging, and good style – all without interfering with the flow of your written report!

The Outlines of a Report

- Sections of the report can be created with different numbers of hash signs.
- Everything outside of a code chunk is part of the report.
- The code chunks and the exposition can be woven together in any manner that makes the most sense for you. I like to write my reports by first implementing the data cleaning steps. Then I move to the written introduction. Each subsequent piece of analysis is placed shortly before the written description for the greatest correspondence between the code and the results.

Running Your Code

- You can press the Knit/Run button in RStudio.
- You can also run an RMarkdown file in .Rmd format using the **render** function in the RMarkdown library. This is especially useful for having more control over the name of the output file or being able to run multiple RMarkdown files from the same data.
- Then you can view the code's output file.

Advantages of RMarkdown

- Interweave different programming languages
- Fast and intuitive development
- Easy to create prototypes – and perhaps products!
- Reporting is fully connected to the programming environment
- Users only have to press one button to get started

Now You're Ready To Start Writing Reports!

- Start working with the existing templates.
- For the homework assignments, a starter document will be provided.
- Additionally, find other ways to practice using RMarkdown.

RMarkdown Templates

- You can use the templates provided for the homework assignments as a guide to writing your code for the assignment.
- Subsequent documents you create can be generated by modifying this template.
- There are also a variety of default templates available within RStudio, which help you produce documents in HTML, Word, and PDF format. Note that PDF outputs require having a version of the LaTeX programming language installed.
- There are additional templates, such as the **prettydoc** library, that allow for alternative aesthetics in HTML reports.

Advanced Strategies for RMarkdown

- Set up reports that other members of your team – including those who don't program in R – can easily run.
- Allow the user to specify the limited parameters that change from iteration to iteration of your report.
- Use the **render** (rmarkdown) and **runApp** (shiny) functions to build multiple reports from the same set of data (and minimize cleaning).

Let's explore each of these issues further

Reports for Other Users

- Your work can expand the capacity of other members of your team to use information effectively.
- This is best enabled **when they can run your reports** with minimal guidance.
- The simplest approach is to set up an RMarkdown report or application that they can run with the click of a button. Other users may simply read the output files of your reports in a conventional way.

Specifying Report-Level Parameters

- You might use RMarkdown to automate routines, like producing monthly reports based on the most updated figures.

- You will likely need to set up a structure for your files and directories. This could mean copying the RMarkdown files (and any source files like recent data reports) into a separate directory for each version of the report. If the source code remains fixed, then it's possible that a single version of the code can be used, while the output of each new version is written to a separate file in an organized directory.

- Updated reports may require the user to specify **parameters** that can be fed into the report.

Steps for Using Report-Level Parameters

- Step 1: Write an RMarkdown file that generates your report.
- Step 2: Make the RMarkdown file dependent on certain parameters that are contained in a list object called **params**.
- Step 3: In a separate file (e.g. a traditional .R file of R code), allow the user to specify the **params** object and **render** the report (or run the application).

Example: User-Specified Report-Level Parameters

- Imagine the following .R file:

```
## Set the working directory to the Source File Location.
library(rmarkdown)
report.period <- "Q1 2019"
beginning.date <- "2019-01-01"
ending.date <- "2019-03-31"
params <- list(report.period = report.period, beginning.date = beginning.date,
    ending.date = ending.date)
output_file <- sprintf("Quarterly Report -- %s.html", report.period)
render(input = "Quarterly Report.Rmd", output_file = output_file,
    params = params)
```

- Assuming that the **Quarterly Report.Rmd** will be able to access the appropriate data, this approach greatly simplifies the generation of each Quarterly Report.

- The RMarkdown file will have to use the parameters appropriately – e.g. by filtering the data to extract information in the relevant period of time.

- A user of this report would only have to specify the values of **report.period**, **beginning.date**, and **ending.date**. Then, they would have to set the working directory (occasionally the most challenging step for a novice!) and run this source code. Most members of your team can be trained to do exactly that.

