

BROOKE ANDERSON, MICHAEL LYONS, MERCEDES GONZALEZ-JUARRERO, MARCELA HENAO-TAMAYO, AND GREGORY ROBERT-SON

IMPROVING THE REPRODUCIBILITY OF EXPERIMENTAL DATA RECORDING AND PRE-PROCESSING

Contents

1	<i>Overview</i>	5
	1.1 License	6
2	<i>Experimental Data Recording</i>	9
	2.1 Separating data recording and analysis	9
	2.2 Principles and power of structured data formats	19
	2.3 The ‘tidy’ data format	33
	2.4 Designing templates for “tidy” data collection	40
	2.5 Example: Creating a template for “tidy” data collection	56
	2.6 Power of using a single structured ‘Project’ directory for storing and tracking research project files	70
	2.7 Creating ‘Project’ templates	85
	2.8 Example: Creating a ‘Project’ template	95
	2.9 Harnessing version control for transparent data recording	108
	2.10 Enhance the reproducibility of collaborative research with version control platforms	114
	2.11 Using git and GitLab to implement version control	121
3	<i>Experimental Data Preprocessing</i>	133
	3.1 Principles and benefits of scripted pre-processing of experimental data	133
	3.2 Introduction to scripted data pre-processing in R	194
	3.3 Simplify scripted pre-processing through R’s ‘tidyverse’ tools	210
	3.4 Complex data types in experimental data pre-processing	225
	3.5 Complex data types in R and Bioconductor	242

4 brooke anderson, michael lyons, mercedes gonzalez-juarrero, marcela henao-tamayo, and gregory robertson

3.6 Example: Converting from complex to ‘tidy’ data formats	288
3.7 Introduction to reproducible data pre-processing protocols	307
3.8 RMarkdown for creating reproducible data pre-processing protocols	319
3.9 Example: Creating a reproducible data pre-processing protocol	334

4 References 351

4.1 Extra quotes	351
------------------	-----

5 Bibliography 421

I

Overview

The recent NIH-Wide Strategic Plan (U.S. Department of Health and Human Services, National Institutes of Health, 2016) describes an integrative view of biology and human health that includes translational medicine, team science, and the importance of capitalizing on an exponentially growing and increasingly complex data ecosystem (U.S. Department of Health and Human Services, National Institutes of Health, 2018). Underlying this view is the need to use, share, and re-use biomedical data generated from widely varying experimental systems and researchers. Basic sources of biomedical data range from relatively small sets of measurements, such as animal body weights and bacterial cell counts that may be recorded by hand, to thousands or millions of instrument-generated data points from various imaging, -omic, and flow cytometry experiments. In either case, there is a generally common workflow that proceeds from measurement to data recording, pre-processing, analysis, and interpretation. However, in practice the distinct actions of data recording, data pre-processing, and data analysis are often merged or combined as a single entity by the researcher using commercial or open source spreadsheets, or as part of an often proprietary experimental measurement system / software combination (Figure I.1), resulting in key failure points for reproducibility at the stages of data recording and pre-processing.

It is widely known and discussed among data scientists, mathematical modelers, and statisticians (Broman and Woo, 2018; Krishnan et al., 2016) that there is frequently a need to discard, transform, and reformat various elements of the data shared with them by laboratory-based researchers, and that data is often shared in an unstructured format, increasing the risks of introducing errors through reformatting before applying more advanced computational methods. Instead, a critical need for reproducibility is for the transparent and clear sharing across research teams of: (1) raw data, directly from hand-recording or directly output from experimental equipment; (2) data that has been pre-processed as necessary (e.g., gating for flow cytometry data, feature identification for metabolomics data), saved in a consistent, structured format, and (3) a clear and repeatable description of how the pre-processed data was



Figure 1.1: Two scenarios where ‘black boxes’ of non-transparent, non-reproducible data handling exist in research data workflows at the stages of data recording and pre-processing. These create potential points of failure for reproducible research. Red arrows indicate where data is passed to other research team members, including statisticians / data analysts, often within complex or unstructured spreadsheet files.

generated from the raw data (Broman and Woo, 2018; Ellis and Leek, 2018).

To enhance data reproducibility, it is critical to create a clear separation among data recording, data pre-processing, and data analysis—breaking up commonly existing “black boxes” in data handling across the research process. Such a rigorous demarcation requires some change in the conventional understanding and use of spreadsheets and a recognition by biomedical researchers that recent advances in computer programming languages, especially the R programming language, provide user-friendly and accessible tools and concepts that can be used to extend a transparent and reproducible data workflow to the steps of data recording and pre-processing. Among our team, we have found that there are many common existing practices—including use of spreadsheets with embedded formulas that concurrently record and analyze experimental data, problematic management of project files, and reliance on proprietary, vendor-supplied point-and-click software for data pre-processing—that can interfere with the transparency, reproducibility, and efficiency of laboratory-based biomedical research projects, problems that have also been identified by others as key barriers to research reproducibility (Broman and Woo, 2018; Bryan, 2018; Ellis and Leek, 2018; Marwick et al., 2018). In these training modules, we have chosen topics that tackle barriers to reproducibility that have straightforward, easy-to-teach solutions, but which are still very common in biomedical laboratory-based research programs.

1.1 License

This book is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License, while all code in the book is under the MIT license.

Click on the **Next** button (or navigate using the links at the top of the page) to continue.

2

Experimental Data Recording

This section includes modules on:

- Module 2.1: Separating data recording and analysis
- Module 2.2: Principles and power of structured data formats
- Module 2.3: The ‘tidy’ data format
- Module 2.4: Designing templates for “tidy” data collection
- Module 2.5: Example: Creating a template for “tidy” data collection
- Module 2.6: Power of using a single structured ‘Project’ directory for storing and tracking research project files
- Module 2.7: Creating ‘Project’ templates
- Module 2.8: Example: Creating a ‘Project’ template
- Module 2.9: Harnessing version control for transparent data recording
- Module 2.10: Enhance the reproducibility of collaborative research with version control platforms
- Module 2.11: Using git and GitLab to implement version control

2.1 Separating data recording and analysis

Many biomedical laboratories currently use spreadsheet programs to jointly record, visualize, and analyze experimental data (Broman and Woo, 2018). These software tools, such as Microsoft Excel[Ref, copyright?] or Google Sheets[Ref, copyright?], provide for manual or automated entry of data into rows and columns of cells. Standard or custom formulas and other operations can be applied to the cells, and are commonly used to reformat or clean the data, calculate various statistics, and to generate simple plots; all of which are embedded as additional data entries and programming elements within the spreadsheet. While these tools greatly improved the paper worksheets on which they were originally based (Campbell-Kelly, 2007), this all-in-one practice impedes the transparency and reproducibility of both recording and analysis of the large and complex data sets that are routinely generated in life science experiments.

To improve the computational reproducibility of a research project, it is critical for biomedical researchers to learn the importance of maintaining recorded

experimental data as “read-only” files, separating data recording from any data pre-processing or data analysis steps (Broman and Woo, 2018; Marwick et al., 2018). Statisticians have outlined specific methods that a laboratory-based scientist can take to ensure that data shared in an Excel spreadsheet are shared in a reliable and reproducible way, including avoiding macros or embedded formulas, using a separate Excel file for each dataset, recording descriptions of variables in a separate code book rather than in the Excel file, avoiding the use of color of the cells to encode information, using “NA” to code missing values, avoiding spaces in column headers, and avoiding splitting or merging cells (Ellis and Leek, 2018; Broman and Woo, 2018). In this module, we will describe this common practice and will outline alternative approaches that separate the steps of data recording and data analysis.

Objectives. After this module, the trainee will be able to:

- Explain the difference between data recording and data analysis
- Understand why collecting data on spreadsheets with embedded formulas impedes reproducibility
- List alternative approaches to improve reproducibility

2.1.1 Data recording versus data analysis

History of spreadsheets.

Spreadsheets have long been an extremely popular tool for recording and analyzing data, in part because they allow people without programming experience to conduct a range of standard computations and statistical analyses through a visual interface that is more immediately user-friendly to non-programmers than programs with command line interfaces. An early target for spreadsheet programs in terms of users was business executives, and so the programs were designed to be very simple and easy to use—just one step up in complexity from crunching numbers on the back of an envelope (Campbell-Kelly, 2007). Spreadsheet programs in fact became so popular within businesses that many attribute these programs with driving the uptake of personal computers (Campbell-Kelly, 2007).

Spreadsheets were innovative and rapidly adapted in part because they allowed users to combine data recording and analysis—while previously, in business settings, any complicated data analysis task needed to be outsourced to mainframe computers and data processing teams, the initial spreadsheet program (VisiCalc) allowed one person to quickly apply and test different models or calculations on recorded data (Levy, 1984). These spreadsheet programs allowed non-programmers to engage with data, including data processing and analysis tasks, in a way that previously required programming expertise (Levy, 1984).

Use of spreadsheets.

Many scientific laboratories use spreadsheets within their data collection process, both to record data and to clean and analyze the data. Illustrative

examples can be found in surveys of over 250 biomedical researchers at the University of Washington (Anderson et al., 2007), and of neuroscience researchers at the University of Newcastle, with most respondents reporting the use of spreadsheets and other general-purpose software in their research (AlTarawneh and Thorne, 2017). A working group on bioinformatics and data-intensive science similarly found spreadsheets were the most common tool used across attendees (Barga et al., 2011).

In some cases, a spreadsheet is used solely to record data, as a simple type of database (Birch et al., 2018). However, biomedical researchers often use spreadsheets to both record and analyze experimental data (Anderson et al., 2007). In this case, data processing and analysis is implemented through the use of formulas and macros embedded within the spreadsheet. When a spreadsheet has formulas or macros within it, the spreadsheet program creates an internal record of how cells are connected through these formulas. For example, if the value in a specific cell is converted from Fahrenheit to Celsius to fill a second cell, and then that value is combined with other values in a column to calculate the mean temperature across several observations, then the spreadsheet program has internally saved how the later cells depend on the earlier ones. When you change the value recorded in a cell of a spreadsheet, the spreadsheet program queries this record and only recalculates the cells that depend on that cell. This process allows the program to quickly “react” to any change in cell inputs, immediately providing an update to all downstream calculations and analyses (Levy, 1984). Starting from the spreadsheet program Lotus 1-2-3, spreadsheet programs also included *macros*, “a single computer instruction that stands for a sequence of operations” (Creeth, 1985).

Spreadsheets have become so popular in part because so many people know how to use them, at least in basic ways, and so many people have the software on their computers that files can be shared with the virtual guarantee that everyone will be able to open the file on their own computer (Hermans et al., 2016). Spreadsheets use the visual metaphor of a traditional gridded ledger sheet (Levy, 1984), providing an interface that is easy for users to immediately understand and create a mental map of (Birch et al., 2018; Barga et al., 2011). This visually clear interface also means that spreadsheets can be printed or incorporated into other documents (Word files, PowerPoint presentations) “as-is”, as a workable and understandable table of data values. In fact, some of the most popular plug-in software packages for the early spreadsheet program Lotus 1-2-3 were programs for printing and publishing spreadsheets (Campbell-Kelly, 2007). This “What You See Is What You Get” interface was a huge advance from previous methods of data analysis for the first spreadsheet program, VisiCalc, providing a “window to the data” that was accessible to business executives and others without programming expertise (Creeth, 1985). Several surveys of researchers have found that spreadsheets were popular because of their simplicity and ease-of-use (Anderson et al., 2007; AlTarawneh and Thorne, 2017; Barga et al., 2011). By contrast, databases and scripted programming

languages can be perceived as requiring a cognitive load and lengthy training that is not worth the investment when an easier tool is available (Hermans et al., 2016; Anderson et al., 2007; Myneni and Patel, 2010; Barga et al., 2011; Topaloglou et al., 2004).

2.1.2 *Hazards of combining recording and analysis*

Raw data often lost.

One of the key tenets of ensuring that research is computationally reproducible is to always keep a copy of all raw data, as well as the steps taken to get from the raw data to a cleaned version of the data through to the results of data analysis. However, maintaining an easily accessible copy of all original raw data for a project is a common problem among biomedical researchers (Goodman et al., 2014), especially as team members move on from a laboratory group (Myneni and Patel, 2010).

The use of spreadsheets to jointly record and analyze data can contribute to this problem. Spreadsheets allow for the immediate and embedded processing of data. As a result, it may become very difficult to pull out the raw data originally recorded in a spreadsheet. At the least, the combination of raw and processed data in a spreadsheet makes it hard to identify which data points within a spreadsheet make up the raw data and which are the result of processing that raw data. One study of operational spreadsheets noted that:

“The data used in most spreadsheets is undocumented and there is no practical way to check it. Even the original developer would have difficulty checking the data.” (Powell et al., 2009)

Further, data in a spreadsheet is typically not saved as “read-only”, so it is possible for it to be accidentally overwritten: in situations where spreadsheets are shared among multiple users, original cell values can easily be accidentally written over, and it may not be clear who last changed a value, when it was changed, or why (AlTarawneh and Thorne, 2017).

Finally, many spreadsheets use a proprietary format. In the development of spreadsheet programs, this use of proprietary binary file formats helped a software program keep users, increasing barriers for a user to switch to a new program (since the new program wouldn’t be able to read their old files) (Campbell-Kelly, 2007). However, this file format may be hard to open in the future, as software changes and evolves (Michener, 2015); by comparison, plain text files should be widely accessible through general purpose tools—a text editor is a type of software available on all computers, for example—regardless of changes to proprietary software like Microsoft Excel.

Opacity of analysis steps and potential for errors.

Previous studies have found that errors are very common within spreadsheets (Hermans et al., 2016). For example, one study of 50 operational spreadsheets found that about 90% contained at least one error (Powell et al., 2009). In part, it is easier to make errors in spreadsheets and harder to catch

errors in later work with a spreadsheet because the formulas and connections between cells aren't visible when you look at the spreadsheet—they're behind the scenes (Birch et al., 2018). This makes it very hard to get a clear and complete view of the pipeline of analytic steps in data processing and analysis within a spreadsheet, or to discern how cells are connected within and across sheets of the spreadsheet. As one early article on the history of spreadsheet programs notes:

“People tend to forget that even the most elegantly crafted spreadsheet is a house of cards, ready to collapse at the first erroneous assumption. The spreadsheet that looks good but turns out to be tragically wrong is becoming a familiar phenomenon.” (Levy, 1984)

Some characteristics of spreadsheets may heighten chances for errors. These include high conditional complexity (i.e., lots of branching of data flow through if / else structures), formulas that depend on a large number of cells or that incorporate many functions (Hermans et al., 2016). Following the logical chain of spreadsheet formulas can be particularly difficult when several calculations are chained in a row (Hermans and Murphy-Hill, 2015). Very long chains of dependent formulas across spreadsheet cells may in some case requiring sketching out by hand the flow of information through the spreadsheet to understand what's going on (Nardi and Miller, 1990). The use of macros can also make it particularly hard to figure out the steps of an analysis and to diagnose and fix any bugs in those steps (Nash, 2006; Creeth, 1985). One study of spreadsheets used in real life applications noted that, “Many spreadsheets are so chaotically designed that auditing (especially of a few formulas) is extremely difficult or impossible.” (Powell et al., 2009)

In some cases, formula dependences might span across different sheets of a spreadsheet file. For the example given previously of a spreadsheet that converts temperature from one unit to another and then averages across observations, for example, the original temperature might be recorded in one sheet while the converted temperature value is calculated and shown in a second sheet. These cross-sheet dependencies can make the analysis steps even more opaque (Hermans et al., 2016), as a change in the cell value of one sheet might not be immediately visible as a change in another cell on that sheet (the same is true for spreadsheets so large that all the cells in a sheet are not concurrently visible on the screen). Other common sources of errors included incorrect references to cells inside formulas and incorrect use of formulas (Powell et al., 2009) or errors introduced through the common practice of copying and pasting when developing spreadsheets (Hermans et al., 2016).

To keep analysis steps clear, whether in scripted code or in spreadsheets or pen-and-paper calculations, it is important to document what is being done at each step and why (Goodman et al., 2014). Scripted languages allow for code comments, which are written directly into the script but not evaluated by the computer, and so can be used to document steps within the code without changing the operation of the code. Further, the program file itself often

presents a linear, step-by-step view of the pipeline, stored separated from the data itself (Creeth, 1985). Calculations done with pen-and-paper (e.g., in a laboratory notebook) can be annotated with text to document the steps. Spreadsheets, on the other hand, are often poorly documented, or documented in ways that are hard to keep track of. Before spreadsheets,

"The formulas appeared in one place and the results in another. You could see what you were getting. That cannot be said of electronic spreadsheets, which don't display the formulas that govern their calculations. As Mitch Kapor explained, with electronic spreadsheets, 'You can just randomly make formulas, all of which depend on each other. And when you look at the final results, you have no way of knowing what the rules are, unless someone tells you.' " (Levy, 1984)

Within spreadsheets, the logic and methods behind the pipeline of data processing and analysis is often not documented, or only documented with cell comments (hard to see as a whole) or in emails, not the spreadsheet file. One study that investigated a large collection of spreadsheets found that most do not include documentation explaining the logic or implementation of data processing and analysis implemented within the spreadsheet (Hermans et al., 2016). A survey of neuroscience researchers at a UK institute found that about a third of respondents included no documentation for spreadsheets used in their research laboratories (AlTarawneh and Thorne, 2017).

When spreadsheet pipelines are documented, it is often through methods that are hard to find and interpret later. One study of scientific researchers found that, when research spreadsheets were documented, it was often through "cell comments" added to specific cells in the spreadsheet, which can be hard to interpret inclusively to understand the flow and logic of a spreadsheet as a whole (AlTarawneh and Thorne, 2017). In some cases, teams discuss and document functionality and changes in spreadsheets through email chains, passing different versions of the spreadsheet file as attachments of emails with discussion of the spreadsheet in the email body. One research team investigated over 700,000 emails from employees of Enron that were released during legal proceedings and investigated the spreadsheets attached to these emails (over 15,000 spreadsheets) as well as discussion of the spreadsheets within the emails themselves (Hermans and Murphy-Hill, 2015). They found that the logic and methods of calculations within the spreadsheets were often documented within the bodies of emails that team members used to share and discuss spreadsheets. This means that, if someone needs to figure out why a step was taken or identify when an error was introduced into a spreadsheet, they may need to dig through the chain of old emails documenting that spreadsheet, rather than being able to find the relevant documentation within the spreadsheet's own file.

Often spreadsheets are designed, and their structure determined, by one person, and this is often done in an *ad hoc* fashion, rather than designing the spreadsheet to follow a common structure for the research field or for the laboratory group (Anderson et al., 2007). Often, data processing and analysis

pipelines for spreadsheets are not carefully designed; instead, it's more typically for spreadsheet user to start by directly entering data and formulas without a clear overall plan (AlTarawneh and Thorne, 2017). Often, the person who created the spreadsheet is the only person who fully knows how it works (Myneni and Patel, 2010), particularly if the spreadsheet includes complex macros or a complicated structure in the analysis pipeline (Creeth, 1985).

This practice creates a heavy dependence on the person who created that spreadsheet anytime the data or results in that spreadsheet need to be interpreted. This is particularly problematic in projects where the spreadsheet will be shared for collaboration or adapted to be used in a future project, as is often done in scientific research groups. One survey of neuroscience researchers at a UK institute, for example, found that “on average, 2–5 researchers share the same spreadsheet”. (AlTarawneh and Thorne, 2017) In this case, it can be hard to “onboard” new people to use the file, and much of the work and knowledge about the spreadsheet can be lost when that person moves on from the business or laboratory group (Creeth, 1985; Myneni and Patel, 2010). If you share a spreadsheet with numerous and complex macros and formulas included to clean and analyze the data, it can take an extensive amount of time, and in some cases may be impossible, for the researcher you share it with to decipher what is being done to get from the original data input in some cells to the final results shown in others and in graphs. Further, if others can't figure out the steps being done through macros and formulas in a spreadsheet, they will not be able to check it for problems in the logic of the overall analysis pipeline or for errors in the specific formulas used within that pipeline. They also will struggle to extend and adapt the spreadsheet to be used for other projects. These problems come up not only when sharing with a collaborator, but also when reviewing spreadsheets that you have previously created and used (as many have noted, your most frequent collaborator will likely be “future you”). In fact, one survey of biomedical researchers at the University of Washington noted that,

“The profusion of individually created spreadsheets containing overlapping and inconsistently updated data created a great deal of confusion within some labs. There was little consideration to future data exchange of submission requirements at the time of publication.” (Anderson et al., 2007)

There are methods that have been brought from more traditional programming work into spreadsheet programming to try to help limit errors, including a tool called assertions that allows users to validate data or test logic within their spreadsheets (Hermans et al., 2016). However, these are often not implemented, in part perhaps because many spreadsheet users see themselves as “end-users”, creating spreadsheets for their own personal use rather than as something robust to future use by others, and so don't seek out strategies adopted by “programmers” when creating stable tools for others to use (Hermans et al., 2016). In practice, though, often a spreadsheet is used much longer, and by more people, than originally intended. From early in the history of spreadsheet programs, users have shared spreadsheet files with interesting

functionality with other users (Levy, 1984), and the lifespan of a spreadsheet can be much longer than originally intended—a spreadsheet created by one user for their own personal use can end up being used and modified by that person or others for years (Hermans et al., 2016).

Subpar software for analysis.

While spreadsheets serve as a widely-used tool for data recording and analysis, in many cases spreadsheets programs are poorly suited to clean and analyze scientific data compared to other programs. As tools and interfaces continue to develop that make other software more user-friendly to those new to programming, scientists may want to reevaluate the costs and benefits, in terms of both time required for training and aptness of tools, for spreadsheet programs compared to using scripted programming languages like R and Python.

Several problems have been identified with spreadsheet programs in the context of recording and, especially, analyzing scientific data. First, some statistical methods may be inferior to those available in other statistical programming language. Since the most popular spreadsheet program (Excel) is closed source, it is hard to identify and diagnose such problems, and there is likely less of an incentive for problems in statistical methodology to be fixed (rather than using development time and funds to increase easier-to-see functionality in the program). Many statistical operations require computations that cannot be perfectly achieved with a computer, since the computer must ultimately solve many mathematical problems using numerical approximations rather than continuous methods (e.g., calculus). The choice of the algorithms used for these approximations heavily influence how closely a result approximates the true answer.

A series of papers examined the quality of statistical methods in several statistical software programs, including Excel, starting in the 1990s (McCullough and Wilson, 1999; McCullough, 1999; McCullough and Wilson, 2002, 2005; McCullough and Heiser, 2008; Mélard, 2014). In the earliest studies, they found some concerns across all programs considered (McCullough and Wilson, 1999; McCullough, 1999). One of the biggest concerns, however, was that there was little evidence over the years that the identified problems in Excel were resolved, or at least improved, over time (McCullough, 2001; McCullough and Heiser, 2008). The authors note that there may be little incentive for checking and fixing problems with algorithms for statistical approximation in closed source software like Excel, where sales might depend more on the more immediately evident functionality in the software, while problems with statistical algorithms might be less evident to potential users (McCullough, 2001).

Open source software, on the other hand, offers pathways for identifying and fixing any problems in the software, including for statistical algorithms and methods implemented in the software's code. Since the full source code is available, researchers can closely inspect the algorithms being used and compare them to the latest knowledge in statistical computing methodology. Further, if an inferior algorithm is in use, most open source software licenses allow

a user to adapt and extend the software, for example to implement better statistical algorithms.

Second, spreadsheet programs can include automated functionality that's meant to make something easier for most users, but that might invisibly create problems in some cases. A critical problem, for example, has been identified when using Excel for genomics data. When Excel encounters a cell value in a format that seems like it could be a date (e.g., "Mar-3-06"), it will try to convert that cell to a "date" class. Many software programs save date as this special "date" format, where it is printed and visually appears in a format like "3-Mar-06" but is saved internally by the program as a number (for Microsoft Excel, the number of days since January 1, 1900 (Willekens, 2013)). By doing this, the software can more easily undertake calculations with dates, like calculating the number of days between two dates or which of two dates is earlier. Bioinformatics researchers at the National Institutes of Health found that Excel was doing this type of automatic and irreversible date conversion for 30 gene names, including "MAR3" and "APR-4", resulting in these gene names being lost for further analysis (Zeeberg et al., 2004).

Avoiding this automatic date conversion required specifying that columns with columns susceptible to these problems, including columns of gene names, should be retained in a "text" class in Excel's file import process. While this problem was originally identified and published in 2004 (Zeeberg et al., 2004), along with tips to identify and avoid the problem, a study in 2016 found that approximately a fifth of genomics papers investigated in a large-scale review had gene name errors resulting from Excel automatic conversion, with the rate of errors actually increasing over time (Ziemann et al., 2016).

Other automatic conversion problems caused the loss of clone identifiers with composed of digits and the letter "E" (Zeeberg et al., 2004; Welsh et al., 2017), which were assumed to be expressing a number using scientific notation and so automatically and irreversibly converted to a numeric class. Further automatic conversion problems can be caused by cells that start with an operator (e.g., "+ control") or with leading zeros in a numeric identifier (e.g., "007") (Welsh et al., 2017).

Finally, spreadsheet programs can be limited as analysis needs become more complex or large (Topaloglou et al., 2004). For example, spreadsheets can be problematic when integrating or merging large, separate datasets (Birch et al., 2018). This can create barriers, for example, in biological studies seeking to integrate measurements from different instruments (e.g., flow cytometry data with RNA-sequencing data). Further, while spreadsheet programs continue to expand in their capacity for data, for very large datasets they continue to face limits that may be reached in practical applications (Birch et al., 2018)—until recently, for example, Excel could not handle more than one million rows of data per spreadsheet. Even when spreadsheets can handle larger data, their efficiency in running data processing and analysis pipelines across large datasets can be slow compared to code implemented with other programming

languages.

Difficulty collaborating with statisticians.

Modern biomedical researchers require large teams, with statisticians and bioinformaticians often forming a critical part of the team to enable sophisticated processing and analysis of experimental data. However, the process of combining data recording and analysis of experimental data, especially through the use of spreadsheet programs, can create barriers in working across disciplines. One group defined these issues as “data friction” and “science friction”—the extra steps and work required at each interface where data passes, for example, from a machine to analysis or from a collaborator in one discipline to one in a separate discipline (Edwards et al., 2011). From a survey of scientific labs, for example, one respondent said:

“I can give data that I think are appropriate to answer a question to a biostatistician, but when they look at it, they see it from a different point of view. And that spreadsheet does not really encapsulate where it came from very well, how was it generated, was it random, how was this data collected. You would run a series of queries that you think are pertinent to what this biostatistician would want to know. They become a part of the exploration and not just a receiver of whatever I decided to put in my spreadsheet on that day. What I get back is almost never fully documented in any way that I can really understand and add more to the process.” (Myneni and Patel, 2010)

When collaborating with statisticians or bioinformaticians, one of the key sources of this “data friction” can result from the use of spreadsheets to jointly record and analyze experimental data. First, spreadsheets are easy to print or copy into another format (e.g., PowerPoint presentation, Word document), and so researchers often design spreadsheets to be immediately visually appealing to viewers. For example, a spreadsheet might be designed to include hierarchically organized headers (e.g., heading and subheading, some within a cell merged across several columns), or to show the result of a calculation at the bottom of a column of observations (e.g., “Total” in the last cell of the column) (Teixeira and Amaral, 2016). Multiple separate small tables might be included in the same sheet, with empty cells used for visual separation, or use a “horizontal single entry” design, where the headers are in the leftmost column rather than the top row (Teixeira and Amaral, 2016).

These spreadsheet design choices make it much more difficult for the contents of the spreadsheet to be read into other statistical programs. These types of data require several extra steps in coding, in some cases fairly complex coding, with regular expressions or logical rules needed to parse out the data and convert it to the needed shape, before the statistical work can be done for the dataset. This is a poor use of time for a collaborating statistician, especially if it can be avoided through the design of the data recording template. Further, it introduces many more chances for errors in cleaning the data.

Further, information embedded in formulas, macros, and extra formatting like color or text boxes is lost when the spreadsheet file is input into

other programs. Spreadsheets allow users to use highlighting to represent information (e.g., measurements for control animals shown in red, those for experiment animals in blue) and to include information or documentation in text boxes. For example, one survey study of biomedical researchers at the University of Washington included this quote from a respondent: “I have one spreadsheet that has all of my chromosomes … and then I’ve gone through and color coded it for homozygosity and linkage.” (Anderson et al., 2007) All the information encoded in this sheet through color will be lost when the data from the spreadsheet is read into another statistical program.

2.1.3 Approaches to separate recording and analysis

In the remaining modules in this section, we will present and describe techniques that can be used to limit or remove these problems. First, in the next few modules, we will walk through techniques to design data recording formats so that data is saved in a consistent format across experiments within a laboratory group, and in a way that removes “data friction” for collaboration with statisticians or later use in scripted code. These techniques can be immediately used to design a better spreadsheet to be used solely for data collection.

In later modules, we will discuss the use of R projects to coordinate data recording and analysis steps within a directory, while using separate files for data recording versus data processing and analysis. These more advanced formats will enable the use of quality assurance / control measures like testing of data entry and analysis functionality, better documentation of data analysis pipelines, and easy use of version control to track projects and collaborate transparently and with a recorded history.

2.2 Principles and power of structured data formats

The format in which experimental data is recorded can have a large influence on how easy and likely it is to implement reproducibility tools in later stages of the research workflow. Recording data in a “structured” format brings many benefits. In this module, we will explain what makes a dataset “structured” and why this format is a powerful tool for reproducible research.

Every extra step of data cleaning is another chance to introduce errors in experimental biomedical data, and yet laboratory-based researchers often share experimental data with collaborators in a format that requires extensive additional cleaning before it can be input into data analysis (Broman and Woo, 2018). Recording data in a “structured” format brings many benefits for later stages of the research process, especially in terms of improving reproducibility and reducing the probability of errors in analysis (Ellis and Leek, 2018). Data that is in a structured, tabular, two-dimensional format is substantially easier for collaborators to understand and work with, without additional data formatting (Broman and Woo, 2018). Further, by using a consistent structured format

across many or all data in a research project, it becomes much easier to create solid, well-tested code scripts for data pre-processing and analysis and to apply those scripts consistently and reproducibly across datasets from multiple experiments (Broman and Woo, 2018). However, many biomedical researchers are unaware of this simple yet powerful strategy in data recording and how it can improve the efficiency and effectiveness of collaborations (Ellis and Leek, 2018).

Objectives. After this module, the trainee will be able to:

- List the characteristics of a structured data format
- Describe benefits for research transparency and reproducibility
- Outline other benefits of using a structured format when recording data

2.2.1 Data recording standards

For many areas of biological data, there has been a push to create standards for how data is recorded and communicated. Standards can clarify both the *content* that should be included in a dataset, the *format* in which that content is stored, and the *vocabulary* used within this data. One article names these three facets of a data standard as the **minimum information, file formats, and ontologies** (Ghosh et al., 2011).

Many people and organizations (including funders) are excited about the idea of developing and using data standards, especially at the community level. Good standards, that are widely adapted by researchers, can help in making sure that data submitted to data repositories are used widely and that software can be developed that is *interoperable* with data from many research group's experiments. There are also many advantages, if there are not community-level standards for recording a certain type of data, to develop and use local data standards for recording data from your own experiments. This section describes the elements that go into a data standard, discusses some choices to be made when defining a data standard (especially choices on data structure and file formats), and some of the advantages and disadvantages of developing and using data recording standards at both the research group and community levels.

Ontology standards.

Although it has the most complex name, an *ontology* (sometimes called a *terminology* (Sansone et al., 2012)) might be the easiest and quickest to adapt in recording data. An ontology helps define a vocabulary that is controlled and consistent to use that researchers can use to refer to concepts and concrete things within an area of research. It helps researchers, when they want to talk about an idea or thing, to use one word, and just one word, and to ensure that it will be the same word used by other researchers when they refer to that idea or thing. Ontologies also help to define the relationships between ideas or concrete things in a research area (Ghosh et al., 2011), but here we'll focus on their use in provided a consistent vocabulary to use when recording data.

For example, when recording a dataset, what do you call a small mammal

"It is important to distinguish between standards that specify how to actually do experiments and standards that specify how to describe experiments. Recommendations such as what standard reporters (probes) should be printed on microarrays or what quality control steps should be used in an experiment belong to the first category. Here we focus on the standards that specify how to describe and communicate data and information."

[@brazma2006standards]

On the root causes for irreproducibility in biomedical research: "First, a lack of standards for data generation leads to problems with the comparability and integration of data sets."

[@waltemath2016modeling]

that is often kept as a pet and that has four legs and whiskers and purrs? Do you record this as “cat” or “feline” or maybe, depending on the animal, even “tabby” or “tom” or “kitten”? Similarly, do you record tuberculosis as “tuberculosis” or “TB” or or maybe even “consumption”? If you do not use the same word consistently in a dataset to record an idea, then while a human might be able to understand that two words should be considered equivalent, a computer will not be able to immediately tell that “TB” should be treated equivalently to “tuberculosis”.

At a larger scale, if a research community can adapt an ontology they agree to use throughout their studies, it will make it easier to understand and integrate datasets produced by different research laboratories. If every research group uses the term “cat”, then code can easily be written to extract and combine all data recorded for cats across a large repository of experimental data. On the other hand, if different terms are used, then it might be necessary to first create a list of all terms used in datasets in the repository, then pick through that list to find any terms that are exchangeable with “cat”, then write script to pull data with any of those terms.

Several ontologies already exist or are being created for biological and other biomedical research (Ghosh et al., 2011). For biomedical science, practice, and research, the BioPortal website (<http://bioportal.bioontology.org/>) provides access to almost 800 ontologies, including several versions of the International Classification of Diseases, the Medical Subject Headings (MESH), the National Cancer Institute Thesaurus, the Orphanet Rare Disease Ontology and the National Center for Biotechnology Information (NCBI) Organismal Classification. For each ontology in the BioPortal website, the website provides a link for downloading the ontology in several formats. If you download the ontology using the “CSV” format, you can open it in your favorite spreadsheet program and explore how it defines specific terms to use for each idea or thing you might need to discuss within that topic area, as well as synonyms for some of the terms. To use an ontology when recording your own data, just make sure you use the ontology’s suggested terms in your data. For example, if you’d like to use the Ontology for Biomedical Investigations (<http://bioportal.bioontology.org/ontologies/OBI>) and you are recording how many children a woman has had who were born alive, you should name that column of the data “number of live births”, not “# live births” or “live births (N)” or anything else. Other collections of ontologies exist for fields of scientific research, including the Open Biological and Biomedical Ontology (OBO) Foundry (<http://www.obofoundry.org/>).

If there are community-wide ontologies in your field, it is worthwhile to use them in recording experimental data in your research group. Even better is to not only consistently use the defined terms, but also to follow any conventions with capitalization. While most statistical programs provide tools to change capitalization (for example, to change all letters in a character string to lower case), this process does require an extra step of data cleaning and an extra

chance for confusion or for errors to be introduced into data.

Minimum information standards. The next easiest facet of a data standard to bring into data recording in a research group is *minimum information*. Within a data recording standard, *minimum information* (sometimes also called *minimum reporting guidelines* (Sansone et al., 2012) or *reporting requirements* (Brazma et al., 2006)) specify what should be included in a dataset (Ghosh et al., 2011). Using minimum information standards help ensure that data within a laboratory, or data posted to a repository, contain a number of required elements. This makes it easier to re-use the data, either to compare it to data that a lab has newly generated, or to combine several posted datasets to aggregate them for a new, integrated analysis, considerations that are growing in importance with the increasing prevalence of research repositories and research consortia in many fields of biomedical science (Keller et al., 2017).

Standardized file formats. While using a standard ontology and a standard for minimum information is a helpful start, it just means that each dataset has the required elements somewhere, and using a consistent vocabulary—it doesn't specify where those elements are in the data or that they'll be in the same place in every dataset that meets those standards. As a result, datasets that all meet a common standard can still be very hard to combine, or to create common data analysis scripts and tools for, since each dataset will require a different process to pull out a given element.

Computer files serve as a way to organize data, whether that's recorded datapoints or written documents or computer programs (Kernighan and Pike, 1984). As the programmer Paul Ford writes,

“Data is just stuff, or rather, structured stuff: The cells of a spreadsheet, the structure of a Word document, computer programs themselves—all data.” (Ford, 2015)

A *file format* defines the rules for how the bytes in the chunk of memory that makes up a certain file should be parsed and interpreted anytime you want to meaningfully access and use the data within that file (Murrell, 2009). There are many file formats you may be familiar with—a file that ends in “.pdf” must be opened with a Portable Document Format (PDF) Reader like Adobe Acrobat, or it won’t make much sense (you can try this out by trying to open a “.pdf” file with a text editor, likeTextEdit or Notepad). The PDF Reader software has been programmed to interpret the data in a “.pdf” file based on rules defining what data is stored where in the section of computer memory for that file. Because most “.pdf” files conform to the same *file format* rules, powerful software can be built that works with any file in that format.

For certain types of biomedical data, the challenge of standardizing a format has similarly been addressed through the use of well-defined rules for not only the content of data, but also the way that content is *structured*. This can be standardized through *standardized file formats* (sometimes also called *data exchange formats* (Brazma et al., 2006)) and often defines not only the upper-level file

“Minimum information is a checklist of required supporting information for datasets from different experiments. Examples include: Minimum Information About a Microarray Experiment (MIAME), Minimum Information About a Proteomic Experiment (MIAPE), and the Minimum Information for Biological and Biomedical Investigations (MIBBI) project.”

[@ghosh2011software]

format (e.g., use of a “.csv” file type), but also how data within that file type should be organized. If data from different research groups and experiments is recorded using the same file format, researchers can develop software tools that can be repeatedly used to interpret and visualize that data; on the other hand, if different experiments record data using different formats, bespoke analysis scripts must be written for each separate dataset. This is a blow not only to the efficiency of data analysis, but also a threat to the accuracy of that analysis. If a set of tools can be developed that will work over and over, more time can be devoted to refining those tools and testing them for potential errors and bugs, while one-shot scripts often can’t be curated with similar care. One paper highlights the dangers that come with working with files that don’t follow a defined format:

“Beware of common pitfalls when working with *ad hoc* bioinformatics formats. Simple mistakes over minor details like file formats can consume a disproportionate amount of time and energy to discover and fix, so mind these details early on.” (Buffalo, 2015)

Some biomedical data file formats have been created to help smooth over the transfer of data that’s captured by complex equipment into software that can analyze that data. For example, many immunological studies need to measure immune cell populations in experiments, and to do so they use piece of equipment called a flow cytometer that probes cells in a sample with lasers and measures resulting intensities to determine characteristics of that cell. The data created by this equipment is large (often measurements from [x] or more lasers are taken for [x] cells in a single run) and somewhat complex, with a need to record not only the intensity measurements from each laser, but also some metadata about the equipment and characteristics of the run. If every company that makes flow cytometers used a different file format for saving the resulting data, then a different set of analysis software would need to be developed to accompany each piece of equipment. For example, a laboratory at a university with flow cytometers from two different companies would need licenses for two different software programs to work with data recorded by flow cytometers, and they would need to learn how to use each software package separately. There is a chance that software could be developed that used shared code for data analysis, but only if it also included separate sets of code to read in data from all types of equipment and to reformat them to a common format.

This isn’t the case, however. Instead, there is a commonly agreed on file format that flow cytometers should use to record the data they collect, called the *FCS file format*. This format has been defined through a series of papers [refs], with several separate versions as the file format has evolved over the years. It provides clear specifications on where to save each relevant piece of information in the block of memory devoted to the data recorded by the flow cytometer (in some cases, leaving a slot in the file blank if no relevant information was collected on that element). As a result, people have been

able to create software, both proprietary and open-source, that can be used with any data recorded by a flow cytometer, regardless of which company manufacturer the piece of equipment that was used to generate the data. Other types of biomedical data also have standardized file formats, including [example popular file formats for biomedical data]. In some cases these were defined by an organization, society, or initiative (e.g., the Metabolomics Standards Initiative) (Ghosh et al., 2011), while in some cases the file format developed by a specific equipment manufacturer has become popular enough that it's established itself as the standard for recording a type of data (Brazma et al., 2006).

For an even simpler example, think about recording dates. The *minimum information standard* for a date might always be the same—a recorded value must include the day of the month, month, and year. However, this information can be structured in a variety of ways. In many scientific data, it's common to record this information going from the largest to smallest units, so March 12, 2006, would be recorded “2006-03-12”. Another convention (especially in the US) is to record the month first (e.g., “3/12/06”), while another (more common in Europe) is to record the day of the month first (e.g., “12/3/06”).

If you are trying to combine data from different datasets with dates, and all use a different structure, it's easy to see how mistakes could be introduced unless the data is very carefully reformatted. For example, March 12 (“3-12” with month-first, “12-3” with day-first) could be easily mistaken to be December 3, and vice versa. Even if errors are avoided, combining data in different structures will take more time than combining data in the same structure, because of the extra needs for reformatting to get all data in a common structure.

2.2.2 Defining data standards for a research group

If some of the data you record from your experiments comes from complex equipment, like flow cytometers or mass spectrometers, you may be recording much of that data in a standardized format without any extra effort, because that format is the default output format for the equipment. However, you may have more control over other data recorded from your experiments, including smaller, less complex data recorded directly into a laboratory notebook or spreadsheet. You can derive a number of benefits from defining and using a standard for collecting this data, as well.

As already mentioned, for many of the complex types of biological data, standardized file formats exist. For example, flow cytometry data is typically collected and recorded in .fcs files. Every piece of flow cytometry equipment can then be built to output data in this format, and every piece of software to analyze flow cytometry data can be built to read in this input. The .fcs file format specifies how both raw data and metadata (e.g., compensation information, equipment details) can be saved within the file—everyone who uses that file format knows where to store data and where to find data of a certain type.

Much of the data collected in a laboratory is smaller, less complex, or less

“Vast swathes of bioscience data remain locked in esoteric formats, are described using nonstandard terminology, lack sufficient contextual information, or simply are never shared due to the perceived cost or futility of the exercise.”

[@sansone2012toward]

structured than these types of data, data that is recorded “by hand”, often into a laboratory notebook or a spreadsheet. One paper describes this type of data as the output of “traditional, low-throughput bench science” (Wilkinson et al., 2016). For this data recording, the data may be written down in an *ad hoc* way—however the particular researcher doing the experiment thinks makes sense—and that format might change with each experiment, even if many experiments have similar data outputs. As a result, it becomes harder to create standardized data processing and analysis scripts that work with this data or that integrate it with more complex data types. Further, if everyone in a laboratory sets up their spreadsheets for data recording in their own way, it is much harder for one person in the group to look at data another person recorded and immediately find what they need within the spreadsheet.

As a step in a better direction, the head of a research group may designate some common formats (e.g., a spreadsheet template) that all researchers in the group should use when recording the data from a specific type of experiments. This provides consistency across the recorded data for the laboratory, making easier for one lab member to quickly understand and navigate data saved by another lab member. It also opens the possibility to create tools or scripts that read in and analyze the data that can be re-used across multiple experiments with minor changes. This helps improve the efficiency and reproducibility of data analysis, visualization, and reporting steps of the research project.

This does require some extra time commitment (Brazma et al., 2006). First, time is needed to design the format, and it does take a while to develop a format that is inclusive enough that it includes a place to put all data you might want to record for a certain type of experiment. Second, it will take some time to teach each laboratory member what the format is and how to make sure they comply with it when they record data.

On the flip side, the longer-term advantages of using a defined, structured format will outweigh the short-term time investments for many laboratory groups for frequently used data types. By creating and using a consistent structure to record data of a certain type, members of a laboratory group can increase their efficiency (since they do not need to re-design a data recording structure repeatedly). They can also make it easier for downstream collaborators, like biostatisticians and bioinformaticians, to work with their output, as those collaborators can create tools and scripts that can be recycled across experiments and research projects if they know the data will always come to them in the same format. These benefits increase even more if data format standards are created and used by a whole research field (e.g., if a standard data recording format is always used for researchers conducting a certain type of drug development experiment), because then the tools built at one institution can be used at other institutions. However, this level of field-wide coordination can be hard to achieve, and so a more realistic immediate goal might be formalizing data recording structures within your research group or department, while keeping an eye out for formats that are gaining popularity as standards in

your field to adopt within your group.

One key advantage to using standardized data formats even for recording simple, “low-throughput” data is that everyone in the research group will be able to understand and work with data recorded by anyone else in the group—data will not become impenetrable once the person who recorded it leaves the group. Also, once a group member is used to the format, the process of setting up to record data from a new experiment will be quicker, as it won’t require the effort of deciding and setting up a *de novo* format for a spreadsheet or other recording file. Instead, a template file can be created that can be copied as a starting point for any new data recording.

Finally, there are huge benefits further down the data analysis pipeline that come with always recording data in the same format. If your group is working with a statistician or data analyst, it becomes much easier for that person to quickly understand a new file if it follows the same format as previous files. Further, if you work with a statistician or data analyst, he or she probably creates code scripts to read in, re-format, analyze, and visualize the data you’ve shared. If you always record data using the same format, these scripts can be reused with very little modification. This saves valuable time, and it helps make more time for more interesting statistical analysis if your collaborator can trim time off reading in and reformatting the data in their statistical programming language.

One paper suggests that the balance can be found, in terms of deciding whether the benefits of developing a standard outweigh the costs, by considering how often data of a certain type is generated and used:

“To develop and deploy a standard creates an overhead, which can be expensive. Standards will help only if a particular type of information has to be exchanged often enough to pay off the development, implementation, and usage of the standard during its lifespan.” (Brazma et al., 2006)

2.2.3 *Two-dimensional structured data format*

So far, this module has explored *why* you might want to use standardized data formats for recording experimental data. The rest of the module aims to give you tips for how to design and define your own standardized data formats, if you decide that is worthwhile for certain data types recorded within your research group.

Once you commit to creating a defined, structured format, you’ll need to decide what that structure should be. There are many options here, and it’s very tempting to use a format that is easy on human eyes (Buffalo, 2015). For example, it may seem appealing to create a format that could easily be copied and pasted into presentations and Word documents and that will look nice in those presentation formats. To facilitate this use, a laboratory might set up a recording format base on a spreadsheet template that includes multiple tables of different data types on the same sheet, or multi-level column headings.

Unfortunately, many of the characteristics that make a format attractive to human eyes will make it harder for a computer to make sense of. For example, if you include two tables in the same spreadsheet, it might make it easier for a person to get a one-screen look at two small data tables. However, if you want to read that data into a statistical program (or work with a collaborator who would), it will likely take some complex code to try to tell the computer how to find the second table in the spreadsheet. The same applies if you include some blank lines at the top of the spreadsheet, or use multi-level headers, or use “summary” rows at the bottom of a table. Further, any information you’ve included with colors or with text boxes in the spreadsheet will be lost when the data’s read into a statistical program. These design elements in a data format make it much harder to read the data embedded in a spreadsheet into other computer programs, including programs for more complex data analysis and visualization, like R and Python.

For most statistical programs, data can be easily read in from a spreadsheet if the computer can parse it in the following way: first, read in the first row, and assign each cell in that row as the *name* of a column. Then, read in the second row, and put each cell in the column the corresponds with the name of the cell in the same position in the first row. Also, set the data type for that column (e.g., number, character) based on the data type in this cell. Then, keep reading in rows until getting to a row that’s completely blank, and that will be the end of the data. If any of the rows has more cell than the first row, then that means that something went wrong, and should result in stopping or giving a warning. If any of the rows have fewer cells than the first row, then that means that there are missing data in that row, and should probably be recorded as missing values for any cells the row is “short” compared to the first row.

One of the easiest format for a computer to read is therefore a two-dimensional “box” of data, where the first row of the spreadsheet gives the column names, and where each row contains an equal number of entries. This type of two-dimensional tabular structure forms the basis for several popular “delimited” file formats that serve as a *lingua franca* across many simple computer programs, like the comma-separated values (CSV) format, the tab-delimited values (TSV) format, and the more general delimiter-separated values (DSV) format, which are a common format for data exchange across databases, spreadsheet programs, and statistical programs (Janssens, 2014; Raymond, 2003; Buffalo, 2015).

If you think of the computer parsing a spreadsheet as described above, hopefully it clarifies why some spreadsheet formats would cause problems. For example, if you have two tables in the same spreadsheet, with blank lines between them, the computer will likely either think it’s read all the data after the first table, and so not read in any data from the second table, or it will think the data from both tables belong in a single table, with some rows of missing data in the center. To write the code to read in data from two tables into two separate datasets in a statistical program, it will be necessary to write some

“Data should be formatted in a way that facilitates computer readability. All too often, we as humans record data in a way that maximizes its readability to us, but takes a considerable amount of cleaning and tidying before it can be processed by a computer. The more data (and metadata) that is computer readable, the more we can leverage our computers to work with this data.”

[@buffalo2015bioinformatics]

“Tabular plain-text data formats are used extensively in computing. The basic format is incredibly simple: each row (also known as a record) is kept on its own line, and each column (also known as a field) is separate by some delimiter.”

[@buffalo2015bioinformatics]

complex code to tell the computer how to search out the start of the second table in the spreadsheet.

Similar problems come up if a spreadsheet diverges from a regular, two-dimensional format, with a single row of column names to start the data. For example, if the data uses multiple rows to create multi-level column headers, anyone reading it into another program will need to either skip some of the rows of the column headers, and so lose information in the original spreadsheet, or write complex code to parse the column headers separately, then read in the later rows with data, and then stick the two elements back together. “Summary” rows at the end of a dataset (for example, the sums or means of all values in a column) will need to be trimmed off when the data is read into other programs, since most of the analysis and visualization someone would want to do in another program will calculate any summaries fresh, and will want each row of a dataset to represent the same “type” and level of data (e.g., one measurement from one animal).

For anything in a data format that requires extra coding when reading data into another program, you are introducing a new opportunity for errors at the interface between data recording and data analysis. If there are strong reasons to use a format that requires these extra steps, it will still be possible to create code to read in and parse the data in statistical programs, and if the same format is consistently used, then scripts can be developed and thoroughly tested to allow this. However, do keep in mind that this will be an extra burden on any data analysis collaborators who are using a program besides a spreadsheet program. The extra time this will require could be large, since this code should be vetted and tested thoroughly to ensure that the data cleaning process is not introducing errors. By contrast, if the data is recorded in a two-dimensional format with a single row of column names as the first row, data analysts can likely read it quickly and cleanly into other programs, with low risks of errors in the transfer of data from the spreadsheet.

2.2.4 Saving two-dimensional structured data in plain text file formats

If you have recorded data in a two-dimensional structured format, you can choose to save it in either a *plain text* format or a *binary* format. With a plain text format, a file is “human readable” when it’s opened in a text editor (Hunt et al., 2000; Janssens, 2014), because each byte that encodes the file translates to a single character (Murrell, 2009), usually using an ASCII or Unicode encoding. Common plain text file formats used for biomedical research include CSV and TSV files (these are distinguished only by the character used as a delimiter—commas for CSV files versus tabs for TSV files) (Buffalo, 2015), other more complex file formats like SAM and XML are also typically saved in plain text.

A binary file format, on the other hand, encodes data within the file using an encoding system that differs from ASCII or Unicode. To extract the data

“Cleaning data is a short-term solution, and preventing errors is promoted as a permanent solution. The drawback to cleaning data is that the process never ends, is costly, and may allow many errors to avoid detection.”

[@keller2017evolution]

in a meaningful way, a computer program must know and use rules for the encoding and structure of that file format, and those rules will be different for each different binary file format (Murrell, 2009). Some binary file formats are “open”, with all the information on these rules and encodings available for anyone to read. On the other hand, other binary file formats are proprietary, without available guidance on how to interpret or use the data stored in them when creating new software tools. Binary files, because they don’t follow the restrictions of plain text encoding and format, can encode and organize data in a way that’s often much more compressed, because it’s optimized to suit a specific type of data. This means that binary file formats can often store more data within a certain amount of computer memory compared to plain text file formats. Binary files can also be designed so that the computer can find and read a specific piece of data, rather than needing to read data in linearly from the start to the end of a file as with plain text formats. This means that programs can often access specific bits of data much more quickly from a binary file format than from a plain text format, making computation processing run much faster.

However, even with the speed and size advantages of many binary file formats, it is often worthwhile to record and save experimental data in a plain text, rather than binary, file format. There are a number of advantages to using a plain text format. A plain text format may take more space (in terms of computer memory) and take longer to process within other programs; however, its benefits typically outweigh these limitations (Hunt et al., 2000). Advantages include: (1) humans can read the file directly (Hunt et al., 2000; Janssens, 2014), and should always be able to, regardless of changes in and future obsolescence of computer programs; (2) almost all software programs for analyzing and processing files can input plain-text files, while binary file formats often require specialized software (Murrell, 2009); (3) the Unix system, which has influenced many existing software programs, especially open-source programs for data analysis and command-line tools, are based on inputting and outputting line-based plain-text files (Janssens, 2014); and (4) plain-text files can be easily tracked with version control (Hunt et al., 2000). These advantages might become particularly important in cases where researchers need to combine and integrate heterogeneous data, for example data coming from different instruments.

Another advantage of storing data in a plain text format is that it makes version control, which we’ll discuss in a later module, a much more powerful tool. With plain text files, you can use version control to see the specific changes to a file. With binary files, you can typically see if a file was changed, but it’s much harder to see exactly what within the file was changed.

The book *The Pragmatic Programmer* highlights some of the advantages of plain text:

“Human-readable forms of data, and self-describing data, will outlive all other forms of data and the applications that created them. Period. As long as the data

survives, you will have a chance to be able to use it—potentially long after the original application that wrote it is defunct. ... Even in the future of XML-based intelligent agents that travel the wild and dangerous Internet autonomously, negotiating data interchange among themselves, the ubiquitous text file will still be there. In fact, in heterogeneous environments the advantages of plain text can outweigh all of the drawbacks. You need to ensure that all parties can communicate using a common standard. Plain text is that standard.” (Hunt et al., 2000)

Paul Ford, by contrast, describes some of the disadvantages of a binary file format, using the Photoshop file format as an example:

“A Photoshop file is a lump of binary data. It just sits there on your hard drive. Open it in Photoshop, and there are your guides, your color swatches, and of course, the manifold pixels of your intent. But outside of Photoshop that file is an enigma. There is not ‘view source’. You can, if you’re passionate, read the standard on the web, and it’s all piled in there, the history of pictures on computers. That’s when it becomes clear: only Photoshop’s creator Adobe can understand this thing.” (Ford, 2014)

Structuring data in a gridded, two-dimensional format, as described in the last section, will be helpful even if it is in a file format that is binary, like Excel. However, there are added benefits to saving the structured data in a plain text format. Older Excel spreadsheets are typically saved in a proprietary file format (“.xls”), while more recently Excel has saved files to an open binary format based on packaging XML files with the data (“.xlsx” file format) (Janssens, 2014). While the open proprietary format is preferable, since tools can be developed to work with them by people other than the Microsoft team, both file formats still face some of the limitations of binary file formats as a way of recording experimental data. However, even if you have used a spreadsheet program like Excel to record data, it’s very easy to still save that data in a plain text file format (Murrell, 2009). In most spreadsheet programs, you can choose to save a file “As CSV”.

2.2.5 Occasions for more complex data structures and file formats

There are some cases where a two-dimensional data format may not be adequate for recording experimental data, despite this format’s advantages in improving reproducibility through later data analysis steps. Similarly, there may be cases where a binary file format, or use of a database, will outweigh the benefits of saving data to a plain text format. Being familiar with different file formats can also be helpful when you need to integrate data stored in different formats (Murrell, 2009).

Non-tabular plain-text formats. First, some data has a linked or hierarchical nature, in terms of how data points are connected through the dataset. For example, data on a family tree have a hierarchical structure, where different numbers of children are recorded for each parent. As another example, if you were building a dataset describing how scientists have collaborated together

as coauthors, that data might form a network. In many cases, it is possible to structure datasets with these types of “non-tabular” structure using the “tidy data” tabular format described in the next section. However, in very complex cases, it may work better to use a non-tabular data format (Raymond, 2003). Popular data formats that are non-tabular include the eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) formats, both of which are well-suited for hierarchically-structured data. You may also have data you would like to use in XML or JSON formats if you are using web services to pull datasets from online repositories, as open data application programming interfaces (APIs) often return data in these formats (Janssens, 2014).

Another use of file formats that are plain text but meant to be streamed, rather than read in as a whole. When reading in data stored in a delimited plain text file, like a CSV file, a statistical program like R will typically read in all the data and then operate on the dataset as a whole. If a data file is very large, then reading in all the data at once might require so much memory that it slows down processing, or even exceed the program’s memory cap [?]. One strategy is to design a data format so that the program can read in a small amount of the file, process that piece of the data, write the result out, and remove that bit of data from the program’s memory before moving into the next portion of data (Buffalo, 2015). This *streaming* approach is sometimes used with some file formats used for biomedical research, including FASTA and FASTQ files.

Databases. When research datasets include not only data that can be expressed in plain text, but also data like images, photographs, or videos, it may be worth considering using a database to store the data (Murrell, 2009). Relational database management system software, like [examples. MySQL? PostgreSQL?] can be used to organize data in a way that records connections (*relations*) between different pieces of data and allows you to access different combinations of that data quickly using Structured Query Language, or SQL (Ford, 2015). Further, some statistical programming languages, including R, now have tools that allow you to directly access and work with data from a database from within the statistical program, and in some cases using scripts that are very similar or identical to the code that would be used if you’d read the data into the program from a plain text file.

It will be more complicated to set up a database for recording experimental data, and so it’s often preferable to instead save data in plain text files within a file directory, if the data is simple enough to allow that. However, there are some fairly simple database solutions that are now available, including SQLite (Buffalo, 2015).

Binary file formats.

There are cases where it may not be best to store laboratory-generated data in a plain text format. For example, the output from a flow cytometer is large and would take up a lot (more) computer memory if stored in a plain text format, and it would take much longer to read and work with the data in analysis software if it were in that format. For very large datasets like this,

“The database is the unsung infrastructure of the world, the shared memory of every corporation, and the foundation of every major web site. And they are everywhere. Nearly every host-your-own-web-site package comes with access to a database called MySQL; just about every cell phone has SQLite3, a tiny, pocket-sized database, built in.”

[@ford2015i]

it may be necessary to use a binary data format, either for size or speed or both (Kernighan and Pike, 1984; Hunt et al., 2000). For very large biomedical datasets, binary file formats are sometimes designed for *out-of-memory approaches* (Buffalo, 2015), where a file format is designed in a way that allows computer programs to find and read only specific pieces of data in a file through a process called *random access*, rather than needing to read the full file into memory before a specific piece of data in the file can be accessed (a.k.a., *sequential access*) (Murrell, 2009).

2.2.6 Levels of standardization—research group to research community

Standards can operate both at the level of individual research groups and at the level of the scientific community as a whole. The potential advantages of community-level standards are big: they offer the chance to develop common-purpose tools and code scripts for data analysis, as well as make it easier to re-use and combine experimental data from previous research that is posted in open data repositories. If a software tool can be reused, then more time can be spent in developing and testing it, and as more people use it, bugs and shortcomings can be identified and corrected. Community-wide standards can lead to databases with data from different experiments, and from different laboratory groups, structured in a way that makes it easy for other researchers to understand each dataset, find pieces of data of interest within datasets, and integrate different datasets (Lynch, 2008). Similarly, with community-wide standards, it can become much easier for different research groups to collaborate with each other or for a research group to use data generated by equipment from different manufacturers (Schadt et al., 2010).

However, there are important limitations to community-wide standards, as well. It can be very difficult to impose such standards top-down and community-wide, particularly for low-throughput data collection (e.g., laboratory bench measurements), where research groups have long been in the habit of recording data in spreadsheets in a format defined by individual researchers or research groups. One paper highlights this point:

“The data exchange formats PSI-MI and MAGE-ML have helped to get many of the high-throughput data sets into the public domain. Nevertheless, from a bench biologist’s point of view benefits from adopting standards are not yet overwhelming. Most standardization efforts are still mainly an investment for biologists.” (Brazma et al., 2006)

Further, in some fields, community-wide standards have struggled to remain stable, which can frustrate community members, as scripts and software must be revamped to handle shifting formats (Buffalo, 2015; Barga et al., 2011). In some cases, a useful compromise is to follow a general data recording format, rather than one that is very prescriptive. For example, committing to recording data in a format that is “tidy” (which we discuss extensively in the next module) may be much more flexible—and able to meet the needs of a large range of

“Without community-level harmonization and interoperability, many community projects risk becoming data silos.”

[@sansone2012oward]

“Solutions to integrating the new generation of large-scale data sets require approaches akin to those used in physics, climatology and other quantitative disciplines that have mastered the collection of large data sets.”

[@schadt2010computational]

experimental designs—than the use of a common spreadsheet template or a more prescriptive standardized data format.

2.3 The ‘tidy’ data format

The “tidy” data format is one implementation of a tabular, two-dimensional structured data format that has quickly gained popularity among statisticians and data scientists since it was defined in a 2014 paper (Wickham, 2014). The “tidy” data format plugs into R’s *tidyverse* framework, which enables powerful and user-friendly data management, processing, and analysis by combining simple tools to solve complex, multi-step problems (Ross et al., 2017; Silge and Robinson, 2016; Wickham, 2016; Wickham and Grolemund, 2016). Since the *tidyverse* tools are simple and share a common interface, they are easier to learn, use, and combine than tools created in the traditional base R framework (Ross et al., 2017; Lowndes et al., 2017; *t*; McNamara, 2016). This *tidyverse* framework is quickly becoming the standard taught in introductory R courses and books (Hicks and Irizarry, 2017; Baumer, 2015; Kaplan, 2018; Stander and Dalla Valle, 2017; *t*; McNamara, 2016), ensuring ample training resources for researchers new to programming, including books (e.g., (Baumer et al., 2017; Irizarry and Love, 2016; Wickham and Grolemund, 2016)), massive open online courses (MOOCs), on-site university courses (Baumer, 2015; Kaplan, 2018; Stander and Dalla Valle, 2017), and Software Carpentry workshops (Wilson, 2014; Pawlik et al., 2017). Further, tools that extend the *tidyverse* have been created to enable high-quality data analysis and visualization in several domains, including text mining (Silge and Robinson, 2017), microbiome studies (McMurdie and Holmes, 2013), natural language processing (Arnold, 2017), network analysis (Tyner et al., 2017), ecology (Hsieh et al., 2016), and genomics (Yin et al., 2012). In this section, we will explain what characteristics determine if a dataset is “tidy” and how use of the “tidy” implementation of a structure data format can improve the ease and efficiency of “Team Science”.

Objectives. After this module, the trainee will be able to:

- List characteristics defining the “tidy” structured data format
- Explain the difference between the a structured data format (general concept) and the ‘tidy’ data format (one popular implementation)

In the previous module, we explained the benefits of saving data in a structured format, and in particular using a two-dimensional format saved to a plain text file when possible. In this section, we’ll talk about the “tidy text” format—a set of principles to use when structuring two-dimensional tabular data. These principles cover some basic rules for ordering the data, and the resulting datasets can be very easily worked with, including to further clean, model, and visualize the data, and to integrate the data with other datasets, using a series of open-source tools on the R platform called the “tidyverse”. These characteristics mean that, if you are planning to use a standardized data

format for recording experimental data in your research group, you may want to consider creating one that adheres to the “tidy data” rules.

We’ll start by describing what rules a dataset’s format must follow for it to be “tidy”, and try to clarify how you can set up your data recording to follow these rules. In a later part of this module, we’ll talk more about the tidyverse tools that you can use with this data, as well as give some resources for finding out more about the tidyverse and how to use its tools.

Since a key advantage of the “tidy data” format is that it works so well with R’s “tidyverse” tools, we’ll also talk a bit in this section about the use of scripting languages like R, and how using them to analyze and visualize the data you collect can improve the overall reproducibility of your research.

2.3.1 *What makes data “tidy”?*

The “tidy” data format describes one way to structure tabular data. The name follows from the focus of this data format and its associated set of tools—the “tidyverse”—on preparing and cleaning (“tidying”) data, in contrast to sets of tools more focused on other steps, like data analysis (Wickham, 2014). The word “tidy” is not meant to apply that other formats are “dirty”, or that they include data that is incorrect or subpar. In fact, the same set of datapoints could be saved in a file in a way that is either “tidy” (in the sense of (Wickham, 2014)) or untidy, depending only on how the data are organized across columns and rows.

The rules for making data “tidy” are pretty simple, and they are defined in detail, and with extended examples, in the journal article that originally defined the data format (Wickham, 2014). Here, we’ll go through those rules, with the hope that you’ll be able to understand what makes a dataset follow the “tidy” data format. If so, you’ll be able to set up your data recording template to follow this template, and you’ll be able to tell if other data you get is in this format and, if it’s not, restructure it so that it does. In the next part of this module, we’ll explain why it’s so useful to have your data in this format.

Tidy data, first, must be in a tabular (i.e., two-dimensional, with columns and rows, and with all rows and columns of the same length—nothing “ragged”). If you recorded data in a spreadsheet using a very basic strategy of saving a single table per spreadsheet, with the first row giving the column names (as described in the previous module), then your data will be in a tabular format. It should not be saved in a hierarchical structure, like XML (although there are now tools for converting data from XML to a “tidy” format, so you may still be able to take advantage of the tidyverse even if you must use XML for your data recording). In general, if your recorded data looks “boxy”, it’s probably in a two-dimensional tabular format.

There are some additional criteria for the “tidy” data format, though, and so not every structured, tabular dataset is in a “tidy” format. The first of these rules are that each row of a “tidy” dataset records the values for a single ob-

“The development of tidy data has been driven by my experience from working with real-world datasets. With few, if any, constraints on their organization, such datasets are often constructed in bizarre ways. I have spent countless hours struggling to get such datasets organized in a way that makes data analysis possible, let alone easy.”
[@wickham2014tidy]

servation, and that each column provides characteristics or measurements of a certain type, in the order of the observations given by the rows (Wickham, 2014). For example, if you have collected data from several experimental samples and plated each sample at several dilutions to count viable bacteria, then you could record the results using one row per dilution—specifying each dilution for each sample as your level of observation for the data—to save the data in a tidy format.

To be able to decide if your data is tidy, then, you need to know what forms a single observation in the data you’re collecting. The *unit of observation* of a dataset is the unit at which you take measurements (Sedgwick, 2014). This idea is different than the *unit of analysis*, which is the unit that you’re focusing on in your study hypotheses and conclusions (this is sometimes also called the “sampling unit” or “unit of investigation”) (Altman and Bland, 1997). In some cases, these two might be equivalent (the same unit is both the unit of observation and the unit of measurement), but often they are not (Sedgwick, 2014). Again, in the example of plating samples at several dilutions each, the unit of observation for the resulting data might be at the level of each dilution for each sample, where the unit of analysis that you are ultimately interested in is simply each sample.

As another example, say you are testing how the immune system of mice responds to a certain drug over time. You may have several replicates of mice measured at several time points, and those mice might be in separate groups (for example, infected with a disease versus uninfected). In this case, if a separate mouse (replicate) is used to collect each observation, and a mouse is never measured twice (i.e., at different time points, or for a different infection status), then the unit of measurement is the mouse. There should be one and only one row in your dataset for each mouse, and that row should include two types of information: first, information about the unit being measured (e.g., the time point, whether the mouse was infected, and a unique mouse identifier) and, second, the results of that measurement (e.g., the weight of the mouse when it was sacrificed, the levels of different immune cell populations in the mouse, a measure of the extent of infection in the mouse if it was infected, and perhaps some notes about anything of note for the mouse, like if it appeared noticeably sick). In this case, the *unit of analysis* might be the drug, or a combination of drug and dose—ultimately, you may want to test something like if one drug is more effective than another. However, the *unit of observation*, the level at which each data point is collected, is the mouse, with each mouse providing a single observation to help answer the larger research question.

As another example, say you conducted a trial on human subjects, to see how the use of a certain treatment affects the speed of recovery, where each study subject was measured at different time points. In this case, the unit of observation is the combination of study subject and time point (while the unit of analysis is the study subject, if the treatments are randomized to the study subjects). That means that Subject 1’s measurement at Time 1 would

“Most statistical datasets are rectangular tables made up of rows and columns ... [but] there are many ways to structure the same underlying data. ... Real datasets can, and often do, violate the three precepts of tidy data in almost every way imaginable.”

[@wickham2014tidy]

“The unit of observation and unit of analysis are often confused. The unit of observation, sometimes referred to as the unit of measurement, is defined statistically as the ‘who’ or ‘what’ for which data are measured or collected. The unit of analysis is defined statistically as the ‘who’ or ‘what’ for which information is analysed and conclusions are made.”

[@sedgwick2014unit]

be one observation, and the same person's measurement at Time 2 would be a separate observation. For a dataset to comply with the “tidy” data format, these two observations would need to be recorded on separate lines in the data. If the data instead had different columns to record each study subject's measurements at different time points, then the data would still be tabular, but it would not be “tidy”.

In this second example, you may initially find the “tidy” format unappealing, because it seems like it would lead to a lot of repeated data. For example, if you wanted to record each study subject's sex, it seems like the “tidy” format would require you to repeat that information in each separate line of data that's used to record the measurements for that subject for different time points. This isn't the case—instead, with a “tidy” data format, different “levels” of data observations should be recorded in separate tables (Wickham, 2014). So, if you have some data on each study subject that does not change across the time points of the study—like the subject's ID, sex, and age at enrollment—those form a separate dataset, one where the unit of observation is the study subject, so there should be just one row of data per study subject in that data table, while the measurements for each time point should be recorded in a separate data table. A unique identifier, like a subject ID, should be recorded in each data table so it can be used to link the data in the two tables. If you are using a spreadsheet to record data, this would mean that the data for these separate levels of observation should be recorded in separate sheets, and not on the same sheet of a spreadsheet file. Once you read the data into a scripting language like R or Python, it will be easy to link the larger and smaller “tidy” datasets as needed for analysis, visualizations, and reports.

Once you have divided your data into separate datasets based on the level of observation, and structured each row to record data for a single observation based on the unit of observation within that dataset, each column should be used to measure a separate characteristic or measurement (*a variable*) for each measurement (Wickham, 2014). A column could either give characteristics of the data that were pre-defined by the study design—for example, the treatment assigned to a mouse, or the time point at which a measurement was taken if the study design defined the time points when measurements would be taken. These types of column values are also sometimes called *fixed variables* (Wickham, 2014). Other columns will record observed measurements—values that were not set prior to the experiment. These might include values like the level of infection measured in an animal and are sometimes called *measured variables* (Wickham, 2014).

2.3.2 Why make your data “tidy”?

This may all seem like a lot of extra work, to make a dataset “tidy”, and why bother if you already have it in a structured, tabular format? It turns out that, once you get the hang of what gives data a “tidy” format, it's pretty simple to

“While the order of variables and observations does not affect analysis, a good ordering makes it easier to scan the raw values. One way of organizing variables is by their role in the analysis: are values fixed by the design of the data collection, or are they measured during the course of the experiment? Fixed variables describe the experimental design and are known in advance. ... Measured variables are what we actually measure in the study. Fixed variables should come first, followed by measured variables, each ordered so that related variables are contiguous. Rows can then be ordered by the first variable, breaking ties with the second and subsequent (fixed) variables.”

[@wickham2014tidy]

design recording formats that comply with these rules. What's more, when data is in a "tidy" format, it can be directly input into a collection of tools in R that belong to something called the "tidyverse". This collection of tools is very straightforward to use and so powerful that it's well worth making an effort to record data in a format that works directly with the tools, if possible. Outside of cases of very complex or very large data, it should be possible.

The "tidyverse" is a collection of tools united by a common philosophy: **very complex things can be done simply and efficiently with small, sharp tools that share a common interface.**

The tidyverse isn't the only popular system that follows this philosophy—one other favorite is Legos. Legos are small, plastic bricks, with small studs on top and tubes for the studs to fit into on the bottom. The studs all have the same, standardized size and are all spaced the same distance apart. Therefore, the bricks can be joined together in any combination, since each brick uses the same *input format* (studs of the standard size and spaced at the standard distance fit into the tubes on the bottom of the brick) and the same *output format* (again, studs of the standard size and spaced at the standard distance at the top of the brick).

This is true if you want to build with bricks of different colors or different heights or different widths or depths. It even allows you to include bricks at certain spots that either don't require input (for example, a solid sheet that serves as the base) or that don't give output (for example, the round smooth bricks with painted "eyes" that are used to create different creatures). With Legos, even though each "tool" (brick) is very simple, the tools can be combined in infinite variations to create very complex structures.

The tools in the "tidyverse" operate on a similar principle. They all input one of a few very straightforward data types, and they (almost) all output data in the same format they input it. For most of the tools, their required format for input and output is the "tidy data" format (Wickham, 2014), called a *tidy dataframe* in R—this is a dataframe that follows the rules detailed earlier in this section.

Some of the tools require input and output of *vectors* instead of *tidy dataframes* (Wickham, 2014); a *vector* in R is a one-dimensional string of values, all of which are of the same data type (e.g., all numbers, or all character strings, like names). In a *tidy dataframe*, each column is a *vector*, and the *dataframe* is essentially several *vectors* of the same length stuck together to make a table. Having functions that input and output *vectors*, then, means that you can use those functions to make changes to the columns in a *tidy dataframe*.

A few functions in the "tidyverse" input a *tidy dataframe* but output data in a different format. For example, visualizations are created using a function called *ggplot*, as well as its helper functions and extensions. This function inputs data in a *tidy dataframe* but outputs it in a type of R object called a "ggplot object". This object encodes the plot the code created, so in this case the fact that the output is in a different format from the endpoint is similar to with the "eye"

"A standard makes initial data cleaning easier because you do not need to start from scratch and reinvent the wheel every time. The tidy data standard has been designed to facilitate initial exploration and analysis of the data, and to simplify the development of data analysis tools that work well together."

[@wickham2014tidy]

"Tidy data is great for a huge fraction of data analyses you might be interested in. It makes organizing, developing, and sharing data a lot easier. It's how I recommend most people share data."

[@leek2017toward]

"The philosophy of the tidyverse is similar to and inspired by the "unix philosophy", a set of loose principles that ensure most command line tools play well together. ... Each function should solve one small and well-defined class of problems. To solve more complex problems, you combine simple pieces in a standard way."

[@ross2017declutter]

blocks in Legos, where it's meant as a final output step, and you don't intend to do anything further in the code once you move into that step.

This common input / output interface, and the use of small tools that follow this interface and can be combined in various ways, is what makes the tidyverse tools so powerful. However, there are other good things about the tidyverse that make it so popular. One is that it's fairly easy to learn to use the tools, in comparison to learning how to write code for other R tools (Robinson, 2017; Peng, 2018). The developers who have created the tidyverse tools have taken a lot of effort to try to make sure that they have a clear and consistent user interface across tools (Wickham, 2017; Bryan and Wickham, 2017). So far, we've talked about the interface between functions, and how a common *input / output interface* means the functions can be chained together more easily. But there's another interface that's important for software tools: the rules for how a computer users employ that tool, or the *user interface*.

To help understand a user interface, and how having a consistent user interface across tools is useful, let's think about a different example—cars. When you drive a car, you get the car to do what you want through the steering wheel, the gas pedal, the break pedal, and different knobs and buttons on the dashboard. When the car needs to give you feedback, it uses different gauges on the dashboard, like the speedometer, as well as warning lights and sounds. Collectively, these ways of interacting with your car make up the car's *user interface*. In the same way, each function in a programming language has a collection of parameters you can set, which let you customize the way the function runs, as well as a way of providing you output once the function has finished running and the way to provide any messages or warnings about the function's run. For functions, the software developer can usually choose design elements for the function's user interface, including which parameters to include for the function, what to name those parameters, and how to provide feedback to the user through messages, warnings, and the final output.

If a collection of tools is similar in its user interfaces, it will make it easier for users to learn and use any of the tools in that collection once they've learned how to use one. For cars, this explains how the rental car business is able to succeed. Even though different car models are very different in many characteristics—their engines, their colors, their software—they are very consistent in their user interfaces. Once you've learned how to drive one car, when you get in a new car, the gas pedal, brake, and steering wheel are almost guaranteed to be in about the same place and to operate about the same way as in the car you learned to drive in. The exceptions are rare enough to be memorable—think how many movies have a laughline from a character trying to drive a car with the driver side on the right if they're used to the left or vice versa.

The tidyverse tools are similarly designed so that they all have a very similar user interface. For example, many of the tidyverse functions use a parameter named “.data” to refer to the tidy dataframe to input into the function, and this

parameter is often the first listed for functions. Similarly, parameters named “.vars” and “.funs” are repeatedly used over tidyverse functions, with the same meaning in each case. What’s more, the tidyverse functions are typically given names that very clearly describe the action that the function does, like filter, summarize, mutate, and group. As a result, the final code is very clear and can almost be “read” as a natural language, rather than code.

As a result, the tidyverse collection of tools is pretty easy to learn, compared to other sets of functions in scripting languages, and pretty easy to expand your knowledge of once you know some of its functions. Several people who teach R programming now focus on first teaching the tidyverse, given these characteristics (Robinson, 2017; Peng, 2018), and it’s often a first focus for online courses and workshops on R programming. Since it’s main data structure is the “tidy data” structure, it’s often well worth recording data in this format so that all these tools can easily be used to explore and model the data.

2.3.3 Using tidyverse tools with data in the “tidy data” format

The tidyverse includes tools for many of the tasks you might need to do while managing and working with experimental data. When you download R, you get what’s called *base R*. This includes the main code that drives anything you do in R, as well as functions for doing many core tasks. However, the power of R is that, in addition to base R, you can also add onto R through what are called *packages* (sometimes also referred to as *extensions* or *libraries*). These are kind of like “booster packs” that add on new functions for R. They can be created and contributed by anyone, and many are collected through a few key repositories like CRAN and Bioconductor.

All the tidyverse tools are included in R extension packages, rather than base R, so once you download R, you’ll need to download these packages as well to use the tidyverse tools. The core tidyverse functions include functions to read in data (the `readr` package for reading in plain text, delimited files, `readxl` to read in data from Excel spreadsheets), clean or summarize the data (the `dplyr` package, which includes functions to merge different datasets, make new columns as functions of old ones, and summarize columns in the data, either as a whole or by group), and reformat the data if needed to get it in a tidy format (the `tidyverse` package). The tidyverse also includes more precise tools, including tools to parse dates and times (`lubridate`) and tools to work with character strings, including using regular expressions as a powerful way to find and use certain patterns in strings (`stringr`). Finally, the tidyverse includes powerful functions for visualizing data, based around the `ggplot2` package, which implements a “grammar of graphics” within R.

You can install and load any of these tidyverse packages one-by-one using the `install.packages` and `library` functions with the package name from within R. If you are planning on using many of the tidyverse packages, you can also install and load many of the tidyverse functions by installing a package called

“Another part of what makes the Tidyverse effective is harder to see and, indeed, the goal is for it to become invisible: conventions. The Tidyverse philosophy is to rigorously (and ruthlessly) identify and obey common conventions. This applies to the objects passed from one function to another and to the user interface each function presents. Taken in isolation, each instance of this seems small and unimportant. But collectively, it creates a cohesive system: having learned one component you are more likely to be able to guess how another different component works.”

[@bryan2017data]

“The goal of [the tidy tools] principles is to provide a uniform interface so that tidyverse packages work together naturally, and once you’ve mastered one, you have a head start on mastering the others.”

[@wickham2017tidy]

“All our code is underpinned by the principles of tidy data, the grammar of data manipulation, and the tidyverse R packages developed by Wickham. This deliberate philosophy for thinking about data helped bridge our scientific questions with the data processing required to get there, and the readability and conciseness of tidyverse operations makes our data analysis read more as a story arc. Operations require less syntax—which can mean fewer potential errors that are easier to identify—and they can be chained together, minimizing intermediate steps and data objects that can cause clutter and confusion. The tidyverse tools for wrangling data have expedited our transformation as coders and made R less intimidating to learn.”

[@lowndes2017our]

“tidyverse”, which serves as an umbrella for many of the tidyverse packages.

In addition to the original tools in the tidyverse, many people have developed tidyverse extensions—R packages that build off the tools and principles in the tidyverse. These often bring the tidyverse conventions into tools for specific areas of science. For example, the tidytext package provides tools to analyze large datasets of text, including books or collections of tweets, using the tidy data format and tidyverse-style tools. Similar tidyverse extensions exist for working with network data (tidygraph) or geospatial data (sf). Extensions also exist for the visualization branch of the tidyverse specifically. These include ggplot extensions that allow users to create things like calendar plots (sugrrants), gene arrow maps (gggene), network plots (igraph), phylogenetic trees (ggtree) and anatogram images (gganatogram). These extensions all allow users to work with data that’s in a “tidy data” format, and they all provide similar user interfaces, making it easier to learn a large set of tools to do a range of data analysis and visualization, compared to if the set of tools lacked this coherence.

2.4 Designing templates for “tidy” data collection

This module will move from the principles of the “tidy” data format to the practical details of designing a “tidy” data format to use when collecting experimental data. We will describe common issues that prevent biomedical research datasets from being “tidy” and show how these issues can be avoided. We will also provide rubrics and a checklist to help determine if a data collection template complies with a “tidy” format.

Objectives. After this module, the trainee will be able to:

- Identify characteristics that keep a dataset from being “tidy”
- Convert data from an “untidy” to a “tidy” format

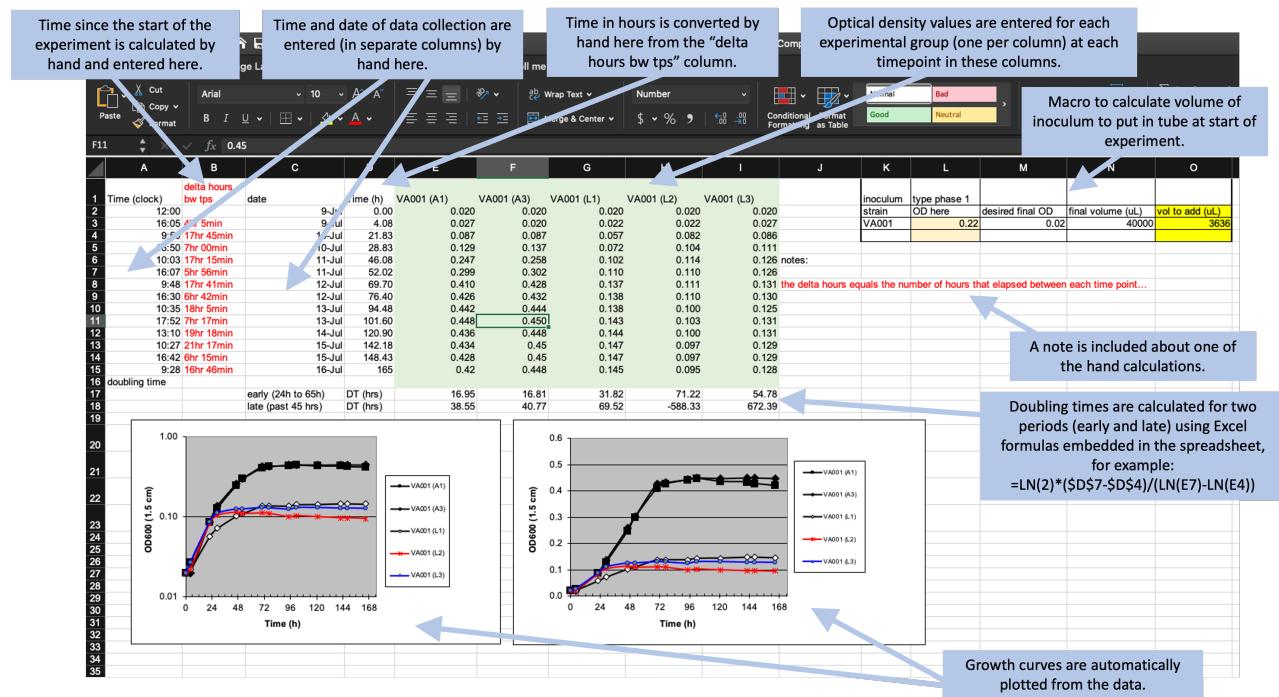
In this module, we will use a real example of data collected in a biomedical laboratory. We’ll use this example to show how data is often collected in a way that is not “tidy”, focusing on the features of data collection that make it “untidy”. We’ll then describe some general principles for why and how to instead create and use tidy (or at least tidier) templates to collect data in the laboratory, and show how this can be the first step in a pipeline to creating useful, attractive, and reproducible reports that describe the data you collected. This module will focus on the principles of templates for tidy data collection, while in the next module we’ll dig deeper into the details of making this conversion for the example dataset that we use as a demonstration in this module.

2.4.1 Example—Data on rate of bacterial growth

Throughout this module, we’ll use a real dataset to illustrate principles of data collection in a biomedical laboratory. First, let’s start by looking at the original

data collection template, and use this to walk through some details of this dataset.

Figure 2.1 provides an annotated view of the data set, showing the format used when the data were originally collected:



These data were collected to measure the compare growth yield and doubling time of *Mycobacterium tuberculosis* (the bacteria that causes tuberculosis in humans) under two conditions—high oxygen and low oxygen. In humans, *M. tuberculosis* can persist for years or decades in granulomas, and the centers of these granulomas are often hypoxic (low in oxygen). Therefore, it's important to understand how these bacteria grow in hypoxic conditions.

To conduct this experiment, the researchers used test tubes that were capped with sealed caps to prevent air exchange between the contents of the tube and the environment. Inside the tubes, the amount of oxygen was controlled by shifting the ratio of the volume of the culture (the liquid with nutrients in which the *M. tuberculosis* will grow) versus the volume of air. In the high oxygen condition, a lower volume of culture was used, which leaves room for a lot of air in the top of the tube. In the low oxygen condition, the tube was filled almost to the top with culture, which left very little air at the top of the tube.

Once the tubes were filled and capped, they were left to grow for about a week. During this time, the researchers took several measurements to determine the growth of the bacteria in each tube. To do this, they used a spec-

Figure 2.1: Example of an Excel spreadsheet used to record and analyze data for a laboratory experiment. Annotations highlight where data is entered by hand, where calculations are done by hand, and where embedded Excel formulas are used. The figures are created automatically using values in a specified column.

trophotometer to track increases in optical density (absorbance at 600 nm) over time. This method gives a measurement of turbidity in each tube that is directly proportional to the cell mass in the tube, and so provides a measure of how much the bacteria has grown since the start of the experiment.

To record data from this experiment, researchers used the spreadsheet shown in Figure 2.1. This spreadsheet is an example of a data collection template—it was created not only for this experiment, but also for other experiments that this research group conducts to measure bacterial growth under different conditions. It was designed to allow a researcher working in the laboratory to record measurements over the course of the experiment. This specific spreadsheet allowed the researcher who was conducting the experiment to (1) calculate the amount of initial inoculum (cell culture) to add to each tube to begin the study, (2) record the raw data absorbance measurements, (3) graph the data on both a log and linear scale, and (4) calculate doubling time in two phases of growth using the equation listed above.

Let's take a closer look at some of the features of this spreadsheet. First, it has a section on the top right that focuses on data collection during the experiment, with one row for each time when the tubes were measured for the cell mass within the tube. This section of the spreadsheet starts with several columns related to the time of each measurement, including the clock time at measurement (column A), the difference in time (hours) between each time point in which data were collected (column B), the date on which data were gathered (column C), and the time in hours for each data point from the start of the study for graphing purposes (column D). The columns for clock time (A) and date (C) were recorded by hand, while the columns for time since the start of the experiment (B and D) were calculated or converted by hand from these values and then entered in the column. The remaining columns (E–I) provide data on the optical density (absorbance at 600 nm), which is directly proportional to cell mass in the tube. There is one column per test tub, and each of these column labels includes a test tube ID (A1, A3, L1, L2, L3). If a tube ID starts with “A”, it was grown in high oxygen conditions, and if it starts with “L”, it was grown in low oxygen conditions.

Next, the spreadsheet has areas that provide summaries of the data, calculated using embedded formulas or through the spreadsheet’s plotting functions. For example, rows 17–18 provide calculations of the doubling time of the bacteria in each tube for two periods (early and late in the experiment), while two growth curves are plotted at the bottom of the spreadsheet.

Finally, the spreadsheet includes a couple of other features, including some written notes about one of the hand calculations and a macro in the top right that can be used by the researcher to calculate the amount of the initial inoculum to add to each tube at the start of the experiment.

What the researchers found appealing about the format of this spreadsheet was the ease with which the researcher collecting data in the laboratory could accomplish the study goals. They also cited transparency of the raw data and

ease with which additional sampling data points could be added. The data being graphed in real time, and the inclusion of a simple macro to calculate doubling time, allowed the research in the laboratory to see tangible differences between the two assay conditions as data were collected over the one-week experiment.

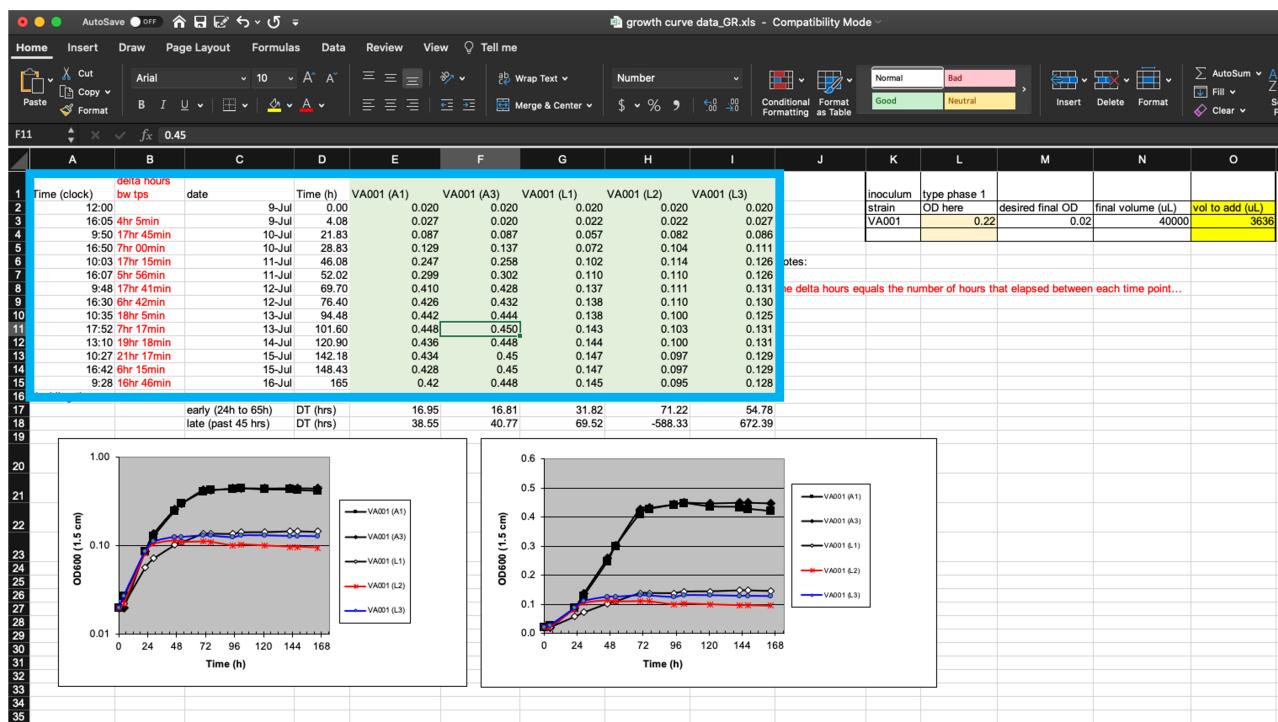
However, many of these features can have undesired consequences. They can increase the chance of errors in recording the data and in calculating summaries based on the data. They also make it hard to move the data into a reproducible pipeline, and so limit opportunities for more sophisticated analysis and visualization. In the next section of this module, we'll highlight features of data collection templates like this one that can make data collection "untidy". In the section after that, we'll discuss how you could create a new data collection template for this example data that would be tidier, and use this to open a more general discussion of principles of "tidy" data collection templates.

2.4.2 Features that make data collection templates "untidy"

There are several features of the data collection template shown in Figure 2.1 that make it "untidy", in the sense of making it difficult to integrate the collected data in a data analysis pipeline that includes reading the data into a statistical program like R, Perl, or Python to conduct data analysis and visualization. There are also features that make it prone to errors in data collection and analysis.

First, these data will be hard to read into a statistical program from this spreadsheet because the raw data (the time points each observation was collected and the optical density for the sample at that time point) form only part of the spreadsheet (Figure 2.2, area highlighted by the blue box). The "extra" elements on the spreadsheet, which include the output from calculations, plots, macros, and notes, make it harder to isolate the raw data from the file when using a statistical program.

While these extra elements make it hard to extract the raw data, it isn't impossible. Programming languages like R include functions to read data in from a spreadsheet, and these functions often provide options to specify the sheet of the file to read in, as well as the rows and columns to read from a specific sheet. In the example spreadsheet in Figure 2.2, for example, you could specify to read in only rows 1–15 of columns A–I, to focus on the raw data. However, one goal of reproducible research is to create tools and pipelines that are **robust**—that is, ones that still work as desired when the raw data is changed in small ways, or even across different raw data files. In later modules, in fact, we'll look at how we can use these principles to create tools that can be applied consistently across multiple studies to make data analysis of laboratory data both more efficient and reproducible. Therefore, while we could customize code to read in data from a specific part of a complex spreadsheet, like that shown in Figure 2.2, this customization would make the code less robust. If we asked the statistical program to read in rows 1–15 of columns A–I,



for example, the code would perform incorrectly if we later added one more time point to the experiment, or if we tried to use the same template for an experiment that used more test tubes. If we instead use a template that only records the raw data, without additional elements, then we can create more robust tools, since we can write code to read in whatever is in a spreadsheet, rather than restricting to certain rows and columns. Any analysis, summaries, or visualizations that we'd like to perform on the raw data can be done through reproducible reports—which we'll show an example of later for this example data—rather than directly in a spreadsheet.

Next, the example template helps demonstrate how specific ways of recording data can make the template less tidy. First, let's look at how the template records the time of each measurement. It does this using four separate columns (Figure 2.2). In column C, the researcher records the date a measurement was taken, and in Column A he or she records the clock time of the measurement. The experiment was started, for example, at 12:00 PM (“12:00” in column A) on July 9 (“9-Jul” in column C). These values are entered by hand by the researcher. Next, these values are used to calculate, for each measurement, how long it had been since the start of the experiment. This value is recorded in two separate ways—as hours and minutes in column B and converted into hours and percents of hours (using decimals) in column D. For example, the second measurement was taken at 4:05 PM on July 9 (“16:05” in column A and “9-Jul” in column C), which is 4 hours and 5 minutes after the start of the ex-

Figure 2.2: Isolating raw data collected in a template from extra elements. The box in this figure highlights the area of the spreadsheet where data are collected. All other elements of the spreadsheet focus on other aims (e.g., summarizing these data, adding notes, macros for experimental design). Those other elements make it difficult to extract the raw data for more advanced analysis and visualization through a statistical program like R, Python, or Perl.

periment (“4hr 5min” in column B) or, since 5 minutes is about 8% of an hour, 4.08 hours after the start of the experiment (“4.08” in column D).

	A	B	C	D	E	F
1	Time (clock)	delta hours bw tps	date	Time (h)	VA001 (A1)	VA001 (A3)
2	12:00			0.00	0.020	0.020
3	16:05	4hr 5min	9-Jul	4.08	0.027	0.020
4	9:50	17hr 45min	10-Jul	21.83	0.087	0.087
5	16:50	7hr 00min	10-Jul	28.83	0.129	0.137
6	10:03	17hr 15min	11-Jul	46.08	0.247	0.258
7	16:07	5hr 56min	11-Jul	52.02	0.299	0.302
8	9:48	17hr 41min	12-Jul	69.70	0.410	0.428
9	16:30	6hr 42min	12-Jul	76.40	0.426	0.432
10	10:35	18hr 5min	13-Jul	94.48	0.442	0.444
11	17:52	7hr 17min	13-Jul	101.60	0.448	0.450
12	13:10	19hr 18min	14-Jul	120.90	0.436	0.448
13	10:27	21hr 17min	15-Jul	142.18	0.434	0.45
14	16:42	6hr 15min	15-Jul	148.43	0.428	0.45
15	9:28	16hr 46min	16-Jul	165	0.42	0.448
16	doubling time					
17		early (24h to 65h)	DT (hrs)	16.95	16.81	
18		late (past 45 hrs)	DT (hrs)	38.55	40.77	
19						

Figure 2.3: Measurements of time in the example data collection template. The four highlighted columns (columns A, B, C, and D) are all used in this spreadsheet to record time. The methods of recording time in this template, however, may make it more likely to create errors in data recording and collection and will make it harder to use the data in a reproducible pipeline.

There are a few things that could be changed about how the time data are recorded here that could make this data collection template tidier. First, it would be better to focus only on recording the raw data, rather than adding calculations based on that data. Columns B and D in Figure 2.2 are both the output from calculations. Anytime a spreadsheet includes a calculation, it creates the room for mistakes in data collection and analysis. Often, calculations in a spreadsheet will be done using embedded formulas. These can cause problems anytime new columns or rows are added to the data, as that can shift the cells meant to be used in the calculation. Further, these formulas are embedded in the spreadsheet, where they can't be seen and checked very easily, which makes it easy to miss a typo or other error in the formula. In the example in Figure 2.2, columns B and D aren't calculated by embedded formulas, but rather calculated by the researcher by hand and then entered. This can create the room for user error with each calculation and each data entry. Later, we'll see how we can “tidy” this data collection template by removing columns that calculate time (columns B and D) and instead doing that calculation once the raw data are read into a statistical program.

The second thing that could be changed is how the template records the date and time of the measurement. Currently, it uses two columns (A and C) to record this information. However, each piece of information is useless without the other—instead, they must be known jointly to do things like calculate the time since the start of the experiment. It would therefore be tidier to record this information in a single column. For example, instead of record-

ing the starting time of the experiment as “12:00” in column A and “9-Jul” in column C, you could record it as “July 9, 2019 12:00” in a single date-time column. In this example, adding the year (“2019”) to the date will also make this data point easier to work with in a programming language, as these often have special functions to work with data in date-time classes, but all elements of the date and/or time must be included to convert data points into these useful classes.

Next, let’s look at how the template collects data related to cell growth in each tube (columns E–I, Figure 2.4).

	A	B	C	D	E	F	G	H	I	J
1	Time (clock)	delta hours bw tps	date	Time (h)	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)	
2	12:00			9-Jul	0.00	0.020	0.020	0.020	0.020	0.020
3	16:05	4hr 5min		9-Jul	4.00	0.027	0.020	0.022	0.022	0.027
4	9:50	17hr 45min		10-Jul	21.83	0.087	0.087	0.057	0.082	0.086
5	16:50	7hr 00min		10-Jul	28.83	0.129	0.137	0.072	0.104	0.111
6	10:03	17hr 15min		11-Jul	46.00	0.247	0.258	0.102	0.114	0.126
7	16:07	5hr 56min		11-Jul	52.00	0.299	0.302	0.110	0.110	0.126
8	9:48	17hr 41min		12-Jul	69.70	0.410	0.428	0.137	0.111	0.131
9	16:30	6hr 42min		12-Jul	76.40	0.426	0.432	0.138	0.110	0.130
10	10:35	18hr 5min		13-Jul	94.40	0.442	0.444	0.138	0.100	0.125
11	17:52	7hr 17min		13-Jul	101.60	0.448	0.450	0.143	0.103	0.131
12	13:10	19hr 18min		14-Jul	120.90	0.436	0.448	0.144	0.100	0.131
13	10:27	21hr 17min		15-Jul	142.10	0.434	0.45	0.147	0.097	0.129
14	16:42	6hr 15min		15-Jul	148.40	0.428	0.45	0.147	0.097	0.129
15	9:28	16hr 46min		16-Jul	160.00	0.42	0.448	0.145	0.095	0.128
16	doubling time									
17		early (24h to 65h)	DT (hrs)		16.95	16.81	31.82	71.22	54.78	
18		late (past 45 hrs)	DT (hrs)		38.55	40.77	69.52	-588.33	672.39	

These data are recorded in a format that will work pretty well. Strictly speaking, they aren’t fully “tidy” (module 2.3), since the column headers include information that we might want to use as variables in analysis and visualization. Specifically, each test tube’s ID is incorporated in the column name where measurements for that tube are recorded, since each test tube is recorded using a separate column. If we want to run analysis where we estimate values for each test tube, or create plots where each test tube’s measurements are shown with a separate line, then we’ll need to convert the format of the data a bit. However, that’s quite easy to do in more statistical programming languages now, and so it’s reasonable to compromise on this element of “tidiness” in the data collection format. As we’ll show in the next module, changing this layout in the original data collection would require the researcher to re-type the measurement date and time several times and would result in the spreadsheet being longer, and so harder to see at once when recording data. We’ll discuss

Figure 2.4: Measurements of bacterial growth in the example data collection template. The five highlighted columns (columns E–I) are all used in this spreadsheet to record optical density in each test tube at each measurement time.

this balance in designing data collection templates more in the next module, when we create a tidier version of this example data collection template.

There is a final element we'd like to highlight on this example template that could make the data hard to integrate into a reproducible pipeline. There are cases in the example template where either column names or cell values are formatted in a way that would be hard to work with when the data is read into a more advanced program like R or Python (Figure 2.5). For example, the column names include spaces and parentheses (e.g., "Time (clock)"). If left as-is, when the data are read into another program, the column names will need to be cleaned up to take these characters out, so that the column names are composed only of alphabetical characters, numbers, or underscores. While this can be done in code like R or Python, it will add to the data cleaning process and could be avoided by using simpler column names in the original data collection template. Similarly, in the example template there are recordings of time in a format that combines numbers with text indicators for units (e.g., "4hr 5min"). While these could be parsed in a programming language, it will take extra code and could be avoided with a better design for recording the data. Also, some of the data in the template is recorded in a format that Excel might try to automatically change into a date (e.g., "9-Jul"). Even if the value is a date, it is better to avoid formats that Excel automatically converts. Excel could, for example, convert the date incorrectly (e.g., convert "12/3/2020" to December 3, 2020, when it was meant to represent March 12, 2020). It is better to record the data in a format that will pass unchanged through to the file that you read in later coding and analysis.

Column names include special characters, like spaces and parentheses.

Time information is recorded in a format that combines letters and numbers and may be hard to parse later in an analysis pipeline.

Spreadsheet program may try to autoformat these entries as dates, which could cause problems later in the analysis pipeline.

A	B	C	D	E	F	G	H	I
Time (clock)	delta hours bw tps	date	Time (h)	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)
12:00			9-Jul	0.00	0.020	0.020	0.020	0.020
16:05	4hr 5min		9-Jul	4.08	0.027	0.020	0.022	0.027
9:50	17hr 45min		10-Jul	21.83	0.087	0.087	0.057	0.082
16:50	7hr 00min		10-Jul	28.83	0.129	0.137	0.072	0.104
10:03	17hr 15min		11-Jul	46.08	0.247	0.258	0.102	0.114
16:07	5hr 56min		11-Jul	52.02	0.299	0.302	0.110	0.126
0:48	2hr 44min		10-Jul	60.70	0.410	0.420	0.127	0.141

Figure 2.5: Examples of special characters and formatting in the example template that could cause problems later in a data analysis pipeline.

2.4.3 Converting to a “tidier” format for data collection templates

Now that we’ve looked at characteristics that can make a data collection template “untidy”, let’s go through some principles for creating “tidy” templates to record the same data. There are three basic principles for designing “tidy” templates that will go a long way to creating ways to collect data in a research group that can be easily used within a reproducible analysis pipeline. These three principles are:

1. Limit the template to the collection of data.
2. Make sensible choices when dividing data collection into rows and columns.
3. Avoid characters or formatting that will make it hard for a computer program to process the data.

The first principle in designing a tidier template for collecting laboratory data is to **limit the template to the collection of data**. The key here is the word “collection”. A tidy template will avoid any calculations done on the original data and instead focus only on the initial data that the researcher records for the experiment. This means that you should exclude from the template any element that provides a calculation, summary, or plot based on the initial recorded element. You should also exclude any special formatting that you are using to encode information. For example, say that you are collecting data, and in some cases you get a warning that the reading may be below the instrument’s detection limit. It may be tempting to highlight the cells with measurements where this warning was displayed as you record the data. However, you should avoid doing this, as any color or other formatting information will be lost when you read the data in the file into a statistical program. Instead, you could add a second column to indicate if the measurement included a warning.

The second principle is to **make sensible choices when dividing data collection into rows and columns**. There are many different ways that you could spread the data collection into rows and columns. One decision is how (and whether) to divide recorded information across columns. Figure 2.6, for example, shows several ways that you could divide data on a date and time into one or more columns. In this example, it typically makes the most sense to use a single column to record all the date and time elements (the top example in Figure 2.6). Most statistical programs have powerful functions for parsing dates and times, after which they store these data in special classes that allow time-related operations (for example, calculating the time difference between two date-time measurements). It will be most efficient to record all date and time elements in a single column.

Conversely if you have complex data with different elements (for example, height in components of inches and feet), it may make sense to use separate columns for each of the components. For example, rather than using one column to record 5'7", you could divide the information into one column with the component that is in feet (5) and one with the component in inches (7). In

The figure consists of three separate tables, each with a row number column on the left.

- Table 1:** A single column 'date_time' containing the value "October 23, 2008 8:05 PM". A blue arrow points from this table to a box stating: "All date and time elements included in one column."
- Table 2:** Two columns: 'date' (containing "October 23, 2008") and 'time' (containing "8:05 PM"). A blue arrow points from this table to a box stating: "One column for date elements and one column for time elements."
- Table 3:** Six columns: 'year' (containing 2008), 'month' (containing "October"), 'day' (containing 23), 'hour' (containing 8), 'minute' (containing 5), and 'am_or_pm' (containing "PM"). A blue arrow points from this table to a box stating: "Separate column for each element of date and time."

the first case, when you read the data into a program like R you would need to use complex code to split the value into its parts to be able to use it. In the second case, you could easily work with the values in the two separate columns to calculate a value to use in further work (e.g., use a formula like `height_ft * 12 + height_in` to calculate the full height in inches).

Another decision at this stage is how “long” versus “wide” you make your template. A “wide” design will include more columns, while a “long” design will include more rows. Often, you can create different designs that allow you to collect the same values but with different designs on this wide-versus-long spectrum. Figure 2.7 gives two examples of templates that collect the same data, but one is using a wider design and the other is using a longer design.

In module 2.3, we described the rules for the “tidy” format for dataframes. If you record data directly into a “tidy” format, it will be very easy to read into a programming language to analyze and visualize. However, this tidy format can sometimes result in datasets that are very long. It may be more convenient to record data into a wider format, especially if you are recording the data in a laboratory setting where it is inconvenient to scroll up and down within a longer-format file. Fortunately, there are some convenient tools in programs like R and Python that can be used to take data that are collected in a wider format and reformat them to the tidy format as soon as they are read into the software program. While this will require some extra code, it is usually code that is fairly simple and straightforward. Therefore, when you design your data collection template, you can balance any practical advantages of using a wider data collection format against the advantages of a fully “tidy” format that apply once your input the data into a statistical program for analysis and visualization. Often, the wider format might win out in this balance, and that’s fine.

The third principle is to **avoid characters or formatting that will make it hard for a computer program to process the data**. This principle is particularly important for the column names for each column. When you read

Figure 2.6: Examples of special characters and formatting in the example template that could cause problems later in a data analysis pipeline.

Column headers identify the test tube

Measurements for all test tubes are recorded in a single column

Separate columns for each test tube

A separate column specifies which test tube the measurement represents

	A	B	C	D	E	F
1	Date and time	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)
2	"July 9, 2019 12:00"	0.020	0.020	0.020	0.020	0.020
3	"July 9, 2019 16:05"	0.027	0.020	0.022	0.022	0.027
4	"July 10, 2019 9:50"	0.087	0.087	0.057	0.082	0.086
5	"July 10, 2019 16:50"	0.129	0.137	0.072	0.104	0.111
6	"July 11, 2019 10:03"	0.247	0.258	0.102	0.114	0.126
7	"July 11, 2019 16:07"	0.299	0.302	0.110	0.110	0.126
8	"July 12, 2019 9:48"	0.410	0.428	0.137	0.111	0.131
9	"July 12, 2019 16:30"	0.426	0.432	0.138	0.110	0.130
10	"July 13, 2019 10:35"	0.442	0.444	0.138	0.100	0.125
11	"July 13, 2019 17:52"	0.448	0.450	0.143	0.103	0.131
12	"July 14, 2019 13:10"	0.436	0.448	0.144	0.100	0.131
13	"July 15, 2019 10:27"	0.434	0.45	0.147	0.097	0.129
14	"July 15, 2019 16:42"	0.428	0.45	0.147	0.097	0.129
15	"July 16, 2019 9:28"	0.42	0.448	0.145	0.095	0.128

	A	B	C
1	Date and time	Test tube	Absorbance at 600 nm
2	"July 9, 2019 12:00"	VA001 (A1)	0.020
3	"July 9, 2019 12:00"	VA001 (A3)	0.020
4	"July 9, 2019 12:00"	VA001 (L1)	0.020
5	"July 9, 2019 12:00"	VA001 (L2)	0.020
6	"July 9, 2019 12:00"	VA001 (L3)	0.020
7	"July 9, 2019 16:05"	VA001 (A1)	0.027
8	"July 9, 2019 16:05"	VA001 (A3)	0.020
9	"July 9, 2019 16:05"	VA001 (L1)	0.022
10	"July 9, 2019 16:05"	VA001 (L2)	0.022
11	"July 9, 2019 16:05"	VA001 (L3)	0.027
12	"July 10, 2019 9:50"	VA001 (A1)	0.087
13	"July 10, 2019 9:50"	VA001 (A3)	0.087
14	"July 10, 2019 9:50"	VA001 (L1)	0.057
15	"July 10, 2019 9:50"	VA001 (L2)	0.082
16	"July 10, 2019 9:50"	VA001 (L3)	0.086
17	"July 10, 2019 16:50"	VA001 (A1)	0.129
18	"July 10, 2019 16:50"	VA001 (A3)	0.137
19	"July 10, 2019 16:50"	VA001 (L1)	0.072
20	"July 10, 2019 16:50"	VA001 (L2)	0.104
21	"July 10, 2019 16:50"	VA001 (L3)	0.111
22	"July 11, 2019 10:03"	VA001 (A1)	0.247
23	"July 11, 2019 10:03"	VA001 (A3)	0.258
24	"July 11, 2019 10:03"	VA001 (L1)	0.102

data into a statistical program like R, these names will automatically be used as the column names in the R data frame object, and the code will regularly use these column names to refer to parts of the data when analyzing and visualizing it. You will find it easiest to use the data in a reproducible pipeline if you follow a couple rules for the column names. The reason that these rules will help is that they replicate the rules for naming objects in programming languages, and so will help in seamlessly transitioning between the stages of data collection and data analysis. First, always start a column name with a letter. Second, only use letters, numbers, or the underscore character ("_") for the rest of the characters in the column name.

Based on these rules, then, you should avoid putting spaces in your column names when you design a data collection template. It is tempting to include spaces to make the names clearer for humans to read, and this is understandable. Often, using an underscore in place of a space can allow for easy human comprehension while still avoiding characters that are difficult for statistical programs. For example, if you have a column named "Optical density", you can change it to "Optical_density" without making it much more difficult for a person to understand. As with other choices in designing a data collection template, these choices about column names can be a balance between making the template easy for researchers to use in the laboratory and easy for the statistical program to parse later in the pipeline. For example, statistical programs

Figure 2.7: Examples of two ways arranging the same data in a data recording template. The format on the left records the optical density measurements for each test tube in a separate column, and the column header identifies the test tube. This is an example of a 'wider' format. The format on the right records the optical density for all test tubes in a single column, using a separate column to record which test tube the measurement represents. This is an example of a 'longer' format.

like R have functions for working with character strings that can be used to replace all the spaces in column names with another character. However, if it isn't unreasonable to follow the recommended rules in writing column names for the data collection template, you can keep code later in the pipeline much simpler, so it's worth considering.