A Major Advantage of the render Function

- Running the previous slide's code will compile the RMarkdown file, generate the output, and **keep all of the calculations in memory**.
- By contrast, the more typical method of processing an RMarkdown file (with the Knit button in RStudio) will not store any of the calculations in memory.
- With the calculations in memory, you have enhanced capabilities to efficiently generate more results.

Strategic Use of render

- You can create multiple reports – processing multiple RMarkdown files – without having to reproduce all of the cleaning steps:

```
render(input = "Quarterly Report.Rmd", output_file = output_file,
    params = params)
output_file_executive_summary <- sprintf("Quarterly Report -- %s -- Executive Summary.html",
    report.period)
render(input = "Quarterly Report -- Executive Summary.Rmd",
    output_file = output_file_executive_summary, params = params)
```

- You can re-run the previous RMarkdown file after fixing some errors – all without reloading the data. To do this, selectively reset the **eval** parameter for any code chunk that does not need to be loaded. If you set up your code properly, this can all be done by changing the defaults or through a find and replace operation.

- With the above approach, I successfully avoided reloading enormous amounts of data from the database (a 30-minute computation) while editing my reports.

Enormous Flexibility and Capacity

- Because of these tremendous capabilities, RMarkdown has become a unifying force for the work that I do.
- Virtually any project can be handled by producing a report, slideshow, or dynamic application in RMarkdown.
- Better yet, this paradigm fully enables us to create this content in a manner that maintains high standards of reproducibility, transparency, and accountability.

Time for a Break

There's More to Reproducibility, Though

Writing a reproducible report with a paradigm like RMarkdown can be very effective. However, the scope of reproducibility encompasses other processes in model development. It's good to give some thought to these additional issues, which include:

- Model Change Management
- Version Control
- Integrating the best practices of coding and computer science to facilitate greater readability and mitigate sources of error.

Let's discuss each of these topics in greater detail.

Model Change Management

- Every analysis is subject to revision. An iterative process of development and review is the best way to create good results.

- However, changing your code over time – and potentially a long period of time at that – is a major source of error. Adding 5 lines of code in the middle of the report can have major consequences if it disrupts the logical connection of the other elements. Historically, this has been a challenge in my own projects, and I frequently see it in the code of others.

- Model Change Management typically refers to an oversight process to ensure that changes are produced in a manner that maintains the accuracy of the analysis. This also requires documentation of the changes and investigations into the validity of the results.

Version Control

- When you make changes to your analyses, it's quite common to simply overwrite your previous files. The old version makes way for the new one.
- But what if you need to refer back to the earlier versions of reports? What if the new versions contain errors? Overwriting seems like a bad strategy.
- Version Control is a process that maintains copies of each version of a report.

Options for Version Control

- **Light**: You can give your files different names or even save them in different directories (Version 1, Version 2, etc.). Ideally each version would include some documentation on the current state of development and the changes that were implemented from the previous version.

- **Medium**: Many technical platforms for file storage (e.g. cloud hard drives) maintain a record of the versions of files. If your new version has some major mistake, it is possible (though not always easy) to retrieve one of the earlier versions. You can certainly give files separate names to combine the Light version as well.

- **Best**: Certain technical development platforms (e.g. Github) provide a fully automated framework for maintaining versions of files and code. Changes are logged in a branching framework that allows for easy review of how versions were developed.

Implementing version control is a recommended practice for all engineering environments that create production-level code.

Full Reproducibilty Requires Great Coding Designs

- All of the best practices of computer science, like **organizing your code** into logical sections, **writing functions** to reduce the amount of code required, **avoiding hard-coded constants** will only **enhance the reproducibility of your work**.

- Reproducibility's goal of improving the accuracy and accountability of an analysis is best realized when the code is designed to minimize the potential for errors.

- Optimal designs for code will also enhance your ability to iteratively develop your work without introducing as many opportunities for error. Model change management is best facilitated by coding practices that are minimally disruptive.

What's Left Unsaid in All of This

- It's easy to appreciate the reproducibility and designs of the final, polished version of a report.
- It's far more difficult to get there.
- The main reason for this is that **coding does not follow a linear process**. Each new step may require revisions to the earlier portions of code. Constant variables and functions are added far upstream. Earlier pieces are re-designed now that a function has been written to handle the second observed case.

With experiene, you'll increasingly recognize the challenges of this work and proactively design solutions. However, this is an ongoing process.

So Let's Build a Reproducible Report!

Case Study: A Real World Clinical Trial for HIV Medicines

We now have the tools and basic methods of analysis to do some real work in health care. So let's take a look at a real clinical trial of a medicine that is used to treat HIV.

Overview of HIV and AIDS

- Patients with Human Immunodeficiency Virus (HIV) patients have a deteriorating immune system. Without treatment, they will continue to lose white blood cells, which will further deteriorate the body's ability to fight off diseases.
- The CD4 count is the number of immune cells per milliliter unit of blood.
- Patients with HIV develop Acquired Immune Deficiency Syndrome (AIDS) when the CD4 count reaches zero.

Treating HIV

- A variety of drugs have been developed, such as nucleoside reverse transcriptase inhibitors (NRTIs) and anti-retroviral therapies.
- In the mid 1990's, an anti-retroviral called a protease inhibitor was introduced.
- A question: Did the new drug help patients when used in combination with the earlier medicines?

A Clinical Trial

Gulick et al. conducted a randomized clinical trial:

^ a b "Treatment with Indinavir, Zidovudine, and Lamivudine in Adults with Human Immunodeficiency Virus Infection and Prior Antiretroviral Therapy". *New England Journal of Medicine*. **337** (11): 734–739. 1997. doi:10.1056/NEJM199709113371102.

- Patient Population: Patients with a CD4 Count below 200.
- Control Group: 2 Standard NRTIs
- Treatment Group: 2 Standard NRTIs plus a protease Inhibitor, the "3-Drug Cocktail".

Measuring Outcomes

- Study 1: Death
- Study 2: Death or a diagnosis of AIDS, whichever occurs first.
- The overall time to these events was measured. For patients without an event, the time to the end of the study was used.

The 3-Drug Cocktail Trial's Data

- Source: Wiley.com
- A copy of the data and a data's dictionary are available on the course's webpage.

Variable	Name	Description	Codes/Values
1	id	Identification Code	1–1156
2	time	Time to AIDS diagnosis or death	Days
3	censor	Event indicator for AIDS defining diagnosis or death	1 = AIDS defining diagnosis or death 0 = Otherwise
4	time_d	Time to death	Days
5	censor_d	Event indicator for death (only)	1 = Death 0 = Otherwise
6	tx	Treatment indicator	1 = Treatment includes IDV 0 = Control group (treatment regime without IDV)
7	txgrp	Treatment group indicator	1 = ZDV + 3TC 2 = ZDV + 3TC + IDV 3 = d4T + 3TC 4 = d4T + 3TC + IDV
8	strat2	CD4 stratum at screening	0 = CD4 <= 50 1 = CD4 > 50
9	sex	Sex	1 = Male 2 = Female
10	raceth	Race/Ethnicity	1 = White Non-Hispanic 2 = Black Non-Hispanic 3 = Hispanic (regardless of race) 4 = Asian, Pacific Islander 5 = American Indian, Alaskan Native 6 = Other/unknown
11	ivdrug	IV drug use history	1 = Never 2 = Currently 3 = Previously
12	hemophil	Hemophiliac	1 = Yes 0 = No
13	karnof	Karnofsky Performance Scale	100 = Normal;no complaint no evidence of disease 90 = Normal activity possible; minor signs/symptoms of disease 80 = Normal activity with effort; some signs/symptoms of disease 70 = Cares for self; normal activity/ active work not possible
14	cd4	Baseline CD4 count (derived from multiple measurements)	Cells/milliliter
15	priorzdv	Months of prior ZDV use	Months
16	age	Age at Enrollment	Years