Beyond spaces, there are a number of other special characters that you might be tempted to include in column names. These could include parentheses, dollar signs, percent signs, hash marks ("#"), and so on. Any of these will require extra code in later steps of an analysis pipeline, and some can cause more severe problems because they have special functionality in the programming language. For example, hash marks are used in the R programming language to add comments within code, while dollar signs are used for subsetting elements of a list or data frame object. It is worth the effort to avoid all these characters in column names in a data collection template.

There are also considerations you can make in terms of how you record data within cells of the data collection template, and these can make a big difference in terms of how hard or easy it is to work with the data within a statistical program. While statistical programs like R are very powerful in terms of being able to handle even very "messy" input data, they require a lot of code to leverage this power. By being thoughtful when you design the template to record the data, you can avoid having to use a lot of code to input and clean the data in later stages of the pipeline.

Figure 2.8 gives an example of a choice that you could make in the format you use to record data. This figure shows two columns from the original data collection template from the example experiment for this module. This template includes two columns that record the time since the start of the experiment, and they use different formats for doing this. In column B, time is recorded in hours and minutes, with the characters "hr" and "min" used to separate the two time components. In column D, the same information is recorded, but in decimals of hours (e.g., 4.08 hours for 4 hours and 5 minutes). While the format in column B is more similar to how humans think of time, it will take more code to parse in a statistical program. When reading this data into a program like R, you would need to use regular expressions to split apart the different elements and then recombine them into a format that the program understands. By contrast, the values recorded in column D could be easily read in by a statistical program, with minimal code needed before they could be used in analysis and visualizations.

Finally, when you are designing the data collection template, you should try to avoid using formats that may be "auto-converted" by the spreadsheet program. For example, if you enter a value like "7-9-19" into a cell, the spreadsheet may try to automatically convert it to a date. Perhaps it is a date, but even if it is, the spreadsheet algorithm might make problematic assumptions in the conversion. For example, it might assume that "7-9-19" means July 9, 2019, when you meant for it to represent September 7, 1919. Further, there are

	A	B	C	D	E	F	
1	Time (clock)	delta hours bw tps	date	Time (h)	VA001 (A1)	VA001 (A3)	VA001
2	12:00		9-Jul	0.00	0.020	0.020	
3	16:05	4hr 5min	9-Jul	4.08	0.027	0.020	
4	9:50	17hr 45min	10-Jul	21.83	0.087	0.087	
5	16:50	7hr 00min	10-Jul	28.83	0.129	0.137	
6	10:03	17hr 15min	11-Jul	46.08	0.247	0.258	
7	16:07	5hr 56min	11-Jul	52.02	0.299	0.302	
8	9:48	17hr 41min	12-Jul	69.70	0.410	0.428	
9	16:30	6hr 42min	12-Jul	76.40	0.426	0.432	
10	10:35	18hr 5min	13-Jul	94.48	0.442	0.444	
11	17:52	7hr 17min	13-Jul	101.60	0.448	0.450	
12	13:10	19hr 18min	14-Jul	120.90	0.436	0.448	
13	10:27	21hr 17min	15-Jul	142.18	0.434	0.45	
14	16:42	6hr 15min	15-Jul	148.43	0.428	0.45	
15	9:28	16hr 46min	16-Jul	165	0.42	0.448	
16	doubling time						
17		early (24h to 65h)	DT (hrs)		16.95	16.81	
18		late (past 45 hrs)	DT (hrs)		38.55	40.77	
19							

Figure 2.8: Examples of two ways of recording time in the original template from the example experiment. Column B uses hours and minutes, with characters embedded to separate hours from minutes, while column D uses hours in decimal degrees. The format in column D will be much easier to integrate into a larger data analysis pipeline.

cases where you might enter a value that is not a date, but that the spreadsheet thinks is based on its formatting. This was found to be a problem, for example, for some gene names. To avoid potential autoconversion by the spreadsheet, consider putting any character strings, including entries with dates and identifiers, inside quotation marks when you enter them in the spreadsheet program. The spreadsheet program will respect this as a sign to leave the entry as-is, rather than attempting automatic formatting into a date or other special class of data.

These three principles are an excellent starting point for designing a “tidy” template for collecting data. By using these, you will be well on your way to collecting data in a way that is easy to integrate in a longer reproducible data analysis pipeline. There are some additional steps that you could consider that can help make it easier to do clever and interesting things with your data once you read it into a statistical program.

For example, you could design column names and column entries so that you will be able to take advantage of statistical programming tools based on something called regular expressions. “Regular expressions” refers to patterns in character strings (which are just strings of one or more characters, like “aerated_1” or “mouseID”) that can be described and searched for using defined patterns. For example, take the following set of character strings: “aerated1”, “aerated3”, “low_oxygen1”, “low_oxygen2”, “low_oxygen3”. These strings currently include two pieces of information. First, they give the growth condition, which is either “aerated” or “low_oxygen”. This information

is given in each string using only alphabetical characters (e.g., a, b, c) and the underscore character. Next, the strings include information on the test tube of the sample for that condition—for example, “aerated1” indicates the first test tube under the aerated conditions. This test tube number is given using only numerical characters (e.g., 1, 2, 3). Since these pieces of information are encoded in the character strings using these patterns, you can use regular expressions in a program like R to isolate only the non-numeric part of each string (“aerated” versus “low_oxygen”) or only the number part (“1”, “2”, or “3”). This functionality can be a very powerful way to use column names and cell values to encode information that you can later separate or extract to use in things like adding color to plots based on certain conditions. In the next module, we’ll show an example of using regular expressions in this way to leverage information taken when collecting the data. When designing a tidy data collection template, it’s worthwhile to think of writing column names or otherwise recording data in a way that uses these types of regular patterns in a meaningful way.

When you convert data collection templates to “tidier” formats, they will typically look much simpler than the templates that your research group may have been using. In the example experiment that we described earlier in this module, this process of tidying the template results in a template like that shown in Figure 2.1 (in the next module, we’ll walk through all the steps to create this tidier template, using this principles we’ve covered in this module). By comparison, the starting template for data collection for this experiment is shown in Figure 2.1.

By comparing these two templates, you can see that the simpler template does not, by itself, provide immediate, real-time summaries of the collected data. The simpler template has removed elements like plots and values calculated by embedded formulas. At first glance, this might seem like a disadvantage of using a tidier template to collect data. However, by combining other tools in a pipeline, it is easy to connect the tidier raw data file to reporting tools. In this way, you can quickly create real-time summaries of the data that are similar to those shown in Figure 2.1, but that are created and reported outside the file used to originally record the data.

Figure 2.10 shows an example of a simple report that could be created for the example experiment. This report is generated using a statistical program, R, which inputs the data from the simple template shown in Figure 2.9. The report then uses R code to generate a PDF or Word file with the output shown below. The file for this report is created in a way that the output can be quickly regenerated with a single button click, and so it can be applied to other data saved using the same template. In fact, you can create templates for reports that coordinate with each data collection template that you create. In the next module, we’ll walk through how you could create the generating file for this report, and in later modules (3.7–3.9), we provide a thorough overview of creating these types of “knitted” documents.

Time and date of data collection are entered in a single column here.

Optical density values are entered for each experimental group (one per column) at each timepoint in these columns.

	B	C	D	E	F	
1	sampling_date_time	aerated1	aerated3	low_oxygen1	low_oxygen2	low_oxygen3
2	"July 9, 2019 12:00"	0.020	0.020	0.020	0.020	0.020
3	"July 9, 2019 16:05"	0.027	0.020	0.022	0.022	0.027
4	"July 10, 2019 9:50"	0.087	0.087	0.057	0.082	0.086
5	"July 10, 2019 16:50"	0.129	0.137	0.072	0.104	0.111
6	"July 11, 2019 10:03"	0.247	0.258	0.102	0.114	0.126
7	"July 11, 2019 16:07"	0.299	0.302	0.110	0.110	0.126
8	"July 12, 2019 9:48"	0.410	0.428	0.137	0.111	0.131
9	"July 12, 2019 16:30"	0.426	0.432	0.138	0.110	0.130
10	"July 13, 2019 10:35"	0.442	0.444	0.138	0.100	0.125
11	"July 13, 2019 17:52"	0.448	0.450	0.143	0.103	0.131
12	"July 14, 2019 13:10"	0.436	0.448	0.144	0.100	0.131
13	"July 15, 2019 10:27"	0.434	0.45	0.147	0.097	0.129
14	"July 15, 2019 16:42"	0.428	0.45	0.147	0.097	0.129
15	"July 16, 2019 9:28"	0.42	0.448	0.145	0.095	0.128

Figure 2.9: Example of a simpler format that can be used to record and analyze data

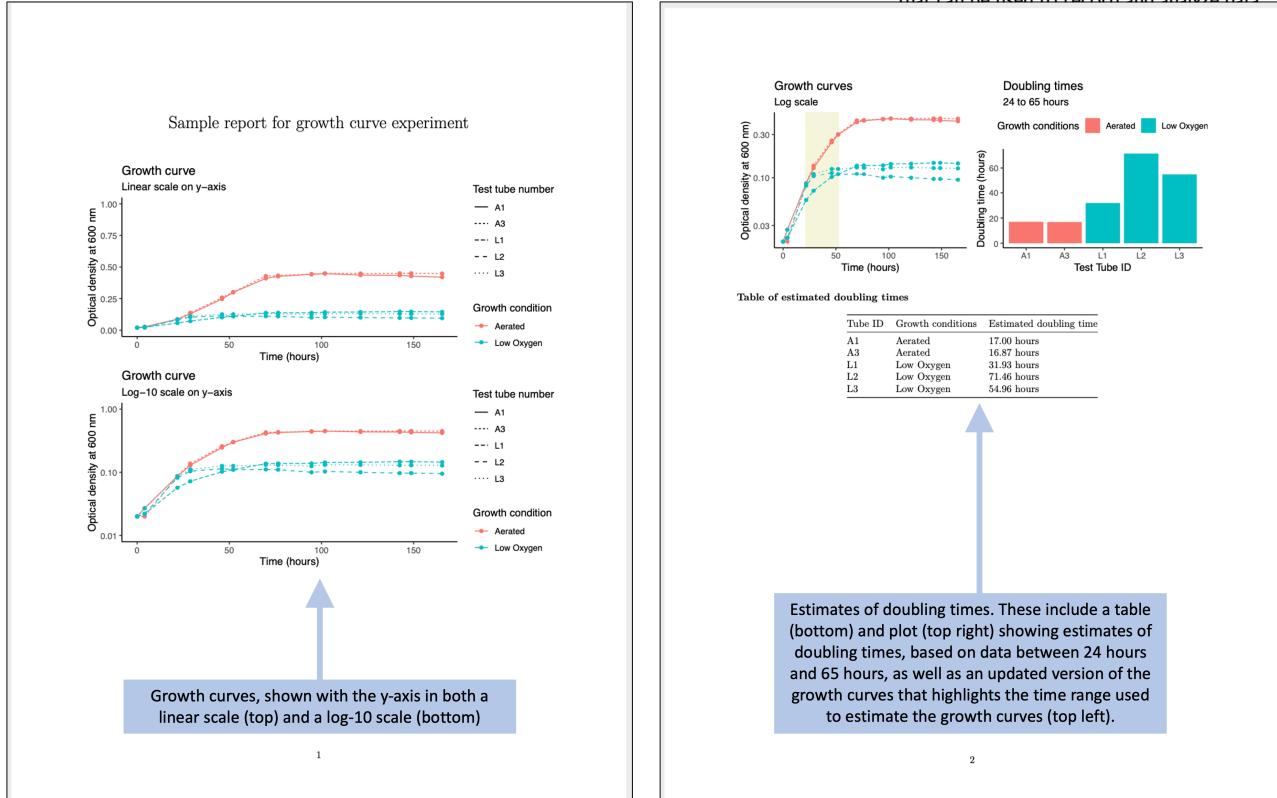


Figure 2.10: Examples of an automated report that can be created to quickly generate summaries and estimates of the data collected in the simplified data collection template for the example experiment.

The report shown in Figure 2.10 repeats some of the same summaries that were shown in the more complex original data collection template (Figure 2.1). There are a number of advantages, however, to using separate steps and files for the processes of collecting versus analyzing the data. The separate report (Figure 2.1) provides a starting point that can be easily adapted to make more complex figures and analysis, as well as to integrate the collected data with data measured in other ways for the experiment.

For example, take a look at the graph in the top left corner on the second page of the report shown in Figure 2.10. This figure shows the growth curve from the collected data, and it adds a shaded area to show the time range that was used to estimate doubling times for each sample. This provides a helpful quality check for this experiment. Bacterial growth goes through several phases, including an initial lag phase, an exponential growth phase (when the bacteria are regularly doubling), a stationary phase (when growth starts to slow down, because of exhaustion of nutrients or buildup of waste), and a dying phase. The doubling time should be calculated only during the exponential phase of growth, as the equation used to calculate it relies on describing growth during a period of regular doubling. When the growth curve is plotted with a log scale on the y-axis, the growth curve will look approximately linear in this exponential growth region. By including the plot on the top left of the second page in Figure 2.10, the researcher can quickly see that, in this experiment, the selected time range for calculating doubling time might not be appropriate—for the low oxygen condition, in particular, this time range looks like it included some measurements made during the transition into the stationary phase of growth. By quickly being able to assess this, the researcher can reassess whether a different time range should be used to calculate the doubling time for this experiment.

The report shown in Figure 2.10 provides results that are very similar to those calculated in the original spreadsheet, to show that you don't need to give up fast and clear summaries and visuals if you simplify the template for collecting data. However, this report template could easily be made more sophisticated. For example, you could add code into the report that would perform quality control checks. In the example case, the cell growth is measured using optical density, and while this measure is proportional to cell density in many cases, the measurement can be prone to error once the optical density is very high. Therefore, you could, for example, add a check into the report to highlight any measures of optical density that are higher than a certain value.

2.4.4 Learning more about tidy data collection in the laboratory

It may take some iteration to develop the data collection templates that are both convenient and appropriate to input to more complex programs for pre-processing, analysis, and visualization. This module and the next module provide guidance and examples, but it can be helpful to see more examples.

Two excellent resources on this topic are articles by Ellis and Leek (2018) and Broman and Woo (2018).

2.5 Example: Creating a template for “tidy” data collection

We will walk through an example of creating a template to collect data in a “tidy” format for a laboratory-based research project, based on a research project on drug efficacy in murine tuberculosis models. We will show the initial “untidy” format for data recording and show how we converted it to a “tidy” format. Finally, we will show how the data can then easily be analyzed and visualized using reproducible tools.

Objectives. After this module, the trainee will be able to:

- Understand how the principles of “tidy” data can be applied for a real, complex research project;
- List advantages of the “tidy” data format for the example project

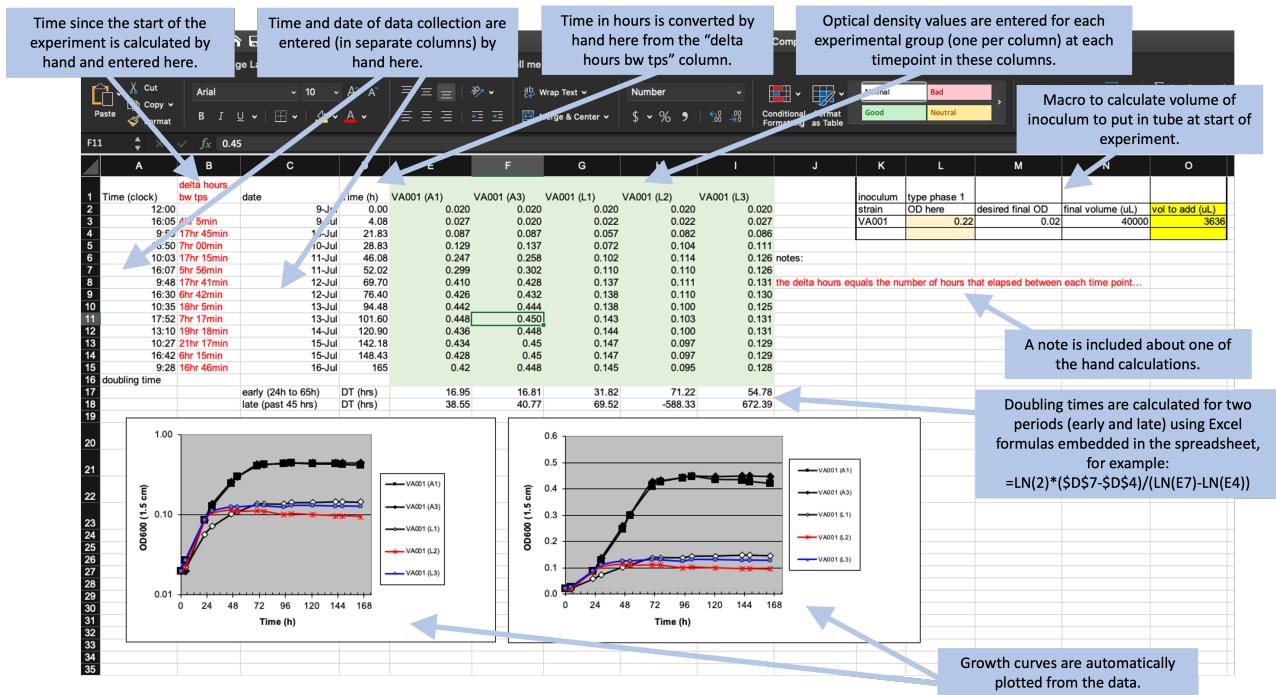
In the last module, we covered three principles for designing tidy templates for data collection in a biomedical laboratory, motivated by an example dataset from a real experiment. In this module, we’ll show you how to apply those principles to create a tidier template for the example dataset from the last module. As a reminder, those three principles are:

1. Limit the template to the collection of data.
2. Make sensible choices when dividing data collection into rows and columns.
3. Avoid characters or formatting that will make it hard for a computer program to process the data.

It is important to note that there’s no reason that you can’t continue to use a spreadsheet program like Excel or Google Sheets to collect data. The spreadsheet program itself can easily be used to create a simple template to use as you collect data. In fact, we’ll continue using a spreadsheet format in the rest of this module and in the next one as we show how to redesign the data collection for this example experiment. It is important, however, to think through how you will arrange that template spreadsheet to make it most useful in the larger context of reproducible research.

2.5.1 Example data—Data on rate of bacterial growth

Here, we’ll walk through an example using real data collected in a laboratory experiment. We described these data in detail in the previous module. As a reminder, they were collected to measure the growth rate of *Mycobacteria tuberculosis* under two conditions—high oxygen and low oxygen. They were collected from five test tubes that were measured regularly over one week for bacteria growth using a measure of optical density. Figure 2.11 shows the original template that the research group used to record these data.



In the previous module, we described features that make this template “untidy” and potentially problematic to include in a larger pipeline of reproducible research. In the next few sections of this module, we’ll walk step-by-step through changes that you could make to make this template tidier. We’ll finish the module by showing how you could then easily design a further step of the analysis pipeline to visualize and analyze the collected data, so that the advantages of real-time plotting from the more complex spreadsheet are not missed when moving to a tidier template.

2.5.2 Limiting the template to the collection of data

The example template (Figure 2.11) includes a number of “extra” elements beyond simple data collection—all the elements outside rows 1–15 of columns A–I. Outside this area of the original spread, there are a number of extra elements, including plots that visualize the data, summaries generated based on the data (rows 16–18, for example), notes about the data, and even a macro (top right) that wasn’t involved in data collection but instead was used by the researcher to calculate the initial volume of inoculum to include in each test tube. None of these “extras” can be easily read into a statistical program like R or Python—at best, they will be ignored by the program. They can even complicate reading in the cells with measurements (rows 1–15 of columns A–I), as most statistical programs will try to read in all the non-empty cells of a spreadsheet unless directed otherwise.

Figure 2.11: Example of an Excel spreadsheet used to record and analyze data for a laboratory experiment. Annotations highlight where data is entered by hand, where calculations are done by hand, and where embedded Excel formulas are used. The figures are created automatically using values in a specified column.

A good starting point, then, would be to start designing a tidy data collection template for this experiment by extracting only the content from the box in Figure 2.2. This would result in a template that looks like Figure 2.12.

	A	B	C	D	E	F	G	H	I	J
1	Time (clock)	delta hours bw tps	date	Time (h)	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)	
2	12:00		9-Jul	0.00	0.020	0.020	0.020	0.020	0.020	
3	16:05 4hr 5min		9-Jul	4.08	0.027	0.020	0.022	0.022	0.027	
4	9:50 17hr 45min		10-Jul	21.83	0.087	0.087	0.057	0.082	0.086	
5	16:50 7hr 00min		10-Jul	28.83	0.129	0.137	0.072	0.104	0.111	
6	10:03 17hr 15min		11-Jul	46.08	0.247	0.258	0.102	0.114	0.126	
7	16:07 5hr 56min		11-Jul	52.02	0.299	0.302	0.110	0.110	0.126	
8	9:48 17hr 41min		12-Jul	69.70	0.410	0.428	0.137	0.111	0.131	
9	16:30 6hr 42min		12-Jul	76.40	0.426	0.432	0.138	0.110	0.130	
10	10:35 18hr 5min		13-Jul	94.48	0.442	0.444	0.138	0.100	0.125	
11	17:52 7hr 17min		13-Jul	101.60	0.448	0.450	0.143	0.103	0.131	
12	13:10 19hr 18min		14-Jul	120.90	0.436	0.448	0.144	0.100	0.131	
13	10:27 21hr 17min		15-Jul	142.18	0.434	0.45	0.147	0.097	0.129	
14	16:42 6hr 15min		15-Jul	148.43	0.428	0.45	0.147	0.097	0.129	
15	9:28 16hr 46min		16-Jul	165	0.42	0.448	0.145	0.095	0.128	
16										
17										
18										

Notice that we've also removed any of the color formatting from the spreadsheet. It is fine to keep color in the spreadsheet if it will help the research to find the right spot to record data while working in the laboratory, but you should make sure that you're not using it to encode information about the data—all color formatting will be ignored when the data are read by a statistical program like R.

While the template shown in Figure 2.12 has removed a lot of the calculated values from the original template, it has not removed all of them. Two of the columns are still values that were determined by calculation after the original data were collected. Column B and column D both provide measures of the length of time since the start of the experiment, and both are calculated by comparing a measurement time to the time at the start of the experiment.

The time since the start of the experiment can easily be calculated later in the analysis pipeline, once you read the data into a statistical program like R. By delaying this step, you can both simplify the data collection template (requiring fewer columns for the research in the laboratory to fill out) and also avoid the chance for mistakes, which could occur both in the hand calculations of these values and in data entry, when the researcher enters the results of the calculations in the spreadsheet cell. Figure 2.13 shows a new version of the template, where these calculated columns have been removed. This template is now restricted to only data points originally collected in the course of the experiment, and has removed all elements that are based on calculations or

Figure 2.12: First step in designing a tidy data collection template for the example project. A template has been created that focuses only on the raw data, removing all extra elements like plots, notes, macros, and summaries.

other derivatives of those original, raw data points.

	A	B	C	D	E	F	G
1	Time (clock)	date	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)
2	12:00	9-Jul	0.020	0.020	0.020	0.020	0.020
3	16:05	9-Jul	0.027	0.020	0.022	0.022	0.027
4	9:50	10-Jul	0.087	0.087	0.057	0.082	0.086
5	16:50	10-Jul	0.129	0.137	0.072	0.104	0.111
6	10:03	11-Jul	0.247	0.258	0.102	0.114	0.126
7	16:07	11-Jul	0.299	0.302	0.110	0.110	0.126
8	9:48	12-Jul	0.410	0.428	0.137	0.111	0.131
9	16:30	12-Jul	0.426	0.432	0.138	0.110	0.130
10	10:35	13-Jul	0.442	0.444	0.138	0.100	0.125
11	17:52	13-Jul	0.448	0.450	0.143	0.103	0.131
12	13:10	14-Jul	0.436	0.448	0.144	0.100	0.131
13	10:27	15-Jul	0.434	0.45	0.147	0.097	0.129
14	16:42	15-Jul	0.428	0.45	0.147	0.097	0.129
15	9:28	16-Jul	0.42	0.448	0.145	0.095	0.128
16							

Figure 2.13: Second step in designing a tidy data collection template for the example project. This template started from the previous one, but removed columns that were hand-calculated and then entered by the researcher in the previous template. This version has removed all calculated values on the template, limiting it to only the original recorded values required for the experiment.

2.5.3 Making sensible choices about rows and columns

The second principle is to **make sensible choices when dividing data collection into rows and columns**. There are many different ways that you could spread the data collection into rows and columns, and in this step, you can consider which method would meet a reasonable balance between making the template easy for the researcher in the laboratory to use to record data and also making the resulting data file easy to incorporate in a reproducible data analysis pipeline.

For the example experiment, Figure 2.2 shows three examples that we can consider for how to arrange data collection across rows and columns. All three build on the changes we made in the earlier step of “tidying” the template, which resulted in the template shown in Figure 2.13.

Panel A (an exact repeat of the template shown in Figure 2.13) shows an example where date and time are recorded in different columns. Panel B is similar to Panel A, but in this case, date and time are recorded in a single column. Panel C shows a classically “tidy” data format, where each measurement’s date-time is repeated for each of the five test tubes, and columns give the test tube ID and absorbance measurement at that time for that tube (only part of the data is shown for this format, while remaining rows are off the page).

In this example, the template that may be the most reasonable is the one shown in Panel B. While Panel C provides the “tidiest” format, it has some practical constraints when used in a laboratory setting. For example, it would require more data entry during data collection (since date-time is entered five times at each measurement time), and its long format prevent it all from being seen at once without scrolling on a computer screen. When comparing Panels A and B, the template in Panel B has an advantage. The information on date and time are useful together, but not individually. For example, to

	A	B	C	D	E	F	G
1	Time (clock)	date	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)
2	12:00	9-Jul	0.020	0.020	0.020	0.020	0.020
3	16:05	9-Jul	0.027	0.020	0.022	0.022	0.027
4	9:50	10-Jul	0.087	0.087	0.057	0.082	0.086
5	16:50	10-Jul	0.129	0.137	0.072	0.104	0.111
6	10:03	11-Jul	0.247	0.258	0.102	0.114	0.126
7	16:07	11-Jul	0.299	0.302	0.110	0.110	0.126
8	9:48	12-Jul	0.410	0.428	0.137	0.111	0.131
9	16:30	12-Jul	0.426	0.432	0.138	0.110	0.130
10	10:35	13-Jul	0.442	0.444	0.138	0.100	0.125
11	17:52	13-Jul	0.448	0.450	0.143	0.103	0.131
12	13:10	14-Jul	0.436	0.448	0.144	0.100	0.131
13	10:27	15-Jul	0.434	0.45	0.147	0.097	0.129
14	16:42	15-Jul	0.428	0.45	0.147	0.097	0.129
15	9:28	16-Jul	0.42	0.448	0.145	0.095	0.128
16							

	A	B	C	D	E	F
1	Date and time	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)
2	"July 9, 2019 12:00"			0.020	0.020	0.020
3	"July 9, 2019 16:05"			0.027	0.020	0.022
4	"July 10, 2019 9:50"			0.087	0.087	0.082
5	"July 10, 2019 16:50"			0.129	0.137	0.072
6	"July 11, 2019 10:03"			0.247	0.258	0.102
7	"July 11, 2019 16:07"			0.299	0.302	0.110
8	"July 12, 2019 9:48"			0.410	0.428	0.137
9	"July 12, 2019 16:30"			0.426	0.432	0.138
10	"July 13, 2019 10:35"			0.442	0.444	0.138
11	"July 13, 2019 17:52"			0.448	0.450	0.143
12	"July 14, 2019 13:10"			0.436	0.448	0.144
13	"July 15, 2019 10:27"			0.434	0.45	0.147
14	"July 15, 2019 16:42"			0.428	0.45	0.147
15	"July 16, 2019 9:28"			0.42	0.448	0.145
16						

	A	B	C
1	Date and time	Test tube	Absorbance at 600 nm
2	"July 9, 2019 12:00"	VA001 (A1)	0.020
3	"July 9, 2019 12:00"	VA001 (A3)	0.020
4	"July 9, 2019 12:00"	VA001 (L1)	0.020
5	"July 9, 2019 12:00"	VA001 (L2)	0.020
6	"July 9, 2019 12:00"	VA001 (L3)	0.020
7	"July 9, 2019 16:05"	VA001 (A1)	0.027
8	"July 9, 2019 16:05"	VA001 (A3)	0.020
9	"July 9, 2019 16:05"	VA001 (L1)	0.022
10	"July 9, 2019 16:05"	VA001 (L2)	0.022
11	"July 9, 2019 16:05"	VA001 (L3)	0.027
12	"July 10, 2019 9:50"	VA001 (A1)	0.087
13	"July 10, 2019 9:50"	VA001 (A3)	0.087
14	"July 10, 2019 9:50"	VA001 (L1)	0.057
15	"July 10, 2019 9:50"	VA001 (L2)	0.082
16	"July 10, 2019 9:50"	VA001 (L3)	0.086
17	"July 10, 2019 16:50"	VA001 (A1)	0.129
18	"July 10, 2019 16:50"	VA001 (A3)	0.137
19	"July 10, 2019 16:50"	VA001 (L1)	0.072
20	"July 10, 2019 16:50"	VA001 (L2)	0.104
21	"July 10, 2019 16:50"	VA001 (L3)	0.111
22	"July 11, 2019 10:03"	VA001 (A1)	0.247
23	"July 11, 2019 10:03"	VA001 (A3)	0.258
24	"July 11, 2019 10:03"	VA001 (L1)	0.102
25			

Figure 2.14: Examples of ways that data collection could be divided into rows and columns in the example template. Panel A shows an example where date and time are recorded in different columns. Panel B is similar to Panel A, but in this case, date and time are recorded in a single column. Panel C shows a classically 'tidy' data format, where each measurement date-time is repeated for each of the five test tubes, and columns give the test tube ID and absorbance measurement at that time for that tube (only part of the data is shown for this format, while remaining rows are off the page). While Panel C provides the 'tidiest' format, it may have some practical constraints when used in a laboratory setting. For example, it would require more data entry during data collection (since date-time is entered five times at each measurement time), and its long format prevent it all from being seen at once without scrolling on a computer screen.

calculate the time since the start of the experiment, you cannot just calculate the difference in dates or just the difference in times, but instead must consider both the date and time of the measurement in comparison to the date and time of the start of the experiment. As a result, at some point in the data analysis pipeline, you'll need to combine information about the date and the time to make use of the two elements. While this combination of two columns can be easily done within a statistical program like R, it can also be directly designed into the original template for collecting the data. Therefore, unless there is a practical reason why it would be easier for the researcher to enter date and time separately, the template shown in Panel B is preferable to that shown in Panel A in terms of allowing for the “tidy” collection of research data into a file that is easy to include in a reproducible pipeline. Figure 2.15 shows the template design at this stage in the process of tidying it, highlighting the column that combines date and time elements in a single column. In this version of the template, we've also been careful about how date and time are recorded, a consideration that we'll discuss more in the next section.

Date and time combined in a single column and formatted to include all date-time elements.

	A	B	C	D	E	F
1	Date and time	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)
2	"July 9, 2019 12:00"	0.020	0.020	0.020	0.020	0.020
3	"July 9, 2019 16:05"	0.027	0.020	0.022	0.022	0.027
4	"July 10, 2019 9:50"	0.087	0.087	0.057	0.082	0.086
5	"July 10, 2019 16:50"	0.129	0.137	0.072	0.104	0.111
6	"July 11, 2019 10:03"	0.247	0.258	0.102	0.114	0.126
7	"July 11, 2019 16:07"	0.299	0.302	0.110	0.110	0.126
8	"July 12, 2019 9:48"	0.410	0.428	0.137	0.111	0.131
9	"July 12, 2019 16:30"	0.426	0.432	0.138	0.110	0.130
10	"July 13, 2019 10:35"	0.442	0.444	0.138	0.100	0.125
11	"July 13, 2019 17:52"	0.448	0.450	0.143	0.103	0.131
12	"July 14, 2019 13:10"	0.436	0.448	0.144	0.100	0.131
13	"July 15, 2019 10:27"	0.434	0.45	0.147	0.097	0.129
14	"July 15, 2019 16:42"	0.428	0.45	0.147	0.097	0.129
15	"July 16, 2019 9:28"	0.42	0.448	0.145	0.095	0.128
16						

Figure 2.15: Third step in designing a tidy data collection template for the example project. This template started from the previous one, but combined collection of the date and time of the measurement into a single column and revised the format to include all date elements and to prevent automatic conversion by the spreadsheet program.

2.5.4 Avoiding problematic characters or formatting

The third principle is to **avoid characters or formatting that will make it hard for a computer program to process the data**. There are a number of special characters and formatting conventions that can be hard for a statistical program to handle. In the example template shown in Figure 2.15, for example,

the column names include spaces (for example, in “Date and time”), as well as parentheses (for example, in “VA 001 (A1)”). While most statistical programs have tools that allow you to handle and convert these characters once the data are read in, it’s even simpler to use simpler column names in the original data collection template, and this will save some extra coding further along in the analysis pipeline. Two general rules for creating easy-to-use column names in a data collection template are: (1) start each column name with a letter and (2) for the rest of the column name, use only letters, numbers, or the underscore character (“_”). For example, “aerated I” would work well, but “I–aerated” would not.

Within the cell values below the column names, there is more flexibility. For example, if you have a column that gives the IDs of different samples, it would be fine to include spaces and other characters in those IDs.

There are a few exceptions, however. A big one is with values that record dates or date-time combinations. First, it is important to include all elements of the date (or date and time, if both are recorded). For example, the year should be included in the recorded date, even if the experiment only took a few days. This is because statistical programs have excellent functions for working with data that are dates or date-times, but to take advantage of these, the data must be converted into a special class in the program, and conversion to that class requires specific elements (for example, a date must include the year, month, and day of month).

Second, it is useful to avoid recording dates and date-times in a way that results in a spreadsheet program automatically converting them. Surrounding the information about a date in quotation marks when entering it (as shown in Figure 2.15) can avoid this.

Finally, consider using a format to record the date that is unambiguous and so less likely to have recording errors. Dates, for example, are sometimes recorded using only numbers—for example, the first date of “July 9, 2019” in the example data could be recorded as “7/9/2019” or “7/9/19”, to be even more concise. However, this format has some ambiguity. It can be unclear if this refers to July 9 or to September 7, both of which could be written as “7/9”. For the version that uses two digits for the year, it can be unclear if the date is for 2019 or 1919 (or any other century). Using the format “July 9, 2019”, as done in the latest version of the sample template, avoids this potential ambiguity.

Figure 2.16 shows the template for the example experiment after the column names have been revised to avoid any problematic characters. This template is now in a very useful format for a reproducible research pipeline—the data collected using this template can be very easily read into and processed using further statistical programs like R or Python.

	B	C	D	E	F	
1	sampling_date_time	aerated1	aerated3	low_oxygen1	low_oxygen2	low_oxygen3
2	"July 9, 2019 12:00"	0.020	0.020	0.020	0.020	0.020
3	"July 9, 2019 16:05"	0.027	0.020	0.022	0.022	0.027
4	"July 10, 2019 9:50"	0.087	0.087	0.057	0.082	0.086
5	"July 10, 2019 16:50"	0.129	0.137	0.072	0.104	0.111
6	"July 11, 2019 10:03"	0.247	0.258	0.102	0.114	0.126
7	"July 11, 2019 16:07"	0.299	0.302	0.110	0.110	0.126
8	"July 12, 2019 9:48"	0.410	0.428	0.137	0.111	0.131
9	"July 12, 2019 16:30"	0.426	0.432	0.138	0.110	0.130
10	"July 13, 2019 10:35"	0.442	0.444	0.138	0.100	0.125
11	"July 13, 2019 17:52"	0.448	0.450	0.143	0.103	0.131
12	"July 14, 2019 13:10"	0.436	0.448	0.144	0.100	0.131
13	"July 15, 2019 10:27"	0.434	0.45	0.147	0.097	0.129
14	"July 15, 2019 16:42"	0.428	0.45	0.147	0.097	0.129
15	"July 16, 2019 9:28"	0.42	0.448	0.145	0.095	0.128

Figure 2.16: Example of a simpler format that can be used to record and analyze data for the same laboratory experiment as the previous figure. Annotations highlight where data is entered by hand. No calculations are conducted or figures created—these are all done later, using a code script.

2.5.5 Moving further data analysis to later in the pipeline

Once you have created a “tidy” template for collecting your data in the laboratory, you can create a report template that will input that data and then provide summaries and visualizations. This allows you to separate the steps (and files) for collecting data from those for analyzing data. Figure 2.17 shows an example of a report template that could be created to pair with the data collection template shown in Figure 2.16.

To create a report template like this, you can use tools for reproducible reports from statistical programs like R and Python. In this section, we will give an overview of how you could create the report template shown in Figure 2.17.

This report is written using a framework called RMarkdown, which allows you to include executable code inside a nicely-formatted document, resulting in a document in Word, PDF, or HTML that is easy for humans to read while also generating results based on R code. We will cover this format in details in modules 3.7–3.9. In the rest of this section, we’ll walk through some of the R code that is “powering” the analysis in this document, but if you’d like to learn how to combine it into an RMarkdown document to create the report shown in Figure 2.17, you can learn much more about RMarkdown in those later modules. The code within the report uses the R language. We’ll cover a few examples here, but if you would like to use R, you’ll find it helpful to learn more. Numerous excellent (and free) resources exist to help learn R. One of the best is the book “R for Data Science” by Hadley Wickham and Garrett Grolemund. It is available in print, as well as free online at <https://r4ds.had.co.nz/>.

The code used to generate the results in Figure 2.17 is all in the programming language R. A programming language can seem, at first glance, much more difficult to learn and use than using a spreadsheet program like Excel to set up formulae and macros. However, languages like R have evolved substantially in recent years to allow for much more straightforward coding than you may have

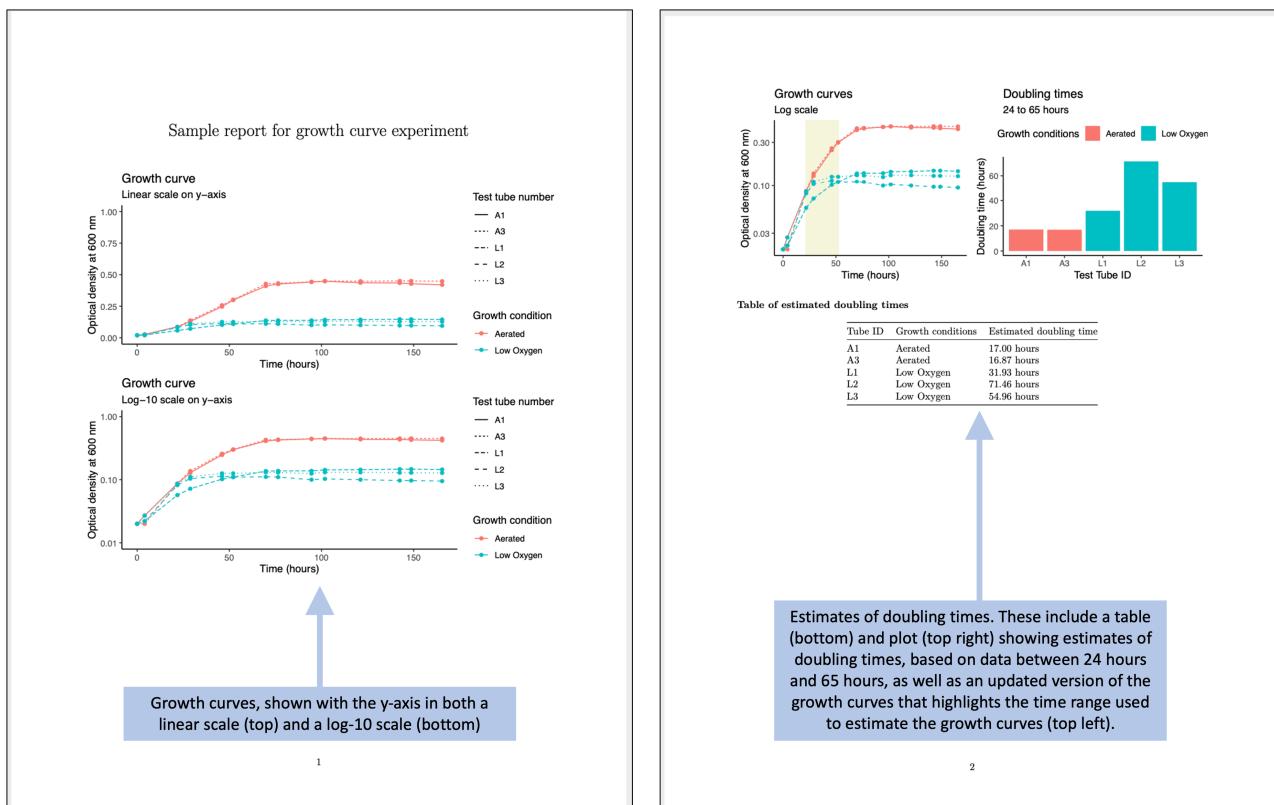


Figure 2.17: Examples of an automated report that can be created to quickly generate summaries and estimates of the data collected in the simplified data collection template for the example experiment.

seen in the past, and the barrier to learning to use them for straightforward data management and analysis is not much higher than the effort required to become proficient in using a spreadsheet program. To demonstrate this, let's look through a few of the tasks required to generate the results shown in Figure 2.17. We won't cover all the code, just highlight some of the key steps. If you'd like to look in details over the code and the output document, you can download those files and explore them: you can access the file for the Rmarkdown file, and you can download the output PDF. If you'd like to try out the code in the Rmarkdown file, you'll also need the example data, which you can download by clicking [here](#).

One key step is to read the collected data into R. When you use a spreadsheet for both data collection and analysis, you don't need to read the data to start working with them, since everything is saved in the same file. Once you separate the steps of data collection and data analysis, however, you do need to take an extra step to read the data file into another program for analysis. Fortunately, this is very simple in R. The data in this example are recorded using an Excel spreadsheet, and there is a simple function in R that lets you read data in from this type of spreadsheet (Figure 2.18). After this step of code, you will have an object in R called `growth_data`, which contains the data in a two-dimensional form very similar to how it is recorded in the spreadsheet (this type of object in R is called a `dataframe`).



The screenshot shows an RStudio interface. At the top, there is a code editor window containing the following R code:

```

20 * ````{r}
21 # Read in data from the Excel spreadsheet template used to collect the
22 # data in the laboratory
23 growth_data <- read_excel("growth_curve_data_GR.xls",
24                         sheet = "simplified_template")
25 ```

```

To the right of the code editor, a callout box with a blue border contains the text: "Code to read data from the data collection template (Excel file) into R."

Below the code editor is a data viewer window titled "A tibble: 14 × 6". It displays a table with the following data:

sampling_date_time	aerated1	aerated3	low_oxygen1	low_oxygen2	low_oxygen3
2019-07-09 12:00:00	0.020	0.020	0.020	0.020	0.020
2019-07-09 16:05:00	0.027	0.020	0.022	0.022	0.027
2019-07-10 09:50:00	0.087	0.087	0.057	0.082	0.086
2019-07-10 16:50:00	0.129	0.137	0.072	0.104	0.111
2019-07-11 10:03:00	0.247	0.258	0.102	0.114	0.126
2019-07-11 16:07:00	0.299	0.302	0.110	0.110	0.126
2019-07-12 09:48:00	0.410	0.428	0.137	0.111	0.131
2019-07-12 16:30:00	0.426	0.432	0.138	0.110	0.130
2019-07-13 10:35:00	0.442	0.444	0.138	0.100	0.125
2019-07-13 17:52:00	0.448	0.450	0.143	0.103	0.131

To the right of the data viewer, a callout box with a blue border contains the text: "Resulting dataset in R."

Another key step is to calculate, for each observation, the time since the start of the experiment. In the original data collection template shown in Figure 2.11, this calculation was done by hand by the researcher and entered into the spreadsheet. When we converted the spreadsheet to a tidier version, we took out all steps that involved calculations with the data, and instead limited the data collection to only raw, observed values. This helps us avoid errors

Figure 2.18: Code to read data from the data collection template into R for cleaning, analysis, and visualization. The data were recorded in the tidy data collection template described earlier in this module. Here, those data are read into R (code shown at top). The resulting data in R are stored in a format that is very similar to the design of a spreadsheet, with rows for observations and columns for the values recorded for each observation (bottom).

and typos—instead of having the researcher calculate the difference in time as they are running the experiment, they can just record the time, and we can write code in the analysis document that handles the further calculations, using well-designed and well-tested tools to do this calculation.

Figure 2.19 shows code that can be used for this calculation. At the start of this code, the data are stored in an object named `growth_data`. The `mutate` function adds a column to the data, named `sampling_delta_time`, that will give the difference between the time of an observation and the start of the experiment. Within the `mutate` call, a special function named `difftime` calculates the difference in two time points. This function lets us specify the time units we'd like to use, and here we can pick "hours" for the units. The `first` function lets us pull out the first value in the data for a recorded time—in other words, the time when the experiment started. This lets us compare each observation time to the time of the start of the experiment. The result of this code is a new version of the `growth_data` dataframe, with a new column giving time since the start of the experiment:

The screenshot shows an RStudio interface with a code editor and a data preview. The code editor contains the following R code:

```

27 + `}`{r}
28 # For each measurement time, calculate the time since the start of the
29 # experiment
30 growth_data <- growth_data %>%
31   mutate(sampling_delta_time = difftime(sampling_date_time,
32                                         first(sampling_date_time),
33                                         units = "hours"))
34 +

```

A callout box with a blue arrow points from the text "New column, with time since the start of the experiment." to the `sampling_delta_time` column in the data preview. The data preview shows a tibble with 14 rows and 7 columns. The columns are labeled `aerated1`, `aerated3`, `low_oxygen1`, `low_oxygen2`, `low_oxygen3`, and `sampling_delta_time`. The `sampling_delta_time` column contains values such as 0.000000 hours, 4.083333 hours, 21.833333 hours, etc.

	<code>aerated1</code>	<code>aerated3</code>	<code>low_oxygen1</code>	<code>low_oxygen2</code>	<code>low_oxygen3</code>	<code>sampling_delta_time</code>
1	0.020	0.020	0.020	0.020	0.020	0.000000 hours
2	0.027	0.020	0.022	0.022	0.027	4.083333 hours
3	0.087	0.087	0.057	0.082	0.086	21.833333 hours
4	0.129	0.137	0.072	0.104	0.111	28.833333 hours
5	0.247	0.258	0.102	0.114	0.126	46.050000 hours
6	0.299	0.302	0.110	0.110	0.126	52.116667 hours
7	0.410	0.428	0.137	0.111	0.131	69.800000 hours
8	0.426	0.432	0.138	0.110	0.130	76.500000 hours
9	0.442	0.444	0.138	0.100	0.125	94.583333 hours
10	0.448	0.450	0.143	0.103	0.131	101.866667 hours

1–10 of 14 rows | 2–7 of 7 columns Previous 1 2 Next

Another key step is to plot results from the data. In R, there is a package called `ggplot2` that provides tools for visualization. The tools in this package work by building a plot using “layers”, adding on small elements line by line through simple functions that each do one simple thing. While the resulting code can be long, each step is simple, and so it becomes simple to learn these different “layers” and learn how to combine them to create complex plots.

Figure 2.20 walks through the code for one of the visualizations in the report. At this point in the report code, the data have been reformatted into an object called `growth_data_tidy`, which has columns for each observation

Figure 2.19: Code to add a column to the data that gives the time since the start of the experiment. This code (top) uses the time recorded for each experiment and compares it to the first recorded time, at the start of the experiment. This determines the time since the start of the experiment for each observation, given in a new column in the data (bottom).

on the time since the start of the experiment (`sampling_delta_time`), the measured optical density (`optical_density`), whether the tube was aerated or low oxygen (`growth_conditions`), and a short ID for the test tube (`short_tube_id`). The code starts by creating a plot object, specifying that in this plot the color will show the growth conditions, the position on the x-axis will show the time since the start of the experiment, and the y-axis will show the optical density. Layers are then added to this plot object that add points and lines to the plot based on these mappings, and for the lines, it's further specified that the type of line should show the test tube ID (for example, one tube will be shown with a dotted line, another with a dashed line). Further layers are added to customize the scale labels with `labs`, including the labels for the x-axis and y-axis and the legends of the color and linetype scales. Another layer is used to customize the appearance of the plot—things like the background color and the font used—and another layer is added to use a log-10 scale for the x-axis.

While this looks like a lot of code, the process isn't any longer than it would be to customize elements of a plot in a spreadsheet program. The advantages of the coded approach are that you maintain a full record of all the steps you took to customize the plot. This is something that you can use to reproduce your plot later, or even to use as a starting point for creating a similar plot with new data.

The next key step that we'd like to point out is how you can write and use small functions to do customized tasks for the experimental data. As one example, for the data in this example, we want to estimate doubling times based on the observed data. The principal investigator has decided that we should do this based on comparing bacteria levels at two times points—the measured time that is closest to 65 hours after the start of the experiment, and the time that is closest to 24 hours after the start of the experiment.

In the original data collection template—where the data were both recorded and analyzed in a spreadsheet—this step was done by hand by the researcher, looking through the data and selecting the cell closest to each of these times, and then connecting that cell to a spreadsheet formula calculation to calculate the doubling time. We can make this process more rigorous and less prone to error by writing a small function that does the same thing, then using that function to automate the process of identifying the relevant observations to use in calculating the doubling rate.

Figure ?? shows how you can write and then use a small function in R. This function will input your `growth_data` dataset, as well as a time that you are aiming for, and will output the sampling time in the data that is closest to—while not larger than—that time. It does that in a few steps within the body of the function. First, the code in the function filters to only observations earlier than the target time. Then it measures the difference between each of the times for these observations and the target time, and uses this to identify the observation with the closest time to the target. It pulls out the time of this

sampling_date_time <S3: POSIXct>	sampling_delta_time <time>	short_tube_id <chr>	growth_condition <chr>	optical_density <dbl>
2019-07-09 12:00:00	0.000000 hours	A1	Aerated	0.020
2019-07-09 12:00:00	0.000000 hours	A3	Aerated	0.020
2019-07-09 12:00:00	0.000000 hours	L1	Low Oxygen	0.020
2019-07-09 12:00:00	0.000000 hours	L2	Low Oxygen	0.020
2019-07-09 12:00:00	0.000000 hours	L3	Low Oxygen	0.020
2019-07-09 16:05:00	4.083333 hours	A1	Aerated	0.027
2019-07-09 16:05:00	4.083333 hours	A3	Aerated	0.020
2019-07-09 16:05:00	4.083333 hours	L1	Low Oxygen	0.022
2019-07-09 16:05:00	4.083333 hours	L2	Low Oxygen	0.022
2019-07-09 16:05:00	4.083333 hours	L3	Low Oxygen	0.027

Format of the data (saved in the object `growth_data_tidy` when the plotting code is run.

```

87 ggplot(growth_data_tidy,
88         # Plot time since start on the x-axis, optical density on the y-axis,
89         # and use color to show the growth condition (aerated versus low oxygen)
90         aes(x = sampling_delta_time,
91              y = optical_density,
92              color = growth_condition)) +
93         # Add a line with different patterns for each test tube
94         geom_line(aes(linetype = short_tube_id)) +
95         # Add points for each measurement
96         geom_point() +
97         # Customize the labels for the legends and x- and y-axes
98         labs(x = "Time (hours)",
99              y = "Optical density at 600 nm",
100             color = "Growth condition",
101             linetype = "Test tube number") +
102         # Customize the plot appearance
103         theme_classic() +
104         # Add a title and subtitle
105         ggtitle("Growth curve", subtitle = "Log-10 scale on y-axis") +
106         # Use a log scale for the y-axis. Ensure that the y-axis ranges
107         # from 0.01 to 1 (you can't start at 0, since that's undefined
108         # as a log transform)
109         scale_y_log10(limits = c(0.01, 1))

```

Code to plot the growth curves for the experiment.

Create the plot object and specify mapping (y-axis shows optical density, etc.)

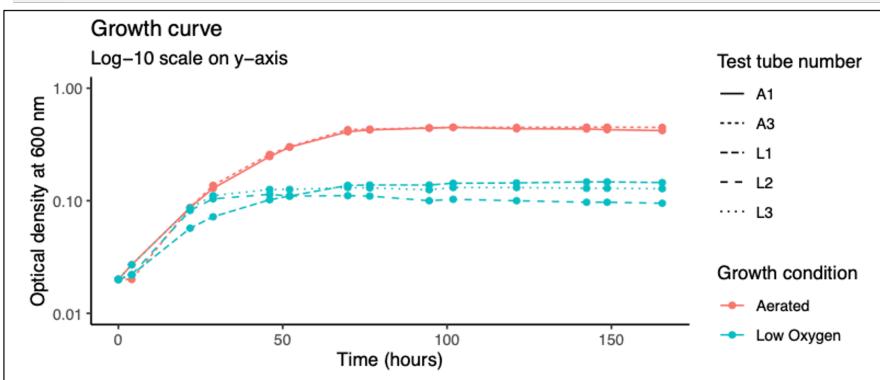
Add elements like points and lines to show the data

Customize the axis and legend labels

Customize the overall plot appearance

Add a title and subtitle

Use a log-10 scale on the x-axis



Resulting growth curve plot.

Figure 2.20: Code to plot growth curves from the data. When the plotting code is run, the data have been transformed into a 'tidy' format (top), with columns that include the time since the start of the experiment, a test tube ID, the growth condition for the test tube, and the optical density measured in that test tube. The code (middle) add layers to implement each element of the plot based on this input data. The final plot is shown at the bottom.

observation and returns it.

```

116 # Write a short function that can find the measurement time that is closest to,
117 # but does not exceed, a specified time. For example, if you want to measure
118 # growth rate between 24 hours and 48 hours, you can use this function to find
119 # the measurement times closest to (while still being below) 24 and 48 hours.
120 find_closest_time <- function(data, check_time) {
121   data %>%
122     filter(sampling_delta_time < check_time) %>%
123     mutate(diff_from_check = check_time - sampling_delta_time) %>%
124     slice_min(n = 1, order_by = diff_from_check) %>%
125     pull(sampling_delta_time)
126 }
```

Create a function that inputs your data and finds the observation time closest to a target time.

Filter to times before the target time

Calculate the difference for each observation from the target time

Extract the observation time with the smallest difference from the target time

```

> closest_to_24 <- find_closest_time(growth_data, 24)
> closest_to_24
Time difference of 21.83333 hours
> closest_to_65 <- find_closest_time(growth_data, 65)
> closest_to_65
Time difference of 52.11667 hours

```

Apply this function to find the observation points in the recorded data that are closest to, without being larger than, 24 hours and 65 hours.

Small functions like this can easily be reused in other code for your research group. By writing the logic of the step out as a function—rather than redoing the steps by hand or step-by-step each time you need to do it—you can save time later, and in return, you have extra time that you can spend in writing the original function and carefully checking to make sure that it works correctly.

Finally, many of these steps require extensions to base R. When you download R, you are getting a base set of tools. Many people have developed helpful extensions that build on this base. These are stored and shared in what are called **R packages**. You can install these extra packages for free, and you use the library function in R to load a package you've installed, giving you access to the extra functions that it provides. Figure 2.22 shows the spot in the Rmarkdown code where we loaded packages we needed for this report. These include packages with functions to read data in R from Excel (the `readxl`) package, as well as a suite of packages with tools for cleaning and visualizing data (the `tidyverse` package). In later modules, we'll talk some more about R coding tools that you might find useful for working with biomedical data, including the tools in the powerful and popular `tidyverse` suite of packages.

```

11 ## Load packages with additional functionality beyond base R
12 library(readxl)
13 library(tidyverse)
14 library(knitr)
15 library(purrr)
16 library(hms)
17 library(gridExtra)
```

Overall, you can see that the code in this document provides a step-by-step

Figure 2.21: Code to create and apply a small function. The code at the top can be used to create a function that can input your dataframe and determine the observation time in that data that is closest to (without being larger than) a target time. The function does this through a series of small steps. This function can then be applied to find the observation time in the data that is closest to specific target times, like 24 hours and 65 hours (bottom).

Load packages that provide extra functionality beyond what is provided by base R. Before you run this code, you will need to install each of the packages.

Figure 2.22: Code to load packages with additional functionality. These provide functions that are not offered in base R, but that are useful in working with the example data. They include packages with functions for reading in data from an Excel file, as well as packages with functions for cleaning and visualizing data.

recipe that documents all the calculations and cleaning that we do with the data, as well as how we create the plots. This code runs every time we create the report shown in Figure 2.17, and it gives us a good starting point if we run additional experiments that generate similar data.

2.5.6 Applied exercise

The Rmarkdown document includes a number of other steps, and you might find it interesting to download the document and the example data and walk through them to get a feel for the process. All the steps are documented in the Rmarkdown document with extensive code comments, to explain what's happening along the way.

2.6 Power of using a single structured ‘Project’ directory for storing and tracking research project files

To improve the computational reproducibility of a research project, researchers can use a single ‘Project’ directory to collectively store all research data, meta-data, pre-processing code, and research products (e.g., paper drafts, figures). We will explain how this practice improves the reproducibility and list some of the common components and subdirectories to include in the structure of a ‘Project’ directory, including subdirectories for raw and pre-processed experimental data.

Objectives. After this module, the trainee will be able to:

- Describe a ‘Project’ directory, including common components and subdirectories
- List how a single ‘Project’ directory improves reproducibility

In previous modules, we've discussed how you can separate the steps of data collection from steps of data cleaning, management, pre-processing, and analysis. In the case of data recorded in a laboratory, this separation will often result in moving from a single file (for example, a spreadsheet), that combines all the steps to using separate files for the different steps. As you move to separate data recording and analysis, then, it is very important to store the set of files for a project in a way that is clear and easy to manage. If the file directory for the project is well-designed, it can even allow you to create software tools and report templates that can be reused over many experiments. In the next few modules, we'll discuss how you can organize the files for an experiment using R's “Project” system. The modules will discuss the advantages of well-designed project directories, tips for arranging files within a project directory, and how to create templates for R Projects that allow you to use consistent file organization across many experiments.

As a motivating example, for the next few modules, we'll use an example based on a set of real immunology experiments. This example highlights how

a research laboratory will often conduct a similar type of experiment many times, so it lets us demonstrate how the design of the project's files within a project directory can be reused across similar experiments. It will allow us to show you how you can move from designing a file directory for a single experiment to designing one that can be used repeatedly, and then how you can take advantage of consistency in the directory structure across projects to make tools and templates that can be reused.

2.6.1 Example project

"To say that scientists erred in assuming that the first-line drugs from the 1950s would be sufficient to combat TB is a profound understatement." (Barry and Cheung, 2009)

"The current treatment course, which was developed in the 1960s, is a demanding regimen consisting of four first-line drugs created in the 1950s and 1960s: isoniazid, ethambutol, pyrazinamide, and rifampin. Patients who follow the regimen as directed take an average of 130 doses of the drugs, ideally under direct observation by a health care worker. This combination is extremely effective against active, drug-susceptible TB as long as patients are compliant and complete the entire six- to nine-month course. Drug-resistant strains develop when patients do not complete the full protocol, whether because they start to feel better or because their drug supply is interrupted for some reason. Inconsistent use of antibiotics gives the bacteria time to evolve into a drug-resistant form. Once a drug-resistant strain has developed in one person, that individual can spread the resistant version to others. ... According to the World Health Organization [as of 2009], nearly 5 percent of the roughly eight million new TB cases that occur each year involve strains of Mtb that are resistant to the two most commonly used drugs in the current first-line regimen: isoniazid and rifampin. Most cases of this so-called multidrug-resistant TB (MDR-TB) are treatable, but they require therapy for up to two years with second-line anti-TB drugs that produce severe side effects. Moreover, MDR-TB treatment can cost up to 1,400 times more than regular treatment. ... Worst of all, over the past few years health surveys have revealed an even more ominous threat, that of extensively drug-resistant TB (XDR-TB). This type ... is resistant to virtually all the highly-effective drugs used in second-line therapy." (Barry and Cheung, 2009)

The examples for this and the next few modules are based on a collection of studies that were conducted with similar designs and similar goals—all aimed to test candidate treatments for tuberculosis. Most studies in this set tested one or more treatments as well as one or more controls. The controls could include negative controls, like saline solution, or positive controls, like a drug already in use to treat the disease, isoniazid. A few of the studies tested only controls, to help in developing baseline expectations for things like the bacterial load in different mouse strains used in studies in the set. The set of studies tested some treatments that were monotherapies (only one drug given to the animal) as well as some that were combinations of two or three different drugs. For many of the drugs that were tested, they were tested at different doses and, in some cases, different methods of delivery or different mouse models.

Each of the treatments were given to several mice that had been infected with *Mycobacterium tuberculosis*. During the treatment, the mice were weighed regularly. This weight measurement helps to determine if a particular treatment is well-tolerated by the animals—if not, it may show through the treated mice losing weight during treatment. For convenience, the mice were not weighed individually. Instead, mice with the same treatment were kept in a single cage, and the entire cage was weighed, the weight of the cage itself factored out, and the average weight of mice for that treatment determined by dividing the weight of all mice in the cage by the number of mice in the cage. After a period of time, the mice were sacrificed and one lobe from their lungs was used to determine each mouse's bacterial load, through plating the material from the lobe and counting the colony forming units (CFUs). One aim of the data analysis is to compare the bacterial load of mice under various treatments to the bacterial load of mice in the control group.

The full set of studies included 19 different studies. These were conducted at different times, but the data for all of the studies can be collected using a common format. In this module, as well as the following two, we'll be exploring how you can use RStudio's Project functionality to organize data from one or more studies. We'll particularly focus on how, by using a common format for data collection, you can create tools that can be used repeatedly for different experiments to ensure that methods are the same across all studies of a similar type, as well as to improve the reproducibility of the studies. Let's walk through the types of data that were collected for each study.

First, there was some metadata recorded for each study. Figure 2.23 gives an example. This includes information about the strain of mouse that was used in the study, treatment details (including the method of giving the drug or drugs, how often they were given each week, and for how many weeks), how much bacteria the animals were exposed to (measured both in terms of the inoculum they were given and their bacterial load one day after they were given that inoculum, which was based on sacrificing one animal the day after challenging all the animals with the bacteria), and, if the study included a novel drug as part of the tested treatment, the batch number of that drug.

Next, the researchers recorded some information about each treatment group within the experiment. This typically included at least one negative control. In some cases, there was also a positive control, in which the animals were treated with a drug that's in standard use against tuberculosis already (e.g., isoniazid). Most studies would also test one or more treatments, which could include monotherapies or combined therapies. Figure 2.24 shows an example of the data that were recorded on each treatment in the study. These data include the names and doses of up to three drugs in each treatment, as well as a column where the researcher can provide detailed specifications of the treatment.

Once the animals were challenged with the bacteria, treatment began, and two main types of data were measured and recorded. First, the mice were

	A	B	C	D	E	F	G	H
1	study	mouse_strain	route	rx_per_week	weeks_of_rx	inoculum	day_1_count	novel_drug_batch_number
2	PIO22-1	Balb/c	intrapulmonary aerosol	3	4	3.55	2.15	COMP-001-TR21

Information about the drug treatment: how was it administered, how often per week, and for how many weeks?

Information about the animals' initial exposure to Mycobacterium tuberculosis, including the inoculum given during the challenge and a measure of bacterial load one day later.

	A	B	C	D	E			
1	rx_group	group	drug_1_name	drug_2_name	drug_3_name	drug_1_dose	drug_2_dose	drug_3_dose
2	0	negative control						full_drug_dose_details
3	1	positive control	isoniazid					Untreated
4	2	monotherapy	novel drug A			10		Isoniazid, 25mg/kg by intrapulmonary aerosol
5	3	combination	pyrazinamide	novel drug A		150	10	Novel drug A, 10mg/kg by intrapulmonary aerosol
								Pyrazinamide, 150mg/kg in 200 ul by gavage + novel drug A, 10 mg/kg t

Figure 2.23: Example of recording metadata for a study in the set of example studies for

Full specifications of the treatment

weighed once a week. These weights were converted to a measure of the percent change in weight since the start of treatment. If the animals' weights decrease during the treatment, it is a marker that the treatment is not well-tolerated by the animals. Figure 2.25 shows an example of how these data were recorded. All animals within a treatment group were kept in the same cage, and this cage was measured once a week. By dividing the weight of all animals in the cage by the number of animals, the researchers could estimate the average weight of animals in that treatment group, which is recorded as shown in Figure ??.

Finally, after the treatment period, the mice were sacrificed and a portion of each mouse's lung was used to estimate the bacterial load in that mouse. Figure 2.26 shows an example of how the data on the bacterial load in each mouse was recorded.

As you can see, these data were all recorded using templates that were designed for the tidy collection of laboratory data (see modules 2.4 and 2.5). These spreadsheets were used only to record the data, and then processing, analysis, and visualization were done in a separate file. Specifically, for this set of studies a preliminary report was designed, with an example shown in Figure [x]. This report uses the first page to provide a nicely formatted version of the metadata for the study, including a table with overall details and a table with details for each specific treatment that was tested. The second page provides a

Figure 2.24: Example of recording treatment details for a study in the set of example studies for this module.

Identifier for the study

Which mouse strain was used?

Information about the animals' initial exposure to *Mycobacterium tuberculosis*, including the inoculum given during the challenge and a measure of bacterial load one day later.

	A	B	C	D	E	F	G	H
1	study	mouse_strain	route	rx_per_week	weeks_of_rx	inoculum	day_1_count	novel_drug_batch_number
2	PI022-1	Balb/c	intrapulmonary aerosol	3	4	3.55	2.15	COMP-001-TR21
3								

Information about the drug treatment: how was it administered, how often per week, and for how many weeks?

If a new drug was tested, what was the batch number from the manufacturer?

Figure 2.25: Example of recording weekly weights of mice in each treatment group for the example set of studies.

Treatment group

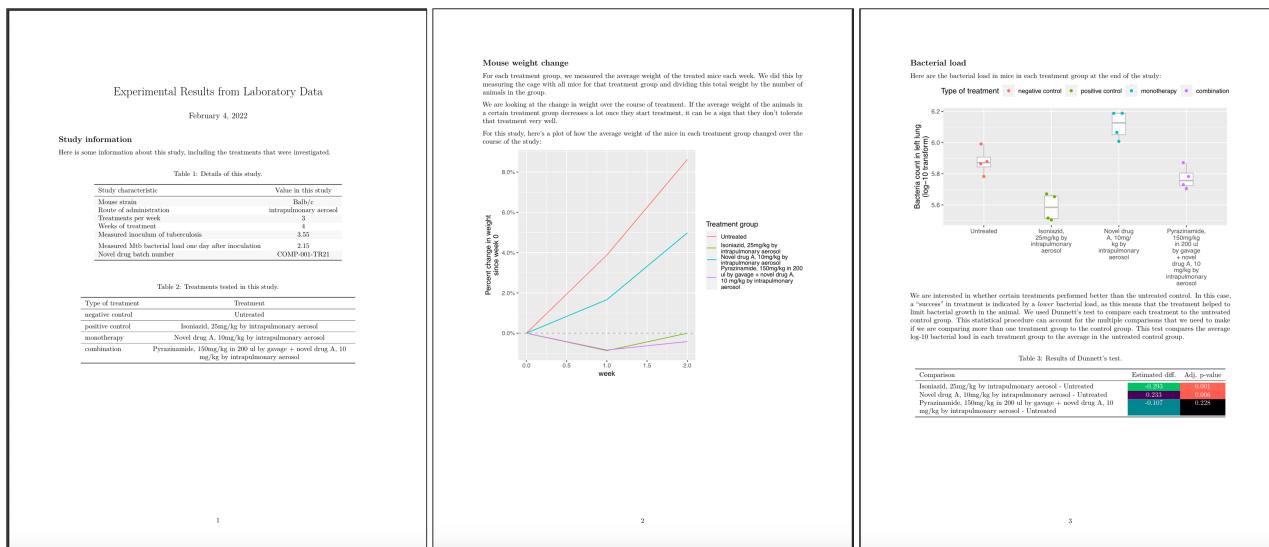
Animal in group

Bacterial load in the mouse's lung at end of treatment

	A	B	C
1	rx_group	animal_in_rx_group	bacteria_count
2	0	1	731250
3	0	2	981250
4	0	3	606250
5	0	4	756250
6	1	1	450000
7	1	2	318750
8	1	3	328125
9	1	4	468750
10	2	1	1543750
11	2	2	1162500
12	2	3	1018750
13	2	4	1543750
14	3	1	606250
15	3	2	506250
16	3	3	743750
17	3	4	537500
18			

Figure 2.26: Example of recording the bacterial load in the lungs of each mouse at the end of treatment for the example set of studies.

graph that shows the percent weight change for mice in each treatment group compared to the weight of that group at the start of treatment. The third page provides a graph that shows the bacterial loads in each mouse, grouped by treatment, as well as the results of running a statistical test to test, for each treatment group, the hypothesis that the mean of a transformed version of the measure of bacterial load (\log_{10}) for the group was the same as for the untreated control group.



Let's take a closer look at a few of these elements. For example, Figure 2.28 shows the tables from the first page of the report shown in Figure 2.27. If you look back to the data collection for this study (e.g., Figures 2.23 and 2.24), you can see that all of the information in these tables was pulled from data recorded at the start of the study.

Figure 2.29 shows the second page of the report. This figure has taken the mouse weights—which were recorded in one of the data collection templates for the project (Figure ??)—and used them to generate a plot of how average mouse weight in each treatment group changed over the course of the treatment.

Figure 2.30 shows the last page of the report. This page starts with a figure that shows the bacterial load in the lungs of each mouse in the study at the end of the treatment period. In this figure, the measurement for each mouse is shown with a point, and these points are grouped by the treatment group of the mouse. Boxplots are added to show the distribution across the mice in each group. The color is used to show whether the treatment was a negative control, a positive control, a monotherapy, or a combined therapy. The second part of the page gives a table with the results from running a statistical analysis to compare the bacterial load for mice in each treatment group to the bacterial

Figure 2.27: Example of the preliminary report generated for each study in the set of example studies for this module. The first page includes metadata on the study, as well as details on each treatment that was tested. The second page shows how mouse weights in each treatment group changed over the course of treatment, to help identify if a treatment was well-tolerated. The third page graphs the bacterial load in each mouse, grouped by treatment, and gives the result of a statistical analysis to test which treatment groups had outcomes that were significantly different from the untreated control group.

Study information

Here is some information about this study, including the treatments that were investigated.

Table 1: Details of this study.

Study characteristic	Value in this study
Mouse strain	Balb/c
Route of administration	intrapulmonary aerosol
Treatments per week	3
Weeks of treatment	4
Measured inoculum of tuberculosis	3.55
Measured Mtb bacterial load one day after inoculation	2.15
Novel drug batch number	COMP-001-TR21

Table with details from the study's metadata, including the mouse strain used in the study and the route of administration of the drug or drugs in the treatment.

Table 2: Treatments tested in this study.

Type of treatment	Treatment
negative control	Untreated
positive control	Isoniazid, 25mg/kg by intrapulmonary aerosol
monotherapy	Novel drug A, 10mg/kg by intrapulmonary aerosol
combination	Pyrazinamide, 150mg/kg in 200 ul by gavage + novel drug A, 10 mg/kg by intrapulmonary aerosol

Table with details on each treatment considered in the study. A treatment could include one or more drugs, or it could be a positive or negative control.

load in the mice in the untreated control group. Color is added to the table to highlight treatments that had a large difference in bacterial load from the untreated control, as well as treatments for which the difference from the untreated control was estimated to be statistically significant. All the data for these results, including the labels for the plot, are from the data collected in the data collection templates shown earlier.

2.6.2 Creating an organized directory for project files

In the example project that we just described, you can see that we ended up with a lot of different files. We used separate files for collecting data and for creating a preliminary report on that data. For collecting the data, we could either use separate sheets in one spreadsheet file, or entirely separate files, or some combination. The end result is that we have several project files, in comparison to if we had used a single sheet of a spreadsheet file to both record and work with the data.

This is a step in the right direction—by separating data collection and analysis into separate files, we can make the file for each step simpler. Further, by separating, we can save the files in plain text, which makes it easier to track them using version controlled (discussed in later modules), which helps create a record of changes made to the data or analysis code during the research

Figure 2.28: Example of one element of the preliminary report generated for each study in the set of example studies for this module. The first page provides tables with metadata about the study and details about each treatment that was tested.

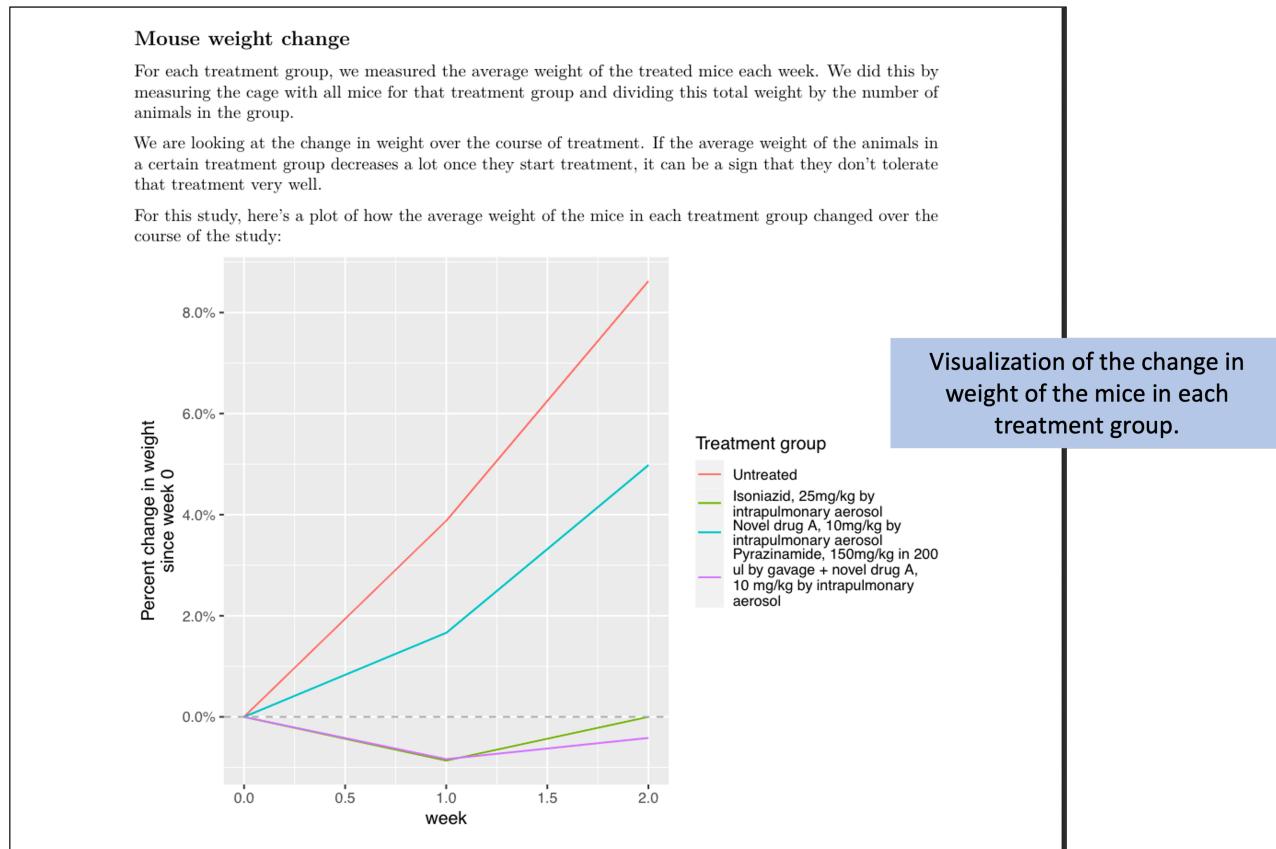
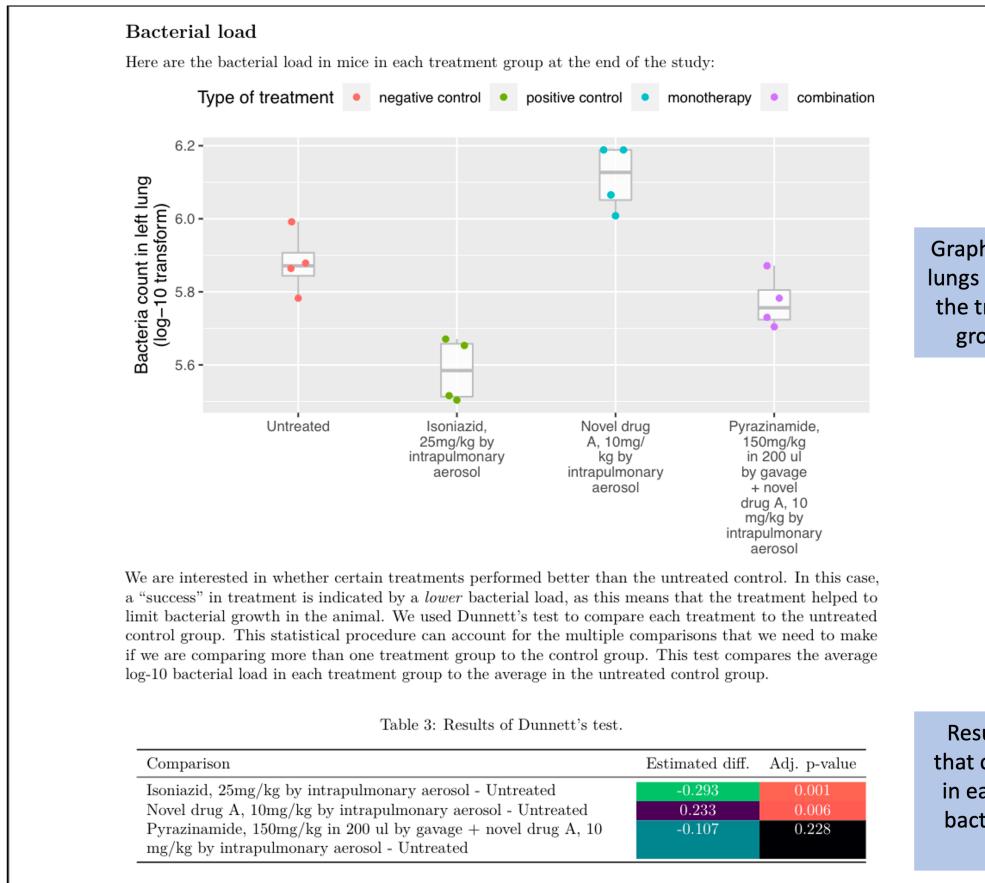


Figure 2.29: Example of one element of the preliminary report generated for each study in the set of example studies for this module. The second page provides a plot of how the weights of mice in each treatment changed over the course of treatment.



Graph of the bacterial load in the lungs of each mouse at the end of the treatment period, with mice grouped by their treatment.

Results of a statistical analysis that compares the bacterial load in each treatment group to the bacterial load in the untreated control group.

Figure 2.30: Example of one element of the preliminary report generated for each study in the set of example studies for this module. The third page provides results on how bacterial load in the lungs compares among treatments at the end of the treatment period.

process. As we continue work on a project, we'll likely generate more types of files. These can include other data files (both "raw" data—directly output from measurement equipment or directly recorded from observations, as well as any "cleaned" version of this data, after steps have been taken to preprocess the data to prepare it for visualization and analysis in papers and reports). These files also include the files with writing and presentations (posters and slides) associated with the project, as well as code scripts for preprocessing data, for conducting data analysis, and for creating and sharing final figures and tables.

However, now that we have multiple files, we should think about how we store and organize them. It is very easy otherwise for separate files for a research project to become disorganized. One study, for example, surveyed over 250 biomedical researchers at the University of Washington. They noted that, "a common theme surrounding data management and analysis was that may researchers preferred to utilize their own individual methods to organize data. The varied ways of managing data were accepted as functional for most present needs. Some researchers admitted to having no organizational methodology at all, while others used whatever method best suited their individual needs." (Anderson et al., 2007) One respondent answered, "They're not organized in any way—they're just thrown into files under different projects," while another said "I grab them when I need them, they're not organized in any decent way," and another, "It's not even organized—a file on a central computer of protocols that we use, common lab protocols but those are just individual Word files within a folder so it's not searchable per se." (Anderson et al., 2007)

First, and at a minimum, your research team should get in the habit of storing all the files for a project (for example, for an experiment or a study) in the same file directory. There are a number of advantages to keeping all files related to a single project inside a dedicated file directory on your computer. First, this provides a clear and obvious place to search for all project files throughout your work on the project, including after lulls in activity (for example, while waiting for reviews from a paper submission). By keeping all project files within a single directory, you also make it easier to share the collection of files for the project. There are several reasons you might want to share these files. An obvious one is that you likely will want to share the project files across members in your research team, so they can collaborate together on the project. However, there are also other reasons you'd need to share files, and one that is growing in popularity is that you may be asked to share files (data, code scripts, etc.) when you publish a paper describing your results.

When files are all stored in one directory, the directory can be compressed and shared as an email attachment or through a file sharing platform like Google Drive. As you learn more tools for reproducibility, you can also share the directory through some more dynamic platforms, that let all those sharing access continue to change and contribute to the files in the directory in a way that is tracked and reversible. In later modules in this book, we will introduce git version control software and the GitHub platform for sharing files under

"Just as a well-organized laboratory makes a scientist's life easier, a well-organized and well-documented project makes a bioinformatician's life easier. Regardless of the particular project you're working on, your project directory should be laid out in a consistent and understandable fashion. Clear project organization makes it easier for both you and collaborators to figure out exactly where and what everything is. Additionally, it's much easier to automate tasks when files are organized and clearly named. For example, processing 300 gene sequences stored in separate FASTA files with a script is trivial if these files are organized in a single directory and are consistently named."

[@buffalo2015bioinformatics]

"All files and directories used in your project should live in a single project directory with a clear name. During the course of a project, you'll have amassed data files, notes, scripts, and so on—if these were scattered all over your hard drive (or worse, across many computers' hard drives), it would be a nightmare to keep track of everything. Even worse, such a disordered project would later make your research nearly impossible to reproduce."

[@buffalo2015bioinformatics]

this type of version control—this is one example of this more dynamic way of sharing files within a directory. To gain the advantages of directory-based project file organization, all the files need to be within a single directory, but they don't all have to be within the same “level” in that directory. Instead, you can use subdirectories to structure and organize these files, while still retaining all the advantages of directory-based file organization. This will help limit the number of files in each “level” of the directory, so none becomes an overwhelming slew of files of different types. It can help you navigate the files in the directory, and also help someone you share the directory with figure out what's in it and where everything is.

Subdirectory organizations can also, it turns out, be used in clever ways within code scripts applied to files in the directory. For example, there are functions in all scripting languages that will list all the files in a specified subdirectory. If you keep all your raw data files of a certain type (for example, all output from running flow cytometry for the project) within a single subdirectory, you can use this type of function with code scripts to list all the files in that directory and then apply code that you've developed to preprocess or visualize the data across all those files. This code would continue to work as you added files to that directory, since it starts by looking in that subdirectory each time it runs and working with all files there as of that moment.

As you decide how to organize project files within a directory, it is worthwhile to take some time to think about the types of files that are often generated by your research projects, because there are also big advantages to creating a standard structure of subdirectories that you can use consistently across the directories for all the projects in your research program. Of course, some projects may not include certain files, and some might have a new or unusual type of file, so you can customize the directory structure to some degree for these types of cases, but it is still a big advantage to include as many common elements as possible across all your projects.

For example, you may want to always include a subdirectory called “data”, and consistently call it “data”, to store data directly from observations or directly output from laboratory equipment. You may want to include subdirectories in that “data” subdirectory for each type of data—maybe a “recorded_data” subdirectory for data that were recorded by a researcher in the laboratory, and another subdirectory called “flow_data” for output from a flow cytometer. By using the same structure and the same subdirectory names for different experiments with similar aims or designs, you will find that code scripts are easier to reuse from one project to another. Again, most scripting languages allow you to leverage this order in how you've arranged your files in the file system, and so using the same order across different projects lets you repeat and reuse code scripts more easily from one project to another. This strategy is used often in handling complex bioinformatics data (Buffalo, 2015), but it can also be leveraged to improve the reproducibility and reliability when only using less complex data recorded in the laboratory, as with the data shown in the

“Organizing data files into a single directory with consistent filenames prepares us to iterate over *all* of our data, whether it's the four example files used in this example, or 40,000 files in a real project. Think of it this way: remember when you discovered you could select many files with your mouse cursor? With this trick, you could move 60 files as easily as six files. You could also select certain file types (e.g., photos) and attach them all to an email with one movement. By using consistent file naming and directory organization, you can do the same programmatically using the Unix shell and other programming languages.”
[@buffalo2015bioinformatics]

example for this module.

Even within a subdirectory, it is worthwhile to give a lot of thought to how you name each file. First, special tools in languages like R can use thoughtfully-designed files name to extract and use information in the file names themselves, based on a technique called *regular expressions*. Second, the file name is a valuable way to be very clear about exactly what is stored in the file. The less murky the organization and naming of subdirectories and files within the project directory is, the more likely that all research team members will be able to correctly navigate and use the data stored for the project.

Finally, if you create a clear and clean organization structure for your project directories, you will find it is much easier to navigate your files in all directories, and also that new lab members and others you share the directories with will be able to quickly learn to navigate them. In other areas of science and engineering, this idea of standardized directory structures has allowed the development of powerful techniques for open-source software developers to work together. For example, anyone may create their own extensions to the R programming language and share these with others through GitHub or several large repositories. This is coordinated by enforcing a common directory structure on these extension “packages”—to create a new package, you must put certain types of files in certain subdirectories within a project directory. With these standardized rules of directory structure and content, each of these packages can interact with the base version of R, since there are functions that can tap into any of these new packages by assuming where each type of file will be within the package’s directory of files. In a similar way, if you impose a common directory structure across all the project directories in your research lab, your collaborators will quickly be able to learn where to find each element, even in projects they are new to, and you will all be able to write code that can be easily applied across all project directories, allowing you to improve reproducibility and comparability across all projects by assuring that you are conducting the same preprocessing and analysis across all projects (or, if you are conducting things differently for different projects, that you are deliberate and aware that you are doing so).

Figure [x] gives an example of a project directory organization that might make sense for the example set of studies described at the beginning of this module. ...

Once you have decided on a structure for your directory, you can create a template of it—a file directory with all the subdirectories included, but without any files (or only template files you’d want to use as a starting point in each project). When you start a new project, you can then just copy this template and rename it. If you are using R and begin to use R Project (described in the next section), you can also create an R Studio Project template to serve as this kind of starting point each time you start a new project. In module 2.8, we’ll walk through an example of creating and using this kind of project template for the example set of studies described earlier in the module.

“Because automating file processing tasks is an integral part of bioinformatics, organizing our projects to facilitate this is essential. Organizing data into subdirectories and using clear and consistent file naming schemes is imperative—both of these practices allow us to *programmatically* refer to files, the first step to automating a task. Doing something programmatically means doing it through code rather than manually, using a method that can effortlessly scale to multiple objects (e.g., files). Programmatically referring to multiple files is easier and safer than typing them all out (because it’s less error prone).”

[@buffalo2015bioinformatics]

“In addition to simplifying working with files, consistent naming is an often overlooked component of robust bioinformatics. Bad naming schemes can easily lead to switched samples. Poorly chosen filenames can also cause serious errors when you or collaborators think you’re working with the correct data, but it’s actually outdated or the wrong file. I guarantee that out of all the papers published in the past decade, at least a few and likely many more contain erroneous results because of a file naming issue.”

[@buffalo2015bioinformatics]

2.6.3 Using RStudio Projects with project file directories

If you are using the R programming language for data preprocessing, analysis, and visualization—as well as RMarkdown for writing reports and presentations—then you can use RStudio’s “Project” functionality to make it even more convenient to work with files within a research project’s directory. You can make any file directory a “Project” in RStudio by choosing “File” -> “New Project” in RStudio’s menu. This gives you the option to create a project from scratch or to make an existing directory an RStudio Project.

When you make a file directory an RStudio Project, it doesn’t change much in the directory itself except adding a “.RProj” file. This file keeps track of some things about the file directory for RStudio, including preferred settings for RStudio to use when working in that project. Also, when you open one of these Projects in RStudio, it will move your working directory into that project’s top-level directory. This makes it very easy and practical to write code using relative pathnames that start from this top-level of the project directory. This is very good practice, because these relative pathnames will work equally well on someone else’s computer, whereas if you use file pathnames that are absolute (i.e., giving directions to the file from the root directory on your computer), then when someone else tries to run the code on their own computer, it won’t work and they’ll need to change the filepaths in the code, since everyone’s computer has its files organized differently. For example, if you, on your personal computer, have the project directory stored in your “Documents” folder, while a colleague has stored the project directory in his or her “Desktop” directory, then the absolute filepaths for each file in the directory will be different for each of you. The relative pathnames, starting from the top level of the project directory, will be the same for both of you, though, regardless of where you each stored the project directory on your computer.

There are some other advantages, as well, to turning each of your research project directories into RStudio Projects. One is that it is very easy to connect each of these Projects with GitHub, which facilitates collaborative work on the project across multiple team members while tracking all changes under version control. This functionality is described in a later module in this book.

As you continue to use R and RStudio’s Project functionality, you may want to take the template directory for your project and create an RStudio Project template based on its structure. Once you do, when you start a new research project, you can create the full directory for your project’s files from within RStudio by going to “File” -> “New Project” and then choosing to create a new project based on that template. The new project will already be set up with the “.RProj” file that allows you to easily navigate into and out of that project, to connect it to GitHub, and all the other advantages of setting a file directory as an RStudio Project. This takes a bit of time to set-up, but can be a powerful tool in ensuring that researchers in your laboratory use a standardized format for project directories across many experiments.

The next module gives step-by-step directions for making a directory an RStudio Project, and also how to create your own RStudio Project template to quickly create a new directory for project files each time you start a new research project.

"We also needed to cope with a large amount of data going from each lab [involved in trying to reproduce results] to a single database. We wrote an iPad app so that measurements were entered directly into the system and not jotted on paper to be entered later. The app prompted us to include a full description for each plate of worms, and ensured that data and metadata for each experiment were proofread (the strain names MY16 and my16 are not the same). This simple technology removed small recording errors that could disproportionately affect statistical analyses." (Lithgow et al., 2017)

"Perhaps the most complex undertaking so far has been developing practices for curating and preserving all the data that underpin a paper, including replicates. This took about a year, working with senior faculty members, the information-technology team and another research manager. We trialled our data-archiving system with a couple of groups, implemented it across our institute for a year and then amended it on the basis of feedback. Instead of squirrelling away data in individual folders and lab books, researchers now archive all published data in a designated central drive, so that the information is accessible for the long haul. Initially, people thought the process was just extra bureaucratic work, or that it had been invented so I could police their data. Now, it has become the norm, and researchers tell me they save time and worry by having their data organized and archived." (Winchester, 2018)

"Any scientist adopting a QA system has to wager that the up-front hassle will pay off in the future. 'It is very difficult to get people to check and annotate everything, because they think it is nonsense,' says Carmen Navarro-Aragay, head of the University of Barcelona quality team that worked with Cirera. 'They realize the value only when they get results that they do not understand and find that the answer is lurking somewhere in their notebooks.' Even when experiments go as expected, quality systems can save time, says Murtaugh. Methods and data sections in papers practically write themselves, with no time wasted in frenzied hunting for missing information. There are fewer questions about how experiments were done and where data are stored, says Murtaugh. 'It allows us to concentrate on biological explanations for results.'" (Baker, 2016)

"Having data that are traceable — down to who did what experiment on which machine, and where the source data are stored — has knock-on benefits for research integrity, says Nett. 'You can't pick out the data that you want.' Researchers who must provide strong explanations about why they chose to leave any information out of their analysis will be less tempted to cherry-pick data. QA can also weed out digital meddling: popular spreadsheet programs such as Microsoft Excel can be vulnerable to errors or manipulation if not properly locked, but QA teams can set up instruments to store read-only files and prevent researchers from tampering with data accidentally or intentionally." (Baker, 2016)

"Data should be logged in a lab notebook, not scribbled onto memo paper or other detritus and carelessly transcribed. Notebooks should be bound or digital; loose paper can too easily be lost or removed." (Baker, 2016)

"Protocols should be followed to the letter or deviations documented." (Baker, 2016)

"Beyond giving my brain the space to more easily take in what I am working with, knolling my work space throughout the day also reduces the likelihood that I'll lose things, and increases the likelihood that I will find them quickly when they do go missing. ... By forcing me to slow down (which I've learned actually allows me to work faster), the whole process saves me time on the other end of my work process." (Savage, 2020)

"Heres how to [knoll:] 1. Examine your work space for all items not in use—tools, materials, books, coffee cups, it doesn't matter what it is. 2. Remove those unused items from your space. When in doubt, leave it on the table. 3. Group all like items—pens with pencils, washers with O-rings, nuts with bolts, etc. 4. Align (parallel) or square (90-degree angle) all objects within each group to each other and then to the surface upon which they sit." (Savage, 2020)

"There's one group of makers who understand intuitively, perhaps more than anyone else, the multifaceted value proposition that knolling represents: chefs. They call it *mise en place*. Coined in the late nineteenth century by famed French chef August Escoffier, *mise en place* translates roughly to 'everything in its place'. The principle behind it, cribbed from Escoffier's military service during the Franco-Prussian War, is really about order and discipline." (Savage, 2020)

"For all the alchemy that goes into building something, the magic of making is only possible because of the many repetitive processes we endure in preparation for final assembly, and then the deliberate way we put it all together. And the only way to get that prep right, to get your *mise* in its proper *place* is to slow down, address your work properly, and lock your shit down." (Savage, 2020)

"Kitchens are pressure cookers in which wasted movement and hasty technique can ruin a dish, slice an artery, burn a hand, land you in the weeds, and ultimately kill a restaurant. *Mise en place* is the only way to reliably create a perfect dish, to exact specifications, over and over again, night after night, for paying customers who demand nothing less." (Savage, 2020)

"A shop is not simply a place to make things. Yes, it's where we collect our materials, our tools, our notes, and our half-completed ideas, but it's also a manifestation of how we think about organization, project management, and working priorities." (Savage, 2020)

"It often seems like our first work spaces are an energized mess of chaos and creation that we will insist at the time we have our arms around, but with some distance—both physical and temporal—we will realize was retarding our creative output in one way or another." (Savage, 2020)

"Not all organizational methodologies are created equal. One could be spotlessly organized, with everything put away and labeled and color coded, and it could feel like a prison with the walls closing in around you. Another could be equally organized but a bit more open and exposed, and it could untap creative genius like no other space you've worked in." (Savage, 2020)

"What truly unifies my shops, especially as I got more experienced, is that they are each built on two, simple philosophical pillars: 1) I want to be able to see everything easily; and 2) I want to be able to reach everything easily." (Savage, 2020)

"I started to clean up before heading upstairs at the end of the day, and lo and behold the shop became a far more efficient and well-oiled machine to work in. The freed-up space in my mind and the open work space at my fingertips allowed me a lot of room, both mental and physical, to pursue a wide variety of projects, and I finally started to understand how much benefit was to be gained by taking the time to clean." (Savage, 2020)

"Luck favors the prepared mind." —Louis Pasteur

"With a science like molecular biology, we inevitably have an image in our heads of the scientist alone in the lab, hunched over a microscope, and stumbling across a major new finding. But Dunbar's study showed that those isolated eureka moments were rarities. Instead, most important ideas emerged during regular lab meetings, where a dozen or so researchers would gather and informally present and discuss their latest work. If you looked at the map of idea formation that Dunbar created, the ground zero of innovation was not the microscope. It was the conference table." (Johnson, 2011)

"Even with all the advanced technology of a leading molecular biology lab, the most productive tool for generating good ideas remains a circle of humans at a table, talking shop." (Johnson, 2011)

"You get a feeling that there's an interesting avenue to explore, a problem that might someday lead you to a solution, but then you get distracted by more pressing matters and the hunch disappears. So part of the secret of hunch cultivation is simple: write everything down." (Johnson, 2011)

"We can track the evolution of Darwin's ideas with such precision because he adhered to a rigorous practice of maintaining notebooks where he quoted other sources, improvised new ideas, interrogated and dismissed false leads, drew diagrams, and generally let his mind roam on the page. We can see Darwin's ideas evolve because on some basic level the notebook platform creates a cultivating space for his hunches; it is not that the notebook is a mere transcription of the ideas, which are happening offstage somewhere in Darwin's mind. Darwin was constantly rereading his notes, discovering new implications. His ideas emerge as a kind of duet between the present-tense thinking brain and all those past observations recorded on paper." (Johnson, 2011)

2.7 Creating 'Project' templates

Researchers can use RStudio's 'Projects' feature to facilitate collecting research files in a single, structured directory, with the added benefit of easy use of version control. Researchers can gain even more benefits by consistently structuring all their 'Project' directories. We will demonstrate how to implement structured project directories through RStudio, as well as how RStudio enables the creation of a 'Project' for initializing consistently-structured directories for all of a research group's projects.

Objectives. After this module, the trainee will be able to:

- Be able to create a structured Project directory within RStudio
- Understand how RStudio can be used to create 'Project' templates

The last module describe the advantages of organizing all the files for a research project within a single file directory, as well as the added advantages of making that file directory an RStudio “Project”. In this module, we’ll walk through the steps required to do that, as well as how to navigate and use the “Project” structure and functionality to make it easier to integrate project files, code, and final output like reports and presentation slides. If you have created a standardized file directory structure that you will use as a starting point for all of your research projects, then it can save time to create an RStudio Project Template with this structure. This way, you can set up a new directory, including all the subdirectories you want to use and any code or report templates that might be useful, by just opening a new project with this template through RStudio.

...

To start, let’s look at how we can organize the files from this set of studies into a directory in an efficient way. ...

...

The full directory of files for this example can be found at [GitHub address], where you can download them or explore them online. All files for this project can be stored within a well-designed directory, and this directory can be enhanced into something called an R Project very easily. In this module, we’ll explore how to use an R Project and what advantages it offers compared to other ways of organizing the files associated with a study. In particular, we’ll build on ideas from earlier modules about creating reproducible data collection templates, as in this example, the use of a common template across many studies in a set makes it very easy to create and apply a common reporting template to the data, easily creating a reproducible report for each of the nineteen studies in the example set of studies. Further, we’ll look at how this organization allows not only for reporting on specific studies in a reproducible way, but also makes it easier to create an overall report that combines results and details from all studies in the set.

...

2.7.1 *Making an existing file directory an RStudio Project*

It’s very easy to turn an existing file directory into an RStudio Project. Open RStudio, and then in its menu go to “File” and then “New Project”. This will open a pop-up window with several options, including the option to create a new RStudio Project from an existing directory. Choose this option, and then in the window that is opened, navigate through your file directories to the directory with your project files.

This will create a new RStudio Project with the same name as the file directory name of the directory you selected [check this]. Once you have created the directory, RStudio will automatically move you into that Project. When you close RStudio and reopen it, it will automatically open in the last Project you

had open. There is a small tab in the top right hand corner of the RStudio window that lists the project you are currently in. To move to a different Project, you can click on the down arrow beside this project name. There will be a list of your most recent projects, as well as options to open any Project on your computer. If you want to work in RStudio, but not in any of the Projects, you can choose to “Close Project”.

When you are working in an RStudio Project, RStudio will automatically move your working directory to be the top-level directory of the Project directory. This makes it easy to write code that uses this directory as the presumed working directory, using relative file paths to identify and files within the directory. For example . . . Now if you share the project directory with someone else, they can similarly open the RStudio Project in their own version of RStudio, and all the relative pathnames to files should work on their system without any problems. This feature helps make code in an RStudio Project directory reproducible across different people’s computers.

The RStudio Project environment has some other features, as well, that may be useful for some projects. For example, if you are tracking the project directory under the git version control system, then when you open the RStudio Project, there will be a special tab in one of the panes to help in using git with the project. This tab provides a visual interface for you to commit changes you’ve made, so they are tracked and can be reversed if needed, and also so you can easily push and pull these committed changes to and from a remote repository, like a GitHub repository, if you are collaborating with others.

For certain types of projects, you may also want to include a “Makefile”. [More about Makefiles]. If you add this to an RStudio Project, you will get a new “Build” tab that allows you to run the Makefile for the project with the click of a button. For some more complex RStudio Projects, like projects that use RMarkdown to create online books or websites using bookdown and blogdown, this “Build” pane will allow you to render the whole book.

2.7.2 *Making an RStudio Project Template*

If you have created a template directory for research projects for your group, you can create an RStudio Project template to make it easy to set up a new project directory every time you start a project. At its most basic, this can be a directory that includes the subdirectories (with standardized names for each subdirectory) that you want to include—for example, you may know that you will always want the project directory to include subdirectories for “raw_data” (with its own subdirectories for different types of data, for example for “cfus” and “flow”), “data” (with clean versions of the data, after conducting and needed preprocessing, like calculating CFUs in a sample based on data from plating at different dilutions, or the output from gating flow cytometry data), “reports” (for writing, posters, and presentation slides), and “R” (for common scripts that you use for preprocessing, visualization, and data analysis).

As you progress, you may also want to add templates that serve as a starting point for files within this project. For example, if you always want to collect observed data in a standard way, you could create a template for data collection, for example as a CSV file. Each researcher in your lab could copy and rename this file each time they collect a new set of data—by ensuring a common structure when collecting the data, including file format, column names, and so on, you can build code scripts that will work on data collected for all your experiments. You may also have some standard reports that you want to create with types of data you commonly collect, and so you could include templates for those reports in your R Project template. Again, these can be copied and adapted within the project—the template serves as a starting point so you don't have to start with a blank slate with every project, but it is not restrictive and can be adapted to each project as you work on that project.

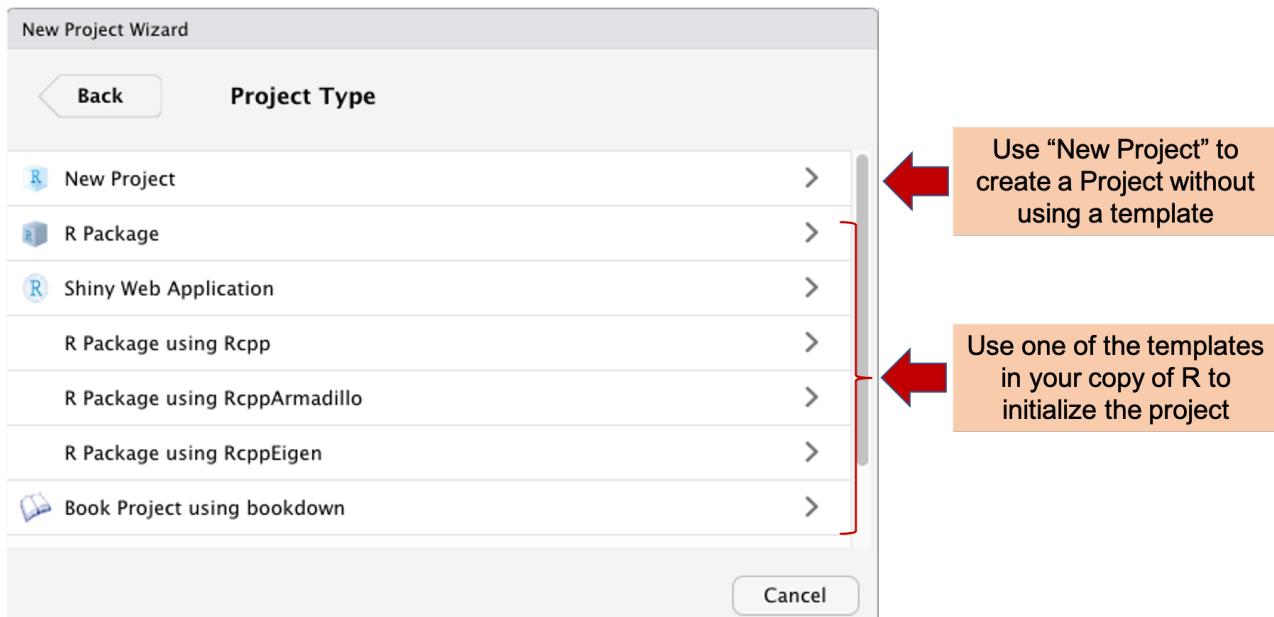
[How to create a template]

You can have different templates to use when you create a new project. A template will provide a basic set-up for the project. For example, R packages are created within a directory, and package developers now often use an R Project for that directory. Since R packages have a consistent set of usual subdirectories—including subdirectories for R code, data, and documentation and a file for metadata—there is now a Project Template specifically for R package development, and this sets up a directory with some of the typical subdirectories and files needed for these projects.

You may find, similarly, that there are typical structures and files that your research laboratory includes in the R Projects it creates for research projects. In this case, someone in your lab can create an R Project template that can be used each time a new project is initialized for the lab. This will help create consistency across projects in the directory structure, which can facilitate the use and re-use of automated tools like code scripts across different experiments.

When you create a new project in R, you will have the option to use any of the available project templates currently downloaded to your copy of R (rst, 2021). To create a new project, go to the “File” menu in the top menu bar in RStudio, and then choose “New Project”. This will open a pop-up box like the one shown in Figure 2.31.

This pop-up contains the New Project Wizard in RStudio. Here, you can either create a new Project without using a template (click on “New Project”) or you can create a Project starting from a template. The templates available in your copy of R will be listed below the “New Project” listing. Depending on which packages you've installed for your copy of R, you will have different choices of project templates available, as project templates tend to be created and shared within R packages (rst, 2021). In the example shown in Figure 2.31, for example, one of the template options is for a “Book Project using bookdown”, available because the bookdown R package has been installed locally.



2.7.3 Setting up a useful template

...

Once you set up this template, a researcher in your group can initialize a project for a new experiment by copying the template directory and renaming it to the name of the experiment. They can then open the directory and replace any placeholder data in the project files with real data from the experiment.

Figure 2.33 gives an example of this process. One of the files that is included in the example template directory shown earlier is a spreadsheet to record metadata on the experiment. This spreadsheet file has two sheets, one that records overall metadata on the study (for example, the weeks of treatment given and the strain of mouse used) and one that records details on each of the treatments that was tested. In the file in the template directory, these spreadsheet pages include placeholder data. These are formatted in red, so that they visually can be identified as placeholders. By including these placeholder data, the researcher can see an example of the format that you expect to be used in recording data in this file. Once the project template is copied, the researcher will replace these data with the real data, and then change the font color to black to indicate that the placeholder data have been replaced (Figure 2.33).

Another sheet of this spreadsheet allows the researcher to record the details of each of the treatments that were tested in the experiment. Again, placeholder data are included in the template in a red font to help show the researcher how to record the data, and these are meant to be replaced with

Figure 2.31: Creating a new project in RStudio. When you chose 'File' then 'New Project' from the RStudio menu, it opens the New Project Wizard shown here. You have the option to create a new project that is not based on a project template by selecting 'New Project'. You also have the chance to create a project using a template by selecting one of the templates. The listed templates will depend on which packages you have downloaded for your copy of R. For example, here the 'bookdown' package has been installed for the local copy of R, and so a template is available for 'Book Project using bookdown'.

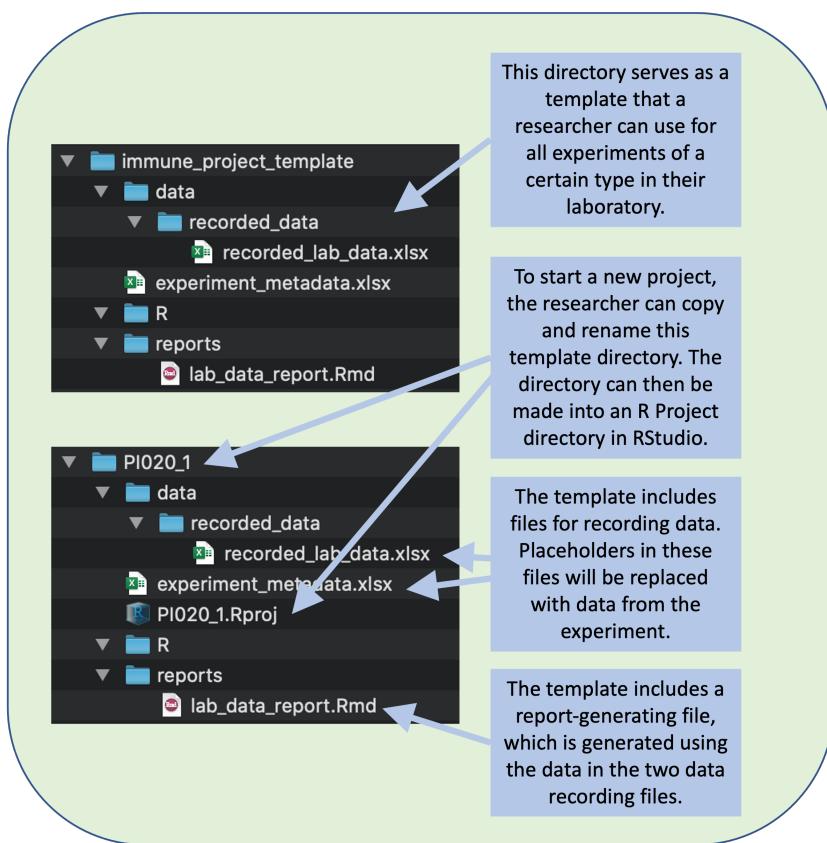


Figure 2.32: A research group can create a file directory that will serve as a template for all the experiments of a certain type in your laboratory. The template can include templates of files for data recording and for generating reports. To start recording data for a new experiment, a researcher can copy and rename this template directory.

The figure consists of three vertically stacked screenshots of Microsoft Excel. The top screenshot shows the 'Files' tab of RStudio with a folder structure. The 'experiment_metadata.xlsx' file is selected, and a callout box points to it with the text: 'Open the "experimental_metadata" file, to the sheet named "study_key".'. The middle screenshot shows the 'experimental_metadata.xlsx' file open in Excel. The 'study_key' sheet contains the following data:

	A	B	C	D	E	F	G	H
1	study	mouse_strain	route	rx_per_week	weeks_of_rx	inoculum	day_1_count	novel_drug_batch_number
2	PI022-1	Balb/c	intrapulmonary aerosol	3	4	3.55	2.15	COMP-001-TR21

The values in columns D, E, F, G, and H are displayed in red. The bottom screenshot shows the same file after the placeholder data has been replaced with real study data. The replaced values are now displayed in black. A callout box points to the second row with the text: 'Replace the placeholder data (in red) with the real data for the study. Change the color to black to show that the placeholder data have been replaced.'

Figure 2.33: The template includes a file with experiment metadata, with a sheet for recording the overall details of the experiment. A user can open this file and replace the placeholder values (in red) with real values for the experiment. By changing the text color to black, the user can have a visual confirmation that the placeholder data have been replaced with real study data.

real data from the specific experiment (Figure 2.34). A similar format is used in the template file to record data from the experiment, including the weights of each animal over each week of treatment and the final bacterial load in each animal at the end of treatment. Again, there are placeholder values in the template file, which the researcher will replace with real data after copying the project template for a new experiment.

The figure consists of three screenshots of Microsoft Excel. The top screenshot shows the 'Files' tab in RStudio with a folder structure: Home > Documents > my_books > improve_repro_set_of_studies > PI020. The middle screenshot shows the 'experimental_metadata.xlsx' file open in Excel. It has a table with columns: rx_group, group, drug_1_name, drug_2_name, drug_3_name, drug_1_dose, drug_2_dose, drug_3_dose, and full_drug_dose_details. Red placeholder text is present in the drug names and doses for rows 2 through 5. The bottom screenshot shows the same file after the placeholder data has been replaced with real study data, indicated by black text in the same cells.

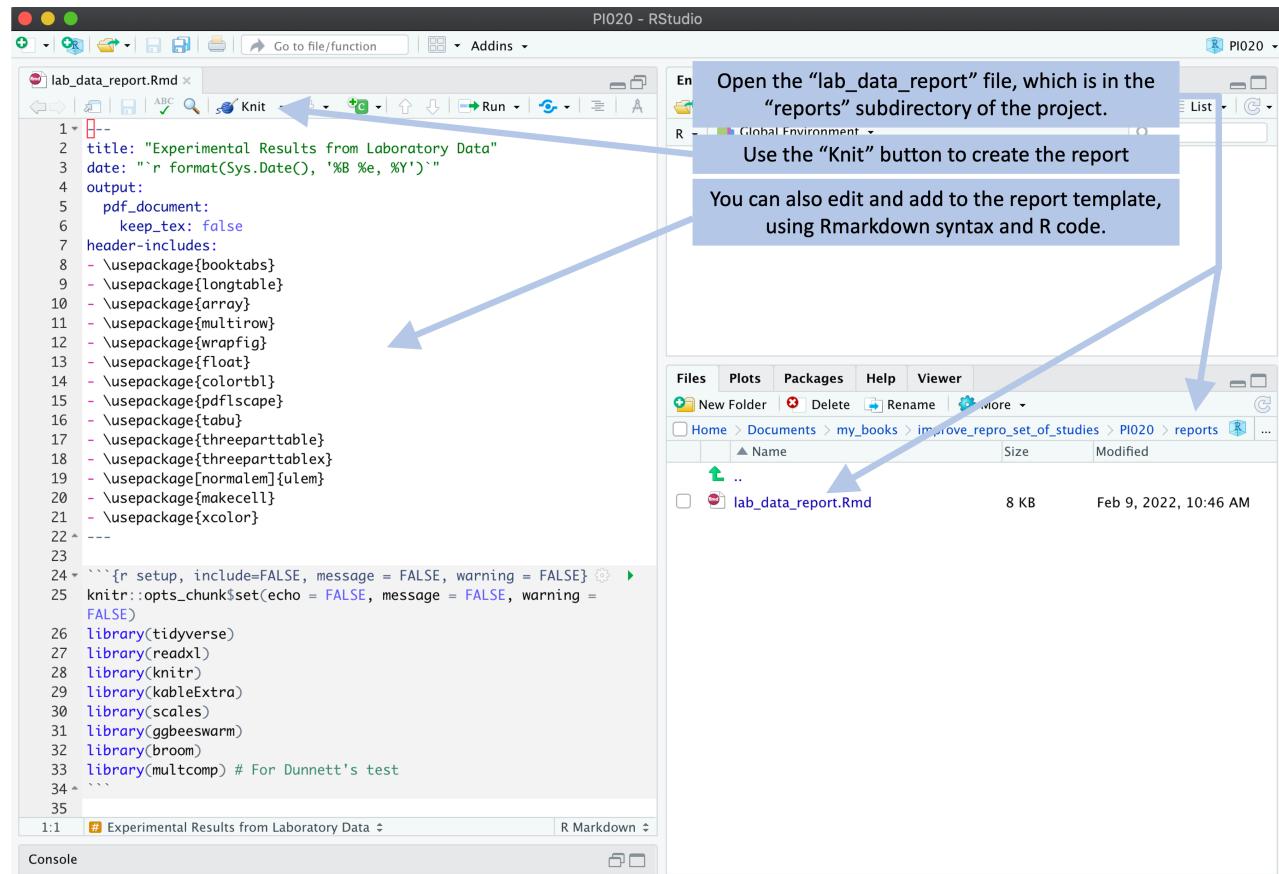
	rx_group	group	drug_1_name	drug_2_name	drug_3_name	drug_1_dose	drug_2_dose	drug_3_dose	full_drug_dose_details
1	0	control							Untreated
2	1	monotherapy	isoniazid						Isoniazid, 25mg/kg by intrapulmonary aerosol
3	2	monotherapy	novel drug A			10			Novel drug A, 10mg/kg by intrapulmonary aerosol
4	3	combination	pyrazinamide	novel drug A		150	10		Pyrazinamide, 150mg/kg in 200 ul by gavage + novel drug A, 10 mg/kg

	rx_group	group	drug_1_name	drug_2_name	drug_3_name	drug_1_dose	drug_2_dose	drug_3_dose	full_drug_dose_details
1	0	negative control							Untreated
2	1	negative control	isoflurane						Isoflurane (anesthetic)
3	2	negative control	saline						0.9% saline
4	4	monotherapy	novel drug A			10			Novel drug A, 10mg/kg by intrapulmonary aerosol
5	5	monotherapy	novel drug A			25			Novel drug A, 25mg/kg by intrapulmonary aerosol
6	6	monotherapy	novel drug A			50			Novel drug A, 50mg/kg by intrapulmonary aerosol

The project template also includes a file that provides a template to create a report based on the data from the experiment. This file is created using the RMarkdown format, which combines text with executable code. You can create this template so that it inputs the experimental data from the file formats created for the data recording files in the project template. By doing this, the researcher should be able to “knit” this report for a new experiment, and it

Figure 2.34: The template includes a file with experiment metadata, with a sheet for recording the details of each treatment. A user can open this file and replace the placeholder values (in red) with real values for the treatments in the experiment. By changing the text color to black, the user can have a visual confirmation that the placeholder data have been replaced with real study data.

should recreate the report based on the data recorded for that experiment (Figure 2.35). By knitting this template report, you can create a nicely formatted version of the report for the experimental data (Figure 2.36).



2.7.4 Creating 'Project' templates in RStudio

Your research group can create your own Project templates. This will allow you to use a standard template for your projects, just like we showed in the last section. However, instead of needing to copy, paste, and rename the template each time, if you create an official RStudio Project template, then the researcher can chose to use this template under the "New Project" option in RStudio (Figure 2.37).

To create your own Project template that can be used in this way, you will need to create them within an R package, but this package does not need to be posted to a public site like CRAN. Instead, it can be shared exclusively among the research group as a zipped file that can be installed directly from source onto each person's computer. Alternatively, you can post the package code as a GitHub repository, and there are straightforward tools for installing R package

Figure 2.35: Example of how a user can create a report from the template. The template includes an example report, which is written using RMarkdown. The user can open this template report file and use the 'Knit' button in RStudio to render the file. As long as the experimental data are recorded using the data template files, the code for this report can process the data to generate a report from the data. The user can also make changes and additions to the template report.

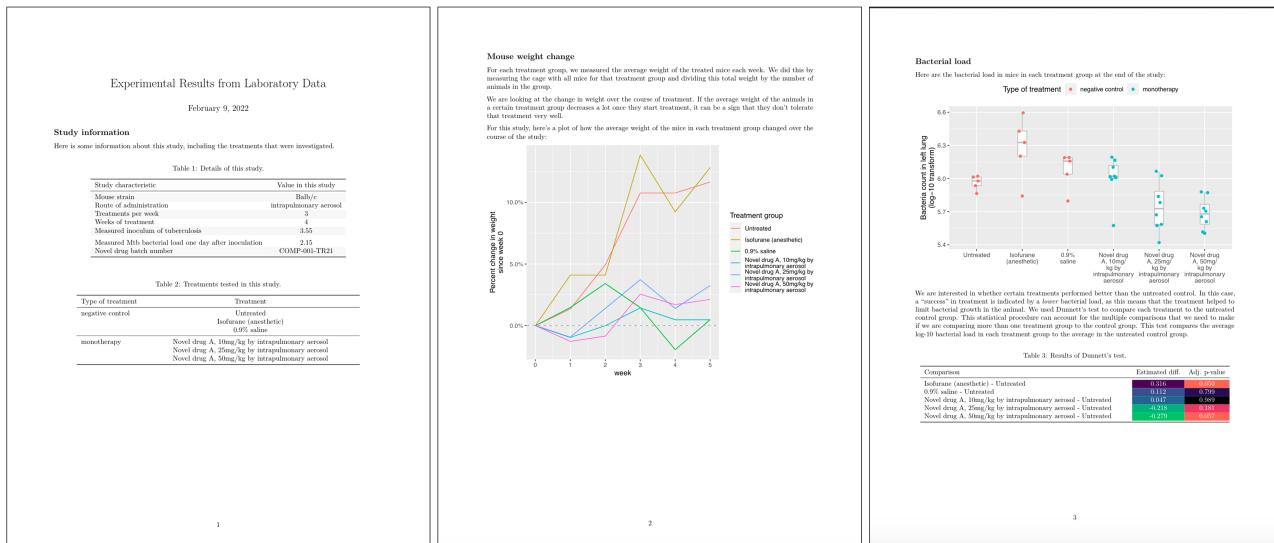


Figure 2.36: Example of the output from 'knitting' a report from the project template

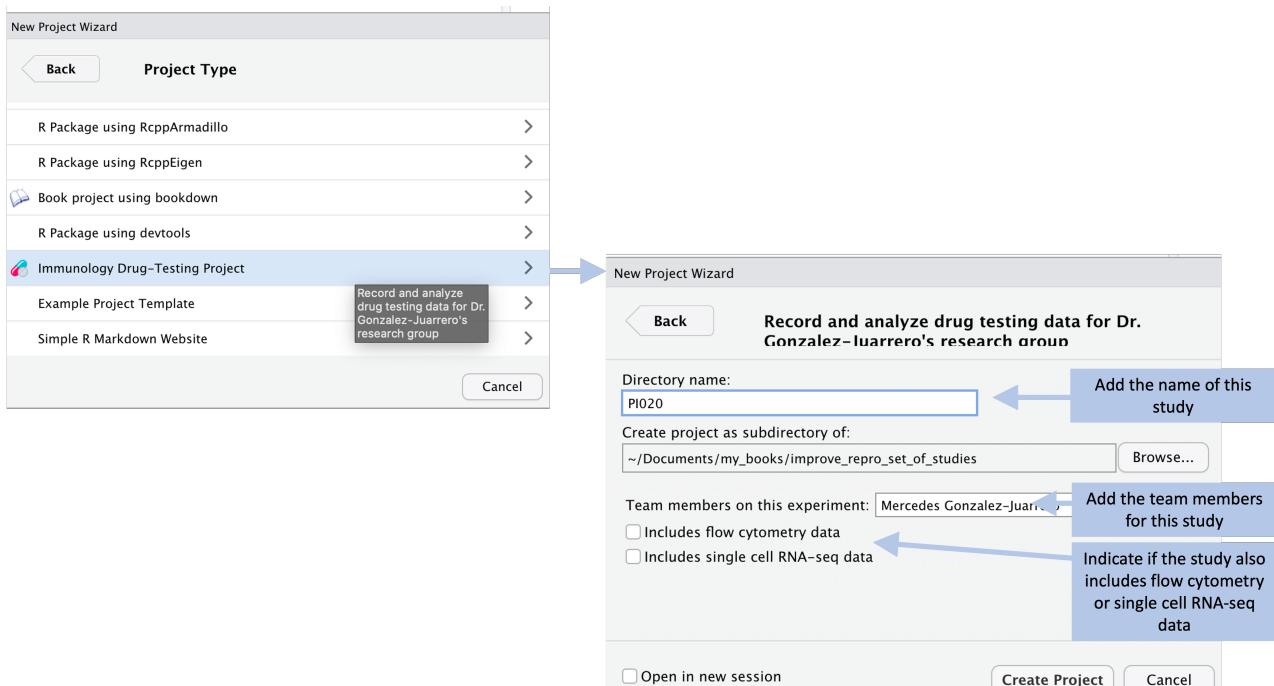


Figure 2.37: To make it easier for members of a group to use a project template, the group can create an official R template for the type of project. Once this type of template is created, a user can access it as a choice when creating a new R Project from RStudio. When doing so, a box will pop up with options for setting up the project. In this example, the user can specify the members of the research team and indicate if the experiment will include data from flow cytometry or single cell RNA-

code from GitHub onto each team member's computer. RStudio has provided a detailed guide to creating your own project template at https://rstudio.github.io/rstudio-extensions/rstudio_project_templates.html. This topic has also been discussed through a short talk at the yearly RStudio::conf: <https://rstudio.com/resources/rstudioconf-2020/rproject-templates-to-automate-and-standardize-your-workflow/>.

"RStudio v1.1 introduces support for custom, user-defined project templates. Project templates can be used to create new projects with a pre-specified structure." (rst, 2021)

"R packages are the primary vehicle through which RStudio project templates are distributed. Package authors can provide a small bit of metadata describing the template functions available in their package—RStudio will discover these project templates on start up, and make them available in the New Project... dialog." (rst, 2021)

"R experts keep all the files associated with a project together—input data, R scripts, analytical results, figures. This is such a wise and common practice that RStudio has built-in support for this via **projects**." (Wickham and Grolemund, 2016)

2.7.5 Discussion questions

2.8 Example: Creating a 'Project' template

We will walk through a real example, based on the experiences of one of our Co-Is, of establishing the format for a research group's 'Project' template, creating that template using RStudio, and initializing a new research project directory using the created template. This example will be from a laboratory-based research group that studies the efficacy of tuberculosis drugs in a murine model.

Objectives. After this module, the trainee will be able to:

- Create a 'Project' template in RStudio to initialize consistently-formatted 'Project' directories
- Initialize a new 'Project' directory using this template

For this module, we'll show how to create an R Project template to manage data from the example set of studies that we described in module 2.7. As a reminder, this example set of studies covers a group of studies to explore novel treatments for tuberculosis. Each study investigates how mice that are challenged with tuberculosis respond to different treatments, both in terms of how well they handle the treatment (assessed by checking if their weight decreases notably while on treatment) and also how well the treatment manages to limit the growth of tuberculosis in the mouse's lungs.

We will walk through the process of creating a project directory template that could be used to manage and analyze data from any of the specific studies

in this set of studies. We'll cover two ways that you could do this. The first is simpler—it involves creating a basic file directory with the desired template files and file directory structure and then copying this file directory every time you want to start a new project for a study in this set of studies. The second way is a bit more complex and time-consuming to set up, but has the benefit of providing a very nice interface for members of your laboratory group to use when they start a new project. This second way is to create a full R Project template that can be accessed from RStudio anytime you create a new R Project. This type of template may not be worth the extra set-up time for project types that your research group only uses rarely, but for types of projects that your group conducts time and time again, it can be a powerful way to enforce a common project directory structure, and this in turn will allow your group to create reusable tools that work in coordination with this project structure.

2.8.1 *Creating and using a basic template*

Let's start by looking at the more basic way to create a project template. This involves no fancy tools—in fact, it's so straightforward that at first it might seem to simple to be useful. For this basic approach, you will create an example file directory that includes template files and that captures your desired project directory structure, and then members of your group will copy and rename that template every time they start a new project of that type.

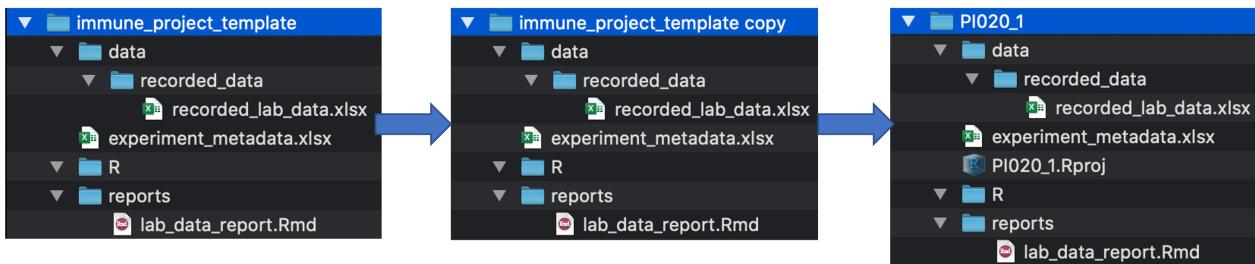
Figure 2.38 gives a basic walk-through of the simple steps you'll use to start a new project directory once you've created this type of template. First, you will find the project directory template in your computer's file system, copy it to where you'd like to save the files for the new project, and rename the directory to your new project's name. At this point, you can use RStudio to make this directory an RStudio Project. Next, you'll open the data collection template files and replace the placeholder example data in the template (shown in red font) with the real data from your study. The placeholder data can help you remember the format you should use to record the real data. Finally, once you've recorded the data for the study or experiment, you can open the example report template file. If you've designed this report template well, it should run with the new data you've recorded to create a report for the experiment. At this stage, you can add to the report or customize it for the new project by changing the Rmarkdown file and re-rendering it to update the report.

There are a few steps you'll need to take to create this type of basic project directory template:

1. List the data you typically collect or files you create for that type of study or experiment
2. Create template files for any data collection that is typical for that type of study or experiment. Use example or placeholder data to create examples

1

Find the project directory template in the file finder program on your computer. Copy the entire directory, paste the copy where you want to store the project directory for your new study, and rename the directory to the name of your new study.

**2**

Open data recording templates and replace the placeholder data (saved in red font to indicate that it's placeholder data) with data from the real project. Change the font color to black to show that these are data from the project, rather than placeholder data.

**3**

Open the project report template. Render it to PDF to create the report. If you'd like, you can make changes to the template Rmarkdown report file to customize it for this project.

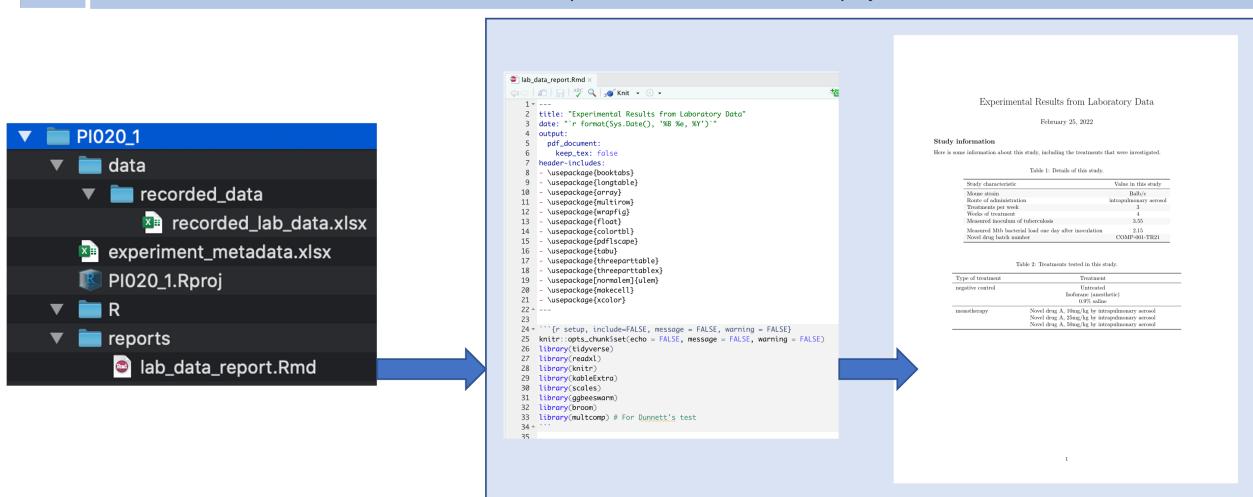


Figure 2.38: Steps in using a basic project directory template that you have created for a type of study or experiment.

of those files.

3. Create a directory structure that divides the types of files into subdirectories of similar types.
4. Create one or more templates of report files that access and report on the data in the project template

In modules [x], we showed how you can create tidy data collection templates to use to collect data, and how these can be paired with reproducible reporting tools to separate the steps of data collection and reporting (modules [x] go into much more depth on these reproducible reporting tools). Once you have decided on the types of data that you will usually collect for the type of study that this template is for, you can use that process to create tidy data collection templates for each type of data.

In addition to the data that you record in the laboratory by hand, the type of study may also typically have data that's generated and recorded by laboratory equipment. For example, the type of study may often include data collected from flow cytometry, to measure certain cell populations in samples, or from mass spectrometry, to measure levels of certain molecules. For these data, the recording format will typically be determined by the equipment, and so you won't need to create data collection templates for the data. However, you should store these data files in your project directory as well, where they are easy to access and integrate with other data as you analyze the data for the study.

The recorded data files and the files that come directly from equipment can all be considered raw data files. In addition, you may typically create some files with pre-processed data. For example, if you have sequencing data [?], you may initially get large [what type] files from the [what type] equipment. You may use a program like [what] to pre-process these files to [do what]. In addition to saving the raw [what type] data files, you'll also want to save the processed data files in your project directory, since these are the files that you'll analyze and integrate with other data from the project.

Once you have determined the types of files that you'll normally include in your project, you can decide how to organize them into subdirectories in a project file directory. As you do this, it will be helpful to have example or template files for each file type. For data that you will record yourself, these can be the templates that you developed to collect the data in a tidy format (modules [x]), while for data from equipment, these can just be one or more example files from the equipment that you have collected for a past project. Having these example files will help you to develop a template project report that can input the type of data that you typically collect for this type of project.

For the example set of studies for this module, there are a few types of data that we plan to typically collect for each study. First, we will be recording metadata for each experiment. This will include a study ID, as well as details like the mouse strain that we used in that experiment, the route used to administer the

treatment, the treatments per week and total weeks of treatment, the inoculum used for the challenge, and so on. Second, we'll be recording some details about each experimental group that was tested. This includes the drug or drugs that were tested, doses of each, and some exact details about the treatment regimen for that group. Both of these types of data can be recorded at the beginning of the study. Two other types of data will also be recorded, both of them during the study rather than at the start. The first is weights of the mice each week. These weights will be recorded for each treatment group each week of treatment, to help see if there are drugs that are poorly tolerated by the mice (which can show up through weight decreases in mice in that group). The second is the bacterial load in the lungs of each mouse at the end of the treatment period.

To create the project directory template for these studies, then, we'll create data collection templates for each of these types of data. We'll create a separate spreadsheet for each type of data, but we can group them into files if we'd like. In our example, we created two files to store this type of data, one for the metadata that are recorded at the start of the experiment (overall experiment details and the details of each tested treatment) and one for the data that are collected over the course of the experiment (mouse weights and bacterial loads). Within each file, we've used separate sheets to record the different types of data. This allows us to keep similar types of data together in the same file, while having a tidy collection format for each specific type of data. [Figure]

All of these data collection files are designed using the principles of tidy data collection (modules [x]). This will ensure that it will be easy to read these recorded data in a programming language like R for analysis and visualization. Specifically, when we designed these template files, we thought a lot about things like using a two-dimensional structure (one row of header names and then values within each of the columns), using column names that would be easy for a programming language to parse (e.g., no special characters or spaces in the column names), and so on.

Now that we have these data collection templates for this type of study, we can decide how to organize the project directory into subdirectories with the different types of data. In this case, we'll use a simple structure. We'll save the metadata file in the top level of the project directory, since it provides metadata on the project as a whole. For the data that are collected during the experiment, we'll move that data collection file into a subdirectory called "data", with its own subdirectory for "recorded_data" (indicating that we recorded it by hand in the laboratory). If you had other types of data, you could create other subdirectories for each type. For example, you could have a "flow_data" subdirectory for data collected through flow cytometry.

If you had raw data that required pre-processing, you could create subdirectories both for the raw data and for the processed data that result from pre-processing steps. For example, if you had data from [RNA sequencing?], you might have [initial files] and [processed files], as well as a [bash?] script that

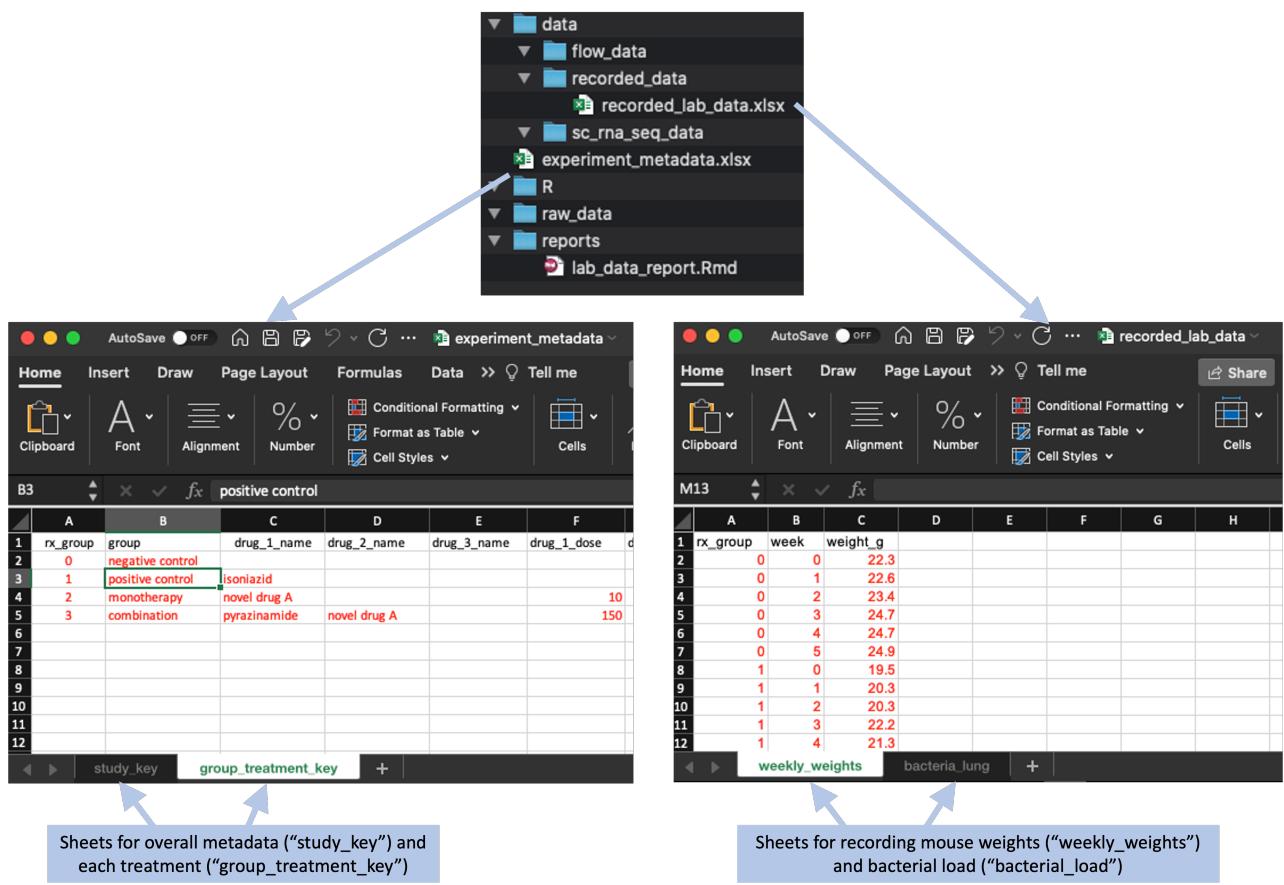


Figure 2.39: Data collection templates for the example project directory template. These templates were created in two files, one for metadata, which is saved in the main directory of the project, and one for data collected in the laboratory during the experiment, which is saved in the 'data' subdirectory. Each file is saved as a spreadsheet file, with two sheets in each file to store different types of data.

you used to generate the processed files from the initial raw data. You could create a directory called “raw_data” to use to store both the initial raw data and the script used to process that data, then a “rna_seq_data” subdirectory in the “data” subdirectory to store the smaller pre-processed files.

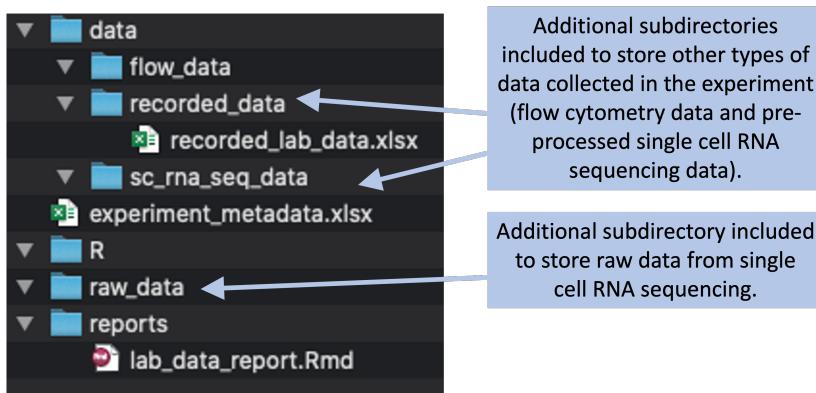


Figure 2.40: Example of a more complex project directory structure that could be created, with directories added to store data collected through flow cytometry and single cell RNA sequencing.

With the previous steps, you will have determined the types of files you normally have for this type of study, as well as structured the project directory to organize these files. The next step is to create a template report. You can create this using tools for reproducible reports—in R, a key tool for this is RMarkdown. Here, we’ll cover using this tool for creating a report briefly, but there are many more details in modules [x].

Briefly, RMarkdown allows you to include both code and text meant for humans within a single, plain text document. This document can then be rendered, a process that executes the code and formats the text meant for humans, producing a document in an easy-to-read format like Word or PDF. When it comes to project directories, it turns out that you can use the directory structure in your favor when you create these reports.

There are functions in R, for example, that will allow you to print all the files in a specified subdirectory. Say that you have several flow cytometry files in a subdirectory of the “data” subdirectory called “flow_data”. You could use this function in R to create a list of all the files in that subdirectory, and then you can run other functions to do the same operations on all of those files.

When you create a project directory template, we recommend that you create a subdirectory named something like “reports” to use to store any Rmarkdown report files for the project. This organization will make it clear where you’ve stored your reports in the project directory. You’ll be able to use file and directory pathnames to access all the data in the project, so it will be easy to use the study’s data in the report even if they’re in separate subdirectories. There’s only one tool you’ll need to do this—you’ll need to learn how to use relative pathnames within R code to access files in a different part of your project directory.

A pathname gives the directions to a file that is stored somewhere, for example on your computer or a server. There are two ways to state a pathname—you can state it as either an absolute pathname or a relative pathname. An absolute pathname gives the directions to the file from the root directory of the computer where it's stored. In other words, it gives the directions starting from the very earliest point in the file directory for that computer. A relative pathname, on the other hand, gives the directions to the file from something called the current working directory—that is, the directory that your current program is currently operating from.

You have likely already used relative pathnames extensively. If you read in a file to a statistical program like R or Python, if that file is in the current working directory, you only have to give its file name to point the program to the file. You may not have realized it, but in this case, you were using the simplest type of relative pathname—since the file is already in your working directory, you don't need to give directions that move through different directories to get from your current working directory to the file.

If you have a file in a subdirectory of the current working directory, you can use a relative pathname to access it by giving the direction through the subdirectories to get to the file. For example, if you want to read in a file named “bacterial_counts.xlsx” that is in a subdirectory of the current working directory called “data”, you could point to that file using the relative pathname “data/bacterial_counts.xlsx”. Instead, if you want to point to a file with the same name that's in a subdirectory called “reported_data” of the “data” subdirectory, you'd use the relative pathname “data/reported_data/bacterial_counts.xlsx”.

You can use relative pathnames to navigate, too, to files that aren't in a subdirectory for the current file. To do this, you can use [abbreviations?] for filenames. One of the most useful is “.”, which stands for the parent (i.e., one above) the current working directory. For example, if you have a project directory, and you've put an RMarkdown file in the “reports” subdirectory, but within the code in that file you want to read in data from the “data” subdirectory of the same project, you could do so with a relative pathname like “./data/bacterial_counts.xlsx”. This pathname says to move up one directory from the current working directory (in other words, to the directory that the current work directory is a subdirectory of) and then look for a different subdirectory of that parent directory called “data”, then read a file from that subdirectory.

This strategy is necessary when you're using RMarkdown for your reports and have created a project directory template with different subdirectories for your data versus your reports. In an RMarkdown document, the code will run using the directory where the RMarkdown file is stored as the working directory [correct even with Projects?]. Therefore, if you organize your project directory in the type of structure we've recommended, then you'll need to use this type of relative pathname to access data elsewhere in the project directory when you write code for the report.

We can take a look at how this works in the project directory template we created for the example set of studies. In module [x] we gave details on what is included in the template report for this project directory template. Briefly, it is a file that will generate a couple of tables with metadata on the experiment (overall experiment details as well as details on each of the treatments), then a graph showing mouse weights over time, then a graph showing the bacterial load at the end of the study in mice grouped by treatment, and finally a table giving the results of comparing the bacterial load in each treatment group to the bacterial load in the control group.

We created an Rmarkdown file that does this analysis and visualization and included it in the project template directory. This means that the report file will be copied and available each time someone copies the project template directory at the start of a new project. We wrote the code in a way that will input data that are stored in the data collection files that also come with the project directory template. Since we named those files in the directory template, we can refer to them with the same name in the code for the report. We wrote the code in the report in a way that it will still run if there are more or fewer observations in any of the data collection files, so the report template has some flexibility in terms of how each study in the set of studies might vary. For example, in the example set of studies, some of the experiments were run using only a control group of mice, while others were run to test as many as [x] different treatment groups. The report template can accommodate these differences across studies in the set of studies.

...

In many cases, you may have a more complex design for your project directory. For example, if you were collecting flow cytometry data for the project as well, then you would want a subdirectory in the project that is specifically designed to store files from the flow cytometry component of the experiment. This subdirectory would likely include several files, rather than just one. Further, you would not know ahead of time what the name of these files would be (as you do with the data collection template files that are included in the template directory). However, you can still easily write code for a template report file that will work with multiple files of a similar type, even if you don't know what the names will be, as long as you know what the name of their subdirectory will be. There are functions in R like `list.files` that can be used to list all the file names for the files in a given directory. You can use this function to create a vector of all the file names. For example, you could run:

to get an object (`flow_filepaths`) that lists each of the filepaths for the files stored in the “`flow_data`” subdirectory within the “`data`” subdirectory of the project. You could then “map” a function or group of functions across these files to read them in, process them, and join them into a single dataframe in R. By using this process, you can write template code in the report for the project that should work in most cases for the data that you collect for a given type of study.

The report template is included in the project directory template, so it will be copied and available for you to use anytime you start a new project using that template. However, you are not obligated to keep the report identical to the template. Instead, the template report serves as a starting point, and you can add to it or adapt it as you work on a study.

For example, in our example template report, we've included the results of applying a statistical test that compares each treatment group to the control group. Instead, for a specific study of this type, you may want to control treatments against each other. For example, some of the studies in this example set of studies include a positive control group, where the mice are treated with a drug that is already in common use for the disease. In some cases, the researcher may want to control the bacterial load in groups treated with a novel drug to groups treated with the positive control. You could easily add to the report template in that case, adding statistical tests where you compare different treatment groups to each other.

2.8.2 *Creating an R Project template*

In the previous section, we covered a very basic way to make a project directory template: you make an example directory of project files, and then anytime you start a project for that type of study, you copy, rename, and use that example directory as a template.

There is a second way to make project templates for your research group. This method requires a lot more work at the beginning. However, it makes it very easy for anyone in your group to use the template once it's been created. This method is to create an R Studio Project template. To be clear, with the basic method covered in the last chapter, you can create a project directory based on your basic template and then convert it to an RStudio Project within RStudio. However, with this second method, you will gain the option to create a new project based on your template from within RStudio, and many of the details of creating the template are encapsulated within the process, so you're more likely to be able to enforce a common project directory structure across projects.

This method does require a good bit of familiarity with R programming, as it requires you to create a new R package. In this section, we'll go over the process, using the example set of studies for this module as a motivating example. ...

...

There are several steps that you'll take to create the RStudio Project template:

1. Decide on project directory structure
2. Create templates for recording data
3. Create a template for a report for the project
4. Create an R package to implement the Project template

5. Move the templates for data recording and the report into the “inst” directory of the Project package
6. Write code in the package to create project structure and copy in templates
7. Customize the Project Wizard for the Project in the package code
8. Build the Project template package and share it with your research group

Several of these are the same as the steps to create a basic project directory template. Specifically, steps 1–3 are the same steps that you would take to create a basic project directory template.

The novel steps for this process come from step 4 and later. Rather than creating a file directory that your research group will copy and rename, we’ll put all the elements of the project template within an R package that you can build and then share with your research group. They will be able to install this package, and then whenever they start a project, they will be able to initialize it as this special type of project from within RStudio, as described in the previous module.

Under this method, the template is put together and shared as an R package. An R package is a collection of R code and data that extend the functionality of base R. If you have coded with R, you’ve likely used these types of packages to get access to useful functions. For example, the “tidyverse” (module [x]) is a suite of popular R packages with functions for working with data. Anyone can create and share an R package, either broadly and publicly by submitting it to a repository like Bioconductor (in which case it will need to follow some additional constraints and pass some checks) or privately as a compressed file that anyone you share it with can download to their computer and install.

The new steps in this process start with step 4, which is to create an R package to implement the Project template. You can create the framework for an R package very easily in R studio. Open RStudio, go to “File”, and select “New Project”, as you would if you were creating any type of RStudio Project. Choose “New Directory”, but then instead of choosing the generic “New Project”, choose “R Package”. This will create an open a new Project directory for you package, one that includes templates for many of the files that you need to start a new R package. (As a note, this is using a special R project template, just like you’ll be creating!)

[More on the initial package directory]

...

For most R packages, you create them to use to share new R functions. Therefore, the package directory is focused on new R scripts. However, the R package structure is very flexible, and it can be effectively used to share other things as well. For example, R packages can be created to share datasets, with very few or no new functions included. Similarly, the R package that we’ll make with your template will allow you to share a directory structure and some “starter” files to go into that structure, rather than R code.

Since we’re using the R package structure for this alternative use, it may

at first seem a bit confusing why we need to put certain files in certain places or write code that moves files around. Briefly, R packages are created and shared essentially as file directories, but these file directories must follow a very specific structure, with specific names for different subdirectories and files within the structure. We'll need to follow these rules as we organize the template files within our R package for our Project template. We will explain as we move through this section where you should save everything in your R package directory, but this is just to clarify that there is a reason that the structure might seem a bit convoluted—we're leveraging a structure normally meant for a bit of a different use.

You can make as many template files as you'd like to include in your Project template. When you make the R project with your template, you'll store these all in a special directory that you should call "inst". In an R package, the "inst" directory can act as a bit of a catch-all, where you can store files that you would like to be included when someone installs your package, but that don't naturally fit in another part of the package directory.

Within this "inst" subdirectory, you can create as many subdirectories as you'd like, to help you organize your template files. The only rule is that there are a handful of names that you should avoid for these subdirectory names. This is because, when the package is installed, all the subdirectories in this "inst" subdirectory will be moved up to the main package directory, and so you should avoid directory names that have a specific meaning in an R package. Namely, you should avoid naming any subdirectories in the "inst" subdirectory any of the following: "data", "R", "src", "man", "demo", "tests", "exec", "po", "tools", "vignettes" [?], "build" [?], "share" [?], "licenses" [?].

In the template for the example set of studies for this module, for example, we created three subdirectories in the "inst" subdirectory to use to store templates for this type of Project: "data_collection_templates", where we stored an Excel file with the template for collecting data during each experiment (i.e., the mice weights and final bacterial loads), "metadata_templates", which includes an Excel file with the template for collecting metadata on the experiment on a whole and on each of the tested treatments, and "report_templates", where we stored an RMarkdown file with the template report for the project. For any templates that include code, keep in mind that you'll need to be careful in setting the pathnames to access any data in the project directory. [More on this?]

Once you've added any templates to the "inst" directory, the next step is to write code in the template package. For this type of package, which aims to set up a new Project structure, you'll only include R code for one purpose—the R code will run when someone first opens a new Project of this type, and it will operate to set up the initial project structure. R can be used to do many different things—while you may have mostly used it before to read in and analyze data, it can also be used to do things like make new directories on your computer, or to copy files from one place in your computer's file directory to

another. We'll be using these types of commands in this template R package, as it will allow us to move the template files that you created into the right place in a user's new project directory when they chose to use your template to make a new project.

There are four key R functions that we'll be using to do these tasks. The first is the `dir.create` function. This can be used to create new directories on the user's computer. You can include the name that you'd like to use as the new subdirectory. We'll use this function to create the subdirectories in the new project, and so this function will let us create a structure within the user's subdirectory.

The second key R function that we'll use is `file.copy`. This function allows us to copy a file from one place in the user's computer to another spot. When the user installs our package, all of the template files will be included with that installation. With `file.copy`, we'll be able to copy these template files into the right spot on each user's computer each time they open a new Project with this template.

The third key R function also helps with this process of copying the template files. In order to copy the files, we need to be able to find where they're stored on the user's computer. Since they've been installed with the package, the template files will be located on the user's computer based on where that user stores their installed R packages. This location can differ by user, but fortunately there's a function called `system.file` that lets us figure out the file path for any file that's stored in an R package that the user has installed, on that user's computer. This function will therefore allow us to get the original filepath for each of the template files, so we can copy them into the new Project.

Finally, the fourth key R function is `file.path`. This function can create a file path, and it's helpful because it can allow you to write code that will generate the full path to a file or subdirectory on the user's computer, based on the relationship of that file or subdirectory to the working directory of the project they've created. We'll use this function, for example, to create the file path for the new subdirectories that should go in the Project when the user opens it. Even though each user could be storing their project in a different part of their computer's file directory, this function can determine the path we should use for each subcomponent in the project.

[How to create directories and move in the template files]

Using these functions, we'll create the subdirectory structure of the project and then move any template files into the right place in that structure. This will happen in an R function—in many cases, this might be the only R function that you'll include in the package that defines the Project template. You can name this R function anything you'd like (later, we'll show how you can connect this function so that it's automatically run when someone makes their new Project with the template). You should save the code for the function in the R subdirectory of your package directory.

In the example for this module, we've named the function `create_project` and stored the code for it in a file called "create_project.R" in the R subdirectory of our package. This function should take at least two inputs: `path`, which will be the filepath to the project's directory, and . . . (which we'll call "dots"), which allows other arguments to pass into the function.

Within the function, we've included code that will create all the subdirectories for this type of project, as well as code that moves the template files into the right place in those subdirectories. First, ...

[How to customize the Project Wizard for the project]

As you work on the package, you can build it and test it on your own computer. If you are working in RStudio, you can use a "Build" Pane that is available when you're working on a R Project that is a package. In this pane, there is a "Install and Reload" button [doublecheck]—if you click this button, it will build the package and install it on your computer in the same place that other R packages are installed (for example, packages that you install from CRAN).

Once you've built and installed your package, give it a try. You can go up to the "File" menu in RStudio and select "New Project". Select "New Directory", and this will take you to a menu with the different types of R Projects you can create. You should now see your Project template as one of the options. Click on this, and you can test out if the template's working like you hoped it would.

[How a different user can download and install a package that isn't on CRAN]

If you would like to create one of these RStudio Project templates for studies in your research group, there are more details on the process at ... In addition, since this method requires building an R package, you may find resources on creating R packages to be useful. One excellent book on the topic, which is available for free online, is [R Packages] by Hadley Wickham and Jenny Bryan.

2.8.3 Applied exercise

2.9 Harnessing version control for transparent data recording

As a research project progresses, a typical practice in many experimental research groups is to save new versions of files (e.g., 'draft1.doc', 'draft2.doc'), so that changes can be reverted. However, this practice leads to an explosion of files, and it becomes hard to track which files represent the 'current' state of a project. Version control allows researchers to edit and change research project files more cleanly, including messages to explain changes, while maintaining the power to 'backtrack' to previous versions. We will explain what version control is and how it can be used in research projects to improve the transparency and reproducibility of research, particularly for data recording.

Objectives. After this module, the trainee will be able to:

- Describe version control

- Explain how version control can be used to improve reproducibility for data recording

2.9.1 What is version control?

Version control developed as a way to coordinate collaborative work on software programming projects. The term “version” here refers to the current state of a document or set of documents, for example the source code for a computer program. The idea of “control” is to allow for safe changes and updates to this version while more than one person is working on it. The general term “version control” can refer to any method of syncing contributions from several people to a file or set of files, and very early on it was done by people rather than through a computer program. While version control of computer files can be done by people, and originally was (Irving, 2011), it’s much more efficient to use a computer program to handle this tracking of the history of a set of files as they evolve.

While the very earliest version control systems tracked single files, these systems quickly moved to tracking sets of files, called *repositories*. You can think of a repository as a computer file directory with some extra overhead added to record how the files in the directory have changed over time. In a repository of files that is under version control, you take regular “snapshots” of how the files look during your work on them. Each snapshot is called a *commit*, and it provides a record of which lines in each file changed from one snapshot to another, as well as exactly how they changed. The idea behind these commits—recording the differences, line-by-line, between an older and newer version of each file derives from a longstanding Unix command line tool called *diff*. This tool, developed early in the history of Unix at AT&T (Raymond, 2003), is an extremely solid and well-tested tool that did the simple but important job of generating a list of all the differences between two plain text files.

When you are working with a directory under version control, you can also explain your changes as you make them—in other words, it allows for *annotation* of the developing and editing process (Raymond, 2009). Each commit requires you to enter a *commit message* describing why the change was made. The commit messages can serve as a powerful tool for explaining changes to other team members or for reminding yourself in the future about why certain changes were made. As a result, a repository under version control includes a complete history of how files in a project directory have changed over the timecourse of the project and why. Further, each of the commits is given its own ID tag (a unique SHA-1 hash), and version control systems have a number of commands that let you “roll back” to earlier versions, by going back to the version as it was when a certain commit was made, providing *reversability* within the project files (Raymond, 2009).

It turns out that this functionality—of being able to “roll back” to earlier versions—has a wonderful side benefit when it comes to working on a large

“Tracking all that detail is just the sort of thing computers are good at and humans are not.”

[@raymond2003art]

project. It means that you *don't* need to save earlier versions of each file. You can maintain one and only one version of each project file in the project's directory, with the confidence that you never "lose" old versions of the file (Perkel, 2018a; Blischak et al., 2016). This allows you to maintain a clean and simple version of the project files, with only one copy of each, ensuring it's always clear which version of a file is the "current" one (since there's only one version). This also provides the reassurance that you can try new directions in a project, and always roll back to the old version if that direction doesn't work well.

Finally, most current version control systems operate under a *distributed* framework. In earlier types of version control programs, there was one central ("main") repository for the file or set of files the team was working on (Raymond, 2009; Target, 2018). Very early on, this was kept on one computer (Irving, 2011). A team member who wanted to make a change would "check out" the file he or she wanted to work on, make changes, and then check it back in as the newest main version (Raymond, 2003). While one team member had this file checked out, other members would often be "locked" out of making any changes to that file—they could look at it, but couldn't make any edits (Raymond, 2009; Target, 2018). This meant that there was no chance of two people trying to change the same part of a file at the same time. In spirit, this early system is pretty similar to the idea of sending a file around the team by email, with the understanding that only one person works on it at a time. While the "main" version is in different people's hands at different times, to do work, you all agree that only one person will work on it at a time. A slightly more modern analogy is the idea of having a single version of a file in Dropbox or Google Docs, and avoiding working on the file when you see that another team member is working on it.

This system is pretty clunky, though. In particular, it usually increases the amount of time that it takes the team to finish the project, because only one person can work on a file at a time. Later types of version control programs moved toward a different style, allowing for *distributed* rather than *centralized* collaborative work on a file or a set of files (Raymond, 2009; Irving, 2011). Under the distributed model, all team members can have their own version of all the files, work on them and make records of changes they make to the files, and then occasionally sync with everyone else to share your changes with them and bring their changes into your copy of the files. This distributed model also means there is a copy of the full repository on every team member's computer, which has the side benefit of provided natural backup of the project files. Remote repositories—which may be on a server in a different location—can be added with another copy of the project, which can similarly be synced regularly to update with any changes made to project files.

While there are a number of software systems for version control, by far the most common currently used for scientific projects is *git*. This program was created by Linus Torvalds, who also created the Linux operating system,

"Early in his graduate career, John Blischak found himself creating figures for his advisor's grant application. Blischak was using the programming language R to generate the figures, and as he iterated and optimized his code, he ran into a familiar problem: Determined not to lose his work, he gave each new version a different filename—analysis_1, analysis_2, and so on, for instance—but failed to document how they had evolved. 'I had no idea what had changed between them,' says Blischak... Using Git, Blischak says, he no longer needed to maintain multiple copies of his files. 'I just keep overwriting it and changing it and saving the snapshots. And if the professor comes back and says, 'oh, you sent me an email back in March with this figure', I can say, 'okay, well, I'll just go back to the March version of my code and I can recreate it'."

[@perkel2018git]

in 2005 as a way to facilitate the team working on Linux development. This program for version control thrives in large collaborative projects, for example open-source software development projects that include numerous contributors, both regular and occasional (Brown, 2018).

In recent years, some complementary tools have been developed that make the process of collaborating together using version control software easier. Other tools, such as bug trackers or issue trackers, facilitate corroborative file-based projects to allow the team to keep a running “to-do” list of what needs to be done to complete the project. These tools—which are discussed in the next chapter—can be used to improve collaboration on scientific projects spread across teams. GitHub, a very popular version control platform with these additional tools, was created in 2008 as a web-based platform to facilitate collaborating on projects running under git version control. It can provide an easier entry to using git for version control than trying to learn to use git from the command line (Perez-Riverol et al., 2016). It also plays well with RStudio, making it easy to integrate a collaborative workflow through GitHub from the same RStudio window on your computer where you are otherwise doing your analysis (Perez-Riverol et al., 2016).

2.9.2 Recording data in the laboratory—from paper to computers

Traditionally, experimental data collected in a laboratory was recorded in a paper laboratory notebook. These laboratory notebooks played a role not only as the initial recording of data, but also can serve as, for example, a legal record of the data recorded in the lab (Mascarelli, 2014). They were also a resource for collaborating across a team and for passing on a research project from one lab member to another (Butler, 2005).

However, paper laboratory notebooks have a number of limitations. First, they can be very inefficient. In a time when almost all data analyses—even simple calculations—are done on a computer, recording research data on paper rather than directly entering it into a computer is inefficient. Also, any stage of copying data from one format to another, especially when done by a human rather than a machine, introduces the chance to copying errors. Handwritten laboratory notebooks can be hard to read (Butler, 2005; Perkel, 2011), and may lack adequate flexibility and expandability to handle the complex experiments often conducted. Further, electronic alternatives can also be easier to search, allowing for deeper and more comprehensive investigations of the data collected across multiple experiments (Giles, 2012; Butler, 2005; Perkel, 2011).

Given a widespread recognition of the limitations of paper laboratory notebooks, in the past couple of decades, there have been a number of efforts, both formal and informal, to move from paper laboratory notebooks to electronic alternatives. In some fields that rely heavily on computational analysis, there are very few research labs (if any) that use paper laboratory notebooks (Butler, 2005). In other fields, where researchers have traditionally used paper lab

“If your software engineering career, like mine, is no older than GitHub, then git may be the only version control software you have ever used. While people sometimes grouse about its steep learning curve or unintuitive interface, git has become everyone’s go-to for version control.”

[@target2018version]

“Handwritten lab notebooks are usually chaotic and always unsearchable.”

[@perkel2011coding]

notebooks, companies have been working for a while to develop electronic laboratory notebooks specifically tailored to scientific research needs (Giles, 2012). These were adopted more early in pharmaceutical industrial labs, where companies had the budgets to get customized versions and the authority to require their use, but have taken longer to be adapted in academic laboratories (Giles, 2012; Butler, 2005). A widely adopted platform for electronic laboratory notebooks has yet to be taken up by the scientific community, despite clear advantages of recording data directly into a computer rather than first using a paper notebook.

Instead of using customized electronic laboratory notebook software, some academics are moving their data recording online, but are using more generalized electronic alternatives, like Dropbox, Google applications, OneNote, and Evernote (Perkel, 2011; Kwok, 2018; Giles, 2012; Powell, 2012). Some scientists have started using version control tools, especially the combination of git and GitHub, as a way to improve laboratory data recording, and in particular to improve transparency and reproducibility standards. These pieces of software share the same pattern as Google tools or Dropbox—they are generalized tools that have been honed and optimized for ease of use through their role outside of scientific research, but can be harnessed as a powerful tool in a scientific laboratory, as well. They are also free—at least, for GitHub, at the entry and academic levels—and, even better, one (git) is open source.

While some generalized tools like Google tools and Dropbox might be simpler to initially learn, version control tools offer some key advantages for recording scientific data and are worth the effort to adopt. A key advantage is their ability to track the full history of files as they evolve, including not only the history of changes to each file, but also a record of why each change was made. Git excels in tracking changes made to plain text files. For these files, whether they record code, data, or text, git can show line-by-line differences between two versions of the file. This makes it very easy to go through the history of “commits” to a plain text file in a git-tracked repository and see what change was made at each time point, and then read through the commit messages associated with those commits to see why a change was made. For example, if a value was entered in the wrong row of a csv, and the researcher then made a commit to correct that data entry mistake, the researcher could explain the problem and its resolution in the commit message for that change.

Platforms for using git often include nice tools for visualizing differences between two files, providing a more visual way to look at the “diffs” between files across time points in the project. For example, GitHub automatically shows these using colors to highlight additions and subtractions of plain text for one file compared to another version of it when you look through a repository’s commit history. Similarly, RStudio provides a new “Commit” window that can be used to compare differences between the original and revised version of plain text files at a particular stage in the commit history.

The use of version control tools and platforms, like git and GitHub, not only

“Since at least the 1990s, articles on technology have predicted the imminent, widespread adoption of electronic laboratory notebooks (ELNs) by researchers. It has yet to happen—but more and more scientists are taking the plunge.”

[@kwok2018lab]

“The purpose of a lab notebook is to provide a lasting record of events in a laboratory. In the same way that a chemistry experiment would be nearly impossible without a lab notebook, scientific computing would be a nightmare of inefficiency and uncertainty without version-control systems.”

[@tippmannmy2014digital]

helps in transparent and trackable recording of data, but it also brings some additional advantages in the research project. First, this combination of tools aids in collaboration across a research group, as we discuss in depth in the next chapter.

Second, if a project uses these tools, it is very easy to share data recorded for the project publicly. In a project that uses git and GitHub version control tools, it is easy to share the project data online once an associated manuscript is published, an increasingly common request or requirement from journals and funding agencies (Blischak et al., 2016). Sharing data allows a more complete assessment of the research by reviewers and readers and makes it easier for other researchers to build off the published results in their own work, extending and adapting the code to explore their own datasets or ask their own research questions (Perez-Riverol et al., 2016). On GitHub, you can set the access to a project to be either public or private, and can be converted easily from one form to the other over the course of the project (Metz, 2015). A private project can be viewed only by fellow team members, while a public project can be viewed by anyone. Further, because git tracks the full history of changes to these documents, it includes functionality that lets you tag the code and data at a specific point (for example, the date when a paper was submitted) so that viewers can look at that specific “version” of the repository files, even while the project team continues to move forward in improving files in the directory. At the more advanced end of functionality, there are even ways to assign a persistent digital identifier (e.g., a DOI, like those assigned to published articles) to a specific version of a GitHub repository (Perez-Riverol et al., 2016).

Third, the combination of git and GitHub can help as a way to backup study data (Blischak et al., 2016; Perez-Riverol et al., 2016; Perkel, 2018a). Together, git and GitHub provide a structure where the project directory (repository) is copied on multiple computers, both the users’ laptop or desktop computers and on a remote server hosted by GitHub or a similar organization. This set-up makes it easy to bring all the project files onto a new computer—all you have to do is clone the project repository. It also ensures that there are copies of the full project directory, including all its files, in multiple places (Blischak et al., 2016). Further, not only is the data backed up across multiple computers, but so is the full history of all changes made to that data and the recorded messages explaining those changes, through the repositories commit messages (Perez-Riverol et al., 2016).

There are, of course, some limitations to using version control tools when recording experimental data. First, while ideally laboratory data is recorded in a plain text format (see the module in section 2.2 for a deeper discussion of why), some data may be recorded in a binary file format. Some version control tools, including git, can be used to track changes in binary files. However, git does not take to these types of files naturally. In particular, git typically will not be able to show users a useful comparison of the differences between two versions of a binary file. More problems can arise if the binary file is very large (Perez-Riverol

“You can version control any file that you put in a Git repository, whether it is text-based, an image, or a giant data file. However, just because you *can* version control something, does not mean that you *should*.¹

[@blischak2016quick]

et al., 2016; Blischak et al., 2016), as some experimental research data files are (e.g., if they are high-throughput output of laboratory equipment like a mass spectrometer). However, there are emerging tools and strategies for improving the ability to include and track large binary files when using git and GitHub (Blischak et al., 2016).

Finally, as with other tools and techniques described in this book, there is an investment required to learn how to use git and GitHub (Perez-Riverol et al., 2016), as well as a bit of extra overhead when using version control tools in a project (Raymond, 2003). However, both can bring dramatic gains to efficiency, transparency, and organization of research projects, even if you only use a small subset of its basic functionality (Perez-Riverol et al., 2016). In Chapter 11 we provide guidance on getting started with using git and Github to track a scientific research project.

“Although Git has a complex set of commands and can be used for rather complex operations, learning to apply the basics requires only a handful of new concepts and commands and will provide a solid ground to efficiently track code and related content for research projects.”

[@perez2016ten]

2.10 Enhance the reproducibility of collaborative research with version control platforms

Once a researcher has learned to use *git* on their own computer for local version control, they can begin using version control platforms (e.g., *GitLab*, *GitHub*) to collaborate with others under version control. We will describe how a research team can benefit from using a version control platform to work collaboratively.

Objectives. After this module, the trainee will be able to:

- List benefits of using a version control platform to collaborate on research projects, particularly for reproducibility
- Describe the difference between version control (e.g., *git*) and a version control platform (e.g., *GitLab*)

2.10.1 What are version control platforms?

The last module introduced the idea of version control, including the popular software tool often used for version control, *git*. In this module, we'll go a step further, telling you about how you can expand the idea of version control to leverage it when collaborating across your research team, using **version control platforms**.

When research groups—or any other professional teams—collaborate on publications and research, the process can be a bit haphazard. Teams often use emails and email attachments to share updates on the project, and email attachments to pass around the latest version of a document for others to review and edit. For example, one group of researchers investigated a large collection of emails from Enron (Hermans and Murphy-Hill, 2015). They found that passing Excel files through email attachments was a common practice, and that messages within emails suggested that spreadsheets were stored locally, rather than in a location that was accessible to all team members (Hermans and

Murphy-Hill, 2015), which meant that team members might often be working on different versions of the same spreadsheet file. They note that “the practice of emailing spreadsheets is known to result in serious problems in terms of accountability and errors, as people do not have access to the latest version of a spreadsheet, but need to be updated of changes via email.” (Hermans and Murphy-Hill, 2015) The same process for collaboration is often used in scientific research, as well: one study found, “Team members regularly pass data files back and forth by hand, by email, and by using shared lab or project servers, websites, and databases.” (Edwards et al., 2011)

These practices make it very difficult to keep track of all project files, and in particular, to track which version of each file is the most current. Further, this process constrains patterns of collaboration—it requires each team member to take turns in editing each file, or for one team member to attempt to merge in changes that were made by separate team members at the same time when all versions are collected. Further, this process makes it difficult to keep track of why changes were made, and often requires one team member to approve the changes of other team members. While the “Track changes” and comment features can help the team communicate with each other, these features often lead to a very messy document at stages in the editing, where it is hard to pick out the current versus suggested wording, and once a change is accepted or a comment deleted, these conversations are typically lost forever. Finally, word processing tools are poorly suited to track changes or add suggestions directly to data or code, as both data and code are usually saved in formats that aren’t native to word processing programs, and copying them into a format like Word can introduce problematic hidden formatting that can cause the data or code to malfunction.

A version control platform allows you to share project files across a group of collaborators while keeping track of what changes are made, who made each change, and why each change was made. It therefore combines the strengths of a “Track changes” feature with those of a file sharing platform like Dropbox. To some extent, Google Docs or Google Drive also combine these features, and some spreadsheet programs are moving toward some rudimentary functionality for version control (Birch et al., 2018). However, there are added advantages of version control platforms. Since open-source version control platforms like GitHub can be set up on a server that you own, they can be used to collaborate on projects with sensitive data, and also can store data directly on the server you would like to use to store large project datasets or to run computationally-intensive pre-processing or analysis. Finally, most version control platforms include tools that help you manage and track the project. These include “Issue Trackers”, tools for exploring the history of each file and each change, and features to assign project tasks to specific team members. The next section will describe the features of version control platforms that make them helpful as a tool for collaborating on scientific research. These systems are being leveraged by some scientists, both to manage research projects and also to

“The most primitive (but still very common) method [of version control] is all hand-hacking. You snapshot the project periodically by manually copying everything in it to a backup. You include history comments in source files. You make verbal or email arrangements with other developers to keep their hands off certain files while you hack them. … The hidden costs of this hand-hacking method are high, especially when (as frequently happens) it breaks down. The procedures take time and concentration; they’re prone to error, and tend to get slipped under pressure or when the project is in trouble—that is exactly when they are needed.”

[@raymond2003art]

collaborate on writing scientific manuscripts and grant proposals (Perez-Riverol et al., 2016).

Version control platforms are always used in conjunction with version control software, like the *git* software described in the last module. Version control itself has been described as “a suite of programs that automates away most of the drudgery involved in keeping an annotated history of your project and avoiding modification conflicts,” (Raymond, 2003). The version control platform leverages the history of commits that were made to the project, as well as the version control software’s capabilities for merging changes made by different people at different times. On top of these facilities, a version control platform also adds attractive visual interfaces for working with the project, free or low-cost online hosting of project files, and team management tools for each project. You can think of *git* as the engine, in other words, and the version control platform as the driver’s seat, with dashboard, steering wheel, and gears to leverage the power of the underlying *git* software.

A number of version control platforms are available. Two that are currently very popular for scientific research are GitHub (<https://github.com/>) and GitLab (<https://about.gitlab.com/>). Both provide free options for scientific researchers, including the capabilities for using both public and private repositories in collaboration with other researchers.

2.10.2 Why use version control platforms?

Version control platforms offer a number of advantages when collaborating on a research project that can help to improve your efficiency, rigor, and reproducibility. Further, there are several high-quality free versions of version control platforms that are available for researchers, and as their use becomes more popular, there are more and more resources to help you learn how to use these platforms effectively. Open-source versions, like GitLab, even allow you to set up a version control platform on a server you own, rather than needing to post data or code on an outside platform, and so you can use these tools even in cases involving sensitive data.

Some of the key advantages of using a version control platform like GitHub to collaborate on research projects include:

- Ability to track and merge changes that different collaborators made to the document
- Ability to create alternative versions of project files (*branches*), and merge them into the main project as desired
- Tools for project management, including Issue Trackers
- Default backup of project files
- Ability to share project information online, including through hosting websites related to the project or supplemental files related to a manuscript

Many of these strengths draw directly on the functions provided by the underlying version control software (e.g., *git*). However, the version control

“Using GitHub or any similar versioning / tracking system is not a replacement for good project management; it is an extension, an improvement for good project and file management.”

[@perez2016ten]

Resources like GitHub are “essential for collaborative software projects because they enable the organization and sharing of programming tasks between different remote contributors.”

[@perez2016ten]

platform will typically allow team members to explore and work with these functions in an easier way than if they try to use the barebones version control software. In earlier years, the use of version control often required users to be familiar with the command line, and to send arcane commands to track the project files through that interface. With the rising popularity of version control platforms, version control for project management can be taught relatively quickly to students with a few months—or even weeks—of coding experience. In fact, version control is beginning to be used as a method of turning in and grading homework in beginning programming classes, with students learning these techniques in the first few weeks of class. This would be practically unimaginable without the user-friendly interface of a version control platform as a wrapper for the power of the version control software itself.

The first strength of using version control—and a version control platform—to collaborate on scientific projects is its ability to track every change made to files in the project, why the change was made, and who made it. Version control creates a full history of the evolution of each file in the project. When a change is committed, the history records the exact change made, including the previous version of the file. No change is ever fully lost, therefore, unless a great deal of extra work is taken to erase something from the project's commit history. Version control also requires a user to provide a *commit message* describing each change that is made. If this feature is used thoughtfully, then the commit history of the project provides a well-documented description of the project's full evolution. If you're working on a manuscript, for example, when it's time to edit, you can cut whole paragraphs, and if you ever need to get them back, they'll be right there in the commit history for your project, with their own commit message about why they were cut (hopefully a nice clear one that will make it easy to find that commit if you ever need those paragraphs again).

These capacities to track changes and histories of project files becomes even more important when working in collaboration on a project. As the proverb about too many cooks in the kitchen captures, any time you have multiple people working on a project, it introduces the chance for conflicts. While higher-level conflicts, like about what you want the final product to look like or who should do which jobs, can't be easily managed by a computer program, now the complications of integrating everyone's contributions—and letting people work in their own space and then bring together their individual work into one final joint project—can be. While these programs for version control were originally created to help with programmers developing code, they can be used now to coordinate group work on numerous types of file-based projects, including scientific manuscripts, books, and websites (Raymond, 2009). And although they can work with projects that include binary code, they thrive in projects with a heavier concentration of text-based files, and so they fit in nicely in a scientific research / data analysis workflow that is based on data stored in plain text formats and data analysis scripts written in plain text files,

"One reason for GitHub's success is that it offers more than a simple source code hosting service. It provides developers and researchers with a dynamic and collaborative environment, often referred to as a social coding platform, that supports peer review, commenting, and discussion. A diverse range of efforts, ranging from individual to large bioinformatics projects, laboratory repositories, as well as global collaborations, have found GitHub to be a productive place to share code and ideas and collaborate."

[@perez2016ten]

"[Version control systems] are a huge boon to productivity and code quality in many ways, even for small single-developer projects. They automate away many procedures that are just tedious work. They help a lot in recovering from mistakes. Perhaps most importantly, they free programmers to experiment by guaranteeing that reversion to a known-good state will always be easy."

[@raymond2003art]

tools we discuss in other parts of this book.

Modern version control systems like *git* take a distributed approach to collaboration on project files. Under the distributed model, all team members can have their own version of all the files, work on them and make records of changes they make to the files, and then occasionally sync with everyone else to share your changes with them and bring their changes into your copy of the files. This functionality is called *concurrency*, since it allows team members to concurrently work on the same set of files (Raymond, 2009). This idea allowed for the development of other useful features and styles of working, including *branching* to try out new ideas that you're not sure you'll ultimately want to go with and *forking*, a key tool used in open-source software development, which among other things facilitates someone who isn't part of the original team getting a copy of the files they can work with and suggesting some changes that might be helpful. So, this is the basic idea of modern version control—for a project that involves a set of computer files, everyone on the team (even if that's just one person) has their own copy of a directory with those files on their own computer, makes changes at the time and in the spots in the files that they want, and then regularly re-syncs their local directory with everyone else's to share changes and updates.

There is one key feature of modern version control that's critical to making this work—merging files that started the same but were edited in different ways and now need to be put back together, bringing along any changes made from the original version. This step is called *merging* the files. While this is typically described using the plural, “files”, at a higher-level, you can think of this as just merging the *changes* that two people have made as they edited a single file, a file where they both started out with identical copies.

Think of the file broken up into each of its separate lines. There will be some lines that neither person changed. Those are easy to handle in the “merge”—they stay the same as in the original copy of the file. Next, there will be some lines that one person changed, but that the other person didn't. It turns out that these are pretty easy to handle, too. If only one person changed the line, then you use their version—it's the most up-to-date, since if both people started out with the same version, it means that the other person didn't make any changes to that part of the file. Finally, there may be a few lines that both people changed. These are called *merge conflicts*. They're places in the file where there's not a clear, easy-to-automate way that the computer can know which version to put into the integrated, latest version of the file. Different version control programs handle these merge conflicts in different ways. For the most common version control program used today, *git*, these spots in the file are flagged with a special set of symbols when you try to integrate the two updated versions of the file. Along with the special symbols to denote a conflict, there will also be *both* versions of the conflicting lines of the file. Whoever is integrating the files must go in and pick the version of those lines to use in the integrated version of the file, or write in some compromise version of those

“In a medium-sized project, it often happens that a (relatively small) number of people work simultaneously on a single set of files, the ‘program’ or the ‘project’. Often these people have additional tasks, causing their working speeds to differ greatly. One person may be working a steady ten hours a day on the project, a second may have barely time to dabble in the project enough to keep current, while a third participant may be sent off on an urgent temporary assignment just before finishing a modification. It would be nice if each participant could be abstracted from the vicissitudes of the lives of the others.”

[@grune1986concurrent]

lines that brings in elements from both people's changes, and then delete all the symbols denoting that was a conflict and save this latest version of the file.

There are a number of other features of version control that make it useful for collaborating on file-based projects with teams. First, these systems allow you to explain your changes as you make them—in other words, it allows for *annotation* of the developing and editing process (Raymond, 2009). This provides the team with a full history of why the files evolved in the way they did across the team. It also provides a way to communicate across the team members.

For example, if one person is the key person working on a certain file, but has run into a problem with one spot and asks another team member to take a go, then the second team member isn't limited to just looking at the file and then emailing some suggestions. Instead, the second person can make sure he or she has the latest version of that file, make the changes they think will help, *commit* those changes with a message (a *commit message*) about why they think this change will fix the problem, and then push that latest version of the file back to the first person. If there are several places where it would help to change the file, then these can be fixed through several separate commits, each with their own message. The first person, who originally asked for help, can read through the updates in the file (most platforms for using version control will now highlight where all these changes are in the file) and read the second person's message or messages about why each change might help. Even better, days or months later, when team members are trying to figure out why a certain change was made in that part of the file, can go back and read these messages to get an explanation.

In recent years, some complementary tools have been developed that make the process of collaborating together using version control software easier. These include *bug trackers* or *issue trackers*, which allow the team to keep a running “to-do” list of what needs to be done to complete the project, which discussed in the next chapter (Perez-Riverol et al., 2016).

Finally, version control platforms like GitHub can be used for a number of supplementary tasks for your research project. These include publishing web-pages or other web resources linked to the project and otherwise improving public engagement with the project, including by allowing other researchers to copy and adapt your project through a process called *forking*. Version control platforms also provide a supplemental backup to project files.

First, GitHub can be used to collaborate on, host, and publish websites and other online content (Perez-Riverol et al., 2016). Version control systems have been used by some for a long time to help in writing longform materials like books (e.g., (Raymond, 2003)); new tools are making the process even easier. The GitHub Pages functionality, for example, is now being used to host a number of books created in R using the bookdown package, including the online version of this book. The blogdown package similarly can be used to create websites, either for individual researchers, for research labs, or for

“You will likely share your code with multiple lab mates or collaborators, and they may have suggestions on how to improve it. If you email the code to multiple people, you will have to manually incorporate all the changes each of them sends.”

[@blischak2016quick]

“You know your code has changed; do you know why? It's easy to forget the reasons for changes, and step on them later. If you have collaborators on a project, how do you know what they have changed while you weren't looking, and who was responsible for each change?”

[@raymond2003art]

specific projects or collaborations. Further, if a project includes the creation of scientific software, it can be used to share that software—as well as associated documentation—in a format that is easy for others to work with. The platform can also be used to share supplemental material for a manuscript, including the code used for preprocessing and analyzing data. The most popular version control platforms, GitHub and GitLab, both allow users to toggle projects between “public” and “private” modes, which can be used to work privately on a project prior to peer review and publication, and then switch to a public mode after publication. This functionality will allow those who access the code to see not only the final product, but also the history of the development of the code and data for the project, providing more transparency in the development process, but without jeopardizing the novelty of the research results prior to publication.

With GitHub, while only collaborators on a public project can directly change the code, anyone else can suggest changes through a process of copying a version of the project (*forking* it). This allows someone to make the changes they would like to suggest directly to a copy of the code, and then ask the project’s owners to consider integrating the changes back into the main version of the project through a *pull request*. GitHub therefore creates a platform where people can explore, adapt, and add to other people’s coding projects, enabling a community of coders (Perez-Riverol et al., 2016), and because of this functionality it has been described as “a social network for software development” (Perkel, 2018a) and as “a kind of bazaar that offers just about any piece of code you might want—and so much of it free.” (Metz, 2015). This same process can be leveraged for others to copy and adapt code—this is particularly helpful in ensuring that a software or research project won’t be “orphaned” if its main developer is unavailable (e.g., retires, dies), but instead can be picked up and continued by other interested researchers. Copyright statements and licenses within code projects help to clarify attribution and rights in these cases.

Finally, version control platforms help in providing additional back-up for project files. As you collaborate with others using version control under a distributed model, each collaborator will have their own copy of all project files on their local computer. All project files are also stored on the remote repository to which you all push and pull commits. If you are using the GitHub platform, this will be GitHub’s servers; if you use GitLab, you can set up the system on your own server. Each time you push or pull from the remote copy of the project repository, you are syncing your copy of the project files with those on other computers.

2.10.3 How to use GitHub

In the next module, we describe practical ways to leverage these resources within your research group. We include instructions both for team leaders—who may not code but may want to use GitHub within projects to help manage

“The traditional way to promote scientific software is by publishing an associated paper in the peer-reviewed scientific literature, though, as pointed out by Buckheir and Donoho, this is just advertising. Additional steps can boost the visibility of an organization. For example, GitHub Pages are simple websites freely hosted by GitHub. Users can create and host blog websites, help pages, manuals, tutorials, and websites related to specific projects.”

[@perez2016ten]

“The astonishment was that you might want to make even your tiny hacks to other people’s code public. Before GitHub, we tended to keep those on our own computer. Nowadays, it is so easy to make a fork, or even edit the code directly in your browser, that potentially anyone can find even your least polished bug fixes immediately.”

[@irving2011astonishments]

“Backup, backup, backup—this is the main action you can take to care for your computers and your data. Many PIs assume that backup systems are inherently permanent and foolproof, and it often takes a loss to remind one that materials break, systems fail, and humans make mistakes. Even if your data are backed up at work, have at least one other backup system. Keep at least one backup off site, in case of a disaster in the lab (yes, fires and floods do happen). It doesn’t make much sense to have two separate backup systems stored next to each other in a drawer.”

[@leips2010helm]

the projects—as well as researchers who work directly with data and code for the research team. There are also a number of excellent resources that are now available that walk users through how to set up and use a version control platform. The process is particularly straightforward when the research project files are collected in an RStudio Project format, as described in earlier modules.

2.11 Using git and GitLab to implement version control

For many years, use of version control required use of the command line, limiting its accessibility to researchers with limited programming experience. However, graphical interfaces have removed this barrier, and RStudio has particularly user-friendly tools for implementing version control. In this module, we will show how to use *git* through RStudio’s user-friendly interface and how to connect from a local computer to *GitLab* through RStudio.

Objectives. After this module, the trainee will be able to:

- Understand how to set up and use *git* through RStudio’s interface
- Understand how to connect with *GitLab* through RStudio to collaborate on research projects while maintaining version control

2.11.1 How to use version control

In this chapter, we will give you an overview of how to use *git* and GitHub for your laboratory research projects. In this chapter, we’ll address two separate groups, in separate sections. First, we’ll provide an overview of how you can leverage and use these tools as the director or manager of a project, without knowing how to code in a language like R. GitHub provides a number of useful tools that can be used by anyone, providing a common space for managing the data recording, analysis and reporting for a scientific research project. In this case, there would need to be at least one member of your team who is comfortable with a programming language, but all team members can participate in many features of the GitHub repository regardless of programming skill.

Second, we’ll provide some details on the “how-tos” of setting up and using *git* and GitHub for scientists who are programmers or learning to program in a language like R or Python. We will not be exhaustive in this section, as there are a number of excellent resources that already go into depth on these topics. Instead, we provide an overview of getting starting, and what tools you might want to try within projects, and then provide advice on more references to follow up with to learn more and fully develop these skills.

As an example, we’ll show different elements from a real GitHub repository, used for scientific projects and papers. The first repository is available at https://github.com/aef1004/cyto-feature_engineering. It provides example data and code to accompany a published article on a pipeline for flow cytometry analysis (Fox et al., 2020).

2.11.2 Leveraging git and GitHub as a project director

Because git has a history in software development, and because most introductions to it quickly present arcane-looking code commands, you may have hesitations about whether it would be useful in your scientific research group if you, and many in your research group, do not have experience programming. This is not at all the case, and in fact, the combination of git and GitHub can become a secret weapon for your research group if you are willing to encourage those in your group who do know some programming (or are willing to learn a bit) and to take them time to try out this environment for project management.

As mentioned in the previous two chapters, repositories that are tracked with git and shared through GitHub provide a number of tools that are useful in managing a project, both in terms of keeping track of what's been done in the project and also for planning what needs to be done next, breaking those goals into discrete tasks, assigning those tasks to team members, and maintaining a discussion as you tackle those tasks.

While git itself traditionally has been used with a command-line interface (think of the black and green computer screens shown when movies portray hackers), GitHub has wrapped git's functionality with an attractive and easy to understand graphical user interface. This is how you will interact with a project repository if you are online and logged into GitHub, rather than exploring it on your own computer (although there are also graphical user interfaces you can use to more easily explore git repositories locally, on your computer).

Key project management tools for GitHub that you can leverage, all covered in subsections below, are:

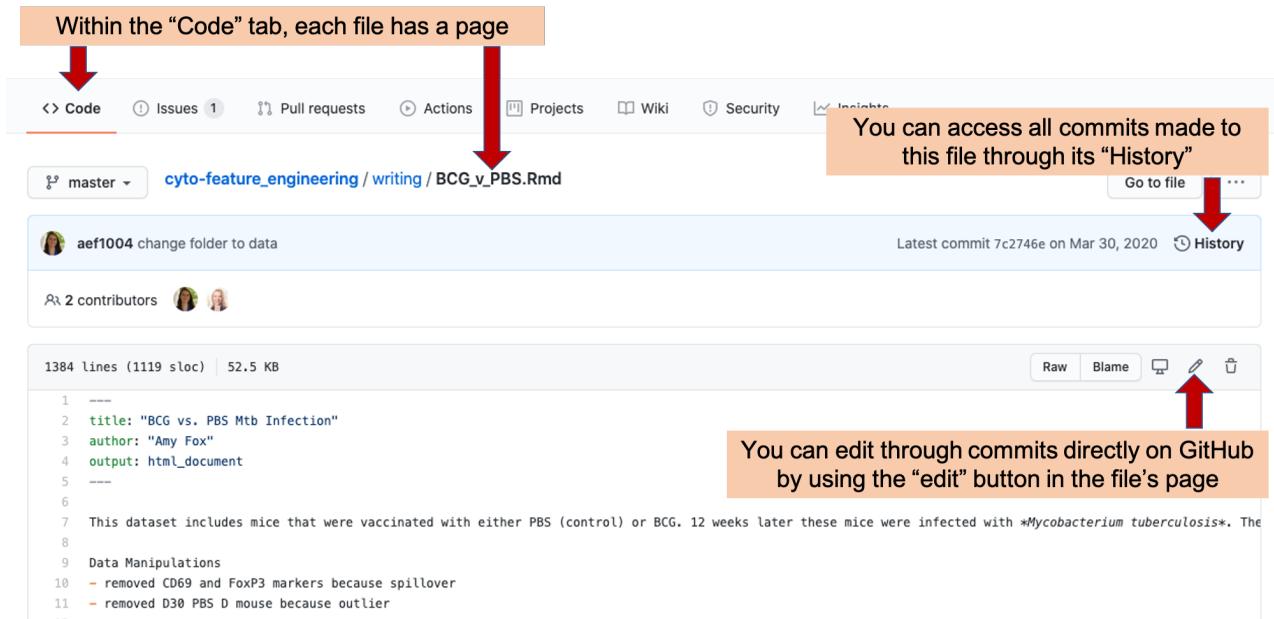
- Commits and commit history
- Issues
- Repository access and ownership
- Insights

Successfully using GitHub to help track and manage a research project does not require using all of these tools, and in fact you can go a long way by just starting with a subset. The first four covered (Commits, Issues, Commit history, and Repository access and ownership) would be a great set to try out in a first project.

Commits

Each time a team member makes a change to files in a GitHub repository, the change is recorded as a **commit**, and the team member must include a short **commit message** describing the change. Each file in the project will have its own page on GitHub (Figure 2.41), and you can see the history of changes to that files by clicking the "History" link on that page.

You can make changes to a file locally, on the repository copy on your own computer. For team members who are working a lot on coding, this will likely



be the primary method they use to make commits, as this allows you to test the code locally before you commit it.

However, it is also possible to make a commit directly on GitHub, and this may be a useful option for team members who are not coding and would like to make small changes to the writing files. On the file's page on GitHub, there is an "Edit" icon (Figure 2.41). By clicking on this, you will get to a page where you can directly edit the file (Figure 2.42). Once you have made your edits, you will need to commit them, and a short description of the commit is required. If you would like to include a longer explanation of your changes, there is space for that, as well, when you make the commit (Figure 2.42).

You can see the full history of changes that have been made to each file in the project (Figure 2.43). Each change is tracked through a commit, which includes markers of who made the change and a message describing the change. Further, this history page for the file provides a line-by-line history of when each line in the file was last changed and what that change is—this allows you to quickly pinpoint changes in a particular part of the file.

If you click on one of the commits listed on a file's History page (Figure 2.43, it will take you to a page providing information on the changes made with that commit (Figure 2.44). This page provides a line-by-line view of each change that was made to project files with that commit. This page includes the commit message for that commit. If the person committing the change included a longer description or commentary on the commit, this information will also be included on this page. Near the commit message are listings of which team member made the commit and when it was made. Within the body of the page,

Figure 2.41: Example of a file page within a GitHub repository. Each file in a repository has its own page. On this page, you can see the history of changes made to the file by looking at 'History'. You can also make a commit an edit directly in GitHub by clicking on the 'Edit' icon.

Within the "Code" tab, each file has a page

You can access all commits made to this file through its "History"

Latest commit 7c2746e on Mar 30, 2020 History

aef1004 change folder to data

2 contributors

1384 lines (1119 sloc) | 52.5 KB

Raw Blame

1 ---
2 title: "BCG vs. PBS Mtb Infection"
3 author: "Amy Fox"
4 output: html_document
5 ---
6
7 This dataset includes mice that were vaccinated with either PBS (control) or BCG. 12 weeks later these mice were infected with *Mycobacterium tuberculosis*. The
8
9 Data Manipulations
10 - removed CD69 and FoxP3 markers because spillover
11 - removed D30 PBS D mouse because outlier
12

You can edit through commits directly on GitHub by using the "edit" button in the file's page

Figure 2.42: Committing changes directly in GitHub. When you click on the 'Edit' button in a file's GitHub page (see previous figure), it will take you to a page where you can edit the file directly. You save the changes

The "History" page for each file shows the history of changes committed for the file

History for cyto-feature_engineering / writing / BCG_v_PBS.Rmd

Commits on Mar 30, 2020

change folder to data
aef1004 committed on Mar 30, 2020

Commits on Mar 26, 2020

clean up code by adding functions to beginning
aef1004 committed on Mar 26, 2020

Merge conflicts ...
aef1004 committed on Mar 26, 2020

Remove stray '
aef1004 committed on Mar 26, 2020

Commits on Mar 24, 2020

Figures for manuscript
aef1004 committed on Mar 24, 2020

Adjust FMO plots due to change in functions
aef1004 committed on Mar 24, 2020

Each commit is given a unique identifier that can be used to reference or revert it later

7c2746e

c34c8cd

d604391

eeb3766

8415dc4

Each commit includes a short commit message describing the change. You can click here to see the change made with this commit.

Figure 2.43: Commit history in GitHub. Each file in a repository has a 'History' page, where you can explore each change committed for the file. Each commit has a unique identifier and commit message describing the change. You can click on the entry for any of these commits to see the changes made to the file with the commit (see next figure).

you can see the changes made with the commit. Added lines will be highlighted in green while deleted lines are highlighted in red. If only part of a line was changed, it will be shown twice, once in red as its version before the commit, and once in green showing its version following the commit.

clean up code by adding functions to beginning

Commit message for this commit

aef1004 committed on Mar 26, 2020

You can determine who made the commit and when it was made

Showing 1 changed file with 20 additions and 172 deletions.

Unified Split

192 writing/BCG_v_PBS.Rmd

```

@@ -17,6 +17,7 @@ knitr::opts_chunk$set(echo = TRUE)
17 17 First, the packages must be loaded.
18 18 library(flowCore) - need this package if going to transform the da
19 19 ````r warning= FALSE, message = FALSE}
20 + library(tibble)
21 library(openCyto)
22 library(data.table)
23 library(ggcyto)

@@ -48,6 +49,10 @@ I have created a few functions to make it easier to analyze the data
49 49 - `viridis_colors`: vector of the colors used for most plots
49 50

```

Changes in this file for this commit are highlighted in green and red. Green highlighting shows lines that have been added with this commit, and red highlighting shows lines that have been deleted.

Unchanged parts of the file will be collapsed in this view. You can expand them if desired.

Issues

GitHub, as well as other version control platforms, includes functionality that will help your team collaborate on a project. A key tool is the “Issues” tracker. Each repository includes this type of tracker, and it can be easily used by all team members, whether they are comfortable coding or not.

Figure 2.45 gives an example of the Issues tracker page for the repository we are using as an example.

The main Issues tracker page provides clickable links to all open issues for the repository. You can open a new issue using the “New Issue” on this main page or on the specific page of any of the repository’s issues (see Figure 2.46 for an example of this button).

On the page for a specific issue (e.g., Figure 2.46), you can have a conversation with your team to determine how to resolve the issue. This conversation can include web links, figures, and “To-do” check boxes, to help you discuss and plan how to resolve the issue. Each issue is numbered, which allows you to track each individually as you work on the project.

Once you have resolved an issue, you can “Close” it. This moves the issue from the active list into a “Closed” list. Each closed issue still has its own page, where you can read through the conversation describing how it was resolved. If you need to, you can re-open a closed issue later, if you determine that it was

Figure 2.44: Commit history in GitHub. Each commit has its own page, where you can explore what changes were made with the commit, who made them, and when they were committed.

aef1004 / cyto-feature_engineering

Code Issues 1 Pull requests Actions Projects Wiki Security Insights

Label issues and pull requests for new contributors

GitHub will help potential first-time contributors discover issues labeled with good first issue Dismiss

Filters is:issue is:open Labels 9 Milestones 0 New issue

1 Open 3 Closed Standardize FMO figure with hexbins #1 opened on Jul 26, 2019 by aef1004 1

Click on “Issues” to get to this page

Click here to see issues that have been resolved

Open issues are listed here.

Figure 2.45: Issues tracker page for an example GitHub repository. Arrows highlight the tab to click to get to the Issues tracker page in a repository, as well as where to

Standardize FMO figure with hexbins #1

Open aef1004 opened this issue on Jul 26, 2019 · 1 comment

aef1004 commented on Jul 26, 2019

I'm trying to find a way in the FMO figures to adjust the hexbins. You can choose the hexbin number, but all of the data files have different numbers of cells, so the scale for the hexbin counts is different for each of the plots. Is there a way to set the number of datapoints that go into each bin (according to ggplot, this is a computed value). Ideally, these files wouldn't be plotted separately, it would be a facet_wrap, but for each of these plots, the x axis is different.

geanders commented on Aug 5, 2019

You can still use a facet wrap with different x axis ranges (it's the ranges, or the actual column you're plotting?) by specifying something like scales = 'free_x' in the facet. If it's a different column for each, there may still be a way to pull all the columns together with gathering / spreading, so you can use the facet_wrap call.

Write Preview

Leave a comment

Add to the conversation

Close the Issue when it's resolved

Close issue Comment

Remember, contributions to this repository should follow our GitHub Community Guidelines.

Assignees None yet

Labels None yet

Projects None yet

Milestone No milestone

Linked pull requests Successfully merging a pull request may close this issue. None yet

Notifications Customize Unsubscribe You're receiving notifications because you're watching this repository.

2 participants

Figure 2.46: Conversation about an Issue on Issues tracker page of an example GitHub repository. In this example, you can see how GitHub Issues trackers allow you to discuss how to resolve an issue across your team. From this page, you can read the current conversation about Issue #1 of the repository and add your own comments. Once the Issue is resolved, you can ‘Close’ the Issue, which moves it off the list of active issues, but allows you to still re-read the conversation and, if

not fully resolved.

Add in background strip color for faceted plots #3

Closed aef1004 opened this issue on Jul 30, 2019 · 3 comments

aef1004 commented on Jul 30, 2019

It would be nice to add in background strip color for the faceted plots (Population Percentages and CFU vs Population) to denote which cell lineage each population is.

aef1004 commented on Jul 30, 2019

The code:
`theme(strip.background = element_rect(color = "blue"))`
 Will give a blue strip outline, but still need to find a way to use multiple colors

aef1004 commented on Jul 30, 2019 · edited

I can get close with the code from <https://stackoverflow.com/questions/3455092/r-ggplot2-change-colour-of-font-and-background-in-facet-strip> but the colors aren't in the correct order.
 The code is located in the BCG_vs_PBS.Rmd

aef1004 added the **help wanted** label on Jul 30, 2019

aef1004 assigned geanders on Jul 30, 2019

Assignees
 geanders

Labels
help wanted

Projects

Issues can be tagged with one of more label

Issues can be assigned to specific team members

The Issues tracker page includes some more advanced functionality, as well (Figure 2.47). For example, you can “assign” an issue to one or more team members, indicating that they are responsible for resolving that issue. You can also tag each issue with one or more labels, allowing you to group issues into common categories. For example, you could tag all issues that cover questions about pre-processing the data using a “pre-processing” label, and all that are related to creating figures for the final manuscript with a “figures” label.

Repository access and ownership

Repositories include functionality for inviting team members, assigning roles, and otherwise managing access to the repository. First, a repository can be either public or private. For a public repository, anyone will be able to see the full contents of the repository through GitHub. You can also set a repository to be private. In this case, the repository can only be seen by those who have been invited to collaborate on the repository, and only when they are logged in to their GitHub accounts. The private / public status of a repository can be changed at any time, so if you want you can maintain a repository for a project as private until you publish the results, and then switch it to be public, to allow

Figure 2.47: Labeling and assigning Issues. The GitHub Issues tracker allows you to assign each issue to one or more team members, clarifying that they will take the lead in resolving the issue. It also allows you to tag each issue with one or more labels, so you can easily navigate to issues of a specific type or identify the category of a specific issue.

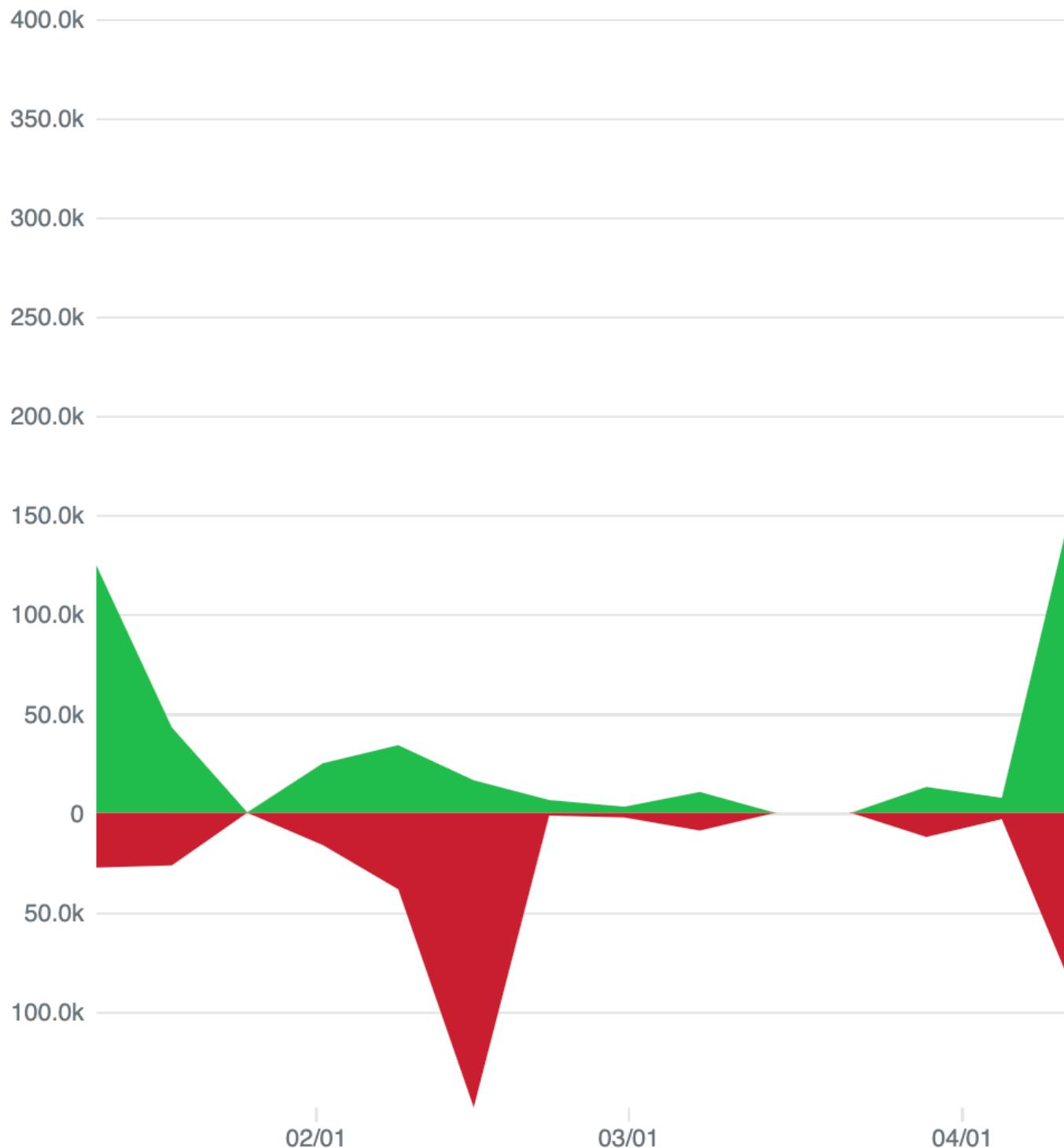
others to explore the code and data that are linked to your published results.

You can invite team members to collaborate on a repository, as long as they have GitHub accounts (these are free to sign up for). While public repositories can be seen by anyone, the only people who can add to or change the contents of the repository are people who have been invited to collaborate on the repository. The person who creates the repository can invite other collaborators through the “Settings” tab of the repository, which will have a “Manage access” function for the repositories maintainer. On this page, you can invite other collaborators by searching using their GitHub “handle” (the short name they chose to be identified by in GitHub). You can also change access rights, for example, allowing some team members to be able to make major changes to the repository—like deleting it—while others can make only smaller modifications.

Insights

Each GitHub repository also provides an “Insights” page, which lets you see who is contributing to the project and, as well when and how much they have contributed, as tracked by the commits they’ve made.

First, this page provides some repository-wide summaries, regardless of who was contributing. The figure below shows an example of the “Code frequency” graph, showing the number of additions and deletions to the code each week (here, “code” means any data in the tracked files, so it would include data recorded for the project or text written up for a project report or presentation [double-check that this is the case]).



During periods when the research team is collecting data, you would expect a lot more additions than deletions, and you could check this plot to ensure that the team is committing data soon after it's recorded (i.e., that there are lots of additions on weeks with major data collection for the experiment, not several weeks after). Periods with a lot of deletions, aren't bad, but instead likely indicate that a lot of work is being done in editing reports and manuscripts. For example, if a paper is being prepared for publication, you'd expect a lot of deletions as the team edits it to meet word count requirements.

The "Insights" page on a GitHub repository also lets you track the frequency of commits to the project, where each commit could be something small (like fixing a typo) or large (adding new data files for all data recorded for a time-point for the experiment). However, the frequency of these commits can help identify periods when the team is working on the project. For example, the commit history graph shown below is for the GitHub repository for a website for a spring semester course in 2020. It's clear to see the dates when the course was in session, as well as how the project required a lot of initial set up (shown by the number of commits early in the project period compared to later). You can even see spring break in mid-March (the week in the middle with no commits).



This window also allows you to track the number and timing of commits of each contributor to the project.

2.11.3 Leveraging git and GitHub as a scientist who programs

To be able to leverage GitHub to manage projects and share data, you will need to have at least one person in the research group who can set up the initial repository. GitHub repositories can be created very easily starting from an RStudio Project, a format for organizing project files that was described in module [x]. In this section, we'll give some advice on how you can use an RStudio Project to create and update a GitHub repository, and how this can allow separate team members to maintain identical copies of the RStudio Project on their own computers, while continually evolving files in the project as data pre-processing, data analysis, and manuscript preparation are done for the project. We will keep this advice limited, as there are excellent existing resources that provide more thorough instructions in this area, but we will introduce the methods and then point to more thorough resources for more detail.

- Interfacing with RStudio
- Initiating a repository and first commit
- Subsequent commits and commit messages
- Fixing merge conflicts when team members make concurrent changes
- Using branches and forks / pull requests to try out new things
- Using GitHub Actions for automation (e.g., automatic testing?)
- [Odds and ends—.DS_Store, \$Word_doc]
- More resources for learning to use git and GitHub

“Get a new repository in the directory you are working in via: `git init`. Ok, you now have a revision control system in place. You might not see it, because Git stores all its files in a directory names `.git`, where the dot means that all the usual utilities like `ls` will take it to be hidden. You can look for it via, e.g., `ls -a` or via a show hidden files option in your favorite file manager. ... Given that all the data about a repository is in the `.git\` subdirectory of your project directory, the analog to freeing a repository is simple:`rm -rf .git`.” (Klemens, 2014)

“Calling `git commit -a` writes a new commit object to the repository based on all the changes the index was able to track, and clears the index. Having saved your work, you can now continue to add more. Further—and this is the real, major benefit of revision control so far—you can delete whatever you want, confident that it can be recovered if you need it back. Don't clutter up the code with large blocks of commented-out obsolete routines—delete!” (Klemens, 2014)

“Having generated a commit object, your interactions with it will mostly consist of looking at its contents... The key metadata is the name of the object, which is assigned via an unpleasant but sensible naming convention: the SHA1 has, a 40-digit hexadecimal number that can be assigned to an object, in a manner that lets us assume that no two objects will have the same hash, and that the same object will have the same name in every copy of the repository. When you commit your

files, you'll see the first few digits of the hash on the screen... Fortunately, you need only as much of the hash as will uniquely identify your commit." (Klemens, 2014)

"Lists aren't external to the creative process, they are intrinsic to it. They are a natural part of any project of scale, whether we like it or not." (Savage, 2020)

"The maker in me knows that this is where lists really shine, that it is their capacity for simplifying the complex that sets them apart from all other planning tools. Not just at the beginning of a project, either, but at every step along the creative process, because no matter how exacting the list you make at the outset, there will always be things that you missed or, more frequently, that change. It's like trying to measure a coastline: it's fractal." (Savage, 2020)

"The value of a list is that it frees you up to think more creatively, by defining a project's scope and scale for you on the page, so your brain doesn't have to hold on to so much information. The beauty of the checkbox is that it does the same thing with regard to progress, allowing you to monitor the status of your project, without having to mentally keep track of everything." (Savage, 2020)

"The best part of making a list is, you guessed it, crossing things off. But when you physically cross them out, like with a pen, you can make them harder to read, which destroys their informational value beyond that single project and, to me at least, makes the whole thing feel incomplete. The checkbox allowed me to cross something off my list, to see clearly *that* I'd crossed it off, and at the same time retain all its information while not also adding to the cognitive load of interpreting the list." (Savage, 2020)

2.11.4 Applied exercise

3

Experimental Data Preprocessing

This section includes modules on:

- Module 3.1: Principles and benefits of scripted pre-processing of experimental data
- Module 3.2: Introduction to scripted data pre-processing in R
- Module 3.3: Simplify scripted pre-processing through R's 'tidyverse' tools
- Module 3.4: Complex data types in experimental data pre-processing
- Module 3.5: Complex data types in R and Bioconductor
- Module 3.6: Example: Converting from complex to 'tidy' data formats
- Module 3.7: Introduction to reproducible data pre-processing protocols
- Module 3.8: RMarkdown for creating reproducible data pre-processing protocols
- Module 3.9: Example: Creating a reproducible data pre-processing protocol

3.1 Principles and benefits of scripted pre-processing of experimental data

The experimental data collected for biomedical research often requires pre-processing before it can be analyzed (e.g., gating of flow cytometry data, feature finding / quantification for mass spectrometry data). Use of point-and-click software can limit the transparency and reproducibility of this analysis stage and is time-consuming for repeated tasks. We will explain how scripted pre-processing, especially using open source software, can improve transparency and reproducibility.

Objectives. After this module, the trainee will be able to:

- Define 'pre-processing' of experimental data
- Describe an open source code script and explain how it can increase reproducibility of data pre-processing

3.1.1 Example datasets

Simpler example dataset: Measuring bacterial growth rates

More complex example dataset: Characterizing lung cell populations

3.1.2 Why do data need preprocessing?

Use of scientific principles to make measurements

For a long time, scientists have leveraged existing scientific principles and knowledge to build equipment that can make measurements to build new scientific knowledge.

When preprocessing biomedical data, many of the steps are to leverage the scientific principles that underlie that measurement technique, so you can move from the measurements made by the equipment to a measurement that can help answer a biological question.

Simple examples: average cage weights, microscope scale, stethoscope, determining date since start of the experiment, determining period of exponential growth

A very simple example of this is a stethoscope. This is a very simple example because, in this case, the “data preprocessing” is done in the mind of the person using the stethoscope. A stethoscope is a simple piece of equipment that It’s used to When using a stethoscope to ask ..., the user must know how to use the sounds of ... to infer This is therefore a very simple example of inputting data (in this case, sounds) from equipment (in this case, the stethoscope) and processing it (in this case, within the mind of the user) to answer a biological question (in this case, ...). If the preprocessing step is not done, the input is useless—more simpler, if the stethoscope is used by someone who can’t interpret the sounds to infer ..., then it’s not help as a tool to determine

More complex examples: Flow cytometry is based on physics—using lasers to excite fluorescent probes that are manufactured to stick to certain surface proteins. To interpret the data, must link the light output at certain frequencies (?) to which probe you made, adjust for light overlapping from several types of probes at certain frequencies, determine how the intensity of light is linked to the number of fluorescent probes, use knowledge of the size, shape, and granularity of certain types of cells to interpret data on forward and side scatter of light, etc.

Data can have technical noise

Simple example: microscope measurements made at different scales (resolutions?)

More complex example: Reading depth of sequencing-based data

Data can have quality control issues

Examples of QC issues: missing data, outliers, errors in data entry, a whole sample that has problems

More complex example: Cells that are poor quality in single cell RNA-seq—membrane has broken? A lot of the RNA molecules have been lost and aren’t represented in the measurements?

Data can be high-dimensional

Many types of biomedical data now are “high-dimensional”—they include

measurements of a very large number of characteristics for each observation. For example, bulk transcriptomics data measure the expression of a large number of genes within each sample, metabolomics data measure the levels of a large number of metabolite features within each sample, proteomics measure the levels of a large number of peptides (?) within each sample, and so on.

With these high-dimensional data, often many of the measured characteristics will be strongly correlated. For example, in metabolomics data many of the metabolite features may be strongly correlated with each other, either because they originally were pieces of the same metabolite (the process of measuring the features, for example through mass spectrometry, breaks the chemicals up into smaller components [?]) or because they are part of a common metabolic pathway [?]. It can be hard to explore and understand a set of data that measures thousands of characteristics of each sample, but patterns and correlations in the data can be used to help you pull out and explore a few key dimensions of variability in the data. Dimension reduction techniques—principal components analysis (PCA) is one very popular one—is therefore often a key element of data preprocessing when working with high-dimensional biological data.

If you envision a dataset as a two-dimensional table, with the observations as the rows and the characteristics measured for each observation as the columns, then high-dimensional data are large in terms of the number of columns in the table. A growing number of biomedical data now also are large in the number of observations they capture—in the number of rows in the table of data, in other words. A longer-standing example of this is flow cytometry data, where the observations are individual cells. Current experiments often capture in the range of [x] cells for each sample in flow cytometry, measuring for each characteristics that can be used to determine cell size, granularity, and surface proteins, all with the aim of characterizing the cells by cell type. A more recent example is with single cell RNA-sequencing. Again, with this technique, observations are taken at the level of the cell, with on the order of [x] cells processed per sample.

As with high-dimensional data, again the sheer size of the data collected in this case can make it difficult to explore and generate knowledge from the data. For data with a large number of observations, clustering techniques can be helpful to explore large-scale patterns across the many observations. These are particularly helpful in data that both have a large number of observations and measure many characteristics of each observation, like single-cell transcriptomics.

3.1.3 Common steps in data preprocessing for biomedical research

Data input

Quality assessment / quality control

Data transformations

Scaling, normalization, log transformations, calculating time since start of

experiment, etc.

One critical process in this category is the process of *normalization*. ...

Extracting information based on scientific principals

Identifying data that are appropriate to answer the scientific question (e.g., period of exponential growth)

Taking data on light intensities at different frequencies and using it to estimate the presence or absence of certain surface proteins (and their amount) on each cell for flow cytometry.

Dimension reduction and clustering

3.1.4 Examples of biomedical data preprocessing

Simpler example: Measuring bacterial growth rates

More complex example: Characterizing lung cell populations

3.1.5 Tools for biomedical data preprocessing

GUI versus code script

Examples of coding languages for data preprocessing: Python, R, Julia, bash scripts

Examples of GUIs for data preprocessing: FlowJo, ...

Advantages / disadvantages of GUIs

Advantages / disadvantages of code scripts

Free and open-source versus proprietary

Examples of free and open-source tools: Python, R, Julia, bash scripts

Examples of proprietary software: FlowJo (?), ...

Advantages / disadvantages of proprietary

Advantages / disadvantages of free and open-source

3.1.6 Example dataset

In this module and several of the following modules, we'll use two example datasets that are based on real data collected from an immunology experiment. These data will help us to describe and motivate the steps of pre-processing data using code scripts, as well as explaining the use of tidyverse tools and the discussion of complex versus tidy data formats in R.

The first dataset was introduced in module [x]. ...

The second dataset have not yet been introduced. They come from an experiment to test a novel vaccine for COVID-19, and we'll be focusing on data collected during this experiment that measured single-cell transcriptomics—in other words, it characterizes the levels of messenger RNA with each of thousands of cells collected from animals in different experimental groups. In this section, we'll give you more details to help you understand this example

dataset, as well as instructions on how to download the data on your own computer, if you would like to follow along with examples.

The results of this experiment have been published [ref], and so you can read the full details in the published paper, but we'll provide an overview here. This experiment tested a potential vaccine called SolaVAX. There are numerous ways to make vaccines; this one uses an attenuated (in other words, weakened) version of the full virus, and it is novel in that it attenuates the virus using a light-activated ... (hence the "Sola" in "SolaVAX").

This experiment tested how well this vaccine worked, not only by itself, but also when it was given in conjunction with something called an adjuvant. In the context of vaccines, an adjuvant is a substance that is meant to shape the body's immune response as it "learns" from a vaccine. For example, an adjuvant can be something as simple as a substance that triggers a larger immune response than the vaccine by itself, to ensure that the immune system responds at a sufficient scale to the core components of the vaccine—that is, the components of the vaccine that the immune system needs to recognize in the future to mount a fast defense against that pathogen. This experiment tested two adjuvants in conjunction with the SolaVAX vaccine, both of which the researchers were hoping might help in switching which type of T helper immune cells would drive the later response to COVID after vaccination. Specifically, they hypothesized that the adjuvants would bring about a later response that was driven more by a type of T helper cell called Th2 rather than one called Th1. [Why this would be good.]

To run the experiment, the researchers used Golden Hamsters as a model animal. [Why?] They created four experimental groups: one control group, one group vaccinated with only SolaVAX, one vaccinated with SolaVAX plus an adjuvant called [x], and one vaccinated with SolaVAX plus an adjuvant called [y]. There were eight hamsters in each of these groups, and these were further divided into two groups of four, so that the vaccine could be tested using two routes of administration: [the two routes].

The hamsters were vaccinated, and then after a period of time, they were challenged with COVID. This allowed the researchers to see how successful each vaccination type was, in terms of how well the animals could limit the replication of COVID in their bodies, and also to explore how the animals' immune systems responded as they tried to limit COVID replication after exposure. Therefore, this experiment could help not only in seeing which vaccine strategies were successful, but also to explore how and why they did or didn't work, at the level of the immune response.

[x] days after the animals were exposed to COVID, they were sacrificed, and the researchers took samples from several areas to use to measure levels of COVID as well as the immune response to the challenge. In tissue samples from the lungs of the animals, they measured things like the [viral numbers?] to see the extent of COVID replication in that animal, and [histopathology], to see the extent of damage that the infection had done to the animal's lungs. If

the vaccine were successful, it would have spurred a fast and powerful immune response, which you'd see through lower [viral numbers] and less damage to the animal's lungs. They also used part of the lung sample to measure immune cell populations. These measures can help to determine things like whether the immune response was driven more by the innate immune response (which you'd expect in an unvaccinated animal) or the adaptive immune response (which you hope to see in a vaccinated animal, as a sign that the vaccine helped in allowing for the immediate immune response to be much more specific than that achieved by the innate immune system).

Out of the many types of data that were collected for this experiment, we'll focus on one type in our examples in this and following modules—data that were collected that measured the single-cell transcriptomics of samples collected from the lungs of each animal. While every cell in a body has the same genome, the cells differ in which of those genes they express at a given time. The expression of different genes in the genome can be measured based on the number of messenger RNAs that are in the cell from each gene. This is what single cell RNA sequencing aims to measure—the number of mRNAs from each of a large number of genes, measuring the number in each cell in the sample separately. Because different types of cells express different patterns and levels of genes, these data can be used to help sort the cells into different types. For example, the data can be used to help identify cells that are from the innate immune system versus those that are hallmarks of an adaptive immune response. The data can also help in categorizing cells within very specific cell categories—for example, identifying Th1 versus Th2 cells out of the group of helper T cells, which can help to address the hypothesis that these researchers had about how the adjuvants might work. Finally, the data can help to identify how certain types of cells work differently under different experimental conditions. For example, do adaptive immune cells tend to express different genes (or different levels of a gene) when the vaccination included an adjuvant versus when only the vaccine was given?

These single cell RNA sequencing data were shared by the researchers through one of NIH's [?] databases for biological data. To get a copy of these data, go to the study's page on the Gene Expression Omnibus (GEO) database: <https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE165190> (you can also search for the study on the database by its accession number, GSE165190). This page includes data from single cell RNA sequencing for twelve samples from this study, three from each of the experimental conditions. You can download data for all the samples by clicking the link on the page to download the file named "GSE165190_RAW.tar". This is a compressed file, so once you download it, you will need to uncompress it, which will give you a folder with a number of data files. On a Mac, you can decompress the file by double-clicking on the file named "GSE165190_RAW.tar" in the Finder program. On a Windows computer, [how to do it. From R?]

Among the data files, there are three files for each sample. One includes

“barcodes” in the file name, one includes “features” in the file name, and one includes “matrix” in the file name. For each sample, the “barcodes” file gives ..., the “features” file gives ..., and the “matrix” file gives

[More about working with these files?]

Collectively, these data provide a count that is related to the number of messenger RNA particles [?] from each of approximately [x] genes within each cell in the sample. These counts can be used to group the cells into groups of similar cell types, identify those cell types, and explore how the gene expression within a cell type varies across experimental conditions.

These data can therefore be used to answer interesting scientific questions. However, before they can, it is important to do some pre-processing on the raw data. This preprocessing serves several purposes. First, there are patterns in the data that can be introduced as a result of the data collection process. These need to be corrected or otherwise accounted for to move the data into a format where it can be meaningfully compared, for example across different cells in a sample or across different samples. Second, some of the pre-processing will help identify and resolve any issues related to quality control. For example, while the process of collecting these data will normally result in isolating single cells, in some cases the values for a cell might be for a poor-quality cell (for example, a dying cell) or might be a case where two cells were captured together [true for this platform?]. Pre-processing can help to identify and exclude these low-quality data points, so that the analysis can focus on the higher-quality data collected for the sample. Finally, some of the pre-processing will help us to prepare the data to be used in analysis algorithms. For example, since the data is high dimensional (that is, measurements are included for many genes [?]), we will often include a step of dimension reduction in our analysis, to help pull out key patterns in the data. Many of the dimension reduction algorithms will need to data to be scaled, so that the mean value of each measurement in a type of measurement has an average of zero and a standard deviation [? variance?] of one.