Getting Started with the Analysis

- We will be using the file **Analysis of HIV Data - Template.Rmd** from the website as a template for the work.
- The file sets up some of the preliminary pieces of the work, and the rest will be filled in as we go.
- The eventual result will be in a file called **Analysis of HIV Data.Rmd** on the website.

Preliminaries for the Report

- Some of the initial work involves setting up the file with the title, author, date, and output settings. This file is set up using the **prettydoc** template to produce an HTML output.

- Then we can set up some code chunks to load the libraries and set the options for the chunks:

```
---
title: "Anaysis of the Clinical Trial for the 3-Drug Cocktail to
Treat HIV"
author: "My Name"
date: "`r Sys.Date()`"
output:
  prettydoc::html_pretty:
  theme: cayman
highlight: github
---
```

```
```{r libraries, echo = FALSE, warning=FALSE, message=FALSE}
library(data.table)
library(rmarkdown)
library(knitr)
library(DT)
```
```

```
```{r chunk_options, echo = FALSE, tidy = TRUE}
opts_chunk$set(echo = FALSE, comment="", warning = FALSE, message =
FALSE, tidy.opts=list(width.cutoff=55), tidy = TRUE)
```
```

Typical Code Chunks

- As you do more analyses, eventually you'll create some organizing principles for what your typical project entails.
- I like to set up separate code chunks to call libraries, run source files, set constant variables, write functions, read in the data, and then clean the data.

````
```{r source_files}

```

```{r constants}

```

```{r functions}

```

```{r read_data}

```

```{r clean_data}

```
````

Typical Sections

- Then you can get into the exposition of the report.
- The sections of written materials are interweaved with code chunks from here on out.

```
## Introduction

We are working with the clinical trial data on the 3-Drug Cocktail,
which was used to treat HIV.

## Analysis

```{r analyze_data}

```
```

Now Let's Build the Analysis

- Every piece of subsequent code added here will go into the **Analysis of HIV Data.Rmd** file on the website.
- For each line of code, I'll provide comments about **which code chunk** it will be included in.
- This will necessarily be a simple analysis. However, it will give you an idea about the process of structuring your code and report in tandem.

Reading in the Data

```
## code chunk: constants
file.hiv.data <- "AIDS data.csv"
## code chunk: read_data
dat <- fread(input = file.hiv.data)

## Entered in console but not in the file directly:
dim(dat)
```

```
[1] 1151   16
```

```
names(dat)
```

```
 [1] "id"       "time"     "censor"   "time_d"   "censor_d" "tx"
 [7] "txgrp"    "strat2"   "sex"      "raceth"   "ivdrug"   "hemophil"
[13] "karnof"   "cd4"      "priorzdv" "age"
```

Adding Variable Names

- For each variable that is used in the analysis, it can help to set a constant with its name. Then, any subsequent analysis of the variable within a data.table can be performed using **get** or **eval** operations on the constant variable's name. This way, if the name is changed in another data file, the analysis can be completed with one change in the **constants** code chunk.

```
## code chunk: constants
id.name <- "id"
time.aids.or.death.name <- "time"
time.death.name <- "time_d"
outcome.aids.or.death.name <- "censor"
outcome.death.name <- "censor_d"
treatment.name <- "tx"
age.name <- "age"
sex.name <- "sex"
```

Understanding the Data

- Using the console, I investigated some elements of the variables. To do this, I **added a code chunk** called **data_investigations**. This was set to **eval = FALSE** because it would not be run in the final report.

```
## code chunk: data_investigations
dat[, .(n = .N, `Number of Unique IDs` = length(unique(get(id.name))))]
```

```
      n Number of Unique IDs
1: 1151                 1151
```

```
dat[, summary(get(time.aids.or.death.name))]
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0   174.0   257.0   230.2   300.0   364.0
```

```
dat[, summary(get(time.death.name))]
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0   194.5   265.0   242.3   306.0   364.0
```

```
dat[, .N, keyby = outcome.aids.or.death.name]
```

```
   censor    N
1:      0 1055
2:      1   96
```

Understanding the Data, Part II

```
## code chunk: data_investigations
dat[, .N, keyby = outcome.death.name]
```

```
   censor_d    N
1:        0 1125
2:        1   26
```

```
dat[, .N, keyby = treatment.name]
```

```
   tx   N
1:  0 577
2:  1 574
```

```
dat[, summary(get(age.name))]
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  15.00   33.00   38.00   38.65   44.00   73.00
```

```
dat[, .N, keyby = sex.name]
```

```
   sex   N
1:   1 951
2:   2 200
```

A Nice Feature

- Most reports don't include the steps used to investigate the data.
- Because of this, the report ends up having the feeling that it magically came together. How did we know that a certain variable was categorical, or that another one required cleaning?
- At the same time, these investigative steps are usually not considered important details to share in the final report.
- Using **eval = FALSE** provides a tremendous opportunity. You can include all of the code that was used to investigate the data without necessarily having to run it or print the output in the final version. In this way, **you can provide a paper trail for your investigations** even if the final output doesn't depend on them directly.

Plans for Data Cleaning

- Based upon the initial investigations, it appears that the sex of the patients is cryptically coded (in categories 1 and 2). We will clean the data up to make the categories more apparent.
- The other variables investigated here appear to be straightforward enough for our purposes.
- There are additional variables in the data set that would require cleaning. This can be an exercise for investigation and practice.

Data Cleaning

- Based on the Data Dictionary, Male is Category 1 and Female is Category 2.

- We will create a new column with the following logic:

```
## code chunk:  constants
male.name <- "Male"
sex.category.male <- 1
sex.category.female <- 2
## code chunk:  data_cleaning
dat[, eval(male.name) := 1*(get(sex.name) == sex.category.male)]
## Then, in the Console, check the values with this line:
dat[, .N, keyby = c(sex.name, male.name)]
```

```
   sex Male   N
1:   1    1 951
2:   2    0 200
```

An Overall Plan for the Analysis

- Count the overall sample size and the size of each cohort based on treatments.
- For each treatment group, **summarize the demographic variables**. Compute a few summary figures (the mean and standard deviation for continuous variables or the count and percentage for categorical variables) of each variable in each group.
- For each treatment group, **compare the percentage** of outcomes and show the mean and standard deviations for the times to each outcome.

Sample Sizes

```
## code chunk: constants
received.treatment <- 1
no.treatment <- 0
## code chunk: analyze_data
n <- dat[, .N]
n.treatment <- dat[get(treatment.name) == received.treatment,
    .N]
n.untreated <- dat[get(treatment.name) == no.treatment,
    .N]
```

Reporting Sample Sizes

- Now we can add a sentence to the exposition in the **Analysis** section underneath the **analyze_data** code chunk.

```
## Analysis
```

```
```{r analyze_data}
n <- dat[, .N]
n.treatment <- dat[get(treatment.name) == received.treatment, .N]
n.untreated <- dat[get(treatment.name) == no.treatment, .N]
```

The sample included a total of `r n` patients, including `r
n.treatment` patients randomized to receive the 3-Drug Cocktail and
`r n.untreated` randomized to receive only the 2-Drug Cocktail.
```

- **Output**: The sample included a total of 1151 patients, including 574 patients randomized to receive the 3-Drug Cocktail and 577 randomized to receive only the 2-Drug Cocktail.