“As a proxy for studying the proteome, many researchers have turned to protein-encoding mRNA molecules (collectively termed the ‘transcriptome’), whose expression correlates well with cellular traits and changes in cellular state. Transcriptomics was initially conducted on ensembles of millions of cells, firstly with hybridization-based microarrays, and later with next-generation sequencing (NGS) techniques referred to as RNA-seq. RNA-seq on pooled cells has yielded a vast amount of information that continues to fuel discovery and innovation in biomedicine. ... Nevertheless, the averaging that occurs in pooling large numbers of cells does not allow detailed assessment of the fundamental biological unit—the cell—or the individual nuclei that package the genome. Since the first scRNA-seq study was published in 2009, there has been increasing interest in conducting such studies. Perhaps one of the most compelling reasons for doing so is that scRNA-seq can describe RNA molecules in individual cells with high resolution and on a genomic scale.” (Haque et al., 2017)

“scRNA-seq permits comparison of the transcriptomes of individual cells. There-

fore, a major use of scRNA-seq has been to assess transcriptional similarities and differences within a population of cells, with early reports revealing unappreciated levels of heterogeneity... Thus, heterogeneity analysis remains a core reason for embarking on scRNA-seq studies. Similarly, assessments of transcriptional differences between individual cells have been used to identify rare cell populations that would otherwise go undetected in analyses of pooled cells, for example malignant tumour cells within a seemingly homogeneous group. ... In addition to resolving cellular heterogeneity, scRNA-seq can also provide important information about fundamental characteristics of gene expression... Importantly, studying gene co-expression patterns at the single-cell level might allow identification of co-regulated gene modules and even inference of gene-regulatory networks that underlie functional heterogeneity and cell-type specification.” (Haque et al., 2017)

“Single-cell RNA sequencing (scRNA-seq) is a recent and powerful technology developed as an alternative to previously existing bulk RNA sequencing methods. Bulk sequencing methods analyzed the average genetic content for individual genes across a large population of input cells within a sample (e.g., a tissue), potentially obscuring transcriptional features and other differences among individual cells. Conversely, scRNA-seq is able to discern such heterogeneous properties within a sample and has great potential to reveal novel subpopulations and cell types.” (Lytal et al., 2020)

“Single-cell RNA sequencing is widely used for high-resolution gene expression studies investigating the behavior of individual cells.” (McCarthy et al., 2017)

“While scRNA-seq data can provide substantial biological insights, the complexity and noise of the data is also much greater than that of conventional bulk RNA-seq. Thus, rigorous analysis of scRNA-seq data requires careful quality control to remove low-quality cells and genes, as well as normalization to adjust for biases and batch effects in the expression data. Failure to carry out these procedures correctly is likely to compromise the validity of all downstream analyses.” (McCarthy et al., 2017)

“Conventional ‘bulk’ methods of RNA sequencing (RNA-seq) process hundreds of thousands of cells at a time and average out the differences. But no two cells are exactly alike, and scRNA-seq can reveal the subtle changes that make each one unique. It can even reveal entirely new cell types.” (Perkel, 2017)

“It’s much more difficult to manipulate individual cells than large populations, and because each cell yields only a tiny amount of RNA, there’s no room for error. Another problem is analyzing the enormous amounts of data that result—not least because the tools can be unintuitive.” (Perkel, 2017)

3.1.7 What is pre-processing?

“Assessing the structure of the data must also take account of the prior knowledge of the system in regard to such matters as the design of the experiment, the known sources of systematic variation (e.g., any blocking factors or known groupings of the experimental units) and so on.” (Chatfield, 1995)

“The three main types of problem data are errors, outliers, and missing observations. ... An error is an observation which is incorrect, perhaps because it has been copied or typed incorrectly at some stage. An outlier is a ‘wild’ or extreme

observation which does not appear to be consistent with the rest of the data. Outliers arise for a variety of reasons and can create severe problems. ... Errors and outliers are often confused. An error may or may not be an outlier, while an outlier may or may not be an error. ... An outlier may be caused by an error, but it is important to consider the alternative possibility that the observation is a genuine extreme result from the 'tail' of the distribution. This usually happens when the distribution is skewed and the outlier comes from the long 'tail'." (Chatfield, 1995)

"There are several types of error..., including the following: 1. A recording error arises, for example, when an instrument is misread. 2. A typing error arises when an observation is typed incorrectly. 3. A transcription error arises when an observation is copied incorrectly, and so it is advisable to keep the amount of copying to a minimum. ..." (Chatfield, 1995)

"Extreme observations which, while large, could still be correct, are more difficult to handle [than errors]. The tests for deciding whether an outlier is significant provide little information as to whether an observation is actually an error.

Rather external subject-matter considerations become paramount. It is essential to get advice from people in the field as to which suspect values are obviously silly or impossible, and which, while physically possible, are extremely unlikely and should be viewed with caution. ... It is sometimes sensible to remove an outlier, or treat it as a missing observation, but this outright rejection of an observation is rather drastic, particularly if there is evidence of a long tail in the distribution. Sometimes the outliers are the most interesting observations.

... An alternative approach is to use robust methods of examination, which automatically downweight extreme observations... My recommended procedure for dealing with outliers, when there is no evidence that they are errors, is to repeat the analysis with and without suspect values... If the conclusions are similar, then the suspect values do not matter. However, if the conclusions differ substantially, then the values do matter and additional effort should be expended to check them. If the matter still cannot be resolved, then it may be necessary to present two lots of results or at least point out that it may be unwise to make judgements from a set of data where the results depend crucially on just one or two observations (called influential observations)." (Chatfield, 1995)

"Missing observations arise for a variety of reasons. ... It is important to find out why an observation is missing. This is best done by asking 'people in the field'.

In particular, there is a world of difference between observations that are lost through random events, and situations where missing observations due to damage or loss are more likely to arise under certain conditions." (Chatfield, 1995)

"There are various reasons for making a transformation, which may also apply to deriving a new variable: 1. to get a more meaningful variable (the best reason!); 2. to stabilize variance; 3. to achieve normality (or at least symmetry); 4. to create additive effects (i.e. remove interaction effects); 5. to enable a linear model to be fitted." (Chatfield, 1995)

[For transformations] "Logarithms are often meaningful, particularly with economic data when proportional, rather than absolute, changes are of interest. Another application of the logarithmic transformation is ... to transform a severely skewed distribution to normality." (Chatfield, 1995)

Some data collected through laboratory experiments is very straightforward and requires little or no pre-processing before it's used in analysis. For example, if you are regularly weighing the animals in an experiment, then the data may require no pre-processing (in other words, you'll directly use the weight recorded from the scale) or very minimal pre-processing (for example, if you are keeping all animals for a treatment group in the same cage, you may weigh the cage as a whole, in which case you could divide that weight by the number of animals in the cage as a pre-processing step to estimate the average weight per animal).

Other data collected in the laboratory may require some pre-processing that takes a few more steps, but is still fairly straightforward. For example, if you plate bacteria from a sample at a variety of dilutions, you might count each plate and determine a measure of Colony Forming Units from the set of plates with different dilutions by deciding which dilution provides the clearest count and then back-calculating based on its dilution to get the total number of colony-forming units in the original sample. Pre-processing these data typically will also involve transforming data, to get them in a format that is easier to visualize or more appropriate for statistical analysis. For example, when bacterial loads are counted as colony forming units (CFUs), it is common to transform these values using a log-10 [?] transform before plotting the data or using them to test a hypothesis. [Why do a log-10 transform with CFUs?]

This step of pre-processing data can become much more complex with data that was collected using complex equipment, like a flow cytometer, mass spectrometer, or sequencer. In these cases, there are often steps required to extract from the machine's readings a biologically-relevant measurement. For example, the data output from a mass spectrometer must be processed to move from measurements of mass and retention time to estimates of concentrations of different molecules in the sample. If you want to compare across multiple samples, then the preprocessing will also involve steps to align the different samples (in terms of ...), as well as to standardize the measurements for each sample, to make the measurements from the different samples comparable.

"Scientists can now study all kinds of cell components and processes—from all the proteins in a cell (a discipline known as proteomics) to the amount of messenger RNA (the templates from which proteins are made) made from every gene ('transcriptomics') to the intermediate and final products of cell metabolism ('metabolomics')." (Barry and Cheung, 2009)

"To reap the full benefits of the omics revolution, we need information technology tools capable of making sense of the vast data sets generated by omics experiments. In fact, the development of such tools has become a discipline unto itself, called bioinformatics. And only with those tools can researchers hope to clear another obstacle to drug development: that posed by so-called emergent properties—behaviors of biological systems that cannot be predicted from the basic biochemical properties of their components." (Barry and Cheung, 2009)

For data collected from a flow cytometer, preprocessing may include steps to disentangle the fluorescence from different markers to ensure that the read for one marker isn't inflated by spillover fluorescence from a different marker.

...

For data from a sequencing machine [? sequencer?], the pre-processing will include a series of steps. We'll use an example of data from single-cell RNA sequencing. First, the sequencing process results in small fragments of complementary DNA [? cDNA], complementary to the original RNA in each cell in the sample, for which the sequence of nucleotide bases along each small fragment have been determined [?]. These raw read data can be used to answer more interesting scientific questions—for example, how do gene expression levels vary across cells in the samples, or what mixture of cell types comprise the sample—but first the raw data require multiple steps of pre-processing. For example, the small fragments of cDNA sequences must be aligned to a reference genome or transcriptome, so that their counts can be used to estimate the number of RNA molecules transcribed from different genes, and the counts for each fragment must also be linked with the cell it originally came from, so that these counts can be determined cell-by-cell.

"Although each [scRNA-seq] experiment is unique ..., most analysis pipelines follow the same steps to clean up and filter the sequencing data, work out which transcripts are expressed and correct for differences in amplification efficiency. Researchers then run one or more secondary analyses to detect subpopulations and other functions." (Perkel, 2017)

"In many cases... the tools used in bulk RNA-seq can be applied to scRNA-seq. But fundamental differences in the data mean that this is not always possible. For one thing, single-cell data are noisier... With so little RNA to work with, small changes in amplification and capture efficiencies can produce large differences from cell to cell and day to day and have nothing to do with biology. Researchers must therefore be vigilant for 'batch effects', in which seemingly identical cells prepared on different days differ for purely technical reasons, and for 'dropouts'—genes that are expressed in the cell but not picked up in the sequence data. Another challenge is the scale... A typical bulk RNA-seq experiment involves a handful of samples, but scRNA-seq studies can involve thousands. Tools that can handle a dozen samples often slow to a crawl when confronted with ten or a hundred times as many." (Perkel, 2017)

"Even the seemingly simple question of what constitutes a good cell preparation is complicated in the world of scRNA-seq. Lun's workflow assumes that most of the cells have approximately equivalent RNA abundances. But 'that assumption isn't necessarily true', he says. For instance, he says, naive T cells, which have never been activated by an antigen and are relatively quiescent, tend to have less messenger RNA than other immune cells and could end up being removed during analysis because a program thinks there is insufficient RNA for processing." (Perkel, 2017)

"Perhaps most significantly, researchers performing scRNA-seq tend to ask different questions from those analysing bulk RNA. Bulk analyses typically investigate

how gene expression differs between two or more treatment conditions. But researchers working with single cells are often aiming to identify new cell types or states or reconstruct developmental cellular pathways. ‘Because the aims are different, that necessarily requires a different set of tools to analyse the data,’ says Lun.” (Perkel, 2017)

“Most scRNA-seq tools exist as Unix programs or packages in the programming language R. But relatively few biologists are comfortable working in those environments... Even if they are, they may lack the time to download and configure everything to make such tools work. Some ready-to-use pipelines have been developed. And there are end-to-end graphical tools too, including the commercial SeqGeq package from FlowJo, as well as a pair of open-source web tools: Granatum ... and ASAP (the Automated Single-cell Analysis Pipeline)... ASAP and Granatum use a web browser to provide relatively simple, interactive workflows that allow researchers to explore their data graphically. Users upload their data and the software walks them through the steps one by one. For ASAP, that means taking data through preprocessing, visualization, clustering, and differential gene-expression analysis; Granatum allows pseudo-time analyses and the integration of protein-interaction data as well.” (Perkel, 2017)

“Appropriate methods are ‘very data-set dependent’... The methods and tuning parameters may need to be adjusted to account for variable such as sequencing length. But John Marioni at Cancer Research UK in Cambridge says it’s important not to put complete faith in the pipeline. ‘Just because the satellite navigation tells you to drive into the river, you don’t drive into the river,’ he says.” (Perkel, 2017)

“For beginners, caution is warranted. Bioinformatics tools can almost always yield an answer; the question is, does that answer mean anything? Dudoit’s advice is do some exploratory analysis, and verify that the assumptions underlying your chosen algorithms make sense.” (Perkel, 2017)

- cDNA libraries
- Mapping sequence fragments to a reference (BowTies, TopHats) / alignment of reads (GSNAP) / pseudoalignment (Salmon)
- Quantification / counting reads for each gene (htseq-count)

“Next, poly[T]-primed mRNA is converted to complementary DNA (cDNA) by a reverse transcriptase. Depending on the scRNA-seq protocol, the reverse-transcription primers will also have other nucleotide sequences added to them, such as adaptor sequences for detection on NGS platforms, unique molecular identifiers (UMIs) to mark unequivocally a single mRNA molecule, as well as sequences to preserve information on cellular origin.” (Haque et al., 2017)

“The minute amounts of cDNA are then amplified either by PCR or, in some instances, *in vitro* transcription followed by another round of reverse transcriptions—some protocols opt for nucleotide barcode-tagging at this stage to preserve information on cellular origin.” (Haque et al., 2017)

“Then, amplified and tagged cDNA from every cell is pooled and sequenced by NGS, using library preparation techniques, sequencing platforms and genomic-alignment tools similar to those used for bulk samples.” (Haque et al., 2017)

"More recently, droplet-based platforms ... have become commercially available... Droplet-based instruments can encapsulate thousands of single cells in individual partitions, each containing all the necessary reagents for cell lysis, reverse transcription and molecular tagging, thus eliminating the need for single-cell isolation through flow-cytometric sorting or micro-dissection." (Haque et al., 2017)

"An important initial step in scRNA-seq data processing is to quantify the expression level of genomic features such as transcripts or genes from the raw sequencing data." (McCarthy et al., 2017)

"Read counts can be obtained from conventional quantification methods such as HTSeq and featureCounts ... another option is to use computationally efficient pseudoalignment methods such as kallisto and Salmon. This is especially appealing for large scRNA-seq datasets containing hundreds to tens of thousands of cells. To this end, scater also provides wrapper functions for kallisto and Salmon so that fast quantification of transcript-level expression can be managed completely within an R programming environment." (McCarthy et al., 2017)

"A common subsequent step for these methods is to collapse transcript-level expression to gene-level expression. Exploiting the biomaRt R/Bioconductor package, scater provides a convenient function for using Ensembl annotations to obtain gene-level expression values and gene or transcript annotations."

(McCarthy et al., 2017)

These early steps of pre-processing create a count matrix for each sample, with a row for each feature of the data (for example, a gene), a column for each cell in the sample, and matrix cell values that estimate the number of messenger RNA molecules from each gene in each cell. However, more pre-processing is required before these count matrices can be used to answer scientific questions.

First, it is necessary to perform quality control on the count matrices, to identify and remove problematic data points. For example, one method for single cell sequencing involves isolating each cell within a droplet. While most of the droplets that are processed will indeed have one and only one cell, a few droplets will end up having two cells or no cells, and these must be identified and removed during a quality control step of pre-processing. For all methods of single cell sequencing, there will also be some poor-quality cells. For example, if a cell's membrane has broken [?] during the process of measurement, then much of its messenger RNA may have leaked out [?] of the cell. These cells can be identified based on a combination of characteristics: they typically will have a combination of low total counts of RNA molecules, low numbers of total genes expressed, and a high percentage of the RNA molecules that are captured will be from mitochondrial RNA [because?]. Using these characteristics, many of these cells can be identified and removed from the count matrix in a quality control step of pre-processing. Other, more sophisticated approaches are also available, based on machine learning approaches to identify these low quality cells (Ilicic et al., 2016).

"Another important challenge is that existing available scRNA-seq protocols often result in the captured cells (whether chambers in microfluidic systems, microwell plates, or droplets) being stressed, broken, or killed. Moreover, some capture sites can be empty and some may contain multiple cells. We refer to all such cells as 'low quality'. These cells can lead to misinterpretation of the data and therefore need to be excluded." (Ilicic et al., 2016)

"We first tested whether each type of low quality cell (broken, empty, multiple) has higher average gene expression in specific functional categories (Gene Ontology terms) compared to high quality cells. Second, we calculated whether gene expression in these functional categories is noisier for low versus high quality cells... Our results suggest that there are indeed several top-level biological processes and components that are significantly different. Specifically, genes related to Cytoplasm ..., Metabolism ..., Mitochondrion ..., Membrane ..., and a few other categories ... are strongly downregulated ... in broken cells. ... Furthermore, broken cells have transcriptome-wide increased noise levels compared to high quality cells. Interestingly, wells containing multiple cells (multiples) show similar expression and noise patterns to broken cells ... This suggests that multiple cells contain a mixture of broken and high quality cells." (Ilicic et al., 2016)

"Genes related to mtDNA were upregulated in low quality cells... This suggested that these cells are broken and thus of low quality." (Ilicic et al., 2016)

"We have shown that there are biological and technical features within the sequencing data that allow automatic identification of the majority of low quality cells." (Ilicic et al., 2016)

"At least a subset of the features considered are cell type specific. ... Somewhat surprisingly, the levels of Membrane, Ribosomes, Metabolism, Apoptosis, and Housekeeping genes are highly cell type specific. In contrast, Mitochondrial (localized or encoded) and Cytoplasmic genes are more generic features. ... Interestingly, only moderately and strongly expressed genes seem to be similar between the datasets [of different cell types]. Genes that are very strong or lowly expressed are highly cell type specific." (Ilicic et al., 2016)

"There is an extensive literature on the relationship between mtDNA, mitochondrially localized proteins, and cell death ... However, upregulation of RNA levels of mtDNA in broken cells suggests losses in cytoplasmic content. In a situation where cell membrane is broken, cytoplasmic RNA will be lost, but RNAs enclosed in the mitochondria will be retained, thus explaining our observation." (Ilicic et al., 2016)

"Overall, our analysis suggests that empty wells can be remarkably clearly distinct from the remainder, while broken cells and multiples are distinct in most but not all of the categories." (Ilicic et al., 2016)

"We designed three features based on the assumption that broken cells contain a lower and multiple cells a higher number of transcripts compared to a typical high quality single cell. For the first feature we calculated the number of highly expressed and highly variable genes. For the second feature we calculated the variance across genes. Lastly, we hypothesized that the number of genes expressed at a particular level would differ between cells. Thus we binned normalized read counts into intervals (very low to very high) and counted the number of genes

in each interval. ... Overall, our results show that technical features [like the number of detected genes and the percent of mapped reads] can help distinguish low and high quality cells." (Ilicic et al., 2016)

"Before further analyses, scRNA-seq data typically require a number of bio-informatic QC checks, where poor-quality data from single cells (arising as a result of many possible reasons, including poor cell viability at the time of lysis, poor mRNA recovery and low efficiency of cDNA production) can be justifiably excluded from subsequent analysis. Currently, there is no consensus on exact filtering strategies, but most widely used criteria include relative library size, number of detected genes and fraction of reads mapped to mitochondria-encoded genes or synthetic spike-in RNAs. ... Other considerations are whether single cells have actually been isolated or whether indeed two or more cells have been mistakenly assessed in a particular sample." (Haque et al., 2017)

"Barcode: Tagging single cells or sequencing libraries with unique oligonucleotide sequences (that is, 'barcodes'), allowing sample multiplexing. Sequencing reads corresponding to each sample are subsequently deconvoluted using barcode sequence information." (Haque et al., 2017)

"Dropout: An event in which a transcript is not detected in the sequencing data owing to a failure to capture or amplify it." (Haque et al., 2017)

"Sequencing depth: A measure of the sequencing capacity spent on a single sample, reported for example as the number of raw reads per cell." (Haque et al., 2017)

"Spike-in: A molecule or set of molecules introduced to the sample in order to calibrate measurements and account for technical variation" (Haque et al., 2017)

"Transcriptional bursting: A phenomenon, also known as 'transcriptional pulsing', of relatively short transcriptionally active periods being followed by longer silent periods, resulting in temporal fluctuation of transcript levels." (Haque et al., 2017)

"Unique molecular identifier: A variation of barcoding, in which the RNA molecules to be amplified are tagged with random n-mer oligonucleotides. The number of distinct tags is designed to significantly exceed the number of copies of each transcript species to be amplified, resulting in uniquely tagged molecules, and allowing control for amplification biases." (Haque et al., 2017)

"A high percentage of counts mapping to spike-ins typically indicates that a small amount of RNA was captured for the cell, suggesting protocol failure or death of the cell in processing that renders it unsuitable for downstream analyses." (McCarthy et al., 2017)

"For each gene, QC metrics such as the average expression level and the proportion of cells in which the gene is expressed are computed. This can be used to identify low-abundance genes or genes with high dropout rates that should be filtered out prior to downstream analyses." (McCarthy et al., 2017)

"Typical scRNA-seq datasets will show a broadly sigmoidal relationship between average expression level and frequency of expression across cells. This is consistent with expected behaviour where genes with greater average expression are more readily captured during library preparation and are detected at greater frequency." (McCarthy et al., 2017)

"It is common to see ERCC spike-ins (if used), mitochondrial and ribosomal genes among the highly expressed genes, while datasets consisting of healthy cells will also show high levels of constitutively expressed genes like ACTB." (McCarthy et al., 2017)

Second, the count matrix will require normalization before the data in it can be used to test scientific hypotheses. To be able to count the small amount of mRNA within single cells, the single-cell RNA-sequencing approach relies on amplification (that is, once cDNA have been created from each RNA molecule, the cDNA are replicated many times through ... [?]). ...

- size factor (approach earlier for bulk RNA-seq)
- technical size factor with spike-ins (Brennecke et al., 2013), plus a biological size factor [?]

"Progress in gene expression analysis using minute amounts of starting material has made single-cell transcriptomics accessible." (Brennecke et al., 2013)

"The low amount of RNA present in a single cell represents the main challenge in single-cell RNA-seq experiments. ... We demonstrated the relationship between technical noise and the amount of starting material with a dilution series, using technical replicates of decreasing amounts of total RNA taken from the same pool of total RNA. ... For 5,000 pg of input material, the noise pattern was comparable to that of technical replicates from bulk RNA-seq experiments, in which the spread can be accounted for by the Poisson distribution. However, the number of genes affected by high levels of technical noise increased notably at lower amounts of starting material (for example, a transcript could have 100–1,000 read counts in one technical replicate but 0 counts in another). ... Nevertheless, genes with a high read showed very good agreement between replicates even for the 10-pg data point—meaning that low-read count genes show strong noise and high-read count genes show weak noise; what changes across the differing amounts of starting material is the read-count range in which noise strength transitions from weak to strong." (Brennecke et al., 2013)

"The crux of the issue is how to examine tens of thousands of genes possibly being expressed in one cell, and provide a meaningful comparison to another cell expressing the same large number of genes, but in a very different matter." (Haque et al., 2017)

"A major issue common to all protocols is how to account for technical variation in the scRNA-seq process from cell to cell. Some protocols 'spike-in' a commercially available, well-characterized mix of polyadenylated mRNA species ... The data from spike-ins can be used for assessing the level of technical variability and for identifying genes with a hgh degree of biological variability. In addition, spike-ins are valuable when computationally correcting for batch effects between samples. However, the use of spike-ins is itself not without problems... An increasingly popular method involves the use of UMIs, which effectively tags every mRNA species recovered from one cell with a unique barcode. Theoretically, this allows estimation of absolute molecule counts, although the UMIs can be subject to saturation at high expression levels. Nevertheless, the use of UMIs can significantly reduce amplification bias and therefore improve precision." (Haque et al., 2017)

"It is at this point that the 'zero or dropout problem' of scRNA-seq should be raised. The efficiency with which poly-adenylated mRNA are captured, converted into cDNA and amplified is currently unclear, and, depending on the study, can range between 10 and 40%. This means that, even if a gene is being expressed, perhaps at a low level, there is a certain probability that it will not be detected by current scRNA-seq methods. A partial solution to this issue is to increase read depth. However, beyond a certain point, this strategy leads to diminishing returns as the fraction of PCR duplicates increases with deeper sequencing. Current data suggest that single-cell libraries from all common protocols are very close to saturation when sequenced to a depth of 1,000,000 reads, and a large majority of genes are detected already with 500,000 reads, although the exact relationships are protocol specific." (Haque et al., 2017)

"While scRNA-seq workflows are conceptually closely related to population-level transcriptomics protocols, data from scRNA-seq experiments have several features that require specific bioinformatics approaches. First, even with the most sensitive platforms, the data are relatively sparse owing to a high frequency of dropout events (lack of detection of specific transcripts). Moreover, owing to the digital nature of gene expression at the single-cell level, and the related phenomenon of transcriptional bursting (in which pulses of transcriptional activity are followed by inactive refractory periods), transcript levels are subject to temporal fluctuation, further contributing to the high frequency of zero observations in scRNA-seq data. Therefore, the numbers of expressed genes detected from single cells are typically lower compared with population-level ensemble measurements. Because of this imperfect coverage, the commonly used unit of normalized transcript levels used for bulk RNA-seq, expressed as 'reads per kilobase per million' (RPKM), is biased on a single-cell level, and instead the related unit 'transcripts per million (TPM)' should be used for scRNA-seq. Second, scRNA-seq data, in general, are much more variable than bulk data. scRNA-seq data typically include a higher level of technical noise (such as dropout events), but also reveal much of the biological variability that is missed by RNA-seq on pooled cells. Biological variability that is present on many levels, and which of these are considered as nuisance variation, depends on the underlying biological question being asked. For example, at the gene level, transcriptional bursting causes variation in transcript quantities, whereas at the global level, the physical size of individual cells can vary substantially, affecting absolute transcript numbers and reflected in the number of detected genes per cell. Cell-size variation can also be closely related to proliferative status and cell-cycle phase. ... Finally, distributions of transcript quantities are often more complex in single-cell datasets than in bulk RNA-seq. In general, single-cell expression measurements follow a negative binomial distribution, and, in heterogeneous populations, multimodal distributions are also observed. As a consequence, statistical tests that assume normally distributed data (used for example for detecting differentially expressed genes) are likely to perform suboptimally on scRNA-seq data." (Haque et al., 2017)

"Methods to quantify mRNA abundance introduce systematic sources of variation that can obscure signals of interest. Consequently, an essential first step in most mRNA-expression analyses is normalization, whereby systemic variations are adjusted to make expression counts comparable across genes and / or samples. Within-sample normalization methods adjust for gene-specific features, such as GC content and gene length, to facilitate comparisons of a gene's expression within

an individual sample; whereas between-sample normalization methods adjust for sample-specific features, such as sequencing depth, to allow for comparisons of a gene's expression across samples." (Bacher et al., 2017)

"A number of methods are available for between-sample normalization in bulk RNA-seq experiments. Most of these methods calculate global scale factors (one factor is applied to each sample, and this same factor is applied to all genes in the sample) to adjust for sequencing depth. These methods demonstrate excellent performance in bulk RNA-seq, but they are compromised in the single-cell setting because of an abundance of zero-expression values and increased technical variability." (Bacher et al., 2017)

"scRNA-seq data show systematic variation in the relationship between transcript-specific expression and sequencing depth (which we refer to as the count-depth relationship) that is not accommodated by a single scale factor common to all genes in a cell. Global scale factors adjust for a count-depth relationship that is assumed to be common across genes. When this relationship is not common across genes, normalization via global scale factors leads to overcorrection for weakly and moderately expressed genes and, in some cases, undernormalization of highly expressed genes. To address this, SCnorm uses quantile regression to estimate the dependence of transcript expression on sequencing depth for every gene. Genes with similar dependence are then grouped, and a second quantile regression is used to estimate scale factors within each group. Within-group adjustment for sequencing depth is then performed using the estimated scale factors to provide normalized estimates of expression." (Bacher et al., 2017)

"To further evaluate SCnorm, we conducted an experiment that, similar to the simulations, sequenced cells at very different depths. ... Prior to normalization, counts in the second group will appear four times higher on average given the increased sequencing depth. However, if normalization for depth is effective, fold-change estimates should be near one... since the cells between the two groups are identical." (Bacher et al., 2017)

"Single-cell RNA-seq technology offers an unprecedented opportunity to address important biological questions, but accurate data normalization is required to ensure that results are meaningful." (Bacher et al., 2017)

"However, individual cells have extremely tiny amounts of input material available, typically on the scale of picograms. The small scale of scRNA-seq input material means that some level of inaccuracy is inevitable even with the most precise instruments, resulting in an additional layer of stochasticity known as technical noise. During the sequencing process, reverse transcription is necessary to convert RNA to cDNA for use in amplification, but this introduces positional bias regardless of where the polymerization begins. The following amplification process counteracts low input material, though this in turn leads to additional bias as some genes may experience preferential amplification, leading to uneven representation in the data. The amplification process also runs the risk of producing dropout events, in which either genes known to be present in a sample are completely absent from the observed gene counts or genes are observed with lower values than their true expression. These events frequently lead to excessive zeros, and in many cases, more than half of all counts. Traditional bulk approaches do not naturally accommodate these differences, and therefore lose their effectiveness when applied to scRNA-seq." (Lytal et al., 2020)

"Normalization is critical to the development of analysis techniques on scRNA-seq and to counteract technical noise or bias. Before observed data can be used to identify differentially expressed genes or potential subpopulations, it must undergo these corrections, for what is observed is seldom exactly what is present within the data set." (Lytal et al., 2020)

"Some scRNA-seq studies involve the use of spike-in molecules for the purpose of normalization. The spike-in RNA set is artificially added to each cell's lysate in the same volume under the assumption that spike-ins and endogenous transcripts will experience similar variation among the cells during the capture process. Since spike-in gene concentrations are known, normalization methods model existing technical variation by utilizing the difference between these known values and the values observed after processing." (Lytal et al., 2020) There are other steps of pre-processing that can be considered for single cell RNA-seq data, but that might not be applied in every case. For example, a technique called scaling could be used to help constrain the influence of highly-expressed genes when conducted further analysis steps, like grouping the cells into clusters based on cell type (with cell type in this case being determined based on similar patterns of gene expression). [More on scaling]

"Scaling normalization is typically required in RNA-seq data analysis to remove biases caused by differences in sequencing depth, capture efficiency or composition effects between samples." (McCarthy et al., 2017)

"In the analysis and interpretation of single-cell RNA-seq (scRNA-seq) data, effective pre-processing and normalization represent key challenges. While unsupervised analysis of single-cell data has transformative potential to uncover heterogeneous cell types and states, cell-to-cell variation in technical factors can also confound these results. In particular, the observed sequencing depth (number of genes or molecules detected per cell) can vary significantly between cells, with variation in molecular counts potentially spanning an order of magnitude, even within the same cell type. Importantly, while the now widespread use of unique molecular identifiers (UMI) in scRNA-seq removes technical variation associated with PCR, differences in cell lysis, reverse transcription efficiency, and stochastic molecular sampling during sequencing also contribute significantly, necessitating technical correction. These same challenges apply to bulk RNA-seq workflows, but are exacerbated due to the extreme comparative sparsity of scRNA-seq data." (Hafemeister and Satija, 2019)

"The primary goal of single-cell normalization is to remove the influence of technical effects in the underlying molecular counts, while preserving true biological variation." (Hafemeister and Satija, 2019)

"In general, the normalized expression level of a gene should not be correlated with the total sequencing depth of a cell. Downstream analytical tasks (dimension reduction, differential expression) should also not be influenced by variation in sequencing depth." (Hafemeister and Satija, 2019)

"The variance of a normalized gene (across cells) should primarily reflect biological heterogeneity, independent of gene abundance or sequencing depth. For example, genes with high variance after normalization should be differentially expressed across cell types, while housekeeping genes should exhibit low variance. Additionally, the variance of a gene should be similar when considering either

deeply sequenced cells, or shallowly sequenced cells.” (Hafemeister and Satija, 2019)

“Sequencing depth variation across single cells represents a substantial technical confounder in the analysis and interpretation of scRNA-seq data.” (Hafemeister and Satija, 2019)

There are also cases where pre-processing steps could be used to remove patterns from technical noise or even from biological patterns that are unrelated to the scientific question of interest. In terms of technical noise, for example, there are cases where pre-processing steps could be used to help remove variation that’s introduced by running the experimental samples in different batches. In terms of biological patterns, one pattern that may be desirable to remove through pre-processing is gene expression related to a cell’s phase in the cell cycle. [More on this.]

“[A] promising application [of scRNA-seq] is the study of transcriptional heterogeneity within supposedly homogeneous cell types, a phenomenon of physiological importance, which can now be studied in a transcriptome-wide manner in single cells. In such analyses, which should be distinguished from the more common two-group comparison setting, it is necessary to account for strong technical noise. Technical noise is unavoidable owing to the low amount of starting material, and it must be quantified in order to avoid mistaking it for genuine differences in biological expression levels.” (Brennecke et al., 2013)

“After scaling normalization, further correction is typically required to ameliorate or remove batch effects. For example, in the case study dataset, cells from two patients were each processed on two C1 machines. Although C1 machine is not one of the most important explanatory variables on a per-gene level, this factor is correlated with the first principal component of the log-expression data. This effect cannot be removed by scaling normalization methods, which target cell-specific biases and are not sufficient for removing large-scale batch effects that vary on a gene-by-gene basis. ... For the dataset here, we fit a linear model to the scran normalized log-expression values with the C1 machine as an explanatory factor. (We also use the log-total counts from the endogenous genes, percentage of counts from the top 100 most highly-expressed genes and percentage of counts from control genes as additional covariates to control for these other unwanted technical effects.) We then use the residuals from the fitted model for further analysis. This approach successfully removes the C1 machine effect as a major source of variation between cells.” (McCarthy et al., 2017)

“We emphasize that it is generally preferable to incorporate batch effects or latent variables into statistical models used for inference. When this is not possible (e.g., for visualization), directly regressing out these uninteresting factors is required to obtain ‘corrected’ expression values for further analysis. Furthermore, a general risk of removing latent factors is that interesting biological variation may be removed along with the presumed unwanted variation. Users should therefore apply such methods with appropriate caution, particularly when an analysis aims to discover biological conditions, such as new cell types.” (McCarthy et al., 2017)

“Once identified, important covariates and latent variables can be flagged for inclusion in downstream statistical models or their effects regressed out of normalized expression values.” (McCarthy et al., 2017)

[Identifying highly-variable genes in terms of expression]

“All genes will display some biological variability in expression from cell to cell, but a high level of variance (exceeding the specified threshold) will indicate genes important in explaining heterogeneity within the cell population under study.”
 (Brennecke et al., 2013)

[Principal component analysis]

“Most approaches seek to reduce these ‘multi-dimensional data’, with each dimension being the expression of one gene, into a very small number of dimensions that can be more easily visualised and interpreted. Principal component analysis (PCA) is a mathematical algorithm that reduces the dimensionality of data, and is a basic and very useful tool for examining heterogeneity in scRNA-seq data.” (Haque et al., 2017)

“Dimensionality reduction and visualization are, in many cases, followed by clustering of cells into subpopulations that represent biologically meaningful trends in the data, such as functional similarity or developmental relationship. Owing to the high dimensionality of scRNA-seq data, clustering often requires special consideration... Likewise, a variety of methods exist for identifying differentially expressed genes across cell populations.” (Haque et al., 2017)

“One common type of single-cell analysis, for instance, is dimension reduction. This process simplifies data sets to facilitate the identification of similar cells. ... scRNA-seq data represent each cell as ‘a list of 20,000 gene-expression values.’ Dimensionality-reduction algorithms such as principal components analysis (PCA) and t-distributed stochastic neighbour embedding (t-SNE) effectively project those shapes into two or three dimensions, making clusters of similar cells apparent.” (Perkel, 2017)

“Multivariate techniques may be used to reduce the dimensionality... It is potentially dangerous to allow the number of variables to exceed the number of observations because of non-uniqueness and singularity problems. Put simply, the unwary analyst may try to estimate more parameters than there are observations.” (Chatfield, 1995)

“It is also possible to take a wider view of IDA by allowing the use, when necessary, of a group of more complicated, multivariate techniques which are data-analytic in character. The adjective ‘data-analytic’ could reasonably be applied to any statistical technique, but I follow modern usage in applying it to techniques which do not depend on a formal probability model except perhaps in a secondary way. Their role is to explore multivariate data, to provide information-rich summaries, to generate hypotheses (rather than test them) and to help generally in the search for structure, both between variables and between individuals. In particular, they can be helpful in reducing dimensionality and in providing two-dimensional plots of the data. The techniques include principal component analysis, multi-dimensional scaling and other forms of cluster analysis... They are generally much more sophisticated than earlier data descriptive techniques and should not be undertaken lightly. However, they are occasionally very fruitful.”
 (Chatfield, 1995)

“Principal component analysis rotates the p observed values to p new orthogonal variables, called principal components, which are linear combinations of the

original variables and are chosen in turn to explain as much of the variation as possible. It is sometimes possible to confine attention to the first two or three components, which reduces the effective dimensionality of the problem. In particular, a scatter diagram of the first two components is often helpful in detecting clusters of individuals or outliers.” (Chatfield, 1995)

“Cluster analysis aims to partition a group of individuals into groups or clusters which are in some sense ‘close together’. There is a wide variety of possible procedures. In my experience the clusters obtained depend to a large extent on the methods used (except where the clusters are really clear-cut) and users are now aware of the drawbacks and the precautions which need to be taken to avoid irrelevant or misleading results.” (Chatfield, 1995)

“Another popular application is pseudo-time analysis. Trapnell developed the first such tool, called Monocle, in 2014. The software uses machine learning to infer from an scRNA-seq experiment the sequence of gene-expression changes that accompany cellular differentiation, much like inferring the path of a foot race by photographing runners from the air, Trapnell says.” (Perkel, 2017)

“Other tools address subpopulation detection … and spatial positioning, which uses data on the distribution of gene expression in tissues to determine where in a tissue each transcriptome arose.” (Perkel, 2017)

- Polymerase chain reaction (PCR): **Amplifies** DNA
- Microarrays: allowed the analysis of multiple genes at once, previously one gene at a time
- Genomic profiles can differ from transcriptional profiles

Proteomics:

“Protein molecules, rather than DNA or RNA, carry out most cellular functions. The direct measurement of protein levels and activity within the cell is therefore the best determinant of overall cell function.” (Lakhani and Ashworth, 2001)

“To sequence a protein ten years ago, a substantial amount had to be purified and a technique known as Edman degradation had to be used. … During the 1990s, mass spectrometry (MS), in which biomolecules are ionized and their mass is measured by following their specific trajectories in a vacuum system, replaced Edman degradation, because it is more sensitive and can fragment the peptides in seconds instead of hours or days. Furthermore, MS does not require proteins or peptides to be purified to homogeneity and has no problem identifying blocked or otherwise modified proteins. In the last few years, further breathtaking technological advances have established MS not only as the definitive tool to study the primary structure of proteins, but also as a central technology for the field of proteomics.” (Steen and Mann, 2004)

“After protein purification, the first step is to convert proteins to a set of peptides using a sequence-specific protease. Even though mass spectrometers can measure the mass of intact proteins, there are a number of reasons why peptides, and not proteins, are analysed in proteomics. Proteins can be difficult to handle and might not all be soluble under the same conditions… In addition, the sensitivity of the mass spectrometer for proteins is much lower than for peptides, and

the protein may be processed and modified such that the combinatorial effect makes determining the masses of the numerous resulting isoforms impossible. ... Most importantly, if the purpose is to identify the protein, sequence information is needed and the mass spectrometer is most efficient at obtaining sequence information from peptides that are up to ~20 residues long, rather than whole proteins. Nevertheless, with very specialized equipment, it is becoming possible to derive partial sequence information from intact proteins, which can then be used for identification purposes or the analysis of protein modifications in an approach called 'top-down' protein sequencing." (Steen and Mann, 2004)

"Digesting the protein into a set of peptides also means that the physico-chemical properties of the protein, such as solubility and 'stickiness', become irrelevant. As long as the protein generates a sequence of peptides, at least some of them can be sequenced by the mass spectrometer, even if the protein itself would have been unstable or insoluble under the conditions used." (Steen and Mann, 2004)

"The peptides that are generated by protein digestion are not introduced to the mass spectrometer all at once. Instead, they are injected onto a microscale capillary high-performance liquid chromatography (HPLC) column that is directly coupled to, or is 'on-line' with, the mass spectrometer. The peptides are eluted from these columns using a solvent gradient of increasing organic content, so that the peptide species elute in order of their hydrophobicity." (Steen and Mann, 2004)

"When a peptide species arrives at the end of the column, it flows through a needle. At the needle tip, the liquid is vaporized and the peptide is subsequently ionized by the action of a strong electric potential. This process is called 'electrospray ionization'." (Steen and Mann, 2004)

"Electrosprayed peptide ions enter the mass spectrometer through a small hole or a transfer capillary. Once inside the vacuum system, they are guided and manipulated by electric fields. There are diverse types of mass spectrometer, which differ in how they determine the mass-to-charge (m/z) ratios of the peptides. Three main types of mass spectrometers are used in proteomics: quadrupole mass spectrometers, time of flight (TOF) mass spectrometers, and quadrupole 'ion traps'. ... Each of these instruments generates a mass spectrum, which is a recording of the signal intensity of the ion at each value of the m/z scale (which has units of daltons (Da) per charge)."

"At the beginning of the 1990s, researchers realized that the peptide-sequencing problem could be converted to a database-matching problem, which would be much simpler to solve. The reason database searching is easier than de novo sequencing is that only an infinitesimal fraction of the possible peptide amino-acid sequences actually occur in nature. A peptide-fragmentation spectrum might therefore not contain sufficient information to unambiguously derive the complete amino-acid sequence, but it might still have sufficient information to match it uniquely to a peptide sequence in the database on the basis of the observed and expected fragment ions. There are several different algorithms that are used to search sequence databases with tandem MS-spectra data, and they have names such as PeptideSearch, Sequest, Mascot, Sonar ms/ms, and ProteinProspector." (Steen and Mann, 2004)

"As a result of rapidly improving technology, the identification of hundreds of proteins is not unusual, even in a single project, and determining the reliability

of these protein hits is especially challenging. This is partly because even small error rates for each of the corresponding proteins can quickly add up when many thousands of peptides are being identified." (Steen and Mann, 2004)

"After all the peptides have been identified, they have to be grouped into protein identifications. Usually, the peptide scores are added up to yield protein scores in a straightforward manner. However, the confidence in the accuracy of a particular peptide identification increases if other peptides identify the same protein and decreases if no other peptides do so." (Steen and Mann, 2004)

"Often, we are interested not only in the identity of a peptide, but also in its quantity. Unfortunately, the intensity of the signal of a peptide ion does not directly indicate the amount of protein present. For example, when digesting a protein, the peptides that are produced should all be equimolar and might be expected to give peaks of equal height in the mass spectrum. However, accessibility to the protease, the solubility of the peptide and the ionization efficiency of the peptide combine to make these signals orders of magnitude different. Fortunately, these factors are reproducible, so the peak height of the same peptides can be a good indicator of the relative amount of the related protein from one experiment to the next." (Steen and Mann, 2004)

"Experiments that are aimed at determining protein expression in whole-cell lysates or tissues (expression proteomics) have been less successful so far. However, intense research efforts are underway at present, because such a strategy would enable the detection/identification of disease-related biomarkers. Such a measurement is essentially the equivalent of a microarray experiment, with the difference being that protein, instead of mRNA, levels are compared. MS experiments that compare protein-expression levels are much more laborious than microarray experiments, but are attractive because proteins are the active agents of the cell, whereas the mRNA population is often a poor indicator of protein levels. However, it is still difficult to identify and quantify all the low-abundance proteins, especially in the presence of highly abundant proteins. Furthermore, as in microarray experiments, the results are 'noisy', because of the extremely large amounts of data, and it can be difficult to distil functional and mechanistic hypotheses from such global experiments." (Steen and Mann, 2004)

"m/z ratio (mass-to-charge ratio): Mass spectrometry does not measure the mass of molecules, but instead measures their m/z value. Electrospray ionization, in particular, generates ions with multiple charges, such that the observed m/z value has to be multiplied by z and corrected for the number of attached protons (which equals z) to calculate the molecular weight of a particular peptide." (Steen and Mann, 2004) (Steen and Mann, 2004)

Flow cytometry:

"In multicellular organisms, cells carry out a diverse array of complex, specialized functions. This specialization occurs mostly through the expression of cell type-specific genes and proteins that generate the appropriate structures and molecular networks. A central challenge in the biomedical sciences, however, has been to identify the distinct lineages and phenotypes of the specialized cells in organ systems, and track their molecular evolution during differentiation." (Benoist and Hacohen, 2011)

"Since the 1970s, fluorescence-based flow cytometry has been the leading technique for studying and sorting cell populations. It involves passing cells through flow chambers at high rates ($> 20,000$ cells/s) and using lasers to excite fluorescent tags ('fluorochromes') that are usually attached to antibodies; different antibodies are tagged with different colors, enabling researchers to quantify molecules that define cell subtypes or reflect activation of specific pathways. Progress in instrument design, multi-laser combinations, and novel fluorochromes has led to experimental configurations that simultaneously measure up to 15 markers. This has enabled very detailed description of cell subtypes, perhaps most extensively in the immune system, where the Immunological Genome Project is profiling >200 distinct cell types. Fluorescence cytometry seems to have reached a technical plateau, however: In practice, researchers typically measure only 6 to 10 cell markers because they are limited by the spectral overlap between fluorochromes." (Benoist and Hacohen, 2011)

"In mass cytometry, fluorochrome tags are replaced by a series of rare earth elements (e.g., lanthanides), which are attached to antibodies through metal-chelator coupling reagents. Cells are labeled by incubation in a cocktail of tagged antibodies; as the cells flow through the instrument, they are vaporized at 5500 K, and the released tags are identified and quantified by time-of-flight mass spectrometry (MS). ... The beauty of this approach stems from three factors: the precision of MS detection, which eliminates overlap between tags (a dream for any investigator who has battled this problem, known as fluorescence compensation); the number of detectable markers (34 here, but easily more); and the absence of background noise (because rare earth elements are essentially absent from biological materials, there is no equivalent of 'autofluorescence')." (Benoist and Hacohen, 2011)

"Because the software tools commonly used for flow cytometry data would be woefully inadequate for analyzing dozens of dimensions [as you have for mass cytometry data], Bendall et al. used software that clusters cell populations into 'minimum-spanning trees' that reproduce known hematopoietic differentiation, but with much finer granularity. As a result, cells that once would have been grouped into one population are now divided into many more; for example, naive CD4+ T lymphocytes, a priori considered a homogeneous population, are now split into more than 10 subsets." (Benoist and Hacohen, 2011)

"The secret is that, in flow cytometry, every cell acts as an independent 'test tube'." (Benoist and Hacohen, 2011)

"Over the past 25 years there have been major technological advances in the ways that CD4+ T cells are enumerated, with flow cytometry now generally regarded as the predicate technique. During this period, flow cytometry has progressed from large, expensive and complex fluorescence-activated cell sorting (FACS) machines that require highly specialized operating personnel, to smaller, more affordable bench-top instruments that can be used by individuals with minimal training. This shift was made possible in part by including multicolour analysis coupled with simplified gating technologies." (Barnett et al., 2008)

"In 1954, Wallace Coulter developed an instrument that could measure cell size and count the absolute number of cells, and thus the discipline of flow cytometry was born. Further developments enabled the production of instruments

that could measure cell size and nucleic acid content using a two-dimensional approach that compared light scatter and light absorption. These instruments were cumbersome and required specialist operators, but immunologists began to use them to investigate the functions of immune cells. ... By the mid-1980s, bench-top flow cytometers were available and as the technology advanced the instruments became progressively smaller. Coupled with the availability of monoclonal antibodies, the increasing number of available fluorochromes (compounds that emit light at a greater wavelength than the light source they are excited with) and computer improvements, flow cytometers are now accessible for almost every clinical laboratory." (Barnett et al., 2008)

"Flow cytometry enables the examination of microscopic particles (such as cells) that are suspended in a stream of fluid which is termed sheath fluid. This fluid is focused hydrodynamically such that the cells flow in single file through a flow cell in which a beam of light (usually a laser) is focused. As the cells pass through the laser beam they scatter the light so that forward scatter (FSC) and side scatter (SSC) light is captured; this enables the size and granularity of the cells to be determined. In addition, if a cell is labelled with antibodies that carry a fluorochrome, as the cell passes in front of the laser beam the fluorochrome emits light at a wavelength that is higher than the single wavelength light source and which can be detected by fluorescence detectors. The flow cytometers that are in clinical use can analyse at least four fluorochromes simultaneously, in addition to the FSC and SSC. This is known as multiparametric analysis. The information that is generated is computer analysed so that specific analysis regions (known as gates) can be created, which allows the user to build up a profile of the size, granularity and antigen profile of the target cell population." (Barnett et al., 2008)

"The fundamental basis of CD4 counting and identification relies on the specific use of electronic gating approaches. One of the earliest and most popular approaches that was used from the mid-1980s to early-1990s relied on using light scatter and anti-CD45 and anti-CD14 monoclonal antibodies. This gating approach was termed the two-colour approach." (Barnett et al., 2008)

"In the early 1990s researchers realized that because of these logistic problems [with the two-colour approach], new technologies would need to be developed to reduce processing time while providing accurate CD4 counts. Two new gating strategies were developed almost simultaneously, one termed CD45 gating and the other T gating, both of which helped to more accurately identify the lymphocyte analysis regions." (Barnett et al., 2008)

"Flow cytometry is a powerful technology that is capable of quantifying individual functions independently and simultaneously on a single-cell basis (and, in some cases, under conditions that can preserve the viability of the cell for future use). Quantification of multiple functions of T cells requires at least six-colour technology: three colors to identify the T-cell lineage (typically, these label CD3, CD4, and CD8), one color for viability and/or as a dump channel to remove unwanted cell populations, and two or more colors devoted to the effector function of interest. Furthermore, delineation of memory T-cell subsets requires an additional two or more colors to distinguish differentiation stage." (Seder et al., 2008)

"Of note, the number of possible functionally unique subsets increases geometrically with the number of measurements made. By applying Boolean gating, all

possible combinations of T cells that produce the cytokines measured can be determined. ... In general, not all possible combinations are represented in any given antigen response; rather, the overall response is typically limited to a subset of the individual combinations of the functions. Based on the frequency of each distinct functional subset, a pie chart can be assembled to show the composition of a total cytokine response." (Seder et al., 2008)

"Boolean gating: A flow cytometric data analysis technique in which cells are divided into all possible combinations of the functions measured by using the Boolean operators 'and' and 'not' on analysis gates applied to those measurements. These populations of cells can be expressed as absolute frequencies or as a fraction of the total response." (Seder et al., 2008)

"T cells have an essential role in protection against a variety of infections. Indeed, the development of successful vaccines against HIV, malaria or tuberculosis will require the generation of potent and durable T-cell responses. ... As T cells are functionally heterogeneous and mediate their effects through a variety of mechanisms, a major hurdle in quantifying protective T-cell responses has been the technical limitations in assessing the complexity of such responses. Methods to define the full characteristics of T cells are crucial for developing preventative and therapeutic vaccines for infections and cancer." (Seder et al., 2008)

"A typical 'gating tree' hierarchically identifies unique functional subsets of CD4+ and CD8+ T cells based on the fluorescence staining of live cells and on the expression of other relevant markers, such as CD3, CD4, CD8, CD45RA, CC-chemokine receptor (CCR7), interferon gamma (INF γ), interleukin-2 (IL-2) and tumour-necrosis factor (TNF), following antigenic stimulation. The viability marker (ViViD) excludes dead cells that can often show non-specific binding to other reagents. Live, CD3+ T cells are separated into CD4+ and CD8+ T-cell lineages; within each lineage, memory T cells are selected based on the expression of (for example) CCR7 and CD45RA. Other combinations of markers can similarly exclude naive T cells from functional analysis." (Seder et al., 2008)

"Flow cytometry has increasingly become a tool of choice for the analysis of cellular phenotype and function in the immune system. It is arguably the most powerful technology available for probing human immune phenotypes, because it measures multiple parameters on many individual cells. Flow cytometry thus allows for the characterization of many subsets of cells, including rare subsets, in a complex mixture such as blood. And because of the wide array of antibody reagents and protocols available, flow cytometry can be used to assess not only the expression of cell-surface proteins, but also that of intracellular phosphoproteins and cytokines, as well as other functional readouts." (Maecker et al., 2012)

"This diversity of reagents and applications has led to a large variety of ways in which flow cytometry is being used to monitor the immune systems of humans and model organisms (mostly mice). These uses include the identification of antigen-specific T cells using tetramers or intracellular cytokine staining; the measurement of T cell proliferation using dyes such as 5,6-CFSE; and the use of immunophenotyping assays to identify lymphocyte, monocyte, and/or granulocyte subsets." (Maecker et al., 2012)

"Animal studies tend to be relatively small and self-contained, as the mice used are inbred (genetically identical) and their environment and treatments are

carefully controlled. Human studies, by contrast, are often larger, to account for genetic and environmental variables. They may require longitudinal assays over a considerable length of time, owing to the difficulty of recruiting available subjects and/or the need to follow those subjects over time. There may also be a need to aggregate data from multiple sites, or even across multiple studies, to achieve sufficient sample sizes for statistical analysis or to compare different treatments. In these situations, the standardization of reagents and protocols becomes crucial.” (Maecker et al., 2012)

“Immune phenotypes: Measurable aspects of the immune system, such as the proportions of various cell subsets or measures of cellular immune function.” (Maecker et al., 2012)

“The steps in a typical flow cytometry experiment … present several variables that need to be controlled for effective standardization. These variables include the general areas of reagents, sample handling, instrument setup and data analysis… The effects of changes in these variables are largely known. For example, the stabilization and control of staining reagents through the use of pre-configured lyophilized-reagent plates, and centralized data analysis, have both been shown to decrease variability in a multi-site study. However, the widespread adoption of standards for controlling such variables has not taken place. This is in contrast to other technologies, such as gene expression microarrays, which have achieved a reasonable degree of standardization in recent years. … Of course, microarray data are less complex than flow cytometry data, which are based on many hierarchical gates. Still, a reasonable degree of standardization of flow cytometry assays should be possible to achieve.” (Maecker et al., 2012)

“Gates: Sequential filters that are applied to a set of flow cytometry data to focus the analysis on particular cell subsets of interest.” (Maecker et al., 2012)

“The study of cells moving in suspension through an image plane—flow cytometry—is a potent tool for immunology research and immune monitoring. Its main advantages are that it makes multiparameter measurements and this it does so on a single-cell basis. The result is that this technique can dissect the phenotypes and functions of cell subsets in ways that are not possible using bulk assays, such as Western blots, microarrays or enzyme-linked immunosorbent assays (ELISAs). Nowhere has this proven more useful than in a mixed suspension of immune cells, such as the blood. Newer instrumentation allows for the analysis of eight or more parameters at flow rates of thousands of cells per second; the resulting rich data sets are unparalleled for the knowledge of immune function that they have contributed.” (Maecker et al., 2012)

“In this context [Human Immunology Project], an ‘immune phenotype’ would encompass not only the proportions of major immune cell subsets, but also, for example, their activation state, responsiveness to stimuli, and T cell receptor or B cell receptor repertoires, among other parameters.” (Maecker et al., 2012)

“A typical flow cytometry experiment. Sample preparation from blood often involves Ficoll gradient separation of mononuclear cells, and sometimes cryopreservation, before staining with fluorescent antibody conjugates. Each of these steps can introduce variability in the assay results. Instrument setup involves setting voltage gains for the photomultiplier tubes (PMTs) so as to achieve optimal sensitivity. To the extent that this is not standardized, it becomes a source of

variability as well. Data acquisition involves passing the stained cells through a laser beam and recording the fluorescence emission from all of the bound antibody conjugates. Here, the main variable is the type of instrument, including the lasers and optical filters used. This is followed by data analysis, in which cell populations of interest are defined and reported on, which is another significant source of variation.” (Maecker et al., 2012)

“The definition of particular subsets of immune cells using cell-surface markers continues to evolve, particularly for cell types that are the subject of intense current research. These include regulatory T (TReg) cells, interleukin-17 (IL-17)-secreting T helper (TH17) cells, dendritic cells (DCs) and natural killer (NK) cells. However, even well-characterized subsets, such as naive and memory T cells, are defined differently in various studies. For example, the classical T cell subsets of naive, central memory, effector memory, and effector T cells were first defined on the basis of their expression of CC-chemokine receptor 7 (CCR7; also known as CD197) and CD45RA ... Other investigators have since used different markers, such as CD62L or CD27 in place of CCR7, and CD45RO in place of CD45RA. Although these different combinations of markers generally define similar cell subsets, they introduce an unknown amount of heterogeneity that makes comparisons between studies difficult. It is reasonable to think that the discovery of new markers and new cell subsets will continue for quite some time in the future. However, we propose that there is sufficient maturity in the field of cellular immunology to reach consensus working definitions for most of the commonly studied subsets of immune cells. ... In other words, it should be possible to define a stable set of markers that delineate the major classes of B, T and NK cells, monocytes and DCs.” (Maecker et al., 2012)

(Maecker et al., 2012) recommend, as ways to minimize the effects of technical variation in flow cytometry data analysis, “Central analysis by one or a few coordinated experts” and “Use of automated gating algorithms”

“Activation markers for these cell types are of course of equal interest to the differentiation markers discussed above, as the activated subsets can provide clues to an individual’s response to infection, vaccination, cancer, or autoimmune disease. For this purpose, one can assess intracellular markers of recent cell division (such as Ki67) or surface markers of activation that have varying kinetics of expression (for example, CD69 is a transient marker, whereas HLA-DR, CD38 and CD71 are expressed for longer periods).” (Maecker et al., 2012)

“How important is it to define the actual antibody cocktails used for immunophenotyping? As described above, the choice of markers is clearly important, but different antibody clones and fluorochromes can also greatly influence results.” (Maecker et al., 2012)

“In addition to optimizing certain settings, such as laser time delays, instrument setup is mostly concerned with the setting of the voltage gains applied to each fluorescence detector, which influences the sensitivity of the detector to dim versus bright signals. This used to be entirely a matter of user preference, but some guiding principles have now emerged based on measuring the variance of a population of dim particles over a series of voltage gains.” (Maecker et al., 2012)

“Data analysis is one of the largest variables in flow cytometry, as shown by studies analyzing data at multiple sites versus centrally. It is also one of the easiest

variables to address, as re-analysis of existing data is always possible, whereas choices made about sample handling and instrument setup are irrevocable. Central analysis is simple in concept, but it can be overwhelming in terms of the demands placed on one or a few coordinated personnel who must review all of the data to achieve consistent gating. Fortunately, automated gating algorithms have proliferated and improved, such that they now compete favorably with expert manual gating.” (Maecker et al., 2012)

“With the major exception of CD4+ T cell counting for immune monitoring in patients with HIV and the immunophenotyping of leukemias and lymphomas, flow cytometry is mostly carried out in a research setting. Particular assays may be internally validated for a given clinical study, but comprehensive standardization is rare to non-existent. However, there is a clear and growing interest in the standardization of human immunophenotyping. This is evidenced by standardization efforts, such as those of the European ENTIRE and United States CTOC networks. The NIH have also invested in a series of awards focused on human immunophenotyping, and both FOCIS and the NIH are working to create standard immunophenotyping panels.” (Maecker et al., 2012)

Metabolomics:

Bulk transcriptomics:

“Much of biochemistry and molecular biology rests on the assumption that the behavior of cells in a population (in a culture, in an organ) can be reliably approximated by the population average that results when cells are lysed and their molecules analyzed as a pool. Increasingly, however, investigators realize that stochastic fluctuations in gene or protein expression, between cells of an otherwise identical group, can lead to major differences in their behavior. This ‘noise’ in gene expression can have profound consequences for cell differentiation, the response to apoptosis-inducing stimuli, or T lymphocyte triggering at the initiation of immune responses.” (Benoist and Hacohen, 2011)

“The challenges of deriving statistically robust conclusions from high-throughput technologies such as microarrays are well known. Yet the exploration of similar challenges for data from sequencers, which can generate orders of magnitude more data than an array, is largely at the beginning stages [as of 2011].” (Mak, 2011)

“Statistics becomes particularly important when sequencing technology is being used quantitatively. In terms of traditional applications of just sequencing a genome, I think that there is an abundance of methods already available that addressed many of the major questions. But as soon as any sort of quantification comes into play, such as using read depth to quantify copy number variation or RNA transcript abundances (RNA-Seq), then statistical modeling—and particularly normalization—becomes much more important.” (Mak, 2011)

“One of the main roles that statistics play in science is explaining variation—variation of observed data. That variation can actually be true signal that you’re interested in, but there can also be variations due to noise or confounding signal. So I think of statistical modeling as the process of explaining variation in the data according to concrete variables that have been measured.” (Mak, 2011)

"This process of accounting for, and possibly removing, sources of variation that are not of biological interest is called normalization. There are two distinct approaches to normalization. One of them I would call 'unsupervised' in that it does not take into account the study design. These are the most popular methods because they require the least amount of statistical modeling and knowledge of statistics. The other approach, which is one I strongly favor, is what I would call 'supervised normalization'. This approach directly takes into account the study design. I find this appealing because one is trying to parse sources of variation, then it seems all sources of variation should be considered. If I perform an experiment with 20 microarrays, say 10 treated and 10 control, then I want to utilize this information when separating and removing technical sources of variation. Another component of normalization, which is gaining popularity, is normalizing by principal components. Again, I think this should be done in the context of the study design, which was the goal of a recent method I worked on called 'surrogate variable analysis.'" (Mak, 2011)

"Let's say you are doing an RNA-Seq experiment. The sequencer may produce a different number of total reads from lane to lane, and that is more likely driven by technology, not biology. And so, normalization would be about accounting for those differences. Unsupervised normalization might involve simply just dividing by the number of lanes and taking each gene as a percentage of the reads in the lane. Why is that less than ideal? Suppose you have two batches of data, one flow cell that was done in November and another flow cell that was done in December. If you're actually accounting for this variation in the total number of reads per lane, my inclination would be to take into account the fact that these two flow cells were processed in different months. And it can be more complicated than that, too. Maybe you've taken clinical samples and there were some differences in the clinical conditions under which they were taken. In supervised normalization, you would take that information into account. For example, the adjustment made to the raw reads may be based on a model that includes the total number of reads per lane as well as the information about the study design, such as batch and biological variables." (Mak, 2011)

"The biggest, the easiest way [for a biologist doing RNA-Seq to tell that better normalization of the data is needed]—the way that I discovered the importance of normalization in the microarray context—is the lack of reproducibility across different studies. You can have three studies that are all designed to study the same thing, and you just see basically no reproducibility, in terms of differentially expressed genes. And every time I encountered that, it could always be traced back to normalization. So, I'd say that the biggest sign and the biggest reason why you want to use normalization is to have a clear signal that's reproducible." (Mak, 2011)

"Typically, RNA-seq data is analysed by laboriously typing commands into a Unix operating system. Data files are passed from one software package to the next, with each tool tackling one step in the process: genome alignment, quality control, variant calling, and so on. The process is complicated. But for bulk RNA-seq, at least, a consensus has emerged as to which algorithms work best for each step and how they should be run. As a result, 'pipelines' now exist that are, if not exactly plug-and-play, at least tractable for non-experts. To analyze differences in gene expression, says Aaron Lun, a computational biologist at Cancer Research UK in Cambridge, bulk RNA-seq is 'pretty much a solved problem'." (Perkel, 2017)

Reproducibility of pre-processing:

"Today, one often hears that life sciences are faced with the 'big data problem.' However, data are just a small facet of a much bigger challenge. The true difficulty is that most biomedical researchers have no capacity to carry out analyses of modern data sets using appropriate tools and computational infrastructure in a way that can be fully understood and reused by others. This struggle began with the introduction of microarray technology, which, for the first time, introduced life sciences to truly large amounts of data and the need for quantitative training. What is new, however, is that next-generation sequencing (NGS) has made this problem vastly more challenging. Today's sequencing-based experiments generate substantially more data and are more broadly applicable than microarray technology, allowing for various novel functional assays, including quantification of protein-DNA binding or histone modifications (using chromatin immunoprecipitation followed by high-throughput sequencing (ChIP-seq)), transcript levels (using RNA sequencing (RNA-seq)), spatial interactions (using Hi-C) and others. These individual applications can be combined into larger studies, such as the recently published genomic profiling of a human individual whose genome was sequenced and gene expression tracked over an extended period in a series of RNA-seq experiments. As a result, meaningful interpretation of sequencing data has become particularly important. Yet such interpretation relies heavily on complex computation—a new and unfamiliar domain to many of our biomedical colleagues—which, unlike data generation, is not universally accessible to everyone." (Nekrutenko and Taylor, 2012)

"The entire field of NGS analysis is in constant flux, and there is little agreement on what is considered to be the 'best practice'. In this situation, it is especially important to be able to reuse and to adopt various analytical approaches reported in the literature. Unfortunately, this is often difficult owing to the lack of necessary details. Let us look at the first and most straightforward of the analyses [for genome variant discovery]: read mapping. To repeat a mapping experiment, it is necessary to have access to the primary data and to know the software and its version, parameter settings and name of the reference genome build. From the 19 papers listed in Box 1 ..., only six satisfy all of these criteria. To investigate this further, we surveyed 50 papers (Box 2) that use the Burrows-Wheeler Aligner (BWA) for mapping (the BWA is one of the most popular mappers for Illumina data). More than half do not provide primary data and list neither the version nor the parameters used and neither do they list the exact version of the genomic reference sequence. If these numbers are representative, then most results reported in today's publications using NGS data cannot be accurately verified, reproduced, adopted, or used to educate others, creating an alarming reproducibility crisis." (Nekrutenko and Taylor, 2012)

"We note that this discussion thus far has dealt only with technical reproducibility challenges or with the ability to repeat published analyses using the original data to verify the results. Most biomedical researchers are much more acquainted with biological reproducibility, in which conceptual results are verified by an alternative analysis of different samples. However, we argue that the computational nature of modern biology blurs the distinction between technical and biological reproducibility." (Nekrutenko and Taylor, 2012)

"The 1000 Genomes Project is an international effort aimed at uncovering human genetic variation with the ultimate goal of understanding the complex relation-

ship between genotype and phenotype. ... The crucial importance of the pilot project [phase of this project] was the establishment of a series of best practices that are used for the analysis of the data from the main phase and are broadly applicable for analysis of resequencing data in general. ... Despite the fact that analytical procedures developed by the project are well documented and rely on freely accessible open-source software tools, the community still uses a mix of heterogeneous approaches for polymorphism detection: we surveyed 299 articles published in 2011 that explicitly cite the 1000 Genomes pilot publication. Nineteen of these were in fact resequencing studies with an experimental design similar to that of the 1000 Genomes Consortium (that is, sequencing a number of individuals for polymorphism discovery ...). Only ten studies used tools recommended by the consortium for mapping and variant discovery, and just four studies used the full workflow involving realignment and quality score recalibration. Interestingly, three of the four studies were co-authored by at least one member of the 1000 Genomes Consortium." (Nekrutenko and Taylor, 2012)

"The above challenges can be distilled into two main issues. First, most biomedical researchers experience great difficulty carrying out computationally complex analyses on large data sets. Second, there is a lack of mechanism for documenting analytical steps in detail. Recently, a number of solutions have been developed that, if widely adopted, would solve the bulk of these challenges. These solutions can collectively be called integrative frameworks, as they bring together diverse tools under the umbrella of a unified interface. These include BioExtract, Galaxy, GenePattern, Gene Prof, Mobyle, and others. These tools record all analysis metadata, including tools, versions and parameter settings used, ultimately transforming research provenance for a task that requires active tracking by the analyst to one that is completely automatic." (Nekrutenko and Taylor, 2012)

"The overwhelming majority of currently published papers using NGS technologies include analyses that are not detailed ... Moreover, the computational approaches used in these publications cannot be readily reused by others." (Nekrutenko and Taylor, 2012)

"Many classical publications in life sciences have become influential because they provide complete information on how to repeat reported analyses so others can adopt these approaches in their own research, such as for chain termination sequencing technology that was developed by Sanger and colleagues and for PCR. Today's publications that include computational analyses are very different. Next-generation sequencing (NGS) technologies are undoubtedly as transformative as DNA sequencing and PCR were more than 30 years ago. As more and more researchers use high-throughput sequencing in their research, they consult other publications for examples of how to carry out computational analyses. Unfortunately, they often find that the extensive informatics component that is required to analyse NGS data makes it much more difficult to repeat studies published today. Note that the lax standards of computational reproducibility are not unique to life sciences; the importance of being able to repeat computational experiments was first brought up in geosciences and became relevant in life sciences following the establishment of microarray technology and high-throughput sequencing. Replication of computational experiments requires access to input data sets, source code or binaries of exact versions of software used to carry out the initial analysis (this includes all helper scripts that are used to convert formats, groom data, and so on) and knowing all parameter settings exactly as they were

used. In our experience, ... publications rarely provide such a level of detail, making biomedical computational analyses almost irreproducible." (Nekrutenko and Taylor, 2012)

Reproducibility in general:

"Engineers expect their work to be subject to an [independent validation and verification] IV&V process, in which the organization conducting the research uses a separate set of engineers to test, for example, whether microprocessors or navigation software work as expected. NASA's IV&V facility was established more than 25 years ago and has around 300 employees testing code and satellite components." (Raphael et al., 2020)

"The biological sciences have depended on other, less-reliable techniques for reproducibility. The most long-standing is the assumption that reproducibility studies will occur organically as different researchers work on related problems. In the past five years or so, funding agencies and journals have implemented more-stringent experimental-reporting and data-availability requirements for grant proposals and submitted manuscripts. A handful of initiatives have attempted to replicate published studies. The peer-reviewed 'Journal of Visualized Experiments' creates videos to disseminate details that are hard to convey in conventional methods sections. Yet pitfalls persist. Scientists might waste resources trying to build on unproven techniques. And real discoveries can be labelled irreproducible because too few resources are available to conduct a validation." (Raphael et al., 2020)

"Data-analysis pipelines are replete with configuration decisions, assumptions, dependencies and contingencies that move quickly beyond documentation, making troubleshooting incredibly difficult. ... Teams had to visit each others' labs more than once to understand and fully implement computational-analysis pipelines for large microscopy datasets." (Raphael et al., 2020)

"The scientific community has lost the connection with the original culture of skepticism which existed in the 17th century with the scientists of the Royal Society who pioneered the scientific method as captured in their motto nullius in verba ('take nobody's word'). They regarded the ability to replicate results in independent studies as a fundamental criterion for the establishment of a scientific fact. Modern scientific practice presents single experiments as proofs. When work is published, it is typically presented without self-criticism." (Neff, 2021)

"Science requires that we constantly question what we know—and that should include looking for advancements in study design or statistical analysis. For instance, the T-test was a great tool when we didn't have powerful computers to do complex computations. Now that we do, it seems logical to rethink this." (Neff, 2021)

"Currently, experimental design and data analysis expertise are not considered critical skills and yet research fundamentally relies on the experimental process and data is now an abundant output of the process. Statisticians are useful support resources, but demand exceeds supply. The scientific process needs quality, hands-on training that steps beyond theoretical concepts to give application skills. I personally find researchers are open; the resistance is arising as scientists are overwhelmed by resisting forces (resources, time, skill, knowledge)." (Neff, 2021)

"Transparency and quality management are key to improving scientific rigor and reproducibility. It is, therefore, good to see that the Open Science movement is gaining traction, and that research institutions increasingly view poor science as a reputation risk." (Neff, 2021)

"Robust findings become established over time as multiple lines of evidence emerge. Achieving robustness takes rigour and reproducibility, plus patience and judicious attention to the big picture." (Garraway, 2017)

"Diverse data that converge on the same observations in aggregate provide robustness, even though any single approach or model system has limitations." (Garraway, 2017)

"We scientists search tenaciously for information about how nature works through reason and experimentation. Who can deny the magnitude of knowledge we have gleaned, its acceleration over time, and its expanding positive impact on society? Of course, some data and models are fragile, and our understanding remains punctuated by false premises. Holding fast to the three Rs [rigor, reproducibility, and robustness] ensures that the path—although tortuous and treacherous at times—remains well lit." (Garraway, 2017)

"Improved reproducibility comes from pinning down methods." (Lithgow et al., 2017)

"Sources of variation include the quality and purity of reagents, daily fluctuations in microenvironment and the idiosyncratic techniques of investigators. With so many ways of getting it wrong, perhaps we should be surprised at how often experimental findings are reproducible." (Lithgow et al., 2017)

"Our first task, to develop a protocol, seemed straightforward. But subtle disparities were endless. In one particularly painful teleconference, we spent an hour debating the proper procedure for picking up worms and placing them on new agar plates. Some batches of worms lived a full day longer with gentler techniques. Because a worm's lifespan is only about 20 days, this is a big deal. Hundreds of e-mails and many teleconferences later, we converged on a technique but still had a stupendous three-day difference in lifespan between labs. The problem, it turned out, was notation—one lab determined age on the basis of when an egg hatched, others on when it was laid." (Lithgow et al., 2017)

"It is a rare project that specifies methods with such precision. In fact, several mouse researchers have argued that standardization is counterproductive—better to focus on results that persist across a wide range of conditions than to chase fragile findings that occur only within narrow parameters. We argue that another way forward is to chase down the variability and try to understand it within a common environment." (Lithgow et al., 2017)

"We have learnt that to understand how life works, describing how the research was done is as important as describing what was observed." (Lithgow et al., 2017)

"From time to time over the past few years, I've politely refused requests to referee an article on the grounds that it lacks enough information for me to check the work. This can be a hard thing to explain. Our lack of a precise vocabulary—in particular the fact that we don't have a word for 'you didn't tell me what you did in sufficient detail for me to check it'—contributes to the crisis of scientific reproducibility." (Stark, 2018)

"In computational science, 'reproducible' often means that enough information is provided to allow a dedicated reader to repeat the calculations in the paper for herself. In biomedical disciplines, 'reproducible' often means that a different lab, starting the experiment from scratch, would get roughly the same experimental result." (Stark, 2018)

"Results that generalize to all universes (or perhaps do not even require a universe) are part of mathematics. Results that generalize to our Universe belong to physics. Results that generalize to all life on Earth underpin molecular biology. Results that generalize to all mice are murine biology. And results that hold only for a particular mouse in a particular lab in a particular experiment are arguably not science." (Stark, 2018)

"Ushering in the Enlightenment era in the late seventeenth century, chemist Robert Boyle put forth his controversial idea of a vacuum and tasked himself with providing descriptions of his work sufficient 'that the person I addressed them to might, without mistake, and with as little trouble as possible, be able to repeat such unusual experiments'. Much modern scientific communication falls short of this standard. Most papers fail to report many aspects of the experiment and analysis that we may not with advantage omit—things that are crucial to understanding the result and its limitations, and to repeating the work. We have no common language to describe this shortcoming. I've been in conferences where scientists argued about whether work was reproducible, replicable, repeatable, generalizable and other '-bles', and clearly meant quite different things by identical terms. Contradictory meanings across disciplines are deeply entrenched." (Stark, 2018)

"The lack of standard terminology means that we do not clearly distinguish between situations in which there is not enough information to attempt repetition, and those in which attempts do not yield substantially the same outcome. To reduce confusion, I propose an intuitive, unambiguous neologism: 'preproducibility'. An experiment or analysis is preproducible if it has been described in adequate detail for others to undertake it. Preproducibility is a prerequisite for reproducibility, and the idea makes sense across disciplines." (Stark, 2018)

"The distinction between a preproducible scientific report and current common practice is like the difference between a partial list of ingredients and a recipe. To bake a good loaf of bread, it isn't enough to know that it contains flour. It isn't even enough to know that it contains flour, water, salt and yeast. The brand of flour might be omitted from the recipe with advantage, as might the day of the week on which the loaf was baked. But the ratio of ingredients, the operations, their timing and the temperature of the oven cannot. Given preproducibility—a 'scientific recipe'—we can attempt to make a similar loaf of scientific bread. If we follow the recipe but do not get the same result, either the result is sensitive to small details that cannot be controlled, the result is incorrect or the recipe was not precise enough (things were omitted to disadvantage). Depending on the discipline, preproducibility might require information about materials (including organisms and their care), instruments and procedures; experimental design; raw data at the instrument level; algorithms used to process the raw data; computational tools used in analyses, including any parameter settings or ad hoc choices; code, processed data and software build environments; or analyses that were tried and abandoned." (Stark, 2018)

"Just as I have pledged not to review papers that are not preproducible, I have also pledged not to submit papers without providing the software I used, and—to the extent permitted by law and ethics—the underlying data. I urge you to do the same." (Stark, 2018)

"If I publish an advertisement for my work (that is, a paper long on results but short on methods) and it's wrong, that makes me untrustworthy. If I say: 'here's my work' and it's wrong, I might have erred, but at least I am honest. If you and I get different results, preproducibility can help us to identify why—and the answer might be fascinating." (Stark, 2018)

"It should not need to be stated, but here goes. Reproducibility is the key underlying principle of science." (Gibb, 2014)

"As chemists, we have to be able to go to the literature, take a procedure, and carry out a similar or identical transformation with our own hands. Frustratingly, this doesn't always happen, and the next-to-worst-case scenario possible is when it's one of your own reactions that can't be reproduced by a lab elsewhere. Unsurprisingly, one step worse than this is when you can't even reproduce one of your own reactions in your own lab!" (Gibb, 2014)

"If there is nothing wrong with the reagents and reproducibility is still an issue, then as I like to tell students, there are two options: (1) the physical constants of the universe and hence the laws of physics are in a state of flux in their round-bottomed flask, or (2) the researcher is doing something wrong and either doesn't know it or doesn't want to know it. Then I ask them which explanation they think I'm leaning towards." (Gibb, 2014)

"Independent of what chemical companies do, it's useful to have a good set of standard operating procedures to maximize molecular hygiene (a phrase I like because it emphasizes that we should treat our molecules in the same way as we treat, say, our teeth). All established labs typically have such procedures, but the problem is that newcomers often end up learning them the hard way." (Gibb, 2014)

"Another approach to maximizing reproducibility is to use automation to cut out sloppy, all-too-human mistakes." (Gibb, 2014)

"Quality systems are an integral part of most commercial goods and services, used in manufacturing everything from planes to paint. Some labs that focus on clinical applications implement certified QA systems such as Good Clinical Practice, Good Manufacturing Practice and Good Laboratory Practice for data submitted to regulatory bodies. There have also been efforts to guide research practices outside these schemes. In 2001, the World Health Organization published guidelines for QA in basic research. And in 2006, the British Association of Research Quality Assurance (now simply the RQA) in Ipswich issued guidelines for basic biomedical research. But few academic researchers know that these standards exist ... Instead, QA tends to be ad hoc in academic settings. Many scientists are taught how to keep lab notebooks by their mentors, supplemented perhaps by a perfunctory training course. Investigators often improvise ways to safeguard data, maintain equipment or catalogue and care for experimental materials. Too often, data quality is as likely to be assumed as assured." (Baker, 2016)

"Many think that antibodies are a major driver of what has been deemed a 'reproducibility crisis', a growing realization that the results of many biomedical experiments cannot be reproduced and that the conclusions based on them may be unfounded. Poorly characterized antibodies probably contribute more to the problem than any other laboratory tool, says Glenn Begley, chief scientific officer at TetraLogic Pharmaceuticals in Malvern, Pennsylvania, and author of a controversial analysis showing that results in 47 of 53 landmark cancer research papers could not be reproduced." (Baker, 2015)

"Antibodies are produced by the immune systems of most vertebrates to target an invader such as a bacterium. Since the 1970s, scientists have exploited antibodies for research. If a researcher injects a protein of interest into a rabbit, white blood cells known as B cells will start producing antibodies against the protein, which can be collected from the animal's blood. For a more consistent product, the B cells can be retrieved, fused with an 'immortalized' cell and cultured to provide a theoretically unlimited supply." (Baker, 2015)

"The sandbox of the budding builder is not *making* as much as it is modification: taking something that exists and making it better, either functionally or aesthetically or both. Often that involves attaching and securing parts that were not originally intended to go together." (Savage, 2020)

Thinking about the data-generating process—to figure out how to preprocess your data, you need to think about all the processes that influenced it to have the characteristics that it does. [Example of a very stressed mouse, or recalibrating equipment between runs of samples]

"That a language is easy for the computer expert does not mean it is necessarily easy for the non-expert, and it is likely non-experts who will do the bulk of the programming (coding, if you wish) in the near future." (Hamming, 1997)

"What is wanted in the long run, of course, is that the man with the problem does the actual writing of the code with no human interface, as we all too often have these days, between the person who knows the problem and the person who knows the programming language. This date is unfortunately too far off to do much good immediately, but I would think by the year 2020 it would be fairly universal practice for the expert in the field of application to do the actual program preparation rather than have experts in computers (and ignorant in the field of application) do the program preparation." (Hamming, 1997)

"Rule 1: Do not attempt to analyze the data until you understand what is being measured and why. ... You may have to ask a lot of questions in order to clarify the objectives, the meaning of each variable, the units of measurement, the meaning of special symbols, whether similar experiments have been carried out before, and so on." (Chatfield, 1995)

"Statistical models usually contain one or more systematic components as well as a random (or stochastic) component. The random component, sometimes called the noise, arises for a variety of reasons, and it is sometimes helpful to distinguish between (i) measurement error and (ii) natural random variability arising from differences between experimental units and from changes in experimental conditions which cannot be controlled. The systematic component, sometimes called

the signal, may be deterministic, but there is increasing interest in the case where the signal evolves through time according to probabilistic laws. In engineering parlance, a statistical analysis can be regarded as extracting information about the signal in the presence of noise." (Chatfield, 1995)

Data pre-processing

When we take measurements of experimental samples, we do so with the goal of using the data we collect to gain scientific knowledge. The data are direct measurement of something, but need to be interpreted to gain knowledge. Sometimes direct measurements line up very closely with a research question—for example if you are conducting a study that investigates the mortality status of each test subject then whether or not each subject to dies is a data point that is directly related to the research question you are aiming to answer. In this case these data may go directly into a statistical analysis model without extensive pre-processing. However, there are often cases where we collect data that are not as immediately linked to the scientific question. Instead, these data may require pre-processing before they can be used to test meaningful scientific hypotheses. This is often the case for data extracted using complex equipment. Equipment like mass spectrometers and flow cytometers leverage physics, chemistry, and biology in clever ways to help us derive more information from samples, but one tradeoff is that the data from such equipment often require a bit of work to move into a format that is useful for answering scientific questions.

One example if the data collected through liquid chromatography-mass spectrometry (LC-MS). This is a powerful and useful technique for chemical analysis, including analysis of biochemical molecules like metabolites and proteins. However, when using this technique, the raw data require extensive pre-processing before they can be used to answer scientific questions.

First, the data that are output by the mass spectrometer are often stored in a specialized file format, like a netCDF or mzML file format. While these file formats are standardized, they are likely formats you don't regularly use in other contexts, and so you may need to find special tools to read the data into programs to analyze it. In some cases, the data are very large, and so it may be necessary to use analysis tools that allow most of the data to stay "on disk" while you analyze it, bringing only small parts into your analysis software at a time.

Once the data are read in, they must be pre-processed in a number of ways. For example, these data can be translated into features that are linked to the chemical composition of the sample, with each feature showing up as a "peak" in the data that are output from the mass spectrometer. A peak can be linked to a specific metabolite feature based on its mass-to-charge ratio (m/z) and its retention time. However, the exact retention time for a metabolite feature may vary a bit from sample to sample. Pre-processing is required both to identify

peaks in the data and also to align the peaks from the same metabolite feature across all samples from your experiment. There may also be technical bias across samples, resulting in differences in the typical expression levels of all peaks from one sample to the next. For example it may be the case that all intensities measured for one sample tend to be higher than for another sample because of technical bias in terms of the settings used for the equipment when the two samples were run. These biases must also be corrected through pre-processing before you can use the data within statistical tests or models to explore scientific hypotheses.

[Image of identifying and aligning peaks in LC-MS data]

In the research process, these pre-processing steps should be done before the data are used for further analysis. There are the first step in working with the data after they are collected by the equipment (or by laboratory personal, in the case of data from simpler process, like plating samples and counting colony-forming units). After the data are appropriately pre-processed, you can use them for statistical tests—for example, to determine if metabolite profiles are different between experimental groups—and also combine them with other data collected from the experiment—for example, to see whether certain metabolite levels are correlated with the bacterial load in a sample.

Approaches for pre-processing data.

There are two main approaches for pre-processing experimental data in this way. First, when data are the output of complex laboratory equipment, there will often be proprietary software that is available for this pre-processing. This software may be created by the same company that made the equipment, or it may be created and sold by other companies. The interface will typically be a graphical-user interface (GUI), where you will use pull-down menus and point-and-click interfaces to work through the pre-processing steps. You often will be able to export a pre-processed version of the data in a common file format, like a delimited file or an Excel file, and that version of the data can then be read into more general data analysis software, like Excel or R.

[Include a screenshot of this type of software in action.]

In the simplest case, the point-and-click interface that is used for this approach to pre-processing could be Excel or another version of spreadsheet software. An Excel spreadsheet might be used with data recorded “by hand” in the laboratory, with the researcher using embedded equations (or calculating and entering new values by hand) for steps like calculating values based on the raw data (for example, determining the time since the start of an experiment based on the time stamp of each data collection timepoint). A spreadsheet may also be used in steps of quality control. For example, if a recorded data point is known to be problematic (for example, ...), then the data point may be highlighted in some way in the spreadsheet (e.g., with color) or removed entirely by deleting or clearing the cell in the spreadsheet.

“There are currently [2017] very few, if any, ‘plug-and-play’ packages that allow researchers to quality control (QC), analyse and interpret scRNA-seq data,

although companies that sell the wet-lab hardware and reagents for scRNA-seq are increasingly offering free software (for example, Loupe from 10x Genomics, and Singular from Fluidigm). These are user-friendly but have the drawback that they are to some extent a ‘black box’, with little transparency as to the precise algorithmic details and parameters employed.” (Haque et al., 2017)

The second approach is to conduct the pre-processing directly within general data analysis software like R or Python. These programs are both open-source, and include extensions that were created and shared by users around the world. Through these extensions, there are often powerful tools that you can use to pre-process complex experimental data. In fact, the algorithms used in proprietary software are sometimes extended from algorithms first shared through R or Python. With this approach, you will read the data into the program (R, for example) directly from the file output from the equipment. You can record all the code that you use to read in and pre-process the data in a code script, allowing you to reproduce this pre-processing work. You can also go a step further, and incorporate your code into a pre-processing protocol, which combines nicely formatted text with executable code, and which we’ll describe in much more detail later in this module and in the following two modules.

There are advantages to taking the second approach—using scripted code in an open-source program—rather than the first—using proprietary software with a GUI interface. The use of codes scripts ensures that the steps of pre-processing are reproducible. This means both that you will be able to re-do all the steps yourself in the future, if you need to, but that also that other researchers can explore and replicate what you do. You may want to share your process with others in your laboratory group, for example, so they can understand the choices you made and steps you took in pre-processing the data. You may also want to share the process with readers of the articles you publish, and this may in fact be required by the journal. Further, the use of a code script encourages you to document this code and this process, even moreso when you move beyond a script and include the code in a reproducible pre-processing protocol. Well-documented code makes it much easier to write up the method section later in manuscripts that leveraged the data collected in the experiment.

Also, when you use scripted code to pre-process biomedical data, you will find that the same script can often be easily adapted and re-used in later projects that use the same type of data. You may need to change small elements, like the file names of files with data you want to use, or some details about the methods used for certain pre-processing steps. However, often almost all of the pre-processing steps will repeat over different experiments that you do. By extending to write a pre-processing protocol, you can further support the ease of adapting and re-using the pre-processing steps you take with one experiment when you run later experiments that are similar.

3.1.8 Approaches to simple preprocessing tasks

There are several approaches for tackling this type of data preprocessing, to get from the data that you initial observe (or that is measured by a piece of laboratory equipment) to meaningful biological measurements that can be analyzed and presented to inform explorations of a scientific hypothesis. While there are a number of approaches that don't involve writing code scripts for this preprocessing, there are some large advantages to scripting preprocessing any time you are preprocessing experimental data prior to including it in figures or further analysis. In this section, we'll describe some common non-scripted approaches and discuss the advantages that would be brought by instead using a code script. In the next module, we'll walk through an example of how scripts for preprocessing can be created and applied in laboratory research.

In cases where the pre-processing is mathematically straightforward and the dataset is relatively small, many researchers do the preprocessing by hand in a laboratory notebook or through an equation or macro embedded in a spreadsheet. For example, if you have plated samples at different dilutions and are trying to calculate from these the CFUs in the original sample, this calculation is simple enough that it could be done by hand. However, there are advantages to instead writing a code script to do this simple preprocessing.

When you write a script to do a task with data, it is like writing a recipe that can be applied again and again. By writing a script, you encode the process a single time, so you can take the time to check and recheck to make sure that you've encoded the process correctly. This helps in avoiding small errors when you do the preprocessing—if you are punching numbers into a calculator over and over, it's easy to mistype a number or forget a step every now and then, while the code will ensure that the same process is run every time and that it faithfully uses the numbers saved in the data for each step, rather than relying on a person correctly entering each number in the calculation.

Scripts can be used across projects, as well, and so they can ensure consistency in the calculation across projects. If different people do the calculation in the lab for different projects or experiments, and they are doing the calculations by hand, they might each do the calculation slightly differently, even if it's only in small details like how they report rounded numbers. A script will do the exact same thing every time it is applied. You can even share your script with colleagues at other labs, if you want to ensure that your data preprocessing is comparable for experiments conducted in different research groups, and many scientific journals will allow supplemental material with code used for data preprocessing and analysis, or links within the manuscript to a repository of this code posted online.

There are also gains in efficiency when you use a script. For small preprocessing steps, these might seem small for each experiment, and certainly when you first write the script, it will likely take longer to write and test the script than it would to just do the calculation by hand (even more if you're just

starting to learn how to write code scripts). However, since the script can be applied again and again, with very little extra work to apply it to new data, you'll save yourself time in the future, and over a lot of experiments and projects, this can add up. This makes it particularly useful to write scripts for preprocessing tasks that you find yourself doing again and again in the lab.

3.1.9 *Approaches to more complex preprocessing tasks*

Other preprocessing tasks can be much more complex, particularly those that need to conduct a number of steps to extract biologically meaningful measurements from the measurements made by a complex piece of laboratory equipment, as well as steps to make sure these measurements can be meaningfully compared across samples.

For these more complex tasks, the equipment manufacturer will often provide software that can be used for the preprocessing. This software might conduct some steps using defaults, and others based on the user's specifications. These are often provided through "GUIs" (graphical user interfaces), where the user does a series of point-and-click steps to process the data. In some software, this series of point-and-click steps is recorded as the user does them, so that these steps can be "re-run" later or on a different dataset.

For many types of biological data, including output from equipment like flow cytometers and mass spectrometers, open-source software has been developed that can be used for this preprocessing. Often, the most cutting edge methods for data preprocessing are first available through open-source software packages, if the methods are developed by researchers rather than by the companies, and often many of the algorithms that are made available through the equipment manufacturer's proprietary software are encoded versions of an algorithm first shared by researchers as open-source software.

It can take a while to develop a code script for preprocessing the raw data from a piece of complex equipment like a mass spectrometer. However, the process of developing this script requires a thoughtful consideration of the steps of preprocessing, and so this is often time well-spent. Again, this initial time investment will pay off later, as the script can then be efficiently applied to future data you collect from the equipment, saving you time in pointing and clicking through the GUI software. Further, it's easier to teach someone else how to conduct the preprocessing that you've done, and apply it to future experiments, because the script serves as a recipe.

When you conduct data preprocessing in a script, this also gives you access to all the other tools in the scripting language. For example, as you work through preprocessing steps for a dataset, if you are doing it through an R script, you can use any of the many visualization tools that are available through R. By contrast, in GUI software, you are restricted to the visualization and other tools included in that particular set of software, and those software developers may not have thought of something that you'd like to do. Open-

source scripting languages like R, Python, and Julia include a huge variety of tools, and once you have loaded your data in any of these platforms, you can use any of these tools.

If you have developed a script for preprocessing your raw data, it also becomes much easier to see how changes in choices in preprocessing might influence your final results. It can be tricky to guess whether your final results are sensitive, for example, to what choice you make for a particular transform for part of your data, or in how you standardize data in one sample to make different samples easier to compare. If the preprocessing is in a script, then you can test making these changes and running all preprocessing and analysis scripts, to see if it makes a difference in the final conclusions. If it does, then it helps you identify parts of preprocessing that need to be deeply thought through for the type of data you’re collecting, and you may want to explore the documentation on that particular step of preprocessing to determine what choice is best for your data, rather than relying on defaults.

3.1.10 Scripting preprocessing tasks

Code scripts can be developed for any open-source scripting languages, including Python, R, and Julia. These can be embedded in or called from literate programming documents, like RMarkdown and Julia, which are described in other modules. The word “script” is a good one here—it really is as if you are providing the script for a play. In an interactive mode, you can send requests to run in the programming language step by step using a console, while in a script you provide the whole list of all of your “lines” in that conversation, and the programming language will run them all in order without you needing to interact from the console.

For preprocessing the data, the script will have a few predictable parts. First, you’ll need to read the data in. There are different functions that can be used to read in data from different file formats. For example, data that is stored in an Excel spreadsheet can be loaded into R using functions in a package called `readxl`. Data that is stored in a plain-text delimited format (like a csv file) can be loaded into R using functions in the `readr` package.

When preprocessing data from complex equipment, you can determine how to read the data into R by investigating the file type that is output by the equipment. Fortunately, many types of scientific equipment follow standardized file formats. This means that open-source developers can develop a single package that can load data from equipment from multiple manufacturers. For example, flow cytometry data is often stored in [file format]. Other biological datasets use file formats that are appropriate for very large datasets and that allow R to work with parts of the data at a time, without loading the full data in. [netCDF?] In these cases, the first step in a script might not be to load in all the data, but rather to provide R with a connection to the larger datafile, so it can pull in data as it needs it.

Once the data is loaded or linked in the script, the script can proceed through steps required to preprocess this data. These steps will often depend on the type of data, especially the methods and equipment used to collect it. For example, for mass spectrometry data, these steps will include For flow cytometry data, these steps would include

The functions for doing these steps will often come from extensions that different researchers have made for R. Base R is a simpler collection of data processing and statistics tools, but the open-source framework of R has allowed users to make and share their own extensions. In R, these are often referred to as “packages”. Many of these are shared through the Comprehensive R Archive Network (CRAN), and packages on CRAN can be directly installed using the `install.packages` function in R, along with the package’s names. While CRAN is the common spot for sharing general-purpose packages, there is a specialized repository that is used for many genomics and other biology-related R packages called Bioconductor. These packages can also be easily installed through a call in R, but in this case it requires an installation function from the `BiocManager` package. Many of the functions that are useful for preprocessing biological data from laboratory experiments are available through Bioconductor.

Table [x] includes some of the primary R packages on Bioconductor that can be used in preprocessing different types of biological data. There are often multiple choices, developed by different research groups, but this list provides a starting point of several of the standard choices that you may want to consider as you start developing code.

Much of the initial preprocessing might use very specific functions that are tailored to the format that the data takes once it is loaded. Later in the script, there will often be a transfer to using more general-purpose tools in that coding language. For example, once data is stored in a “dataframe” format in R, it can be processed using a powerful set of general purpose tools collected in a suite of packages called the “tidyverse”. This set of packages includes functions for filtering to specific subsets of the data, merging separate datasets, adding new measurements for each observation that are functions of the initial measurements, summarizing, and visualizing. The tidyverse suite of R tools is very popular in general R use and is widely taught, including through numerous free online resources. By moving from specific tools to these more general tools as soon as possible in the script, a researcher can focus his or her time in learning these general purpose tools well, as these can be widely applied across many types of data.

By the end of the script, data will be in a format that has extracted biologically relevant measurements. Ideally, this data will be in a general purpose format, like a dataframe, to make it easier to work with using general purpose tools in the scripting language when the data is used in further data analysis or to create figures for reports, papers, and presentations. Often, you will want to save a version of this preprocessed version of the data in your project files,

and so the last step of the script might be to write out the cleaned data in a file that can be loaded in later scripts for analysis and visualization. This is especially useful if these data preprocessing steps are time consuming, as is often the case for the large raw datasets output by laboratory equipment like flow cytometers and mass spectrometers.

Figure [x] gives an example of a data preprocessing script, highlighting these different common areas that often show up in these scripts.

[Some data may be incorporated into the preprocessing by downloading it from databases or other online sources. These data downloads can be automated and recorded by using scripted code for the download in many cases, as long as the database or online source offers web services or another API for this type of scripted data access. In this case, you can incorporate the script in a RMarkdown document to record the date the data was downloaded, as well as the code used to download it. R is able to run system calls, and one of these will provide the current date, so this can be included in an RMarkdown file to record the date the file is run. Further, there may be a call that can be made to the online data source's API that returns the working version of the database or source, and if so this can also be included in the RMarkdown code used to access the data.]

RMarkdown files can be used to combine both code and more manual document (for example, a record of which collaborator provided each type of data file). While traditionally this more manual documentation was recommended to be recorded in plain-text README files in a project's directory and subdirectories (Buffalo, 2015), RMarkdown files provide some advantages over this traditional approach. First, RMarkdown files are themselves in plain text, and so they offer the advantages of simple plain text documentation files (e.g., ones never rendered to another format) in terms of being able to use script-based tools to search them. Further, they can be rendered into attractive formatted documents that may be easier to share with project team members who do not code.

[Example of a function: recipe for making a vinaigrette. There will be a “basic” way that the function can run, which uses its default parameters. However, you can also specify and customize certain inputs (for example, using walnut oil instead of olive oil, or adding mustard) to tweak the recipe in slight ways each time you use it, and to get customized outputs.]

[History of the mouse—enable GUIs, before everything was from the terminal.]

3.1.11 Potential quotes

For bioinformatics, “all too often the software is developed without thought toward future interoperability with other software products. As a result, the bioinformatics software landscape is currently characterized by fragmentation and silos, in which each research group develops and uses only the tools created within their lab.” (Barga et al., 2011)

"The group also noted the lack of agility. Although they may be aware of a new or better algorithm they cannot easily integrate it into their analysis pipelines given the lack of standards across both data formats and tools. It typically requires a complete rewrite of the code in order to take advantage of a new technique or algorithm, requiring time and often funding to hire developers." (Barga et al., 2011)

"The benefit of working with a programming language is that you have the code in a file. This means that you can easily reuse that code. If the code has parameters it can even be applied to problems that follow a similar pattern." (Janssens, 2014)

"Data exploration in spreadsheet software is typically conducted via menus and dialog boxes, which leaves no record of the steps taken." (Murrell, 2009)

"One reason Unix developers have been cool toward GUI interfaces is that, in their designers' haste to make them 'user-friendly' each one often becomes frustratingly opaque to anyone who has to solve user problems—or, indeed, interact with it anywhere outside the narrow range predicted by the user-interface designer." (Raymond, 2003)

"Many operating systems touted as more 'modern' or 'user friendly' than Unix achieve their surface glossiness by locking users and developers into one interface policy, and offer an application-programming interface that for all its elaborateness is rather narrow and rigid. On such systems, tasks the designers have anticipated are very easy—but tasks they have not anticipated are often impossible or at best extremely painful. Unix, on the other hand, has flexibility in depth. The many ways Unix provides to glue together programs means that components of its basic toolkit can be combined to produce useful effects that the designers of the individual toolkit parts never anticipated." (Raymond, 2003)

"The good news is that a computer is a general-purpose machine, capable of performing any computation. Although it only has a few kinds of instructions to work with, it can do them blazingly fast, and it can largely control its own operation. The bad news is that it doesn't do anything itself unless someone tells it what to do, in excruciating detail. A computer is the ultimate sorcerer's apprentice, able to follow instructions tirelessly and without error, but requiring painstaking accuracy in the specification of what to do." (Kernighan, 2011)

"Software is the general term for sequences of instructions that make a computer do something useful. It's 'soft' in contrast with 'hard' hardware, because it's intangible, not easy to put your hands on. Hardware is quite tangible: if you drop a computer on your foot, you'll notice. Not true for software." (Kernighan, 2011)

"Modern systems increasingly use general purpose hardware—a processor, some memory, and connections to the environment—and create specific behaviors by software. The conventional wisdom is that software is cheaper, more flexible, and easier to change than hardware is (especially once some device has left the factory)." (Kernighan, 2011)

"An algorithm is a precise and unambiguous recipe. It's expressed in terms of a fixed set of basic operations whose meanings are completely known and specified; it spells out a sequence of steps using those operations, with all possible situations covered; it's guaranteed to stop eventually. On the other hand, a *program* is the

opposite of abstract—it's a concrete statement of the steps that a real computer must perform to accomplish a task. The distinction between an algorithm and a program is like the difference between a blueprint and a building; one is an idealization and the other is the real thing." (Kernighan, 2011)

"One way to view a program is as one or more algorithms expressed in a form that a computer can process directly. A program has to worry about practical problems like inadequate memory, limited processor speed, invalid and even malicious input data, faulty hardware, broken network connections, and (in the background and often exacerbating the other problems) human frailty. So if an algorithm is an idealized recipe, a program is the instructions for a cooking robot preparing a month of meals for an army while under enemy attack." (Kernighan, 2011)

"During the late 1950s and early 1960s, another step was taken towards getting the computer to do more for programmers, arguably the most important step in the history of programming. This was the development of 'high-level' programming languages that were independent of any particular CPU architecture. High-level languages make it possible to express computations in terms that are closer to the way a person might express them." (Kernighan, 2011)

"Programming in the real world tends to happen on a large scale. The strategy is similar to what one might use to write a book or undertake any other big project: figure out what to do, starting with a broad specification that is broken into smaller and smaller pieces, then work on the pieces separately, while making sure that they hang together. In programming, pieces tend to be of a size such that one person can write the precise computational steps in some programming language. Ensuring that the pieces written by different programmers work together is challenging, and failing to get this right is a major source of errors. For instance, NASA's Mars Climate Orbiter failed in 1999 because the flight system software used metric units for thrust, but course correction data was entered in English units, causing an erroneous trajectory that brought the Orbiter too close to the planet's surface." (Kernighan, 2011)

"If you're going to build a house today, you don't start by cutting down trees to make lumber and digging clay to make your own bricks. Instead, you buy prefabricated pieces like doors, windows, plumbing fixtures, a furnace, and a water heater. House construction is still a big job, but it's manageable because you can build on the work of many others and rely on an infrastructure, indeed an entire industry, that will help. The same is true of programming. Hardly any significant program is created from nothing. Many components written by others can be taken off the shelf and used. For instance, if you're writing a program for Windows or a Mac, there are libraries of prefabricated menus, buttons, text editors, graphics, network connections, database access, and so on. Much of the job is understanding the components and gluing them together in your own way. Of course, many of these components in turn rest on other simpler and more basic ones, often for several layers. Below that, everything runs on the operating system, a program that manages the hardware and controls everything that happens." (Kernighan, 2011)

"At the simplest level, programming languages provide a mechanism called functions that make it possible for one programmer to write code that performs a useful a useful task, then package it in a form that other programmers can use in their programs without having to know how it works." (Kernighan, 2011)

"A function has a name and a set of input data values that it needs to do its job; it does a computation and returns a result to the part of the program that called it. ... Functions make it possible to create a program by building on components that have been created separately and can be used as necessary by all programmers. A collection of related functions is usually called a *library*. ... The services that a function library provides are described to programmers in terms of an *Application Programming Interface*, or *API*, which lists the functions, what they do, how to use them in a program, what input data they require, and what values they produce. The API might also describe data structures—the organization of data that is passed back and forth—and various other bits and pieces that all together define what a programmer has to do to request services and what will be computed as a result. This specification must be detailed and precise, since in the end the program will be interpreted by a dumb literal computer, not by a friendly and accomodating human." (Kernighan, 2011)

"The code that a programmer writes, whether in assembly language or (much more likely) in a high-level language, is called *source code*. ... Source code is readable by other programmers, though perhaps with some effort, so it can be studied and adapted, and any innovations or ideas it contains are visible." (Kernighan, 2011)

"In early times, most software was developed by companies and most source code was unavailable, a trade secret of whoever developed it." (Kernighan, 2011)

"An *operating system* is the software underpinning that manages the hardware of a computer and makes it possible to run other programs, which are called *applications*. ... It's a clumsy but standard terminology for programs that are more or less self-contained and focused on a single task." (Kernighan, 2011)

"Software, like many other things in computing, is organized into layers, analogous to geological strata, that separate one concern from another. Layering is one of the important ideas that help programmers to manage complexity." (Kernighan, 2011)

"I think that it's important for a well-informed person to know something about programming, perhaps only that it can be surprisingly difficult to get very simple programs working properly. There is nothing like doing battle with a computer to teach this lesson, but also to give people a taste of the wonderful feeling of accomplishment when a program does work for the first time. It may also be valuable to have enough programming experience that you are cautious when someone says that programming is easy, or that there are no errors in a program. If you have trouble making 10 lines of code work after a day of struggle, you might be legitimately skeptical of someone who claims that a million-line program will be delivered on time and bug-free." (Kernighan, 2011)

"Programming languages share certain basic ideas, since they are all notations for spelling out a computation as a sequence of steps. Every programming language thus will provide ways to get input data upon which to compute; do arithmetic; store and retrieve intermediate values as computation proceeds; display results along the way; decide how to proceed on the basis of previous computations; and save results when the computation is finished. Languages have *syntax*, that is, rules that define what is grammatically legal and what is not. Programming languages are picky on the grammatical side: you have to say it right or there will

be a complaint. Languages also have *semantics*, that is, a defined meaning for every construction in the language." (Kernighan, 2011)

"In programming, a *library* is a collection of related pieces of code. A library typically includes the code in compiled form, along with needed source code declarations [for C++]. Libraries can include stand-alone functions, classes, type declarations, or anything else that can appear in code." (Spraul, 2012)

"One way to write R code is simply to enter it interactively at the command line... This interactivity is beneficial for experimenting with R or for exploring a data set in a casual manner. ... However, interactively typing code at the R command line is a very bad approach from the perspective of recording and documenting code because the code is lost when R is shut down. A superior approach in general is to write R code in a file and get R to read the code from the file." (Murrell, 2009)

"The features of R are organized into separate bundles called *packages*. The standard R installation includes about 25 of those packages, but many more can be downloaded from CRAN and installed to expand the things that R can do. ... Once a package is installed, it must be *loaded* within an R session to make the extra features available. ... Of the 25 packages that are installed by default, nine packages are *loaded* by default when we start a new R session; these provide the basic functionality of R. All other packages must be loaded before the relevant features can be used." (Murrell, 2009)

"The R environment is the software used to run R code." (Murrell, 2009)

"*Document your methods and workflows.* This should include full command lines (copied and pasted) that are run through the shell that generate data or intermediate results. Even if you use the default values in software, be sure to write these values down; later versions of the program may use different default values. Scripts naturally document all steps and parameters ..., but be sure to document any command-line options used to run this script. In general, any command that produces results in your work needs to be documented somewhere." (Buffalo, 2015)

"*Document the version of the software that you ran.* This may seem unimportant, but remember the example from 'Reproducible Research' on page 6 where my colleagues and I traced disagreeing results down to a single piece of software being updated. These details matter. Good bioinformatics software usually has a command-line option to return the current version. Software managed with a version control system such as Git has explicit identifiers to every version, which can be used to document the precise version you ran... If no version information is available, a release date, link to the software, and download date will suffice." (Buffalo, 2015)

"*Document when you downloaded data.* It's important to include when the data was downloaded, as the external data source (such as a website or server) might change in the future. For example, a script that downloads data directly from a database might produce different results if rerun after the external database is updated. Consequently, it's important to document when data came into your repository." (Buffalo, 2015)

"All of this [documentation] information is best stored in plain-text README files. Plain text can easily be read, searched, and edited directly from the command line, making it the perfect choice for portable and accessible README files. It's also available on all computer systems, meaning you can document your steps when working directly on a server or computer cluster. Plain text also lacks complex formatting, which can create issues when copying and pasting commands from your documentation back into the command line." (Buffalo, 2015)

"The computer is a very flexible and powerful tool, and it is a tool that is ours to control. Files and documents, especially those in open standard formats, can be manipulated using a variety of software tools, not just one specific piece of software. A programming language is a tool that allows us to manipulate data stored in files and to manipulate data held in RAM in unlimited ways. Even with a basic knowledge of programming, we can perform a huge variety of data processing tasks." (Murrell, 2009)

"Computer code is the preferred approach to communicating our instructions to the computer. The approach allows us to be precise and expressive, it provides a complete record of our actions, and it allows others to replicate our work." (Murrell, 2009)

"Programming in R is carried out, primarily, by manipulating and modifying data structures. These different transformations are carried out using functions and operators. In R, virtually every operation is a function call, and though we separate our discussion into operators and function calls, the distinction is not strong ... The R evaluator and many functions are written in C but most R functions are written in R itself." (Gentleman, 2008)

"Many biologists are first exposed to the R language by following a cookbook-type approach to conduct a statistical analysis like a t-test or an analysis of variance (ANOVA). Although R excels at these and more complicated statistical tests, R's real power is as a data programming language you can use to explore and understand data in an open-ended, highly interactive, iterative way. Learning R as a data programming language will give you the freedom to experiment and problem solve during data analysis—exactly what we need as bioinformaticians." (Buffalo, 2015)

"Popularized by statistician John W. Tukey, EDA is an approach that emphasizes understanding data (and its limitations) through interactive investigation rather than explicit statistical modeling. In his 1977 book *Exploratory Data Analysis*, Tukey described EDA as 'detective work' involved in 'finding and revealing the clues' in data. As Tukey's quote emphasizes, EDA is much more an approach to exploring data than using specific statistical methods. In the face of rapidly changing sequencing technologies, bioinformatics software, and statistical methods, EDA skills are not only widely applicable and comparatively stable—they're also essential to making sure that our analyses are robust to these new data and methods." (Buffalo, 2015)

"Developing code in R is a back-and-forth between writing code in a rerunnable script and exploring data interactively in the R interpreter. To be reproducible, all steps that lead to results you'll use later must be recorded in the R script that accompanies your analysis and interactive work. While R can save a history of the commands you've entered in the interpreter during a session (with the command

`savehistory()`), storing your steps in a well-commented R script makes your life much easier when you need to backtrack to understand what you did or change your analysis.” (Buffalo, 2015)

“It’s a good idea to avoid referring to specific dataframe rows in your analysis code. This would produce code fragile to row permutations or new rows that may be generated by rerunning a previous analysis step. In every case in which you might need to refer to a specific row, it’s avoidable by using subsetting... Similarly, it’s a good idea to refer to columns by their column name, *not* their position. While columns may be less likely to change across dataset versions than rows, it still happens. Column names are more specific than positions, and also lead to more readable code.” (Buffalo, 2015)

“In bioinformatics, we often need to extract data from strings. R has several functions to manipulate strings that are handy when working with bioinformatics data in R. Note, however, that for most bioinformatics text-processing tasks, R is *not* the preferred language to use for a few reasons. First, R works with all data stored in memory; many bioinformatics text-processing tasks are best tackled with the stream-based approaches..., which explicitly avoid loading all data in memory at once. Second, R’s string processing functions are admittedly a bit clunky compared to Python’s.” (Buffalo, 2015)

“Versions for R and any R packages installed change over time. This can lead to reproducibility headaches, as the results of your analyses may change with the changing version of R and R packages. ... you should always record the versions of R and any packages you use for analysis. R actually makes this incredibly easy to do—just call the `sessionInfo()` function.” (Buffalo, 2015)

“Bioconductor is an open source R software project focused on developing tools for high-throughput genomics and molecular biology data.” (Buffalo, 2015)

“Bioconductor’s package system is a bit different than those on the Comprehensive R Archive Network (CRAN). Bioconductor packages are released on a set schedule, twice a year. Each release is coordinated with a version of R, making Bioconductor’s versions tied to specific R versions. The motivation behind this strict coordination is that it allows for packages to be thoroughly tested before being released for public use. Additionally, because there’s considerable code re-use within the Bioconductor project, this ensures that all package versions within a Bioconductor release are compatible with one another. For users, the end result is that packages work as expected and have been rigorously tested before you use it (this is good when your scientific results depend on software reliability!). If you need the cutting-edge version of a package for some reason, it’s always possible to work with their development branch.” (Buffalo, 2015)

“When installing Bioconductor packages, we use the `biocLite()` function. `biocLite()` installs the correct version of a package for your R version (and its corresponding Bioconductor version).” (Buffalo, 2015)

“In addition to a careful release cycle that fosters package stability, Bioconductor also has extensive, excellent documentation. The best, most up-to-date documentation for each package will always be a Bioconductor [web address]. Each package has a full reference manual covering all functions and classes included in a package, as well as one or more in-depth vignettes. Vignettes step through many examples and common workflows using packages.” (Buffalo, 2015)

“Quite often, users don’t appreciate the opportunities. Noncomputational biologists don’t know when to complain about the status quo. With modest amounts of computational consulting, long or impossible jobs can become much shorter or richer.” — Barry Demchak in (Altschul et al., 2013)

“People not doing the computational work tend to think that you can write a program very fast. That, I think, is frankly not true. It takes a lot of time to implement a prototype. Then it actually takes a lot of time to really make it better.” — Heng Li in (Altschul et al., 2013)

“There is also a problem with discovering software that exists; often people reinvent the wheel just because they don’t know any better. Good repositories for software and best practice workflows, especially if citable, would be a start.” — James Taylor in (Altschul et al., 2013)

” Now there are a lot of strong, young, faculty members who label themselves as computational analysts, yet very often want wet-lab space. They’re not content just working off data sets that come from other people. They want to be involved in data generation and experimental design and mainstreaming computation as a valid research tool. Just as the boundaries of biochemistry and cell biology have kind of blurred, I think the same will be true of computational biology. It’s going to be alongside biochemistry, or molecular biology or microscopy as a core component.” — Richard Durbin in (Altschul et al., 2013)

“I would say that computation is now as important to biology as chemistry is. Both are useful background knowledge. Data manipulation and use of information are part of the technology of biology research now. Knowing how to program also gives people some idea about what’s going on inside data analysis. It helps them appreciate what they can and can’t expect from data analysis software.” — Richard Durbin in (Altschul et al., 2013)

“Does every new biology PhD student need to learn how to program? To some, the answer might be “no” because that’s left to the experts, to the people downstairs who sit in front of a computer. But a similar question would be: does every graduate student in biology need to learn grammar? Clearly, yes. Do they all need to learn to speak? Clearly, yes. We just don’t leave it to the literature experts. That’s because we need to communicate. Do students need to tie their shoes? Yes. It has now come to the point where using a computer is as essential as brushing your teeth. If you want some kind of a competitive edge, you’re going to want to make as much use of that computer as you can. The complexity of the task at hand will mean that canned solutions don’t exist. It means that if you’re using a canned solution, you’re not at the edge of research.” — Martin Krzywinski in (Altschul et al., 2013)

“Although we are tackling many different types of data, questions, and statistical methods hands-on, we maintain a consistent computational approach by keeping all the computation under one roof: the R programming language and statistical environment, enhanced by the biological data infrastructure and specialized method packages from the Bioconductor project.” (Holmes and Huber, 2018)

“The availability of over 10,000 packages [in R] ensures that almost all statistical methods are available, including the most recent developments. Moreover, there are implementations of or interfaces to many methods from computer science,

mathematics, machine learning, data management, visualization and internet technologies. This puts thousands of person-years of work by experts at your fingertips.” (Holmes and Huber, 2018)

“Bioconductor packages support the reading of many of the data types and formats produced by measurement instruments used in modern biology, as well as the needed technology-specific ‘preprocessing’ routines. This community is actively keeping these up-to-date with the rapid developments in the instrument market.” (Holmes and Huber, 2018)

“An equivalent to the laboratory notebook that is standard good practice in lab-work, we advocate the use of a computational diary written in the R markdown format. ... Together with a version control system, R markdown helps with tracking changes.” (Holmes and Huber, 2018)

“There are (at least) two types of data visualization. The first enables a scientist to explore data and make discoveries about the complex processes at work. The other type of visualization provides informative, clear and visually attractive illustrations of her results that she can show to others and eventually include in a publication.” (Holmes and Huber, 2018)

“A common task in biological data analysis is comparison between several samples of univariate measurements. ... As an example, we’ll use the intensities of a set of four genes... A popular way to display [this] is through barplots [and boxplots, violin plots, dot plots, and beeswarm plots].” (Holmes and Huber, 2018)

“At different stages of their development, immune cells express unique combinations of proteins on their surfaces. These protein-markers are called CDs (clusters of differentiation) and are collected by flow cytometry (using fluorescence...) or mass cytometry (using single-cell atomic mass spectrometry of heavy metal reporters). An example of a commonly used CD is CD4; this protein is expressed by helper T cells that are referred to as being ‘CD4+’. Note, however, that some cells express CD4 (thus are CD4+) but are not actually helper T cells. We start by loading some useful Bioconductor packages for flow cytometry, *flowCore* and *flowViz*. ... First we load the table data that reports the mapping between isotopes and markers (antibodies), and then we replace the isotope names in the column names ... with the marker names. Changing the column names makes the subsequent analysis and plotting easier to read. ... Plotting the data in two dimensions... already shows that the cells can be grouped into subpopulations. Sometimes just one of the markers can be used to define populations on its own; in that case, simple rectangular gating is used to separate the populations. For instance, CD4+ populations can be gating by taking the sub-population with high values for the CD4 marker. Cell clustering can be improved by carefully choosing transformations of the data. ... [Such a transformation] reveals bimodality and the existence of two cell populations... It is standard to transform both flow and mass cytometry data using one of several special functions. We take the example of the inverse hyperbolic arcsine (asinh) ... for large values of x , $\text{asinh}(x)$ behaves like the \log and is practically equal to $\log(x) + \log(2)$; for small x the function is close to linear in x This is another example of a variance-stabilizing transformation.” (Holmes and Huber, 2018)

“Consider a set of measurements that reflect some underlying true values (say, species represented by DNA sequences from their genomes) but have been

degraded by technical noise. Clustering can be used to remove such noise.”
 (Holmes and Huber, 2018)

“In the bacterial 16SrRNA gene there are so-called variable regions that are taxon-specific. These provide fingerprints that enable taxon identification. The raw data are FASTQ-files with quality scored sequences of PCR-amplified DNA regions. We use an iterative alternating approach to build a probabilistic noise model from the data. We call this a *de novo* method, because we use clustering, and we use the cluster centers as our denoised sequence variants... After finding all the denoised variants, we create contingency tables of their counts across the different samples. ... these tables can be used to infer properties of the underlying bacterial communities using networks and graphs. **In order to improve data quality, we often have to start with the raw data and model all the sources of variation carefully.** We can think of this as an example of cooking from scratch. ... The DADA method ... uses a parameterized model of substitution errors that distinguishes sequencing errors from real biological variation. ... The dereplicated sequences are read in, and then divisive denoising and estimation is run with the dada function... In order to verify that the error transition rates have been reasonably well estimated, we inspect the fit between the observed error rates ... and the fitted error rates .. Once the errors have been estimated, the algorithm is rerun on the data to find the sequence variants. ... Sequence inference removes nearly all substitution and indel errors from the data. [Footnote: ‘The term indel stands for insertion-deletion; when comparing two sequences that differ by a small stretch of characters, it is a matter of viewpoint whether this is an insertion or a deletion, hence the name]. We merge the inferred forward and reverse sequences while removing paired sequences that do not perfectly overlap, as a final control against residual errors.” (Holmes and Huber, 2018)

“Chimera are sequences that are artificially created during the PCR amplification by the melding of two (or, in rare cases, more) of the original sequences. To complete our denoising workflow, we remove them with a call to the function removeBimeraDenovo, leaving us with a clean contingency table that we will use later.” (Holmes and Huber, 2018)

“We load up the RNA-Seq dataset airway, which contains gene expression measurements (gene-level counts) of four primary human airway smooth muscle cell lines with and without treatment with dexamethasone, a synthetic glucocorticoid. We’ll use the DESeq2 method ... it performs a test for differential expression for each gene.” (Holmes and Huber, 2018)

“In many cases, different variables are measured in different units, so they have different baselines and different scales. [Footnote: ‘Common measures of scale are the range and the standard deviation...’] For PCA and many other methods, we therefore need to transform the numeric values to some common scale in order to make comparisons meaningful. Centering means subtracting the mean, so that the mean of the centered data is at the origin. Scaling or standardizing then means dividing by the standard deviation, so that the new standard deviation is 1. ... To perform these operations, there is the R function `scale`, whose default behavior when given a matrix or a data frame is to make each column have a mean of 0 and a standard deviation of 1. ... We have already encountered other data transformation choices in Chapters 4 and 5, where we used the `log` and `asinh` functions. The aim of these transformations is (usually) variance

stabilization, i.e., to make the variances of the replicate measurements of one and the same variable in different parts of the dynamic range more similar. In contrast, the standardizing transformation described above aims to make the scale (as measured by mean and standard deviation) of different variables the same. Sometimes it is preferable to leave variables at different scales because they are truly of different importance. If their original scale is relevant, then we can (and should) leave the data alone. In other cases, the variables have different precisions known a priori. We will see in Chapter 9 that there are several ways of weighting such variables. After preprocessing the data, we are ready to undertake data simplification through dimension reduction.” (Holmes and Huber, 2018)

With data that give the number of reads for each gene in a sample, “The data have a large dynamic range, starting from zero up to millions. The variance and, more generally, the distribution shape of the data in different parts of the dynamic range are very different. We need to take this phenomenon, called heteroscedascity, into account. The data are non-negative integers, and their distribution is not symmetric—thus normal or log-normal distribution models may be a poor fit. We need to understand the systematic sampling biases and adjust for them. Confusingly, such adjustment is often called normalization. Examples are the total sequencing depth of an experiment (even if the true abundance of a gene in two libraries is the same, we expect different numbers of reads for it depending on the total number of reads sequenced) and differing sampling probabilities (even if the true abundance of two genes within a biological sample is the same, we expect different numbers of reads for them if they have differing biophysical properties, such as length, GC content, secondary structure, binding partners).” (Holmes and Huber, 2018)

“Often, systematic biases affect the data generation and are worth taking into account. Unfortunately, the term normalization is commonly used for that aspect of the analysis, even though it is misleading; it has nothing to do with the normal distribution, nor does it involve a data transformation. Rather, what we aim to do is identify the nature and magnitude of systematic biases and take them into account in our model-based analysis of the data. The most important systematic bias [for count data from high-throughput sequencing applications like RNA-Seq] stems from variations in the total number of reads in each sample. If we have more reads for one library than for another, then we might assume that, everything else being equal, the counts are proportional to each other with some proportionality factor s . Naively, we could propose that a decent estimate of s for each sample is simply given by the sum of the counts of all genes. However, it turns out that we can do better...” (Holmes and Huber, 2018)

“When testing for differential expression, we operate on raw counts and use discrete distributions. For other downstream analyses—e.g., for visualization or clustering—it can be useful to work with transformed versions of the count data. Maybe the most obvious choice of transformation is the logarithm. However, since count values for a gene can be zero, some analysts advocate the use of pseudocounts, i.e., transformations of the form $y = \log_2(n + 1)$ or more generally $y = \log_2(n + n_0)$.” (Holmes and Huber, 2018)

“The data sometimes contain isolated instances of very large counts that are apparently unrelated to the experimental or study design and may be considered outliers. Outliers can arise for many reasons, including rare technical or

experimental artifacts, read mapping problems in the case of genetically differing samples, and genuine but rare biological events. In many cases, users appear primarily interested in genes that show consistent behavior, and this is the reason why, by default, genes that are affected by such outliers are set aside by DESeq. The function calculates, for every gene and for every sample, a diagnostic test for outliers called Cook's distance. Cook's distance is a measure of how much a single sample is influencing the fitted coefficients for a gene, and a large value of Cook's distance is intended to indicate an outlier count. DESeq2 automatically flags genes with Cook's distance above a cutoff and sets their p-values and adjusted p-values to NA. ... With many degrees of freedom—i.e., many more samples than number of parameters to be estimated—it might be undesirable to remove entire genes from the analysis just because their data include a single count outlier. An alternative strategy is to replace the outlier counts with the trimmed mean over all sample, adjusted by the size factor for that sample. This approach is conservative: it will not lead to false positives, as it replaces the outlier value with the value predicted by the null hypothesis." (Holmes and Huber, 2018)

"Since the sampling depth is typically different for different sequencing runs (replicates), we need to estimate the effect of this variable parameter and take it into account in our model. ... Often this part of the analysis is called normalization (the term is not particularly descriptive, but unfortunately it is now well established in the literature)." (Holmes and Huber, 2018)

"Our experimental interventions and our measurement instruments have limited precision and accuracy; often we don't know these limitations at the outset and have to collect preliminary data to estimate them." (Holmes and Huber, 2018)

"Our treatment conditions may have undesired but hard-to-avoid side effects; our measurements may be overlaid with interfering signals or 'background noise'." (Holmes and Huber, 2018)

"Sometimes we explicitly know about factors that cause bias, for instance, when different reagent batches were used in different phases of the experiment. We call these batch effects (Leek et al., 2010). At other times, we may expect that such factors are at work but have no explicit record of them. We call these latent factors. We can treat them as adding to the noise, and in Chapter 4 we saw how to use mixture models to do so. But this may not be enough; with high-dimensional data, noise caused by latent factors tends to be correlated, and this can lead to faulty inference (Leek et al., 2010). The good news is that these same correlations can be exploited to estimate latent factors from the data, model them as bias, and thus reduce the noise (Leek and Storey 2007; Stegle et al. 2010)." (Holmes and Huber, 2018)

"Regular noise can be modeled by simple probability models such as independent normal distributions or Poissons, or by mixtures such as the gamma-Poisson or Laplace. We can use relatively straightforward methods to take such noise into account in our data analyses and to compute the probability of extraordinarily large or small values. In the real world, this is only part of the story: measurements can be completely off-scale (a sample swap, a contamination, or a software bug), and they can all go awry at the same time (a whole microtiter plate went bad, affecting all data measured from it). Such events are hard to model or even correct for—our best chance of dealing with them is data quality assessment, outlier detection, and documented removal." (Holmes and Huber, 2018)

"In Chapters 4 and 8 we saw examples of data transformations that compress or stretch the space of quantitative measurements in such a way that the measurements' variance is more similar throughout. Thus the variance between replicated measurements is no longer highly dependent on the mean value. The mean-variance relationship of our data before transformation can, in principle, be any function, but in many cases, the following prototypic relationships are found, at least approximately: 1. Constant: the variance is independent of the mean...; 2. Poisson: the variance is proportional to the mean...; 3. Quadratic: the standard deviation is proportional to the mean; therefore the variance grows quadratically... The mean-variance relationship in real data can also be a combination of these basic types. For instance, with DNA microarrays, the fluorescence intensities are subject to a combination of background noise that is largely independent of the signal, and multiplicative noise whose standard deviation is proportional to the signal (Rocke and Durbin 2001). ... What is the point of applying a variance-stabilizing transformation? Analyzing the data on the transformed scale tends to: 1. Improve visualization, since the physical space on the plot is used more 'fairly' throughout the range of the data. A similar argument applies to the color space in the case of a heatmap. 2. Improve the outcome of ordination methods such as PCA or clustering based on correlation, as the results are not so much dominated by the signal from a few very highly expressed genes, but more uniformly from many genes throughout the dynamic range. 3. Improve the estimates and inference from statistical models that are based on the assumption of identically distributed (and, hence, homoscedastic) noise." (Holmes and Huber, 2018)

"We distinguish between data quality assessment (QA)—steps taken to measure and monitor data quality—and quality control—the removal of bad data. These activities pervade all phases of an analysis, from assembling the raw data over transformation, summarization, model fitting, hypothesis testing or screening for 'hits' to interpretation. QA-related questions include: 1. How do the marginal distributions of the variables look (histograms, ECDF plots)? 2. How do their joint distributions look (scatterplots, pair plots)? 3. How well do replicated agree (as compared to different biological conditions)? Are the magnitudes of different between several conditions plausible? 4. Is there evidence of batch effects? These could be of a categorical (stepwise) or continuous (gradual) nature, e.g., due to changes in experimental reagents, protocols or environmental factors. Factors associated with such effects may be explicitly known, or unknown and latent, and often they are somewhere in between (e.g., when a measurement apparatus slowly degrades over time, and we have recorded the times, but don't really know exactly when the degradation becomes bad). For the last two sets of questions, heatmaps, principal components plots, and other ordination plots (as we have seen in Chapters 7 and 9) are useful." (Holmes and Huber, 2018)

"It's not easy to define quality, and the word is used with many meanings. The most pertinent for us is fitness for purpose, and this contrasts with other definitions that are based on normative specifications. For instance, in differential expression analysis with RNA-Seq data, our purpose may be the detection of differentially expressed genes between two biological conditions. We can check specifications such as the number of reads, read length, base calling quality and fraction of aligned reads, but ultimately these measures in isolation have little bearing on our purpose. More to the point will be the identification of samples that are not behaving as expected, e.g., because of a sample swap or degradation, or genes that were not measured properly. ... Useful plots include ordination

plots ... and heatmaps ... A quality metric is any value that we use to measure quality, and having explicit quality metrics helps in automating QA/QC." (Holmes and Huber, 2018)

"Use literate programming tools. Examples are Rmarkdown and Jupyter. This makes code more readable (for yourself and for others) than burying explanations and usage instructions in comments in the source code or in separate README files. In addition, you can directly embed figures and tables in these documents. Such documents are good starting points for the supplementary material of your paper. Moreover, they're great for reporting analyses to your collaborators." (Holmes and Huber, 2018)

"Use functions. It's better than copy-pasting (or repeatedly source-ing) stretches of code." (Holmes and Huber, 2018)

Use the R package system. Soon you'll note recurring function or variable definitions that you want to share between your different scripts. It is fine to use the R function `source` to manage them initially, but it is never too early to move them into your own package—at the latest when you find yourself staring to write emails or code comments explaining to others (or to yourself) how to use some functionality. Assembling existing code into an R package is not hard, and it offers you many goodies, including standardized ways of composing documentation, showing code usage examples, code testing, versioning and provision to others. And quite likely you'll soon appreciate the benefits of using namespaces." (Holmes and Huber, 2018)

"Think in terms of cooking recipes and try to automate them. When developing downstream analysis ideas that bring together several different data types, you don't want to do the conversion from data-type-specific formats into a representation suitable for machine learning or a generic statistical method each time anew, on an ad hoc basis. Have a recipe script that assembles the different ingredients and cooks them up as an easily consumable [footnote: In computer science, the term data warehouse is sometimes used for such a concept] matrix, dataframe or Bioconductor `SummarizedExperiment`." (Holmes and Huber, 2018)

"Centralize the location of the raw data files and automate the derivation of intermediate data. Store the input data on a centralized file server that is professionally backed up. Mark the files as read-only. Have a clear and linear workflow for computing the derived data (e.g., normalized, summarized, transformed, etc.) from the raw files, and store these in a separate directory. Anticipate that this workflow will need to be run several times, and version it. Use the `BiocFileCache` package to mirror these files on your personal computer. [footnote: A more basic alternative is the `rsync` utility. A popular solution offered by some organizations is based on `ownCloud`. Commercial options are `Dropbox`, `Google Drive` and the like]." (Holmes and Huber, 2018)

"Keep a hyperlinked webpage with an index of all analyses. This is helpful for collaborators (especially if the page and the analysis can be accessed via a web browser) and also a good starting point for the methods part of your paper. Structure it in chronological or logical order, or a combination of both." (Holmes and Huber, 2018)

"Getting data ready for analysis or visualization often involves a lot of shuffling until they are in the right shape and format for an analytical algorithm or a graphics routine." (Holmes and Huber, 2018)

"Data analysis pipelines in high-throughput biology often work as 'funnels' that successively summarize and compress the data. In high-throughput sequencing, we may start with individual sequencing reads, then align them to a reference, then only count the aligned reads for each position, summarize positions to genes (or other kinds of regions), then 'normalize' these numbers by library size to make them comparable across libraries, etc. At each step, we lose information, yet it is important to make sure we still have enough information for the task at hand. [footnote: For instance, for the RNA-Seq differential expression analysis we saw in Chapter 8, we needed the actual read counts, not 'normalized' versions; for some analyses, gene-level summaries might suffice, for others, we'll want to look at the exon or isoform level.] The problem is particularly acute if we build our data pipeline with a series of components from separate developers. Statisticians have a concept for whether certain summaries enable the reconstruction of all the relevant information in the data: sufficiency. ... Iterative approaches akin to what we saw when we used the EM algorithm can sometimes help to avoid information loss. For instance, when analyzing mass spectroscopy data, a first run guesses at peaks individually for each sample. After this preliminary spectrum-spotting, another iteration allows us to borrow strength from the other samples to spot spectra that may have been overlooked (or looked like noise) before." (Holmes and Huber, 2018)

Try to avoid adding in supplementary / meta data "by hand". For example, if you need to add information about the ID of each sample, and this information is included somewhere in the filename, it will be more robust to use regular expressions to extract this information from the file names, rather than entering it by hand. If you later add new files, the automated approach will be robust to this update, while errors might be introduced for information added by hand.

Example of quality control functionality in xcms: "Below we create boxplots representing the distribution of total ion currents per file. Such plots can be very useful to spot problematic or failing MS runs. ... Also, we can cluster the samples based on similarity of their base peak chromatogram. This can also be helpful to spot potentially problematic samples in an experiment or generally get an initial overview of the sample grouping in the experiment. Since the retention times between samples are not exactly identical, we use the bin function to group intensities in fixed time ranges (bins) along the retention time axis. In the present example we use a bin size of 1 second, the default is 0.5 seconds. The clustering is performed using complete linkage hierarchical clustering on the pairwise correlations of the binned base peak chromatograms." (Smith, 2013)

After some quality checks on the data from LCMS, the next step is to detect the chromatographic peaks in the samples. This requires some specifications to the algorithm that will depend on the settings used on the equipment when the samples were run. "Next we perform the chromatographic peak detection using the centWave algorithm [2]. Before running the peak detection it

is however strongly suggested to visually inspect e.g. the extracted ion chromatogram of internal standards or known compounds to evaluate and adapt the peak detection settings since the default settings will not be appropriate for most LCMS experiments. The two most critical parameters for centWave are the peakwidth (expected range of chromatographic peak widths) and ppm (maximum expected deviation of m/z values of centroids corresponding to one chromatographic peak; this is usually much larger than the ppm specified by the manufacturer) parameters. To evaluate the typical chromatographic peak width we plot the EIC for one peak.” (Smith, 2013)

“Peak detection will not always work perfectly leading to peak detection artifacts, such as overlapping peaks or artificially split peaks. The refineChromPeaks function allows to refine peak detection results by either removing identified peaks not passing a certain criteria or by merging artificially split chromatographic peaks.” (Smith, 2013)

“The time at which analytes elute in the chromatography can vary between samples (and even compounds). Such a difference was already observable in the extracted ion chromatogram plot shown as an example in the previous section. The alignment step, also referred to as retention time correction, aims at adjusting this by shifting signals along the retention time axis to align the signals between different samples within an experiment. A plethora of alignment algorithms exist (see [3]), with some of them being implemented also in xcms. The method to perform the alignment/retention time correction in xcms is adjustRtime which uses different alignment algorithms depending on the provided parameter class. ... In some experiments it might be helpful to perform the alignment based on only a subset of the samples, e.g. if QC samples were injected at regular intervals or if the experiment contains blanks. Alignment method in xcms allow to estimate retention time drifts on a subset of samples (either all samples excluding blanks or QC samples injected at regular intervals during a measurement run) and use these to adjust the full data set.” (Smith, 2013)

“The final step in the metabolomics preprocessing is the correspondence that matches detected chromatographic peaks between samples (and depending on the settings, also within samples if they are adjacent). The method to perform the correspondence in xcms is groupChromPeaks. We will use the peak density method [5] to group chromatographic peaks. The algorithm combines chromatographic peaks depending on the density of peaks along the retention time axis within small slices along the mz dimension.” (Smith, 2013)

“The performance of peak detection, alignment and correspondence should always be evaluated by inspecting extracted ion chromatograms e.g. of known compounds, internal standards or identified features in general.” (Smith, 2013)

Normalization can help adjust for technical bias across the samples:

“At last we perform a principal component analysis to evaluate the grouping of the samples in this experiment. Note that we did not perform any data normalization hence the grouping might (and will) also be influenced by technical biases. ... We can see the expected separation between the KO and WT samples on PC2. On PCI samples separate based on their ID, samples with an ID <= 18

from samples with an ID > 18. This separation might be caused by a technical bias (e.g. measurements performed on different days/weeks) or due to biological properties of the mice analyzed (sex, age, litter mates etc).” (Smith, 2013)

“Normalizing features’ signal intensities is required, but at present not (yet) supported in xcms (some methods might be added in near future). It is advised to use the SummarizedExperiment returned by the quantify method for any further data processing, as this type of object stores feature definitions, sample annotations as well as feature abundances in the same object. For the identification of e.g. features with significant different intensities/abundances it is suggested to use functionality provided in other R packages, such as Bioconductor’s excellent limma package.” (Smith, 2013)

3.1.12 Discussion questions

3.2 Introduction to scripted data pre-processing in R

We will show how to implement scripted pre-processing of experimental data through R scripts. We will demonstrate the difference between interactive coding and code scripts, using R for examples. We will then demonstrate how to create, save, and run an R code script for a simple data cleaning task.

Objectives. After this module, the trainee will be able to:

- Describe what an R code script is and how it differs from interactive coding in R
- Create and save an R script to perform a simple data pre-processing task
- Run an R script
- List some popular packages in R for pre-processing biomedical data

3.2.1 What is a code script?

Interactive coding you can do in what’s called the console. With this style of coding, you enter a single command or function call at the cursor in the console, tell the program to execute that one element of code (for example, by pressing the Return key), and then wait until it executes it before you enter the next command or function call.

A script, on the other hand, is a longer document that gives all the steps in a process. You can think of a script as being like a script for a play—it’s a record of everything that happens over the course of the event. For a play, the script records the dialogue and stage directions for a play, while for a data preprocessing task, it can record all the steps from inputting the data through to saving the data in a processed form for further analysis, visualization, and statistical testing.

You can run the same code whether you’re using a script or typing in the commands one at a time in the console. However, when you code interactively at the console, you’re not making a record of each of your steps (as a note, there are ways to save the history of commands typed at a console, but it can

be very messy to try to use later to reproduce and remember what you did originally, so you should consider commands that are typed at the console to not be recorded for the purposes of reproducibility). When you write your code in a script, on the other hand, you do have a record that you can later reopen to see what you did or to repeat the steps. In a very broad way, you can visualize this process as walking in wet sand—you are making a record (footsteps) of the path you took while you make that path by walking it the first time.

A code script is typically written in a plain text document, and you can create, edit, and save code scripts in any interactive development environment (like RStudio if you are programming in R). The program (R for example) can then read and run this script as a “batch” at any time. In other words, it can walk through and execute each piece of code that you recorded in the script, rather than you needing to enter each line of code one at a time in the console. For many programming languages, you can also run the code in a script in smaller sections, executing just one or a few lines at a time to explore what’s happening in each line of the code. With this combination of functionality, as well as recording of code for future reference or reproduction, code scripts provide an excellent method for building and using pipelines of code to preprocess biomedical data.

In later sections of this module, we’ll walk through the practical steps of writing one of these code scripts, as well as look at an example script for a simple task in biomedical data preprocessing, calculating the rate of growth of bacteria under different growing conditions.

3.2.2 How code scripts improve reproducibility of preprocessing

3.2.3 How to write an R code script

In this section, we’ll go through some basics to help you get started writing a code script in R. The process of writing a code script is similar in many other interpreted languages, like Python and Julia. If you are familiar with writing code scripts in R, you may want to skip this section.

We’ll start with a few basics of the conventions of the R programming language. If you have never used R before, it is critical to understand these basic pieces—just enough so you can understand how an R code script is put together and run. In later modules, we’ll go into some more detail about some helpful tools in R, including Bioconductor data structures and the suite of “tidyverse” tools that are now taught in most beginner R programming courses. We will have room to provide a full course on how to program in R, but we are aiming to give you enough of a view that you can understand how R programming can fit in to the data preprocessing and analysis pipeline for laboratory-based biomedical research projects. In a later chapter, we’ll provide directions to more resources if you would like to continue developing your expertise in R programming beyond the basics covered in these modules.

What is an R object?

First, you'll need to understand where R "keeps" data while you're working with it. When you work in R, any piece of data that you work with will be available in something called an "object". The simplest way to think of this R object is simply as a container for data. Different objects can be structured in different ways, in terms of how they arrange the data—which has implications about how you can access the data from that object—but regardless of this structure, all R objects share the same purpose of storing data in a way that's available to you as you work in R.

One of the first steps in most R scripts, therefore, will be to create some of these objects. Until you have some data available, there's not much interesting stuff that you can do in R. If you want to work with data that are stored in a file—for example, data that you recorded in the laboratory and saved in an Excel file—then you can create an R object with that data by reading in the data using a specific R function (we'll cover these in a minute). This will read the data in R and store it in an object where you can access it later.

To keep track of the objects you have in your R session, you typically assign each object a name. Any time you want to use the data in that object, or work with the object in any way, you can then refer to it by that name, rather than needing to repeat all the code you used to initially create it. You can assign an object its name using a special function in R called the "gets arrow" or assignment operator. It's an arrow made of the less than and hyphen keys, with no spaces between the two (<-). You'll put the name you want to give the object to the left of this arrow and the code to create the object (for example, to read in data from a file) to the right. Therefore, the beginning of your R script will often have one or more lines of code that look like this:

In this example, the line of code is reading in data from an Excel file named "my_recorded_data.xlsx" and storing in an R object that is assigned the name my_data. When you want to work with these data later in the code pipeline, you can do so by referencing my_data, which now stores the data from that file.

In addition to creating objects from the data that you initially read in, you will likely create more intermediate objects along the way. For example, if you take your initial data and filter it down to a subset, then you might assign that version of the data to a separate object name, so you can work with that version later in your code. Alternatively, in some cases you'll just overwrite the original object with the new version, using the same object name (for example, creating a subset of the my_data object and assigning it the same name of my_data). This reassigns the object name—when you refer to my_data from that point on, it will contain the subsetted version. However, in some cases this can be useful because it helps keep the collection of R objects you have in your session a bit smaller and simpler. What's more, you can make these changes to simplify the version of the data you're working with in R without worrying about it changing your raw data. Once you read the data in from an outside file, like an Excel file, R will work on a copy of that data, not the original data.

You can make as many changes as you want to the data object in R without it changing anything in your raw data.

What are R functions and an R function calls?

The next key component of the R programming language is the idea of R functions and R function calls. These are the parts of R that do things (whereas the objects in R are the “things” that these functions operate on). An R function is a tool that can take one or more R objects as inputs, do something based on those inputs, and return a new R object as the output (occasionally they’ll also have “side effects” beyond returning this R object—for example, some functions will make a plot and show it in the plotting window of RStudio).

The R objects that you input can be ones that you’ve assigned to a name (for example, `my_data`). They can also be simple objects that you make on the fly, just to have to input to that function. For example, if you’re reading in data from a file, one of the R object inputs you’ll need to give the function is the path to that file, which you could either save as an object (e.g., `my_data_filepath <- "my_recorded_data.xlsx"` and then reference `my_data_filepath` when you call the function) or create as an object on the fly when you call the function (e.g., just put `"my_recorded_data.xlsx"` directly in the function call, as shown in the example above).

The function itself is the tool, which encapsulates the code to do something with input objects. When you use that tool, it’s called *calling* the function. Therefore, all of the lines of code in your script will give *function calls*, where you are asking R to run a specific function (or, in some cases, a linked set of functions) based on specified inputs.

For example, the following function call would read in data from the Excel file `"my_recorded_data.xlsx"`:

This line of code is calling the function `read_excel`, which is a tool for inputting data from an Excel file into an R object with a specific data structure. By running this line of code, either at the console or in an R script, you are asking R to input data from the file named `"my_recorded_data.xlsx"`, which is the R object that you’re giving as an input to the function. This particular call would only read the data in—it won’t assign the resulting object to a name, but instead will just print out the data at the R console.

If you’d like to read the data in and save it in an object to use later, you’ll want to add another function to this call, so that you assign the output object a name. For this, you’ll use the gets arrow that we described earlier. This is a special type of function in R. Most R functions consist of the function’s name, followed by parentheses inside of which you put the objects to input to the function (e.g., `read_excel("my_recorded_dat.xlsx")`). The gets arrow is a different type of function called an operator. These functions go between two objects, both of which are input to the operator function. They’re used often for arithmetic (for example, the `+` operator adds the values in the objects before and after it, so that you can call `1 + 2` to add one and two). For the gets arrow, it will go between the name that you want to assign to

the object (e.g., `my_data`) and the function call that creates that object (e.g., `read_excel("my_recorded_data.xlsx")`):

```
my_data <- read_excel("my_recorded_data.xlsx")
```

In this case, the line that R will execute will include two functions, where the output of one gets linked straight into the second, and the result will be the output from the second function (that the data in the Excel file is stored in an object assigned the name `my_data`).

As you write an R script, you will use function calls to work through the preprocessing steps. You can use different function calls to do things like apply a transformation, average values across groups, or reduce dimensions of a high-dimensional dataset. Once you've preprocessed the data, you can also use function calls to run statistical tests with the data and to visualize results through figures and tables. The process of writing a script is normally very iterative—you'll write the code to do the first few steps (e.g., read in the data), look at what you've got, plan out some next steps, try to write some code for those steps, run it and check your output, and so on. The process is very similar to drafting a paper. You can try things out in early steps—and some steps won't work out at first, or it will turn out that you don't need them. As you continue, you'll refine the script, editing it down to the essential steps and making sure each function call within those steps is operating as you intend. While it can be intimidating to start with a blank file and develop some code—just like it is with a blank piece of paper when writing a manuscript—just like with writing, you are able to start with something rough and then iterate until you arrive at the version you want.

This process might seem a bit overwhelming when you first learn it, but it suffices at this point if you understand that, in R code, you'll be working with objects (your materials) and functions (your tools). As we look through R scripts in later parts of this module, we'll see these two pieces—objects and functions—used again and again in the scripts. They are the building blocks for your R scripts.

What is an R library?

There's one last component of R that will be helpful to understand as we move through the rest of this module and the next few modules. That's the idea of an R package, and fortunately, it's a pretty straightforward one.

We just talked about how functions in R are tools, which you can use to do interesting things with your data (including all the preprocessing steps we talked about in the last module). However, the version of R that you initially install to your computer (available for free for all major operating systems at <https://cran.r-hub.io/>) doesn't include all the tools that you will likely want to use. The initial download gives you the base of the programming language, which is called base R, as well as a few extensions of this for very common tasks, like fitting some common statistical models.

Because R is an open-source software, people who use R can build on

top of this simple base. R users can create new functions that combine more rudimentary tools in base R to create customized tools suited to their own tasks. R users can create these tools for their own personal use, and often do, but there is also a mechanism for them to share these new tools with others if they'd like. They can bundle a set of R functions they've created into an *R package* and then post this package on a public repository where others can download it and use the functions in it.

In some of the examples in these modules, we'll be using tools from these packages, and it's rare that someone uses R without using at least some of these supplementary packages, so it's good to get an idea of how to get and use them. The people who make packages can share them in a number of repositories, but the most standard repository for sharing R packages widely is the Comprehensive R Archive Network (CRAN). If a package is shared through CRAN, you can get it using the function `install.packages` along with the package's name. For example, in the code we showed earlier, the `read_excel` function does not come with base R, but instead is part of a package called `readxl`, which is shared on CRAN. To download that package so that you can use its functions, you can run:

```
install.packages("readxl")
```

This will download the code for the package and unpack it in a special part of your computer where R can easily find it. You only need to install a package once, at least until you get a new computer or update your version of base R. However, to use the functions in that package, you'll need to *load* the package in your current R session. This makes the functions in that package available to you as you work in that R session. To do this, you use the `library` function, along with the name of the package. For example, to load the `readxl` package in an R session, you'd need to run:

```
library("readxl")
```

While you only need to install a package once, you need to load it every time you open a new R session to do work, if you want to use its functions in that R session. Therefore, you'll often see a lot of calls to the `library` function in R scripts. You can use this call anywhere in the script as long as you put it before code where you use the library's functions, but it's great to get in the habit of putting all the `library` function calls at the start of your R script. That way, if you share the script with someone else, they can quickly check to see if they'll need to install any new packages before they can run the code in the script.

Using a plain text file, edited in a plain text editor (can open in RStudio IDE)

Based on the points that we've just discussed, hopefully you can envision now that an R script will ultimately include a number of lines of code, covering a number of R function calls that work with data stored in objects. You can

expect there to be lots of calls that assign objects their own names (with `<-`), and the function calls will typically include both a function called by name and some objects as input to that function, contained inside parentheses after the function name.

This type of script should be written in plain text, and so the best way to create an R script is by using a text editor. Your computer likely came with a text editor as one of the pieces of utility software that was installed by default. However, with R scripts, it can be easier to use the text editor that comes as part of RStudio. This allows you to open and edit your scripts in a nice environment, one that includes a console area where you can test out pieces of code, a pane for viewing figures, and so on.

In RStudio, you can create a new R script by going to the “File” menu at the top of the screen, choosing “New File” and then choosing “R Script”. This will open a new plain text file that, by default, will have the file extension “.R” (e.g., “my_file.R”), which is the standard file extension for R scripts. Once you’ve created an R script file, you can begin writing your script. In the next section, we’ll walk through how you can run code that you’ve put into your script. However, we think it’s worth mentioning that, as you get started on this process, you might find it easiest to start not by writing your own R script from scratch, but instead by starting with someone else’s and walking through that. You can explore how it works (reverse engineer it). Then you can try changing small parts, to see if it acts as you expect when you do. This process will help you get a feel for how these scripts are organized and how they operate. Later in this module, we’ll provide an R script for a basic laboratory data preprocessing task and walk you through it, so you can use that as a starting point to understand how it would work to create, edit, and run your own R script.

3.2.4 How to run code in an R script

Once you’ve written code in an R script, you can run (execute) that code in a number of ways. First, you can run all the code in the script at once, which is known as *batch execution*. When you do this, all the code in the script will be executed by R, and while it’s executed by R one line at a time, you won’t have the chance to make changes along the way. If you compare it to the idea of a code script as like a play script, you can think of this as being like when the play is performed for an audience—you start the play, but then you don’t have the chance to stop and work on it as it’s going. Instead, it will go straight through to the end. If there is an error somewhere along the way, then the code will stop running at that point and you’ll get an error message, but otherwise when you run the code as a batch, R won’t stop executing the lines until it gets to the end. This mode of running the code is great for once you’ve developed a pipeline that you’re happy with—it quickly runs everything and provides the output.

The other way that you can execute the code is by running a single line, or

a small set of lines, of the code at a time. In the play analogy, this is similar to what might happen during rehearsals, when you go through part of the play script and then stop to get comments from the director, then either re-try that part with a few changes or move on to the next small part. This mode of running the code is great for when you're developing the pipeline. Just like with a play's rehearsals, you'll want a lot of chances to explore and change things as you develop the final product, and this mode of running code is excellent for exploration and editing. Often, most of your time when you code will be spent doing this style of code execution. Running in batch mode will get a lot of work done, but is very quick for the programmer—developing the code is what takes time, and just like with writing a manuscript, this time comes from drafting a rough draft and then editing it until you arrive at a clean and clear final version.

Both of these methods of code execution are easy to do in RStudio. Since you'll usually start by using line-by-line execution, we'll start with showing how you can do that. In RStudio, you can open your code script (a file ending in ".R"), and you will still be able to see the console, which is a space for submitting function calls to R. To execute the code in the script one line at a time, there's a few quick ways that you can tell RStudio to send that line in the script to the console to run. Start by putting your cursor on that line of code. One way to now execute this line (i.e., send it to the console to run) is to click on the "Run" button in the top right-hand corner of the script file. If you try this, you should see that this line of code gets sent to the console pane of RStudio, and the results from running that line are shown in the console.

Even quicker is a keyboard shortcut that does the same thing. (Keyboard shortcuts are short control sequences that you type in your keyboard to run a command. They're faster than clicking buttons because you can do them without taking your hands off the keyboard. Ctrl-C is one very common one that you might have used before, which in most programs will copy the current selection.) With your cursor on the line of the function call that you want to execute, use the keyboard shortcut Ctrl-Enter (depending on your operating system, you may need to use Command rather than Ctrl).

You can use a similar method to run a few lines of code at once. All you have to do is highlight the code that you want to run, and then you can use either of the two methods (click the "Run" button or use the Ctrl-Enter keyboard shortcut).

[Figure—how to execute one or a few lines of code in RStudio]

To execute an R script in batch mode, there are again a could of ways you can do it. First, there is a "Source" button in the top right of the R script file when you open it in RStudio. You can click on this button and it will run the entire script as a batch. There is also an R command that you can use to source a file based on its file name, `source`. If you have a file in your working directory named "my_pipeline.R", for example, you can execute the code in it in a batch by running `'source("my_pipeline.R")'`.

[Figure—how to execute an R script as a batch]

To get started, it's probably easiest to just use the buttons "Run" and "Source" that RStudio provides in the window for the R script file. As you do more work, you may find some of these other methods help you work faster, or allow you to do more interesting things, so it's good to know they're there, but you don't need to try to navigate them all as you learn how to run code in an R script.

3.2.5 *Simple example—Bacterial growth rate*

(A simple example of writing a code script for data preprocessing)

3.2.6 *Style guidelines for writing code scripts*

Find favorite tools and get to know them

Modify rather than start from scratch

As you start learning to write code in R, don't force yourself to stare at an empty R script file and try to come up with a full script from scratch. One of the best ways to learn R is to find some scripts that others have written for tasks that are similar to the ones that you want to do, then work through those to figure out each function call, and how those function calls add up to the full pipeline.

This method of reverse engineering is useful in many areas when you're trying to figure out how things work. ...

Once you understand a few other R scripts, you can start trying to modify them and to pull pieces from different scripts to use as building blocks as you put together your own script. There's no need to reinvent the wheel—if someone else has shared an R script that comes close to doing what you need to do, start there and then change and evolve that idea to suit your own needs.

To find some starting scripts to learn from, there are a few tactics you can try. First, check around with colleagues to see if they have R code for data preprocessing tasks that they do in their lab. If they work with similar types of data, and use R, they're likely to have come up with some scripts that achieve tasks you also need to do.

Another excellent source of example R code are the vignettes and examples that come with many R packages. If you are using functions from an R package, then there is likely a vignette that comes with that package, and there may also be examples within the helpfiles for each of the package's functions. A package vignette is a tutorial that walks you through the major functionality of the package, showing how to use the key functions in the package in an extended example. Some packages will have multiple vignettes, showing a range of things that you can do with the package.

To find out if there is a vignette for a package that you're using, you can google the package name and "vignette". You can also find out from the console in R using the function `vignette`. For example, to find out if the package `readxl`, which we used in the earlier code example, has any vignettes, you can

`run vignette(package = "readxl")`. This will tell you that the package has two, one called “cell-and-column-types” and one called “sheet-geometry”. To open one of these, you can again use the `vignette` function. For example, `vignette("cell-and-column-types", package = "readxl")` would open the first of the two vignettes.

To open the helpfile for any function in R, at the console type a question mark and then the function name. For example, `?read_excel` will open the helpfile for the `read_excel` function (you will need to make sure you’ve run `library("readxl")` to load the package with this function). The helpfile provides useful information for running the function, and one of the most useful parts is the “Examples” section. Scroll down to the bottom of the helpfile to find this section. It includes several examples that you can copy into your R script or console and try yourself, to figure out the types of inputs that the function needs and how different options for the function modify how it works.

Modular, not monolithic

Another key thing to remember as you develop R scripts is that they are all, even those that do very complex things, made up of lots of smaller, simpler pieces. R scripts are, in other words, modular, not monolithic.

As you develop a script, plan to create it by chaining together simpler steps. You can start by trying to map out the key things that you’ll need to do in the pipeline, and then you can dive into each of those sections and start writing the code to achieve that step. For example, if you are preprocessing data that measured bacterial growth rates, some of the broad sections you may have in your pipeline are ones to read in the data, to convert it into a tidy data format so it’s easier to work with, to determine the time range in the experiment when each sample was in its exponential growth phase, to use data in that time range to estimate the growth rate for each sample, and so on. Start by dividing your script into these sections (you can mark each off with a code comment—how to use these are described later in this section), and then you can start to develop the code, starting with the code to input the data.

Thinking of a code script as something that is modular is critical as you start learning how to code in a language like R. By dividing the full pipeline into smaller steps, you can tackle it one piece at a time. You can look for resources that are linked to a specific part of the problem, rather than feeling like you need to find other examples of full, start-to-finish pipelines that have done the same thing that you’re trying to do. For example, the step of reading in data that you recorded in an Excel spreadsheet comes down to a simple task—reading data into R from an Excel file. It doesn’t matter whether that file has data on bacterial growth rates or on mice weights or on anything else—the process of reading it into R is the same. Therefore, when you’re working on this step, you can google help on reading Excel files into R if you’re stuck, without worrying about whether the resources that you find to help used examples of the type of data you have or other types of data.

Iterate!

Do not repeat yourself

As you become more familiar with programming with R, you can start to evolve your style of writing scripts in more advanced ways. A key one is to learn how to limit how often you repeat the same code. As you write data preprocessing pipelines, you'll find that you often need to do the same thing, or variations on the same thing, over and over. For example, you may need to read in and clean several files of the same type and structure. You will likely, at first at least, find yourself copying and pasting the same code to several parts of your script, with only minor changes to that code (e.g., changing the R object that you input each time).

Don't worry too much about this as you start to learn how to write R scripts. This is a normal part of the drafting process. However, as you get better at using R, you'll want to learn techniques that can help you avoid this repetition.

There are a few reasons that you'll want to avoid repetition in your code when possible. First, these repeated copies of the same or similar code will make your code script much longer and harder to read through later to figure out what you did. Second, it is hard to keep these copies of code in sync with each other. For example, if you have several copies of the code you use to check for outliers in your data, and you decide you want to change how you are doing that, you'll need to find every copy of the code in your script and make sure you make the same change in each place. Instead, if you have less repetition in your code, then you can make the change in a single place and ensure that the change will be in place everywhere you are doing that process.

There are a few tools that are useful to develop to help avoid repetition. The first is to learn how to write your own R functions. Any R user can write a new function, and you can write these for your personal use, in addition to writing them in packages you plan to share with others. When you wrap a function, it encapsulates the code for something that you need to do, and it allows you to do that thing anywhere else in your code just by calling that new function, rather than copying all the lines of the original code. This is an excellent way to write the code you need to use often in one place, rather than copying and pasting the same code throughout your R script.

Since you need to run the code that defines the function before you use it, it often makes sense to write any code that creates these functions near the top of your code script. If you find that you've written a lot of functions, or that you've written functions that you'd like to use in more than one of your data preprocessing scripts, you can even save the code that creates the functions in a separate R script and just source that separate script at the top of each script that uses the function, using the source call (and eventually you could even think of creating your own package with those functions).

There is one other excellent set of tool for avoiding repetition that we want to mention. Again, it is likely more complex than what you'll want to start off with as you learn to write R scripts, but once you are comfortable with

the basics, it's a powerful tool for creating code scripts that are as short and simple as possible while doing very powerful things. This set of tools all focus on iteration. They include `for` loops, which allow you to step through elements in a data structure and apply the same code to each. They also include a set of tools in the `purrr` library that allow you to apply the same code, through a function, to each element in a larger data structure. These are excellent tools when you are doing something like reading in a lot of similar files and combining them into a single R object for preprocessing.

We will not go into details about how to write R functions or these iteration tools in these modules, as our aim here is to get you started and give you an overview of where you might want to go next. If you do want to learn to write your own R functions, there's a chapter describing the process in the free online book "R for Data Science" with guidance on this topic (<https://r4ds.had.co.nz/functions.html>). If you'd like to learn more about tools for iteration, the same book also has a chapter on that (<https://r4ds.had.co.nz/iteration.html>).

Scripts are for humans to read, not just computers

Some things belong in the console, not the script

Finally, keep in mind that all of the code you write, as you develop a script with a pipeline, does *not* need to be recorded in the script. Of course you will want to include all the code that is necessary for the script as a whole to work on its own. However, as you develop code, you'll take some steps to explore your data or to do things like installing packages that you don't have yet.

For example, as you work on your code, you'll likely want to look at the contents of your R objects as you work on them. If you have read in your data from a file into an R object, you'll want to look and make sure it looks like it read in correctly. If you have created a new object that summarizes the original data by taking the average of each group, you'll probably want to look at that as you develop the code to create it, again to check and explore as you build the code. To do this, you can call the object's name (e.g., type `my_data` and run it) or use functions like `head`, which prints out the first few rows or items of the object (e.g., `head(my_data)`), `tail`, which prints out the last few rows or items of the object, or `str`, which summarizes the structure and contents of the object.

All of these are useful to run as you draft and edit your code, but calls like this that print out pieces of the data to check aren't necessary in the final R script (and in fact can make it messier than it needs to be and result in a lot of extra print-out at the console when you run the code as a batch once you've finalized it). As you develop your code, then, try to get in the habit of not writing these types of exploratory function calls in the script you're developing. Instead, write them directly in the console and run them from there.

[Figure—writing exploratory code in the console versus the R script. Same for installing packages]

The other piece of code that you should run in the console rather than

saving in the R script is code to install new packages. Since you only need to install a package once (until you get a different computer or update base R), you don't need to run `install.package` function calls everytime you run the code in a script. Including these functions in the script will therefore just slow it down. Instead, go to your console directly to write the function calls to install new packages (or, if you prefer, in RStudio you can go to Tools on the menu bar and select Install Packages).

The final goal is to develop an R script that has everything it needs to run the full pipeline from in a fresh R session. In other words, if it uses functions from packages, it will include the code to load those packages (library function calls). For every R object that it uses, there will be code that creates that object in the script. It will not include, however, extra pieces of code that were used to explore the objects as you built the code. To test that the R script can run in a fresh R session, you can close R and reopen it (make sure that you've set your global options in R to never save the workspace to `.Rdata` on exit, to not restore the `.Rdata` into the workspace on exit, and to not save the history, all of which you can set by going to the RStudio Tools menu item and selecting Global Options). When you reopen R, there will not be any objects in the environment, and there will not be a history of any of the previous function calls that you ran. Try running the code in the script in this fresh environment and make sure that you weren't relying on anything that you did outside the script to make the code work. If the code can run in a fresh environment like this, then any R user should be able to rerun everything in the script themselves (although they may need to install a few new libraries first if they don't have all the required libraries yet).

3.2.7 *Compiled versus interpreted programming languages*

When computers were first being developed, they were very tricky to program, as they required humans to translate appropriate logic down to a very granular level that the computers of the time could process. As computer development continued, development of programming techniques and languages developed as well. These evolved to allow a programmer to write at a level of logic that is more straightforward for humans, and then the inner design of the programming language did the work of translating those instructions for the computer.

One key development in programming languages was the development of *interpreted* programming languages. These are in contrast to a type of programming languages called *compiled languages*. With compiled languages, you must write the full set of instructions for the computer to run. This full set of instructions is then sent through a programmer called a *compiler*, which translates the instructions for the computer, and then the program can be run, either once or repeatedly. By contrast, interpreted languages do this type of compiling

(translating for the computer) “on the fly”, and so they allow you to run each step of the instructions as you write them, and then check the output a step at a time.

It may be easier to understand this difference with an analogy, so we’ll make a comparison with teaching someone how to cook a recipe. With an interpreted language, it is as if you are in the kitchen with the person you are teaching. You can tell them to do the first step (“chop the onion into small dice”). Then, you can take a look at the result. If you don’t like it (“those dice aren’t small enough—make them smaller”), you can give a new instruction. You can work through the entire recipe like this, checking and adjusting as you go. By contrast, with a compiled language, it is as if you have to write down the whole recipe and mail it off to someone in a different city, and then hope it all works okay.

Compiled languages have a number of advantages—speed of running the code being a key one—that mean they are still widely used. However, interpreted languages are much easier for a new programmer to learn, as they allow this process of checking and adjusting, really allowing someone to see what’s going on with each thing they ask the computer to do. Interpreted languages are often now taught as a programmer’s first language, with Python as a particularly popular first language. Other interpreted languages include Julia and R, with R being particularly popular for data science in general and for bioinformatics and other biological research in particular.

3.2.8 Code scripts versus interactive coding

When you use an interactive programming language, like R, you will likely start to explore your data by working interactively, running one call, looking at the results, and then running the next call, adapting as needed based on the results you see at each step.

3.2.9 Process of building a code script

A code script is essentially a recipe for cleaning and analyzing data.

3.2.10 Section

“Every maker needs to give themselves the space to screw up in the pursuit of perfecting a new skill or in learning something they’ve never tried before. Screwing up IS learning.” (Savage, 2020)

“Mistake tolerance is particularly valuable in this aspect of the creative process. When you know what you want to make, but you’re not exactly sure what it should look like or how it should operate, you need to give yourself permission to experiment, to iterate your way there. That’s not just how you get to what you want, it’s how you get good at it. You have to do it over and over and over again. Anticipating mistakes is how you put space around the unfamiliar and the unknown.” (Savage, 2020)

"If you expect to nail it the first go round every time you build something new—or worse, you demand it of yourself and you punish yourself when you come up short—you will never be happy with what you make and making will never make you happy". (Savage, 2020)

BioC Conference

"Just as writing a book involves an outline and a rough draft (so many drafts!), which get polished into a final manuscript, making things often benefits from a preliminary stage where the big details get worked out, and then a final fabrication stage where the small details get worked out. Cardboard is a low threshold material that can make discussion of ideas at the preliminary stage so much easier and more complete." (Savage, 2020)

"In my professional life, I have worked with every conceivable type of client and collaborator, from those who were makers with the same or greater expertise as me, who understood deeply what I was talking about when we discussed a build, to clients who couldn't glue two blocks of wood together if you put the blocks in their hands, covered with glue, and told them to clap. Being able to communicate your ideas to clients and collaborators is one of the most important skills to possess as a maker, otherwise some of your projects may never get off the ground." (Savage, 2020)

"To make anything, it's critical to have a physical understanding of how all the component parts of your project will fit together." (Savage, 2020)

"What material can you wrap your arms around to gain a complete sense for the skills you want to master and the objects you want to make? Is it cardboard? Muslin fabric? Crappy butcher cuts? Scrap wood? The backside of recycled printer paper? A word processor? It really doesn't matter as long as it allows you to be messy and it keeps you moving forward in your journey as a maker." (Savage, 2020)

"Once you have a new tool you thing you need, spend some time getting to know it physically. With certain tools, I'll go so far as to take them apart, just to understand them better, inside and out. ... If you are unfamiliar with a tool or inexperienced with the techniques required to use it, getting comfortable like this is the most important thing you can do, because you might really need this thing, but if you are intimidated by it, you aren't going to want to use it, and then what's the point?" (Savage, 2020)

"If you've never used a tool before, reviews and articles about it can only get you so far. You need to work with a tool in order to see how the tool works for YOU. You need on-the-ground experience with it in your hands." (Savage, 2020)

With open-source, you can think of it as builing a collection of different tools, rather than a black box. The tools can come from different companies (like real tools)—you don't have to limit to consuming the tools created by a single producer (as with most proprietary software systems).

For free and open-source software, you don't have to invest money to get more tools. Instead, the investment for each new tool is the time that it takes to learn how to use it in your workflow. This includes several elements. You'll

need to understand what primary input is used by the function, both in terms of the input's conceptual content and the format or structure in which those data are stored. You'll need to understand the content and format of the output of the function in a similar way, so that you can join it with other functions in your workflow. You'll want to make sure you understand the main choices that you can modify with the function, through setting different parameters, as well as the reason behind the defaults that are used for those parameters. Finally, ideally you'll want to understand a bit about how the function operates to move from the input you give it to the output it gives back to you.

"'Freemon Dyson, a famous physicist, suggested that science moves forward by inventing new tools,' Kevin [Kelly, founding editor of Wired magazine] began as we talked on the phone one morning about tools. 'When we invented the telescope, suddenly we had astrophysicists, and astronomy, and we moved forward. The invention of the microscope opened up the small world of biology to us. In a broad sense, science moves forward by inventing tools, because when you have those tools they give you a new way of thinking.' (Savage, 2020)

"My initial reaction was shock—at both the ingenuity of the solution and the fact that I'd forgotten all about it—but that quickly resolved into gratitude. I was grateful to myself for taking the time a year before to save me this time now. 'Thank you, me-from-the-past!' I literally said to myself." (Savage, 2020)

"Complex equipment breaks, and when it breaks you need the technical expertise to fix it, and you need replacement parts. ... Some studies suggest that as much as 95 percent of medical technology donated to developing countries breaks within the first five years of use." (Johnson, 2011)

"Good ideas ... are, inevitably, constrained by the parts and skills that surround them. We have a natural tendance to romanticize breathrough innovations, imagining momentous ideas transcending their surroundings, a gifted mind somehow seeing over the detritus of old ideas and ossified tradition. But ideas are works of bricolage; they're built out of that detritus. We take the ideas we've inherited or that we've stumbled across, and we jigger them together into some new shape." (Johnson, 2011)

"Good ideas are not conjured out of thin air; they are built out of a collection of existing parts, the composition of which expands (and, occasionally, contracts) over time. Some of these parts are conceptual: ways of solving problems, or new definitions of what constitutes a problem in the first place. Some of them are, literally, mechanical parts." (Johnson, 2011)

"What kind of environment creates good ideas? The simplest way to answer is this: innovative environments are better at helping their inhabitants explore the adjacant possible, because they expose a wide and diverse sample of spare parts—mechanical or conceptual—and they encourage novel ways of recombining those parts. Environments that block or limit new combinations—by punishing experimentation, by obscuring certain branches of possibility, by making the current state so satisfying that no one bothers to explore the edges—will, on average, generate and circulate fewer innovations than environments that encourage exploration." (Johnson, 2011)

“Part of coming up with a good idea is discovering what those spare parts are, and ensuring that you’re not just recycling the same old ingredients. ... The trick to having good ideas is not to sit around in glorious isolation and try to think big thoughts. The trick is to get more parts on the table.” (Johnson, 2011)

3.3 Simplify scripted pre-processing through R’s ‘tidyverse’ tools

The R programming language now includes a collection of ‘tidyverse’ extension packages that enable user-friendly yet powerful work with experimental data, including pre-processing and exploratory visualizations. The principle behind the ‘tidyverse’ is that a collection of simple, general tools can be joined together to solve complex problems, as long as a consistent format is used for the input and output of each tool (the ‘tidy’ data format taught in other modules). In this module, we will explain why this ‘tidyverse’ system is so powerful and how it can be leveraged within biomedical research, especially for reproducibly pre-processing experimental data.

Objectives. After this module, the trainee will be able to:

- Define R’s ‘tidyverse’ system
- Explain how the ‘tidyverse’ collection of packages can be both user-friendly and powerful in solving many complex tasks with data
- Describe the difference between base R and R’s ‘tidyverse’.

3.3.1 What are data structures / containers?

Data types versus data structures

You can think of *data structures* as containers that hold your data in R, holding it in a way that lets you access and work with the data.

One important distinction is between data structures and data types. A *data type* refers to the characteristic of the data. Is it a date, for example, or a number, or a character string? R can do different types of things with different types of data—for example, R can add together two pieces of data that are numbers, while for two pieces of data that are dates, it can tell which is the later date. For character strings, R can look for patterns in the string (for example, does it include any capital letters? Does it start with “b”?). All pieces of data are, at the deepest level, stored as a string of 0s and 1s. By assigning a data type like “character string” or “numeric” to each piece of data, R can make more sense of each piece of data in terms of what operations are reasonable to perform on the data, helping to translate those 0s and 1s into something more meaningful.

In R, your data can be stored as different types of data: whole numbers can be stored as an *integer* data type, continuous [?] numbers through a few types of *floating* data types, character strings as a *character* data type, and logical data (which can only take the two values of “TRUE” and “FALSE”) as a *logical* data type. More complex data types can be built using these—for example, there’s a

special data type for storing dates that's based on a combination of an [integer?] data type, with added information counting the number of days [?] from a set starting date (called the [Unix epoch?]), January 1, 1970. (This set-up for storing dates allows them to be printed to look like dates, rather than numbers, but at the same time allows them to be manipulated through operations like finding out which date comes earliest in a set, determining the number of days between two dates, and so on.) R uses these different data types for several reasons. First, by using different data types, R can improve its efficiency [?] in storing data. Each piece of data must—as you go deep in the heart of how the computer works—as a series of binary digits (0s and 1s). Some types of data can be stored using fewer of these *bits* (binary digits). Each measurement of logical data, for example, can be stored in a single bit, since it only can take one of two values (0 or 1, for FALSE and TRUE, respectively). For character strings, these can be divided into each character in the string for storage (for example, “cat” can be stored as “c”, “a”, “t”). There is a set of characters called the ASCII character set that includes the lowercase and uppercase of the letters and punctuation sets that you see on a standard US keyboard [?], and if the character strings only use these characters, they can be stored in [x] bits per character. For numeric data types, integers can typically be stores in [x] bits per number, while continuous [?] numbers, stored in single or double floating point notation [?], are stored in [x] and [x] bits respectively. When R stores data in specific types, it can be more memory efficient by packing the types of data that can be stored in less space (like logical data) into very compact structures.

Data structures, on the other hand, allow you to keep together, as well as refer to, data that you have loaded into R. A single object can contain pieces of data with different data types, and the object's data structure defines how all the pieces of data in the object are organized and how you can access and work with the data in the structure. For example, one of the simplest data structures in R is the vector—this data structure requires that all the data stored in it have the same data type (e.g., all be numeric), and it holds together a one-dimensional string of pieces of data of that type. For example, a vector of the numbers one to five would be the string of those numbers—1, 2, 3, 4, 5—while a vector of the letters a to e would be the string of data with the “character” type—“a”, “b”, “c”, “d”, “e”.

One of the “building block” data structures in R is the vector. This data structure is one dimensional and can only contain data that have the same data type—you can think of this as a bead string of values, each of the same type. For example, you could have a vector that gives a series of names of study sites (each a character string), or a vector that gives the dates of time points in a study (each a date data type), or a vector that gives the weights of mice in a study (each a numeric data type). You cannot, however, have a vector that includes some study site names and then some dates and then some weights, since these should be in different data types. Further, you can't arrange the data in any structure except a straight, one-dimensional series if you are using

a vector. The dataframe structure provides a bit more flexibility—you can expand into two dimensions, rather than one, and you can have different data types in different columns of the dataframe (although each column must itself have a single data type).

A second key data structure in R, the dataframe, provides a structure that stores one or more of these vectors, and so it allows you to store data with different types in the same object. For example, you could create an object with a dataframe structure that contains one column with a vector of the numbers one to five and another with the letters a through e. This structure would look something like this:

```
##   numbers letters
## 1      1      a
## 2      2      b
## 3      3      c
## 4      4      d
## 5      5      e
```

While the dataframe structure can combine vectors with data in different types (in the example, the numbers column has a numeric data type and the letters column has a character data type), it does have a rule for combining different vectors. They all must be the same length. This means that the dataframe structure is always rectangular, with each column having the same number of rows. In the example above, both the numbers and letters columns are vectors with five values, so the dataframe ends up having two columns and five rows.

The dataframe object type is a very basic two-dimensional format for storing data in R. When you print it out, it will remind you of looking at data in a spreadsheet. The two dimensions—rows and columns—allow you to include data for one or more observations, with different values that were measured for each. For example, if you were conducting a study of children's BMI and blood sugar, you might have an observation for each child in the study, and values measured for each child of height, weight, a blood sugar measure, study ID, and date of the observation.

The two-dimensional structure of a dataframe keeps the values measured for each observation lined up with each other, and lets you keep them aligned as you work with the data. You could also store data for each value as separate objects, in one-dimensional vectors, which you can visualize as strings of values of the same data type, like the dates that each observation was made, or the weight of each study subject. However, when the data is in separate vectors, it is easy to make coding mistakes, and coding is often less efficient. If you want to remove one observation, for example, because you find it is a duplicate, you would need to carefully make sure you remove it correctly from each vector. When data are stored in a dataframe, you can remove the row for that observation with one command, and you can be sure that you've removed the

value you meant to from each of the measured values.

To be able to understand some key differences in the Bioconductor approach and the tidyverse approach, you first need to understand how programming uses **data structures** to store data, and that there can be numerous different data structures available within a programming language to handle different types of data.

When you process data using a programming language, there will be different structures that you can use to store data as you work with it. You can think of these data structures as containers where you keep your data in the programming environment while you work with it, and different structures organize the data in different ways.

If you've read the earlier modules, you've already seen one example of a data structure. In other modules, we've discussed the "tidyverse" approach to processing data in R—this approach emphasizes the *dataframe* as a way to store data while you're working with it (in other words, a data structure). In fact, the use of the *dataframe* as data structure for data storage is one of the defining features of the "tidyverse" approach. We mentioned in earlier modules that the tidyverse approach is based on using a common interface, so that you can mix and match small functions in different ways—the common interface is the *dataframe*. The tidyverse approach is built on the use of a common structure for storing data, the *dataframe*—almost all functions take data in this structure and almost all return data in this structure.

Figure 3.1 shows an annotated example of a *dataframe*, highlighting some of the key elements of its structure. A *dataframe* stores data in a two-dimensional structure, combining rows and columns. Each column is constrained to have data of the same type—in other words, all values in a column could be numeric (e.g., 1, 4, 10), or all could be character strings (e.g., "Mouse 1", "Mouse 3"), but the same column cannot combine some values that are numeric and some that are character strings. Across the *dataframe*, all columns must have the same length (i.e., if you printed out the full *dataframe*, it would look like a rectangle). All the column values should be lined up, so that as you are reading across a row, the values in the column cells are from the same observation or unit.

Common R data structures

We just covered two of the most common data structures in R: the vector and the *dataframe*. You will come across a number of other structures as you work in R.

One other very common data structure is the list. This is a very flexible data structure, and it allows enormous flexibility in collecting other types of data structures (including other lists) into a single R object.

In addition to *dataframes*, there are a number of other simple, general purpose data structures that are often used to store data in R, and that you're likely to come across as you work in R. These include **vectors**, which are used to store one-dimensional strings of data of a single type (e.g., all numeric, or

gene <chr>	sample <chr>	sample.id <fctr>	num.tech.reps <dbl>	protocol <fctr>
ENSRNOG000000000001	SRX020102	SRX020102	1	control
ENSRNOG000000000007	SRX020102	SRX020102	1	control
ENSRNOG000000000008	SRX020102	SRX020102	1	control
ENSRNOG000000000009	SRX020102	SRX020102	1	control
ENSRNOG000000000010	SRX020102	SRX020102	1	control
ENSRNOG000000000012	SRX020102	SRX020102	1	control
ENSRNOG000000000014	SRX020102	SRX020102	1	control
ENSRNOG000000000017	SRX020102	SRX020102	1	control
ENSRNOG000000000021	SRX020102	SRX020102	1	control
ENSRNOG000000000024	SRX020102	SRX020102	1	control

all character strings; as a note, you can think of each column in a dataframe as a vector), **matrices**, which are also used to store data of a single type, but with a two-dimensional structure, and **arrays**, which, like matrices and vectors, store data of a single type, but in three dimensions. Another common general purpose data structure in R is the *list*, which allows you to combine data stored in any type of structure to create a single R object, giving enormous flexibility (but minimal set structure from one object to another). This data structure is the building block for some of the more complex specific data structures, which we'll cover next. The list structure in R has enormous flexibility in terms of storing lots of data in lots of possible places. This data can have different types and even different substructures. Some data structures in R are very constrained in what type of data they can store and what structure they use to store it.

There are a number of simple, general purpose data structures that are often used to store data in R. These include **vectors**, which are used to store one-dimensional strings of data of a single type (e.g., all numeric, or all character strings), **matrices**, which are also used to store data of a single type, but with a two-dimensional structure, and **dataframes**, which are used to store multiple vectors of the same length, and so allow for storing measurements of different data types for multiple observations.

[Figure: examples of these three structures]

Other data structures are more customized for specific types of data. For example, there are a number of specialized data structures that are commonly used in Bioconductor packages to store specific types of genomic [?] data. These structures have been specially designed to meaningfully arrange the types of data that are commonly generated in certain types of experiments [?]. While the common types of data structures the we mentioned before (vectors, dataframes, and lists) are all provided as part of base R, many of these more specialized data structures are defined in R extensions (packages that you install

Figure 3.1: An example of the dataframe data structure. This data structure is the most frequently used data structure within the tidyverse approach, and its use is in fact a defining element of the approach.

once you've installed base R), and so you cannot access those data structures until you have installed additional packages.

In a later module, we will go into more depth about some of these specialized data structures in R, and how in certain cases they are powerful tools for complex analysis or for working with complex data.

Link between functions and object classes

In a language like R, you will find that data structures can be closely tied to data structures and data types. In many cases, a function is designed to only work with data that is of a certain type or that is stored in a certain data structure.

For example, the package named `lubridate` provides helpful functions for working with data that represent dates. Most of the functions in this package will only work on a vector (the data structure) that contains pieces of data that are dates (the data type). There are functions in this package that can do things like extract the year, month, or day of month from a date, but only if you input a vector of data in with the date data type.

Many other functions require that you input a dataframe. For example, there are functions in the `dplyr` package that allow you to extract a subset of a dataframe—for example, to extract a smaller dataframe with only certain rows or certain columns from the original. These functions require that you input data in a dataframe structure.

For more complex or customized data structures, like the data structures within the Bioconductor project, there will often be a whole suite of customized data structures, and functions associated with those data structures, that are used during a pipeline of analyzing data from a certain type of experiment. For example, there are data structures, and functions associated with those structures, that are customized to work with flow cytometry data, and other collections of data structures and functions for working with metabolomics data, and so on.

As a note, there are a few functions that are “generics”—they have been coded in such a way that they will work with a variety of data structures. For these functions, they will often output different things depending on the type of data structure that is input when the function is called. For example, the `summary` function is a generic function—you can input objects with a variety of data structures and the function will work, providing some type of summary of the object in each case. If you input an object with a vector data structure, the `summary` function will provide a summary of the contents of that vector—if the data type is numeric, it will give values like the median, the range, and the number of missing values, while if the data type is logical (TRUE or FALSE for each data piece), it will give a count of the number of TRUE and FALSE values in the vector. If you input an object instead that is a dataframe, the `summary` function will output a summary of each of the columns in the dataframe. Although there are such generic functions that work with different data structures, the general rule in R is that a function is typically designed to work with a certain data

structure (in fact, the generic functions are coded to have different functions that work with each type of data structure, and so a structure-specific function is called “under the hood” once one of the generic functions is called).

“Tidy” data and data structures

In this module, we aim to introduce you to something called the “tidyverse” approach to programming in R. This is a powerful approach, and we will detail it more extensively in the next section. Here, we want to introduce a key element of the approach—it is based on storing data in a single data structure throughout your pipeline of code. Specifically, it focuses on tools and techniques that work when you use a dataframe structure to store data as you work with the data, with a set of functions that both input and output objects that use the dataframe structure (as well as functions that input and output vectors, which are used to perform operations on the data in the columns within the dataframe—if you recall, each column in a dataframe has a vector data structure).

The tidyverse approach requires one step beyond the data structure, and that is how the data are arranged within that structure. You can store data in a dataframe structure as long as you can put it in two dimensions (rows and columns), with the same type of data in each column. With the same set of data, there are often many different arrangements you could make that satisfy these constraints. The “tidy” part of the “tidy data” structure—which is at the heart of the tidyverse approach—are a set of standards describing exactly how you arrange the data within the dataframe structure.

The R object class—dataframe, and more specifically, tibble—of the standard format for data for a tidyverse approach is just the first part of the standard data format for the tidyverse approach. The second part of the standard format is how you organize your data in this format. To easily work with tidyverse functions, you’ll want to make sure that your data is stored within that dataframe following “tidy” data principals. These are fully described in an earlier module in this book [which module]. If you use this data format to initially collect your data, as described in an earlier module, you will find it very easy to read the data into R and work within the tidyverse approach. When working with larger and more complex data collected from laboratory equipment, you may find you need to do some preprocessing of the data using an object-oriented approach before you can move the data into this tidy format, but at that point, you can continue with analysis and visualization of your data using a tidyverse approach.

The tibble data structure

As a final note, there is a specialized data structure that is often associated with the tidyverse approach. It is called the “tibble”, and it is a specialized version of the dataframe. What do we mean by it being a “specialized” version of a more common data structure? This means that there are a few functions that will give different results if the data are stored in a tibble structure rather than the more generic dataframe structure. However, if a function does not have a

specialized method for the tibble structure (and most functions don't), then the function will treat the object as a regular data frame. Some of the few functions that have specialized methods for tibbles include the `print` function, which is also the default function that is run if you just type an object's name at the console and press "Enter". The `print` function, when called with an object in the tibble structure, will only print out the first few rows (whereas, with a generic data frame, it will print out all rows, sometimes resulting in a very long print-out). Similarly, if there are many columns, it will only print out a certain number and just provide summaries for the rest (again, with a generic data frame, everything would be printed). It also provides some nice summaries of the data frame as a whole, as well as of the type of data in each column. Overall, the `print` method for a tibble structure provides a clearer and typically more useful overview of the data in the object than does the `print` method for a generic data frame structure. All this being said, as you first begin to work in the tidyverse approach, you often may not notice whether your data is stored in a more generic data frame structure or the more specialized tibble version, and it often won't matter much which of the two structures is being used, since the tibble structure in most cases (i.e., for most functions) is just treated as the more generic data frame structure.

Sometimes, you'll see that data in a tidyverse approach are stored in a special type of data frame called a "tibble"—this isn't very different from a data frame, and in fact is a special type of data frame. It's only differences in practice are that it has a slightly different `print` method. The `print` method is the method that's run, by default, when you just type the R object's name at the console. A tibble prints more nicely than a basic data frame. By default, it will only print the first few lines. By contrast, a data frame will, by default, print everything—if you have a lot of data, this can create an overwhelming amount of output when you just want to check out what the data looks like. The printout of a tibble will also include some interesting annotations to help you see what's in the data, including the dimensions of the full data frame and the data type of each column in the data.

Sometimes, you'll see that data in a tidyverse approach are stored in a special type of data frame called a "tibble"—this isn't very different from a data frame, and in fact is a special type of data frame. It's only differences in practice are that it has a slightly different `print` method. The `print` method is the method that's run, by default, when you just type the R object's name at the console. A tibble prints more nicely than a basic data frame. By default, it will only print the first few lines. By contrast, a data frame will, by default, print everything—if you have a lot of data, this can create an overwhelming amount of output when you just want to check out what the data looks like. The printout of a tibble will also include some interesting annotations to help you see what's in the data, including the dimensions of the full data frame and the data type of each column in the data.

3.3.2 An overview of the “tidyverse” approach

Once data are in the “tidy” data format, you can create a pipeline of code that uses small tools, each of which does one simple thing, to work with the data. This work can include cleaning the data, adding values that are functions of the original values for each observation (e.g., adding a column with BMI based on values for each observation on height and weight), applying statistical models to test hypotheses, summarizing data to create tables, and visualizing the data.

The “tidyverse” approach is an approach to using R that has grown enormously in popularity in recent years. Most R courses and workshops for beginning programmers are now structured around this approach. It provides a powerful yet flexible approach for working with data in R, and it is one of the easier ways to start learning R. In this section, we’ll cover the philosophy that provides a framework for this approach. In the following sections of the modules, we’ll go deeper into specific tools under this approach that can be used for common data preprocessing tasks when working with biomedical data, as well as provide information on more resources that can be used to continue learning this approach.

The term “elegance” often captures styles and approaches that are beautiful and functional without unneeded extras or complexity. Engineers and scientists sometimes use this term to capture approaches that achieve a desired result with minimal complexity and friction. A coding problem, for example, could be solved by an average coder with a hundred lines of code that get the job done, but a very good coder might be able to solve the same problem with five lines of code that are easy to follow. The second approach would be applauded as the “elegant” solution. In mathematics, similarly, proofs can be complex and unwieldy, or they can be simple and elegant—this idea was beautifully captured by the Hungarian mathematician Paul Erdos, who famously described very elegant mathematical proofs as being from “The Book”—that is, God’s own version of the proof of the mathematical idea.

“Paul Erdos liked to talk about The Book, in which God maintains the perfect proofs for mathematical theorems, following the dictum of G. H. Hardy that there is no permanent place for ugly mathematics. Erdos also said that you need not believe in God but, as a mathematician, you should believe in The Book.” [Proofs from the Book, Third Edition, Preface]

The “tidyverse” approach in R is elegant. It is powerful, and gives you immense flexibility once you’ve gotten the hang of it, but it’s also so straightforward that the basics can be quickly taught to and applied by beginning coders. It focuses on keeping data in a simple, standard format called “tidy” dataframes. By keeping data in this format while working with it, common tools can be applied that work with the data at any stage of a “tidy” coding pipeline. These tools take a “tidy” dataframe as their input, and they also output a “tidy” dataframe, with whatever change the function implements applied. Because each of these “tidyverse” tools input and output data in the same standard

format, they can be strung together in order you want. By contrast, when functions input and output data in different object types, they can only be joined in a specified order, because you can only apply certain functions to certain object types.

Since the “tidyverse” tools can be strung together in any order, they can be used very flexibly to build up to do interesting tasks. The tidyverse tools generally each do very small and simple things. For example, one function (`select`) just limits the data to a subset of its original columns; another (`mutate`) adds or changes values in columns of the dataset, while another (`distinct`) limits the dataframe to remove any rows that are duplicates. These small, simple steps can be combined together in different patterns to add up to complex operations on the data, while keeping each step very simple and clear. Since the data stays in a standard and simple object type, it is easy to check in on your data at any stage, as the common visualization tools for this approach (from the `ggplot2` package and its extensions) can be always be applied to data stored in a tidy dataframe.

Uses the same data structure throughout

The centralizing principal of the tidyverse approach is the format in which data is stored throughout “tidyverse” coding—the tidy dataframe. Briefly, you can think of this format in two parts. First, there’s the R object type that the data should be stored in—a basic “dataframe” object.

As we mentioned in the last section, the tidyverse approach hinges on using the same data structure throughout your coding pipeline—specifically, the dataframe structure (or its more specialized version, the tibble structure). By insisting on the same data structure throughout, this approach is able to offer small functions that can be chained together to solve complex problems.

This idea rests on the idea of the power of modularity. You can think of this in terms of children’s toys—building bricks like Legos are powerful because they are modular, while a toy like a stuffed animal is not. Each individual Lego is small and simple, and would be pretty boring by itself. However, because the blocks can be combined in different ways, they can be used to create very complex and interesting structures. By contrast, something that is not modular, like a stuffed animal, always retains the same structure. While it might be more interesting and complex to start with than a single Lego block, it will not evolve or contribute to something more interesting.

This modularity works in the same way that it does for Lego bricks. Lego bricks can be combined in interesting ways because they all take the same input and give the same output—the shape of the holes on the bottom of each brick accept the shape of the pins [?] at the top of each brick, so they can be put together in essentially infinite combinations. The tidyverse approach in R works in a similar way—the functions in this approach almost all input data that are in a dataframe structure (or in a vector structure, for functions that operate on columns in the dataframe) and they almost all output data in the same structure that they input it. As a result, the functions can be chained

together in interesting ways, where the output of one function can feed directly into the input of another.

Small, simple tools

Since the functions in the tidyverse approach are designed to work in a modular way—in other words, to be combined in interesting ways to solve larger problems, rather than solving a large problem with a single function—each of the functions tends to be a small, simple tool. In other words, each function tends to do one thing simply but well. This makes it fairly easy to start learning the tidyverse approach, as each of the functions that you learn as you begin does one thing that is fairly straightforward. As you get better and better at coding using this approach, you often find that it is not because you are using more complex functions, but rather that you’re becoming more clever at combining sets of simple functions in interesting ways.

Functions available through packages

The tidyverse functions do not come with base R, but rather are available through extensions to base R, commonly referred to as “packages”. Like base R, these are all open-source and free. Many are available through a repository called CRAN, and you can download them directly from R using the `install.packages` function.

The heart of the tidyverse functions are available through an umbrella package called “tidyverse”. This package includes a number of key tidyverse packages (e.g., “dplyr”, “tidyr”, “stringr”, “forcats” [?], “ggplot2”) and allows you to quickly install this set of packages on your computer. When you are coding in R, you will then need to load the package in your R session, which you can do using the `library` call (e.g., `library("tidyverse")`).

In addition to the packages that come with the umbrella “tidyverse” package, there are numerous other packages that build on the tidyverse approach. Some are created by the creator of the tidyverse approach (Hadley Wickham) or others on his team, while others are created by other R programmers but follow the standards of the tidyverse approach. Some of the most helpful of these for working with biomedical data include ...

3.3.3 Useful tidyverse tools for data preprocessing

As you learn to code, a good strategy is to start collecting “tools” for your “toolbox” in R—functions that you have learned to use very well and that you understand thoroughly. This will help make you proficient in R more quickly, and it will also limit the chance of bugs and errors in your code, making your data work more robust and rigorous. In this section, we’ll cover some tidyverse tools that we have found helpful for preprocessing biological data. These are not exhaustive, but may help you to identify some sets of tools to focus on learning well for data preprocessing and analysis of biological data.

Some key tools from the tidyverse for pre-processing laboratory data:

- Tools to “tidy” data: pivoting, etc. Allows you to design a data collection

template that is more convenient in the lab, but quickly move it into the “tidy” format once you read it into R

- Visualization: Not just for final product, also for exploratory analysis
- Tools to work with dates, character strings (regular expressions), numbers
- Tools to map/apply: Reading in lots of files, doing the same thing to all of them.
- Working with lists, converting from lists to tidy dataframes
- Examples from code in earlier modules

Tools for data input

Tools for standardization and normalization

Tools for working with character strings

Tools for working with dates and times

Tools for statistical modeling

3.3.4 Resources to learn more on tidyverse tools

Here we have introduced the tidyverse approach, as well as covered some key tools within it for biomedical data preprocessing. However, we strongly recommend that you continue to learn more in this approach. In this section, we'll point you to resources that you can use to continue to learn this approach to working with data in R.

The tidyverse approach is now widely taught, both in in-person courses at universities and through a variety of online resources. Since there are so many excellent resources available—many for free—to learn how to code in R using the tidyverse approach, we consider it beyond the scope of these modules to go more deeply into these instructions. However, we do think it is critical that biological researchers learn how to connect this approach to the type of coding that is often necessary for pre-processing large and complex data that is output from laboratory equipment. Through many of the modules in this book, we provide advice on how to make these connections, so that data from different sources—including different types of laboratory equipment and hand-recorded data collected by personnel in the lab, like colony forming units measured from plating samples—can all be connected in a tidyverse pipeline by recording hand-recorded data following a tidy format and by pre-processing data with the aim of moving data toward a tidy dataframe that can be integrated with other “tidy” data for analysis and visualization.

Classes and workshops

Most R programming classes at universities, as well as workshops at conferences and other venues, now focus on the tidyverse approach, especially if they are geared to new R users. An R programming class can be a worthwhile investment of time if this resource is available to you, and if you head a research group and do not have time to take one yourself, you could instead consider encouraging trainees in your research group to take this type of class. Programming in other scripted languages, like Python and Julia, provides sim-

ilar skills, although the collection of extension packages that are available for biomedical data tends to be most extensive for R (at least at this time). Classes in programming languages like Java or C++, on the other hand, would have less immediate relevance for most biologists and other bench scientists, and so if you would like to become better at working with biomedical data, it would be worthwhile to focus on programming languages that are scripted.

[Software Carpentry]

Cheatsheets

For many of the key tidyverse packages, there are two-page “cheatsheets” that have been developed by the package creators to help users learn and remember the functions that are available in the package. These are available here [?].

Each cheatsheet includes numerous working examples. One excellent way to familiarize yourself with the tools in a package, then, is to work through the examples on the cheatsheet one at a time, making sure that you understand the inputs and outputs to the function and how the function has created the output. Once you have worked through a cheatsheet in this way, you can keep it close to your desk to serve as a quick reminder of the names and uses of different functions in the package, until you have used them enough that you don’t need this memory jog.

Online books

There are a number of excellent free online books that are available to help you learn more about R (many of which can also be purchased as a hard copy, if you prefer that format). These typically include lots of examples of code that help you try out concepts as you learn them. Two excellent books for biomedical data preprocessing and analysis are *R for Data Science* by ... and *Modern Statistics for Modern Biology* by ...

R for Data Science, which is available at ..., covers ...

One key resource for learning the tidyverse approach for R is the book *R for Data Science* by Hadley Wickham (the primary developer of the tidyverse) and Garrett Grolemund. This book is available as a print edition through O'Reilly Media. It is also freely available online at <https://r4ds.had.co.nz/>. This book is geared to beginners in R, moving through to get readers to an intermediate stage of coding expertise, which is a level that will allow most scientific researchers to powerfully work with their experimental data. The book includes exercises for practicing the concepts, and a separate online book is available with solutions for the exercises (<https://jrnold.github.io/r4ds-exercise-solutions/>).

Modern Statistics for Modern Biology, which is available at ..., covers ...

3.3.5 Subsection 1

"There is a now-old trope in the world of programming. It's called the 'worse is better' debate; it seeks to explain why the Unix operating systems (which include Mac OS X these days), made up of so many little interchangeable parts, were so much more successful in the marketplace than LISP systems, which were ideologically pure, based on a single language (again, LISP), which itself was exceptionally simple, a favorite of 'serious' hackers everywhere. It's too complex to rehash here, but one of the ideas inherent within 'worse is better' is that systems made up of many simple pieces that can be joined together, even if those pieces don't share a consistent interface, are likely to be more successful than systems that are designed with consistency in every regard. And it strikes me that this is a fundamental drama of new technologies. Unix beat out the LISP machines. If you consider mobile handsets, many of which run descendants of Unix (iOS and Android), Unix beat out Windows as well. And HTML5 beat out all of the various initiatives to create a single unified web. It nods to accessibility: it doesn't get in the way of those who want to make something huge and interconnected. But it doesn't enforce; it doesn't seek to change the behavior of page creators in the same way that such lost standards as XHTML 2.0 (which emerged from the offices of the World Wide Web Consortium, and then disappeared under the weight of its own intentions) once did. It's not a bad place to end up. It means that there is no single framework, no set of easy rules to learn, no overarching principles that, once learned, can make the web appear like a golden statue atop a mountain. There are just components: HTML to get the words on the page, forms to get people to write in, videos and images to put up pictures, moving or otherwise, and JavaScript to make everything dance." (Ford, 2014)

"One of the fundamental contributions of the Unix system [is] the idea of a *pipe*. A pipe is a way to connect the output of one program to the input of another program without any temporary file; a *pipeline* is a connection of two or more programs through pipes. ... Any program that reads from a terminal can read from a pipe instead; any program that writes on the terminal can write to a pipe. ... The programs in a pipeline actually run at the same time, not one after another. This means that the programs in a pipeline can be interactive; the kernel looks after whatever scheduling and synchronization is needed to make it all work. As you probably suspect by now, the shell arranges things when you ask for a pipe; the individual programs are oblivious to the redirection." (Kernighan and Pike, 1984)

"Even though the Unix system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is an approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many Unix programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools." (Kernighan and Pike, 1984)

"What is 'Unix'? In the narrowest sense, it is a time-sharing operating system *kernel*: a program that controls the resources of a computer and allocates them among its users. It lets users run their programs; it controls the peripheral devices (disks, terminals, printers, and the like) connected to the machine; and it provides a file system that manages the long-term storage of information such

as programs, data, and documents. In a broader sense, ‘Unix’ is often taken to include not only the kernel, but also essential programs like compilers, editors, command languages, programs for copying and printing files, and so on. Still more broadly, ‘Unix’ may even include programs developed by you or others to be run on your system, such as tools for document preparation, routines for statistical analysis, and graphics packages.” (Kernighan and Pike, 1984)

“A common observation is that more of the data scientist’s time is occupied with data cleaning, manipulation, and ‘munging’ than it is with actual statistical modeling (Rahm and Do, 2000; Dasu and Johnson, 2003). Thus, the development of tools for manipulating and transforming data is necessary for efficient and effective data analysis. One important choice for a data scientist working in R is how data should be structured, particularly the choice of dividing observations across rows, columns, and multiple tables. The concept of ‘tidy data,’ introduced by Wickham (2014a), offers a set of guidelines for organizing data in order to facilitate statistical analysis and visualization. … This framework makes it easy for analysts to reshape, combine, group and otherwise manipulate data. Packages such as ggplot2, dplyr, and many built-in R modeling and plotting functions require the input to be in a tidy form, so keeping the data in this form allows multiple tools to be used in sequence in a seamless analysis pipeline (Wickham, 2009; Wickham and Francois, 2014).” (Robinson, 2014)

“A smart visualization can transform biologists’ understanding of their data”
(Callaway, 2016)

“Scientific figures are typically rendered as static images. But these are divorced from the underlying data, which prevents readers from exploring them in more detail by, for instance, zooming in on features of interest. For genomicists needing to cram millions of data points into dense visuals a few centimeters big, this can be particularly problematic. The same is true for researchers working with computational algorithms. Scientists often post software on open-source repositories such as GitHub, but getting the code to run properly is easier said than done. Reviewers and other interested parties often require extra software and configuration to make the algorithms work. Some journals now bridge that gap by supporting interactive figures and code. One of those is F1000Research...” (Perkel, 2018b)

“Open-source options also exist for creating interactive images, including Bokeh, htmlwidgets, pygal and ipywidgets. Most are used programmatically, generally within either R or Python code, which is commonly used in the sciences. … Two other products let researchers create interactive apps that make use of widgets such as drop-down menus and slider controls to blend data, graphics and code: Shiny, made by RStudio in Boston, Massachusetts, for R, and Plotly’s Dash for Python. They work by transmitting the user’s widget actions to a remote server, which runs the underlying code and updates the package.” (Perkel, 2018b)

“Very little of what I’ve built over the years is monolithic—just a single chunk. Most of the time, I build things in components, then attach those pieces together as I go. So yes, the component parts are pieces that have been made small in precise ways from larger chunks of material, but eventually they will be assembled to create much larger and more complex objects than any of the raw source materials.” (Savage, 2020)

"After they've been built in, mechanical fasteners make everything after that easier. They allow for disassembly, reconfiguration, as well as replacement."
 (Savage, 2020)

"That's the reason I prefer mechanical solutions. They can be undone. Whatever I'm putting together can be pulled apart again without damaging the construction. ... it takes more engineering, more fiddling, and definitely more time. But the trade-off is more options. And I want options. That's the space I like to exist in as a maker." (Savage, 2020)

"Similar to early many, beginner makers start with a rudimentary set of tools for basic creative tasks: a hammer (of course), a set of screwdrivers, scissors, some pliers, maybe a crescent wrench, and some kind of cutting device. Almost everyone who has strived to make things has some combination of this list. Then, as we get more experienced, we seek out better versions of the tools we already have as well as new tools that can facilitate the learning of new techniques—new ways of cutting things apart, and new ways of putting them back together."

(Savage, 2020)

"Once we start to expand past the basic complement of tools, what to add to our collections becomes a multifactor calculus based on reliability, cost, space, time, repairability, skill, and need. These choices are nontrivial, because the tools we use are extensions of our hands and our minds. The best tools 'wear in' to fit you based on how you use them, they get smooth where you grab them. They tell the story of their utility with their patina of use. A toolbox of tools you know well and use lovingly is a magnificent thing." (Savage, 2020)

"The reality is that tool choice is both less important and more important than you think it is. It is less important to the extent that tool usage is entirely subjective, which means there is no one right way to do things. But it is more important, because the best tool for any job is the one you're most comfortable with, the one that you can make do what you want it to do, whose movements you fully understand." (Savage, 2020)

"You must concentration on fundamentals, at least what *you think* at the time are fundamentals, and also develop the ability to learn new fields of knowledge when they arise so you will not be left behind..." (Hamming, 1997)

"How are you to recognize 'fundamentals'? One test is that they have lasted a long time. Another test is from the fundamentals all the rest of the field can be derived by using the standard methods in the field." (Hamming, 1997)

3.3.6 Subsection 2

3.3.7 Practice quiz

3.4 Complex data types in experimental data pre-processing

Raw data from many biomedical experiments, especially those that use high-throughput techniques, can be very large and complex. Because of the scale and complexity of these data, software for pre-processing the data in R often uses complex, 'untidy' data formats. While these formats are necessary for

computational efficiency, they add a critical barrier for researchers wishing to implement reproducibility tools. In this module, we will explain why use of complex data formats is often necessary within open source pre-processing software and outline the hurdles created in reproducibility tool use among laboratory-based scientists.

Objectives. After this module, the trainee will be able to:

- Explain why R software for pre-processing biomedical data often stores data in complex, ‘untidy’ formats
- Describe how these complex data formats can create barriers to laboratory-based researchers seeking to use reproducibility tools for data pre-processing

In previous modules, we have gone into a lot of detail about all of the advantages of the tidyverse approach. However as you work with biomedical data, particularly complex data from complex research equipment, like mass spectrometers and flow cytometers, you may find that it is unreasonable to start with a tidyverse approach from the first steps of pre-processing the data.

In this module, we will explain why the tidyverse approach is currently not always ideal throughout all steps of pre-processing, analysis, and visualization of the types of data that you may collect through a biomedical research experiment. We will explain what data structures are, and present some of the types of data structures commonly used in packages in the Bioconductor project. These more complex data structures largely leverage a system in R called the S4 object-oriented system, which translates some ideas from object-oriented programming to use to handle large and complex data in R.

In this module, we will cover several of the most popular data structures (each available as an S4 object class) that are used to work with data within Bioconductor packages. In a later module, we will explain how you can build a pipeline that combines Bioconductor and tidyverse approaches, in which early steps in data pre-processing use the Bioconductor approach to handle large and complex initial data, and later steps shift to use a tidyverse approach, once it is appropriate to store data in simpler structures like dataframes.

In this and following modules, we will therefore explain the advantages and disadvantages of complex versus simpler data storage formats in R. We will also explain how these advantages and disadvantages weigh out differently in different stages of a data preprocessing and analysis workflow. Finally, we will describe how you can leverage both to your advantage, and in particular the tools and approaches that you can use to shift from a Bioconductor-style approach—with heavy use of complex data storage formats—early in your preprocessing pipeline to a tidyverse approach—centered on storing data in a simple, tidy dataframe object—at later stages, when the data are more suitable to this simpler storage format, which allows you to leverage the powerful and widely-taught tidyverse approach in later steps of analysis and visualization.

In these modules, we will focus on explaining these ideas within the R programming language. This language is a very popular one for both biomedical

data sets and also for more general tasks in data management and analysis. However, these principles also apply to other programming languages, particularly those that can be used in an interactive format, including Python and Julia.

Data in R can be stored in a variety of other formats, too. When you are working with biological data—in particular, complex or large data output from laboratory equipment—there can be advantages to using data structures besides dataframes. In this section, we'll discuss some of the complex characteristics of biomedical data that recommend the use of data structures in R beyond the dataframe. We'll also discuss how the use of these other data structures can complicate the use of “tidyverse” functions and principles that you might learn in beginning R programming courses and books. In later modules, we'll discuss how to connect your work in R to clean and analyze data by performing earlier pre-processing steps using more complex data structures and then transferring when possible to dataframes for storing data, to allow you to take advantage of the power and ease of the “tidyverse” approach as early as possible in your pipeline.

As you learn R, you will almost certainly learn how to create and work with the more general data structures, including how to explore the data stored in each of them. By contrast, you may never learn many of the more complex data storage formats, especially if you are not using packages from Bioconductor. However, there are a number of good reasons why R packages—especially those shared through Bioconductor—define and use more complex data formats. In this and following modules, we will explain the advantages and disadvantages of complex versus simpler data storage formats in R. We will also explain how these advantages and disadvantages weigh out differently in different stages of a data preprocessing and analysis workflow. Finally, we will describe how you can leverage both to your advantage, and in particular the tools and approaches that you can use to shift from a Bioconductor-style approach—with heavy use of complex data storage formats—early in your preprocessing pipeline to a tidyverse approach—centered on storing data in a simple, tidy dataframe object—at later stages, when the data are more suitable to this simpler storage format, which allows you to leverage the powerful and widely-taught tidyverse approach in later steps of analysis and visualization.

In these modules, we will focus on explaining these ideas within the R programming language. This language is a very popular one for both biomedical data sets and also for more general tasks in data management and analysis. However, these principles also apply to other programming languages, particularly those that can be used in an interactive format, including Python and Julia.

3.4.1 What are complex data structures?

The R programming language offers a wide variety of structures that can be used to store data as you work with it, including steps of preprocessing and analysis of the data. Some of these structures are defined through the base R language that you first install, while other structures are specially defined through the extension R packages you add as you continue to work with R. These packages are specific to the tasks you aim to do, and if they define their own data storage structures, those structures are typically customized to that task.

For example, there are packages—including the `xcms` package, for example—that allow you to load and preprocess data from LC-MS experiments. These packages include functionality to load data from a specialized format output by mass spectrometry equipment, as well as identify and align peaks within the data that might indicate, for example, metabolite features for a metabolomics analysis. The `xcms` package defines its own structures that are used to store data during this preprocessing, and also draws on specialized data structures defined in other R extension packages, including the `OnDiskMSnExp` data object class that is defined by the `MSnbase` package.

Complex data structures like these can be very precise in defining what types of data they contain and where each component of the data goes. Later in this and other modules, we will provide more details about the advantages and disadvantages of these types of specialized data storage formats, especially in the context of improving transparency, rigor, and reproducibility across the steps of preprocessing experimental biomedical data.

Dataframes are very clearly and simply organized. However, they can be too restrictive in some cases. Sometimes, you might have data that are taken at different levels of observation—for example, you might have some measurements that are specific to a specific sample, but then other measurements or data that are common to the experiment as a whole (metadata).

Also, the simple dataframe structure doesn't have the capacity to store data taken at these different levels within the same structure (at least, not without a lot of repetition). Further, the dataframe won't work for massive datasets. Sometimes, you will get massive amounts of data from equipment like spectrometers or cytometers, and these datasets can be so big that they can't be easily read into R. One strategy with massive data is to read in only the bits you need as you need them, rather than reading in all the data and then using R commands to work with the full dataset. We'll look later in this module about how more complex data structures can facilitate this approach when working with massive data in R.

The dataframe structure has the advantage of being simple to understand and use. By using the dataframe as a common structure, the tidyverse approach is able to create a powerful environment for working with data, because the use of a common structure allows you to program using small, simple functions that

can be combined in different ways to solve complex tasks.

However, the dataframe lacks flexibility in storing data that does not naturally follow a two-dimensional structure, and it can struggle to handle massive datasets. Therefore, in some cases, it is appropriate to adopt approaches that store data in more complex data structures.

There are two main features of biomedical data—in particular, data collected from laboratory equipment like flow cytometers and mass spectrometers—that make it useful to use more complex data structures in R in the earlier stages of preprocessing the data. First, the data are often very large, in some cases so large that it is difficult to read them into R. Second, the data might combine various elements, each with their own natural structures, that you'd like to keep together as you move through the steps of preprocessing the data.

The code for different implementations of a method (in other words, different ways it will run with new object classes) can come in different R packages. This allows a developer to add his or her own applications of methods, suited for object classes he or she creates.

A class defines the structure for a way of storing data. When you create an object that follows this structure, it's an instance of that class. The new function is used to create new instances of a class.

When a generic function determines what code to run based on the class of the object, it's called method dispatch.

Examples of complex biomedical data suited for complex data structures

In the last module, we covered the tidyverse approach to working with data in R. This approach hinges on using a common data structure, the dataframe. With many tasks in data management and analysis, this data structure is sufficient, and so can be leveraged with the tidyverse approach. However, there are some cases where data are not well suited for a dataframe structure. In these cases, you may need to use some of R's more complex data structures, at least for part of your data preprocessing and analysis pipeline.

What characteristics might make data less suited for a dataframe structure in R? There are a variety of characteristics that could cause this. First, if data is extremely large, it may exceed a size that is reasonable for a dataframe structure. Some of the more complex data structures in R allow much of the data to stay “on disc”, rather than be loaded into RAM, when the data are structured to provide access from R.

Other datasets may not fit well within the two-dimensional, non-ragged structure that is characteristic of the dataframe structure. For example, some biomedical data may have data that records characteristics at several levels of the data. It may have records on the levels of gene expression within each sample, separate information about each gene that was measured, and another separate set of information that characterizes each of the samples. While it is critical to keep “like” measurements aligned with data like this—in other words, to ensure that you can connect the data that characterizes a gene with

the data that provides measures of the level of expression of that gene in each sample—these data do not naturally have a two-dimensional structure and so do not fit naturally into a dataframe structure.

Building on list data structures

R allows for very complex and specialized data structures, suitable to be customized for very specific needs with large or complex data. These structures tend to build on the generic list data structure.

Another common general purpose data structure in R is the *list*, which allows you to combine data stored in any type of structure to create a single R object, giving enormous flexibility (but minimal set structure from one object to another). This data structure is the building block for some of the more complex specific data structures, which we'll cover next.

The R programming language offers a wide variety of structures that can be used to store data as you work with it, including steps of preprocessing and analysis of the data. Some of these structures are defined through the base R language that you first install, while other structures are specially defined through the extension R packages you add as you continue to work with R. These packages are specific to the tasks you aim to do, and if they define their own data storage structures, those structures are typically customized to that task.

Many of these more specific data structures are built on the more generic idea of the **list** data structure, which provides a very flexible way to combine other data structures to create a single R object. In R, you can think of a list data structure as having various slots where it can store data, and each of these slots can store data stored in another structure. For example, one slot of a list might store a dataframe of data, while another might store a vector or a dataframe for a different level of observation. A slot could even store another list. As an example, if we want to store the type of data shown in Figure 3.2, we could use a list data structure with one slot that stores the metadata for the experiment, another that stores a dataframe or matrix with the assay data, another with a dataframe or matrix that stores the phenotype data, and another with a dataframe or matrix that stores the gene-level data. Each of these slots in the list will get a name, and we can use that name to access each of these pieces of data later. The list, in this case, allows us to store all these different types of data from an experiment in the same structure in R, so we can make sure we keep the data together in a single structure, even though it's too complex to fit in a simple dataframe.

Many R data structures are built on a general structure called a “list”. This data structure is a useful basic general data structure, because it is extraordinarily flexible. The list data structure is flexible in two important ways: it allows you to include data of different types in the same data structure, and it allows you to include data with different dimensions—and data stored hierarchically, including various other data structures—within the list structure. We'll cover each of these points a bit more below and describe why they're helpful in

making the list a very good general purpose data structure.

The list data structure is much more flexible than the vector or dataframe. It essentially allows you to create different “slots”, and you can store any type of data in each of these slots. In each slot you can store any of the other types of data structures in R—for example, vectors, dataframes, or other lists. You can even store unusual things like R environments [?] or pointers that give the directions to where data is stored on the computer without reading the data into R (and so saving room in the RAM memory, which is used when data is “ready to go” in R, but which has much more limited space than the mass [?] storage on your computer).

Since you can put a list into the slot of a list, it allows you to create deep, layered structures of data. For example, you could have one slot in a list where you store the metadata for your experiment, and this slot might itself be a list where you store one dataframe with some information about the settings of the laboratory equipment you used to collect the data, and another dataframe that provides information about the experimental design variables (e.g., which animal received which treatment). Another slot in the larger list then might have experimental measurements, and these might either be in a dataframe or, if the data are very large, might be represented through pointers to where the data is stored in memory, rather than having the data included directly in the data structure.

Given all these advantages of the list data structure, then, why not use it all the time? While it is a very helpful building block, it turns out that its flexibility can have some disadvantages in some cases. This flexibility means that you can't always assume that certain bits of data are in a certain spot in each instance of a list in R. Conversely, if you have data stored in a less flexible structure, you can often rely on certain parts of the data always being in a certain part of the data structure. In a “tidy” dataframe, for example, you can always assume that each row represents the measurements for one observation at the unit of observation for that dataframe, and that each column gives values across all observations for one particular value that was measured for all the observations. For example, if you are conducting an experiment with mice, where a certain number of mice were sacrificed at certain time points, with their weight and the bacteria load in their lungs measured when the mouse was sacrificed, then you could store the data in a dataframe, with a row for each mouse, and columns giving the experimental characteristics for each mouse (e.g., treatment status, time point when the mouse was sacrificed), the mouse's weight, and the mouse's bacteria load when sacrificed. You could store all of this information in a list, as well, but the defined, two-dimensional structure of the dataframe makes it much more clearly defined where all the data goes in the dataframe structure, while you could order the data in many ways within a list.

There is a big advantage to having stricter standards for what parts of data go where when it comes to writing functions that can be used across a lot of

data. You can think of this in terms of how cars are set up versus how kitchens are set up. Cars are very standardized in the “interface” that you get when you sit down to drive them. The gas and brakes are typically floor pedals, with the gas to the right of the brake. The steering is almost always provided through a wheel centered in front of the driver’s torso. The mechanism for shifting gears (e.g., forward, reverse) is typically to the right of the steering wheel, while mechanisms for features like lights and windshield wipers, are typically to the left of the steering wheel. Because this interface is so standardized, you can get into a car you’ve never driven before and typically figure out how to drive it very quickly. You don’t need a lot of time exploring where everything is or a lot of directions from someone familiar with the car to figure out where things are. Think of the last time that you drove a rental car—within five minutes, at most, you were probably able to orient yourself to figure out where everything you needed was. This is like a dataframe in R—you can pretty quickly figure out where everything you might need is stored in the data structure, and people can write functions to use with these dataframes that work well generally across lots of people’s data because they can assume that certain pieces of data are in certain places.

By contrast, think about walking into someone else’s kitchen and orienting yourself to use that. Kitchen designs do tend to have some general features—most will have a few common large elements, like a stove somewhere, a refrigerator somewhere, a pantry somewhere, and storage for pots, pans, and utensils somewhere. However, there is a lot of flexibility in where each of these are in the kitchen design, and further flexibility in how things are organized within each of these structures. If you cook in someone else’s kitchen, it is easy to find yourself disoriented in the middle of cooking a recipe, where a utensil that you can grab almost without thinking in your own kitchen requires you to stop and search many places in someone else’s kitchen. This is like a list in R—there are so many places that you can store data in a list, and so much flexibility, that you often find yourself having to dig around to find a certain element in a list data structure that someone else has created, and you often can’t assume that certain pieces are in certain places if you are writing your own functions, so it becomes hard to write functions that are “general purpose” for generic list structures in R.

There is a way that list structures can be used in R in a way that retains some of their flexibility while also leveraging some of the benefits of standardization. This is R’s system for creating objects. These object structures are built on the list data structure, but each object is constrained to have certain elements of data in certain structures of the data. These structures cannot be used as easily as dataframes in a “tidyverse” approach, since the tidyverse tools are built based on the assumption that data is stored in a tidy dataframe. However, they are used in many of the Bioconductor approaches that allow powerful tools for the earlier stages in preprocessing biological data. The types of standards that are imposed in the more specialized objects include which slots the

list can have, the names they have, what order they're in (e.g., in a certain object, the metadata about the experiment might always be stored in the first slot of the list), and what structures and/or data types the data in each slot should have.

S3 and S4 objects in R

Many complex data structures in R are defined as S3 or S4 objects.

3.4.2 Advantages of complex data structures

Bundling and aligning

One characteristic that can make data complex is if it includes measurements that are taken at several different levels of observation. For example, a single data set may have some observations or measurements that are taken at the level of the sample (e.g., the age, sex, and treatment of each study subject from which the samples are taken). It may have other measurements that are taken at the level of the gene (e.g., [values that you would have per gene]), and then measurements that are the result of a specific assay [?] (e.g., the expression level of each of the measured genes within each sample).

These different types of data could be stored in different dataframes, with a common identifier to help link them. They could even be stored in separate dataframes with the positions used to link them (for example, if each column represents a gene in one dataframe and each row represents a gene in another, then you assume that the second column in the first dataframe aligns with the second row in the second dataframe), although this method can be prone to errors, especially one you begin subsetting the original dataframes in the course of data preprocessing and analysis.

More complex data structures can help in both bundling all these pieces of data into a single object, while at the same time enforcing alignment across the pieces of data, so you can line up the measurements for a specific gene or sample across data taken at different levels.

The data from genomic and other high-throughput experiments often are too complex and/or large to make dataframes practical as data structures, at least until data can be simplified through pre-processing. In this section, we'll look at an approach to pre-processing these data, leveraging more complex data structures when needed. Once data have been pre-processed, they are often simplified to the point where they can be stored in a dataframe, and so it is possible to create workflows that move into a tidyverse approach once data can reasonably be stored in dataframes. This creates a powerful pipeline, using more complex tools when necessary and then moving into the more straightforward tidyverse approach when possible. In the next module, we'll discuss how you can adopt this type of combined approach.

Most laboratory equipment can output a raw data file that you can then read into R. For many types of laboratory equipment, these raw data files follow a strict format. The file formats will often have different pieces of data stored

in specific spots. For example, the equipment might record not only the measurements taken for the sample, but also information about the setting that were applied to the equipment while the measurements were taken, the date of the measurements, and other metadata that may be useful to access when preprocessing the data. Each piece of data may have different “dimensions”. For example, the measurements might provide one measurement per metabolite feature or per marker. Some metadata might also be provided with these dimensions (e.g., metadata about the markers for flow cytometry data), but other metadata might be provided a single time per sample or even per experiment—for example, the settings on the equipment when the sample or samples were run.

When it comes to data structures, dataframes and other two-dimensional data storage structures (you can visualize these as similar to the format of data in a spreadsheet, with rows and columns) work well to store data where all data conform to a common dimension. For example, a dataframe would work well to store the measurements for each marker in each sample in a flow cytometry experiment. In this case, each column could store the values for a specific marker and each row could provide measurements for a sample. In this way, you could read the measurements for one marker across all samples by reading down a column, or read the measurements across all markers for one sample by reading across a row.

When you have data that doesn't conform to these common dimensions [unit of measurement?] however, a dataframe may work poorly to store the data. For example, if you have measurements taken at the level of the equipment settings for the whole experiment, these don't naturally fit into the dataframe format. In the “tidyverse” approach, one approach to handling data with different units of measurement is to store data for each unit of measurement in a different dataframe and to include identifiers that can be used to link data across the dataframes. More common, however, in R extensions for preprocessing biomedical data is to use more complex data structures that can store data with different units of measurement in different slots within the data structure, and use these in conjunction with specific functions that are built to work with that specific data structure, and so know where to find each element within the data structure.