Splitting the analyze_data Code Chunk

- Since our plan of analysis has a number of well-delineated steps, it would make sense to split the code up into separate chunks.
- Perhaps we should have renamed the **analyze_data** chunk, to, say, **compute_sample_size**. But, since this is how we started off, we'll maintain it for now.
- The subsequent code chunks will come after the exposition for the sample sizes.

Summarizing Demographics by Treatment Group: Male

Let's calculate the number and percentage of males in each treatment group:

```
## code chunk: functions
count.and.percentage <- function(dat, variable.name, by.names,
    na.rm = TRUE) {
    require(data.table)
    setDT(dat)

    tab <- dat[, .(C1 = sum(get(variable.name), na.rm = na.rm),
        N = .N, P1 = 100 * mean(get(variable.name), na.rm = na.rm)),
        by = by.names]
    setnames(x = tab, old = c("C1", "P1"), new = c(sprintf("%s: Count",
        variable.name), sprintf("%s: Percentage", variable.name)))
    return(tab)
}
## code chunk: summarize_demographics_by_treatment
tab.male <- count.and.percentage(dat = dat, variable.name = male.name,
    by.names = treatment.name)
datatable(data = tab.male, rownames = FALSE)
```

Show 10 entries Search:

tx	Male: Count	N	Male: Percentage
0	483	577	83.7088388214905
1	468	574	81.5331010452962

Showing 1 to 2 of 2 entries Previous 1 Next

What About Rounding?

- Let's add our **round.numerics** function:

```
## code chunk: constants
one.digit <- 1
## code chunk: functions
round.numerics <- function(x, digits) {
    if (is.numeric(x)) {
        x <- round(x = x, digits = digits)
    }
    return(x)
}
## code chunk: summarize_demographics_by_treatment
datatable(data = tab.male[, lapply(X = .SD, FUN = "round.numerics",
    digits = one.digit)], rownames = FALSE)
```

Show 10 entries Search:

tx	Male: Count	N	Male: Percentage
0	483	577	83.7
1	468	574	81.5

Showing 1 to 2 of 2 entries Previous 1 Next

Summarizing Age by Treatment Group

- Let's compute the mean and standard deviation of the age of the patients in each group:

```
## code chunk: functions
mean.and.sd <- function(dat, variable.name, by.names, na.rm = TRUE) {
    require(data.table)
    setDT(dat)

    tab <- dat[, .(M1 = mean(get(variable.name), na.rm = na.rm),
        S1 = sd(get(variable.name), na.rm = na.rm)), by = by.names]
    setnames(x = tab, old = c("M1", "S1"), new = c(sprintf("%s: Mean",
        variable.name), sprintf("%s: SD", variable.name)))
    return(tab)
}
## code chunk: summarize_demographics_by_treatment
tab.age <- mean.and.sd(dat = dat, variable.name = age.name,
    by.names = treatment.name)
datatable(data = tab.age[, lapply(X = .SD, FUN = "round.numerics",
    digits = one.digit)], rownames = FALSE)
```

Show 10 entries Search:

tx	age: Mean	age: SD
0	38.6	8.8
1	38.7	8.8

Showing 1 to 2 of 2 entries Previous 1 Next

Adding Some Exposition

- The Rmarkdown file included the following lines: The above tables show the summary measures of sex and age. The percentage of males and mean age are very similar in each group. This is in line with the expectations of a randomized trial.

The other points below were not included, but they are nonetheless good to think about in analyzing the data from a randomized trial:

- This is of course a very basic analysis. Comparing these percentages or means using statistical tests can further establish the similarity of the cohorts in terms of their baseline factors.

- Showing that randomization achieved good balance in the groups further allows us to infer the effect of the treatment on the outcomes.

Analyzing the Outcomes

- We are now ready to investigate the outcomes of the study. Let's remind ourselves about what we're measuring:

- **censor**: Whether the patient had an outcome of death or AIDS diagnosis (1) or not (0) at the end of follow-up.