Let's start by looking at how some data might have a structure that is too complex to fit into a two-dimensional dataframe. Figure 3.2 shows an example of different components of data that may need to be stored in a data structure that is more complex than a dataframe. Data from a genomic experiment may include data from several levels, including metadata that describes the entire experiment, as well as assay data, phenotype data, and gene-level data. More complex data structures in R can be used to store all these pieces of data inside a single data structure.

Handling large data

Two approaches for this are on disc and through R environments.

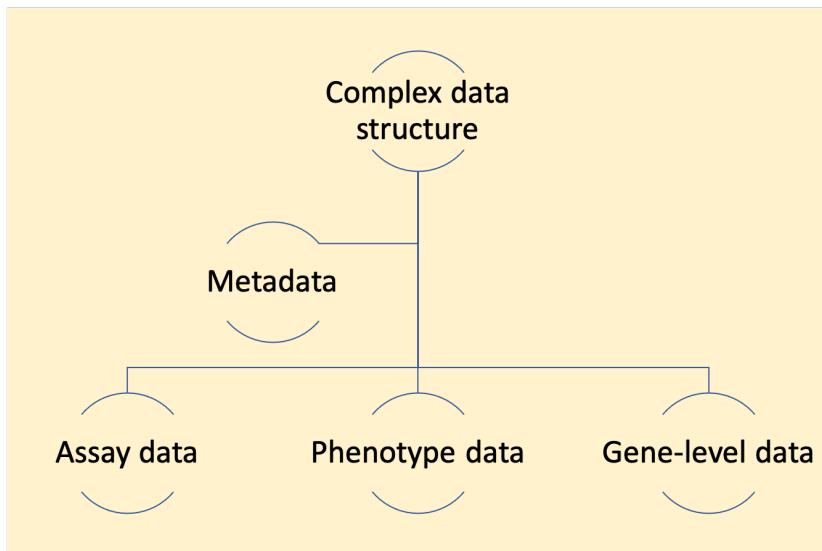


Figure 3.2: An example of different components of data that may need to be stored in a data structure that is more complex than a dataframe. Data from a genomic experiment may include data from several levels, including metadata that describes the entire experiment, as well as assay data, phenotype data, and gene-level data. More complex data structures in R can be used to store all these pieces of data inside a single data structure.

There's a second reason why dataframe structures don't always work for data from biological experiments, which has to do with the size of data (and so how much memory it requires). A computer has several ways that it can store data. The primary storage is closely connected with the computer's processing unit, where calculations are made, and so data stored in this primary storage can be processed by code very quickly. R uses this approach, and so when you load data in R to be stored in one of its traditional data structures, that data is moved into part of the computer's primary storage (its random access memory, or RAM).

Data can also be stored in other devices on a computer, including hard drives and solid state drives that are built into the computer (the computer's secondary storage devices) or even onto storage devices that can be removed from the computer, like USB drives or external hard drives (the computer's tertiary storage). The size of available storage in these devices tends to be much, much larger than the storage size of the computer's RAM. However, it takes longer to access data in these secondary storage devices because they aren't directly connected to the processor, and instead require the data to move into RAM before it can be accessed by the processor, which is the only part of the computer that can do things to analyze, modify, or otherwise process the data.

Your data will often be saved on a file in the computer's secondary memory (e.g., in a file stored on the computer's solid state drive) before you read it into R, then moved into memory (RAM, part of the primary storage) when you ask R to load it into a data structure that you can access with code commands in R. However, the storage size available in RAM will always be much, much smaller

"Big data is encountered in genomics for two reasons: the size of the genome and the heterogeneity of populations. Complex organisms, such as plants and animals, have genomes on the order of billions of base pairs (the human genome consists of over three billion base pairs). The diversity of populations, whether of organisms, tissues or cells, means we need to sample deeply to detect low frequency events. To interrogate long and/or numerous genomic sequences, many measurements are necessary. For example, a typical whole genome sequencing experiment will consist of over one billion reads of 75–100 bp each. The reads are aligned across billions of positions, most of which have been annotated in some way. This experiment may be repeated for thousands of samples. Such a data set does not fit within the memory of a current commodity computer, and is not processed in a timely and interactive manner. To successfully wrangle a large data set, we need to intimately understand its structure and carefully consider the questions posed of it."

[@lawrence2014scalable]
 "A major challenge in the analysis of scRNA-seq data is the scalability of analysis methods as datasets increase in size over time. This is particularly problematic as experiments now frequently produce millions of cells [50–53], possibly across multiple batches, making it challenging to even load the data into memory and perform downstream analyses including quality control, batch correction and dimensionality reduction. Providing analysis methods, such as unsupervised clustering, that do not require data to be loaded into memory is an imperative step for scalable analyses. While large-scale scRNA-seq data are now routinely stored in on-disk data formats (e.g. HDF5 files), the methods to process and analyze these data are lagging."

[@hicks2021mbkmeans]

than the storage size in secondary storage devices like solid state drives, and so with larger data, problems can arise if you try to read data into RAM that is too large for that primary storage device to accommodate.

The traditional dataframe structure in R is therefore built after reading the data in memory, into RAM. However, many biological experiments now create data that is much too large to read into memory for R in a reasonable way (Lawrence and Morgan, 2014; Hicks et al., 2021). More complex data structures can allow more sophisticated ways to handle massive data, and so they are often necessary when working with massive biological datasets, particularly early in pre-processing, before the data can be summarized in an efficient way. For example, a more complex data structure could allow much of the data to be left on disk, and only read into memory [RAM?] on demand, as specific portions of the data are needed (Gatto, 2013; Hicks et al., 2021). This approach can be used to iterate across subsets of the data, only reading parts of the data into memory at a time (Lawrence and Morgan, 2014). Such structures can be designed to work in a way that, if you are the user, you won't notice the difference in where the data is kept (on disk versus in memory)—this means you won't have to worry about these memory management issues, but instead can just gain from everything going smoothly, even as datasets get very large (Gatto, 2013).

[Data size, on-disk backends for files, like HDF5 and netCDF—used for flow cytometry file format?]

[Potential future direction—developments of tidyverse based front ends for data stored in databases or on-disk file formats—sergeant package is one example, also running tidyverse commands on data in database, matter package?, disk.frame package?]

Allow software development across large, diverse teams

Robert Gentleman, one of the creators of Bioconductor [?], highlights this as a key reason for using these more complex data structures. R is an open-source language, and it allows contributions from software developers worldwide. The Bioconductor project leverages the base R software to build tools for working with complex genomic [?] data, also allowing for the development of new extensions from developers worldwide. The software extensions are developed, therefore, not by a unified team that is all employed at the same place, but instead by a collection of people who are making software as part of their research program, or perhaps as part of an industry job, and who write that software with the intent of having it work smoothly with other software packages that are available through Bioconductor.

If you have ever been in a role where you organized volunteers to perform a large task, you can probably imagine how challenging it is to design a project in a way that allows for this type of software development to work. The use of specialized data structures is one element that helps in this coordination for Bioconductor. It enforces a standard that all the programmers can work toward—they can work under the assumption that the output from other

"Reading in a large dataset for which you do not have enough RAM is one easy way to freeze up your computer (or at least your R session). This is usually an unpleasant experience that usually requires you to kill the R process, in the best case scenario, or reboot your computer, in the worst case."

[@peng2016r]

"If you use too much memory, R will complain. The key issue is that R holds all the data in RAM. This is a limitation if you have huge datasets. The up-side is flexibility—in particular, R imposes no rules on what data are like."

[@burns2011r]

"Random access memory (RAM) is a type of computer memory that can be accessed randomly: any byte of memory can be accessed without touching the preceding bytes. RAM is found in computers, phones, tablets and even printers. The amount of RAM R has access to is incredibly important. Since R loads objects into RAM, the amount of RAM you have available can limit the size of data set you can analyse."

[@gillespie2016efficient]

"A rough rule of thumb is that your RAM should be three times the size of your data set."

[@gillespie2016efficient]

"RAM is cheap and thinking hurts."

Uwe Ligges (about memory requirements in R) R-help (June 2007)

"The strengths of R are also its weaknesses: the R API encourages users to store entire data sets in memory as vectors. These

functions in the Bioconductor ecosystem will be structured in a certain way, and so they can work under the assumption that certain elements of the data will be in certain spots in the data structure of the object.

[Example of the party game where you each add a sentence, only seeing part of the story. Exquisite corpse.]

The downside of a general list data structure, however, comes in when it comes to developing software for data stored in that structure. The general list structure is very flexible, which is why it can store such different types of data, but this flexibility means that there's no guarantee about where in the data structure specific elements might be stored. By comparison, in a dataframe each row can be assumed to capture an observation, and each column will capture measurements or characteristics of that observation. Someone can therefore develop software to work with data stored in that structure, relying on finding those type of data in those locations in the data structure. When more complex data are stored in a general list object, the different components could be stored in different ways by different users. For example, for the type of complex data shown in Figure 3.2, one person might store the metadata in the first slot of a list and the phenotype data in the second, while another person might store the data in a list with the metadata in the last slot and the phenotype data in the first.

The way around this problem is to create data structures that build off the general list structure, but impose some rules that constrain the structure, so that the same types of data are always stored in the same spot. By doing this, software developers can develop code to work with data stored in that structure, with the guarantee that they can always find certain elements of the data in certain spots in the data structure. Bioconductor makes heavy use of these types of specialized data structures, typically called "classes" in Bioconductor tutorials and user manuals. This is because, in R, they are created using one of R's object-oriented approaches, most often one called S4.

Complex data structures like these can be very precise in defining what types of data they contain and where each component of the data goes. For example, they may have a "phenoData" slot that only will store a specialized dataframe with phenotype data describing each sample in the experiment [?], and another slot named "featureData" that will only store a specialized dataframe with data about each feature (e.g., gene) investigated in the experiment. With this structure, a software developer can develop a program that inputs data in this structure, always knowing where to find the feature data or the phenotype data.

[Validation of data as it's entered in an S4 class]

R programmers get a lot of advantages from using these classes because they can write functions under the assumption that certain pieces of the data will always be in the same spot for that type of object. There is still flexibility in the object, in that it can store lots of different types of data, in a variety of different structures. While this "object oriented" approach in R data structures

does provide great advantages for programmers, and allow them to create powerful tools for you to use in R, it does make it a little trickier in some cases for you to explore your data by hand as you work through preprocessing. This is because there typically are a variety of these object classes that your data will pass through as you go through different stages of preprocessing, because different structures are suited to different stages of analysis. Functions often can only be used for a single class of objects, and so you have to keep track of which functions pair up with which classes of data. Further, it can be a bit tricky—at least in comparison to when you have data in a dataframe—to explore your data by hand, because you have to navigate through different slots in the object. By contrast, a dataframe always has the same two-dimensional, rectangular structure, and so it's very easy to navigate and explore data in this structure, and there are a large number of functions that are built to be used with dataframes, providing enormous flexibility in what you can do with data stored in this structure.

3.4.3 *Biomedical data preprocessing with complex data structures*

As we noted briefly earlier, the Bioconductor project is one area of R programming where complex data structures, rather than a tidyverse approach, tends to dominate (although there does seem to be an evolution toward more tidyverse techniques in Bioconductor, as we'll discuss some in later modules). Bioconductor is critical to learn if you are working with genomic [?] data, as many of the key tools and algorithms for this type of data are shared through that project. This means that, for many biomedical researchers who are now generating complex, high-throughput data, it is worth learning how to use complex data structures in R.

Powerful algorithms

One of the advantages of these complex data structures for biomedical data preprocessing is that they can be leveraged to develop very powerful algorithms for working with complex biomedical data.

[Examples]

Examples of types of complex pre-processing algorithms

In some cases, more complex algorithms may be used to normalize data across different samples [?—for example, the normalization algorithm may leverage the assumption that the vast majority of values (e.g., expression levels of genes) are the same in all samples, with only a few varying between the samples. Starting from this assumption, an algorithm can be developed to normalize across the samples, to help in identifying genes with important differences in expression across the samples.

[Other examples]

Challenges

It can be harder to explore your data along the way. With a tidyverse approach, since data are always in dataframes, you can quickly learn some tools

that let you extract parts of the data, summarize the data, and visualize it. This allows for exploratory data analysis at each stage as you develop your preprocessing and analysis pipeline. Conversely, if you have a pipeline where the data is continuously being moved from one data structure to another, you have to learn how to use and explore each type of data structure to become facile with exploring your data along the way.

This can also make it harder to experiment, since often a function will be tied to a specific data structure, and so instead of learning a small set of general use functions that can be combined together in different ways (the tidyverse approach), you must learn more functions, since functions tend to be tied to a data structure (outside of generic functions), and the data structure tends to change frequently across the pipeline.

To be clear, a pipeline in R that includes these complex data structures will typically still be modular, in the sense that you can adapt and separate specific parts of the pipeline. However, they tend to be much less flexible than pipelines developed with a tidyverse approach. The data structure changes often, with certain functions outputting a data structure that is needed for the next step, then the function of the next step outputting the data in a different structure, and so on. This changing data structure means that the functions for each step often are constrained to always be put in the same order. By comparison, the small tools that make up tidyverse functions can often be combined in many different orders, letting you build a much larger variety of pipelines with them. Also, many of the functions that work with complex data types will do many things within one function, so they can be harder to learn and understand, and they are often much more customized to a specific action, which means that you have to learn more functions (since each does one specific thing).

3.4.4 Disadvantages of complex data structures

Harder to see all the data

Harder to access all the data

Harder to make use of general purpose tools

Higher entry barrier to learn to use well

In previous sections, we described how the R programming language allows for object-oriented programming, and how customized objects are often used in preprocessing for biological data. This is a helpful approach for preprocessing, because it can handle complexities in biological data at its early stages of preprocessing, when R must handle complex input formats from equipment like flow cytometers or mass spectrometers, and data sizes that are often very large.

However, once you have preprocessed your data, it is often possible to work with it in a smaller, more consistent object type. This will give you a lot of flexibility and power. While object-oriented approaches can handle complex data, it can be a little hard to write and work with code that is built on an

object oriented approach. Working with this type of code requires you to keep track of what object type your data is in at each stage of a code pipeline, as well as which functions can work with that type of object.

Further, this type of coding, in practice at least, can be a bit inflexible. Often, specific functions only work with a single or few types of functions. In theory, object-oriented programming allows for *methods* that work in customized ways with different types of objects to apply customized code to that type of object for similar, common-sense results. For example, there are often summary and plot methods for most types of objects, and these apply code that is customized to that object type and output, respectively, summarized information about the data in the object and a plot of the data in the object. However, when you want to do more with the object that summarize it or create its default plot, you often end up needing to move to more customized functions that work only with a single or few object types. When you get to this point, you find that you have to remember which functions work with which object type, and you have to use different functions at different stages of your code pipeline, as your code changes from one object class to another.

Further, many of these functions input one object type and output a different one. This evolution of object types for storing data can be difficult to navigate and keep track of. Different object types store data in different ways, and so this evolution of data object types for storage can make it tricky to figure out how to extract and explore data along the pipeline. It makes it hard to write your own code to explore and visualize the data along the way, as well, and so users are often restricted to the visualization and analysis functions pre-made and shared in packages when working with data in complex object types, especially until the user becomes very comfortable with coding in R.

Overall, what does this all mean? Object-oriented approaches offer real advantages early in the process of pre-processing biological data, especially complex and large data output from complex laboratory equipment. However, once this pre-processing is completed, there is a big advantage in moving the data into a simple format and then continuing coding, data analysis, and visualization using tools that work with this simple format. This is the approach taken by a suite of R packages called the “tidyverse”, as well as extensions that build off the approach that this suite of tools embraces. This “tidyverse” approach is described in the next section.

In the previous parts of this module, we've highlighted some of the ways that complex data structures are useful (and even necessary) for parts of the data pre-processing you may do in R. However, they also have some downsides. In a later module, we'll talk about how you can use a combined workflow that uses the Bioconductor approach (with more complex data structures) when necessary, but then shifts into a tidyverse approach (based on keeping data in a data frame structure) as soon as possible in the workflow. Here, we'll describe some of the limitations of complex data structures to help explain why it's worthwhile to develop workflows with this combined approach.

The first limitation of using complex data structures is that it requires you to learn each of the data structures and where they keep different elements of the data. Each specialized data structure (“class” in Bioconductor) has defined rules for each of its data storage slots, and you must become familiar with these class-specific rules to be able to explore and extract data stored in that structure.

For example, the ExpressionSet data structure (defined in the Biobase package in Bioconductor) is used to hold information from high-throughput assays, like . . . It includes different slots for data from the assay, phenotype data (e.g., . . .), and feature data (e.g., . . .), as well as slots that can be used to store things like metadata about the experiment. Each of these slots has its own name, and you would need to know these names to extract and explore each of these elements of the data from the structure. If you are doing an analysis of these type of data, the data might move from this ExpressionSet data structure into other data structures, for example a DGEList data structure (defined in the edgeR package in Bioconductor) after you have . . ., and then a DGEExact data structure (also defined in the edgeR package in Bioconductor) after you have performed [differential expression analysis?] . . . To explore the data after each of these steps, you would need to know the rules, including the names of each slot, of each of these separate, specialized data structures.

The second limitation of using complex data structures is that it requires you to know which functions work with which data structures, and to only use the appropriate functions with the appropriate data structures. In essence, this requires you to develop an understanding of how the data moves from one data structure to another throughout the workflow so that you can apply appropriate functions at each step.

This also means that you often have to learn a larger set of functions, as different functions are needed to do similar things to data in different data structures. For example, if you have a function that can normalize data in one data structure, it may not work for data stored in a different data structure. By contrast, when data are stored in a common structure (like a dataframe), the same functions can always be used, across any step in the pipeline.

There are some approaches that programmers have taken to get around this limitation, using what are called “methods”. Methods are functions that run differently depending on what type of data structure they are given. One example in R is the `summary` function, which outputs a summary of the object you input. This function first checks the class of the object (i.e., the data structure) and then has different sets of code that it uses depending on what the class is. In this way, a user can call `summary` on data stored in many different data structures and get back something that is appropriate for that data structure. This helps in limiting the number of function names you, as a user, must remember. A number of methods exist in R, including `print`, `summary`, and `plot`, that apply across many different classes of objects (and so across many of the Bioconductor data structures). However, many of these methods are

geared toward providing “final” output—a final summary or plot that you might read and interpret directly, but not output that serves as a “step” in a longer workflow, so not output that will become input to another function.

Therefore, even the approach of methods can’t get around another way that specialized data structures and their use of specialized functions constrain you when pre-processing and analyzing data. When data are stored in specialized data structures, with different structures used for each step of the process, it is often the case that instead of being able to flexibly combine different functions in different orders to use small steps to build up to complex processes, you instead are often constrained to use functions in a predefined order, as they shift your data from one structure to another. Although the Bioconductor functions are powerful in pre-processing and analyzing these large, complex data, they also constrain you somewhat to follow the steps and analysis imagined by the original package creator, rather than providing flexibility to create your own series of steps, as the tidyverse approach does.

“There is a cost to the free lunch. That `print` is generic means that what you see is not what you get (sometimes). In the printing of an object you may see a number that you want—an R-squared for example—but don’t know how to grab that number.”

[@burns2011r]

3.4.5 Practice quiz

3.5 Complex data types in R and Bioconductor

Many R extension packages for pre-processing experimental data use complex (rather than ‘tidy’) data formats within their code, and many output data in complex formats. Very recently, the *broom* and *biobroom* R packages have been developed to extract a ‘tidy’ dataset from a complex data format. These tools create a clean, simple connection between the complex data formats often used in pre-processing experimental data and the ‘tidy’ format required to use the ‘tidyverse’ tools now taught in many introductory R courses. In this module, we will describe the ‘list’ data structure, the common backbone for complex data structures in R and provide tips on how to explore and extract data stored in R in this format, including through the *broom* and *biobroom* packages.

Objectives. After this module, the trainee will be able to:

- Describe the structure of R’s ‘list’ data format
- Take basic steps to explore and extract data stored in R’s complex, list-based structures
- Describe what the *broom* and *biobroom* R packages can do
- Explain how converting data to a ‘tidy’ format can improve reproducibility

3.5.1 Bioconductor data structures

When you are writing scripts in R to work with your code, if you are at a point in your pipeline when you can use a “tidyverse” approach, then you will “keep” your data in a data frame, as your data structure, throughout your work. However, at earlier stages in your preprocessing, you may need to use tools

that use other data structures. It's helpful to understand the basic building blocks of R data structures, so you can find elements of your data in these other, more customized data structures.

For example, metabolomics data can be collected from the mass spectrometer with the goal of measuring levels of a large number of metabolite features in each sample. The data collected from the mass spectrometer will be very large, as these data describe the full spectra [?] measured for each sample. Through pre-processing, these data can be used to align peaks across different samples and measure the area under each peak [?] to estimate the level of each metabolite feature in each sample. This pre-processing will produce a much smaller table of data, with a structure that can be easily stored in a dataframe structure (for example, a row for each sample and a column for each metabolite feature, with the cell values giving the level of each metabolite feature in each sample). Therefore, before pre-processing, the data will be too complex and large to reasonably be stored in a dataframe structure, but instead will require a Bioconductor approach and the use of more complex data structures, while after pre-processing, the workflow can move into a tidyverse approach, centered on keeping the data in a dataframe structure.

Come in packages

Help files for data structures

Data structure often changes over pipeline

Generic versus structure-specific functions

How to access data in a complex structure

3.5.2 Example—Seurat data structure

A tour of the Seurat data structure

3.5.3 Other popular Bioconductor data structures

Some object classes in BioConductor:

- eSet from Biobase
- Sequence from IRanges
- MAlist from limma
- ExpressionSet from Biobase

Some of the most important data structures in Bioconductor are (Huber et al., 2015a) (from Table 2 in this reference):

- ExpressionSet (Biobase package)
- SummarizedExperiment (GenomicRanges package)
- GRanges (GenomicRanges package)
- VCF (VariantAnnotation package)
- VRanges (VariantAnnotation package)
- BSgenome (BSgenome package)

Structures for sequence data

Structures for mass spectrometry data

Structures for flow cytometry data

Structures for gene expression data

Structures for single-cell gene expression data

BiocViews to find more tools

3.5.4 Working with data in Bioconductor structures

How to explore data in Bioconductor structures

While it is a bit trickier to explore your data when it is stored in a list—either a general list you created, or one that forms the base for a specialized class structure through functions from a Bioconductor package—you can certainly learn how to do this navigation. This is a powerful and critical tool for you to learn as you learn to preprocess your data in R, as you should never feel like your data is stored in a “black box” structure, where you can’t peek in and explore it. You should *always* feel like you can take a look at any part of your data at any step in the process of preprocessing, analyzing, and visualizing it.

You can use generic functions, like `View()` and `str()`.

You can use `typeof` to determine the data type and `is.[x]` (`is.logical`, `is.character`, `is.double`, and `is.integer`) to test if data has a certain type (Wickham, 2019).

To feel comfortable exploring your data at any stage during the preprocessing steps, you should learn how to investigate and explore data that’s stored in a list structure in R. Because the list structure is the building block for complex data structures, including Bioconductor class structures, this will serve you well throughout your work. You should get in the habit of checking the structure and navigating where each piece of data is stored in the data structure at each step in preprocessing your data. Also, by checking your data throughout preprocessing, you might find that there are bits of information tucked in your data at early stages that you aren’t yet using. For example, many file formats for laboratory equipment include slots for information about the equipment and its settings during when running the sample. This information might be read in from the file into R, but you might not know it’s there for you to use if you’d like, to help you in creating reproducible reports that include this metadata about the experimental equipment and settings.

First, you will want to figure out whether your data is stored in a generic list, or if it’s stored in a specific class-based data structure, which means it will have a bit more of a standardized structure. To do this, you can run the `class` function on your data object. The output of this might be a single value (e.g., “list” `[?]`) or a short list. If it’s a short list, it will include both the specific class of the object and, as you go down the list, the more general data structure types that this class is built on. For example, if the `class` function returns this list:

[Example list of data types---maybe some specific class, then "list"?]

it means that the data's in a class-based structure called ... which is built on the more general structure of a list. You can apply to this data any of the functions that are specifically built for ... data structures, but you can also apply functions built for the more general list data structure.

There are several tools you can use to explore data structured as lists in R. R lists can sometimes be very large—in terms of the amount of data stored in them—particularly for some types of biomedical data. With some of the tools covered in this subsection, that will mean that your first look might seem overwhelming. We'll also cover some tools, therefore, that will let you peel away levels of the data in a bit more manageable way, which you can use when you encounter list-structured data that at first feels overwhelming.

First, if your data is stored in a specific class-based data structure, there likely will also be help files specifically for the class structure that can help you navigate it and figure out where things are. [Example]

[More about exploring data in list structures.]

You can use the `getSlots` function with S4 objects to see all the slots within the object.

How to extract data from Bioconductor structures

By using the accessor function, instead of `@`, your code will be more robust to changes that the developers make. They will be sensitive to insuring that the accessor function for a particular part of the data continues to work regardless of changes they make to the structure that is used to store data in objects in that class. They will be less committed, however, to keeping the same slots, and in the same positions, as they develop the software. The “contract” with the user is through the accessor function, in other words, rather than through the slot name in the object.

Finding functions that work with a data structure

Chaining together preprocessing steps with Bioconductor structures

3.5.5 Example—Preprocessing single cell RNA sequencing data

3.5.6 R's list data structure and list-based structures

When you are writing scripts in R to work with your code, if you are at a point in your pipeline when you can use a “tidyverse” approach, then you will “keep” your data in a data frame, as your data structure, throughout your work. However, at earlier stages in your preprocessing, you may need to use tools that use other data structures. It’s helpful to understand the basic building blocks of R data structures, so you can find elements of your data in these other, more customized data structures.

For example, metabolomics data can be collected from the mass spectrome-

ter with the goal of measuring levels of a large number of metabolite features in each sample. The data collected from the mass spectrometer will be very large, as these data describe the full spectra [?] measured for each sample. Through pre-processing, these data can be used to align peaks across different samples and measure the area under each peak [?] to estimate the level of each metabolite feature in each sample. This pre-processing will produce a much smaller table of data, with a structure that can be easily stored in a dataframe structure (for example, a row for each sample and a column for each metabolite feature, with the cell values giving the level of each metabolite feature in each sample). Therefore, before pre-processing, the data will be too complex and large to reasonably be stored in a dataframe structure, but instead will require a Bioconductor approach and the use of more complex data structures, while after pre-processing, the workflow can move into a tidyverse approach, centered on keeping the data in a dataframe structure.

Many R data structures are built on a general structure called a “list”. This data structure is a useful basic general data structure, because it is extraordinarily flexible. The list data structure is flexible in two important ways: it allows you to include data of different *types* in the same data structure, and it allows you to include data with different dimensions—and data stored hierarchically, including various other data structures—within the list structure. We’ll cover each of these points a bit more below and describe why they’re helpful in making the list a very good general purpose data structure.

In R, your data can be stored as different *types* of data: whole numbers can be stored as an *integer* data type, continuous [?] numbers through a few types of *floating* data types, character strings as a *character* data type, and logical data (which can only take the two values of “TRUE” and “FALSE”) as a *logical* data type. More complex data types can be built using these—for example, there’s a special data type for storing dates that’s based on a combination of an [integer?] data type, with added information counting the number of days [?] from a set starting date (called the [Unix epoch?]), January 1, 1970. (This set-up for storing dates allows them to be printed to look like dates, rather than numbers, but at the same time allows them to be manipulated through operations like finding out which date comes earliest in a set, determining the number of days between two dates, and so on.) R uses these different data types for several reasons. First, by using different data types, R can improve its efficiency [?] in storing data. Each piece of data must—as you go deep in the heart of how the computer works—as a series of binary digits (0s and 1s). Some types of data can be stored using fewer of these *bits* (binary digits). Each measurement of logical data, for example, can be stored in a single bit, since it only can take one of two values (0 or 1, for FALSE and TRUE, respectively). For character strings, these can be divided into each character in the string for storage (for example, “cat” can be stored as “c”, “a”, “t”). There is a set of characters called the ASCII character set that includes the lowercase and uppercase of the letters and punctuation sets that you see on a standard US keyboard [?], and if the

character strings only use these characters, they can be stored in [x] bits per character. For numeric data types, integers can typically be stored in [x] bits per number, while continuous [?] numbers, stored in single or double floating point notation [?], are stored in [x] and [x] bits respectively. When R stores data in specific types, it can be more memory efficient by packing the types of data that can be stored in less space (like logical data) into very compact structures.

The second advantage of the list structure in R is that it has enormous flexibility in terms of storing lots of data in lots of possible places. This data can have different types and even different substructures. Some data structures in R are very constrained in what type of data they can store and what structure they use to store it. For example, one of the “building block” data structures in R is the vector. This data structure is one dimensional and can only contain data that have the same data type—you can think of this as a bead string of values, each of the same type. For example, you could have a vector that gives a series of names of study sites (each a character string), or a vector that gives the dates of time points in a study (each a date data type), or a vector that gives the weights of mice in a study (each a numeric data type). You cannot, however, have a vector that includes some study site names and then some dates and then some weights, since these should be in different data types. Further, you can’t arrange the data in any structure except a straight, one-dimensional series if you are using a vector. The dataframe structure provides a bit more flexibility—you can expand into two dimensions, rather than one, and you can have different data types in different columns of the dataframe (although each column must itself have a single data type).

The list data structure is much more flexible. It essentially allows you to create different “slots”, and you can store any type of data in each of these slots. In each slot you can store any of the other types of data structures in R—for example, vectors, dataframes, or other lists. You can even store unusual things like R environments [?] or pointers that give the directions to where data is stored on the computer without reading the data into R (and so saving room in the RAM memory, which is used when data is “ready to go” in R, but which has much more limited space than the mass [?] storage on your computer).

Since you can put a list into the slot of a list, it allows you to create deep, layered structures of data. For example, you could have one slot in a list where you store the metadata for your experiment, and this slot might itself be a list where you store one dataframe with some information about the settings of the laboratory equipment you used to collect the data, and another dataframe that provides information about the experimental design variables (e.g., which animal received which treatment). Another slot in the larger list then might have experimental measurements, and these might either be in a dataframe or, if the data are very large, might be represented through pointers to where the data is stored in memory, rather than having the data included directly in the data structure.

Given all these advantages of the list data structure, then, why not use

it all the time? While it is a very helpful building block, it turns out that its flexibility can have some disadvantages in some cases. This flexibility means that you can't always assume that certain bits of data are in a certain spot in each instance of a list in R. Conversely, if you have data stored in a less flexible structure, you can often rely on certain parts of the data always being in a certain part of the data structure. In a “tidy” dataframe, for example, you can always assume that each row represents the measurements for one observation at the unit of observation for that dataframe, and that each column gives values across all observations for one particular value that was measured for all the observations. For example, if you are conducting an experiment with mice, where a certain number of mice were sacrificed at certain time points, with their weight and the bacteria load in their lungs measured when the mouse was sacrificed, then you could store the data in a dataframe, with a row for each mouse, and columns giving the experimental characteristics for each mouse (e.g., treatment status, time point when the mouse was sacrificed), the mouse's weight, and the mouse's bacteria load when sacrificed. You could store all of this information in a list, as well, but the defined, two-dimensional structure of the dataframe makes it much more clearly defined where all the data goes in the dataframe structure, while you could order the data in many ways within a list.

There is a big advantage to having stricter standards for what parts of data go where when it comes to writing functions that can be used across a lot of data. You can think of this in terms of how cars are set up versus how kitchens are set up. Cars are very standardized in the “interface” that you get when you sit down to drive them. The gas and brakes are typically floor pedals, with the gas to the right of the brake. The steering is almost always provided through a wheel centered in front of the driver's torso. The mechanism for shifting gears (e.g., forward, reverse) is typically to the right of the steering wheel, while mechanisms for features like lights and windshield wipers, are typically to the left of the steering wheel. Because this interface is so standardized, you can get into a car you've never driven before and typically figure out how to drive it very quickly. You don't need a lot of time exploring where everything is or a lot of directions from someone familiar with the car to figure out where things are. Think of the last time that you drove a rental car—within five minutes, at most, you were probably able to orient yourself to figure out where everything you needed was. This is like a dataframe in R—you can pretty quickly figure out where everything you might need is stored in the data structure, and people can write functions to use with these dataframes that work well generally across lots of people's data because they can assume that certain pieces of data are in certain places.

By contrast, think about walking into someone else's kitchen and orienting yourself to use that. Kitchen designs do tend to have some general features—most will have a few common large elements, like a stove somewhere, a refrigerator somewhere, a pantry somewhere, and storage for pots, pans, and

utensils somewhere. However, there is a lot of flexibility in where each of these are in the kitchen design, and further flexibility in how things are organized within each of these structures. If you cook in someone else's kitchen, it is easy to find yourself disoriented in the middle of cooking a recipe, where a utensil that you can grab almost without thinking in your own kitchen requires you to stop and search many places in someone else's kitchen. This is like a list in R—there are so many places that you can store data in a list, and so much flexibility, that you often find yourself having to dig around to find a certain element in a list data structure that someone else has created, and you often can't assume that certain pieces are in certain places if you are writing your own functions, so it becomes hard to write functions that are “general purpose” for generic list structures in R.

There is a way that list structures can be used in R in a way that retains some of their flexibility while also leveraging some of the benefits of standardization. This is R's system for creating *objects*. These object structures are built on the list data structure, but each object is constrained to have certain elements of data in certain structures of the data. These structures cannot be used as easily as dataframes in a “tidyverse” approach, since the tidyverse tools are built based on the assumption that data is stored in a tidy dataframe. However, they are used in many of the Bioconductor approaches that allow powerful tools for the earlier stages in preprocessing biological data. The types of standards that are imposed in the more specialized objects include which slots the list can have, the names they have, what order they're in (e.g., in a certain object, the metadata about the experiment might always be stored in the first slot of the list), and what structures and/or data types the data in each slot should have.

R programmers get a lot of advantages from using these classes because they can write functions under the assumption that certain pieces of the data will always be in the same spot for that type of object. There is still flexibility in the object, in that it can store lots of different types of data, in a variety of different structures. While this “object oriented” approach in R data structures does provide great advantages for programmers, and allow them to create powerful tools for you to use in R, it does make it a little trickier in some cases for you to explore your data by hand as you work through preprocessing. This is because there typically are a variety of these object classes that your data will pass through as you go through different stages of preprocessing, because different structures are suited to different stages of analysis. Functions often can only be used for a single class of objects, and so you have to keep track of which functions pair up with which classes of data. Further, it can be a bit tricky—at least in comparison to when you have data in a dataframe—to explore your data by hand, because you have to navigate through different slots in the object. By contrast, a dataframe always has the same two-dimensional, rectangular structure, and so it's very easy to navigate and explore data in this structure, and there are a large number of functions that are built to be used

with dataframes, providing enormous flexibility in what you can do with data stored in this structure.

While it is a bit trickier to explore your data when it is stored in a list—either a general list you created, or one that forms the base for a specialized class structure through functions from a Bioconductor package—you can certainly learn how to do this navigation. This is a powerful and critical tool for you to learn as you learn to preprocess your data in R, as you should never feel like your data is stored in a “black box” structure, where you can’t peek in and explore it. You should *always* feel like you can take a look at any part of your data at any step in the process of preprocessing, analyzing, and visualizing it.

“There are four primary types of atomic vectors: logical, integer, double, and character (which contains strings). Collectively, integer and double vectors are known as numeric vectors.” (Wickham, 2019)

“... the most important family of data types in base R [is] vectors. ... Vectors come in two flavours: atomic vectors and lists. They differ in terms of their elements’ types: for atomic vectors, all elements must have the same type; for lists, elements can have different types. ... Each vector can also have **attributes**, which you can think of as a named list of arbitrary metadata. Two attributes are particularly important. The **dimension** attribute turns vectors into matrices and arrays and the **class** attribute powers the S3 object system.” (Wickham, 2019)

“A few places in R’s documentation call lists generic vectors to emphasise their difference from atomic vectors.” (Wickham, 2019)

“Some of the most important S3 vectors [are] factors, dates and times, data frames, and tibbles.” (Wickham, 2019)

You can use `typeof` to determine the data type and `is.[x]` (`is.logical`, `is.character`, `is.double`, and `is.integer`) to test if data has a certain type (Wickham, 2019).

“You may have noticed that the set of atomic vectors does not include a number of important data structures like matrices, arrays, factors, or date-times. These types are all built on top of atomic vectors by adding attributes.” (Wickham, 2019)

“Adding a `dim` attribute to a vector allows it to behave like a 2-dimensional **matrix** or a multi-dimensional **array**. Matrices and arrays are primarily mathematical and statistical tools, not programming tools...” (Wickham, 2019)

“One of the most important vector attributes is `class`, which underlies the S3 object system. Having a `class` attribute turns an object into a **S3 object**, which means it will behave differently from a regular vector when passed to a `generic` function. Every S3 object is built on top of a base type, and often stores additional information in other attributes.” (Wickham, 2019)

“Lists are a step up in complexity from atomic vectors: each element can be any type, not just vectors.” (Wickham, 2019)

"Lists are sometimes called **recursive** vectors because a list can contain other lists. This makes them fundamentally different from atomic vectors." (Wickham, 2019)

"The two most important S3 vectors built on top of lists are data frames and tibbles. If you do data analysis in R, you're going to be using data frames. A data frame is a named list of vectors with attributes for (column) `names`, `row.names`, and its class,"`data.frame`"... In contrast to a regular list, a data frame has an additional constraint: the length of each of its vectors must be the same. This gives data frames their rectangular structure..." (Wickham, 2019)

"Data frames are one of the biggest and most important ideas in R, and one of the things that makes R different from other programming languages. However, in the over 20 years since their creation, the ways that people use R have changed, and some of the design decisions that made sense at the time data frames were created now cause frustration. This frustration led to the creation of the tibble [Muller and Wickham, 2018], a modern reimagining of the data frame. Tibbles are meant to be (as much as possible) drop-in replacements for data frames that fix those frustrations. A concise, and fun, way to summarise the main differences is that tibbles are lazy and surly: they do less and complain more." (Wickham, 2019)

"Tibbles are provided by the `tibble` package and share the same structure as data frames. The only difference is that the class vector is longer, and includes `tbl_df`. This allows tibbles to behave differently in [several] key ways. ... tibbles never coerce their input (this is one feature that makes them lazy)... Additionally, while data frames automatically transform non-syntactic names (unless `check.names = FALSE`), tibbles do not... While every element of a data frame (or tibble) must have the same length, both `data.frame()` and `tibble()` will recycle shorter inputs. However, while data frames automatically recycle columns that are an integer multiple of the longest column, tibbles will only recycle vectors of length one. ... There is one final difference: `tibble()` allows you to refer to variables created during construction. ... [Unlike data frames,] tibbles do not support row names. ... One of the most obvious differences between tibbles and data frames is how they print... Tibbles tweak [a data frame's subsetting] behaviours so that `a` always returns a tibble, and `a $` doesn't do partial matching and warns if it can't find a variable (this is what makes tibbles surly). ... List columns are easier to use with tibbles because they can be directly included inside `tibble()` and they will be printed tidily." (Wickham, 2019)

"Since the elements of lists are references to values, the size of a list might be much smaller than you expect." (Wickham, 2019)

"[The] behavior [of environments] is different from that of other objects: environments are always modified in place. This property is sometimes described as **reference semantics** because when you modify an environment all existing bindings to that environment continue to have the same reference. ... This basic idea can be used to create functions that 'remember' their previous state... This property is also used to implement the R6 object-oriented programming system..." (Wickham, 2019)

3.5.7 Exploring and extracting data from R list data structures

To feel comfortable exploring your data at any stage during the preprocessing steps, you should learn how to investigate and explore data that's stored in a

list structure in R. Because the list structure is the building block for complex data structures, including Bioconductor class structures, this will serve you well throughout your work. You should get in the habit of checking the structure and navigating where each piece of data is stored in the data structure at each step in preprocessing your data. Also, by checking your data throughout preprocessing, you might find that there are bits of information tucked in your data at early stages that you aren't yet using. For example, many file formats for laboratory equipment include slots for information about the equipment and its settings during when running the sample. This information might be read in from the file into R, but you might not know it's there for you to use if you'd like, to help you in creating reproducible reports that include this metadata about the experimental equipment and settings.

First, you will want to figure out whether your data is stored in a generic list, or if it's stored in a specific class-based data structure, which means it will have a bit more of a standardized structure. To do this, you can run the `class` function on your data object. The output of this might be a single value (e.g., "list" [?]) or a short list. If it's a short list, it will include both the specific class of the object and, as you go down the list, the more general data structure types that this class is built on. For example, if the `class` function returns this list:

[Example list of data types---maybe some specific class, then "list"?]

it means that the data's in a class-based structure called ... which is built on the more general structure of a list. You can apply to this data any of the functions that are specifically built for ... data structures, but you can also apply functions built for the more general list data structure.

There are several tools you can use to explore data structured as lists in R. R lists can sometimes be very large—in terms of the amount of data stored in them—particularly for some types of biomedical data. With some of the tools covered in this subsection, that will mean that your first look might seem overwhelming. We'll also cover some tools, therefore, that will let you peel away levels of the data in a bit more manageable way, which you can use when you encounter list-structured data that at first feels overwhelming.

First, if your data is stored in a specific class-based data structure, there likely will also be help files specifically for the class structure that can help you navigate it and figure out where things are. [Example]

[More about exploring data in list structures.]

"Use [to select any number of elements from a vector. ... **Positive integers** return elements at the specified positions. ... **Negative integers** exclude elements at the specified positions... **Logical vectors** select elements where the corresponding logical vector is TRUE. This is probably the most useful type of subsetting because you can write an expression that uses a logical vector... If the

vector is named, you can also use **character vectors** to return elements with matching names.” (Wickham, 2019)

“Subsetting a list works in the same way as subsetting an atomic vector. Using [always returns a list; [[and \$... lets you pull out elements of a list.” (Wickham, 2019)

“[[is most important when working with lists because subsetting a list with [[always returns a smaller list. ... Because [[can return only a single item, you must use it with either a single positive integer or a single string.” (Wickham, 2019)

“\$ is a shorthand operator: x\$y is roughly equivalent to x[["y"]]. It’s often used to access variables in a data frame... The one important difference between \$ and [[is that \$ does (left-to-right) partial matching [which you likely want to avoid to be safe].” (Wickham, 2019)

“There are two additional subsetting operators, which are needed for S4 objects: @ (equivalent to \$), and slot() (equivalent to [[]).” (Wickham, 2019)

“The environment is the data structure that powers scoping. ... Understanding environments is not necessary for day-to-day use of R. But they are important to understand because they power many important features like lexical scoping, name spaces, and R6 classes, and interact with evaluation to give you powerful tools for making domain specific languages, like dplyr and ggplot2.” (Wickham, 2019)

“The job of an environment is to associate, of **bind**, a set of names to a set of values. You can think of an environment as a bag of names, with no implied order (i.e., it doesn’t make sense to ask which is the first element in an environment).” (Wickham, 2019)

“... environments have reference semantics: unlike most R objects, when you modify them, you modify them in place, and don’t create a copy.” (Wickham, 2019)

“As well as powering scoping, environments are also useful data structures in their own right because they have reference semantics. There are three common problems that they can help solve: **Avoiding copies of large data**. Since environments have reference semantics, you’ll never accidentally create a copy. But bare environments are painful to work with, so I instead recommend using R6 objects, which are built on top of environments. ...” (Wickham, 2019)

“Generally in R, functional programming is much more important than object-oriented programming, because you typically solve complex problems by decomposing them into simple functions, not simple objects. Nevertheless, there are important reasons to learn each of the three [object-oriented programming] systems [S3, R6, and S4]: S3 allows your functions to return rich results with user-friendly display and programmer-friendly internals. S3 is used throughout base R, so it’s important to master it if you want to extend base R functions to work with new types of input. R6 provides a standardised way to escape R’s copy-on-modify semantics. This is particularly important if you want to model objects that exist independently of R. Today, a common need for R6 is to model

data that comes from a web API, and where changes come from inside or outside R. S4 is a rigorous system that forces you to think carefully about program design. It's particularly well-suited for building large systems that evolve over time and will receive contributions from many programmers. This is why it's used by the Bioconductor project, so another reason to learn S4 is to equip you to contribute to that project." (Wickham, 2019)

"The main reason to use OOP is **polymorphism** (literally: many shapes). Polymorphism means that a developer can consider a function's interface separately from its implementation, making it possible to use the same function form for different types of input. This is closely related to the idea of **encapsulation**: the user doesn't need to worry about details of an object because they are encapsulated behind a standard interface. ... To be more precise, **OO** systems call the type of an object its **class**, and an implementation for a specific class is called a **method**. Roughly speaking, a class defines what an object is and methods describe what that object can do. The class defines the **fields**, the data possessed by every instance of that class. Classes are organised in a hierarchy so that if a method does not exist for one class, its parent's method is used, and the child is said to **inherit** behaviour. ... The process of finding the correct method given a class is called **method dispatch**." (Wickham, 2019)

"There are two main paradigms of object-oriented programming which differ in how methods and classes are related. In this book, we'll borrow the terminology of Extending R [Chambers 2016] and call these paradigms encapsulated and functional: In **encapsulated** OOP, methods belong to objects or classes, and method calls typically look like `object.method(arg1, arg2)`. This is called encapsulated because the object encapsulates both data (with fields) and behaviour (with methods), and is the paradigm found in most popular languages. In **functional** OOP, methods belong to **generic** functions, and method calls look like ordinary function calls: `generic(object, arg2, arg3)`. This is called functional because from the outside it looks like a regular function call, and internally the components are also functions." (Wickham, 2019)

"**S3** is R's first OOP system... S3 is an informal implementation of functional OOP and relies on common conventions rather than ironclad guarantees. This makes it easy to get started with, providing a low cost way of solving many simple problems. ... **S4** is a formal and rigorous rewrite of S3... It requires more upfront work than S3, but in return provides more guarantees and greater encapsulation. S4 is implemented in the base **methods** package, which is always installed with R." (Wickham, 2019)

"While everything is an object, not everything is object-oriented. This confusion arises because the base objects come from S, and were developed before anyone thought that S might need an OOP system. The tools and nomenclature evolved organically over many years without a single guiding principle. Most of the time, the distinction between objects and object-oriented objects is not important. But here we need to get into the nitty gritty details so we'll use the terms **base objects** and **OO objects** to distinguish them. ... Technically, the difference between base and OO objects is that OO objects have a 'class attribute'." (Wickham, 2019)

"An S3 object is a base type with at least a `class` attribute (other attributes may be used to store other data). ... An S3 object behaves differently from

its underlying base type whenever it's passed to a **generic** (short for generic function). ... A generic function defines an interface, which uses a different implementation depending on the class of an argument (almost always the first argument). Many base R functions are generic, including the important `print`... The generic is a middleman: its job is to define the interface (i.e., the arguments) then find the right implements for the job. The implementation for a specific class is called a **method**, and the generic finds the method by performing **method dispatch**.” (Wickham, 2019)

“If you have done object-oriented programming in other languages, you may be surprised to learn that S3 has no formal definition of a class: to make an object an instance of a class, you simply set the **class attribute**. ... You can determine the class of an S3 object with `class(x)`, and see if an object is an instance of a class using `inherits(x, "classname")`.” (Wickham, 2019)

“The job of an S3 generic is to perform method dispatch, i.e., find the specific implementation for a class.” (Wickham, 2019)

“An important new component of S4 is the **slot**, a named component of the object that is accessed using the specialised subsetting operator `@` (pronounced ‘at’). The set of slots, and their classes, forms an important part of the definition of an S4 class.” (Wickham, 2019)

“Given an S4 object you can see its class with `is()` and access slots with `@` (equivalent to `$`) and `slot()` (equivalent to `[[]]` ... Generally, you should only use `@` in your methods. If you’re working with someone else’s class, look for **accessor** functions that allow you to safely set and get slot values. ... Accessors are typically S4 generics allowing multiple classes to share the same external interface.” (Wickham, 2019)

“If you’re using an S4 class defined in a package, you can get help on it with `class?Person`. To get help for a method, put `?` in front of a call (e.g., `?age(john)`) and `?` will use the class of the arguments to figure out which help file you need.” (Wickham, 2019)

“Slots [in S4 objects] should be considered an internal implementation detail: they can change without warning and user code should avoid accessing them directly. Instead, all user-accessible slots should be accompanied by a pair of **accessors**. If the slot is unique to the class, this can just be a function... Typically, however, you’ll define a generic so that multiple classes can use the same interface” (?)

“The strictness and formality of S4 make it well suited for large teams. Since more structure is provided by the system itself, there is less need for convention, and new contributors don’t need as much training. S4 tends to require more upfront design than S3, and this investment is more likely to pay off on larger projects where greater resources are available. One large team where S4 is used to good effect is Bioconductor. Bioconductor is similar to CRAN: it’s a way of sharing packages amongst a wider audience. Bioconductor is smaller than CRAN (~1,300 versus ~10,000 packages, July 2017) and the packages tend to be more tightly integrated because of the shared domain and because Bioconductor has a stricter review process. Bioconductor packages are not required to use S4, but most will because the key data structures (e.g., `SummarizedExperiment`, `IRanges`, `DNAStringSet`) are built using S4.” (Wickham, 2019)

“The biggest challenge to using S4 is the combination of increased complexity and absence of a single source of documentation. S4 is a complex system and it can be challenging to use effectively in practice. This wouldn’t be such a problem if S4 wasn’t scattered through R documentation, books, and websites. S4 needs a book length treatment, but that book does not (yet) exist. (The documentation for S3 is no better, but the lack is less painful because S3 is much simpler.)” (Wickham, 2019)

3.5.8 *Interfacing between object-based and tidyverse workflows*

The tidyverse approach in R is based on keeping data in a dataframe structure. By keeping this common structure, the tidyverse allows for straightforward but powerful work with your data by chaining together simple, single-purpose functions. This approach is widely covered in introductory R programming courses and books. A great starting point is the book *R Programming for Data Science*, which is available both in print and freely online at [site]. Many excellent resources exist for learning this approach, and so we won’t recover that information here. Instead, we will focus on how to interface between this approach and the object-based approach that’s more common with Bioconductor packages. Bioconductor packages often take an object-based approach, and with good reason because of the complexity and size of many early versions of biomedical data in the preprocessing process. There are also resources for learning to use specific Bioconductor packages, as well as some general resources on Bioconductor, like *R Programming for Bioinformatics* [ref]. However, there are fewer resources available online that teach how to coordinate between these two approaches in a pipeline of code, so that you can leverage the needed power of Bioconductor approaches early in your pipeline, as you preprocess large and complex data, and then shift to use a tidyverse approach once your data is amenable to this more straightforward approach to analysis and visualization.

The heart of making this shift is learning how to convert data, when possible, from a more complex, class-type data structure (built on the flexible list data structure) to the simpler, more standardized two-dimensional dataframe structure that is required for the tidyverse approach. In this subsection, we’ll cover approaches for converting your data from Bioconductor data structures to dataframes.

If you are lucky, this might be very straightforward. A pair of packages called `broom` and `biobroom` have been created specifically to facilitate the conversion of data from more complex structures to dataframes. The `broom` package was created first, by David Robinson, to convert the data stored in the objects that are created by fitting statistical models into tidy dataframes. Many of the functions in R that run statistical tests or fit statistical models output results in a more complex, list-based data structure. These structures have nice “print” methods, so if fitting the model or running the test is the very last step of your pipeline, you can just read the printed output from R. However, often you want

to include these results in further code—for example, creating plots or tables that show results from several statistical tests or models. The `broom` package includes several functions for pulling out different bits of data that are stored in the complex data structure created by fitting the model or running the test and convert those pieces of data into a tidy dataframe. This tidy dataframe can then be easily used in further code using a tidyverse approach.

The `biobroom` package was created to meet a similar need with data stored in some of the complex structures commonly used in Bioconductor packages.

[More about `biobroom`.]

[How to convert data if there isn't a `biobroom` method.] If you are unlucky, there may not be a `broom` or `biobroom` method that you can use for the particular class-based data structure that your data's in, or it might be in a more general list, rather than a specific class with a `biobroom` method. In this case, you'll need to extract the data “by hand” to move it into a dataframe once your data is simple enough to work with using a tidyverse approach. If you've mastered how to explore data stored in a list (covered in the last subsection), you'll have a headstart on how to do this. Once you know where to find each element of the data in the structure of the list, you can assign these specific pieces to their own R objects using typical R assignment (e.g., with the `gets` arrow, `<-`, or with `=`, depending on your preferred R programming style). ...

3.5.9 Extras

[Comparison of complexity of biological systems versus complexity of code and algorithms for data pre-processing—for the later, nothing is unknowable or even unknown. Someone somewhere is guaranteed to know exactly how it works, what it's doing, and why. By contrast, with biological systems, there are still things that noone anywhere completely understands. It's helpful to remember that all code and algorithms for data pre-processing is knowable, and that the details are all there if and when you want to dig to figure out what's going on.]

[There are ways to fully package up and save the computer environment used to run a pipeline of pre-processing and analysis, including any system settings, all different software used in analysis steps, and so on. Some of the approaches that are being explored for this include the use of “containers”, including Docker containers. This does allow, typically, for full reproducibility of the workflow. However, this approach isn't very proactive in emphasizing the robustness of a workflow or its comprehensibility to others—instead, it makes the workflow reproducible by putting everything in a black box that must be carefully unpackaged and explored if someone wants to understand or adapt the pipeline.]

“Object-oriented design doesn't have to be over-complicated design, but we've observed that too often it is. Too many OO designs are spaghetti-like tangles of is-a and has-a relationships, or feature thick layers of glue in which many

of the objects seem to exist simply to hold places in a steep-sided pyramid of abstractions. Such designs are the opposite of transparent; they are (notoriously) opaque and difficult to debug." (Raymond, 2003)

"Unix programmers are the original zealots about modularity, but tend to go about it in a quiter way [that with OOP]. Keeping glue layers thin is part of it; more generally, our tradition teaches us to build lower, hugging the ground with algorithms and structures that are designed to be simple and transparent." (Raymond, 2003)

"A *standard* is a precise and detailed description of how some artifact is built or is supposed to work. Examples of software standards include programming languages (the definition of syntax and semantics), data formats (how information is represented), algorithmic processing (the steps necessary to do a computation), and the like. Some standards, like the Word .doc file format, are *de facto* standards—they have no official standing but everyone uses them. The word 'standard' is best reserved for formal descriptions, often developed and maintained by a quasi-neutral party like a government or a consortium, that define how something is built or operated. The definition is sufficiently complete and precise that separate entities can interact or provide independent implementations. We benefit from hardware standards all the time, though we may not notice how many there are. If I buy a new television set, I can plug it into the electrical outlets in my home thanks to standards for the size and shape of plugs and the voltage they provide. The set itself will receive signals and display pictures because of standards for broadcast and cable television. I can plug other devices into it through standard cables and connectors like HDMI, USB, S-video and so on. But every TV needs its own remote control and every cell phone needs a different charger because those have not been standardized. Computing has plenty of standards as well, including character sets like ASCII and Unicode, programming languages like C and C++, algorithms for encryption and compression, and protocols for exchanging information over networks." (Kernighan, 2011)

"Standards are important. They make it possible for independently created things to cooperate, and they open an area to competition from multiple suppliers, while proprietary systems tend to lock everyone in. ... Standards have disadvantages, too—a standard can impede progress if it is inferior or outdated yet everyone is forced to use it. But these are modest drawbacks compared to the advantages." (Kernighan, 2011)

"A *class* is a blueprint for constructing a particular package of code and data; each variable created according to a class's blueprint is known as an *object* of that class. Code outside of a class that creates and uses an object of that class is known as a *client* of the class. A *class declaration* names the class and lists all of the *members*, or items inside that class. Each item is either a *data member*—a variable declared within the class—or a *method* (also known as a *member function*), which is a function declared within the class. Member functions can include a special type called a *constructor*, which has the same name as the class and is invoked implicitly when an object of the class is declared. In addition to the normal attributes of a variable or function declaration (such as type, and for functions, the parameter list), each member has an *access specifier*, which indicates what functions can access the member. A *public member* can be accessed by any code using the object: code inside the class, a client of the class, or code in a *subclass*, which is

a class that ‘inherits’ all the code and data of an existing class. A *private member* can be accessed only by the code inside the class. *Protected members* ... are similar to private members, except that methods in subclasses can also reference them. Both private and protected members, though, are inaccessible from client code.” (Spraul, 2012)

“An object should be a meaningful, closely knit collection of data and code that operates on the data.” (Spraul, 2012)

“Recognizing a situation in which a class would be useful is essential to reaching the higher levels of programming style, but it’s equally important to recognize situations in which a class is going to make things worse.” (Spraul, 2012)

“The word *encapsulation* is a fancy way of saying that classes put multiple pieces of data and code together in a single package. If you’ve ever seen a gelatin medicine capsule filled with little spheres, that’s a good analogy: The patient takes one capsule and swallows all the individual ingredient spheres inside. ... From a problem-solving standpoint, encapsulation allows us to more easily reuse the code from previous problems to solve current problems. Often, even though we have worked on a problem similar to our current project, reusing what we learned before still takes a lot of work. A fully encapsulated class can work like an external USB drive; you just plug it in and it works. For this to happen, though, we must design the class correctly to make sure that the code and data is truly encapsulated and as independent as possible from anything outside of the class. For example, a class that references a global variable can’t be copied into a new project without copying the global variable, as well.” (Spraul, 2012)

“Beyond reusing classes from one program to the next, classes offer the potential for a more immediate form of code reuse: inheritance. ... Using inheritance, we create parent classes with methods common to two or more child classes, thereby ‘factoring out’ not just a few lines of code [as with helper functions in procedural code] but whole methods.” (Spraul, 2012)

“One technique we’re returned to again and again is dividing a complex problem into smaller, more manageable pieces. Classes are great at dividing programs up into functional units. Encapsulation not only holds data and code together in a reusable package; it also cordons off that data and code from the rest of the program, allowing us to work on that class, and everything else separately. The more classes we make in a program, the greater the problem-dividing effect.” (Spraul, 2012)

“Some people use the terms *information hiding* and *encapsulation* interchangeable, but we’ll separate the ideas here. As described previously, encapsulation is packaging data and code together. Information hiding means separating the interface of a data structure—the definition of the operations and their parameters—from the implementation of a data structure, or the code inside the functions. If a class has been written with information hiding as a goal, then it’s possible to change the implementation of the methods without requiring any changes in the client code (the code that uses the class). Again, we have to be clear on the term *interface*; this means not only the name of the methods and their parameter list but also the explanation (perhaps expressed in code documentation) of what the different methods do. When we talk about changing the implementation without changing the interface, we mean that we change *how* the class methods work but

not what they do. Some programming authors have referred to this as a kind of implicit contract between the class and the client: The class agrees never to change the effects of existing operations, and the client agrees to use the class strictly on the basis of its interface and to ignore any implementation details.” (Spraul, 2012)

“So how does information hiding affect problem solving? The principle of information hiding tells the programmer to put aside the class implementation details when working on the client code, or more broadly, to be concerned about a particular class’s implementation only when working inside that class. When you can put implementation details out of your mind, you can eliminate distracting thoughts and concentrate on solving the problem at hand.” (Spraul, 2012)

“A final goal of a well-designed class is expressiveness, or what might be broadly called writability—the ease with which code can be written. A good class, once written, makes the rest of the code simpler to write in the way that a good function makes code simpler to write. Classes effectively extend a language, becoming high-level counterparts to basic low-level features such as loops, if statements, and so forth. ... With classes, programming actions that previously took many steps can be done in just a few steps or just one.” (Spraul, 2012)

“Right now, in labs across the world, machines are sequencing the genomes of the life on earth. Even with rapidly decreasing costs and huge technological advancements in genome sequencing, we’re only seeing a glimpse of the biological information contained in every cell, tissue, organism, and ecosystem. However, the smidgen of total biological information we’re gathering amounts to mountains of data biologists need to work with. At no other point in human history has our ability to understand life’s complexities been so dependent on our skills to work with and analyze data.” (Buffalo, 2015)

“Bioinformaticians are concerned with deriving biological understanding from large amounts of data with specialized skills and tools. Early in biology’s history, the datasets were small and manageable. Most biologists could analyze their own data after taking a statistics course, using Microsoft Excel on a personal desktop computer. However, this is all rapidly changing. Large sequencing datasets are widespread, and will only become more common in the future. Analyzing this data takes different tools, new skills, and many computers with large amounts of memory, processing power, and disk space.” (Buffalo, 2015)

“In a relatively short period of time, sequencing costs dropped drastically, allowing researchers to utilize sequencing data to help answer important biological questions. Early sequencing was low-throughput and costly. Whole genome sequencing efforts were expensive (the human genome cost around \$2.7 billion) and only possible through large collaborative efforts. Since the release of the human genome, sequencing costs have decreased exponentially until about 2008 ... With the introduction of next-generation sequencing technologies, the cost of sequencing a megabase of DNA dropped even more rapidly. At this crucial point, a technology that was only affordable to large collaborative sequencing efforts (or individual researchers with very deep pockets) became affordable to researchers across all of biology. ... What was the consequence of this drop in sequencing costs due to these new technologies? As you may have guessed, lots and lots of data. Biological databases have swelled with data after exponential growth.

Whereas once small databases shared between collaborators were sufficient, now petabytes of useful data are sitting on servers all over the world. Key insights into biological questions are stored not just in the unanalyzed experimental data sitting on your hard drive, but also spinning around a disk in a data center thousands of miles away." (Buffalo, 2015)

"To make matters even more complicated, new tools for analyzing biological data are continually being created, and their underlying algorithms are advancing. A 2012 review listed over 70 short-read mappers ... Likewise, our approach to genome assembly has changed considerably in the past five years, as methods to assemble long sequences (such as overlap-layout-consensus algorithms) were abandoned with the emergence of short high-throughput sequencing reads.

Now, advances in sequencing chemistry are leading to longer sequencing read lengths and new algorithms are replacing others that were just a few years old. Unfortunately, this abundance and rapid development of bioinformatics tools has serious downsides. Often, bioinformatics tools are not adequately benchmarked, or if they are, they are only benchmarked in one organism. This makes it difficult for new biologists to find and choose the best tool to analyze their data. To make matters more difficult, some bioinformatics programs are not actively developed so that they lose relevance or carry bugs that could negatively affect results. All of this makes choosing an appropriate bioinformatics program in your own research difficult. More importantly, it's imperative to critically assess the output of bioinformatics programs run on your own data." (Buffalo, 2015)

"With the nature of biological data changing so rapidly, how are you supposed to learn bioinformatics? With all of the tools out there and more continually being created, how is a biologist supposed to know whether a program will work appropriately on her organism's data? The solution is to approach bioinformatics as a bioinformatician does: try stuff, and assess the results. In this way, bioinformatics is just about having the skills to experiment with data using a computer and understanding your results. The experimental part is easy: this comes naturally to most scientists. The limiting factor for most biologists is having the data skills to freely experiment and work with large data on a computer." (Buffalo, 2015)

"Unfortunately, many of the biologist's common computational tools can't scale to the size and complexity of modern biological data. Complex data formats, interfacing numerous programs, and assessing software and data make large bioinformatics datasets difficult to work with." (Buffalo, 2015)

"In 10 years, bioinformaticians may only be using a few of the bioinformatics software programs around today. But we most certainly will be using data skills and experimentation to assess data and methods of the future." (Buffalo, 2015)

"Biology's increasing use of large sequencing datasets is changing more than the tools and skills we need: it's also changing how reproducible and robust our scientific findings are. As we utilize new tools and skills to analyze genomics data, it's necessary to ensure that our approaches are still as reproducible and robust as any other experimental approaches. Unfortunately, the size of our data and the complexity of our analysis workflows make these goals especially difficult in genomics." (Buffalo, 2015)

"The requisite of reproducibility is that we share our data and methods. In the pre-genomics era, this was much easier. Papers could include detailed method

summaries and entire datasets—exactly as Kreitman's 1986 paper did with a 4,713bp *Adh* gene flanking sequence (it was embedded in the middle of the paper). Now papers have long supplementary methods, code, and data. Sharing data is no longer trivial either, as sequencing projects can include terabytes of accompanying data. Reference genomes and annotation datasets used in analyses are constantly updated, which can make reproducibility tricky. Links to supplemental materials, methods, and data on journal websites break, materials on faculty websites disappear when faculty members move or update their sites, and software projects become stale when developers leave and don't update code. ... Additionally, the complexity of bioinformatics analyses can lead to findings being susceptible to errors and technical confounding. Even fairly routine genomics projects can use dozens of different programs, complicated input parameter combinations, and many sample and annotation datasets; in addition, work may be spread across servers and workstations. All of these computational data-processing steps create results used in higher-level analyses where we draw our biological conclusions. The end result is that research findings may rest on a rickety scaffold of numerous processing steps. To make matters worse, bioinformatics workflows and analyses are usually only run once to produce results for a publication, and then never run or tested again. These analyses may rely on very specific versions of all software used, which can make it difficult to reproduce on a different system. In learning bioinformatics data skills, it's necessary to concurrently learn reproducibility and robust best practices.” (Buffalo, 2015)

“When we are writing code in a programming language, we work most of the time with RAM, combining and restructuring data values to produce new values in RAM. ... The computer memory in RAM is a series of 0's and 1's, just like the computer memory used to store files in mass storage. In order to work with data values, we need to get those values into RAM in some format. At the basic level of representing a single number or a single piece of text, the solution is the same as it was in Chapter 5 [on file formats for mass storage]. Everything is represented as a pattern of bits, using various numbers of bytes for different sorts of values. In R, in an English locale, and on a 32-bit operating system, a single character usually takes up one byte, an integer takes up four bytes, and a real number 8 bytes. Data values are stored in different ways depending on the **data type**—whether the values are numbers or texts.” (Murrell, 2009)

“Although we do not often encounter the details of the memory representation, except when we need a rough estimate of how much RAM a data set might require, it is important to keep in mind what sort of data type we are working with because the computer code that we will produce different results for different data types. For example, we can only calculate an average if we are dealing with values that have been stored as text.” (Murrell, 2009)

“Another important issue is how *collections* of values are stored in memory. The tasks that we will consider will typically involve working with an entire data set, or an entire variable from a data set, rather than just a single value, so we need to have a way to represent several related values in memory. This is similar to the problem of deciding on a storage format for a data set... However, rather than talking about different file formats, [in this case] we will talk about different **data structures** for storing a collection of data values in RAM. ... It will be important to always keep close in our minds what data type we are working with and what sort of data structure we are working with.” (Murrell, 2009)

“Every individual data value has a data type that tells us what sort of value it is. The most common data types are numbers, which R calls **numeric values**, and text, which R calls **character values**.” (Murrell, 2009)

“**Vectors:** A collection of values that all have the same data type. The **elements** of a vector are all numbers, giving a **number vector**, or all character values, giving a **character vector**.” (Murrell, 2009)

“**Data frames:** A collection of vectors that all have the same length. This is like a matrix, except that each column can contain a different data type.” (Murrell, 2009)

“**Lists:** A collection of data structures. The **components** of a list can be simply vectors—similar to a data frame, but with each column allowed to have a different length. However, a list can also be a much more complicated structure. This is a very flexible data structure. Lists can be used to store any combination of data values together.” (Murrell, 2009)

“Notice the way that lists are displayed. The first component of the list starts with the component index, [[1]], followed by the contents of this component...The second component of the list starts with the component index [[2]] followed by the contents of this component...” (Murrell, 2009)

“A list is a very flexible data structure. It can have any number of **components**, each of which can be any data structure of any length or size. A simple example is a data-frame-like structure where each column can have a different length, but much more complex structures are also possible. For example, it is possible for a component of a list to be another list.” (Murrell, 2009)

“Anyone who has worked with a computer should be familiar with the idea of a list containing another list because a directory or folder of files has this sort of structure: a folder contains multiple files of different kinds and sizes and a folder can contain other folders, which can contain more files or even more folders, and so on. Lists allow for this kind of hierarchical structure.” (Murrell, 2009)

“One of the most basic ways that we can manipulate data structures is to **subset** them—select a smaller portion from a larger data structure. This is analogous to performing a query on a database. ... R has very powerful mechanisms for subsetting... A subset from a vector may be obtained by appending an **index** within square brackets to the end of a symbol name. ... The index can be a vector of any length ... The index does not have to be a contiguous sequence, and it can include repetitions... As well as using integers for indices, we can use logical values... A data frame can also be indexed using square brackets, though slightly differently because we have to specify both which rows *and* which columns we want ... When a data structure has named components, a subset may be selected using those names.” (Murrell, 2009)

“Single square bracket subsetting on a data frame is like taking an egg container that contains a dozen eggs and chopping up the *container* so that we are left with a smaller egg container that contains just a few eggs. Double square bracket subsetting on a data frame is like selecting just one egg from an egg container.” (Murrell, 2009)

"We can often get some idea of what sort of data structure we are working with by simply viewing how the data are displayed on screen. However, a more definitive answer can be obtained by calling the `class()` function. ... Many R functions return a data structure that is not one of the basic data structures that we have already seen [like the 'xtabs' and 'table' classes]. ... We have not seen either of these data structures before. However, much of what we know about working with the standard data structures ... will work with any new class that we encounter. For example, it is usually possible to subset any class using the standard square bracket syntax. ... Where appropriate, arithmetic and comparisons will also generally work... Furthermore, if necessary, we can often resort to coercing a class to something more standard and familiar." (Murrell, 2009)

"Dates are an important example of a special data structure. Representing dates as just text is convenient for humans to view, but other representations are better for computers to work with. ... Having a special class for dates means that we can perform tasks with dates, such as arithmetic and comparisons, in a meaningful way, something we could not do if we stored the date as just a character value." (Murrell, 2009)

"The Date class stores date values as integer values, representing the number of days since January 1st 1970, and automatically converts the numbers to a readable text value to display the dates on the screen." (Murrell, 2009)

"When working with anything but tiny data sets, basic features of the data set cannot be determined by just viewing the data values. [There are] a number of functions that are useful for obtaining useful summary features from a data structure. The `summary()` function produces summary information for a data structure... The `length()` function is useful for determining the number of values in a vector or the number of components in a list. ... The `str()` function (short for 'structure') is useful when dealing with large objects because it only shows a sample of the values in each part of the object, although the display is very low-level so it may not always make things clearer. ... Another function that is useful for inspecting a large object is the `head()` function. This shows just the first few elements of an object, so we can see the basic structure without seeing all of the values." (Murrell, 2009)

"Generic functions ... will accept many different data structures as arguments. ... a generic function adapts itself to the data structure it is given. Generic functions do different things when given different data structures." (Murrell, 2009)

"An example of a generic function is the `summary()` function. The result of a call to `summary()` will depend on what sort of data structure we provide." (Murrell, 2009)

"Generic functions are another reason why it is easy to work with data in R; a single function will produce a sensible result no matter what data structure we provide. However, generic functions are also another reason why it is so important to be aware of what data structures we are working with. Without knowing what sort of data we are using, we cannot know what sort of result to expect from a generic function." (Murrell, 2009)

"R has become very popular and is now being used for projects that require substantial software engineering as well as its continued widespread use as an

interactive environment for data analysis. This essentially means that there are two masters—*reliability* and *ease of use*. S3 is indeed easy to use, but can be made unreliable through nothing other than bad luck, or a poor choice of names, and hence is not a suitable paradigm for constructing large systems. S4, on the other hand, is better suited for developing large software projects but has an increased complexity of use.” (Gentleman, 2008)

“Object-oriented programming has become a widely used and valuable tool for software engineering. Much of its value derives from the fact that it is often easier to design, write, and maintain software when there is some clear separation of the data representation from the operations that are to be performed on it. In an OOP system, real physical things … are generally represented by classes, and methods (functions) are written to handle the different manipulations that need to be performed on the objects.” (Gentleman, 2008)

“The views that many people have of OOP have been based largely on exposure to languages like Java, where the system can be described as class-centric. In a class-centric system, classes define objects and are repositories for the methods that act on those objects. In contrast, languages such as … R separate the class specification from the specification of generic functions, and could be described as function-centric systems.” (Gentleman, 2008)

“The genome of every organism is encoded in chromosomes that consist of either DNA or RNA. High throughput sequencing technology has made it possible to determine the sequence of the genome for virtually any organism, and there are many that are currently available. … However, in many cases, either the exact nucleotide at any location is unknown, or is variable, and the International Union of Pure and Applied Chemistry (IUPAC) has provided a standard nomenclature suitable for representing such sequences. The alphabet for dealing with protein sequences is based on the 20 amino acids. … The basic class used to hold strings [in the **Biostrings** package] is the *BString* class, which has been designed to be efficient in its handling of large character strings. Subclasses include *DNAString*, *RNAString*, and *AAString* (for holding amino acid sequences). The *BStringViews* class holds a set of views on a single *BString* instance; each view is essentially a substring of the underlying *BString* instance. Alignments are stored using the *BStringAlign* class.” (Gentleman, 2008) [More on functions that work with these classes on p. 171]

“A number of complete genomes, represented as *DNAString* objects, are provided through the Bioconductor project. They rely on the infrastructure in the **BSgenome** package, and all such packages have names that begin with *BSgenome*. You can find the list of available genomes using the *available.genomes* function.” (Gentleman, 2008)

“Atomic vectors are the most basic of all data structures. An atomic vector contains some number of values of the same type; that number could be zero. Atomic vectors can contain integers, doubles, logicals or character strings. Both complex numbers and raw (pure bytes) have atomic representations … Character vectors in the S language are vectors of character strings, not the vectors of characters. For example, the string ‘super’ would be represented as a character vector of length one, not of length five…” (Gentleman, 2008)

“Lists can be used to store items that are not all of the same type. … Lists are also referred to as generic vectors since they share many of the properties of vectors, but the elements are allowed to have different types.” (Gentleman, 2008)

"Lists can be of any length, and the elements of a list can be named, or not. Any R object can be an element of a list, including another list..." (Gentleman, 2008)

"A `data.frame` is a special kind of list. Data frames were created to provide a common structure for storing rectangular data sets and for passing them to different functions for modeling and visualization. In many cases a data set can be thought of as a rectangular structure with rows corresponding to cases and columns corresponding to the different variables that were measured on each of the cases. One might think that a matrix would be the appropriate representation, but that is only true if all of the variables are of the same type, and this is seldom the case." (Gentleman, 2008)

"[Data frames] are essentially a list of vectors, with one vector for each variable. It is an error if the vectors are not all of the same length." (Gentleman, 2008)

"Sometimes it will be helpful to find out about an object. Obvious functions to try are `class` and `typeof`. But many find that both `str` and `object.size` are more useful. ... The functions `head` and `tail` are convenience functions that list the first few, or the last few, rows of a matrix." (Gentleman, 2008)

"The S language has its roots in the Algol family of languages and has adopted some of the general vector subsetting and subscripting techniques that were available in languages such as APL. This is perhaps one area where programmers more familiar with other languages fail to make appropriate use of the available functionality. ... There are slight differences between subsetting of vectors, arrays, lists, `data.frames`, and environments that can sometimes catch the unwary. But there are also many commonalities. ... Subsetting can be carried out by three different operators: the single square bracket `[`, the double square bracket `[[`, and the dollar, `$`. We note that each of these three operators are actually generic functions and users can write methods that extend and override them... One way of describing the behavior of the single bracket operator is that the type of the return value matches the type of the value it is applied to. Thus, a single bracket subset of a list is a list itself. ... Both `[[` and `$` extract a single value. There are some differences between the two; `$` does not evaluate its second argument while `[[` does, and hence one can use expressions. The `$` operator uses partial matching when extracting named elements but `[` and `[[` do not." (Gentleman, 2008)

"Subsetting plays two roles in the S language. One is an extraction role, where a subset of a vector is identified by a set of supplied indices and the resulting subset is returned as a value. Venables and Ripley (2000) refer to this as *indexing*. The second purpose is subset assignment, where the goal is to identify a subset of values that should have their values changed; we call this subset assignment." (Gentleman, 2008)

"There are four basic types of subscript indices: positive integers, negative integers, logical vectors, and character vectors. These four types cannot be mixed... For matrix and array subscripting, one can use different types of subscripts for the different dimensions. Not all vectors, or recursive objects, support all types of subscripting indices. For example, atomic vectors cannot be subscripted using `$`, while environments cannot be subscripted using `[[`." (Gentleman, 2008)

"In bioinformatics, the plain-text data we work with is often encoded in ASCII. ASCII is a character encoding scheme that uses 7 bits to represent 128 different

values, including letters (upper- and lowercase), numbers, and special nonvisible characters. While ASCII only uses 8 bits, nowadays computers use an 8-bit byte (a unit representing 8 bits) to store ASCII characters. More information about ASCII is available in your terminal through `man ascii`.” (Buffalo, 2015)

“Some files will have non-ASCII encoding schemes, and may contain special characters. The most common character encoding scheme is UTF-8, which is a superset of ASCII but allows for special characters.” (Buffalo, 2015)

“Bioinformatics data is often text—for example, the As, Cs, Ts, and Gs in sequencing read files or reference genomes, or tab-delimited files for gene coordinates. The text data in bioinformatics is often large, too (gigabytes or more that can’t fit into your computer’s memory at once). This is why Unix’s philosophy of handling text streams is useful to bioinformatics: text streams allow us to do processing on a *stream* of data rather than holding it all in memory.” (Buffalo, 2015)

“Exploratory data analysis plays an integral role throughout an entire bioinformatics project. Exploratory data analysis skills are just as applicable in analyzing intermediate bioinformatics data (e.g., are fewer reads from this sequencing lane aligning?) as they are in making sense of results from statistical analyses (e.g., what’s the distribution of these p-values, and do they correlate with possible confounders like gene length?). These exploratory analyses need not be complex or exceedingly detailed (many patterns are visible with simple analyses and visualization); it’s just about wanting to look into the data and having the skill set to do so.” (Buffalo, 2015)

“Functions like `table()` are generic—they are designed to work with objects of all kinds of classes. Generic functions are also designed to do the right thing depending on the class of the object they’re called on (in programming lingo, we say that the function is *polymorphic*).” (Buffalo, 2015)

“It’s quite common to encounter genomics datasets that are difficult to load into R because they’re large files. This is either because it takes too long to load the entire dataset into R, or your machine simply doesn’t have enough memory. In many cases, the best strategy is to reduce the size of your data somehow: summarizing data in earlier processing steps, omitting unnecessary columns, splitting your data into chunks (e.g., working with a chromosome at a time), or working on a random subset of your data. Many bioinformatics analyses do not require working on an entire genomic dataset at once, so these strategies can work quite well. These approaches are also the only way to work with data that is truly too large to fit in your machine’s memory (apart from getting a machine with more memory).” (Buffalo, 2015)

“If your data is larger than the available memory on your machine, you’ll need to use a strategy that keeps the bulk of your data out of memory, but still allows for each access from R. A good solution for moderately large data is to use SQLite and query out subsets for computation using the R package `RSQLite`. ... Finally ... many Unix data tools have versions that work on gzipped files: `zless`, `zcat` (`gzcat` on BSD-derived systems like Mac OS X), and others. Likewise, R’s data-reading functions can also read gzipped files directly—there’s some slight performance gains in reading in gzipped files, as there are fewer bytes to read off of (slow) hard disks.” (Buffalo, 2015)

"Quite often, data we load in to R will be in the wrong *shape* for what we want to do with it. Tabular data can come in two different formats: *long* and *wide*. ... In many cases, data is recorded by humans in wide format, but we need data in long format when working with and plotting statistical modeling functions." (Buffalo, 2015)

"Exploratory data analysis emphasizes visualization as the best tool to understand and explore our data—both to learn what the data says and what its limitations are." (Buffalo, 2015)

"R vectors require all elements to have the same data type (that is, vectors are *homogeneous*). They only support the six data types discussed earlier (integer, double, character, logical, complex, and raw). In contrast, R's lists are more versatile: Lists can contain elements of different types (they are *heterogeneous*); Elements can be *any* object in R (vectors with different types, other lists, environments, dataframes, matrices, functions, etc.); Because lists can store other lists, they allow for storing data in a recursive way (in contrast, vectors cannot contain other vectors)." (Buffalo, 2015)

"The versatility of lists make them indispensable in programming and data analysis with R." (Buffalo, 2015)

"As with R's vectors, we can extract subsets of a list or change values of specific elements using indexing. However, accessing elements from an R list is slightly different than with vectors. Because R's list can contain objects with different types, a subset containing multiple list elements could contain objects with different types. Consequently, the only way to return a subset of more than one list element is with another list. As a result, there are two indexing operators for lists: one for accessing a subset of multiple elements as a list (the single bracket...) and one for accessing an element within a list (the double bracket...)." (Buffalo, 2015)

"Because R's lists can be nested and contain any type of data, list-based data structures can grow to be quite complex. In some cases, it can be difficult to understand the overall structure of some lists. The function `str()` is a convenient R function for inspecting complex data structures. `str()` prints a line for each contained data structure, complete with its type, length (or dimensions), and the first few elements it contains. ... For deeply nested lists, you can simplify `str()`'s output by specifying the maximum depth of nested structure to return with `str()`'s second argument, `max.level`. By default, `max.level` is NA, which returns all nested structures." (Buffalo, 2015)

"Understanding R's data structures and how subsetting works are fundamental to having the freedom in R to explore data any way you like." (Buffalo, 2015)

"Some of Bioconductor's core packages: **GenomicRanges**: Used to represent and work with genomic ranges; **GenomicFeatures**: used to represent and work with ranges that represent gene models and other features of a genome (genes, exons, UTRs, transcripts, etc.); **Biostrings** and **BSgenome**: Used for manipulating genome sequence data in R... **rtracklayer**: Used for reading in common bioinformatics formats like BED, GTF/GFF, and WIG." (Buffalo, 2015)

"The **GenomicRanges** package introduces a new class called **GRanges** for storing genomic ranges. The **GRanges** builds off of **IRanges**. **IRanges** objects are

used to store ranges of genomic regions on a single sequence, and GRanges objects contain the two other pieces of information necessary to specify a genomic location: sequence name (e.g., which chromosome) and strand. GRanges objects also have *metadata columns*, which are the data linked to each genomic range.” (Buffalo, 2015)

“All metadata attached to a GRanges object are stored in a DataFrame, which behaves identically to R’s base data.frame but supports a wider variety of column types. For example, DataFrames allow for run-length encoded vectors to save memory ... in practice, we can store any type of data: identifiers and names (e.g., for genes, transcripts, SNPs, or exons), annotation data (e.g., conservation scores, GC content, repeat content, etc.), or experimental data (e.g., if ranges correspond to alignments, data like mapping quality and the number of gaps). ... the union of genomic location with any type of data is what makes GRanges so pwoerful.” (Buffalo, 2015)

Some object classes in BioConductor:

- eSet from Biobase
- Sequence from IRanges
- MAlist from limma
- ExpressionSet from Biobase

You can use the getSlots function with S4 objects to see all the slots within the object.

“Methods and classes in the S language are essentially programming concepts to enable good organization of functions and of general objects, respectively.” (Chambers, 2006)

“Programming in R starts out usually as writing functions, at least once we get past the strict cut-and-paste stage. Functions are the actions of the language; calls to them express what the user wants to happen. The arguments to the functions and the values returned by function calls are the objects. These objects represent everything we deal with. Actions create new objects (such as summaries and models) or present the information in the objects (by plots, printed summaries, or interfaces to other software). R is a functional, object-based system where users program to extend the capacity of the system in terms of new functionality and new kinds of objects.” (Chambers, 2006)

“Languages to which the object-oriented programming (OOP) term is typically applied mostly support what might better be called *class-oriented programming*, well-known examples being C++ and Java. In these languages the essential programming unit is the class definition. Objects are generated as instances of a class and computations on the objects consist of *invoking methods on* that object. Depending on how strict the language is, all or most of the computations must be expressed in this form. Method invocation is an operator, operating on an instance of a class. Software organization is essentially simple and hierarchical, in the sense that all methods are defined as part of a particular class. That’s not how S works; as mentioned, the first and most important programming unit is the function. From the user’s perspective, it’s all done by calling functions (even if some of the functions are hidden in the form of operators). Methods and classes

provide not class-oriented programming but function- and class-oriented programming. It's a richer view, but also a more complicated one." (Chambers, 2006)

"A generic function will collect or *cache* all the methods for that function belonging to all the R packages that have been loaded in the session. When the function is called, the R evaluator then *selects* a method from those available, by examining how well different methods match the actual arguments in the call." (Chambers, 2006)

"From the users' view, the generic function has (or at least should have) a natural definition in terms of what it is intended to do: `plot()` displays graphics to represent an object or the relation between two objects; arithmetic operators such as '`+`' carry out the corresponding intuitive numerical computations or extensions of those. Methods should map those intuitive notions naturally and reliably into the concepts represented by the class definitions." (Chambers, 2006)

"The class definition contains a definition of the slots in objects from the class and other information of various kinds, but the most important information for the present discussion defines what other classes this class extends; that is, the inheritance or to use the most common term, the *superclasses* of this class. In R, the names of the superclasses can be seen as the value of `extends(thisClass)`. By definition, an object from any class can be used in a computation designed for any of the superclasses of that class. Therefore, it's precisely the superclasses of the class of an argument that define candidate methods in a particular function call." (Chambers, 2006)

"Conceptually, a generic function extends the idea of a function in R by allowing different methods to be selected corresponding to the classes of the objects supplied as arguments in a call to the function." (Chambers, 2006)

The code for different implementations of a method (in other words, different ways it will run with new object classes) can come in different R packages. This allows a developer to add his or her own applications of methods, suited for object classes he or she creates.

A class defines the structure for a way of storing data. When you create an object that follows this structure, it's an instance of that class. The new function is used to create new instances of a class.

When a generic function determines what code to run based on the class of the object, it's called *method dispatch*.

By using the accessor function, instead of `@`, your code will be more robust to changes that the developers make. They will be sensitive to insuring that the accessor function for a particular part of the data continues to work regardless of changes they make to the structure that is used to store data in objects in that class. They will be less committed, however, to keeping the same slots, and in the same positions, as they develop the software. The "contract" with the user is through the accessor function, in other words, rather than through the slot name in the object.

"Bioconductor is an open-source, open-development software project for the analysis and comprehension of high-throughput data in genomics and molecular

biology. The project aims to enable interdisciplinary research, collaboration and rapid development of scientific software. Based on the statistical programming language R, Bioconductor comprises 934 interoperable packages contributed by a large, diverse community of scientists. Packages cover a range of bioinformatic and statistical applications. They undergo formal initial review and continuous automated testing.” (Huber et al., 2015a)

“Bioconductor provides core data structures and methods that enable genome-scale analysis of highthroughput data in the context of the rich statistical programming environment offered by the R project. It supports many types of high-throughput sequencing data (including DNA, RNA, chromatin immunoprecipitation, Hi-C, methylomes and ribosome profiling) and associated annotation resources; contains mature facilities for microarray analysis; and covers proteomic, metabolomic, flow cytometry, quantitative imaging, cheminformatic and other high-throughput data. Bioconductor enables the rapid creation of workflows combining multiple data types and tools for statistical inference, regression, network analysis, machine learning and visualization at all stages of a project from data generation to publication.” (Huber et al., 2015a)

“Bioconductor is also a flexible software engineering environment in which to develop the tools needed, and it offers users a framework for efficient learning and productive work. The foundations of Bioconductor and its rapid coevolution with experimental technologies are based on two motivating principles. The first is to provide a compelling user experience. Bioconductor documentation comes at three levels: workflows that document complete analyses spanning multiple tools; package vignettes that provide a narrative of the intended uses of a particular package, including detailed executable code examples; and function manual pages with precise descriptions of all inputs and outputs together with working examples. In many cases, users ultimately become developers, making their own algorithms and approaches available to others. The second is to enable and support an active and open scientific community developing and distributing algorithms and software in bioinformatics and computational biology. The support includes guidance and training on software development and documentation, as well as the use of appropriate programming paradigms such as unit testing and judicious optimization. A primary goal is the distributed development of interoperable software components by scientific domain experts. **In part we achieve this by urging the use of common data structures that enable workflows integrating multiple data types and disciplines.** To facilitate research and innovation, we employ a high-level programming language. This choice yields rapid prototyping, creativity, flexibility and reproducibility in a way that neither point-and-click software nor a general-purpose programming language can. We have embraced R for its scientific and statistical computing capabilities, for its graphics facilities and for the convenience of an interpreted language. R also interfaces with low-level languages including C and C++ for computationally intensive operations, Java for integration with enterprise software and JavaScript for interactive web-based applications and reports.” (Huber et al., 2015a)

“Case study: high-throughput sequencing data analysis. Analysis of large-scale RNA or DNA sequencing data often begins with aligning reads to a reference genome, which is followed by interpretation of the alignment patterns. Alignment is handled by a variety of tools, whose output typically is delivered as

a BAM file. The Bioconductor packages Rsamtools and GenomicAlignments provide a flexible interface for importing and manipulating the data in a BAM file, for instance for quality assessment, visualization, event detection and summarization. The regions of interest in such analyses are genes, transcripts, enhancers or many other types of sequence intervals that can be identified by their genomic coordinates. Bioconductor supports representation and analysis of genomic intervals with a ‘Ranges’ infrastructure that encompasses data structures, algorithms and utilities including arithmetic functions, set operations and summarization (Fig. 1). It consists of several packages including IRanges, GenomicRanges, GenomicAlignments, GenomicFeatures, VariantAnnotation and rtracklayer. The packages are frequently updated for functionality, performance and usability. **The Ranges infrastructure was designed to provide tools that are convenient for end users analyzing data while retaining flexibility to serve as a foundation for the development of more complex and specialized software. We have formalized the data structures to the point that they enable interoperability, but we have also made them adaptable to specific use cases by allowing additional, less formalized userdefined data components such as application-defined annotation.** Workflows can differ vastly depending on the specific goals of the investigation, but a common pattern is reduction of the data to a defined set of ranges in terms of quantitative and qualitative summaries of the alignments at each of the sites. Examples include detecting coverage peaks or concentrations in chromatin immunoprecipitation–sequencing, counting the number of cDNA fragments that match each transcript or exon (RNA-seq) and calling DNA sequence variants (DNA-seq). Such summaries can be stored in an instance of the class GenomicRanges.” (Huber et al., 2015a)

“To facilitate the analysis of experiments and studies with multiple samples, Bioconductor defines the SummarizedExperiment class. The computed summaries for the ranges are compiled into a rectangular array whose rows correspond to the ranges and whose columns correspond to the different samples . . . For a typical experiment, there can be tens of thousands to millions of ranges and from a handful to hundreds of samples. The array elements do not need to be single numbers: the summaries can be multivariate. The SummarizedExperiment class also stores metadata on the rows and columns. Metadata on the samples usually include experimental or observational covariates as well as technical information such as processing dates or batches, file paths, etc. Row metadata comprise the start and end coordinates of each feature and the identifier of the containing polymer, for example, the chromosome name. Further information can be inserted, such as gene or exon identifiers, references to external databases, reagents, functional classifications of the region (e.g., from efforts such as the Encyclopedia of DNA Elements (ENCODE)) or genetic associations (e.g., from genome-wide association studies, the study of rare diseases, or cancer genetics). The row metadata aid integrative analysis, for example, when matching two experiments according to overlap of genomic regions of interest. Tight coupling of metadata with the data reduces opportunities for clerical errors during reordering or subsetting operations.” (Huber et al., 2015a)

“The integrative data container SummarizedExperiment. Its assays component is one or several rectangular arrays of equivalent row and column dimensions. Rows correspond to features, and columns to samples. The component rowData stores metadata about the features, including their genomic ranges. The colData component keeps track of samplelevel covariate data. The exptData component

carries experiment-level information, including MIAME (minimum information about a microarray experiment)-structured metadata. The R expressions exemplify how to access components. For instance, provided that these metadata were recorded, `rowData(se)$entrezId` returns the NCBI Entrez Gene identifiers of the features, and `set$issue` returns the tissue descriptions for the samples. Range-based operations, such as `%in%`, act on the `rowData` to return a logical vector that selects the features lying within the regions specified by the data object `CNVs`. Together with the bracket operator, such expressions can be used to subset a `SummarizedExperiment` to a focused set of genes and tissues for downstream analysis.” (Huber et al., 2015a)

“A genomics-specific visualization type is plots along genomic coordinates. There are several packages that create attractive displays of along-genome data tracks, including Gviz and ggbio ... **These packages operate directly on common Bioconductor data structures and thus integrate with available data manipulation and modeling functionality.** A basic operation underlying such visualizations is computing with genomic regions, and the biovizBase package provides a bridge between the Ranges infrastructure and plotting packages.” (Huber et al., 2015a)

“Genomic data set sizes sometimes exceed what can be managed with standard in-memory data models, and then tools from high performance computing come into play. An example is the use of hdf5—an interface to the HDF5 large data management system (<http://www.hdfgroup.org/HDF5>)—by the h5vc package to slice large, genome-size data cubes into chunks that are amenable for rapid interactive computation and visualization. Both ggbio and Gviz issue range-restricted queries to file formats including BAM, BGZIP/Tabix and BigWig via Rsamtools and rtracklayer to quickly integrate data from multiple files over a specific genomic region.” (Huber et al., 2015a)

“Developers are constantly updating their packages to extend capabilities, improve performance, fix bugs and enhance documentation. These changes are introduced into the development branch of Bioconductor and released to end users every 6 months; changes are tracked using a central, publicly readable Subversion software repository, so details of all changes are fully accessible. Simultaneously, R itself is continually changing, typically around performance enhancements and increased functionality. Owing to this dynamic environment, all packages undergo a daily testing procedure. Testing is fully automated and ensures that all code examples in the package documentation, as well as further unit tests, run without error. Successful completion of the testing will result in the package being built and presented to the community.” (Huber et al., 2015a)

“Interoperability between software components for different stages and types of analysis is essential to the success of Bioconductor. Interoperability is established through the definition of common data structures that package authors are expected to use ... Technically, Bioconductor’s common data structures are implemented as classes in the S4 object-oriented system of the R language. In this manner, useful software concepts including encapsulation, abstraction of interface from implementation, polymorphism, inheritance and reflection are directly available. It allows core tasks such as matching of sample data and metadata to be adopted across disciplines, and it provides a foundation on which community development is based. It is instructive to compare such a representation to popular alternatives in bioinformatics: file-based data format conventions and

primitive data structures of a language such as matrices or spreadsheet tables. With file-based formats, operations such as subsetting or data transformation can be tedious and error prone, and the serialized nature of files discourages operations that require a global view of the data. In either case, validity checking and reflection cannot rely on preformed or standardized support and need to be programmed from scratch again for every convention—or are missing altogether. As soon as the data for a project are distributed in multiple tables or files, the alignment of data records or the consistency of identifiers is precarious, and interoperability is hampered by having to manipulate disperse, loosely coordinated data collections.” (Huber et al., 2015a)

Some of the most important data structures in Bioconductor are (Huber et al., 2015a) (from Table 2 in this reference):

- ExpressionSet (Biobase package)
- SummarizedExperiment (GenomicRanges package)
- GRanges (GenomicRanges package)
- VCF (VariantAnnotation package)
- VRanges (VariantAnnotation package)
- BSgenome (BSgenome package)

“For Bioconductor, which provides tools in R for analyzing genomic data, interoperability was essential to its success. We defined a handful of data structures that we expected people to use. For instance, if everybody puts their gene expression data into the same kind of box, it doesn’t matter how the data came about, but that box is the same and can be used by analytic tools. Really, I think it’s data structures that drive interoperability.” — Robert Gentlemen in (Altschul et al., 2013)

“I have found that real hardcore software engineers tend to worry about problems that are just not existent in our space. They keep wanting to write clean, shiny software, when you know that the software that you’re using today is not the software you’re going to be using this time next year.” — Robert Gentlemen in (Altschul et al., 2013)

“Biology, formerly a science with sparse, often only qualitative data, has turned into a field whose production of quantitative data is on par with high energy physics or astronomy and whose data are wildly more heterogeneous and complex.” (Holmes and Huber, 2018)

“Any biological system or organism is composed of tens of thousands of components, which can be in different states and interact in multiple ways. Modern biology aims to understand such systems by acquiring comprehensive—and this means high dimensional—data in their temporal and spatial context, with multiple covariates and interactions.” (Holmes and Huber, 2018)

“Biological data come in all sorts of shapes: nucleic acid and protein sequences, rectangular tables of counts, multiple tables, continuous variables, batch factors, phenotypic images, spatial coordinates. Besides data measured in lab experiments, there are clinical data, longitudinal information, environmental measurements, networks, lineage trees, annotation from biological databases in free text or controlled vocabularies, …” (Holmes and Huber, 2018)

“Bioconductor packages support the reading of many of the data types and formats produced by measurement instruments used in modern biology, as well as the needed technology-specific ‘preprocessing’ routines. This community is actively keeping these up-to-date with the rapid developments in the instrument market.” (Holmes and Huber, 2018)

“The Bioconductor project has defined specialized data containers to represent complex biological datasets. These help to keep your data consistent, safe and easy to use.” (Holmes and Huber, 2018)

“Bioconductor in particular contains packages from diverse authors that cover a wide range of functionalities but still interoperate because of the common data containers.” (Holmes and Huber, 2018)

“IRanges is a general container for mathematical intervals. We create the biological context with the next line [which uses GRanges]. [Footnote: ‘The ‘I’ in IRanges stands for ‘interval’, the ‘G’ in GRanges for ‘genomic’.’]” (Holmes and Huber, 2018)

“Here we had to assemble a copy of the expression data (`exprs(x)`) and the sample annotation data (`pData(x)`) all together into the dataframe `dftx`—since this is the data format that `ggplot2` functions most easily take as input.” (Holmes and Huber, 2018)

GRanges is “a specialized class from the Bioconductor project for storing data that are associated with genomic coordinates. The first three columns are obligatory: `seqnames`, the name of the containing biopolymer (in our case, the names of human chromosomes); `ranges`, the genomic coordinates of the intervals (in this case, the intervals all have lengths 1, as they refer to a single nucleotide), and the DNA strand from which the RNA is transcribed. You can find out more on how to use this class and its associated infrastructure in the documentation, e.g., the vignette of the `GenomicRanges` package. Learning it is worth the effort if you want to work with genome-associated datasets, as it enables convenient, efficient and safe manipulation of these data and provides many powerful utilities.” (Holmes and Huber, 2018)

ChiP-Seq data “are sequences of pieces of DNA that are obtained from chromatin immunoprecipitation (ChIP). This technology enables the mapping of the locations along genomic data of transcription factors, nucleosomes, histone modifications, chromatin remodeling enzymes, chaperones, polymerases and other proteins. It was the main technology used by the Encyclopedia of DNA Elements (ENCODE) project. Here we use an example (Kuan et al., 2011) from the `mosaicsExample` package, which shows data measured on chromosome 22 from a ChIP-Seq of antibodies for the STAT1 protein and the H3K4me3 histone modification applied to the GM12878 cell line. Here we do not show the code used to construct the `binTFBS` object that contains the binding sites for one chromosome (22) [in a `BinData` class, it looks like from the `mosaics` package perhaps].” (Holmes and Huber, 2018)

“At different stages of their development, immune cells express unique combinations of proteins on their surfaces. These protein-markers are called CDs (clusters of differentiation) and are collected by flow cytometry (using fluorescence...) or mass cytometry (using single-cell atomic mass spectrometry of heavy

metal reporters). An example of a commonly used CD is CD4; this protein is expressed by helper T cells that are referred to as being ‘CD4+’. Note, however, that some cells express CD4 (thus are CD4+) but are not actually helper T cells. We start by loading some useful Bioconductor packages for flow cytometry, `flowCore` and `flowViz`.” (Holmes and Huber, 2018)

“Many datasets consist of several variables measured on the same set of subjects: patients, samples or organisms. For instance, we may have biometric characteristics such as height, weight, age as well as clinical variables such as blood pressure, blood sugar, heart rate and genetic data for, say, a thousand patients. The raison d’être for multivariate analysis is the investigation of connections or associations between the different variables measured. Usually the data are reported in a tabular data structure, with one row for each subject and one column for each variable. ... in the special case where each of the variables is numeric, ... we can represent the data structure as a matrix in R. If the columns of the matrix are independent of each other (unrelated), we can simply study each column separately and do standard ‘univariate’ statistics on them one by one; there would be no benefit in studying them as a matrix. More often, there will be patterns and dependencies. For instance, in the biology of cells, we know that the proliferation rate will influence the expression of many genes simultaneously. Studying the expression of 25,000 genes (columns) on many samples (rows) of patient-derived cells, we notice that many of the genes act together; either they are positively correlated or they are anti-correlated. We would miss a lot of important information if we were to only study each gene separately. Important connections between genes are detectable only if we consider the data as a whole, each row representing the many measurements made on the same observational unit. However, having 25,000 dimensions of variation to consider at once is daunting; [you can] reduce our data to a smaller number of the most important dimensions without losing too much information.” (Holmes and Huber, 2018)

“RNA-Seq transcriptome data report the number of sequence reads matching each gene [or sub-gene structure, such as exons] in each of several biological samples... It is customary in the RNA-Seq field ... to report genes in rows and samples in columns. Compared with the other matrices we look at here, this is transposed: rows and columns swapped. Such different conventions easily lead to errors, so they are worth paying attention to. [Footnote: ‘The Bioconductor project tries to help users and developers to avoid such ambiguities by defining data containers in which such conventions are explicitly fixed...’]” (Holmes and Huber, 2018)

“Proteomic profiles: Here the columns are aligned mass spectroscopy peaks or molecules identified through their m/z ratios; the entries in the matrix are measured intensities.” (Holmes and Huber, 2018)

“... unlike regression, PCA treats all variables equally (to the extent that they were preprocessed to have equivalent standard deviations). However, it is still possible to map other continuous variables or categorical factors onto plots in order to help interpret the results. Often we have supplementary information on the samples, for example diagnostic labels in the diabetes data or cell types in the T-cell gene expression data. Here we see how we can use such extra variables to inform our interpretation. The best place to store such so-called **metadata** is in appropriate slots of the data object (such as in the Bioconductor

`SummarizedExperiment` class); the second best is in additional columns of the data frame that also contains the numeric data. In practice, such information is often stored in a more or less cryptic manner in the row names of the matrix.” (Holmes and Huber, 2018)

“Multivariate data analyses require ‘conscious’ preprocessing. After consulting all the means, variances, and one-dimensional histograms, we saw how to rescale and center the data.” (Holmes and Huber, 2018)

“Many measurement devices in biotechnology are based on massively parallel sampling and counting of molecules. One example is high-throughput DNA sequencing. Its applications fall broadly into two main classes of data output. In the first case, the outputs of interest are the sequences themselves, perhaps also their polymorphisms or differences from other sequences seen before. In the second case, the sequences themselves are more or less well understood (if, say, we have a well-assembled and annotated genome) and our interest is the abundance of different sequence regions in our sample. For instance, in RNA-Seq..., we sequence the RNA molecules found in a population of cells or in a tissue. In ChIP-Seq, we sequence DNA regions that are bound to a particular RNA-binding protein. In DNA-Seq, we sequence genomic DNA and are interested in the prevalence of genetic variants in heterogeneous populations of cells, for instance the clonal composition of a tumor. In high-throughput chromatin conformation capture (HiC) we aim to map the 3D spatial arrangement of DNA. In genetic screens (using, say, RNAi or CRISPR-Cas9 libraries for perturbation and high-throughput sequencing for readout), we’re interested in the proliferation or survival of cells upon gene knockdown, knockout, or modification. In microbiome analysis, we study the abundance of different microbial species in complex microbial habitats. Ideally, we might want to sequence and count *all* molecules of interest in the sample. Generally this is not possible; the biochemical protocols are not 100% efficient, and some molecules or intermediates get lost along the way. Moreover, it’s often also not even necessary. Instead, we sequence and count a *statistical sample*. The sample size will depend on the complexity of the sequence pool assayed; it can go from tens of thousands to billions. This *sampling* nature of the data is important when it comes to analyzing them. We hope that the sampling is sufficiently representative for us to identify interesting trends and patterns.” (Holmes and Huber, 2018)

“DESeq2 uses a specialized data container, called `DESeqDataSet`, to store the datasets it works with. Such use of specialized containers—or, in R terminology, classes—is a common principle of the Bioconductor project, as it helps users keep related data together. While this way of doing things requires users to invest a little more time up front to understand the classes, compared with just using basic R data types like `matrix` and `dataframe`, it helps in avoiding bugs due to loss of synchronization between related parts of the data. It also enables the abstraction and encapsulation of common operations that could be quite wordy if always expressed in basic terms [footnote: Another advantage is that classes can contain validity methods, which make sure that the data always fulfill certain expectations, for instance, that the counts are positive integers, or that the columns of the counts matrix align with the rows of the sample annotation `dataframe`.] `DESeqDataSet` is an extension of the class `SummarizedExperiment` in Bioconductor. The `SummarizedExperiment` class is also used by many other packages, so learning to work with it will enable you to use a large range of

tools. We will use the constructor function `DESeqDataSetFromMatrix` to create a `DESeqDataSet` from the count data matrix ... and the sample annotation dataframe ... The `SummarizedExperiment` class—and therefore `DESeqDataSet`—also contains facilities for storing annotations of the rows of the count matrix.” (Holmes and Huber, 2018)

“We introduced the R `data.frame` class, which allows us to combine heterogeneous data types: categorical factors and continuous measurements. Each row of the dataframe corresponds to an object, or a record, and the columns are the different variables or features. Extra information about sample batches, dates of measurement and different protocols is often misnamed metadata. This information is actually real data that needs to be integrated into analyses. Here we show an example of an analysis that was done by Holmes et al. (2011) on bacterial abundance data from Phylochip microarrays. The experiment was designed to detect differences between a group of healthy rats and a group that had irritable bowel disease. This example shows how nuisance batch effects can become apparent in the analysis of experimental data. It illustrates why best practices in data analysis are sequential and why it is better to analyze data as they are collected—to adjust for severe problems in the experimental design as they occur—instead of trying to deal with deficiencies post mortem. When data collection started on this project, data for days 1 and 2 were delivered and we made the plot ... This shows a definite day effect. When investigating the source of this effect, we found that both the protocol and the array were different on days 1 and 2. This leads to uncertainty about the source of variation; we call this confounding of effects.” (Holmes and Huber, 2018)

“Many programs and workflows in biological sequence analysis or assays separate the environmental and contextual information they call metadata from the assays or sequence read numbers; we discourage this practice, as the exact connections between the samples and covariates are important. The lost connections between the assays and covariates makes later analyses impossible. Covariates such as clinical history, time, batch and location are important and should be considered components of the data.” (Holmes and Huber, 2018)

“The data provide an example of an awkward way of combining batch information from the actual data. The day information has been combined with the array data and encoded as a number and could be confused with a continuous variable. We will see in the next section a better practice for storing and manipulating heterogeneous data using a Bioconductor container called `SummarizedExperiment`” (Holmes and Huber, 2018)

“A more rational way of combining the batch and treatment information into compartments of a composite object is to use `SummarizedExperiment` classes. These include special slots for the assay(s) where rows represent features of interest (e.g., genes, transcripts, exons, etc.) and columns represent samples. Supplementary information about the features can be stored in a `DataFrame` object, accessible using the function `rowData`. Each row of the `DataFrame` provides information on the feature in the corresponding row of the `SummarizedExperiment` object. ... This is the best way to keep all the relevant data together. It will also enable you to quickly filter the data while keeping all the information aligned properly. ... Columns of the `DataFrame` represent different attributes of the features of interest, e.g., gene or transcript IDs. This is an ex-

ample of a hybrid data container from a single-cell experiment..." (Holmes and Huber, 2018)

"The success of the tidyverse attests to the power of its underlying ideas and the quality of its implementation. ... Nevertheless, dataframes in the long format are not a panacea. ... When we write a function that expects to work on an object like `xldf`, we have no guarantee that the column probe does indeed contain valid probe identifiers, or that such a column even exists. There is not even a proper way to express programmatically what 'an object like `xldf` means in the tidyverse. Object-oriented (OO) programming, and its incarnation S4 in R, solves such questions. For instance, the above-mentioned checks could be performed by a `isValidObject` method for a suitably defined class, and the class definition would formalize the notion of 'an object like `xldf`'. Addressing such issues is behind the object-oriented design of the data structures in Bioconductor, such as the `SummarizedExperiment` class. Other potentially useful features of OO data representations include: 1. Abstraction of interface from implementation and encapsulation: the user accesses the data only through defined channels and does not need to see how the data are stored 'inside'—which means that inside can be changed and optimized without breaking user-level code. 2. Polymorphism: you can have different functions with the same name, such as `plot` or `filter`, for different classes of objects, and R figures out for you which to call. 3. Inheritance: you can build up more complex data representations from simpler ones. 4. Reflection and self-documentation: you can send programmatic queries to an object to ask for more information about itself. All of these make it easier to write high-level code that focuses on the 'big picture' functionality rather than on implementation details of the building blocks—albeit at the cost of more initial investment in infrastructure and 'bureaucracy'." (Holmes and Huber, 2018)

Data provenance and metadata. THere is no obvious place in an object like `xldf` to add information about data provenance: e.g., who performed the experiment, where it was published, where the data were downloaded from, or which version of the data we're looking at (data bugs exist ...). Neither are there any explanations of the columns, such as units and assay type. Again, the data classes in Bioconductor try to address this." (Holmes and Huber, 2018)

Matrix-like data. Many datasets in biology have a natural matrix-like structure, since a number of features (e.g., genes: conventionally, the rows of the matrix) are assayed on several samples (conventionally, the columns of the matrix). Unrolling the matrix into a long form like `xldf` makes some operations (say, PCA, SVD, clustering of features or samples) more awkward." (Holmes and Huber, 2018)

Out-of memory data and chunking. Some datasets are too big to load into random access memory (RAM) and manipulate all at once. Chunking means splitting the data into manageable portions ('chunks') and then sequentially loading each portion, computing on it, storing the results and removing it from memory before loading the next portion. R also offers infrastructure for working with large datasets that are stored on disk in a relational database management systems (the `DBI` package) or in `HDF5` (the `rhdf5` package). The Bioconductor project provides the class `SummarizedExperiment`, which can store big data matrices either in RAM or in an `HDF5` backend in a manner that is transparent to the user of objects of this class." (Holmes and Huber, 2018)

Many of the statistical algorithms rely on matrices—these store data all of the same data type (e.g., numeric or counts). If you store extra variables, like binary outcome classifications (sick/well; alive/dead) or categorical variables, it will complicate these operations. Further, if these aren't to be used in things like dimension reduction and clustering, then you will continuously need to subset as you perform those matrix-based processes. Conversely, once you move to using ggplot to visualize your data and other tidy tools to create summary tables and other output for reports, it's handy to have all the information relevant to each of your observations handy within a dataframe—a structure that can hold and align data of many different types in its different columns. It therefore makes sense to evolve from more complex object types, in which different types of variables for each observation are stored in their own places, and where variables with similar types can be collected in a matrix that is ready for statistical processing, to the simpler dataframe at later stages in the pipeline, when working on publication-ready tables and figures. This requires a switch at some point in the pipeline from a coding approach that stores data in more complex Bioconductor S4 objects to one that stores data in a simple and straightforward tidy dataframe.

One file format called a *fasta* file is used to store DNA sequence data. The Biostrings package has a function for reading these data in from a fasta file. It stores the data in an instance of the DNAStringSet class from that package. Within this class are DNAString objects for each sample.

Bioconductor is used for many of the R packages for working with genomic and other bioinformatic data. One characteristic of packages on Bioconductor is that they make heavy use of a system for object-oriented programming in R. There are several systems for object-oriented programming in R. Bioconductor relies heavily on one called S4.

Object-oriented programming allows developers to create *methods*. These are functions in R that first check the class of the object that is input, and then run different code for different functions. For example, `summary` is one of these method-style functions. If you call the `summary` function with the input as a numeric vector, one set of code will be run: you will get numeric summaries of the values in that vector, including the minimum, maximum and median. However, if you run the same function, `summary`, on a dataframe with columns of factor data, the output will be a small summary for each column, giving the levels of the factor in each column and the number of column values in the most common of those levels.

With this system of writing methods, the same function call can be used for many different object types. By contrast, other approaches to programming might constrain a function to work with a single class of object—for example, a function might work on a dataframe and only a dataframe, not a vector, matrix, or other more complex types of objects.

These methods often have very names. Examples of these method-style functions include `plot`, `summary`, `head` [?], [others]. You can try running these method-style functions on just about any object class that you're using to store your data, and chances are good that it will work on the object and output something interesting.

The S4 system of object-oriented programming in R allows for something called *inheritance*. [More on this.]

As you use R with the Bioconductor packages, you often might not notice how much S4 objects are being used “under the hood”, as you pre-process and analyze data. By contrast, you may have learned the “tidyverse” approach in R, which is a powerful general approach for working with data. The tidyverse approach is centered on the object class `is` predominated uses, the `dataframe`, and so a lot of attention is given to thinking about that style of data storage in an object when learning the approach.

A pre-processing pipeline in Bioconductor might take the data through a number of different object classes over the course of the pipeline. Different functions within a Bioconductor package may manage this progression without you being very aware of it. For example, one function may read the data from the file format for the equipment and move the data directly into a certain complex object class, which a second function might input this object class, do some actions on the data, and then output the result in a different class.

Generally, if the functions in a pipeline handle these object transitions gracefully, you may not feel the need to dig more deeply into the object types. However, ideally you should feel comfortable taking a peek at your data at any step in the process. This can include seeing snippets of the data in its object (e.g., the first few elements in each component of the data at that stage) and also feel comfortable visualizing parts of the data in simple ways.

This is certainly possible even when data are stored in complex or unfamiliar object classes. However, it's a bit less natural than exploring your data when it's stored in an object class that you feel very comfortable with. For example, most R programmers have several go-to strategies for checking any data they have stored in a `dataframe`. You can develop these same go-to strategies for data in more complex object classes once you understand a few basics about the S4 system and the object classes created using this system.

First, there are a few methods you can use to figure out what's in a data object. [More on this. `str`, some on interactive ways to look at objects?] Further, most S4 objects will have their own helpfiles [doublecheck], and you can use this resource to learn more about what it's storing and where it puts each piece of data. [More on accessing these help files. `?ExpressionSet`, for example.]

Once you know what's in your object, there are a few ways that you can pull out different elements of the data. One basic way (it's a bit heavy-handed, and best to save for when you're struggling with other methods) is to extract *slots* from the object using the `@` symbol. If you have worked much with base

R, you will be familiar with pulling out elements of more basic object classes using \$. For example, if you wanted to extract a column named weight from a dataframe object called experiment_1, you could do so using the syntax experiment_1\$weight. The \$ operator does not work in the same way with S4 objects. With these, we say that the elements are stored in different slots of the object, and each slot can be extracted using @. So if you had an S4 object with data on animal weights stored in a slot called weight, you could extract it from an S4 object instanced named experiment_2 with experiment_2@weight.

A more elegant approach is to access elements stored in the object using a special type of function called an **accessor** function.

One important object class in Bioconductor is ExpressionSet. This object class helps to keep different elements of data from an experiment aligned—for example, it helps ensure that higher-level data about each sample is kept well-aligned with data on more specific values—like measurements from each metabolite feature [? better example? gene expression values for each gene?] specific to each sample. The three slots in this object class are pData, exprs, and fData. The data in these three slots can be accessed using the accessor functions of pData, exprs, and fData.

Often, the contents of the slots within a Bioconductor class will be a more generic object type that you're familiar with, like a matrix or vector.

S4 objects can be set to check that the inputs are valid for that object class when someone creates a new object of that class. This helps with quality control in creating new objects, where these issues can be caught early, before functions are run on the object that assume certain characteristics of its data.

Methods are also referred to as generic functions within the S4 system?

“the whole point of OOP is not to have to worry about what is inside an object. Objects made on different machines and with different languages should be able to talk to each other” — Alan Kay

This idea in object-oriented programming may be very helpful for large, multi-developer programming, since different people, or even whole teams could develop their parts independently. As long as the teams have all agreed on the way that messages will be passed between different objects and parts of the code, they could have independence in how they conduct work on their own objects. There are rules for how things connect, and independence in how each part works. If the rules for connecting different objects are set, then this approach allows for immense flexibility in how the code to work with the objects on their own can be written and changed, without breaking the whole system of code.

However, the idea of not worrying about what's inside an object is at odds with some basic principles for working with experimental data. Exploratory data analysis is a key principle for improving quality control, rigor, and even

creativity in working with scientific data sets. [More on EDA, including from Tukey] EDA requires a researcher to be able to explore the data stored inside an object, ideally at any stage along a pipeline of pre-processing and then analyzing those data. Therefore, there's a bit of tension in the S4 approach in R, between using a system that allows for powerful development of tools to explore data and the fundamental needs of the researcher to access and explore their data as they work with it—to “see inside” the objects storing their data at every step.

Objects store data. They are data structures, with certain rules for where they store different elements of the data. They also are associated with specific functions that work with the way they store the data.

“A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of concepts for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is ‘close to the machine’, so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is ‘close to the problem to be solved’ so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.” — Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986

“The basis for OOP started in the early 1960s. A breakthrough involving instances and objects was achieved at MIT with the PDP-1, and the first programming language to use objects was Simula 67. It was designed for the purpose of creating simulations, and was developed by Kristen Nygaard and Ole-Johan Dahl in Norway. They were working on simulations that deal with exploding ships, and realized they could group the ships into different categories. Each ship type would have its own class, and the class would generate its unique behavior and data. Simula was not only responsible for introducing the concept of a class, but it also introduced the instance of a class. The term ‘object oriented programming’ was first used by Xerox PARC in their Smalltalk programming language. The term was used to refer to the process of using objects as the foundation for computation. The Smalltalk team was inspired by the Simula 67 project, but they designed Smalltalk so that it would be dynamic. The objects could be changed, created, or deleted, and this was different from the static systems that were commonly used. Smalltalk was also the first programming language to introduce the inheritance concept. It is this feature that allowed Smalltalk to surpass both Simula 67 and the analog programming systems. While these systems were advanced for their time, they did not use the inheritance concept.” — <http://www.exforsys.com/tutorials/oops-concepts/the-history-of-object-oriented-programming.html>

“Object-oriented programming is first and foremost about objects. Initially object-oriented languages were geared toward modeling real world objects so the objects in a program corresponded to real world objects. Examples might include: 1. Simulations of a factory floor—objects represent machines and raw materials 2. Simulations of a planetary system—objects represent celestial

bodies such as planets, stars, asteroids, and gas clouds 3. A PC desktop—objects represent windows, documents, programs, and folders 4. An operating system—objects represent system resources such as the CPU, memory, disks, tapes, mice, and other I/O devices” <https://www.ephemeralobjects.org/2014/02/03/a-brief-history-of-object-oriented-programming/>

“The idea with an object is that it advertises the types of data that it will store and the types of operations that it allows to manipulate that data. However, it hides its implementation from the user. For a real world analogy, think of a radio. The purpose of a radio is to play the program content of radio stations (actually translate broadcast signals into sounds that humans can understand). A radio has various dials that allow you to control functions such as the station you are tuned to, the volume, the tone, the bass, the power, and so on. These dials represent the operations that you can use to manipulate the radio. The implementation of the radio is hidden from you. It could be implemented using vacuum tubes or solid state transistors, or some other technology. The point is you do not need to know. The fact that the implementation is hidden from you allows radio manufacturers to upgrade the technology within radios without requiring you to relearn how to use a radio.” <https://www.ephemeralobjects.org/2014/02/03/a-brief-history-of-object-oriented-programming/>

“The set of operations provided by an object is called its **interface**. The interface defines both the names of the operations and the behavior of these operations. In essence the interface is a contract between the object and the program that uses it. The object guarantees that it will provide the advertised set of operations and that they will behave in a specified fashion. Any object that adheres to this contract can be used interchangeably by the program. Hence the implementation of an object can be changed without affecting the behavior of a program.” <https://www.ephemeralobjects.org/2014/02/03/a-brief-history-of-object-oriented-programming/>

“An object is not much good if each one must be custom crafted. For example, radios would not be nearly as prevalent if each one was handcrafted. What is needed is a way to provide a blueprint for an object and a way for a ‘factory’ to use this blueprint to mass produce objects. Classes provide this mechanism in object-oriented programming. A **class** is a factory that is able to mass produce objects. The programmer provides a class with a blueprint of the desired type of object. A ‘blueprint’ is actually composed of: 1. A declaration of a set of variables that the object will possess, 2. A declaration of the set of operations that the object will provide, and 3. A set of function definitions that implements each of these operations. The set of variables possessed by each object are called **instance variables**. The set of operations that the object provides are called **methods**. For most practical purposes, a method is like a function. When a program wants a new instance of an object, it asks the appropriate class to create a new object for it. The class allocates memory to hold the object’s instance variables and returns the object to the program. Each object knows which class created it so that when an operation is requested for that object, it can look up in the class the function that implements that operation and call that function.” <https://www.ephemeralobjects.org/2014/02/03/a-brief-history-of-object-oriented-programming/>

“In object-oriented programming, **inheritance** means the inheritance of another object’s interface, and possibly its implementation as well. Inheritance is accom-

plished by stating that a new class is a **subclass** of an existing class. The class that is inherited from is called the **superclass**. The subclass always inherits the superclass's complete interface. It can extend the interface but it cannot delete any operations from the interface. The subclass also inherits the superclass's implementation, or in other words, the functions that implement the superclass's operations. However, the subclass is free to define new functions for these operations. This is called **overriding** the superclass's implementation. The subclass can selectively pick and choose which functions it overrides. Any functions that are not overridden are inherited." <https://www.ephemeralobjects.org/2014/02/03/a-brief-history-of-object-oriented-programming/>

For xcms, basic object class is now SCMSnExp. (Holmes and Huber, 2018) This is the container the data is stored in while pre-processing LCMS data with the xcms package.

"xcms supports analysis of LC/MS data from files in (AIA/ANDI) NetCDF, mzML/mzXML and mzData format. For the actual data import Bioconductor's mzR is used." (Smith, 2013)

"Subsequently we load the raw data as an OnDiskMSnExp object using the readMSData method from the MSnbase package. The MSnbase provides base structures and infrastructure for the processing of mass spectrometry data. ... The resulting OnDiskMSnExp object contains general information about the number of spectra, retention times, the measured total ion current etc, but does not contain the full raw data (i.e. the m/z and intensity values from each measured spectrum). Its memory footprint is thus rather small making it an ideal object to represent large metabolomics experiments while allowing to perform simple quality controls, data inspection and exploration as well as data sub-setting operations. The m/z and intensity values are imported from the raw data files on demand, hence the location of the raw data files should not be changed after initial data import." (Smith, 2013)

Important skills for working with data in Bioconductor classes:

- How to get to the help file specific for that type of class for generic functions. For example, how to see specific parameters that can be included in a plot call.
- How to find/list all the functions / methods available for that class of object.
- How to access a help file that explains the structure of that object class.

What slots are included? What's in each slot? How can you create a new instance of the class? `?`Chromatogram-class`` accesses the helpfile that describes the Chromatogram class in the MSnbase package. It includes info on what is typically stored in objects of this class (retention time-intensity value pairs for chromatographic mass spectroscopy data). It tells how to create a new object of that class using its constructor function. It lists accessor functions for objects in that class: `rtime` to get retention times, `intensity` to get the intensities, `mz` to get the range of the chromatogram, etc. It also lists some functions, including generic functions like `length`, that can be used with objects in that class, as well as some details on how

the class's method for that generic function works (in terms of what it will return). It provides the usage, and defines the parameters, for functions that work with this object class.

- How to figure out and use accessor functions to access specific pieces of data from objects of that class.

Often, you'll have a class that stores data for one sample (e.g., `Chromatogram` from the `MSnbase` package, which stores chromatographic mass spectrometry data), and then another class that will collectively store these sample-specific data in a larger object (e.g., `Chromatograms` class, also from the `MSnbase` package, which stores multiple `Chromatogram` objects, from different samples, in a structure derived from the matrix structure).

You can use the pipe operator from `magrittr` in Bioconductor workflows, too. It works by “piping” the output from one function call as the input into the next function call (typically, the parameter in the first position among parameters to that function call).

Calling the object name at the R console will run the `print` method for that object's class on the object. Often, this will provide a print out of useful metadata, descriptions, and summaries for the data stored in that object. If you want a more granular look at what's contained in the object, you can use the `str` function.

Object classes are often set up to inherit from another class. This means that a method that works for one class might also work for a similar class, if the second inherits from the first. “The results are returned as an `XCMSnExp` object which extends the `OnDiskMSnExp` object by storing also LC/GC-MS preprocessing results. This means also that all methods to sub-set and filter the data or to access the (raw) data are inherited from the `OnDiskMSnExp` object and can thus be re-used. Note also that it is possible to perform additional rounds of peak detection (e.g. on MS level > 1 data) on the `xdata` object by calling `findChromPeaks` with the parameter `add = TRUE`.” (Smith, 2013)

Sometimes there will be a class just for storing the parameters for running an algorithm, for example, the “`CentWaveParam`” and “`MergeNeighboringPeaksParam`” classes in the `xcms` package. Presumably this is to allow validity checking before using them in the algorithm?

Moving into a more general object class after pre-processing:

“Results from the `xcms`-based preprocessing can be summarized into a `SummarizedExperiment` object from the `SummarizedExperiment` package with the `quantify` method. This object will contain the feature abundances as the assay matrix, the feature definition (their m/z, retention time and other metadata) as `rowData` (i.e. row annotations) and the sample/phenotype information as `colData` (i.e. column annotations). All the processing history will be put into the object's metadata. This object can then be used for any further (`xcms`-independent) processing and analysis.” (Smith, 2013)

“The concept in R of attributes of an object allows an exceptionally rich set of data objects. S3 methods make the class attribute the driver of an object-oriented

system. It is an optional system. Only if an object has a class attribute do S3 methods really come into effect." (Burns, 2011)

"There are some functions that are generic. Examples include print, plot, summary. These functions look at the class attribute of their first argument. If that argument does have a class attribute, then the generic function looks for a method of the generic function that matches the class of the argument. If such a match exists, then the method function is used. If there is no matching method or if the argument does not have a class, then the default method is used. Let's get specific. The lm (linear model) function returns an object of class 'lm'. Among the methods for print are print.lm and print.default. The result of a call to lm is printed with print.lm. The result of 1:10 is printed with print.default." (Burns, 2011)

"S3 methods are simple and powerful. Objects are printed and plotted and summarized appropriately, with no effort from the user. The user only needs to know print, plot and summary." (Burns, 2011)

"If your mystery number is in obj, then there are a few ways to look for it:
`print.default(obj)` `print(unclass(obj))` `str(obj)`

The first two print the object as if it had no class, the last prints an outline of the structure of the object. You can also do: `names(obj)` to see what components the object has—this can give you an overview of the object." (Burns, 2011)

"median is a generic function as evidenced by the appearance of `UseMethod`. What the new user meant to ask was, 'How can I find the default method for median?' The most sure-fire way of getting the method is to use `getS3method`:
`getS3method('median', 'default')`." (Burns, 2011)

"The `methods` function lists the methods of a generic function [for classes loaded in the current session]. Alternatively given a class it returns the generic functions that have methods for the class." (Burns, 2011)

```
## [1] "print.acf"          "print.AES"           "print.anova"
## [4] "print.ansi_string" "print.ansi_style"  "print.aov"
## [1] ExpressionSet,environment-method ExpressionSet,matrix-method
## [3] ExpressionSet,missing-method
## see '?methods' for accessing help and source code
```

"Inheritance should be based on similarity of the structure of the objects, not similarity of the concepts for the objects. Matrices and data frames have similar concepts. Matrices are a specialization of data frames (all columns of the same type), so conceptually inheritance makes sense. However, matrices and data frames have completely different implementations, so inheritance makes no practical sense. The power of inheritance is the ability to (essentially) reuse code." (Burns, 2011)

"S3 methods are simple and powerful, and a bit ad hoc. S4 methods remove the ad hoc—they are more strict and more general. The S4 methods technology is a stiffer rope—when you hang yourself with it, it surely will not break. But that is basically the point of it—the programmer is restricted in order to make the results more dependable for the user. That's the plan anyway, and it often works." (Burns, 2011)

"S4 is quite strict about what an object of a specific class looks like. In contrast S3 methods allow you to merely add a class attribute to any object—as long as a method doesn't run into anything untoward, there is no penalty. A key advantage in strictly regulating the structure of objects in a particular class is that those objects can be used in C code (via the .Call function) without a copious amount of checking." (Burns, 2011)

"Along with the strictures on S4 objects comes some new vocabulary. The pieces (components) of the object are called slots. Slots are accessed by the @ operator." (Burns, 2011)

"By now you will have noticed that S4 methods are driven by the class attribute just as S3 methods are. This commonality perhaps makes the two systems appear more similar than they are. In S3 the decision of what method to use is made in real-time when the function is called. In S4 the decision is made when the code is loaded into the R session—there is a table that charts the relation. (Burns, 2011)

3.5.10 Subsection 2

3.5.11 Applied exercise

- [Example data in a basic list]
- [Example data in a Bioconductor list-based class]
- [Explore each example dataset. What slots do each have? What are the names of each slot? What data structures / data types are in each slot?]
- [Extract certain elements from each dataset by hand. Assign to its own object name so you can use it by itself.]
- [Use `biobroom` to extract pieces of data in the Bioconductor dataset as tidy dataframes. Try using this with further tidyverse code to create a nice table/visualization.]

3.6 Example: Converting from complex to 'tidy' data formats

We will provide a detailed example of a case where data pre-processing in R results in a complex, 'untidy' data format. We will walk through an example of applying automated gating to flow cytometry data. We will demonstrate the complex initial format of this pre-processed data and then show trainees how a 'tidy' dataset can be extracted and used for further data analysis and visualization using the popular R 'tidyverse' tools. This example will use real experimental data from one of our Co-Is research on the immunology of tuberculosis.

Objectives. After this module, the trainee will be able to:

- Describe how tools like `biobroom` were used in this real research example to convert from the complex data format from pre-processing to a format better for further data analysis and visualization
- Understand how these tools would fit in their own research pipelines

3.6.1 Pipelines that combine complex and tidy data structures

The tidyverse approach in R is based on keeping data in a dataframe structure. By keeping this common structure, the tidyverse allows for straightforward but powerful work with your data by chaining together simple, single-purpose functions. This approach is widely covered in introductory R programming courses and books. A great starting point is the book *R Programming for Data Science*, which is available both in print and freely online at [site]. Many excellent resources exist for learning this approach, and so we won't recover that information here. Instead, we will focus on how to interface between this approach and the object-based approach that's more common with Bioconductor packages. Bioconductor packages often take an object-based approach, and with good reason because of the complexity and size of many early versions of biomedical data in the preprocessing process. There are also resources for learning to use specific Bioconductor packages, as well as some general resources on Bioconductor, like *R Programming for Bioinformatics* [ref]. However, there are fewer resources available online that teach how to coordinate between these two approaches in a pipeline of code, so that you can leverage the needed power of Bioconductor approaches early in your pipeline, as you preprocess large and complex data, and then shift to use a tidyverse approach once your data is amenable to this more straightforward approach to analysis and visualization.

The heart of making this shift is learning how to convert data, when possible, from a more complex, class-type data structure (built on the flexible list data structure) to the simpler, more standardized two-dimensional dataframe structure that is required for the tidyverse approach. In this subsection, we'll cover approaches for converting your data from Bioconductor data structures to dataframes.

If you are lucky, this might be very straightforward. A pair of packages called `broom` and `biobroom` have been created specifically to facilitate the conversion of data from more complex structures to dataframes. The `broom` package was created first, by David Robinson, to convert the data stored in the objects that are created by fitting statistical models into tidy dataframes. Many of the functions in R that run statistical tests or fit statistical models output results in a more complex, list-based data structure. These structures have nice "print" methods, so if fitting the model or running the test is the very last step of your pipeline, you can just read the printed output from R. However, often you want to include these results in further code—for example, creating plots or tables that show results from several statistical tests or models. The `broom` package includes several functions for pulling out different bits of data that are stored in the complex data structure created by fitting the model or running the test and convert those pieces of data into a tidy dataframe. This tidy dataframe can then be easily used in further code using a tidyverse approach.

The `biobroom` package was created to meet a similar need with data stored

in some of the complex structures commonly used in Bioconductor packages.

[More about `biobroom`.]

[How to convert data if there isn't a `biobroom` method.] If you are unlucky, there may not be a `broom` or `biobroom` method that you can use for the particular class-based data structure that your data's in, or it might be in a more general list, rather than a specific class with a `biobroom` method. In this case, you'll need to extract the data "by hand" to move it into a `dataframe` once your data is simple enough to work with using a tidyverse approach. If you've mastered how to explore data stored in a list (covered in the last subsection), you'll have a headstart on how to do this. Once you know where to find each element of the data in the structure of the list, you can assign these specific pieces to their own R objects using typical R assignment (e.g., with the `gets` arrow, `<-`, or with `=`, depending on your preferred R programming style). ...

Use complex data structures early and convert to tidyverse approach when possible

"Converting" as restructuring the data

Converting along the way for exploration

May "drop" some data when converting

3.6.2 Techniques for connecting from complex to tidy data in a pipeline

Accessor functions

`broom` and `biobroom`

"tidy" data structures for single cell gene expression data

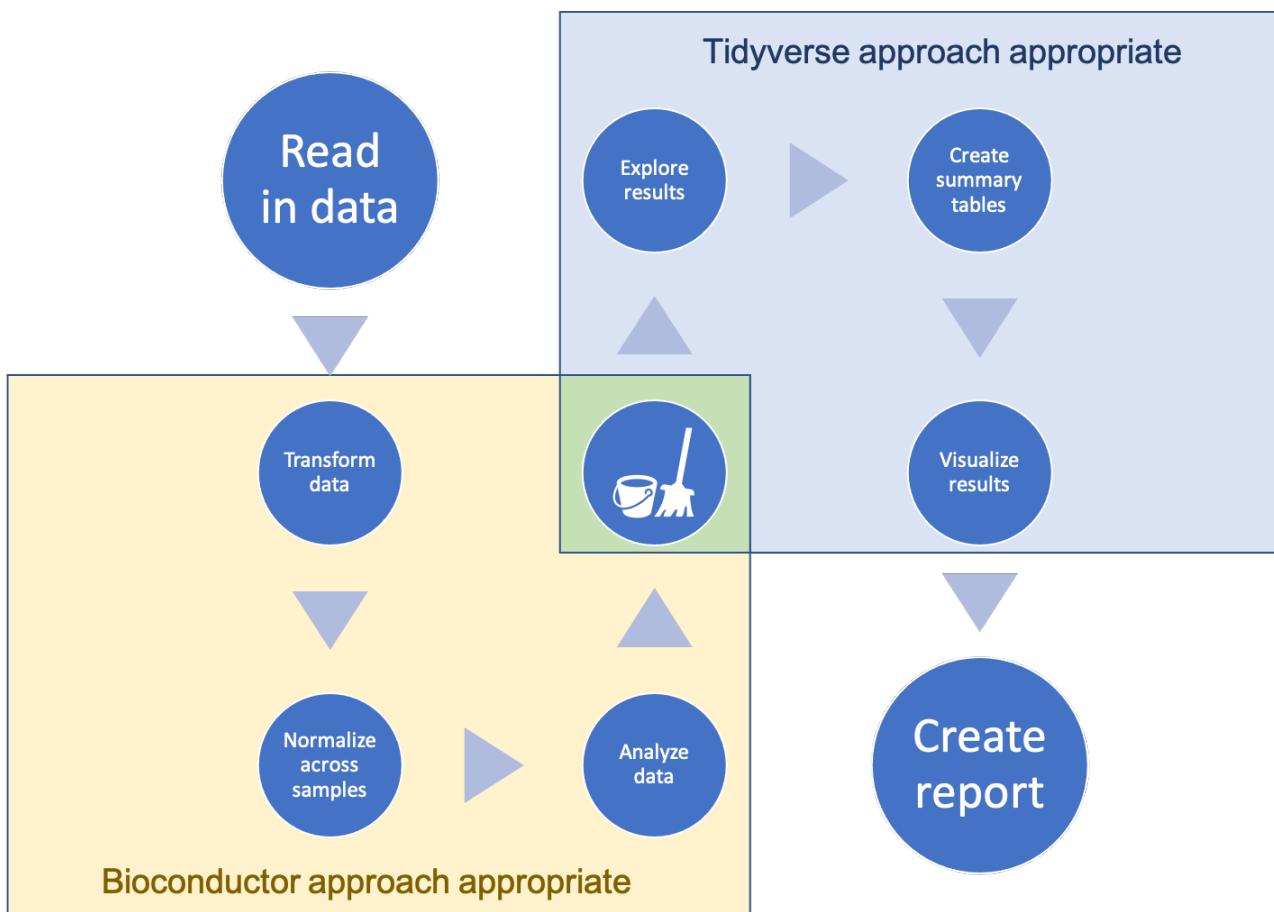
3.6.3 Example—Complex to tidy pipeline for scRNA-Seq data analysis

3.6.4 Combining Bioconductor and tidyverse approaches in a workflow

In the previous modules, we have talked about two topics. First we have talked about the convenience and power of tidyverse tools. These tools can be used at points in your workflow when the data can be stored in a simple standard format: the tidy `dataframe` format. We have also talked about reasons why there are advantages to using more complex data storage formats earlier in the process.

Work with research data will typically require a series of steps for pre-processing, analysis, exploration, and visualization. Collectively, these form a **workflow** or **pipeline** for the data analysis. With large, complex biological data, early steps in this workflow might need to use a Bioconductor approach, given the size and complexity of the data. However, this doesn't mean that you must completely give up the power and efficiency of the tidyverse approach described in earlier modules. Instead, you can combine the two, in a workflow like that shown in Figure 3.3. In this combined approach, you start the work-

flow in the Bioconductor approach and transition when possible to a tidyverse approach, transitioning by “tidying” from a more complex data structure to a simpler dataframe data structure along the way. In this module, we will describe how you can make this transition to create this type of combined workflow. This is a useful approach, because once your workflow has advanced to a stage where it is straightforward to store the data in a a dataframe, there are a large advantages to shifting into the tidyverse approach as compared to using more complex object-oriented classes for storing the data, in particular when it comes to data analysis and visualization at later stages in your workflow.



There are two key tools that have been developed as our packages that facilitate the shift of data from being stored in a more customized object-oriented class, for example one of the S4 type classes that we discussed when talking about complex data formats for Bioconductor. These packages move data from one of those storage containers into a tidy dataframe format. By doing this it moves the data into a format that is very easy to use in conjunction with the tidyverse tools and the tidyverse approach.

Figure 3.3: An overview of a workflow that moves from a Bioconductor approach—for pre-processing of the data—through to a tidyverse approach one pre-processing has created smaller, simpler data that can be reasonably stored in a dataframe structure.

In this module we will focus specifically on the `biobroom` package. Of the two packages this focuses specifically on moving data out of many of the common Bioconductor classes and into tidy dataframes. This package draws and an object-oriented approach in that it provides generic functions for extracting data from many different object classes that are coming in by a conductor. You will call the same function regardless of the class that the dad is in. If that object class has a `bio broom` method for that generic function, then the function will be able to extract parts of the data into a tidy data frame.

In this module we will also discuss another tool from the tidyverse, or rather a tool that draws on the tidy verse approach, that can be easily used in conjunction with biomedical data that has been processed using Bioconductor tools. This is a package called `ggbio` that facilitates the visualization of biomedical data. It includes functions and Specialized gian's or geometrical objects that are customized for some of the tasks that you might want to conduct in visualizing biomedical data in R. By drawing on tools and an approach from `ggplot` which is part of the tidyverse approach, these tools allow you to work with this data while still leveraging the powerful visualization tools and philosophy underlying the `ggplot` package.

Finally it is quite likely better purchase will continue to evolve through are, and that in the future there might be tidy data frame format that are adaptable enough to handle earlier stages in the data preprocessing. Tidy first dataframe have already been adapted to enable them to include more complex types of data within certain columns of the data frame any special list type column. This functionality is being leveraged through the `ffs` package to an evil a tidy approach to working with geographical data. This allows those who are working with geographical data, for example data from shapefiles for creating Maps, to use the standard tidyverse approaches while still containing complex data needed for this geographical information. It seems very possible that similar approaches may be adapted in the near future to allow for biomedical or genomic data to be stored in a way that both accounts for complexity early and pre-processing of these data but also allows for a more natural integration with the wealth of powerful tools available through the tidyverse approach.

3.6.5 *The biobroom package*

The `biobroom` package includes three main generic functions (methods), which can be used on a number of Bioconductor object classes. When applied to object stored in one of these Bioconductor classes, these functions will extract part of the data into a tidy dataframe format. In this format, it is easy to use the tools from the tidyverse to further explore, analyze, and visualize the data.

The three generic functions of `biobroom` are the functions `tidy`, `augment`, and `glance`. These function names mimic the names of the three main functions in the `broom` package, which is a more general purpose package for extracting tidy datasets from more complex R object containers. The `broom`

package focuses on the output from functions in R for statistical testing and modeling, while the newer `biobroom` package replicates this idea, but for many of the common object classes used to store data through Bioconductor packages and workflows.

The `biobroom` package includes methods for the following object classes (Bass et al., 2020):

- `qvalue` objects, which are used ...
- `DESeqDataSet` objects, which are used ...
- `DGEEexact` objects, which are used ...
- [`limma` objects]
- [`ExpressionSet` objects]
- `MSnSet` objects, which are used ...

As an example, we can look at how the `biobroom` package can be used to convert output generated by functions in the `edgeR` package into a tidy data frame, and how that output can then be explored and visualized using functions from the `tidyverse`.

The `edgeR` package is a popular Bioconductor package that can be used on gene expression data to explore which genes are expressed differently across experimental groups (*differential expression analysis*) (Robinson et al., 2010). Before using the functions in the package, the data must be preprocessed to align sequence reads from the raw data and then to create a table with the counts of each read at each gene across each sample. The `edgeR` package includes functions for pre-processing through its own functions, as well, including capabilities for filtering out genes with low read counts across all samples and model-based normalization across samples to help handle technical bias, including differences in sequencing depth (Chen et al., 2014).

The `edgeR` package operates on data stored in a special object class defined by the package, the `DGEList` object class (Chen et al., 2014). This object class includes areas for storing the table of read counts, in the form of a matrix appropriate for analysis by other functions in the package, as well as other spots for storing information about each sample and, if needed, a space to store annotations of the genes (Chen et al., 2014).

[Example from the `biobroom` help documentation—uses the `hammer` data that comes with the package. These data are stored in an `ExpressionSet` object, an object class defined by the `Biobase` package. You can see how the `tidy` function extracts these data in a tidy format. Then, the data are put in a `DGEList` class so they are in the right container for operations from `edgeR`. Then functions from the `edgeR` package are run to perform differential expression analysis on the data. The result is an object in the `DGEEexact` class, which is defined by the `edgeR` package. To extract data from this class in a tidy format, you can use the `tidy` and `glance` functions from `biobroom`.]

```
library(biobroom)
library(Biobase)
```

```
library(edgeR)

data(hammer)

class(hammer)

## [1] "ExpressionSet"
## attr(,"package")
## [1] "Biobase"

tidy(hammer)

## # A tibble: 236,128 x 3
##   gene           sample  value
##   <chr>          <chr>   <int>
## 1 ENSRNOG000000000001 SRX020102     2
## 2 ENSRNOG000000000007 SRX020102     4
## 3 ENSRNOG000000000008 SRX020102     0
## 4 ENSRNOG000000000009 SRX020102     0
## 5 ENSRNOG000000000010 SRX020102    19
## 6 ENSRNOG000000000012 SRX020102     7
## 7 ENSRNOG000000000014 SRX020102     0
## 8 ENSRNOG000000000017 SRX020102     4
## 9 ENSRNOG000000000021 SRX020102     7
## 10 ENSRNOG000000000024 SRX020102    86
## # ... with 236,118 more rows

## Example from `biobroom` help documentation
hammer.counts <- exprs(hammer) [, 1:4]
hammer.treatment <- phenoData(hammer)$protocol[1:4]

y <- DGEList(counts=hammer.counts, group=hammer.treatment)
y <- calcNormFactors(y)
y <- estimateCommonDisp(y)
y <- estimateTagwiseDisp(y)
et <- exactTest(y)

class(et)

## [1] "DGEEExact"
## attr(,"package")
## [1] "edgeR"

tidy(et)

## # A tibble: 29,516 x 4
```

```

##   gene          estimate  logCPM    p.value
##   <chr>        <dbl>    <dbl>      <dbl>
## 1 ENSRNOG000000000001  2.65     1.49  0.00000131
## 2 ENSRNOG000000000007 -0.409   -0.226  1
## 3 ENSRNOG000000000008  2.22    -0.407  0.129
## 4 ENSRNOG000000000009  0       -1.31   1
## 5 ENSRNOG000000000010  0.0331   1.79   1
## 6 ENSRNOG000000000012 -3.39    0.0794  0.00375
## 7 ENSRNOG000000000014  3.65    -0.854  0.252
## 8 ENSRNOG000000000017  2.42     1.11  0.0000638
## 9 ENSRNOG000000000021 -2.02    0.211  0.0373
## 10 ENSRNOG000000000024 0.133    3.97  0.508
## # ... with 29,506 more rows

glance(et)

##   significant    comparison
## 1           6341 control/L5 SNL

```

The creator of the `broom` package listed some of the common ways that statistical model output objects—the focus on ‘tidying’ in `broom`—tend to be untidy. These include that important information is stored in the row names, where it is harder to access, that the names of some columns can be tricky to work with because they use non-standard conventions (i.e., they don’t follow the rules for naming objects in R), that some desired information is not available in the object, but rather is typically computed with later methods for the object, like when `summary` is run on the object, or are only available as the result of a `print` method run on the object, and vectors that a user may want to explore in tandem are stored in different places in the object. (Robinson, 2014)

[Examples of these in Bioconductor objects?]

These ‘messy’ characteristics show up in the data stored in Bioconductor objects, as well, in terms of characteristics that impede working with data stored in these formats easily using tools from the tidyverse. As an example, one common class for storing data in Bioconductor work is the `ExpressionSet` object class, defined in the `Biobase` package (Huber et al., 2015b). This object class can be used to store the data from high-throughput assays. It includes slots for the assay data, as well as slots for storing metadata about the experiment, which could include information like sampling time points or sample strains, as well as the experimental group of each sample (control versus treated, for example).

Data from the assay for the experiment—for example, gene expression or intensity [?] measurements for each gene and each sample [?—can be extracted from an `ExpressionSet` object using an extractor function called `exprs`. Here is an example using the `hammer` example dataset available with

the `biobroom` package. The code call here extracts the assay data from the `hammer` R object, which is an instance of the `ExpressionSet` object class. It uses indexing (`[1:10, 1:3]`) to limit printing to the first ten rows and first three columns of the output, so we can investigate a small snapshot of the data:

```
##                               SRX020102 SRX020103 SRX020104
## ENSRNOG000000000001          2        4       18
## ENSRNOG000000000007          4        1        3
## ENSRNOG000000000008          0        1        4
## ENSRNOG000000000009          0        0        0
## ENSRNOG000000000010         19       10       19
## ENSRNOG000000000012          7        5        1
## ENSRNOG000000000014          0        0        2
## ENSRNOG000000000017          4        1       12
## ENSRNOG000000000021          7        5        2
## ENSRNOG000000000024         86       53       86
```

These data are stored in a matrix format. The gene identifiers [?] are given in the rownames and the samples in the column names. Each cell of the matrix provides the expression level (number of reads [?]) of a specific gene in a specific sample.

These data are structured and stored in such a way that they have some of the characteristics that can make data difficult to work with the data using tidyverse tools. For example, they store gene identifiers in the rownames, rather than in a separate column where they can be easily accessed when using tidyverse functions. Also, there are phenotype / meta data that are stored in other parts of the `ExpressionSet` data but that may be interesting to explore in conjunction with these assay data, including the experimental group of each sample (control versus animals in which chronic neuropathic pain was induced, in these example data).

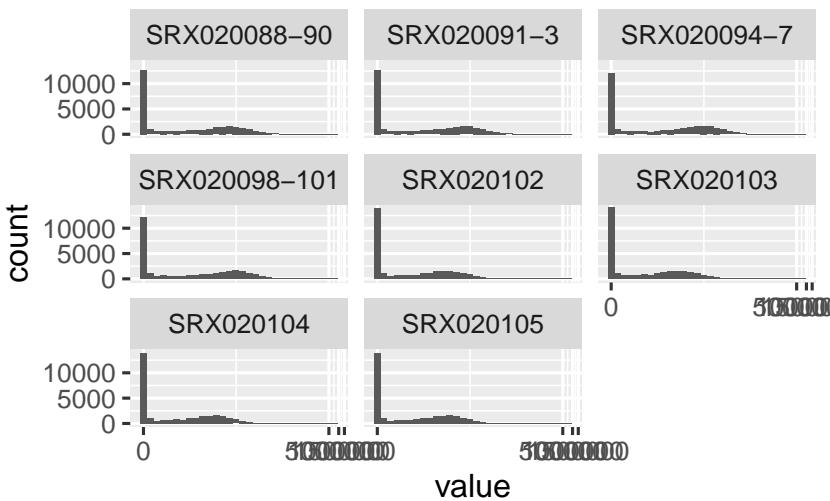
The `tidy` function from the `biobroom` package extracts these data and restructures them into a 'tidy' format, ready to use easily with tidyverse tools.

```
## # A tibble: 236,128 x 3
##   gene           sample  value
##   <chr>          <chr>   <int>
## 1 ENSRNOG000000000001 SRX020102     2
## 2 ENSRNOG000000000007 SRX020102     4
## 3 ENSRNOG000000000008 SRX020102     0
## 4 ENSRNOG000000000009 SRX020102     0
## 5 ENSRNOG000000000010 SRX020102    19
## 6 ENSRNOG000000000012 SRX020102     7
## 7 ENSRNOG000000000014 SRX020102     0
## 8 ENSRNOG000000000017 SRX020102     4
## 9 ENSRNOG000000000021 SRX020102     7
```

```
## 10 ENSRNOG000000000024 SRX020102     86
## # ... with 236,118 more rows
```

This output is a tidy dataframe object, with three columns providing the gene name, the sample identifier, and the expression level. In this format, the data can easily be explored and visualized with tidyverse tools. For example, you could easily create a set of histograms, one per sample, showing the distribution of expression levels across all genes in each sample: [better example visualization here?]

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

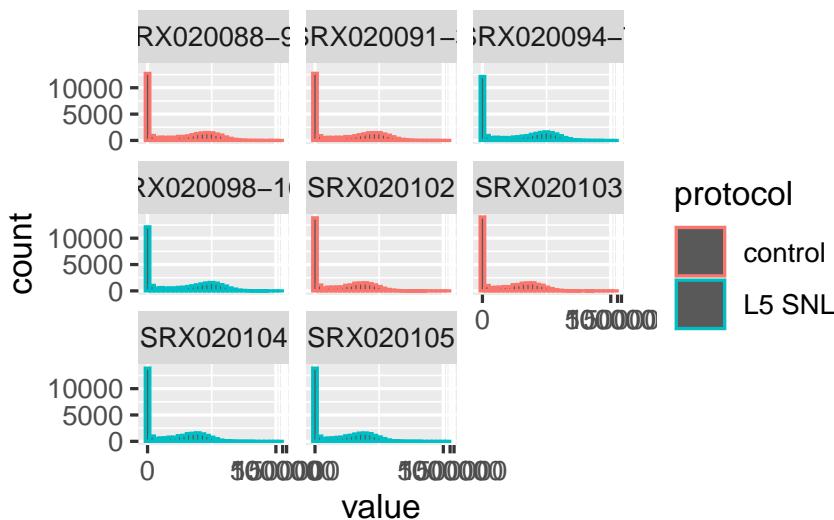


You can also incorporate data that are stored in the phenoData slot of the ExpressionSet object by specifying addPheno = TRUE:

```
## # A tibble: 236,128 x 8
##   gene           sample sample.id num.tech.reps protocol strain Time  value
##   <chr>          <chr>    <fct>        <dbl> <fct>    <fct> <fct> <int>
## 1 ENSRNOG000000000001 SRX02~ SRX020102          1 control Sprag~ 2 mo~     2
## 2 ENSRNOG000000000007 SRX02~ SRX020102          1 control Sprag~ 2 mo~     4
## 3 ENSRNOG000000000008 SRX02~ SRX020102          1 control Sprag~ 2 mo~     0
## 4 ENSRNOG000000000009 SRX02~ SRX020102          1 control Sprag~ 2 mo~     0
## 5 ENSRNOG000000000010 SRX02~ SRX020102          1 control Sprag~ 2 mo~    19
## 6 ENSRNOG000000000012 SRX02~ SRX020102          1 control Sprag~ 2 mo~     7
## 7 ENSRNOG000000000014 SRX02~ SRX020102          1 control Sprag~ 2 mo~     0
## 8 ENSRNOG000000000017 SRX02~ SRX020102          1 control Sprag~ 2 mo~     4
## 9 ENSRNOG000000000021 SRX02~ SRX020102          1 control Sprag~ 2 mo~     7
## 10 ENSRNOG000000000024 SRX02~ SRX020102         1 control Sprag~ 2 mo~    86
## # ... with 236,118 more rows
```

With this addition, visualizations can easily be changed to also show the experimental group of each sample:

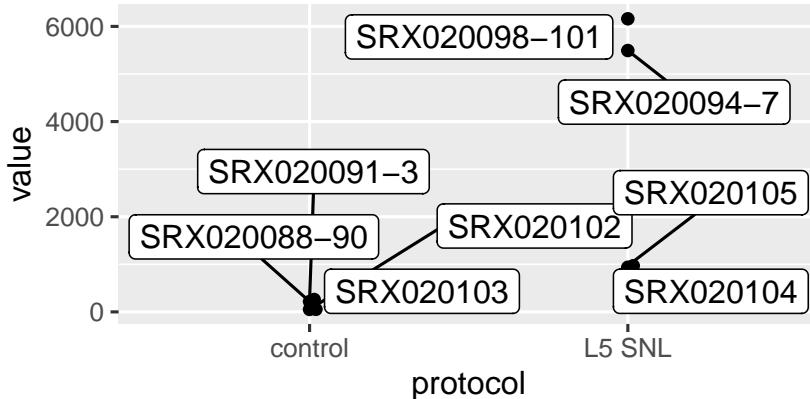
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



You can also do things like look at differences in values for specific genes, pairing tools for exploring data with tools for visualization, both from the tidyverse:

Values by sample for gene ENSRNOG0001

Samples are labeled with their sample ID



The data for a subset of the sample can be analyzed using functions from the edgeR package, to complete needed pre-processing (for example, calculating normalizing factors with `calcNormFactors`, to reduce impacts from technical bias [?]), estimate dispersion using conditional maximum likelihood and empirical Bayesian methods (`estimateCommonDisp` and `estimateTagwiseDisp`), and then perform a statistical analysis, conducting a differential expression analysis (`exactTest`) (Chen et al., 2014).

The data are stored in a special Bioconductor class, as an instance of `DGEList`, throughout most of this process. This special class can be initialized with data from the original `ExpressionSet` object, specifically, assay data with the counts per gene in each sample and data on the experimental phenotypes for the experiment—specifically, the protocol for each sample, in terms of whether it was a control or if the sample was from an animal in which

chronic neuropathic pain was induced (Hammer et al., 2010).

```
## [1] "DGEList"
## attr(,"package")
## [1] "edgeR"
```

Once the data are stored in this special DGEList class, different steps of the preprocessing can be conducted. In each case, the results are stored in special slots of the DGEList object. In this way, the original data and results from preprocessing are all kept together in a single object, each in a special slot within the object's structure.

```
## [1] "DGEList"
## attr(,"package")
## [1] "edgeR"

## [1] "DGEList"
## attr(,"package")
## [1] "edgeR"

## [1] "DGEList"
## attr(,"package")
## [1] "edgeR"
```

After the preprocessing, the data can be analyzed using the exactText function. This inputs the data stored in a DGEList object and outputs results into a different object class, a DGEEexact class.

```
## [1] "DGEEexact"
## attr(,"package")
## [1] "edgeR"
```

The DGEEexact class is defined by the edgeR package and was created specifically to store the results from a differential expression analysis (Chen et al., 2014). It has slots for a dataframe giving the estimates of differential change in expression across the experimental groups for each gene, within a table slot. Again, in this output, gene identifiers are stored as rownames—which makes them hard to access with tidyverse tools—rather than in their own column:

```
##          logFC      logCPM      PValue
## ENSRNOG000000000001  2.64635814  1.49216267 1.309933e-06
## ENSRNOG000000000007 -0.40869816 -0.22616605 1.000000e+00
## ENSRNOG000000000008  2.22296029 -0.40665547 1.288756e-01
## ENSRNOG000000000009  0.00000000 -1.31347471 1.000000e+00
## ENSRNOG000000000010  0.03307909  1.79448965 1.000000e+00
## ENSRNOG000000000012 -3.39210151  0.07939132 3.745676e-03
```

The DGEExact object also has a slot that contains a vector with identifiers for the two experimental groups that are being compared in the differential expression analysis, under the slot `comparison`:

```
## [1] "control" "L5 SNL"
```

There is also a space in this object class where information about each gene can be stored, if desired.

Two `biobroom` methods are defined for the DGEExact object class, `glance` and `tidy`. The `tidy` method extracts the results from the differential expression analysis, but moves these results into a dataframe where the gene names are given their own column, rather than being stored in the hard-to-access rownames:

```
## # A tibble: 29,516 x 4
##   gene       estimate logCPM    p.value
##   <chr>     <dbl>    <dbl>      <dbl>
## 1 ENSRNOG000000000001  2.65     1.49  0.00000131
## 2 ENSRNOG000000000007 -0.409   -0.226   1
## 3 ENSRNOG000000000008  2.22    -0.407  0.129
## 4 ENSRNOG000000000009  0        -1.31   1
## 5 ENSRNOG000000000010  0.0331   1.79   1
## 6 ENSRNOG000000000012 -3.39     0.0794 0.00375
## 7 ENSRNOG000000000014  3.65    -0.854  0.252
## 8 ENSRNOG000000000017  2.42     1.11  0.0000638
## 9 ENSRNOG000000000021 -2.02     0.211  0.0373
## 10 ENSRNOG000000000024 0.133    3.97  0.508
## # ... with 29,506 more rows
```

Now that the data are in this tidy format, tools from the `tidyverse` can be easily applied. For example, you could use functions from the `dplyr` package to see the genes for which the differential expression analysis resulted in both a very low p-value and a large difference in expression across the experimental groups:

```
## # A tibble: 803 x 4
##   gene       estimate logCPM    p.value
##   <chr>     <dbl>    <dbl>      <dbl>
## 1 ENSRNOG00000013496   6.39     3.41 5.95e- 46
## 2 ENSRNOG00000001338   6.01     2.25 1.37e- 22
## 3 ENSRNOG00000020136   5.17     3.89 3.19e- 52
## 4 ENSRNOG00000018808   4.78     6.46 1.75e-169
## 5 ENSRNOG00000006151   4.43     2.99 2.39e- 30
## 6 ENSRNOG00000009768   4.40     7.83 5.57e-293
## 7 ENSRNOG00000030927  -4.29     2.45 6.32e- 23
## 8 ENSRNOG00000001476   4.25     3.76 1.45e- 47
```

```
##  9 ENSRNOG00000004805      4.05   6.64 2.51e-185
## 10 ENSRNOG00000014327     3.85   4.51 5.39e- 63
## # ... with 793 more rows
```

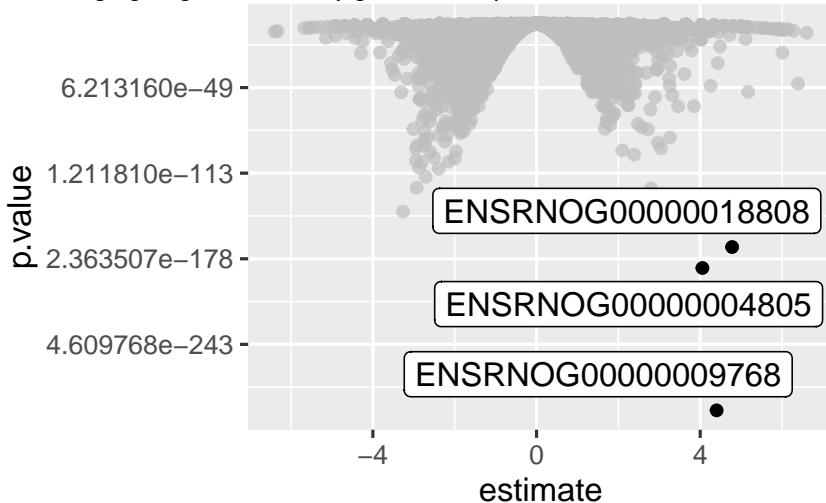
Other exploratory analysis will also be straightforward with the data using tidyverse tools, now that they are in a “tidy” format.

The `glance` method can also be applied to data that are stored in a `DGEExact` class. In this case, the method will extract the names of the experimental groups being compared (from the `comparison` slot of the object) as well as count the number of genes with statistically significant differences in expression level, based on the values in the `table` slot of the object.

```
##   significant      comparison
## 1           6341 control/L5 SNL

##   significant      comparison
## 1           4225 control/L5 SNL
```

As another example, you can now use tools from `ggplot2`, as well as extensions built on this package, to do things like create a volcano plot of the data with highlighting of noteworthy genes on the plot:



If you wanted to access the help files for these `biobroom` methods for this object class, you could do so by calling `help` in R (?) using the name of the method, a dot, and then the name of the object class, e.g., `?tidy.DGEExact`.

3.6.6 The `ggbio` package

3.6.7 Subsection 2

“The `biobroom` package contains methods for converting standard objects in Bioconductor into a ‘tidy format’. It serves as a complement to the popular `broom` package, and follows the same division (`tidy/augment/glance`) of tidying methods.” (Bass et al., 2020)

"Tidying data makes it easy to recombine, reshape and visualize bioinformatics analyses. Objects that can be tidied include: ExpressionSet object, GRanges and GRangesList objects, RangedSummarizedExperiment object, MSnSet object, per-gene differential expression tests from limma, edgeR, and DESeq2, qvalue object for multiple hypothesis testing." (Bass et al., 2020)

"We are currently working on adding more methods to existing Bioconductor objects." (Bass et al., 2020)

"All biobroom tidy and augment methods return a `tbl_df` by default (this prevents them from printing many rows at once, while still acting like a traditional `data.frame`)." (Bass et al., 2020)

"The concept of 'tidy data' offers a powerful framework for structuring data to ease manipulation, modeling and visualization. However, most R functions, both those builtin and those found in third-party packages, produce output that is not tidy, and that is therefore difficult to reshape, recombine, and otherwise manipulate. Here I introduce the broom package, which turns the output of model objects into tidy data frames that are suited to further analysis, manipulation, and visualization with input-tidy tools." (Robinson, 2014)

"Tools are classified as 'messy-output' if their output does not fit into this [tidy] framework. Unfortunately, the majority of R modeling tools, both from the built-in stats package and those in common third party packages, are messy-output. This means the data analyst must tidy not only the original data, but the results at each intermediate stage of an analysis." (Robinson, 2014)

"The broom package is an attempt to solve this issue, by bridging the gap from untidy outputs of predictions and estimations to create tidy data that is easy to manipulate with standard tools. It centers around three S3 methods, `tidy`, `augment`, and `glance`, that each take an object produced by R statistical functions (such as `lm`, `t.test`, and `nls`) or by popular third-party packages (such as `glmnet`, `survival`, `lme4`, and `multcomp`) and convert it into a tidy data frame without rownames (Friedman et al., 2010; Therneau, 2014; Bates et al., 2014; Hothorn et al., 2008). These outputs can then be used with input-tidy tools such as `dplyr` or `ggplot2`, or downstream statistical tests. broom should be distinguished from packages such as `reshape2` and `tidyr`, which rearrange and reshape data frames into different forms (Wickham, 2007b, 2014b). Those packages perform essential tasks in tidy data analysis but focus on manipulating data frames in one specific format into another. In contrast, broom is designed to take data that is not in a data frame (sometimes not anywhere close) and convert it to a tidy data frame." (Robinson, 2014)

"`tidy` constructs a data frame that summarizes the model's statistical components, which we refer to as the component level. In a regression such as the above it may refer to coefficient estimates, p-values, and standard errors for each term in a regression. The `tidy` generic is flexible- in other models it could represent per-cluster information in clustering applications, or per-test information for multiple comparison functions. ... `augment` add columns to the original data that was modeled, thus working at the observation level. This includes predictions, residuals and prediction standard errors in a regression, and can represent cluster assignments or classifications in other applications. By convention, each new column starts with `.` to ensure it does not conflict with existing columns. To ensure

that the output is tidy and can be recombined, rownames in the original data, if present, are added as a column called `.rownames`. ... Finally, `glance` constructs a concise one-row summary of the model level values. In a regression this typically contains values such as `R2`, `adjusted R2`, residual standard error, Akaike Information Criterion (AIC), or deviance. In other applications it can include calculations such as cross validation accuracy or prediction error that are computed once for the entire model. ... These three methods appear across many analyses; indeed, the fact that these three levels must be combined into a single S3 object is a common reason that model outputs are not tidy. Importantly, some model objects may have only one or two of these methods defined. (For example, there is no sense in which a Student's T test or correlation test generates information about each observation, and therefore no `augment` method exists)." (Robinson, 2014)

"While model inputs usually require tidy inputs, such attention to detail doesn't carry over to model outputs. Outputs such as predictions and estimated coefficients aren't always tidy. For example, in R, the default representation of model coefficients is not tidy because it does not have an explicit variable that records the variable name for each estimate, they are instead recorded as row names. In R, row names must be unique, so combining coefficients from many models (e.g., from bootstrap resamples, or subgroups) requires workarounds to avoid losing important information. This knocks you out of the flow of analysis and makes it harder to combine the results from multiple models." (Wickham, 2014)

"edgeR can be applied to differential expression at the gene, exon, transcript or tag level. In fact, read counts can be summarized by any genomic feature. edgeR analyses at the exon level are easily extended to detect differential splicing or isoform-specific differential expression." (Chen et al., 2014)

"edgeR provides statistical routines for assessing differential expression in RNA-Seq experiments or differential marking in ChIP-Seq experiments. The package implements exact statistical methods for multigroup experiments developed by Robinson and Smyth [33, 34]. It also implements statistical methods based on generalized linear models (glms), suitable for multifactor experiments of any complexity, developed by McCarthy et al. [22], Lund et al. [20], Chen et al. [5] and Lun et al. [19]. ... A particular feature of edgeR functionality, both classic and glm, are empirical Bayes methods that permit the estimation of gene-specific biological variation, even for experiments with minimal levels of biological replication." (Chen et al., 2014)

"edgeR performs differential abundance analysis for pre-defined genomic features. Although not strictly necessary, it usually desirable that these genomic features are non-overlapping. For simplicity, we will hence-forth refer to the genomic features as 'genes', although they could in principle be transcripts, exons, general genomic intervals or some other type of feature. For ChIP-seq experiments, abundance might relate to transcription factor binding or to histone mark occupancy, but we will henceforth refer to abundance as in terms of gene expression." (Chen et al., 2014)

"edgeR stores data in a simple list-based data object called a DGEList. This type of object is easy to use because it can be manipulated like any list in R. ... The main components of an DGEList object are a matrix `counts` containing the integer counts, a `data.frame` `samples` containing information about the samples

or libraries, and a optional data.frame genes containing annotation for the genes or genomic features. The data.frame samples contains a column lib.size for the library size or sequencing depth for each sample. If not specified by the user, the library sizes will be computed from the column sums of the counts. For classic edgeR the data.frame samples must also contain a column group, identifying the group membership of each sample.” (Chen et al., 2014)

“Genes with very low counts across all libraries provide little evidence for differential expression. In the biological point of view, a gene must be expressed at some minimal level before it is likely to be translated into a protein or to be biologically important. In addition, the pronounced discreteness of these counts interferes with some of the statistical approximations that are used later in the pipeline. These genes should be filtered out prior to further analysis. As a rule of thumb, genes are dropped if they can’t possibly be expressed in all the samples for any of the conditions. Users can set their own definition of genes being expressed. Usually a gene is required to have a count of 5-10 in a library to be considered expressed in that library. Users should also filter with count-per-million (CPM) rather than filtering on the counts directly, as the latter does not account for differences in library sizes between samples.” (Chen et al., 2014)

“The most obvious technical factor that affects the read counts [in data for edgeR and so requires normalizations], other than gene expression levels, is the sequencing depth of each RNA sample. edgeR adjusts any differential expression analysis for varying sequencing depths as represented by differing library sizes. This is part of the basic modeling procedure and flows automatically into fold-change or p-value calculations. It is always present, and doesn’t require any user intervention.” (Chen et al., 2014)

“In edgeR, normalization takes the form of correction factors that enter into the statistical model. Such correction factors are usually computed internally by edgeR functions, but it is also possible for a user to supply them. The correction factors may take the form of scaling factors for the library sizes, such as computed by calcNormFactors, which are then used to compute the effective library sizes. Alternatively, gene-specific correction factors can be entered into the glm functions of edgeR as offsets. In the latter case, the offset matrix will be assumed to account for all normalization issues, including sequencing depth and RNA composition. Note that normalization in edgeR is model-based, and the original read counts are not themselves transformed. This means that users should not transform the read counts in any way before inputting them to edgeR.” (Chen et al., 2014)

“Recent work on a grammar of graphics could be extended for biological data. The grammar of graphics is based on modular components that when combined in different ways will produce different graphics. This enables the user to construct a combinatoric number of plots, including those that were not preconceived by the implementation of the grammar. Most existing tools lack these capabilities.” (Yin et al., 2012)

“A new package, ggbio, has been developed and is available on Bioconductor. The package provides the tools to create both typical and non-typical biological plots for genomic data, generated from core Bioconductor data structures by either the high-level autoplot function, or the combination of low-level components of the grammar of graphics. Sharing data structures with the rest of Bioconductor enables direct integration with Bioconductor workflows.” (Yin et al., 2012)

"In ggbio, most of the functionality is available through a single command, autoplot, which recognizes the data structure and makes a best guess of the appropriate plot. ... Compared to the more general qplot API of ggplot2, autoplot facilitates the creation of specialized biological graphics and reacts to the specific class of object passed to it. Each type of object has a specific set of relevant graphical parameters, and further customization is possible through the low-level API." (Yin et al., 2012)

The ggbio package has autoplot functions for many Bioconductor object classes, including GRangesList, ... [Yin et al. (2012)]

"The grammar [of graphics] is composed of interchangeable components that are combined according to a flexible set of rules to produce plots from a wide range of types." (Yin et al., 2012)

"Data are the first component of the grammar... The ggbio package attempts to automatically load files of specific formats into common Bioconductor data structures, using routines provided by Bioconductor packages... The type of data structure loaded from a file or returned by an algorithm depends on the intrinsic structure of the data. For example, BAM files are loaded into a GappedAlignments, while FASTA and 2bit sequences result in a DNAStringSet. The ggbio package handles each type of data structure differently... In summary, this abstraction mechanism allows ggbio to handle multiple file formats, without discarding any intrinsic properties that are critical for effective plotting." (Yin et al., 2012)

"Genomic data have some specific features that are different from those of more conventional data types, and the basic grammar does not conveniently capture such aspects. The grammar of graphics is extended by ggbio in several ways... These extensions are specific to genomic data, that is, genomic sequences and features, like genes, located on those sequences." (Yin et al., 2012)

"A geom is responsible for translating data to a visual, geometric representation according to mappings between variables and aesthetic properties on the geom. In comparison to regular data elements that might be mapped to the ggplot2 geoms of points, lines, and polygons, genomic data has the basic currency of a range. Ranges underlie exons, introns, and other features, and the genomic coordinate system forms the reference frame for biological data. We have introduced or extended several geoms for representing ranges and gaps between ranges. ... For example, the alignment geom delegates to two other geoms for drawing the ranges and gaps. These default to rectangles and chevrons, respectively. Having specialized geoms for commonly encountered entities, like genes, relegates the tedious coding of primitives, and makes use code simpler and more maintainable." (Yin et al., 2012)

"Coordinate systems locate points in space, and we use coordinate transformations to map from data coordinates to plot coordinates. The most common coordinate system in statistical graphics is cartesian. The transformation of data to cartesian coordinates involves mapping points onto a plane specified by two perpendicular axes (x and y). Why would two plots transform the coordinates differently for the same data? The first reason is to simplify, such as changing curvilinear graphics to linear, and the second reason is to reshape a graphic so

that the most important information jumps out at the viewer or can be more accurately perceived. Coordinate transformations are also important in genomic data visualization. For instance, features of interest are often small compared to the intervening gaps, especially in gene models. The exons are usually much smaller than the introns. If users are generally interested in viewing exons and associated annotations, we could simply cut or shrink the intervening introns to use the plot space efficiently.” (Yin et al., 2012)

“Almost all experimental outputs are associated with an experimental design and other meta-data, for example, cancer types, gender, and age. Faceting allows users to subset the data by a combination of factors and then lay out multiple plots in a grid, to explore relationships between factors and other variables. The ggplot2 package supports various types of facetting by arbitrary factors. The ggbio package extends this notion to facet by a list of ranges of interest, for example, a list of gene regions. There is always an implicit facetting by sequence (chromosome), because when the x axis is the chromosomal coordinate, it is not sensible to plot data from different chromosomes on the same plot.” (Yin et al., 2012)

“For custom use cases, ggbio provides a low-level API that maps more directly to components of the grammar and thus expresses the plot more explicitly. Generally speaking, we strive to provide sensible, overrideable defaults at the high-level entry points, such as autoplot, while still supporting customizability through the low-level API. All lower level functions have a special prefix to indicate their role in the grammar, like layout, geom, stat, coord, and theme. The objects returned by the low-level API may be added together via the conventional + syntax. This facilitates the creation of new types of plots. A geom in ggplot2 may be extended to work with more biological data model, for example, geom_rect will automatically figure out the boundary of rectangles when the data is a GRanges, as to geom_bar, geom_segment, and so on.” (Yin et al., 2012)

“We use ggplot2 as the foundation for ggbio, due to its principled style, intelligent defaults and explicit orientation towards the grammar of graphics model.” (Yin et al., 2012) (Yin et al., 2012)

When you have different types and formats of input, you need different tools to operate on them. For example, you’d (ideally) use different types of scissors depending on whether you needed to cut paper or cloth or metal (Savage, 2020). In a similar way, you can think of these “classes” of objects in R as different materials or types of inputs, and therefore a function that works when you input data in one object class won’t work with a different object class. The beauty of the tidyverse set of tools is that their system enforces a common input “material”—the tidy dataframe—and as a result all the functions work on this type of input. You can always, in other words, assume that you are working with one material (say paper), and what’s more, you can (almost) always assume that you’ll get your output in that “material” as well. That means that you have a set of tools that works on one type of material (scissors for paper, glue for paper, tape for paper, pens for writing on paper), but you can combine them in different ways across your pipeline, because no matter how much you’ve cut or glued or written on your paper at any given point, it’s still paper, and so you can still use paper-focused tools.

3.6.8 Applied exercise

3.7 Introduction to reproducible data pre-processing protocols

Reproducibility tools can be used to create reproducible data pre-processing protocols—documents that combine code and text in a “knitted” document, which can be re-used to ensure data pre-processing is consistent and reproducible across research projects. In this module, we will describe how reproducible data pre-processing protocols can improve reproducibility of pre-processing experimental data, as well as to ensure transparency, consistency, and reproducibility across the research projects conducted by a research team.

Objectives. After this module, the trainee will be able to:

- Define a “reproducible data pre-processing protocol”
- Explain how such protocols improve reproducibility at the data pre-processing phase
- List other benefits, including improving efficiency and consistency of data pre-processing
- Understand how a “knitted” document can be used to combine text and executable code to create a reproducible data pre-processing protocol

3.7.1 Introducing reproducible data pre-processing protocols

If you have ever worked in a laboratory, you are likely familiar with protocols. For a wet lab, protocols are used as “recipes” for conducting certain experiments or processes. They are written to be clear enough that everyone in the lab could follow the same steps in the process by following the protocol. In this way, they help to standardize processes done in the laboratory, and they can also play a role in improving safety and the quality of data collection. Protocols are similarly used for medical procedures and tests, as well as for clinical trials. In all cases, they help to define in detail the steps of the procedure, so they can be done in a way that is comparable from one case to the next and with high precision.

You can apply a similar idea to pre-processing and analyzing the data that you collect in a laboratory. Just as a wet lab protocol can help standardize your data collection to the point that the data are recorded, a separate protocol can help define how you manage and work with that data. The basic content of a data-focused protocol will include a description of the type of data you expect to input, the type of data you expect at the end of the process, and the steps you take to get from the input to the output. A data-focused protocol can include steps for quality control of the collected data, as well as pre-processing steps like transformations and scaling of the data.

In module 3.9, we’ll walk through an example of creating a data pre-processing protocol that focuses on data collected by plating samples to estimate bacterial load. In this case, a key step in pre-processing the data is to identify a “good” dilution to be used for estimating bacterial load in each sample—each sample

On clinical imaging protocols: “When one is composing a protocol, it is helpful to imagine that all the technologists at the facility won the lottery and quit. What would a newly hired technologist need to know to image a patient in the exact same manner as in the past to produce the same results?”

[@thomas2015write]

is plated at several dilutions, and to work with the data, you must identify a dilution for each sample for which enough bacteria grew to be countable, but not so many that there are too many colonies to count. In high throughput experiments, like RNA-seq experiments, there may be important steps in the data pre-processing that help check for batch effects across samples, for signs of a poor-quality sample, or for normalizing and scaling the data in preparation for applying other algorithms, like algorithms to estimate differential expression across samples or to identify clusters within the data.

A data-focused protocol brings many of the same advantages as wet lab protocols. It can help standardize the process of data pre-processing across members of the laboratory, as well as from experiment to experiment. It can also help ensure the quality of the data collection, by defining clear rules, steps, and guidelines for completing the data pre-processing. Finally, it can help ensure that someone else could recreate the process at a later time, and so can improve the reproducibility of the experiment. Not only do data-focused protocols help with improving quality and reproducibility, but they also help improve efficiency. These protocols should include clearly defined steps, as well as explanations for each step, and they should illustrate these with example data. By having this “recipe”, a new lab member can quickly learn how to do the data pre-processing, and a long-term lab member remember the exact steps more quickly.

You can create a data pre-processing protocol using any document processing program that you’d like. For example, you could write one in Google Docs or in Word. However, there is a better format. With programming languages like R and Python, you can create a type of document called a **knitted document**. A knitted document interweaves two elements: first, text written for humans and second, executable code meant for the computer. These documents can be “rendered” in R or another programming language, which executes all the code and adds all the output from that code at the appropriate place in the text. The end result is a document in a format that is easy to share and read (PDF, Word, or HTML), which includes text, example code, and output. You can use these documents to record the data pre-processing process for a type of data in your laboratory, and by using a knitted document, you ensure that the code is “checked” every time you render the document. In this module, we will give an overview of how these knitted documents work, as well as how they can improve the reproducibility and efficiency of experimental work. In the next module, we’ll show how you can make them in the free RStudio software. Finally, in module 3.9, we’ll walk through a full example of writing a data pre-processing protocol in this way—you can take a look now to get an idea by downloading the example protocol here. There are also some excellent data-focused protocols that have been published in journals like *Nature Protocols*. Some recent examples of such protocols include Schrode et al. (2021), Quintelier et al. (2021), and Majumder et al. (2021). You may find it useful to take a look at one or more to get an idea of how data-focused protocols can be

useful.

3.7.2 Using knitted documents for protocols

When it comes to protocols that are focused on data pre-processing and analysis, there are big advantages to creating them as something called **knitted documents**. In this section, we'll walk through what a knitted document is, and in the next section we'll cover some of the advantages of using this format to create data-focused protocols.

A knitted document is one that is written in plain text in a way that “knits” together text with executable code. Once you have written the document, you can render it, which executes the code, adds to the document results from this execution (figures, tables, and code output, for example), and formats all text using the formatting choices you've specified. The end result is a nicely format document, which can be in one of several output formats, including PDF, Word, or HTML. Since the code was executed to create the document, you can ensure that all the code has worked as intended.

If you have coded using a scripting language like R or Python, you likely have already seen many examples of knitted documents. For both these languages, there are many tutorials available that are created as knitted documents. Figure 3.4 shows an example from the start of a vignette for the `xcms` package in R. This is a package that helps with pre-processing and analyzing data from liquid chromatography–mass spectrometry (LC–MS) experiments. You can see that this document includes text to explain the package and also example code and the output from that code. As a larger example, all the modules in this online book were written as knitted documents.

3 Initial data inspection

The `OnDiskMSnExp` organizes the MS data by spectrum and provides the methods `getSpectra()`, `mz` and `rtime` to access the raw data from the files (the measured intensity and the corresponding m/z and retention time values). In addition, the `spectra` method is used to return all data encapsulated in `Spectrum` objects. Below we extract the retention time values from the object.

```
head(rtime(raw_data))

## F1.S0001 F1.S0002 F1.S0003 F1.S0004 F1.S0005 F1.S0006
## 2501.378 2502.943 2504.508 2506.073 2507.638 2509.203
```

Figure 3.4: An example of a knitted document. This shows a section of the online vignette for the ‘`xcms`’ package from Bioconductor. The two types of content are highlighted: formatted text for humans to read, and executable computer code.

You can visualize the full process of creating and rendering a knitted document in the following way. Imagine that you write a document by hand on sheets of paper. There are parts where you need a team member to add their data or to run a calculation, so you include notes in square brackets telling your team member where to do these things. Then, you use some editing marks to show where text should be italicized and which text should be section a header:

Results

We measured the bacterial load of
Mycobacterium tuberculosis for each sample.

[Kristina: Calculate bacterial loads for each sample based on dilutions and add table with results here.]

You send the document to your team member Kristina first, and she does her calculations and adds the results at the indicated spot in the paper, so that the note to her gets replaced with results. She focuses on the notes to her in square brackets and ignores the rest of the document. Next, Kristina sends the document, with her additions, to an assistant, Tom, to type up the document. Tom types the full document, paying attention to any indications that are included for formatting. For example, he sees that “Results” is meant to be a section heading, since it is on a line that starts with “#”, your team’s convention for section headings. He therefore types this on a line by itself in larger font. He also sees that “*Mycobacterium tuberculosis*” is surrounded by asterisks, so he types this in italics.

Knitted documents work in the same way, but the computer does the steps that Kristina and Tom did in this toy example. The way the document was written in this example is analogous to writing up a knitted document in plain text with appropriate “executable” sections, designated with special markings, and with other markings used to show how the text should be formatted in its final version. When Kristina looked for the section that was marked for her, generated results in that section, and replaced the note with the results, it was analogous to the first stage of rendering a knitted document, where the document is passed through software that looks for executable code and ignores everything else, executing that code and adding in results in the right place. When Tom took that output and used formatting marks in the text to create a nicely formatted final report, the step was analogous to the second stage of rendering a formatted document, when a software program takes the output of the first stage and formats the full document into an attractive, easy-to-read final document, using any markings you include to format the document.

Knitted documents therefore build on two key techniques. The first is the ability to include executable code in a document, in a way that a computer can go through the document, find that code, execute it, and fill in the results at the appropriate spot in the document. The second is a set of conventions for formatting marks that can be put in the plain text of the document to indicate formatting that should be added, like headers and italic text. Let’s take a closer look at each of these necessary techniques.

The first technique that's needed to create knitted documents is the ability to include executable code within the plain text version of the document. The idea here is that you can use special markers to indicate in the document where code starts and where it ends. With these markings, a computer program can figure out the lines of the document that it should run as code, and the ones it should ignore when it's looking for executable code. In the toy example above, notes to Kristina were put in square brackets, with content that started with her name and a colon. To "process" this document, then, she could just scan through it for square brackets with her name inside and ignore everything else in the document.

The same idea happens with knitted documents, but a computer program takes the place of Kristina in the example. With markings in place to indicate executable code, the document will be run through two separate programs as it is rendered. The first program will look for code to execute and ignore any other lines of the file. It will execute this code and then place any results, like figures, tables, or code output, into the document right after that piece of code. We will talk about the second program in just a minute, when we talk about markup languages.

This technique comes from an idea that you could include code to be executed in a document that is otherwise easy for humans to read. This is an incredibly powerful idea. It originated with a famous computer scientist named Donald Knuth, who realized that one key to making computer code sound is to make sure that it is clear to humans what the code is doing. Computers will faithfully do exactly what you tell them to do, so they will do what you're hoping they will as long as you provide the correct instructions. The greatest room for error, then, comes from humans not giving the right instructions to computers. To write sound code, and code that is easy for yourself and others to maintain and extend, you must make sure that you and other humans understand what it is asking the computer to do. Donald Knuth came up with a system called "literate programming" that allows programmers to write code in a way that focuses on documenting the code for humans, while also allowing the computer to easily pull out just the parts that it needs to execute, while ignoring all the text meant for humans. This process flips the idea of documenting code by including plain text comments in the code—instead of the code being the heart of the document, the documentation of the code is the heart, with the code provided to illustrate the implementation. When used well, this technique results in beautiful documents that clearly and comprehensively document the intent and the implementation of computer code. The knitted documents that we can build with R or Python through systems like RMarkdown and Jupyter Notebooks build on these literate programming ideas, applying them in ways that complement programming languages that can be run interactively, rather than needing to be compiled before they're run.

The second technique required for knitted documents is one that allows you to write text in plain text, include formatting specifications in that plain text,

and render this to an attractive output document in PDF, Word, or HTML. This part of the process uses a tool from a set of tools called **Markup languages**. Here, we will use a markup language called **Markdown**. It is one of the easiest markup languages to learn, as it has a fairly small set of formatting indicators that can be used to “markup” the formatting in a document. This small set, however, covers much of the formatting you might want to do, and so this language provides an easy introduction to markup languages while still providing adequate functionality for most purposes.

The Markdown markup language evolved starting in spaces where people could communicate in plain text only, without point-and-click methods for adding formatting like bold or italic type (Buffalo, 2015). For example, early versions of email only allowed users to write using plain text. These users eventually evolved some conventions for how to “mark-up” this plain text, to serve the purposes normally served by things like italics and bold in formatted text (e.g., emphasis, highlighting). For example, to emphasize a word, a user could surround it with asterisks, like:

I just read a **really** interesting article!

In this early prototype for a markup language, the reader’s mind was doing the “rendering”, interpreting these markers as a sign that part of the text was emphasized. In Markdown, the text can be rendered into more attractive output documents, like PDF, where the rendering process has actually changed the words between asterisks to print in italics.

The Markdown language has developed a set of these types of marks—like asterisks—that are used to “mark up” the plain text with the formatting that should be applied when the text is rendered. There are marks that you can use for a number of formatting specifications, including: italics, bold, underline, strike-through, bulleted lists, numbered lists, web links, headers of different levels (e.g., to mark off sections and subsections), horizontal rules, and block quotes. Details and examples of the Markdown syntax can be found on the Markdown Guide page at <https://www.markdownguide.org/basic-syntax/>, and we’ll cover more examples of using Markdown in the next two modules. Once a document is run through a program to execute any code, it will then be run through a program that interprets this formatting markup (a markup renderer), which will format the document based on any of the mark up indications and will output an attractive document in a format like PDF, Word, or HTML.

3.7.3 Advantages of using knitted documents for data-focused protocols

There are several advantages to using knitted documents when writing code to pre-process or analyze research data. These include improvements in terms of reliability, efficiency, transparency, and reproducibility.

First, when you have written your code within a knitted document, this code is checked every time you render the document. In other words, you

“Markdown originates from the simple formatting conventions used in plain-text emails. Long before HTML crept into email, emails were embellished with simple markup for emphasis, lists, and blocks of text. Over time, this became a defacto plain-text email formatting scheme. This scheme is very intuitive: underscores or asterisks that flank text indicate emphasis, and lists are simply lines of text beginning with dashes.”

[@buffalo2015bioinformatics]

are checking your code to ensure it operates as you intend throughout the process of writing and editing your document, checking the code each time you render the document to its formatted version. This helps to increase the **reliability** of the code that you have written. Open-source software evolves over time, and by continuing to check code as you work on protocols and reports with your data, you can ensure that you will quickly identify and adapt to any such changes. Further, you can quickly identify if updates to your research data introduce any issues with the code. Again, by checking the code frequently, you can identify any issues quickly, and this often will allow you to easily pinpoint and fix these issues. By contrast, if you only identify a problem after writing a lot of code, it is often difficult to identify the source of the issue. By including code that is checked each time of document is rendered, you can quickly identify when a change in open source software affects the analysis that you were conducting or the pre-processing and work to adapt to any changes quickly.

Second, when you write a document that includes executable code, it allows you to easily rerun the code as you update your research data set, or adopt the code to work with a new data set. If you are not using a knitted document to write pre-processing protocols and research reports, then your workflow is probably to run all your code—either from a script or the command line—and copy the results into a document in a word processing program like Word or Google Docs. If you do that, you must recopy all your results every time you adapt any part of the code or add new data. By contrast, when you use a knitted document, the rendering process executes the code and incorporates the results directly and automatically into a nicely formatted final document. The use of knitted documents therefore can substantially improve the **efficiency** of pre-processing and analyzing your data and generating the reports that summarize this process.

Third, documents that are created in knitted format are created using plain text. Plain text files can easily be tracked well and clearly using version control tools like git, and associated collaboration tools like GitHub, as discussed in earlier modules (modules 2.9–2.11). This substantially increases the **transparency** of the data pre-processing and analysis. It allows you to clearly document changes you or others make in the document, step-by-step. You can document who made the change, and that person can include a message about why they made the change. This full history of changes is recorded and can be searched to explore how the document has evolved and why.

The final advantage of using knitted documents, especially for pre-processing research data, is that it allows the code to be clearly and thoroughly documented. This can help increase the **reproducibility** of the process. In other words, it can help ensure that another researcher could repeat the same process, making adaptations as appropriate for their own data set, or ensuring they arrive at the same results if using the original data. It also ensures that you can remember exactly what you did, which is especially useful if you plan to reuse

or adopt the code to work with other data sets, as will often be the case for a pre-processing protocol. If you are not using a knitted document, but are using code for preprocessing, then as an alternative you may be documenting your code through comments in a code script. A code script does allow you to include documentation about the code through these code comments, which are demarcated from code in the script through a special symbol (# in R). However these code comments are much less expressive and harder to read than nicely formatted text, and it is hard to include elements like mathematical equations and literature citations in code comments. A knitted document allows you to write the documentation in a format that is clear and attractive for humans to read, while including code that is clear and easy for a computer to execute.

3.7.4 How knitted documents work

Now that we've gotten a top-level view of the idea of knitted documents, let's take a closer look at how they work. We'll wrap up this module by covering some of the mechanics of how all knitted documents work, and then in the next module (3.8) we'll look more closely at how you can leverage these techniques in the RMarkdown system specifically.

There are seven components of how these documents work. It is helpful to understand these to understand these to begin creating and adapting knitted documents. Knitted documents can be created through a number of programs, and while we will later focus on Rmarkdown, these seven components are in play regardless of the exact system used to create a knitted document, and therefore help in gaining a general understanding of this type of document. We have listed the seven components here and in the following paragraphs will describe each more fully:

1. Knitted documents start as plain text;
2. A special section at the start of the document (**preamble**) gives some overall directions about the document;
3. Special combinations of characters indicate where the executable code starts;
4. Other special combinations show where the regular text starts (and the executable code section ends);
5. Formatting for the rest of the document is specified with a **markup language**;
6. You create the final document by **rendering** the plain text document. This process runs through two software programs; and
7. The final document is attractive and **read-only**—you should never make edits to this output, only to your initial plain text document.

First, a knitted document should be written in plain text. In an earlier module, we described some of the advantages of using plain text file formats, rather than proprietary and/or binary file formats, especially in the context of saving

research data (e.g., using csv file formats rather than Excel file formats). Plain text can also be used to write documentation, including through knitted documents. Figure 3.5 shows an example of what the plain text might look like for the start of the xcms tutorial shown in Figure 3.4.

Initial data inspection

The `OnDiskMSnExp` organizes the MS data by spectrum and provides the methods `intensity`, `mz` and `rtimes` to access the raw data from the files (the measured intensity values, the corresponding m/z and retention time values). In addition, the `spectra` method could be used to return all data encapsulated in `Spectrum` objects. Below we extract the retention time values from the object.

```
```{r data-inspection-rtimes, message = FALSE }
head(rttime(raw_data))
```

```

There are a few things to keep in mind when writing plain text. First, you should always use a text editor rather than a word processor when you are writing a document in plain text. Text editors can include software programs like Notepad on Microsoft operating systems and TextEdit on Mac operating systems. You can also use a more advanced text editor, like vi/vim or emacs. Rstudio can also serve as a text editor, and if you are doing other work in Rstudio, this is often the most obvious option as a text editor to use to write knitted documents.

You must use a text editor to write plain text for knitted documents for the same reasons that you must use one to write code scripts. Word processors often introduce formatting that is saved through underlying code rather than clearly evident on the document that you see as you type. This hidden formatting can complicate the written text. Conversely, text written in a text editor will not introduce such hard-to-see formatting. Word processing programs also tend to automatically convert some symbols into slightly fancier versions of the symbol. For example, they may change a basic quotation symbol into one with shaping, depending on whether the mark comes at the beginning or end of a quotation. This subtle change in formatting can cause issues in both the code and the formatting specifications that you include in a knitted document.

Further, when are writing plain text, typically you should only use characters from the American Standard Code for Information Interchange, or ASCII. This is a character set from early in computing that includes 128 characters. Such a small character set enforces simplicity: this character set mostly includes what you can see on your keyboard, like the digits 0 to 9, the lowercase and uppercase alphabet, some symbols, including punctuation symbols like the exclamation point and quotation marks, some mathematical symbols like plus, minus, and division, and some control codes, including ones for a new line, a tab, and even ringing a bell. The full set of characters included in ASCII can be found in a number of sources including a very thorough Wikipedia page on this

Figure 3.5: An example of a the plain text used to write a knitted document. This shows a section of the plain text used to write the online vignette for the 'xcm' package from Bioconductor. The full plain text file used for the vignette can be viewed on GitHub [here](<https://github.com/sneumann/xcm/blob/master/vignettes/xcm.Rmd>)

character set (<https://en.wikipedia.org/wiki/ASCII>).

Because the character set available for plain text files is so small, you will find that it becomes important to leverage the limited characters that are available. One example is **white space**. White space can be created in ASCII with both the space character and with the new line command. It is an important component that can be used to make plain text files clear for humans to read. As we begin discussing the convention for markdown languages, we will find that white space is often used to help specify formatting as well.

The second component of how knitted documents work is that each knitted document will have a special section at its start called the **preamble**. This preamble will give some overall directions regarding the document, like its title and authors and the format to which it should be rendered. Knitted documents are created using a **markup language** to specify formatting for the document, and there are a number of different markup languages including HTML, LaTeX, and Markdown. The specifications for the document's preamble will depend on the markup language being used.

In Rmarkdown, we will be focusing on Markdown, for which the preamble is specified using something called YAML (short for YAML Ain't Markup Language). Here is an example of the YAML for a sample pre-processing protocol created using RMarkdown:

```
---
```

```
title: "Preprocessing Protocol for LC-MS Data"
author: "Jane Doe"
date: "1/25/2021"
output: pdf_document
---
```

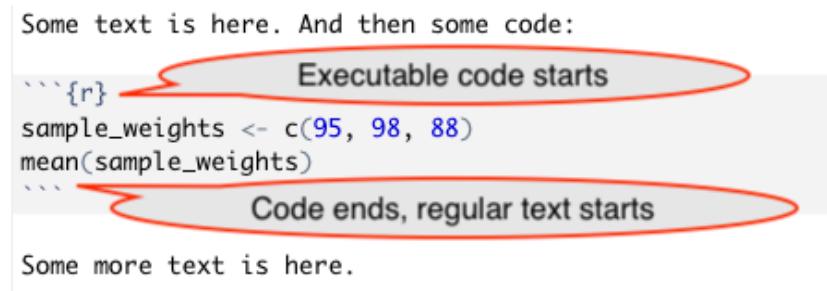
This YAML preamble specifies information about the document with **keys** and **values**. For example, the title is specified using the YAML key title, followed by a colon and a space, and then the desired value for that component of the document, "Preprocessing Protocol for LC-MS Data". Similarly, the author is specified with the author key and the desired value for that component, and the date with the date key and associated component.

Different keys can take different types of values in the YAML (this is similar to how different parameters in a function can take different values). For example, the keys of author, title, and date all take a character string with any desired character combination, and the quotation marks surrounding the values for each of these keys denote those character strings. By contrast, the output key—which specifies the format that the knitted document should be rendered to—can only take one of a few set values, each of which is specified without surrounding quotation marks (pdf_document in this case, to render the document as a PDF report).

The rules for which keys can be included in the preamble will depend on the markup language being used. Here, we are showing an example in Markdown,

but you can also use other markup languages like LaTeX and HTML, and these will have their own convention for specifying the preamble. In the next module, when we talk more specifically about Rmarkdown, we will give some resources where you can find more about how to customize the preamble in Rmarkdown specifically. If you are using a different markup language, there are numerous websites, cheatsheets, and other resources you can use to find which keywords are available for the preamble in that markup language, as well as the possible values those keywords can take.

The next characteristic of knitted documents is that they need to clearly demarcate where executable code starts and where regular formatted text starts (in other words, where the executable code section ends). To do this, knitted documents have two special combination of characters, one that can be used in the plain text to indicate where executable code starts and one to indicate where it ends. For example, Figure 3.6 shows the plain text that could be used in an RMarkdown document to write some regular text, then some executable code, and then indicate the start of more regular text:

Some text is here. And then some code:


 Some more text is here.

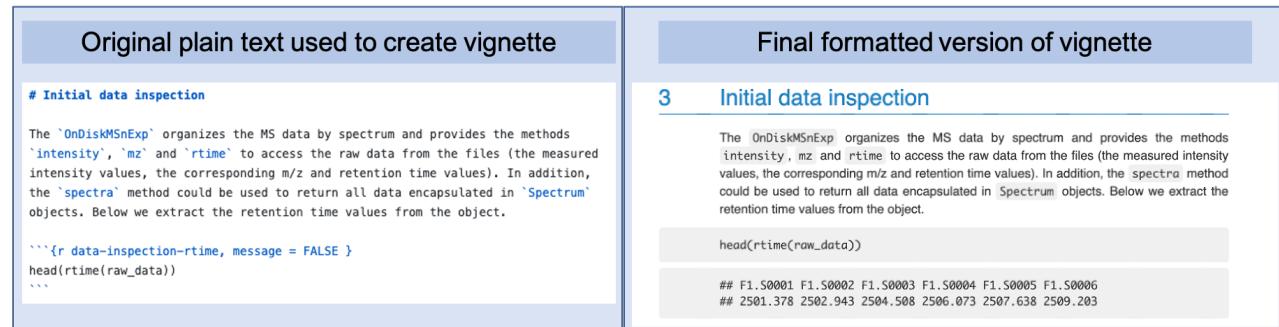
Figure 3.6: An example of how special combinations of characters are used to demarcate code in an RMarkdown file. The color formatting here is applied automatically by RStudio; all the text in this example is written in plain text.

The combination that indicates the start of executable code will vary depending on the markup language being. You may have noticed that these markers, which indicate the beginning and end of executable code, seem like very odd character combination. There is a good reason for this. By making this character combination unusual, there will be less of a chance that it shows up in regular text. This way there are fewer cases where the writer unintentionally indicate the start of a new section of executable code when trying to write regular text in the knitted document.

The next characteristic of knitted documents is that formatting for the regular text in the document—that is, everything that is not executable code—is specified using what is called a **markup language**. When you were writing in plain text, you do not have buttons to click on for formatting, for example, to specify words or phrases that should be in bold or italics, font size, headings, and so on. Instead you use special characters or character combinations to specify formatting in the final document. These character combinations are defined based on the markup language you use. As mentioned earlier, Rmarkdown uses the Markdown language; other knitted documents can be created using

LaTeX or HTML. As an example of how these special character combinations work, in Markdown, you place two asterisks around a word or phrase to make it bold. To write “**this**” in the final document, in other words, you’ll write ****“this”**** in the plain text in the initial document.

You can start to see how this works by looking at the example of the `xcms` vignette shown earlier in Figures 3.4 and 3.5. In Figure 3.7, we’ve recreated these two parts side-by-side, so they’re easier to compare.



You can look for several formatting elements here. First, the section is headed “Initial data inspection”. You can see that in the original plain text document, this is marked using a `#` to start the line with the text for the header. You can also see that words or phrases that are formatted in a computer-style font in the final document—to indicate that they are values from computer code, rather than regular English words—are surrounded by backticks in the plain text file.

The final characteristics of knitted documents is that, to create the final document, you will render the plain text document. That is the process that will create an attractive final document. To visualize this, **rendering** is the process that takes the document from the plain text format, as shown in the left of Figure 3.7, to the final format, shown in the right of that figure.

When you render the document, it will be run through two software programs, as described earlier. The first will look only for sections with executable code, based on the character combination that is used to mark these executable code sections. This first software will execute that code and take any output—including data results, figures, and tables—and insert those at the relevant spot in the document’s file. Next, the output file from this software will be run through another software program. This second program will look for all the formatting instructions and render the final document in an attractive format. This final output can be in a number of file formats, depending what you specify in the preamble, including a PDF document, an HTML file, or a Word document.

You should consider the final document, regardless of the output format, as read-only. This means that you should never make edits or changes to the

Figure 3.7: The original plain text for a knitted document and the final output, side by side. These examples are from the ‘`xcms`’ package vignette, a package available on Bioconductor. The left part of the figure shows the plain text that was written to create the output, which is shown in the left part of the figure. You can see how elements like sections headers and different font styles are indicated in the original plain text through special characters or combinations of characters, using the Markdown language syntax.

final version of the document. Instead you should make any changes to your initial plain text file. This is because the rendering process will overwrite any previous versions of the final document. Therefore any changes that you have made to your final document will be overwritten anytime you re-render from the original plain text document.

3.8 RMarkdown for creating reproducible data pre-processing protocols

The R extension package RMarkdown can be used to create documents that combine code and text in a ‘knitted’ document, and it has become a popular tool for improving the computational reproducibility and efficiency of the data analysis stage of research. This tool can also be used earlier in the research process, however, to improve reproducibility of pre-processing steps. In this module, we will provide detailed instructions on how to use RMarkdown in RStudio to create documents that combine code and text. We will show how an RMarkdown document describing a data pre-processing protocol can be used to efficiently apply the same data pre-processing steps to different sets of raw data.

Objectives. After this module, the trainee will be able to:

- Define RMarkdown and the documents it can create
- Explain how RMarkdown can be used to improve the reproducibility of research projects at the data pre-processing phase
- Create a document in RStudio using RMarkdown
- Describe more advanced features of Rmarkdown and where you can find out more about them

3.8.1 Creating knitted documents in R

In the last module (3.7), we described what knitted documents are, as well as the advantages of using knitted documents to create data pre-processing protocols for common pre-processing tasks in your research group. We also described the key elements of creating a knitted document, regardless of the software system you are using. In this module, we will go into more detail about how you can create these documents using R and RStudio, and in the next module (3.9) we will walk through an example data pre-processing protocol created using this method. We strongly recommend that you read the previous module (3.7) before working through this one.

R has a special format for creating knitted documents called **Rmarkdown**. In the previous module, we talked about the elements of a knitted document, and later in this module we’ll walk through how they apply to Rmarkdown. However, the easiest way to learn how to use Rmarkdown is to try an example, so we’ll start with a very basic one. If you’d like to try it yourself, you’ll need to download R and RStudio. The RStudio IDE can be downloaded and installed as a free software, as long as you use the personal version (RStudio

creates higher-powered versions for corporate use).

Like other plain text documents, an Rmarkdown file should be edited using a text editor, rather than a word processor like Word or Google Docs. It is easiest to use the RStudio IDE as the text editor when creating and editing an R markdown document, as this IDE has incorporated some helpful functionality for working with plain text documents for Rmarkdown. In RStudio, you can create a number of types of new files through the “File” menu. To create a new R markdown file, open RStudio and then choose “New File”, then choose “Rmarkdown” from the choices in that menu. Figure 3.8 shows an example of what this menu option looks like.

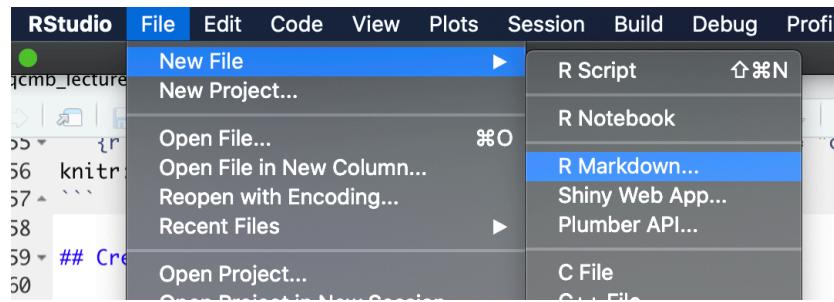


Figure 3.8: RStudio pull-down menus to help you navigate to open a new Rmarkdown file.

This will open a window with some options you can specify some of the overall information about the document (Figure 3.9), including the title and the author. You can specify the output format that you would like. Possible output formats include HTML, Word, and PDF. You should be able to use the HTML and Word output formats without any additional software, so we'll start there with this example. If you would like to use the PDF output, you will need to install one other piece of software: Miktex for Windows, MacTex for Mac, or TeX Live for Linux. These are all pieces of software with an underlying TeX engine and all are open-source and free. The example in the next module was created as a PDF using one of these tools.

Once you have selected the options in this menu you can choose the “Okay” button (Figure 3.9). This will open a new document. This document, however, won't be blank. Instead it will include an example document written in Rmarkdown (Figure 3.10). This example document helps you navigate how the Rmarkdown process works, by letting you test out a sample document. It also gives you a starting point—once you understand how the example document works, you can edit it and change it to convert it into the document you would like to create.

If you have not used Rmarkdown before, it is very helpful to try knitting this example document before making changes, to explore how pieces in the document align with elements in the rendered output document. Once you are familiar with the line-up between elements in this file in the output document, you can delete parts of the example file and insert your own text and code.

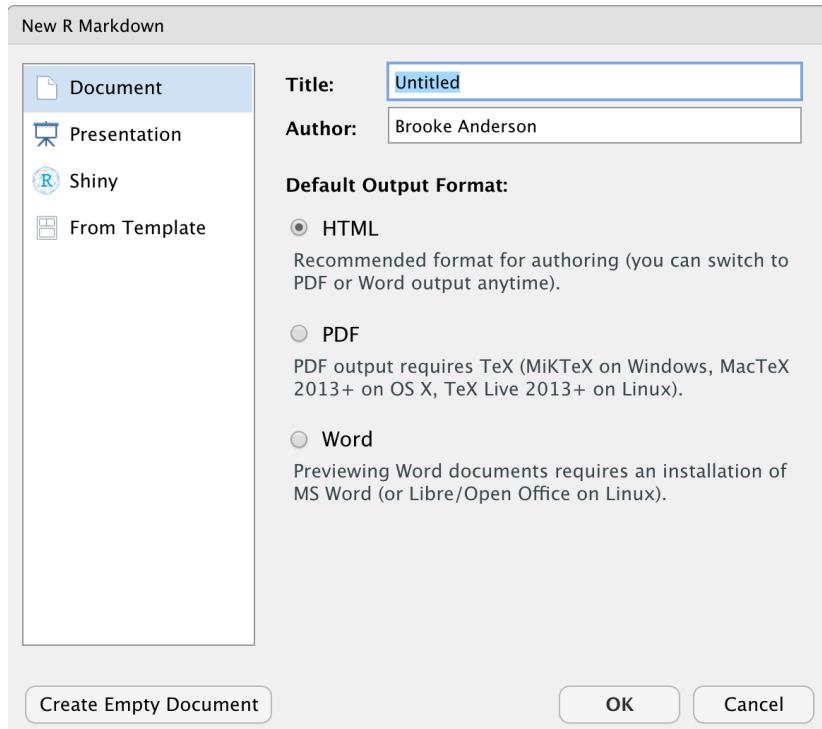


Figure 3.9: Options available when you create a new RMarkdown file in RStudio. You can specify information that will go into the document's preamble, including the title and authors and the format that the document will be output to (HTML, Word, or PDF).

```
---
title: "Untitled"
author: "Brooke Anderson"
date: "2/13/2021"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

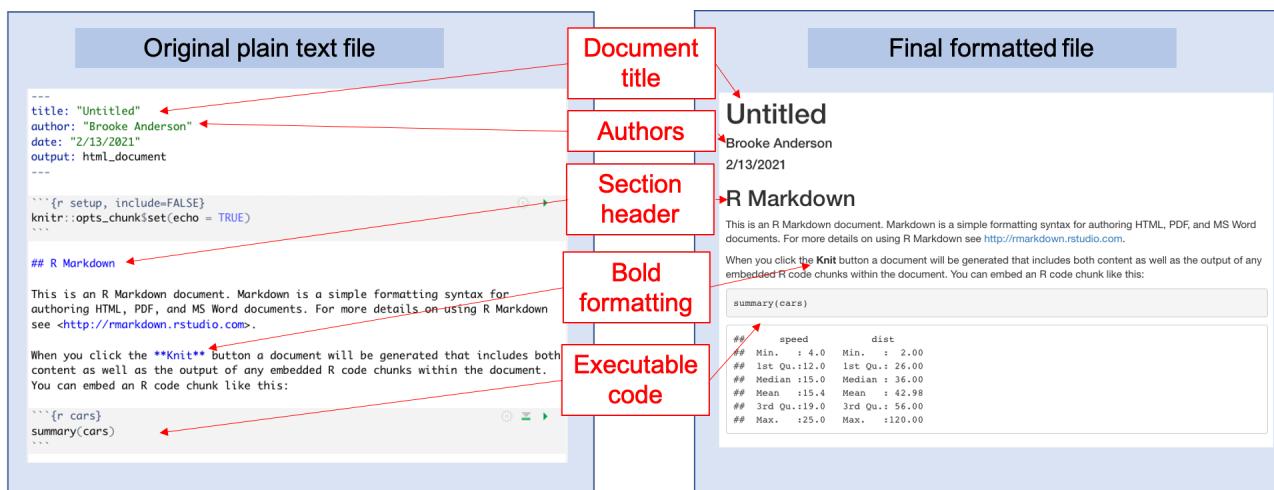
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```

```

Figure 3.10: Example of the template RMarkdown document that you will see when you create a new RMarkdown file in RStudio. You can explore this template and try rendering (knitting) it. Once you are familiar with how this example works, you can edit the text and code to adapt it for your own document.

Let's walk through and explore this example document, aligning it with the formatted output document (Figure 3.11). First, to render this or any Rmarkdown document, if you are in RStudio you can use the "Knit" button at the top of the file, as shown in Figure 3.12. When you click on this button, it will render the entire document to the output format you've selected (HTML, PDF, or Word). This rendering process will both run the executable code and apply all formatting. The final output (Figure 3.11, right) will pop up in a new window. As you start with Rmarkdown, it is useful to look at this output to see how it compares with the plain text Rmarkdown file (Figure 3.11, left).



A screenshot of the RStudio interface showing the R Markdown file 'qcmb_lecture_2021.Rmd'. The 'Knit' button in the toolbar is highlighted with a red circle and a green arrow pointing to it. The code editor shows the R Markdown code, with a red circle and a green arrow pointing to the green 'Run' button at the bottom right of the code area.

```

1  ---
2  title: "Untitled"
3  author: "Brooke Anderson"
4  date: "1/24/2021"
5  output: html_document
6  ---
7
8  ```{r setup, include=FALSE}
9  knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see
15 <http://rmarkdown.rstudio.com>.
16
17 When you click the **Knit** button a document will be generated that includes both
18 content as well as the output of any embedded R code chunks within the document. You can
19 embed an R code chunk like this:
20
21 ````{r cars}
22 summary(cars)
23 ````
```

Figure 3.11: Example of the template RMarkdown document that you will see when you create a new RMarkdown file in RStudio. You can click the 'Knit' button to highlight how the document will be generated. The 'Run' button highlighted at the top of the figure, will render the entire document. The green arrow, highlighted lower in the figure within a code chunk, can be used to run the code in that specific code chunk.

You will also notice, after you first render the document, that your work-

ing directory has a new file with this output document. For example, if you are working to create an HTML document using an Rmarkdown file called “my_report.Rmd”, once you knit your Rmarkdown file, you will notice a new file in your working directory called “my_report.html”. This new file is your output file, the one that you would share with colleagues as a report. You should consider this output document to be read only—in other words, you can read and share this document, but you should not make any changes directly to this document, since they will be overwritten anytime you re-render the original Rmarkdown document.

Next, let’s compare the example Rmarkdown document (the one that is given when you first open an Rmarkdown file in RStudio) with the output file that is created when you render this example document (Figure 3.11). If you look at the output document (Figure 3.11, right), you can notice how different elements align with pieces in the original Rmarkdown file (Figure 3.11). For example, the output document includes a header with the text “R Markdown”. This second-level header is created by the Markdown notation in the original file of:

```
## R Markdown
```

This header is formatted in a larger font than other text, and on a separate line—the exact formatting is specified within the style file for the Rmarkdown document, and will be applied to all second-level headers in the document. You can also see formatting specified through things like bold font for the word “Knit”, through the Markdown syntax ****Knit****, and a clickable link specified through the syntax <http://rmarkdown.rstudio.com>. At the beginning of the original document, you can see how elements like the title, author, date, and output format are specified in the YAML. Finally, you can see that special character combinations demarcate sections of executable code.

Let’s look a little more closely in the next part of the module at how these elements of the Rmarkdown document work.

3.8.2 *Formatting text with Markdown in Rmarkdown*

If you remember from the last module, one element of knitted documents is that they are written in plain text, with all the formatting specified using a markup language. For the main text in an Rmarkdown document, all formatting is done using Markdown as the markup language. Markdown is a popular markup language, in part because it is a good bit simpler than other markup languages like HTML or LaTeX. This simplicity means that it is not quite as expressive as other markup languages. However, Markdown probably provides adequate formatting for at least 90% of the formatting you will typically want to do for a research report or pre-processing protocol, and by staying simpler, it is much easier to learn the Markdown syntax quickly compared to other markup languages.

As with other markup languages, Markdown uses special characters or combinations of characters to indicate formatting within the plain text of the original document. When the document is rendered, these markings are used by the software to create the formatting that you have specified in the final output document. Some example formatting symbols and conventions for Markdown include:

- to format a word or phrase in bold, surround it with two asterisks (**)
- to format a word or phrase in italics, surround it with one asterisk (*)
- to create a first-level header, put the header text on its own line, starting the line with #
- to create a second-level header, put the header text on its own line, starting the line with ##
- separate paragraphs with empty lines
- use hyphens to create bulleted lists

One thing to keep in mind when using Markdown, in terms of formatting, is that white space can be very important in specifying the formatting. For example when you specify a new paragraph, you must leave a blank line from your previous text. Similarly when you use a hash (#) to indicate a header, you must leave a blank space after the hash before the word or phrase that you want to be used in that header. To create a section header, you would write:

```
# Initial Data Inspection
```

On the other hand, if you forgot the space after the hash sign, like this:

```
#Initial Data Inspection
```

then in your ouput document you would get this:

```
#Initial Data Inspection
```

Similarly, white space is needed to separate paragraphs. For example, this would create two paragraphs:

```
This is a first paragraph.
```

```
This is a second.
```

Meanwhile this would create one:

```
This is a first paragraph.
```

```
This is still part of the first paragraph.
```

The syntax of Markdown is fairly simple and can be learned quickly. For more details on this syntax, you can refer to the Rmarkdown reference guide at <https://rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf>. The basic formatting rules for Markdown are also covered in some more extensive resources for Rmarkdown that we will point you to later in this module.

3.8.3 Preambles in Rmarkdown documents

In the previous module, we explained how knitted documents include a preamble to specify some metadata about the document, including elements like the title, authors, and output format. In R, this preamble is created using YAML. In this subsection, we provide some more details on using this YAML section in Rmarkdown documents.

In an Rmarkdown document, the YAML is a special section at the top of an RMarkdown document (the original, plain text file, not the rendered version). It is set off from the rest of the document using a special combination of characters, using a process very similar to how executable code is set off from other text with a special set of characters so it can be easily identified by the software program that renders the document. For the YAML, this combination of characters is three hyphens (---) on a line by themselves to start the YAML section and then another three on a line by themselves to end it. Here is an example of what the YAML might look like at the top of an RMarkdown document:

```
---
title: "Laboratory report for example project"
author: "Brooke Anderson"
date: "1/12/2020"
output: word_document
---
```

Within the YAML itself, you can specify different options for your document. You can change simple things like the title, author, and date, but you can also change more complex things, including how the output document is rendered. For each thing that you want to specify, you specify it with a special keyword for that option and then a valid choice for that keyword. The idea is very similar to setting parameter values in a function call in R. For example, the `title:` keyword is a valid one in RMarkdown YAML. It allows you to set the words that will be printed in the title space, using title formatting, in your output document. It can take any string of characters, so you can put in any text for the title that you'd like, as long as you surround it with quotation marks. The `author:` and `date:` keywords work in similar ways. The `output:` keyword allows you to specify the output that the document should be rendered to. In this case, the keyword can only take one of a few set values, including `word_document` to output a Word document, `pdf_document` to output a pdf document (see later in this section for some more set-up required to make that work), and `html_document` to output an HTML document.

As you start using RMarkdown, you will be able to do a lot without messing with the YAML much. In fact, you can get a long way without ever changing the values in the YAML from the default values they are given when you first create an RMarkdown document. As you become more familiar with R, you may want to learn more about how the YAML works and how you

can use it to customize your document—it turns out that quite a lot can be set in the YAML to do very interesting customizations in your final rendered document. The book *R Markdown: The Definitive Guide* (Xie et al., 2018), which is available free online, has sections discussing YAML choices for both HTML and pdf output, at <https://bookdown.org/yihui/rmarkdown/html-document.html> and <https://bookdown.org/yihui/rmarkdown/pdf-document.html>, respectively. There is also a talk that Yihui Xie, the creator of RMarkdown, gave on this topic at a past RStudio conference, available at <https://rstudio.com/resources/rstudioconf-2017/customizing-extending-r-markdown/>.