- **censor_d**: Whether the patient had an outcome of death (1) or not (0) at the end of follow-up.

- **time**: For those patients with **censor = 1**, how many days after baseline did the event of death or AIDS diagnosis (whichever occurred first) happen?

- **time_d**: For those patients with **censor_d = 1**, how many days after baseline did death occur?

Percentage of AIDS or Death Outcomes by Treatment Group

Let's add a code chunk called **outcomes_percentages**:

```
## code chunk: outcomes_percentages
tab.pct.aids.or.death <- count.and.percentage(dat = dat,
    variable.name = outcome.aids.or.death.name, by.names = treatment.name)
datatable(data = tab.pct.aids.or.death[, lapply(X = .SD,
    FUN = "round.numerics", digits = one.digit)], rownames = FALSE)
```

Show 10 entries Search:

tx	censor: Count	N	censor: Percentage
0	63	577	10.9
1	33	574	5.7

Showing 1 to 2 of 2 entries Previous 1 Next

Percentage of Death Outcomes by Treatment Group

```
## code chunk: outcomes_percentages
tab.pct.death <- count.and.percentage(dat = dat, variable.name = outcome.death.name,
    by.names = treatment.name)
datatable(data = tab.pct.death[, lapply(X = .SD, FUN = "round.numerics",
    digits = one.digit)], rownames = FALSE)
```

Show 10 entries Search:

tx	censor_d: Count	N	censor_d: Percentage
0	18	577	3.1
1	8	574	1.4

Showing 1 to 2 of 2 entries Previous 1 Next

- Then add some exposition under the tables: The above tables show the percentage of patients in the treatment group with adverse outcomes of AIDS or death (composite) or death.

Mean Time to AIDS or Death Among Patients with Adverse Outcomes

Let's add a code chunk called **event_times**:

```
## code chunk: event_times
tab.time.aids.or.death <- mean.and.sd(dat = dat, variable.name = time.aids.or.death.name,
    by.names = treatment.name)
datatable(data = tab.time.aids.or.death[, lapply(X = .SD,
    FUN = "round.numerics", digits = one.digit)], rownames = FALSE)
```

Show 10 entries Search:

tx	time: Mean	time: SD
0	223.6	92
1	236.8	87.3

Showing 1 to 2 of 2 entries Previous 1 Next

Mean Time to Death Among Patients with Adverse Outcomes

```
## code chunk: event_times
tab.time.death <- mean.and.sd(dat = dat, variable.name = time.death.name,
    by.names = treatment.name)
datatable(data = tab.time.death[, lapply(X = .SD, FUN = "round.numerics",
    digits = one.digit)], rownames = FALSE)
```

Show 10 entries Search:

tx	time_d: Mean	time_d: SD
0	239.1	84.8
1	245.5	80.3

Showing 1 to 2 of 2 entries Previous 1 Next

Adding Some Conclusions

- Then, after reading over the report, I added a section for the Conclusion with some additional exposition.
- This was in the report: Based on the observed statistics, the 3-Drug Cocktail appears to have been effective in improving the outcomes for patients with HIV. A smaller percentage of these patients progressed to AIDS or Death compared to those taking the 2-Drug Cocktail. Among those with an outcome, the overall time to the event was greater for the treatment group as well, indicating lengthier survival free of the disease.
- Lots more to do: Other variables to analyze, more numbers to crunch, statistical tests, multivariate survival models, etc.

And That's a Reproducible Report

- By pressing the Knit button, the RMarkdown file can be compiled to produce a report in HTML format.
- All of the code and exposition were written in a reproducible manner. Any update to the data would automatically lead to updated calculations for all of the calculated values, tables, etc.
- Better yet, the code we produced was highly organized, efficient, reusable, and readable. It would be very simple for another data scientist to review this work, understand the calculations, and reproduce the results.