3.8.4 Executable code in Rmarkdown files

In the previous module, we described how knitted documents use special markers to indicate where sections of executable code start and stop. In RMarkdown, the markers you will use to indicate executable code look like this:

```
```r{}  
my_object <- c(1, 2, 3)
~~~
```

In RMarkdown, the following combination indicates the start of executable code:

```
~~~{r}
```

while this combination indicates the end of executable code (in other words the start of regular text):

```
~~~
```

In the example above, we have shown the most basic version of the markup character combination used to specify the start of executable code (~~~{r}). This character combination can be expanded, however, to include some specifications for how you want the code in the section following it to be run, as well as how you want output to be shown. For example, you could use the following indications to specify that the code should be run, but the code itself should not be printed in the final document, by specifying echo = FALSE, as well as that the created figure should be centered on the page, by specifying fig.align = "center":

```
~~~{r echo = FALSE, fig.align = "center"}
```

There are numerous options that can be used to specify how the code will be run. These specifications are called **chunk options**, and you specify them in the special character combination where you mark the start of executable code. For example, you can specify that the code should be printed in the document, but not executed, by setting the eval parameter to FALSE with ~~{r eval = FALSE} as the marker to start the code section.

The chunk options also include echo, which can be used to specify whether to print the code in that code chunk when the document is rendered. For

some documents, it is useful to print out the code that is executed, where for other documents you may not want that printed. For example, for a pre-processing protocol, you are aiming to show yourself and others how the pre-processing was done. In this case, it is very helpful to print out all of the code, so that future researchers who read that protocol can clearly see each step. By contrast, if you are using Rmarkdown to create a report or an article that is focused on the results of your analysis, it may make more sense to instead hide the code in the final document.

As part of the code options, you can also specify whether messages and warnings created when running the code should be included in the document output, and there are number of code chunk options that specify how tables and figures rendered by the code should be shown. For more details on the possible options that can be specified for how code is evaluated within an executable chunk of code, you can refer to the Rmarkdown cheat sheet available at <https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf>

RStudio has some functionality that is useful when you are working with code in Rmarkdown documents. Within each code chuck are some buttons that can be used to test out the code in that chunk of executable code. One is the green right arrow key to the right at the top of the code chunk, highlighted in Figure 3.12. This button will run all of the code in that chunk and show you the output in an output field that will open directly below the code chunk. This functionality allows you to explore the code in your document as you build it, rather than waiting until you are ready to render the entire document. The button directly to the left of that button, which looks like an upward-pointing arrow over a rectangle, will execute all code that comes before this chunk in the document. This can be very helpful in making sure that you have set up your environment to run this particular chunk of code.

### 3.8.5 More advanced Rmarkdown functionality

The details and resources that we have covered so far focus on the basics of Rmarkdown. You can get a lot done just with these basics. However, the Rmarkdown system is very rich and allows complex functionality beyond these basics. In this subsection, we will highlight just a few of the ways Rmarkdown can be used in a more advanced way. Since this topic is so broad, we will focus on elements that we have found to be particularly useful for biomedical researchers as they become more advanced Rmarkdown users. For the most part, we will not go into extensive detail about how to use these more advanced features in this module, but instead point to resources where you can learn more as you are ready. If you are just learning Rmarkdown, at this point it will be helpful to just know that some of these advanced features are available, so you can come back and explore them when you become familiar with the basics. However, we will provide more details for one advanced element that

we find particularly useful in creating data pre-processing protocols: including bibliographical references.

#### **Including bibliographical references.**

To include references in RMarkdown documents, you can use something called **BibTeX**. This is a software system that is free and open source and works in concert with LaTeX and other markup languages. It allows you to save bibliographical information in a plain text file—following certain rules—and then reference that information in a document. In this way, it can serve the role of a bibliographical reference manager (like Endnote or Mendeley) while being free and keeping all information in plain text files, where they can easily be tracked with version control like git. By using BibTeX with RMarkdown, you can include bibliographical references in the documents that you create, and RMarkdown will handle the creation of the references section and the numbering of the documents within your text.

To use BibTeX to add references to an RMarkdown document, you'll need to take three steps:

1. Create a plain text file with listings for each of your references (**BibTeX file**). Save this file with the extension .bib. These listings need to follow a special format, which we'll describe in just a minute.
2. In your RMarkdown document, include the filepath to this BibTeX file, so that RMarkdown will be able to find the bibliographical listings.
3. In the text of the RMarkdown file, include a key and special character combination anytime you want to reference a paper. This referencing also follows a special format, which we'll describe below.

Let's look at each of these steps in a bit more detail. The first step is to create a plain text file with a listing for each of the documents that you'd like to cite. The plain text document should be saved with the file extension .bib (for example, "mybibliography.bib"), and the listings for each document in the file must follow specific rules.

Let's take a look at one to explore these rules. Here's an example of a BibTeX listing for a scientific article:

```
@article{fox2020,
 title={Cyto-feature engineering: A pipeline for flow cytometry
 analysis to uncover immune populations and associations with
 disease},
 author={Fox, Amy and Dutt, Taru S and Karger, Burton and Rojas,
 Mauricio and Obreg\'on-Henao, Andr\'eis and
 Anderson, G Brooke and Henao-Tamayo, Marcela},
 journal={Scientific Reports},
 volume={10},
 number={1},
 pages={1--12},
```

```
year=[2020]
}
```

You can see that this listing is for an article, because it starts with the keyword @article. BibTeX can record a number of different types of documents, including articles, books, and websites. You start by specifying the document type because different types of documents need to include different elements in their listings. For example, a website should include the date when it was last accessed, while an article typically will not.

Within the curly brackets for the listing shown above, there are key-value pairs—elements where the type of value is given with a keyword (e.g., title), and then the value for that element is given after an equals sign. For example, to specify the journal in which the article was published, this listing has journal={Scientific Reports}. Finally, the listing has a key that you will use to identify the listing in the main text. In this case, the listing is given the key fox2020, which combines the first author and publication year. You can use any keys you like for the items in the bibliography, as long as they are different for every listing, so that the computer can identify which bibliographical listing you are referring to when you use a key.

This format may seem overwhelming, but fortunately you will rarely have to create these listings by hand. Instead, you can get them directly from Google Scholar. To do this, look up the paper on Google Scholar (Figure 3.13). When you see it, look for a small quotation mark symbol at the bottom of the article listing (shown with the top red arrow in Figure 3.13). If you click on this, it will open a pop-up with the citation for the article. At the bottom of that pop-up is a link that says “BibTeX” (bottom red arrow in Figure 3.13). If you click on that, it will take you to a page that gives the full BibTeX listing for that article, and you can just copy and paste this into your plain text BibTeX file.

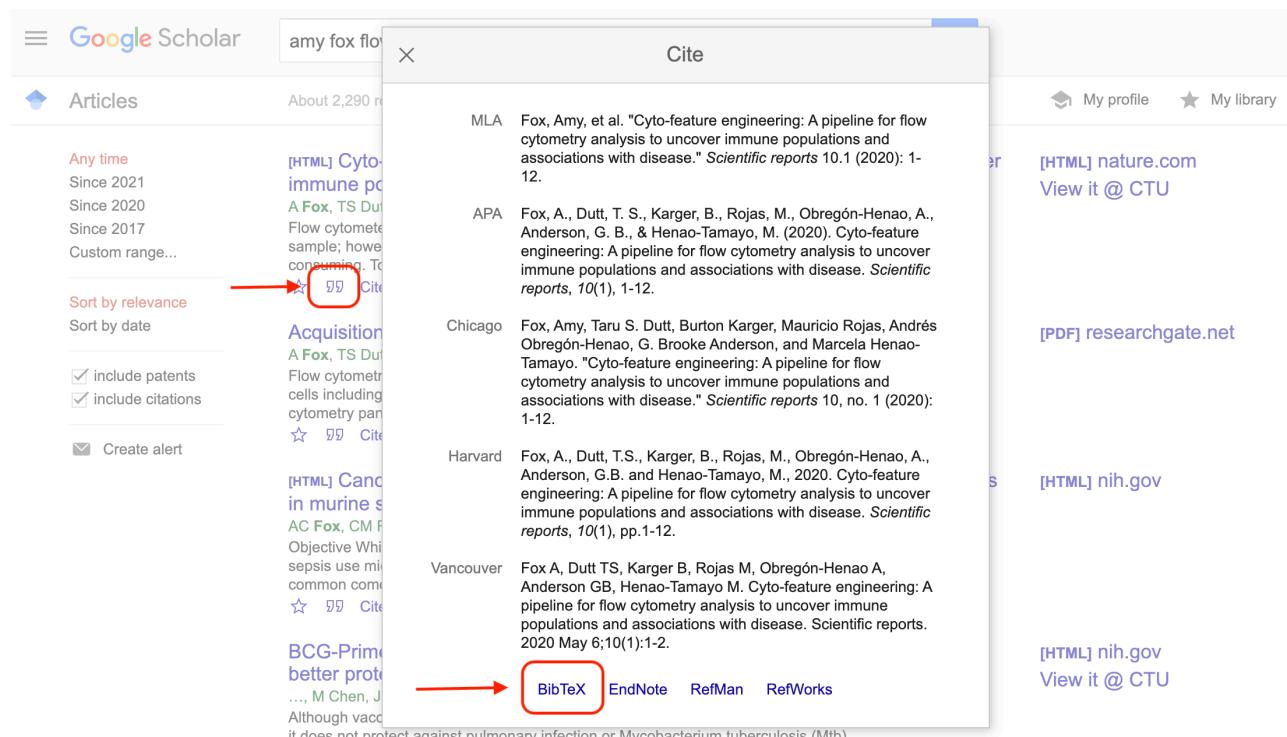
Once you have this plain text BibTeX file, you will tell your computer how to find it by including its path in the YAML. For example, if you created a BibTeX file called “mybibliography.tex” and saved it in the same directory as a RMarkdown document, you could use the following to indicate this file for the RMarkdown document:

```

title: "Reproducible Research with R"
author: "Brooke Anderson"
date: "1/25/2021"
output: beamer_presentation
bibliography: mybibliography.bib

```

This shows the YAML for the document—the part that goes at the beginning of the RMarkdown document and gives some metadata and overall instructions for the document. In this example, we’ve added an extra line: bibliography:



`mybibliography.bib`. This says that you'd like to link to a BibTeX file when this document is rendered, as well as where to find that file (the file named “`mybibliography.bib`” in the directory of the RMarkdown file).

Now that you have created the BibTeX file and told the RMarkdown file where to find it, you can connect the two. As you write in the RMarkdown file, you can refer to any of your BibTeX listings by using the key that you set for that document. For example, if you wanted to reference the Fox et al. paper we used in the example listing above, you would use the key that we set for that listing, `fox2020`. You will follow a special convention when you reference this key: you'll use the @ symbol directly followed by that key. Typically, you will surround this with square brackets. Therefore, to reference the Fox et al. paper, you'd use `[@fox2021]`.

Here's how that might look in practice. If you write in the RMarkdown document:

This technique follows earlier work `[@fox2020]`.

In the output from rendering that RMarkdown document you'd get:

“This technique follows earlier work (Fox et al. 2020).”

The full paper details will then be included at the end of the document, in a reference section.

Figure 3.13: Example of using Google Scholar to get bibliographical information for a BibTeX file. When you look up an article on Google Scholar, there is an option (the quotation mark icon under the article listing) to open a pop-up window with bibliographical information. At the bottom of this pop-up box, you can click on ‘BibTeX’ to get a plain text version of the BibTeX entry for the article. You can copy and paste this into your BibTeX file.

### Other advanced Rmarkdown functionality

There are a number of other advanced things that you can do with Rmarkdown, once you have mastered the basics. First, you can use Rmarkdown to build different types of documents, not just reports in Word, PDF, or HTML. For example, you can use the bookdown package to create entire online and print books using the Rmarkdown framework. This book of modules was created using this system. You can also create websites and web dashboards, using the blogdown and flexdashboard packages, respectively. The blogdown package allows you to create professionally-styled websites, including blog sections where you can include R code and results. Figure 3.14 gives an example of a website created using blogdown—you can see the full website here if you'd like to check out some of the features that this framework provides. The flexdashboard package lets you create “dashboards” with data, similar to the dashboards that many public health departments using during the COVID-19 pandemic to share case numbers in specific counties and states.



With Rmarkdown, you can also create reports that are more customized than the default style that we explored above. First, you can create templates that add customized styling to the document. In fact, many journals have created journal-specific templates that you can use in Rmarkdown. With these templates, you can write up your research results in a reproducible way, using Rmarkdown, and submit the resulting document directly to the journal, in the correct format. An example of the first page of an article created in Rmarkdown using one of these article templates is shown in Figure 3.8.5. The `rticles` package in R provides these templates for several different journal families.

Figure 3.14: Example of a website created using blogdown, leveraging the Rmarkdown framework.

## Ten simple rules for finding and selecting R packages

Caroline J. Wendt<sup>1</sup> · <sup>2</sup>, G. Brooke Anderson<sup>3</sup> \*

<sup>1</sup> Department of Statistics, Colorado State University, Fort Collins, Colorado, United States of America  
<sup>2</sup> Department of Mathematics, Colorado State University, Fort Collins, Colorado, United States of America  
<sup>3</sup> Department of Environmental & Radiological Health Sciences, Colorado State University, Fort Collins, Colorado, United States of America

\* Corresponding author: Brooke.Anderson@colostate.edu

### Abstract

R is an increasingly preferred software environment for data analytics and statistical computing among scientists and practitioners. Packages markedly extend R's utility and ameliorate inefficient solutions to data science problems. We outline ten simple rules for finding relevant packages and determining which package is best for your desired use. We begin in Rule 1 with tips on how to consider your purpose, which will guide your search to follow, where, in Rule 2, you'll learn best practices for finding and collecting options. Rules 3 and 4 will help you navigate packages' profiles and explore the extent of their online resources, so that you can be confident in the quality of the package you choose, and assured that you'll be able to access support. In Rules 5 and 6, you'll become familiar with how the R Community evaluates packages, and learn how to assess the popularity and utility of packages for yourself. Rules 7 and 8 will teach you how to investigate and track package development processes, so you can further evaluate their merit. We end in Rules 9 and 10 with more hands-on approaches, which involve digging into package code.

### Disclaimer

GBA is a volunteer associate editor at ROpenSci and is an instructor through the Coursera platform, both of which are mentioned as potential resources in this article. The views described here reflect the authors' own views without input from any third party organization.

### Funding acknowledgment

This research was supported in part by the National Institute of General Medical Sciences through R25GM132797 (GBA). CJW also received support from the Honors Undergraduate Program at Colorado State University. The funders had no role in the manuscript preparation or the decision to publish.

February 18, 2021

1 [27]

```
\begin{figure}
\caption[Example of a manuscript written in Rmarkdown using a template]{Example of a manuscript written in Rmarkdown using a template. This figure shows the first page of an article written for submission to PLoS Computation Biology, written in Rmarkdown while using the PLoS template from the \texttt{rticles} package. The full article, including the Rmarkdown input and final pdf, are available on GitHub at \url{https://github.com/cjwendt/plos_ten.} \end{figure}
```

Rmarkdown also has some features that make it easy to run code that is computationally expensive or code that is written in another programming language. If code takes a long time to run, there are options in Rmarkdown to **cache** the results—that is, run the code once when you render the document, and then only re-run it in later renderings if the inputs have changed. Rmarkdown does this through by saving intermediate results, as well as using a system to remember which pieces of code depend on which earlier code. With very computationally expensive code, it can be a big time saver, although it can also use more storage, since it is saving more results. To include code in lan-

guages other than R, you can change something called the **engine** of the code chunk. Essentially, this is the language that your computer will use to run the code in that chunk. You can change the engine so that certain chunks of code are run using Python, Julia, and other languages by specifying the engine you'd like to use in the marker in the document that indicates the start of a piece of executable code. Earlier in this module, we showed you that executable code is normally introduced in Rmarkdown with `~~{r}`. The `r` in this string is specifying that the R engine should be used to run the code.

Finally, Rmarkdown allows you to create very customized formatting, as you move into more advanced ways to use the framework. As mentioned earlier, Markdown is a fairly simple markup language. Occasionally, this simplicity means that you might not be able to create fancier formatting that you might desire. There is a method that allows you to work around this constraint in RMarkdown.

In Rmarkdown documents, when you need more complex formatting, you can shift into a more complex markup language for part of the document. Markup languages like LaTeX and HTML are much more expressive than Markdown, with many more formatting choices possible. However, there is a downside—when you include formatting specified in these more complex markup languages, you will limit the output formats that you can render the document to. For example, if you include LaTeX formatting within an RMarkdown document, you must output the document to PDF, while if you include HTML, you must output to an HTML file. Conversely, if you stick with the simpler formatting available through the Markdown syntax, you can easily switch the output format for your document among several choices.

One area of customization that is particularly useful and simple to implement is with customized tables. The Markdown syntax can create very simple tables, but does not allow the creation of more complex tables. There is an R package called `kableExtra` that allows you to create very attractive and complex tables in RMarkdown documents. This package leverages more of the power of underlying markup languages, rather than the simpler Markdown language. The `kableExtra` package is extensively documented through two vignettes that come with the package, one if the output will be in pdf ([https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome\\_table\\_in\\_pdf.pdf](https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_pdf.pdf)) and one if it will be in HTML ([https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome\\_table\\_in\\_html.html](https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html)).

### 3.8.6 Learning more about Rmarkdown.

To learn more about RMarkdown, you can explore a number of excellent resources. The most comprehensive are shared by RStudio, where RMarkdown's developer and maintainer, Yihui Xie, works. These resources are all freely available online, and some are also available to buy as print books, if you prefer that

format.

First, you should check out the online tutorials that are provided by RStudio on RMarkdown. These are available at RStudio's RMarkdown page: <https://rmarkdown.rstudio.com/>. The page's "Getting Started" section (<https://rmarkdown.rstudio.com/lesson-1.html>) provides a nice introduction you can work through to try out RMarkdown and practice the overview provided in the last subsection of this module. The "Articles" section (<https://rmarkdown.rstudio.com/articles.html>) provides a number of other documents to help you learn RMarkdown. RStudio's RMarkdown page also includes a "Gallery" (<https://rmarkdown.rstudio.com/gallery.html>). This resource allows you to browse through example documents, so you can get a visual idea of what you might want to create and then access the example code for a similar document. This is a great resource for exploring the variety of documents that you can create using RMarkdown.

To go more deeply into RMarkdown, there are two online books from some of the same team that are available online. The first is *R Markdown: The Definitive Guide* by Yihui Xie, J. J. Allaire, and Garrett Grolemund (Xie et al., 2018). This book is available free online at <https://bookdown.org/yihui/rmarkdown/>. It moves from basics through very advanced functionality that you can implement with RMarkdown, including several of the topics we highlight later in this subsection.

The second online book to explore from this team is *R Markdown Cookbook*, by Yihui Xie, Christophe Dervieux, and Emily Riederer (Xie et al., 2020). This book is available free online at <https://bookdown.org/yihui/rmarkdown-cookbook/>. This book is a helpful resource for dipping in to a specific section when you want to learn how to achieve a specific task. Just like a regular cookbook has recipes that you can explore and use one at a time, this book does not require a comprehensive end-to-end read, but instead provides "recipes" with advice and instructions for doing specific things. For example, if you want to figure out how to align a figure that you create in the center of the page, rather than the left, you can find a "recipe" in this book to do that.

### 3.9 Example: Creating a reproducible data pre-processing protocol

We will walk through an example of creating a reproducible data pre-processing protocol. As an example, we will look at how to pre-process and analyze data that are collected in the laboratory to estimate bacterial load in samples. These data come from plating samples from an immunological experiment at serial dilutions, using data from an experiment lead by one of the coauthors. This data pre-processing protocol was created using RMarkdown and allows the efficient, transparent, and reproducible pre-processing of plating data for all experiments in the research group. We will go through how RMarkdown techniques can be applied to develop this type of data pre-processing protocol for a laboratory research group.

**Objectives.** After this module, you should be able to:

- Explain how a reproducible data pre-processing protocol can be developed for a real research project
- Understand how to design and implement a data pre-processing protocol to replace manual or point-and-click data pre-processing tools
- Apply techniques in RMarkdown to develop your own reproducible data pre-processing protocols

### 3.9.1 *Introduction and example data*

In this module, we'll provide advice and an example of how you can use the tools for knitted documents to create a reproducible data preprocessing protocol. This module builds on ideas and techniques that were introduced in the last two modules (3.7 and 3.8), to help you put them into practical use for data preprocessing that you do repeatedly for research data in your laboratory.

In this module, we will use an example of a common pre-processing task in immunological research: estimating the bacterial load in samples by plating at different dilutions. For this type of experiment, the laboratory researcher plates each of the samples at several dilutions, identifies a good dilution for counting colony-forming units (CFUs), and then back-calculates the estimated bacterial load in the original sample based on the colonies counted at this “good” dilution. This experimental technique dates back to the late 1800s, with Robert Koch, and continues to be widely used in microbiology research and applications today (Ben-David and Davidson, 2014). These data are originally from an experiment in one of our authors’ laboratory and are also available as example data for an R package called `bactcountr`, currently under development at <https://github.com/aef1004/bactcountr/tree/master/data>.

These data are representative of data often collected in immunological research. For example, you may be testing out some drugs against an infectious bacteria and want to know how successful different drugs are in limiting bacterial load. You run an experiment and have samples from animals treated with different drugs or under control and would then want to know how much viable (i.e., replicating) bacteria are in each of your samples.

You can find out by plating the sample at different dilutions and counting the colony-forming units (CFUs) that are cultured on each plate. You put a sample on a plate with a medium they can grow on and then give them time to grow. The idea is that individual bacteria from the original sample end up randomly around the surface of the plate, and any that are viable (able to reproduce) will form a new colony that, after a while, you’ll be able to see.

To get a good estimate of bacterial load from this process, you need to count CFUs on a “countable” plate—one with a “just right” dilution (and you typically won’t know which dilution this is for a sample until after plating). If you have too high of a dilution (i.e., one with very few viable bacteria), randomness will play a big role in the CFU count, and you’ll estimate the original bacterial

load with more variability. If you have too low of a dilution (i.e., one with lots of viable bacteria), it will be difficult to identify separate colonies, and they may compete for resources. To translate from diluted concentration to original concentration, you can then do a back-calculation, incorporating both the number of colonies counted at that dilution and how dilute the sample was. There is therefore some pre-processing required (although it is fairly simple) to prepare the data collected to get an estimate of bacterial load in the original sample. This estimate of bacterial load can then be used in statistical testing and combined with other experimental data to explore questions like whether a candidate vaccine reduces bacterial load when a research animal is challenged with a pathogen.

We will use this example of a common data pre-processing task to show how to create a reproducible pre-processing protocol in this module. If you would like, you can access all the components of the example pre-processing protocol and follow along, re-rendering it yourself on your own computer. The example data are available as a csv file, downloadable [here](#). You can open this file using spreadsheet software, or look at it directly in RStudio. The final pre-processing protocol for these data can also be downloaded, including both the original RMarkdown file and the output PDF document. Throughout this module, we will walk through elements of this document, to provide an example as we explain the process of developing data pre-processing modules for common tasks in your research group. We recommend that you go ahead and read through the output PDF document, to get an idea for the example protocol that we're creating.

This example is intentionally simple, to allow a basic introduction to the process using pre-processing tasks that are familiar to many laboratory-based scientists and easy to explain to anyone who has not used plating in experimental work. However, the same general process can also be used to create pre-processing protocols for data that are much larger or more complex or for pre-processing pipelines that are much more involved. For example, this process could be used to create data pre-processing protocols for automated gating of flow cytometry data or for pre-processing data collected through single cell RNA sequencing.

### 3.9.2 *Advice on designing a pre-processing protocol*

Before you write your protocol in a knitted document, you should decide on the content to include in the protocol. This section provides tips on this design process. In this section, we'll describe some key steps in designing a data pre-processing protocol:

1. Defining input and output data for the protocol;
2. Setting up a project directory for the protocol;
3. Outlining key tasks in pre-processing the input data; and
4. Adding code for pre-processing.

We will illustrate these design steps using the example protocol on pre-processing plating data.

**Defining input and output data for the protocol.**

The first step in designing the data pre-processing protocol is to decide on the starting point for the protocol (the data input) and the ending point (the data output). It may make sense to design a separate protocol for each major type of data that you collect in your research laboratory. Your input data for the protocol, under this design, might be the data that is output from a specific type of equipment (e.g., flow cytometer) or from a certain type of sample or measurement (e.g., metabolomics run on a mass spectrometer), even if it is a fairly simple type of data (e.g., CFUs from plating data, as used in the example protocol for this module). For example, say you are working with three types of data for a research experiment: data from a flow cytometer, metabolomics data measured with a mass spectrometer, and bacterial load data measured by plating data and counting colony forming units (CFUs). In this case, you may want to create three pre-processing protocols: one for the flow data, one for the metabolomics data, and one for the CFU data. These protocols are modular and can be re-used with other experiments that use any of these three types of data.

With an example dataset, you can begin to create a pre-processing protocol before you collect any of your own research data for a new experiment. If the format of the initial data is similar to the format you anticipate for your data, you can create the code and explanations for key steps in your pre-processing for that type of data. Often, you will be able to adapt the RMarkdown document to change it from inputting the example data to inputting your own experimental data with minimal complications, once your data comes in. By thinking through and researching data pre-processing options before the data is collected, you can save time in analyzing and presenting your project results once you've completed the experimental data collection for the project. Further, with an example dataset, you can get a good approximation of the format in which you will output data from the pre-processing steps. This will allow you to begin planning the analysis and visualization that you will use to combine the different types of data from your experiment and use it to investigate important research hypotheses. Again, if data follow standardized formats across steps in your process, it will often be easy to adapt the code in the protocol to input the new dataset that you created, without major changes to the code developed with the example dataset.

While pre-processing protocols for some types of data might be very complex, others might be fairly simple. However, it is still worthwhile to develop a protocol even for simple pre-processing tasks, as it allows you to pass along some of the details of pre-processing the data that might have become "common sense" to longer-tenured members of your research group. For example, the pre-processing tasks in the example protocol are fairly simple. This protocol inputs data collected in a plain-text delimited file (a csv file, in the example).

Within the protocol, there are steps to convert initial measurements from plating at different dilutions into an estimate of the bacterial load in each sample. There are also sections in the protocol for exploratory data analysis, to allow for quality assessment and control of the collected data as part of the pre-processing. The output of the protocol is a simple data object (a dataframe, in this example) with the bacterial load for each original sample. These data are now ready to be used in tables and figures in the research report or manuscript, as well as to explore associations with the experimental design details (e.g., comparing bacterial load in treated versus untreated animals) or merged with other types of experimental data (e.g., comparing immune cell populations, as measured with flow cytometry data, with bacterial loads, as measured from plating and counting CFUs).

Once you have identified the input data type to use for the protocol, you should identify an example dataset from your laboratory that you can use to create the protocol. This could be a dataset that you currently need to pre-process, in which case the development of the protocol will serve a second purpose, allowing you to complete this task at the same time. However, you may not have a new set of data of this type that you currently need to pre-process, and in this case you can build your protocol using a dataset from a previous experiment in your laboratory. In this case, you may already have a record of the steps that you used to pre-process the data previously, and these can be helpful as a starting point as you draft the more thorough pre-processing protocol. You may want to select an example dataset that you have already published or are getting ready to publish, so you won't feel awkward about making the data available for people to practice with. If you don't have an example dataset from your own laboratory, you can explore example datasets that are already available, either as data included with existing R packages or through open repositories, including those hosted through national research institutions like the NIH. In this case, be sure to cite the source of the data and include any available information about the equipment that was used to collect it, including equipment settings used when the data were collected.

For the example protocol for this module, we want to pre-process data that were collected “by hand” by counting CFUs on plates in the laboratory. These counts were recorded in a plain text delimited file (a csv file) using spreadsheet software. The spreadsheet was set up to ensure the data can easily be converted to a “tidy” format, as described in module 2.3. The first few rows of the input data look like this:

```
A tibble: 6 x 6
group replicate dilution_0 dilution_1 dilution_2 dilution_3
<dbl> <chr> <chr> <chr> <dbl> <dbl>
1 2 2-A 26 10 0 0
2 2 2-B TNTC 52 10 5
3 2 2-C 0 0 0 0
```

```
4 3 3-A 0 0 0 0
5 3 3-B TNTC TNTC 30 10
6 3 3-C 0 0 0 0
```

Each row represents the number of bacterial colonies counted after plating a certain sample, where each sample represents one experimental animal and several experimental animals (replicates) were considered for each experimental group. Columns are included with values for the experimental group of the sample (group), the specific ID of the sample within that experimental group (replicate, e.g., 2-A is mouse A in experimental group 2), and the colony-forming units (CFUs) counted at each of several dilutions. If a cell has the value “TNTC”, this indicates that CFUs were too numerous to count for that sample at that dilution.

When you have identified the input data type you will use for the protocol, as well as selected an example dataset of this type to use to create the protocol, you can include a section in the protocol that describes these input data, what file format they are in, and how they can be read into R for pre-processing (Figure 3.15).

**Data input in final pdf output of the protocol**

**Reading data into R**

The data are stored in a comma-separated plain text file called "cfu\_data.csv." They can be read into R using the following code:

```
library(tidyverse)
cfu_data <- read_csv("cfu_data.csv")
head(cfu_data)
```

## # A tibble: 6 x 6  
## group replicate dilution\_0 dilution\_1 dilution\_2 dilution\_3  
## <dbl> <chr> <chr> <dbl> <dbl>  
## 1 2 2-A 26 10 0 0  
## 2 2 2-B TNTC 52 10 5  
## 3 2 2-C 0 0 0 0  
## 4 3 3-A 0 0 0 0  
## 5 3 3-B TNTC 30 10 0  
## 6 3 3-C 0 0 0 0

Once you run this command, the data will be available in your R session in the object `cfu_data`. You can see the first few rows by running:

```
head(cfu_data)
```

## # A tibble: 6 x 6  
## group replicate dilution\_0 dilution\_1 dilution\_2 dilution\_3  
## <dbl> <chr> <chr> <dbl> <dbl>  
## 1 2 2-A 26 10 0 0  
## 2 2 2-B TNTC 52 10 5  
## 3 2 2-C 0 0 0 0  
## 4 3 3-A 0 0 0 0  
## 5 3 3-B TNTC 30 10 0  
## 6 3 3-C 0 0 0 0

**Associated inputs in the RMarkdown file**

# Reading data into R

The data are stored in a comma-separated plain text file called "cfu\_data.csv". They can be read into R using the following code:

```
```{r}
library(tidyverse)
cfu_data <- read_csv("cfu_data.csv")
head(cfu_data)
````
```

Once you run this command, the data will be available in your R session in the object "cfu\_data". You can see the first few rows by running:

```
```{r}
head(cfu_data)
````
```

For the data output, it often makes sense to plan for data in a format that is appropriate for data analysis and for merging with other types of data collected from the experiment. The aim of pre-processing is to get the data from the format in which they were collected into a format that is meaningful for combining with other types of data from the experiment and using in statistical hypothesis testing.

In the example pre-processing protocol, we ultimately output a simple dataset, with one row for each of the original samples. The first few rows of this output data are:

Figure 3.15: Providing details on input data in the pre-processing protocol. Once you have an example data file for the type of data that will be input for the protocol, you can add a section that provides the code to read the data into R. You can also add code that will show the first few rows of the example dataset, as well as a description of the data. This figure shows examples of how these elements can be added to an RMarkdown file for a pre-processing protocol, and the associated elements in the final pdf of the protocol, using the example protocol for this module.

```
A tibble: 6 x 3
group replicate cfu_in_organ
<dbl> <chr> <dbl>
1 2 2-A 260
2 2 2-B 2500
3 2 2-C 0
4 3 3-A 0
5 3 3-B 7500
6 3 3-C 0
```

For each original sample, an estimate of the CFUs of *Mycobacterium tuberculosis* in the full spleen is given (`cfu_in_organ`). These data can now be merged with other data collected about each animal in the experiment. For example, they could be joined with data that provide measures of the immune cell populations for each animal, to explore if certain immune cells are associated with bacterial load. They could also be joined with experimental information and then used in hypothesis testing. For example, these data could be merged with a table that describes which groups were controls versus which used a certain vaccine, and then a test could be conducted exploring evidence that bacterial loads in animals given a vaccine were lower than in control animals.

#### **Setting up a project directory for the protocol**

Once you have decided on the input and output data formats, you will next want to set up a file directory for storing all the inputs needed in the protocol. You can include the project files for the protocol in an RStudio Project (see module 2.6) and post this either publicly or privately on GitHub (see modules 2.9–2.11). This creates a “packet” of everything that a reader needs to use to recreate what you did—they can download the whole GitHub repository and will have a nice project directory on their computer with everything they need to try out the protocol.

Part of the design of the protocol involves deciding on the files that should be included in this project directory. Figure 3.16 provides an example of the initial files included in the project directory for the example protocol for this module. The left side of the figure shows the files that are initially included, while the right side shows the files in the project after the code in the protocol is run.

Generally, in the project directory you should include a file with the input example data, in whatever file format you will usually collect this type of data. You will also include an RMarkdown file where the protocol is written. If you are planning to cite articles and other references, you can include a BibTeX file, with the bibliographical information for each source you plan to cite (see module 3.8). Finally, if you would like to include photographs or graphics, you can include these image files in the project directory. Often, you might want to group these together in a subdirectory of the project named something like “figures”.

Once you run the RMarkdown file for the protocol, you will generate additional files in the project. Two typical files you will generate will be the output file for the protocol (in the example, this is output to a pdf file). Usually, the code in the protocol will also result in output data, which is pre-processed through the protocol code and written into a file to be used in further analysis.

| Initial files in project directory                                                                                     |         | Final files in project directory                                                                                              |         |
|------------------------------------------------------------------------------------------------------------------------|---------|-------------------------------------------------------------------------------------------------------------------------------|---------|
|  <a href="#">cfu_data.csv</a>         | 402 B   |  <a href="#">cfu_data.csv</a>                | 402 B   |
|  <a href="#">example_bib.bib</a>      | 2.7 KB  |  <a href="#">example_bib.bib</a>             | 2.7 KB  |
|  <a href="#">example_protocol.Rmd</a> | 31.9 KB |  <a href="#">example_protocol.pdf</a>        | 4 MB    |
|  <a href="#">figures</a>              |         |  <a href="#">figures</a>                     | 31.9 KB |
|                                                                                                                        |         |  <a href="#">processed_cfu_estimates.csv</a> | 231 B   |

Figure 3.16: Example of files in the project directory for a data pre-processing protocol. On the left are the files initially included in the project directory for the example protocol for this module. These include a file with the input data (`cfu_data.csv`), a BibTeX file with bibliographical information for references (`example_bib.bib`), the RMarkdown file for the protocol (`example_protocol.Rmd`), and a subdirectory with figures to include in the protocol (`figures`). On the right is shown the directory after the code in the protocol RMarkdown document is run, which creates an output pdf with the protocol (`example_protocol.pdf`) as well as the output data (`processed_cfu_estimates.csv`).

## **Outlining key tasks in pre-processing the input data.**

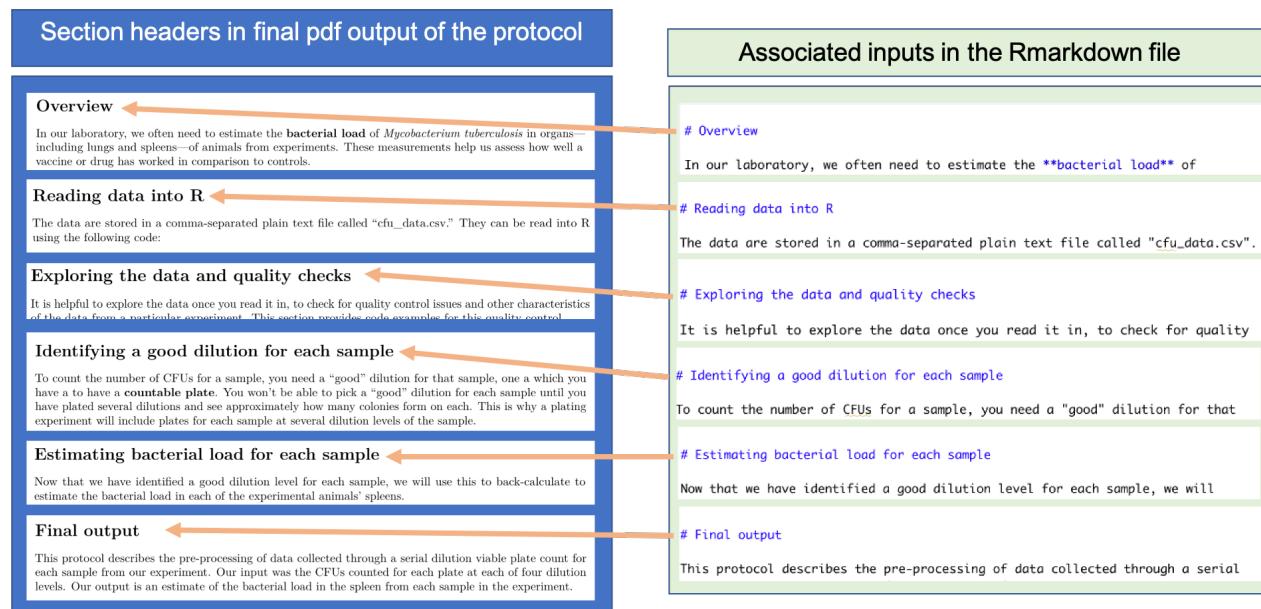
The next step is to outline the key tasks that are involved in moving from the data input to the desired data output. For the plating data we are using for our example, the key tasks to be included in the pre-processing protocol are:

1. Read the data into R
  2. Explore the data and perform some quality checks
  3. Identify a “good” dilution for each sample—one at which you have a countable plate
  4. Estimate the bacterial load in each original sample based on the CFUs counted at that dilution
  5. Output data with the estimated bacterial load for each sample

Once you have this basic design, you can set up the RMarkdown file for the pre-processing protocol to include a separate section for each task, as well as an “Overview” section at the beginning to describe the overall protocol, the data being pre-processed, and the laboratory procedures used to collect those data. In RMarkdown, you can create first-level section headers by putting the text for the header on its own line and beginning that line with #, followed by a space. You should include a blank line before and after the line with this header text. Figure 3.17 shows how this is done in the example protocol for this module, showing how text in the plain text RMarkdown file for the protocol align with section headers in the final pdf output of the protocol.

### **Adding code for pre-processing.**

For many of these steps, you likely have code—or can start drafting the code—required for that step. In RMarkdown, you can test this code as you write it. You insert each piece of executable code within a special section, separated from the regular text with special characters, as described in previous modules.



For any pre-processing steps that are straightforward (e.g., calculating the dilution factor in the example module, which requires only simple mathematical operations), you can directly write in the code required for the step. For other pre-processing steps, however, the algorithm may be a bit more complex. For example, complex algorithms have been developed for steps like peak identification and alignment that are required when pre-processing data from a mass spectrometer.

For these more complex tasks, you can start to explore available R packages for performing the task. There are thousands of packages available that extend the basic functionality of R, providing code implementations of algorithms in a variety of scientific fields. Many of the R packages relevant for biological data—especially high-throughput biological data—are available through a repository called Bioconductor. These packages are all open-source (so you can explore their code if you want to) and free. You can use vignettes and package manuals for Bioconductor packages to identify the different functions you can use for your pre-processing steps. Once you have identified a function for the task, you can use the helpfile for the function to see how to use it. This help documentation will allow you to determine all of the function's parameters and the choices you can select for each.

You can add each piece of code in the RMarkdown version of the protocol using the standard method for RMarkdown (module 3.8). Figure 3.18 shows an example from the example protocol for this module. Here, we are using code to help identify a “good” dilution for counting CFUs for each sample. The code is included in an executable code chunk, and so it will be run each time the protocol is rendered. Code comments are included in the code to provide

Figure 3.17: Dividing an RMarkdown data pre-processing protocol into sections. This shows an example of creating section headers in a data pre-processing protocol created with RMarkdown, showing section headers in the example pre-protocol for this module.

finer-level details about what the code is doing.

### Pre-processing code in final pdf output of the protocol

```
Here is the code we used to identify the best dilution level for each sample, based on these criteria:
cfu_data <- cfu_data %>%
 # For each dilution, calculate how far the CFUs counted on the plate are
 # from 25.
 mutate(diff_from_25 = abs(CFU - 25)) %>%
 # For each original sample (ID-ed by the 'replicate' column), determine
 # first if CFUs are 0 at all dilutions and second which dilution
 # had a CFU count closest to 25. Finally, include a check to see if there
 # are non-zero ties for the sample, in terms of plates with non-zero CFU counts
 # equally close to 25.
 group_by(replicate) %>%
 mutate(all_zeros = all(CFU == 0),
 closest_to_25 = dilution[which.min(diff_from_25)],
 ties_closest_to_25 = sum(diff_from_25 == min(diff_from_25)) > 1)
```

### Associated inputs in the Rmarkdown file

```
Here is the code we used to identify the best dilution level for each sample,
based on these criteria:
```{r}
cfu_data <- cfu_data %>%
  # For each dilution, calculate how far the CFUs counted on the plate are
  # from 25.
  mutate(diff_from_25 = abs(CFU - 25)) %>%
  # For each original sample (ID-ed by the 'replicate' column), determine
  # first if CFUs are 0 at all dilutions and second which dilution
  # had a CFU count closest to 25. Finally, include a check to see if there
  # are non-zero ties for the sample, in terms of plates with non-zero CFU counts
  # equally close to 25.
  group_by(replicate) %>%
  mutate(all_zeros = all(CFU == 0),
        closest_to_25 = dilution[which.min(diff_from_25)],
        ties_closest_to_25 = sum(diff_from_25 == min(diff_from_25)) > 1)
...```

```

For each step of the protocol, you can also include potential problems that might come up in specific instances of the data you get from future experiments. This can help you adapt the code in the protocol in thoughtful ways as you apply it in the future to new data collected for new studies and projects.

3.9.3 Writing data pre-processing protocols

Now that you have planned out the key components of the pre-processing protocol, you can use RMarkdown’s functionality to flesh it out into a full pre-processing protocol. This gives you the chance to move beyond a simple code script, and instead include more thorough descriptions of what you’re doing at each step and why you’re doing it. You can also include discussions of potential limitations of the approach that you are taking in the pre-processing, as well as areas where other research groups might use a different approach. These details can help when it is time to write the Methods section for the paper describing your results from an experiment using these data. They can also help your research group identify pre-processing choices that might differ from other research groups, which opens the opportunity to perform sensitivity analyses regarding these pre-processing choices and ensure that your final conclusions are robust across multiple reasonable pre-processing approaches.

Protocols are common for wet lab techniques, where they provide a “recipe” that ensures consistency and reproducibility in those processes. Computational tasks, including data pre-processing, can also be standardized through the creation and use of protocol in your research group. While code scripts are becoming more common as a means of recording data pre-processing steps, they are often not as clear as a traditional protocol, in particular in terms of providing a thorough description of what is being done at each step and why it is being done that way. Data pre-processing protocols can provide these more thorough descriptions, and by creating them with RMarkdown or with similar types of “knitted” documents (modules 3.7 and 3.8), you can combine the

Figure 3.18: Example of including code in a data pre-processing protocol created with RMarkdown. This figure shows how code can be included in the RMarkdown file for a pre-processing protocol (right), and the corresponding output in the final pdf of the protocol (left), for the code to identify a ‘good’ dilution for counting CFUs for each sample. Code comments are included to provide finer-level details on the code.

executable code used to pre-process the data with extensive documentation. As a further advantage, the creation of these protocols will ensure that your research group has thought carefully about each step of the process, rather than relying on cobbling together bits and pieces of code they've found but don't fully understand. Just as the creation of a research protocol for a clinical trial requires a careful consideration of each step of the ultimate trial (Al-JunDi and SAkkA, 2016), the creation of data pre-processing protocols ensure that each step in the process is carefully considered, and so helps to ensure that each step of this process is conducted as carefully as the steps taken in designing the experiment as a whole and each wet lab technique conducted for the experiment.

A data-preprocessing protocol, in the sense we use it here, is essentially an annotated recipe for each step in preparing your data from the initial, "raw" state that is output from the laboratory equipment (or collected by hand) to a state that is useful for answering important research questions. The exact implementation of each step is given in code that can be re-used and adapted with new data of a similar format. However, the code script is often not enough to helpfully understand, share, and collaborate on the process. Instead, it's critical to also include descriptions written by humans and for humans. These annotations can include descriptions of the code and how certain parameters are standardized the algorithms in the code. They can also be used to justify choices, and link them up both with characteristics of the data and equipment for your experiment as well as with scientific principles that underlie the choices. Protocols like this are critical to allow you to standardize the process you use across many samples from one experiment, across different experiments and projects in your research laboratory, and even across different research laboratories.

As you begin adding text to your pre-processing protocol, you should keep in mind these general aims. First, a good protocol provides adequate detail that another researcher can fully reproduce the procedure (Al-JunDi and SAkkA, 2016). For a protocol for a trial or wet lab technique, this means that the protocol should allow another researcher to reproduce the process and get results that are *comparable* to your results (Al-JunDi and SAkkA, 2016); for a data pre-processing protocol, the protocol must include adequate details that another researcher, provided they start with the same data, gets *identical* results (short of any pre-processing steps that include some element of sampling or random-number generation, e.g., Monte Carlo methods). This idea—being able to exactly re-create the computational results from an earlier project—is referred to as **computational reproducibility** and is considered a key component in ensuring that research is fully reproducible.

By creating the data pre-processing protocol as a knitted document using a tool like RMarkdown (modules 3.7 and 3.8), you can ensure that the protocol is computationally reproducible. In an RMarkdown document, you include the code examples as *executable* code—this means that the code is run every time you render the document. You are therefore “checking” your code every time

“Writing a research proposal is probably one of the most challenging and difficult task as research is a new area for the majority of postgraduates and new researchers. ... Protocol writing allows the researcher to review and critically evaluate the published literature on the interested topic, plan and review the project steps and serves as a guide throughout the investigation.”

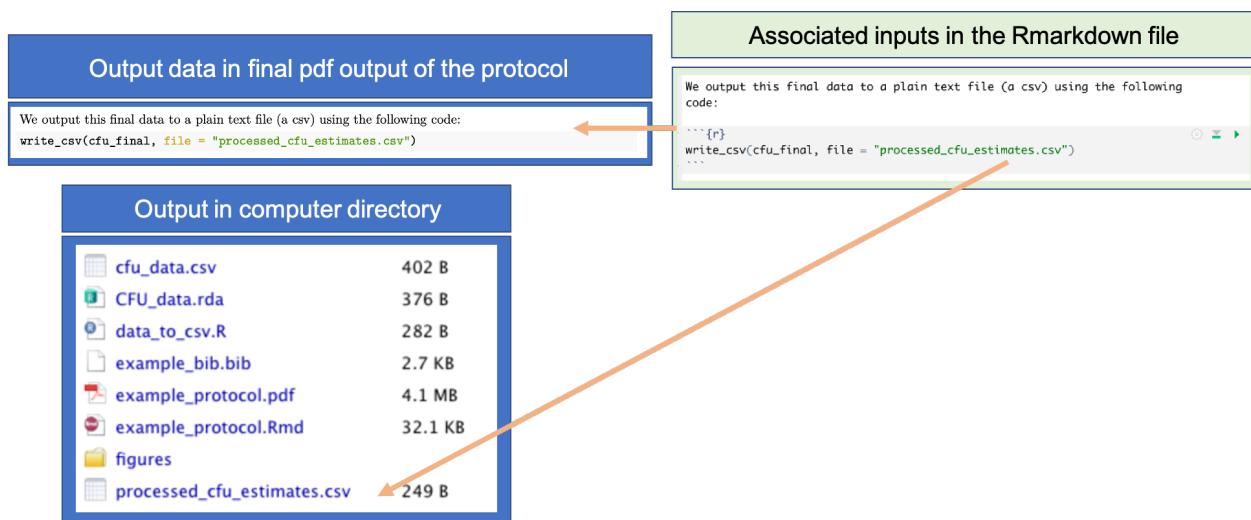
[@al2016protocol]

“Now, all scientific research involves the use of powerful computers, whether it is for the data collection, the data analysis, or both. ... We are all computational scientists now, and thus the concept of reproducibility is relevant to all scientists.”

[@peng2021reproducible]

that you run it. As the last step of your pre-processing protocol, you should output the copy of the pre-processed data that you will use for any further analysis for the project. You can use functions in R to output this to a plain text format, for example a comma-separated delimited file (modules 2.4 and 2.5). Each time you render the protocol, you will re-write this output file, and so this provides assurance that the code in your protocol can be used to reproduce your output data (since that's how you yourself created that form of the data).

Figure 3.19 provides an example from the example protocol for this module. The RMarkdown file for the protocol includes code to write out the final, pre-processed data to a comma-separated plain text file called “`processed_cfu_estimates.csv`”. This code writes the output file into the same directory where you've saved the RMarkdown file. Each time the RMarkdown file is rendered to create the pdf version of the protocol, the input data will be pre-processed from scratch, using the code throughout the protocol, and this file will be overwritten with the data generated. This guarantees that the code in the protocol can be used by anyone—you or other researchers—to reproduce the final data from the protocol, and so guarantees that these data are computationally reproducible.



In your data pre-processing protocol, show the code that you use to implement this choice and also explain clearly in the text why you made this choice and what alternatives should be considered if data characteristics are different. Write this as if you are explaining to a new research group member (or your future self) how to think about this step in the pre-processing, why you're doing it the way you're doing it, and what code is used to do it that way. You should also include references that justify choices when they are available—include these using BibTeX (module 3.8). By doing this, you will make it much easier on yourself when you write the Methods section of papers that report on the

Figure 3.19: Example of using code in pre-processing protocol to output the final, pre-processed data that will be used in further analysis for the research project. This example comes from the example protocol for this module, showing both the executable code included in the RMarkdown file for the protocol (right) and how this code is included in the final pdf of the protocol. Outputting the pre-processed data into a plain text file as the last step of the protocol helps ensure computational reproducibility for this step of working with experimental data.

data you have pre-processed, as you'll already have draft information on your pre-processing methods in your protocol.

Good protocols include not only *how* (for data pre-processing protocols, this is the code), but also *why* each step is taken. This includes explanations that are both higher-level (i.e., why a larger question is being asked) and also at a fine level, for each step in the process. A protocol should include some background, the aims of the work, hypotheses to be tested, materials and methods, methods of data collection and equipment to analyze samples (Al-JunDi and SAkkA, 2016).

This step of documentation and explanation is very important to creating a useful data pre-processing protocol. Yes, the code itself allows someone else to replicate what you did. However, only those who are very, very familiar with the software program, including any of the extension packages you include, can "read" the code directly to understand what it's doing. Further, even if you understand the code very well when you create it, it is unlikely that you will stay at that same level of comprehension in the future, as other tasks and challenges take over that brain space. Explaining for humans, in text that augments and accompanies the code, is also important because function names and parameter names in code often are not easy to decipher. While excellent programmers can sometimes create functions with clear and transparent names, easy to translate to determine the task each is doing, this is difficult in software development and is rare in practice. Human annotations, written by and for humans, are critical to ensure that the steps will be clear to you and others in the future when you revisit what was done with this data and what you plan to do with future data.

The process of writing a protocol in this way forces you to think about each step in the process, why you do it a certain way (include parameters you choose for certain functions in a pipeline of code), and include justifications from the literature for this reasoning. If done well, it should allow you to quickly and thoroughly write the associated sections of Methods in research reports and manuscripts and help you answer questions and challenges from reviewers. Writing the protocol will also help you identify steps for which you are uncertain how to proceed and what choices to make in customizing an analysis for your research data. These are areas where you can search more deeply in the literature to understand implications of certain choices and, if needed, contact the researchers who developed and maintained associated software packages to get advice.

For example, the example protocol for this module explains how to pre-process data collected from counting CFUs after plating serial dilutions of samples. One of the steps of pre-processing is to identify a dilution for each sample at which you have a "countable" plate. The protocol includes an explanation of why it is important to identify the dilution for a countable plate and also gives the rules that are used to pick a dilution for each sample, before including the code that implements those rules. This allows the protocol to provide research

group members with the logic behind the pre-processing, so that they can adapt if needed in future experiments. For example, the count range of CFUs used for the protocol to find a good dilution is about a quarter of the typically suggested range for this process, and this is because this experiment plated each sample on a quarter of a plate, rather than using the full plate. By explaining this reasoning, in the future the protocol could be adapted when using a full plate rather than a quarter of a plate for each sample.

One tool in RMarkdown that is helpful for this process is its built-in referencing system. In the previous module, we showed how you can include bibliographical references in an RMarkdown file. When you write a protocol within RMarkdown, you can include references in this way to provide background and support as you explain why you are conducting each step of the pre-processing. Figure 3.20 shows an example of the elements you use to do this, showing each element in the example protocol for this module.

Referencing in final pdf output of the protocol

```

We typically estimate bacterial load in an animal organ using the **plate count method** with **serial dilutions**. Serial dilutions allow you to create a highly diluted sample without needing a massive amount of diluent, as you increase the dilution one step at a time, steadily bringing the samples down to lower bacterial loads per volume through increased, step-by-step dilutions. This method is common across laboratories that study tuberculosis drug efficacy as a method for estimating bacterial load in animal organs (Franzblau et al. 2012) and is a well-established method across microbiology in general, dating back to Koch in the late 1800s (Wilson 1922; Ben-David and Davidson 2014).

```

References

```

Ben-David, Aviatal, and Charles E Davidson. 2014. "Estimation Method for Serial Dilution Experiments." Journal of Microbiological Methods 107: 214–21.
Franzblau, Scott G., Mary Ann DeGroot, Sang Hyun Cho, Koen Andries, Eric Nuermberger, Ian M Orme, Khisimuzi Mduli, et al. 2012. "Comprehensive Analysis of Methods Used for the Evaluation of Compounds Against Mycobacterium Tuberculosis." Tuberculosis 92 (6): 453–88.
Goldman, Emanuel, and Lorrence H Green. 2015. Practical Handbook of Microbiology. CRC press.
Jennison, Marshall W, and George P Wadsworth. 1940. "Evaluation of the Errors Involved in Estimating Bacterial Numbers by the Plating Method." Journal of Bacteriology 39 (4): 389.
Pathak, Sharad, Jane A Auhu, Nils Anders Leversen, Trude H Flo, and Birgitta Åsjo. 2012. "Counting Mycobacteria in Infected Human Cells and Mouse Tissue: A Comparison Between qPCR and CFU." PLoS One 7 (4): e34931.
Sakamoto, K. 2012. "The Pathology of Mycobacterium Tuberculosis Infection." Veterinary Pathology 49 (3): 423–39.
Tomasiewicz, Diane M, Donald K Hotchkiss, George W Reinbold, Ralston B Read, and Paul A Hartman. 1980. "The Most Suitable Number of Colonies on Plates for Counting." Journal of Food Protection 43 (4): 282–86.
Wilson, GS. 1922. "The Proportion of Viable Bacteria in Young Cultures with Special Reference to the Technique Employed in Counting." Journal of Bacteriology 7 (4): 405.
———. 1935. "The Bacteriological Grading of Milk." The Bacteriological Grading of Milk, no. 206.

```

Associated inputs in the RMarkdown file

```

---
title: "Protocol: Estimating bacterial loads from plating samples at different dilutions"
author: "Henao-Tamayo Research Laboratory"
date: "Last edited: `r Sys.Date()`"
output: pdf_document
bibliography: example_bib.bib
---

We typically estimate bacterial load in an animal organ using the **plate count method** with **serial dilutions**. Serial dilutions allow you to create a highly diluted sample without needing a massive amount of diluent, as you increase the dilution one step at a time, steadily bringing the samples down to lower bacterial loads per volume through increased, step-by-step dilutions. This method is common across laboratories that study tuberculosis drug efficacy as a method for estimating bacterial load in animal organs ([#Franzblau2012comprehensive]) and is a well-established method across microbiology in general, dating back to Koch in the late 1800s ([#Wilson1922portion]; [#ben2014estimation]).

## References

```

Associated inputs in the Bibtex file

```

@article{franzblau2012comprehensive,
  title={Comprehensive analysis of methods used for the evaluation of compounds against Mycobacterium tuberculosis},
  author={Franzblau, Scott G and DeGroot, Mary Ann and Cho, Sang Hyun and Andries, Koen and Nuermberger, Eric and Orme, Ian M and Mduli, Khisimuzi and Angulo-Barturen, Iñaki and Dick, Thomas and Dartois, Veronique and others},
  journal={[Tuberculosis]},
  volume={92},
  number={6},
  pages={453–488},
  year={2012},
  publisher={Elsevier}
}

```

Other helpful tools in RMarkdown are tools for creating equations and tables. As described in the previous module, RMarkdown includes a number of formatting tools. You can create simple tables through basic formatting, or more complex tables using add-on packages like `kableExtra`. Math can be typeset using conventions developed in the LaTeX mark-up language. The previous module provided advice and links to resources on using these types of tools. Figure 3.21 gives an example of them in use within the example protocol for this module.

You can also include figures, either figures created in R or outside figure files. Any figures that are created by code in the RMarkdown document will

Figure 3.20: Including references in a data pre-processing protocol created with RMarkdown. RMarkdown has a built-in referencing system that you can use, based on the BibTeX system for LaTeX. This figure shows examples from the example protocol for this module of the elements used for referencing. You create a BibTeX file with information about each reference, and then use the key for the reference within the text to cite that reference. All cited references will be printed at the end of the document; you can chose the header that want for this reference section in the RMarkdown file ('References' in this example). In the YAML of the RMarkdown file, you specify the path to the BibTeX file (with the 'bibliography: ' key), so it can be linked in when the RMarkdown file is rendered.

Equations and tables in pdf output			Associated inputs in the Rmarkdown file																	
<p>These following general equations apply for determining the total dilution in any of the tubes:</p> $\text{Dilution factor in tube} = 5^x$ $\text{Dilution in tube} = \frac{1}{5^x}$ <p>where x is the dilution level in the tube.</p> <p>As you move through the levels of dilution, each level will become diluted by an additional factor of 5 compared to the homogenate in the first tube (Figure 1):</p> <table border="1"> <thead> <tr> <th>Dilution level</th> <th>Dilution factor in tube</th> <th>Dilution in tube</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>$5^0 = 1$</td> <td>1</td> </tr> <tr> <td>1</td> <td>$5^1 = 5$</td> <td>$\frac{1}{5}$</td> </tr> <tr> <td>2</td> <td>$5^2 = 25$</td> <td>$\frac{1}{25}$</td> </tr> <tr> <td>3</td> <td>$5^3 = 125$</td> <td>$\frac{1}{125}$</td> </tr> </tbody> </table>			Dilution level	Dilution factor in tube	Dilution in tube	0	$5^0 = 1$	1	1	$5^1 = 5$	$\frac{1}{5}$	2	$5^2 = 25$	$\frac{1}{25}$	3	$5^3 = 125$	$\frac{1}{125}$	<p>These following general equations apply for determining the total dilution in any of the tubes:</p> <pre>\$\boxed{\text{Dilution factor in tube}} = 5^x</pre> <pre>\$\boxed{\text{Dilution in tube}} = \frac{1}{5^x}</pre> <p>where $\\$x\\$ is the dilution level in the tube.</p> <p>As you move through the levels of dilution, each level will become diluted by an additional factor of 5 compared to the homogenate in the first tube shown in Figure 1:</p> <pre> Dilution level Dilution factor in tube Dilution in tube ----- ----- ----- 0 \$5^{\{0\}} = 1\\$ \\$1\\$ 1 \$5^{\{1\}} = 5\\$ \\$\frac{1}{5}\\$ 2 \$5^{\{2\}} = 25\\$ \\$\frac{1}{25}\\$ 3 \$5^{\{3\}} = 125\\$ \\$\frac{1}{125}\\$ </pre>		
Dilution level	Dilution factor in tube	Dilution in tube																		
0	$5^0 = 1$	1																		
1	$5^1 = 5$	$\frac{1}{5}$																		
2	$5^2 = 25$	$\frac{1}{25}$																		
3	$5^3 = 125$	$\frac{1}{125}$																		

automatically be included in the protocol. For other graphics, you can include image files (e.g., png and jpeg files) using the `include_graphics` function from the `knitr` package. You can use options in the code chunk options to specify the size of the figure in the document and to include a figure caption. The figures will be automatically numbered in the order they appear in the protocol.

Figure 3.22 shows an example of how external figure files were included in the example protocol. In this case, the functionality allowed us to include an overview graphic that we created in PowerPoint and saved as an image as well as a photograph taken by a member of our research group.

Finally, you can try out even more complex functionality for RMarkdown as you continue to build data pre-processing protocols for your research group. Figure 3.23 shows an example of using R code within the YAML of the example protocol for this module; this allows us to include a “Last edited” date that is updated with the day’s date each time the protocol is re-rendered.

3.9.4 Applied exercise

To wrap up this module, try downloading both the source file and the output of this example data pre-processing protocol. Again, you can find the source code (the RMarkdown file) here and the output file here. If you would like to try re-running the file, you can get all the additional files you’ll need (the original data file, figure files, etc.) here. See if you can compare the elements of the RMarkdown file with the output they produce in the PDF file. Read through the descriptions of the protocol. Do you think that you could recreate the process if your laboratory ran a new experiment that involved plating samples to estimate bacterial load?

Figure 3.21: Example of including tables and equations in an RMarkdown data pre-processing protocol.

Figures in output pdf

Figure 1: Visual overview of the dilution and plating process for this experiment. For each animal, half the spleen was homogenized in 500 microliters phosphate buffer saline (PBS) for plating ('Homogenate' tube in graphic). Three serial dilutions were created by resuspending 100 microliters from the homogenate or previous dilution in 400 microliters PBS ('Level 1 dilution', 'Level 2 dilution' and 'Level 3 dilution' tubes in graphic). From each tube, 100 microliters were plated in one quarter of a 7H11 agar plate (circle at bottom of graphic). After 3–5 weeks of incubation at 37°C, colony-forming units were counted from each quarter of the plate and recorded. These are the input data for this protocol.

Figure 2: Example of a plate from this process. Each plate is divided into quarters, with a single sample (i.e., from a specific tube shown in Figure 1) spread in each quarter of the plate. The shows the plate after enough time has passed following plating for colony forming units (CFUs) to grow. In this example, CFUs can easily be counted in the bottom two quadrants of the plate, but may be too numerous to count in the top two quadrants.

Associated inputs in the Rmarkdown file

```
```{r platingexample, echo = FALSE, out.width = "\textwidth", fig.cap = "Visual overview of the dilution and plating process for this experiment. For each animal, half the spleen was homogenized in 500 microliters phosphate buffer saline (PBS) for plating ('Homogenate' tube in graphic). Three serial dilutions were created by resuspending 100 microliters from the homogenate or previous dilution in 400 microliters PBS ('Level 1 dilution', 'Level 2 dilution' and 'Level 3 dilution' tubes in graphic). From each tube, 100 microliters were plated in one quarter of a 7H11 agar plate (circle at bottom of graphic). After 3–5 weeks of incubation at 37°C, colony-forming units were counted from each quarter of the plate and recorded. These are the input data for this protocol."}
knitr::include_graphics("figures/protocol_graphic.png")```
```{r platingexample2, echo = FALSE, out.width = "\textwidth", fig.cap = "Example of a plate from this process. Each plate is divided into quarters, with a single sample (i.e., from a specific tube shown in Figure 1) spread in each quarter of the plate. The shows the plate after enough time has passed following plating for colony forming units (CFUs) to grow. In this example, CFUs can easily be counted in the bottom two quadrants of the plate, but may be too numerous to count in the top two quadrants."}
knitr::include_graphics("figures/bacteria_plate.JPG")```
```

```

### Figure files in Project directory

Figure 3.22: Example of including figures from image files in an RMarkdown data pre-processing protocol.

### Use of date in final pdf output

Protocol: Estimating bacterial loads from plating samples at different dilutions  
Henao-Tomayo Research Laboratory  
Last edited: 2021-04-02

### Associated inputs in the Rmarkdown file

```

title: "Protocol: Estimating bacterial loads from plating samples at different dilutions"
author: "Henao-Tomayo Research Laboratory"
date: "Last edited: `r Sys.Date()`"
output: pdf_document
bibliography: example_bib.bib

```

Figure 3.23: Example of using more advanced RMarkdown functionality within a data pre-processing protocol. In this example, R code is incorporated into the YAML of the document to include the date that the document was last rendered, marking this on the pdf output as the \*Last edited\* date of the protocol.



# 4

## References

### 4.1 Extra quotes

Quotes from research that we haven't used yet, but might want to as we edits.

"Many journals provide mechanisms to make reproducibility possible, including *PLoS*, *Nature*, and *Science*. This entails ensuring access to the computer code and datasets used to produce the results of a study. In contrast, replication of scientific findings involves research conducted by independent researchers using their own methods, data, and equipment that validate or confirm the findings of earlier studies. Replication is not always possible, however, so reproducibility is a minimum and necessary standard for confirming scientific findings." (Keller et al., 2017)

"Reproducibility goes well beyond validating statistical results and includes empirical, computational, ethical, and statistical analyses. For example, empirical reproducibility ensures that the same results are obtained from the data and code used in the original study, and statistical reproducibility focuses on statistical design and analysis to ensure replication of an experiment. There are also definitions of ethical reproducibility, such as documenting the methods used in biomedical research or in social and behavioral science research so others can reproduce algorithms used in analysis." (Keller et al., 2017)

"Many studies have been undertaken to understand the reproducibility of scientific findings and have come to different conclusions about the findings. For example, one scientist argues that half of all scientific discoveries are false (Ioannidis 2005), others find that a large portion of the reproduced findings produce weaker evidence compared with the original findings (Nosek et al. 2015), and others find that 4/5 of the results are true positives (Jager and Leek 2013). ... Despite this controversy, the premise underlying reproducibility is data quality in the form of good experimental design and execution, documentation, and making scientific inputs available for reproducing the scientific work." (Keller et al., 2017)

"In the computer science, engineering, and business worlds, data quality management focuses largely on administrative data and is driven by the need to have accurate, reliable data for daily operations. The kinds of data traditionally discussed in this data quality literature are fundamental to the functioning of an organization—if the data are bad, firms will lose money or defective products will be manufactured. The advent of data quality in the engineering and business

worlds traces back to the 1940s and 1950s with Edward Deming and Joseph Juran. Japanese companies embraced these methods and transformed their business practices using them. Deming's approach used statistical process control that focused on measuring inputs and processes and thus minimized product inspections after a product was build." (Keller et al., 2017)

[In business, ] "Data quality is further defined from the perspective of the ease of use of the data with respect to the integrity, accuracy, interpretability, and value assessed by the data user and other attributes that make the data valuable." (Keller et al., 2017)

"Many scientists spend a lot of time using Excel, and without batting an eye will change the value in a cell and save the results. I strongly discourage modifying data this way. Instead, a better approach is to treat all data as *read-only* and only allow programs to read data and create new, separate files of results. Why is treating data as read-only important in bioinformatics? First, modifying the data in place can easily lead to corrupted results. For example, suppose you wrote a script that directly modifies a file. Midway through processing a large file, your script encounters an error and crashes. Because you've modified the original file, you can't undo the changes and try again (unless you have a backup)! Essentially, this file is corrupted and can no longer be used. Second, it's easy to lose track of how we've changed a file when we modify it in place. Unlike a workflow where each step has an input file and an output file, a file modified in place doesn't give us any indication of what we've done to it. Were we to lose track of how we've changed a file and don't have a backup copy of the original data, our changes are essentially irreproducible. Treating data as read-only may seem counterintuitive but may seem counterintuitive to scientists familiar with working extensively in Excel, but it's essential to robust research (and prevents catastrophe, and helps reproducibility). The initial difficulty is well worth it; it also fosters reproducibility. Additionally, any step of the analysis can easily be redone, as the input data is unchanged by the program." (Buffalo, 2015)

"'Plain text' data files are encoded in a format (typically UTF-8) that can be read by humans and computers alike. The great thing about plain text is their simplicity and their ease of use: any programming language can read a plain text file. The most common plain text format is .csv, comma-separated values, in which columns are separated by commas and rows are separated by line breaks." (Gillespie and Lovelace, 2016)

**Designed data** is "data that have traditionally been used in scientific discovery. Designed data include statistically designed data collections, such as surveys or experiments, and intentional observational collections. Examples of intentional observational collections include data obtained from specially designed instruments such as telescopes, DNA sequencers, or sensors on an ocean buoy, and also data from systematically designed case studies such as health registries. Researchers have frequently devoted decades of systematic research to understanding and characterizing the properties of designed data collections." This contrasts with administrative data and opportunity data. (Keller et al., 2017)

"The need to address data quality is a persistent one in the physical and biological sciences, where scientists often seek to understand subtle effects that leave minute traces in large volumes of data. ... For most scientists, three factors motivate their work on data quality: first, the need to create a strong foundation

of data from which to draw their own conclusions; second, the need to protect their data and conclusions from the criticisms of others; and third, the need to understand the potential flaws in data collected by others. The work of these scientists in data quality primarily concentrates on the design and execution of experiments, including in laboratory, field, and clinical settings. The key ingredients are measurement implementation, laboratory and experimental controls, documentation, analysis, and curation of data.” (Keller et al., 2017)

“The concept of data quality management developed in the 1980s in step with the technological ability to access stored data in a random fashion. Specifically, as data management encoding process moved from the highly controlled and defined linear process of transcribing information to tape, to a system allowing for the random updating and transformation of data fields in a record, the need for a higher level of control over what exactly can go into a data field (including type, size, and values) became evident. Two key data quality concepts came from these data management advances—ensuring data integrity and cleansing the legacy data. Data integrity refers to the rules and processes put in place to maintain and assure the accuracy and consistency of a system that stores, processes, and retrieves data. Data cleaning refers to the identification of incomplete, incorrect, inaccurate, or irrelevant parts of the data and then replacing, modifying, or deleting this so-called dirty or coarse data.” (Keller et al., 2017)

“As the capability to store increasing amounts of data grew, so did the business motivation to improve the quality of administrative data and thereby improve decision making, reduce costs, and gain trust of customers.” (Keller et al., 2017)

“Determine whether there is a community-based metadata schema or standard (i.e., preferred sets of metadata elements) that can be adopted.” (Michener, 2015)

Might make more sense in tidy data section: “Although many standards have been defined for data and model representations, they only ensure that data and models that comply with these standards can be used by software that support these standards; they do not ensure that multiple software tools can be used seamlessly. When software tools are developed by independent research groups or companies without an explicit agreement as to how they can be integrated, this can cause problems when forming a workflow of multiple tools. This is because the tools are likely to be inconsistent in their operating procedures and their use of various non-standardized data formats. Thus, users often have to convert data formats, to learn operating procedures for each tool, and sometimes even to adjust operating environments. This impedes productivity, undermines the flexibility of the workflow, and is prone to errors.” (Ghosh et al., 2011)

“Ontologies define the relationships and hierarchies between different terms and allow the unique, semantic annotation of data.” (Ghosh et al., 2011)

“There are several international and national bodies, such as OMG, W3C, IEEE, ANSI, and IETF... that formally approve standards or provide a framework for standards development. Although some of the systems biology standards have been certified (for example, SBML is officially adapted by IETF), in general in life sciences this procedure is not particularly important—many of the most successful standards such as GO have not undergone any official approval procedure, but instead have become *de facto* standards. In fact, many of the most successful standards in other domains are *de facto* standards.” (Brazma et al., 2006)

"There are four steps involved in developing a complete and self-contained standard: conceptual model design, model formalization, development of a data exchange format, and implementation of the supporting tools." (Brazma et al., 2006)

"Two competing goals should be balanced when developing the conceptual model of a domain: domain specificity and the need to find the common ground for all related applications. Arguably, the most useful standards are those that consist of the minimum number of most informative parameters. Keeping the list short makes it simple and practicable, while selecting the most informative features ensures accuracy and efficiency. The need for minimalism is reflected by the titles for many such standards—Minimum Information About XYZ."

"For a standard to be successful, laboratory information management systems (LIMS), databases and data analysis, and modeling tools should comply with it. One way of fostering this is to develop easy-to-use 'middleware'—software components that hide the technical complexities of the standard and facilitate manipulation of the standard format in an easy way." (Brazma et al., 2006)

**Semantics:** The meaning of something; in computer science, it is usually used in opposition to syntax (that is, format)." (Brazma et al., 2006)

"Excel is by far the most common spreadsheet software, but many other spreadsheet programs exist, notably the Open Office Calc software, which is an open source alternative and stores spreadsheets in an XML-based open standard format called Open Document Format (ODF). This allows the data to be accessed by a wider variety of software. However, even ODF is not ideal as a storage format for a research data set because spreadsheet formats contain not only the data that is stored in the spreadsheet, but also information about how to display the data, such as fonts and colors, borders and shading." (Murrell, 2009)

"Spreadsheet software is useful because it displays a data set in a nice rectangular grid of cells. The arrangement of cells into distinct columns shares the same benefits as fixed-width format text files: it makes it very easy for a human to view and navigate within the data. ... because most spreadsheet software provides facilities to import a data set from a wide variety of formats, these benefits of the spreadsheet software can be enjoyed without having to suffer the negatives of using a spreadsheet format for data storage. For example, it is possible to store the data set in a CSV format and use spreadsheet software to view or explore the data." (Murrell, 2009)

"Thinking about what sorts of questions will be asked of the data is a good way to guide the design of data storage." (Murrell, 2009)

First, you can still use spreadsheets, but reduce their use to recording data, leaving all data cleaning and analysis to be handled with other software. To make it easier to collaborate with statisticians and to interface with a program like R for data cleaning and analysis, it will be easiest if you set up your data recording to include with other statistical programs like R or Python. These steps are described in a later section, "...".

- Each sheet of the spreadsheet should contain data from a single experiment.

- Never use whitespace to represent a meaningful separation in data within a spreadsheet. Never include multiple tables of data in the same sheet.
- The first row of the spreadsheet should include a short column name for each column with data. All column name information should be within a single row (i.e., avoid subheadings). Avoid any special characters (e.g., "%") in column names. Instead, use only letters, numbers, and underscores ("\_"), and start with a letter. It is especially helpful if you can avoid spaces in column names.
- Missing data should be represented consistently in cells. "NA" is one choice. If you want to clarify why data is missing, it's much better to add a column (e.g., "why\_missing") where you can provide those details in text, rather than combining within a single column numerical observation data with textual reasons for missingness in cells with missing values.

Next, you could record data using a statistical language like R. There is an excellent Integrated Development Environment for R called RStudio, and it creates a much clearer interface with R compared to running R from a command line, particularly for new users. RStudio allows you to open delimited plain text files, like csvs, using a grid-style interface. This grid-style interface looks very similar to a spreadsheet, but lacks the ability to include formulas or macros. Therefore, this format enforces a separation of the recording of raw data from the cleaning and analysis of the data.

#### [R Project templates]

Data cleaning and analysis can then be shifted away from the files used to record the data and into reproducible scripts. These scripts can be clearly documented, either through comments in the code or through open source documentation tools like RMarkdown than interweave code and text in a way that allows the creation of documents that are easier to read than commented code.

This documentation should explain why each step is being done. In cases where it is not immediately evident from the code *how* the step is being done, this should be documented as well. Any assumptions being used should be clarified in the documentation.

"When we have only the letters, digits, special symbols, and punctuation marks that appear on a standard (US) English keyboard, then we can use a single byte to store each character. This is called an ASCII encoding (American Standard Code for Information Exchange). ... UNICODE is an attempt to allow computers to work with all of the characters in all of the languages of the world. Every character has its own number, called a 'code point', often written in the form U+xxxxxx, where every x is a hexadecimal digit. ... There are two main 'encodings' that are used to store a UNICODE code point in memory. UTF-16 always uses two bytes per character of text and UTF-8 uses one or more bytes, depending on which characters are stored. If the text is only ASCII, UTF-8 will only use one byte per character. ... This encoding is another example of additional information that may have to be provided by a human before the computer can read data correctly from a plain text file, although many software packages will

cope with different encodings automatically." (Murrell, 2009)

"A PDF document is primarily a description of how to *display* information. Any data values within a PDF document will be hopelessly entwined with information about how the data values should be displayed." (Murrell, 2009)

"Another major weakness of free-form text files is the lack of information *within the file itself* about the structure of the file. For example, plain text files do not contain information about which special character is being used to separate fields in a delimited file, or any information about the widths of fields within a fixed-width format. This means that the computer cannot automatically determine where different fields are within each row of a plain text file, or even how many fields there are. A fixed-width format avoids this problem, but enforcing a fixed length for fields can create other difficulties if we do not know the maximum possible length for all variables. Also, if the values for a variable can have very different lengths, a fixed-width format can be inefficient because we store lots of empty space for short values. The simplicity of plain text files make it easy for a computer to read a file as a series of characters, but the computer cannot easily distinguish individual data values from the series of characters. Even worse, the computer has no way of telling what sort of data is stored in each field. ... In practice, humans must supply additional information about a plain text file before a computer can successfully determine where the different fields are within a plain text file *and* what sort of data is stored in each field." (Murrell, 2009)

"In bioinformatics, the plain-text data we work with is often encoded in ASCII. ASCII is a character encoding scheme that uses 7 bits to represent 128 different values, including letters (upper- and lowercase), numbers, and special nonvisible characters. While ASCII only uses 7 bits, nowadays computers use an 8-bit byte (a unit representing 8 bits) to store ASCII characters. ... Because plain-text data uses characters to encode information, our encoding scheme matters. When working with a plain-text file, 98% of the time you won't have to worry about the details of ASCII and how your file is encoded. However, the 2% of the time when encoding data does matter—usually when an invisible non-ASCII character has entered the data—it can lead to major headaches." (Buffalo, 2015)

"Programs [in Unix] retrieve the data in a file by a system call (a subroutine in the kernel) called `read`. Each time `read` is called, it returns the next part of a file—the next line of text typed on the terminal, for example. `read` also says how many bytes of the file were returned, so end of file is assumed when a `read` says 'zero bytes are being returned'. If there were any bytes left, `read` would have returned some of them." (Kernighan and Pike, 1984)

"The format of a file is determined by the programs that use it; there is a wide variety of file types, perhaps because there is a wide variety of programs. But since file types are not determined by the file system, the kernel can't tell you the type of a file: it doesn't know it. The `file` command makes an educated guess ... To determine the types, `file` doesn't pay attention to the names (although it could have), because naming conventions are just conventions, and thus not perfectly reliable. For example, files suffixed '.c' are almost always C source, but there is nothing to prevent you from creating a '.c' file with arbitrary contents. Instead, `file` reads the first hundred bytes of a file and looks for clues to that file type. ... In Unix systems there is just one kind of file, and all that is required

to access a file is its name. The lack of file formats is an advantage overall—programmers don't need to worry about file types, and all the standard programs will work on any file.” (Kernighan and Pike, 1984)

“The Unix file system is organized so you can maintain your own personal files without interfering with files belonging to other people, and keep people from interfering with you too.” (Kernighan and Pike, 1984)

“The clinical patient health record is a longitudinal administrative record of an individual's health information: all the data related to an individual's or a population's health. The health record is a set of nonstandardized data that spans multiple levels of aggregation, from a single measurement element (blood pressure) to collections of diagnoses and related clinical observations. This complexity is compounded by the high degree of human interaction involved in the production of clinical records, including self-reported data, medical diagnosis, and other patient information.” (Keller et al., 2017)

“Sharing data through repositories enhances both the quality and the value of the data through standardized processes for curation, analysis, and quality control. By allowing broad access to data, these repositories encourage and support the use of previously collected data to test and extend previous results. Data repositories are quite common in science fields such as astronomy, genomics, and earth sciences. ... These repositories have accelerated discovery by expanding the reach of these data to scientists who are not involved in the initial data collection and experiments. Repositories address challenges that affect data quality through governance, interoperability across systems, and costs.” (Keller et al., 2017)

One example of a repository is “the sharing of cDNA microarray data through research consortia, which has led to a common set of standards and relatively homogeneous data classes. There are many issues with the sharing of these data, which requires the transformation of biologic to numeric data. These issues may include loss of context, such as laboratory processes followed, and therefore lack of information about the quality of the data when they are transformed. To avoid this loss of information, the consortium ensures that documentation is comprehensive so that other researchers can assess the quality of the data and make comparisons with other studies using the same data. This documentation also includes information on when incorrect assignments of sequence identity are made so that errors are not perpetuated in other studies.” (Keller et al., 2017)

**Data exchange format:** A file or message format that is formally defined so that software can be built that ‘knows’ where to find various pieces of information.” (Brazma et al., 2006)

“Everything in the Unix system is a file. That is less of an oversimplification than you might think. When the first version of the system was designed, before it even had a name, the discussions focused on the structure of a file system that would be clean and easy to use. The file system is central to the success and convenience of the Unix system. It is one of the best examples of the ‘keep it simple’ philosophy, showing the power achieved by careful implementation of a few well-chosen ideas.” (Kernighan and Pike, 1984)

“A file is a sequence of bytes. (A byte is a small chunk of information, typically 8 bits long. For our purposes, a byte is equivalent to a character.) No structure

is imposed on a file by the system, and no meaning is attached to its contents—the meaning of the bytes depends solely on the programs that interpret the file. Furthermore, ... this is true not just of disc files but of peripheral devices as well. Magnetic tapes, mail messages, characters typed on the keyboard, line printer output, data flowing in pipes—each of these is just a sequence of bytes as far as the systems and the programs in it are concerned.” (Kernighan and Pike, 1984)

“The Comma-Separated Value (CSV) format is a special case of a plain text format. Although not a formal standard, CSV files are very common and are a quite reliable plain text delimited format that at least solves the problem of where the fields are in each row of the file. The main rules for the CSV format are: (1) **Comma-delimited:** Each field is separated by a comma (i.e., the character , is the delimiter); (2) **Double-quotes are special:** Fields containing commas must be surrounded by double-quotes .... (3) **Double-quote escape sequence:** Fields containing double quotes must be surrounded by double-quotes and each embedded double-quote must be represented using two double quotes ... ; (4) **Header information:** There can be a single header containing the names of the fields.” (Murrell, 2009)

“A data file metaformat is a set of syntactic and lexical conventions that is either formally standardized or sufficiently well established by practice that there are standard service libraries to handle marshaling and unmarshaling it. Unix has evolved or adopted metaformats suitable for a wide range of applications [including delimiter-separated values and XML]. It is good practice to use one of these (rather than an idiosyncratic custom format) whenever possible. The benefits begin with the amount of custom parsing and generation code that you may be able to avoid writing by using a service library. But the most important benefit is that developers and even many users will instantly recognize these formats and feel comfortable with them, which reduces the friction costs of learning new programs.” (Raymond, 2003)

“There are three flavors you will encounter: tab-delimited, comma-separated, and variable space-delimited. Of these three formats, tab-delimited is the most commonly used in bioinformatics. File formats such as BED, GTF/GFF, SAM, tabular BLAST output, and VCF are all examples of tab-delimited files. Columns of a tab-delimited file are separated by a single tab character (which has the escape code \t). A common convention (but not a standard) is to include metadata on the first few lines of a tab-delimited file. These metadata lines begin with # to differentiate them from the tabular data records. Because tab-delimited files use a tab to delimit columns, tabs in data are not allowed. Comma-separated values (CSV) is another common format. CSV is similar to tab-delimited, except the delimiter is a comma character. While not a common in bioinformatics, it is possible that the data stored in CSV format contain commas (which would interfere with the ability to parse it). Some variants just don't allow this, while others use quotes around entries that could contain commas. Unfortunately, there's no standard CSV format that defines how to handle this and many other issues with CSV—though some guidelines are given in RFC 4180. Lastly, there are space-delimited formats. A few stubborn bioinformatics programs use a variable number of spaces to separate columns. In general, tab-delimited formats and CSV are better choices than space-delimited formats because it's quite common to encounter data containing spaces.” (Buffalo, 2015)

“There are long-standing Unix traditions about how textual data formats ought

to look. Most of these derive from one or more of the standard Unix metaformats ... just described [e.g., DSV, XML]. It is wise to follow these conventions unless you have strong and specific reasons to do otherwise. ... (1) *One record per newline-terminated line, if possible.* This makes it easy to extract records with text-stream tools. For data interchange with other operating systems, it's wise to make your file-format parser indifferent to whether the line ending is LF or CR-LF. It's also conventional to ignore trailing whitespace in such formats; this protects against common editor bobbles. (2) *Less than 80 characters per line if possible.* This makes the format browseable in an ordinary-sized terminal window. If many records must be longer than 80 characters, consider a stanza format... (3) *Use # as an introducer for comments.* It's good to have a way to embed annotations and comments in data files. It's best if they're actually part of the file structure, and so will be preserved by tools that know its format. For comments that are not preserved during parsing, # is the conventional start character. (4) *Support the backslash convention.* The least surprising way to support nonprintable control characters is by parsing C-like backslash escapes ..." (Raymond, 2003)

"You can take apart these formats and find out which decisions were made to create them ... even old Microsoft Word, which in a long and painful political bottle, finally settled down and 'opened' its format, countless hundreds of pages of documentation defining how words appear, how tables of contents are registered, how all of the things that make up a Word document are to be represented. The Microsoft Office File Formats specifications are of a most disturbing, fascinating quality: one can read through them and think: Yes, I see this, I think I understand. But why? ... Even Word is opened now, just regular XML. Strange XML to be sure. All the codes once hidden are revealed." (Ford, 2014)

"We wish to draw a distinction between data that is machine-actionable as a result of specific investment in software supporting that data-type, for example, bespoke parsers that understand life science PDB files or space science Space Physics Archive Search and Extract (SPASE) files, and data that is machine-actionable exclusively through the utilization of general-purpose, open technologies. To reiterate the earlier point—ultimate machine actionability occurs when a machine can make a useful decision regarding data that it has not encountered before. This distinction is important when considering both (a) the rapidly growing and evolving data environment, with new technologies and new, more complex data-types continuously being developed, and (b) the growth of general-purpose repositories, where the data-types encountered by an agent are unpredictable. Creating bespoke parsers, in all computer languages, is not a sustainable activity." (Wilkinson et al., 2016)

"[One] way data can come from the Internet is through a web API, which stands for *application programming interface*. The number of APIs that are being offered by organizations is growing at an ever increasing rate... Web APIs are not meant to be presented in a nice layout, such as websites. Instead, most web APIs return data in a structured format, such as JSON or XML. Having data in a structured format has the advantage that the data can be easily processed by other tools." (Janssens, 2014)

"The *pileup format* [is] a plain-text format that summarizes reads' bases at each chromosome position by stacking or 'piling up' aligned reads." (Buffalo, 2015)

"Data compression, the process of condensing data so that it takes up less space (on disk drives, in memory, or across network transfers), is an indispensable technology in modern bioinformatics. For example, sequences from a recent Illumina HiSeq run when compressed with Gzip take up 21,408,674,240 bytes, which is a bit under 20 gigabytes. Uncompressed, this file is a whopping 63,203,414,514 bytes (around 58 gigabytes). This FASTQ file has 150 million 200bp reads, which is 10x coverage of the hexaploid wheat genome. The compression ratio (uncompressed size/ compressed size) of this data is approximately 2.95, which translates to a significant space saving of about 66%. Your own bioinformatics projects will likely contain much more data, especially as sequencing costs continue to drop and it's possible to sequence genomes to higher depth, include more biological replicates or time points in expression studies, or sequence more individuals in genotyping studies. For the most part, data can remain compressed on the disk throughout processing and analysis. Most well-writtent bioinformatics tools can work natively with compressed data as input, without requiring us to decompress it to disk first. Using pipes and redirection, we can stream compressed data and write compressed files directly to the disk. Additionally, common Unix tools like *cat*, *grep*, and *less* all have variants that work with compressed data, and Python's *gzip* module allows us to read and write compressed data from within Python. So while working with large datasets in bioinformatics can be challenging, using the compression tools in Unix and software libraries make our lives much easier."

(Buffalo, 2015)

"Non-text files definitely have their place. For example, very laarge databases usually need extra address information for rapid access; this has to be binary for efficiency. But every file format that is not text must have its own family of support programs to do things that the standard tools could perform if the format were text. Text files may be a little less efficient in machine cycles, but this must be balanced against the cost of extra software to maintain more specialized formats. If you design a file format, you should think carefully before choosing a non-textual representation." (Kernighan and Pike, 1984)

"*Out-of-memory approaches* [are] computational strategies built arouond storing and working with data kept out of memory on the disk. Reading data from a disk is much, much slower than working with data in memory... but in many cases this is the approach we have to take when in-memory (e.g., loading the entire dataset into R) or streaming approaches (e.g., using Unix pipes ...) aren't appropriate."

(Buffalo, 2015)

"In general, it is possible to jump directly to a specific location within a binary format file, whereas it is necessary to read a text-based format from the beginning and one character at a time. This feature of accessing binary formats is called **random access** and it is generlaly faster than the typically **sequential access** of text files." (Murrell, 2009)

"We often need fast read-only access to data linked to a genomic location or range. For the scale of data we encounter in genomics, retrieving this type of data is not trivial for a few reasons. First the data might not fit entirely in memory, requiring an approach where data is kept out of memory (in other words, on a slow disk). Second, even powerful relational database systems can be sluggish when querying out millions of entries that overlap a specific region—an incredibly common operation in genomics. [BGZF and Tabix] are specifically

designed to get around these limitations, allowing fast random-access of tab-delimited genome position data." (Buffalo, 2015)

"Samtools now supports (after version 1) a new, highly compressed file format known as CRAM. Compressing alignments with CRAM can lead to a 10%–30% filesize reduction compared to BAM (and quite remarkably, with no significant increase in compression or decompression time compared to BAM). CRAM is a *reference-based* compression scheme, meaning only the aligned sequence that's different from the reference sequence is recorded. This greatly reduces file size, as many sequence may align with minimal difference from the reference. As a consequence of this reference-based approach, it is imperative that the reference is available and does not change, as this would lead to a loss of data kept in the CRAM format. Because the reference is so important, CRAM files contain an MD5 checksum of the reference file to ensure it has not changed. CRAM also has support for multiple different *lossy compression* methods. Lossy compression entails some information about an alignment and the original read is lost. For example, it's possible to bin base quality scores using a lower resolution binning scheme to reduce the filesize." (Buffalo, 2015)

"Very often we need efficient random access to subsequences of a FASTA file (given regions). At first glance, writing a script to do this doesn't seem difficult. We could, for example, write a script that iterates through FASTA entries, extracting sequences that overlaps the range specified. However, this is not an efficient method when extracting a few *random* subsequences. To see why, consider accessing the sequence from position chromosome 8 (123,407,082 to 123,419,742) from the mouse genome. This approach would needlessly parse and load chromosomes 1 through 7 into memory, even though we don't need to extract subsequences from these chromosomes. Reading entire chromosomes from disk and copying them into memory can be quite inefficient—we would have to load all 125 megabytes of chromosome 8 to extract 3.6kb! Extracting numerous random subsequences from a FASTA file can be quite computationally costly. A common computational strategy that allows for easy and fast random access is *indexing* the file. Indexed files are ubiquitous in bioinformatics." (Buffalo, 2015)

"We can avoid needlessly reading the entire file off of the disk by using an index that points to where certain blocks are in the file. In the case of our FASTA file, the index essentially stores the location of where each sequence begins in the file (as well as other necessary information). When we look up a range like chromosome 8 (123,407,082–123,410,744), *samtools faidx* uses the information in the index to quickly calculate exactly where in the file those bases are. Then, using an operation called a file seek, the program jumps to this exact position (called the *offset*) in the file and starts reading the sequence. Having precomputed file offsets combined with the ability to jump to those exact positions is what makes accessing sections of an indexed file fast." (Buffalo, 2015)

"The data revolution within the biological and physical science world is generating massive amounts of data from ... a wide range of ... projects, such as those undertaken at the Large Hadron Collider and genomics-proteomics-metabolomics research." (Keller et al., 2017)

"Community standards for data description and exchange are crucial. These facilitate data reuse by making it easier to import, export, compare, combine,

and understand data. Standards also eliminate the need for the data creator to develop unique descriptive practices. They open the door to development of disciplinary repositories for specific classes of data and specialized software management tools. GenBank, the US NIH genetic sequence database, and the US National Virtual Observatory are good examples of what is possible here. In 2007, the US National Science Foundation, recognizing the importance of such standards, established the Community Based Data Interoperability Networks (INTEROP) funding programme for the development of tools, standards, and data management best practices within specific disciplinary communities. ... Although many classes of scientific data aren't ready, or aren't appropriate, for standardization, well chosen investments in standardization show a consistently high pay-off." (Lynch, 2008)

"For certain types of important digital objects, there are well-curated, deeply-integrated, special-purpose repositories such as Genbank, Worldwide Protein Data Bank, and UniProt... However, not all datasets or even data types can be captured by, or submitted to, these repositories. Many important datasets emerging from traditional, low-throughput bench science don't fit in the data models of these special-purpose repositories, yet these datasets are no less important with respect to integrative research, reproducibility, and reuse in general. Apparently in response to this, we see the emergence of numerous general-purpose data repositories [e.g., FigShare, Mendeley]. ... Such repositories accept a wide range of data types in a wide range of formats, generally do not attempt to integrate or harmonize the distributed data, and place few restrictions (or requirements) on the descriptors of the data deposition. The resulting data ecosystem, therefore, appears to be moving away from centralization, is becoming more diverse, and less integrated, thereby exacerbating the discovery and re-usability problem for both human and computational stakeholders." (Wilkinson et al., 2016)

"It would be unwise to bet that these formats [SAM/BAM files] won't change (or even be replaced at some point)—the field of bioinformatics is notorious for inventing new data formats (the same goes with computing in general) ... So learning how to work with specific bioinformatics formats may seem like a lost cause, skills such as following a format specification, manipulating binary files, extracting information from bitflags, and working with application programming interfaces (API) are essential skills when working with any format." (Buffalo, 2015)

From a working group on bioinformatics and data-intensive science: "Many simple analyses are not automated because data formats are a moving target. ... The community has been slow to share tools, partially because tools are not robust against different input formats." (Barga et al., 2011)

"Different centres generate data in different formats, and some analysis tools require data to be in particular formats or require different types of data to be linked together. Thus, time is wasted reformatting and reintegrating data multiple times during a single analysis. For example, next-generation sequencing companies do not deliver raw sequencing data in a format common to all platforms, as there is no industry-wide standard beyond simple text files that include the nucleotide sequence and the corresponding quality values. As a result, carrying out sequencing analyses across different platforms requires tools to be adapted to

specific platforms. It is therefore crucial to develop interoperable sets of analysis tools that can be run on different computational platforms depending on which is best suited for a given application, and then stitch those tools together to form analysis pipelines.” (Schadt et al., 2010)

“Many important datasets emerging from traditional, low-throughput bench science don’t fit in the data models of … special-purpose repositories [like Genbank, Worldwide Protein Data Bank, and UniProt], yet these datasets are no less important with respect to integrative research, reproducibility, and reuse in general. Apparently in response to this, we see the emergence of numerous general-purpose data repositories [e.g., FigShare, Mendeley]. … Such repositories accept a wide range of data types in a wide range of formats, generally do not attempt to integrate or harmonize the distributed data, and place few restrictions (or requirements) on the descriptors of the data deposition. The resulting data ecosystem, therefore, appears to be moving away from centralization, is becoming more diverse, and less integrated, thereby exacerbating the discovery and re-usability problem for both human and computational stakeholders.” (Wilkinson et al., 2016)

“Simplicity, but not oversimplification, is the key to success [in developing standards].” (Brazma et al., 2006)

“Minimum reporting guidelines, terminologies, and formats (hereafter referred to as reporting standards) are increasingly used in the structuring and curation of datasets, enabling data sharing to varying degrees. However, the mountain of frameworks needed to support data sharing between communities inhibits the development of tools for data management, reuse and integration. … The same framework [on the other hand] enables researchers, bioinformaticians, and data managers to operate within an open data commons.” (Sansone et al., 2012)

“‘One of the core issues of Bioinformatics is dealing with a profusion of (often poorly defined or ambiguous) file formats. Some *ad hoc* simple human readable formats have over time attained the status of de facto standards.’— Peter Cock et al. (2010)” (Buffalo, 2015)

“Developing and using a standard is often an investment that will not pay off immediately, therefore there is a much better chance of success if the user community decides that the respective standard is needed.” (Brazma et al., 2006)

“Although standardization is not a goal in itself, its importance is growing in a high-throughput era. This is similar to what happened to manufacturing during industrialization. The data from high-throughput technologies are being generated at a rate that makes managing and using these data sets impossible on a case-by-case basis. Although some of the data generated by the newest technologies might have a low signal-to-noise ratio to make data re-usable, the data quality is improving as the technology matures, and it is a waste of resources not to share and re-use these expensive datasets. However, this is only possible if the instrumentation that generates these data, laboratory-based storage information management systems and databases, data analysis tools, and systems modeling software can talk to each other easily. This is the purpose of standardization.” (Brazma et al., 2006)

“A standard is successful only if it is used, and it is important to ensure that supporting software tools are designed and implemented.” (Brazma et al., 2006)

"In the late 2000s, there arose the 'NoSQL movement', coalescing around a collective desire of many programmers to move beyond the strictures of the relational model and unshackle themselves from SQL. *Our data varied and diverse, they said, even if programmers weren't that varied and diverse, and we are tired of pretending that one technology will address the need for speed.* Dozens of new databases appeared, each with different merits. There were key-value databases, like Kyoto Cabinet, which optimized for speed of retrieval. There were search-engine libraries, like Apache Lucene, which made it relatively easy to search through enormous corpora of text—your own Google. There was Mongo DB, which allowed for 'documents', big arbitrary blobs of data, to be stored without nice rows and consistent structure. People debated, and continue to debate, the value of each. ... There is as yet no absolute challenger to the relationship model. When people think *database*, they still think *SQL*." (Ford, 2015)

"By information or data communication standard we mean a convention on how to encode data or information about a particular domain (such as gene function) that enables unambiguous transfer and interpretation of this information or data." (Brazma et al., 2006)

"The proper acquisition and handling of data is crucially important for both the generation and verification of hypotheses. The rapid development of high-throughput experimental techniques is transforming life-science research into 'big data' science, and although numerous data-management systems exist, the heterogeneity of formats, identifiers, and data schema pose serious challenges. In this context, data-management systems need standardized formats for data exchange, globally unique identifiers for data mapping, and common interfaces that allow the integration of disparate software tools in a computational workflow." (Ghosh et al., 2011)

"Data quality [for health registries data] is driven by multiple dimensions such as clinical data standardization, the existence of common definitions of data fields, and the validity of self-reported patient conditions and outcomes. Recognized issues include the definitions of data fields and their relational structure, the training of personnel related to data collection data processing issues (data cleaning), and curation." (Keller et al., 2017)

If you have data in a structured, tabular format that doesn't follow these rules, you don't need to consider it "dirty", though—just think of "tidy" as the tagname for this particular structure of data (the name, in this case, connects the data format with a set of tools in R called the "tidyverse").

"Software systems are transparent when they don't have murky corners or hidden depths. Transparency is a passive quality. A program is passive when it is possible to form a simple mental model of its behavior that is actually predictive for all or most cases, because you can see through the machinery to what is actually going on." (Raymond, 2003)

"Software systems are discoverable when they include features that are designed to help you build in your mind a correct mental model of what they do and how they work. So, for example, good documentation helps discoverability to a programmer. Discoverability is an active quality. To achieve it in your software, you cannot merely fail to be obscure, you have to go out of your way to be helpful." (Raymond, 2003)

"Elegant code does much with little. Elegant code is not only correct but visibly, transparently correct. It does not merely communicate an algorithm to a computer, but also conveys insight and assurance to the mind of a human that reads it. By seeking elegance in our code, we build better code. Learning to write transparent code is a first, long step toward learning how to write elegant code—and taking care to make code discoverable helps us learn how to make it transparent. Elegant code is both transparent and discoverable." (Raymond, 2003)

"To design for transparency and discoverability, you need to apply every tactic for keeping your code simple, and also concentrate on the ways in which your code is a communication to other human beings. The first questions to ask, after 'Will this design work?' are 'Will it be readable to other people? Is it elegant?' We hope it is clear ... that these questions are not fluff and that elegance is not a luxury. These qualities in the human reaction to software are essential for reducing its bugginess and increasing its long-term maintainability." (Raymond, 2003)

"The Unix style of design applies the do-one-thing-well approach at the level of cooperating programs as well as cooperating routines within a program, emphasizing small programs connected by well-defined interprocess communication or by shared files. Accordingly, the Unix operating system encourages us to break our programs down into simple subprocesses, and to concentrate on the interfaces between these subprocesses." (Raymond, 2003)

"The ability to combine programs [with piping] can be extremely useful. But the real win here is not cute combinations; it's that because both pipes and *more()* exist, *other programs can be simpler*. Pipes mean that programs like *ls()* (and other programs that write to standard out) don't have to grow their own pagers—and we're saved from a world of a thousand built-in pagers (each, naturally, with its own divergent look and feel). Code bloat is avoided and global complexity reduced. As a bonus, if anyone needs to customize pager behavior, it can be done in one place, by changing one program. Indeed, multiple pagers can exist, and will all be useful with every application that writes to standard output." (Raymond, 2003)

"Unix was born in 1969 and has been in continuous production use ever since. That's several geological eras by computer industry standards. ... Unix's durability and adaptability have been nothing short of astonishing. Other technologies have come and gone like mayflies. Machines have increased a thousand-fold in power, languages have mutated, industry practice has gone through multiple revolutions—and Unix hangs in there, still producing, still paying the bills, and still commanding loyalty from many of the best and brightest software technologists on the planet." (Raymond, 2003)

"One of the many consequences of the exponential power-versus-time curve in computing, and the corresponding pace of software development, is that 50% of what one knows becomes obsolete over every 18 months. Unix does not abolish this phenomenon, but does do a good job of containing it. There's a bedrock of unchanging basics—languages, system calls, and tool invocations—that one can actually keep for entire years, even decades. Elsewhere it is impossible to predict what will be stable; even entire operating systems cycle out of use. Under Unix, there is a fairly sharp distinction between transient knowledge and lasting

knowledge, and one can know ahead of time (with about 90% certainty) which category something is likely to fall in when one learns it. Thus the loyalty Unix commands.” (Raymond, 2003)

“Unix is famous for being designed around the philosophy of small, sharp tools, each intended to do one thing well. This philosophy is enabled by using a common underlying format—the line-oriented, plain text file. Databases used for system administration (users and passwords, network configuration, and so on) are all kept as plain text files. … When a system crashes, you may be faced with only a minimal environment to restore it (you may not be able to access graphics drivers, for instance). Situations such as this can really make you appreciate the simplicity of plain text.” (Hunt et al., 2000)

“Unix is the foundational computing environment in bioinformatics because its design is the antithesis of [a] inflexible and fragile approach. The Unix shell was designed to allow users to easily build complex programs by interfacing smaller modular programs together. This approach is the Unix philosophy: ‘This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.’—Doug McIlroy”. (Buffalo, 2015)

“Passing the output of one program directly into the input of another program with pipes is a computationally efficient and simple way to interface Unix programs. This is another reason why bioinformaticians (and software engineers in general) like Unix. Pipes allow us to build larger, more complex tools from modular parts. It doesn’t matter what language a program is written in, either; pipes will work between anything as long as both programs understand the data passed between them. As the lowest common denominator between most programs, plain-text streams are often used—a point that McIlroy makes in his quote about the Unix philosophy.” (Buffalo, 2015)

If the data is the same regardless of whether it’s “tidy” or not, then why all the fuss about following the “tidy” principles when you’re designing the format you’ll use to record your data? The magic here is this—if you follow these principles, then your data can be immediately input into a collection of powerful tools for visualizing and analyzing the data, without further cleaning steps. What’s more, all those tools (the set of tools is called the “tidyverse”) will typically *output* your data in a “tidy” format, as well.

These small tools can be combined together because they take the same input (data in a “tidy” format) and they output in the same format (also data in a “tidy” format). This is such a powerful idea that many of the best loved toys work on the same principle. Think of interlocking plastic block sets, like Lego. You can create almost anything with a large enough set of Legos, because they can be combined in almost any kind of way. Why? Because they all follow a standard size for the ... on top of each block, and they all “input” ... of that same size on the bottom of the block. That means they can be joined together in any order and combination, and as a result very complex structures can be created. It also means that each piece can be small and easy to understand—if you’re building a Lego structure, even something very fancy, you’ll probably use

lots of rectangular bricks that are two ... across and four ... long, and that's easy enough to describe that you could probably get a young child to help you find those pieces when you need them.

The “tidy” data format is an implementation of a structured data format popular among statisticians and data scientists. By consistently using this data format, researchers can combine simple, generalizable tools to perform complex tasks in data processing, analysis, and visualization.

“Base R graphics came historically first: simple, procedural, conceptually motivated by drawing on a canvas. There are specialized functions for different types of plots. These are easy to call—but when you want to combine them to build up more complex plots or exchange one for another, this quickly gets messy, or even impossible. The user plots ... directly onto a (conceptual) canvas. She explicitly needs to deal with decisions such as how much space to allocate to margins, axis labels, titles, legends, subpanels; once something is ‘plotted’, it cannot be moved or erased. There is a more high-level approach: in the *grammar of graphics*, graphics are built up from modular logical pieces, so that we can easily try different visualization types for our data in an intuitive and easily deciphered way, just as we can switch in and out parts of a sentence in human language. There is no concept of a canvas or a plotter; rather, the user gives `ggplot2` a high-level description of the plot she wants, in the form of an R object, and the rendering engine takes a holistic view of the scene to lay out the graphics and render them on the output device.” (Holmes and Huber, 2018)

---

### **Older text**

It is usually very little work to record data in a structure that follows the “tidy data” principles, especially if you are planning to record the data in a two-dimensional, tabular format already, and following these principles can bring some big advantages. We explain these rules and provide examples of biomedical datasets that both comply and don’t comply with these principles, to help make it clearer how you could structure a “tidy-compliant” structure for recording experimental data for your own research.

If the data is the same regardless of whether it’s “tidy” or not, then why all the fuss about following the “tidy” principles when you’re designing the format you’ll use to record your data? The magic here is this—if you follow these principles, then your data can be immediately input into a collection of powerful tools for visualizing and analyzing the data, without further cleaning steps (as discussed in the previous module). What’s more, all those tools (the set of tools is called the “tidyverse”) will typically *output* your data in a “tidy” format, as well.

Once you have tools that input and output data in the same way, it becomes very easy to model each of the tools as “small, sharp tools”—each one does one thing, and does it really well. That’s because, if each tool needs the same type of input and creates that same type of output, those tools can be chained together to solve complex problems. The alternative is to create large software

tools, ones that do a lot to the input data before giving you some output. “Big” tools are harder to understand, and more importantly, they make it hard to adapt your own solutions, and to go beyond the analysis or visualization that the original tool creators were thinking of when they created it. Think of it this way—if you were writing an essay, how much more can you say when you can mix and match words to create your own sentences versus if you were made to combine pre-set sentences?

#### *4.1.1 Creating the rules for collecting data in the same time each time*

It is likely that there are certain types of experiments that you conduct regularly, and that they’re often trying to answer the same type of question and generate data of a consistent type and structure. This is a perfect chance to lay down rules or a pattern for how members of your research group will record that data.

These rules can include:

1. How many units of observation does the experiment typically have? Say, for example, that you are measuring the influence of two drugs on bacterial load in an animal at three time points. There may be some measurements taken at the unit of the drug (for example, measurements related to its chemical composition) and some taken at the unit of animal and time point (for example, the concentration of drug in an animal’s blood at a certain time point). This will help you define how many tables you should use to collect the data—one for each unit of observation.
2. Which measurements will be recorded for each observation? In tidy data, the measurements taken for an observation are recorded in rows, so you then specify what column names should be used for each measurement (e.g., “sample\_time”, “animal\_weight”). If data is being recorded using multiple tables (because there are multiple units of observation), make sure that each table include “ID” columns that can be used to link across the tables. For example, each table might have a column with a unique ID for each drug being tested, or tables with measurements on animals might each have a column that uniquely identifies the animal in an observation.
3. What units will be used for recording each measurement? For timestamp-type measurements, like the date and time that an experiment started and the time of each sample measurement, what timezone will be used? (Even if it’s always the same one, this can come in useful every now and then if you need to figure out something like whether that location’s timezone followed Daylight Savings Time, for an experiment that spans the switch between Standard and Daylight Savings).

[Figure: Three tables—measurements on a drug (chemistry), measurements on an animal (weight), measurements on an animal at time points (drug concentration)]

You can then take this information and design a template for collecting that type of data. A template is, in this case, a file that gives the “skeleton” of the table or tables. You will create this template file and save it somewhere easy for lab members to access, with a filename that makes it clear that this is a template. For example, you may create a folder with all the templates for tables for your experiment, and name a template in it for collecting something like animal weights at the start of the experiment something like “animal\_wt\_table\_template.csv” or “animal\_wt\_table\_template.xlsx”. Each time someone starts an experiment collecting that type of data, he or she can copy that template file, move it to the directory with files for that experiment and rename it. When you open that copy of the file, you can record observations directly into it.

### [Figure: Example template file]

```
#####
Column names and meanings
#
animal_id: A unique identifier for each animal.
animal_wt_g: The weight of the animal, recorded in grams.
date_wt_measured: The date that the animal's weight was measured, recorded as
"month day, year", e.g., "Jan 1, 2019"
cage_id: A unique identifier for the case in which the animal was housed
#
Other table templates for this experiment type:
drug_conc_by_time.csv: A template for recording drug concentrations in the animals
by time point
#
animal_id, animal_wt_g, date_wt_measured, cage_id
"A101", 50.2, "Jan 1, 2019", "B"
```

Adding in one row of sample values, to be deleted each time the template is copied and used, can be a very helpful addition. This will help the user remember the formats that are expected for each column (for example, the format the date should be recorded in), as well as small details like which columns should include quotation marks.

These template tables can be created as flat files, like comma-separated value files. However, if this is too big of a jump, they can also be created as spreadsheet files. Many of the downsides of spreadsheet files are linked to the use of embedded macros, integration of raw and processed / calculated data, and other factors, rather than related to their use as a method to record data. However, do note that plain text files like flat files can be opened in RStudio in a spreadsheet-like view in RStudio. Data can be recorded directly here, in a format that will feel comfortable for spreadsheet users, but without all the bells

and whistles that we're aiming to avoid in spreadsheet programs like Excel.

[Figure—Opening a csv file with a spreadsheet like view]

There are some advantages to shifting to record data in flat files like CSVs, rather than Excel files, and using the spreadsheet-style view in RStudio to work with those files if you find it easier than working with the files in a text editor (which can get tough, since the values in a column don't always visually line up, and you have to remember to put in the right number of columns). By recording the data in a plain text file, you can later move to tracking changes that are made to the data using the version control tool *git*. This is a powerful tool that can show who made changes to a file and when, with exact details on the changes made and room for comments on why the change was made. However, *git* does not provide useful views of changes made to binary files (like Excel), only those made in plain text files. Further, plain text files are guaranteed to not try to “outsmart” you—for example, they will not try to convert something that looks like a date into a date. Instead, they will leave things exactly as you typed them in. Finally, later in this book we will build up to creating templates that do even more—for example, templates for reports you need to write and presentations you need to give, as well as templates for the whole structure of a project. Plain text files fit very nicely into this developing framework, while files in complex binary formats like xlxs don't fit as naturally.

Google Sheets is another tool that might come in useful. [More about using this with R.]

This idea of creating template files for data recording isn't revolutionary—many laboratory groups have developed spreadsheet template files that they share and copy to use across similar experiments that they conduct. The difference here is in creating a table for recording data *that follows the tidy data principles*, or at least comes close to them (any steps away from characteristics like embedded macros and use of color to record information will be helpful).

The next chapter will walk through two examples of changing from non-tidy table templates to ones that record data in a way that follows the tidy data principles.

#### 4.1.2 Subsection 1

“Or maybe your goal is that your data is *usable* in a wide range of applications? If so, consider adopting standard formats and metadata standards early on. At the very least, keep track of versions of data and code, with associated dates.” (Goodman et al., 2014)

“Standards for data include, for example, data formats, data exchange protocols, and meta-data controlled vocabularies.” (Barga et al., 2011)

“Software systems are transparent when they don't have murky corners or hidden depths. Transparency is a passive quality. A program is passive when it is possible to form a simple mental model of its behavior that is actually predictive for all or most cases, because you can see through the machinery to what is actually going on.” (Raymond, 2003)

"Software systems are discoverable when they include features that are designed to help you build in your mind a correct mental model of what they do and how they work. So, for example, good documentation helps discoverability to a programmer. Discoverability is an active quality. To achieve it in your software, you cannot merely fail to be obscure, you have to go out of your way to be helpful." (Raymond, 2003)

"Elegant code does much with little. Elegant code is not only correct but visibly, *transparently* correct. It does not merely communicate an algorithm to a computer, but also conveys insight and assurance to the mind of a human that reads it. By seeking elegance in our code, we build better code. Learning to write transparent code is a first, long step toward learning how to write elegant code—and taking care to make code discoverable helps us learn how to make it transparent. Elegant code is both transparent and discoverable." (Raymond, 2003)

"To design for transparency and discoverability, you need to apply every tactic for keeping your code simple, and also concentrate on the ways in which your code is a communication to other human beings. The first questions to ask, after 'Will this design work?' are 'Will it be readable to other people? Is it elegant?' We hope it is clear ... that these questions are not fluff and that elegance is not a luxury. These qualities in the human reaction to software are essential for reducing its bugginess and increasing its long-term maintainability." (Raymond, 2003)

"Software is maintainable to the extent that people who are not its author can successfully understand and modify it. Maintainability demands more than code that works; it demands code that follows the Rule of Clarity and communicates successfully to human beings as well as the computer." (Raymond, 2003)

"An equivalent to the laboratory notebook that is standard good practice in lab-work, we advocate the use of a computational diary written in the R markdown format. ... Together with a version control system, R markdown helps with tracking changes." (Holmes and Huber, 2018)

"R.A. Fisher, one of the fathers of experimental design, is quoted as saying 'To consult the statistician after an experiment is finished is often merely to ask him to conduct a post mortem examination. He can perhaps say what the experiment died of.' So it is important to design an experiment with the analysis already in mind. Do not delay thinking about how to analyze the data until after they have been acquired. ... Dailies: start with the analysis as soon as you have acquired some data. Don't wait until everything is collected, as then it's too late to troubleshoot. ... Start writing the paper while you're analyzing the data. Only once you're writing and trying to present your results and conclusions will you realize what you should have done properly to support them." (Holmes and Huber, 2018)

"In the same way a file director will view daily takes to correct potential lighting or shooting issues before they affect too much footage, it is a good idea not to wait until all runs of an experiment have been finished before looking at the data. Intermediate data analyses and visualizations will track unexpected sources of variation and enable you to adjust the protocol. Much is known about the sequential design of experiments, but even in a more pragmatic setting it is important to be aware of your sources of variation as they occur and adjust for them." (Holmes and Huber, 2018)

“Analysis projects often begin with a simple script, perhaps to try out a few initial ideas and explore the quality of the pilot data. Then more ideas are added, more data come in, other datasets are integrated, more people become involved. Eventually the paper need to be written, the figures need to be done ‘properly’ and the analysis needs to be saved for the scientific record and to document its integrity.” (Holmes and Huber, 2018)

**“Use literate programming tools.** Examples are Rmarkdown and Jupyter. This makes code more readable (for yourself and for others) than burying explanations and usage instructions in comments in the source code or in separate README files. In addition, you can directly embed figures and tables in these documents. Such documents are good starting points for the supplementary material of your paper. Moreover, they’re great for reporting analyses to your collaborators.” (Holmes and Huber, 2018)

#### 4.1.3 Don’t Repeat Yourself!

One of the core tenets of programming is the philosophy of “Don’t Repeat Yourself” (a.k.a., the “DRY Principle”).[Source of “Don’t Repeat Yourself”—*The Pragmatic Programmer*] With programming, you can invest a little bit of time to code your computer to do things that take a lot of your time otherwise. In this way, you can automate repetitive tasks.

“The DRY principle, for Don’t Repeat Yourself, is one of the colloquial tenets of programming. That is, you should name things once, do things once, create a function once, and let the computer repeat itself.” [ford2015code]

“Code, in other words, is really good at making things scale. Computers may require utterly precise instructions, but if you get the instructions right, the machine will tirelessly do what you command over and over and over again, for users around the world. … Solve a problem once, and you’ve solved it for everyone.” [Coders, p. 20]

“Since they have, at their beck and call, machines that can repeat instructions with robotic perfection, coders take a dim view of doing things repetitively themselves. They have a dislike of inefficiency that is almost aesthetic—they recoil from it as if from a disgusting smell. Any opportunity they have to automate a process, to do something more efficiently, they will.” [Coders, p. 20]

“Programmers are obsessed with efficiency. … Removing the friction from a system is an aesthetic joy; [programmers’] eyes blaze when they talk about making something run faster, or how they eliminated some bothersome human effort from a process.” [Coders, p. 122]

“Computers, in many ways, inspire dreams of efficiency greater than any tool that came before. That’s because they’re remarkably good at automating repetitive tasks. Write a script once, set it running, and the computer will tirelessly execute it until it dies or the power runs out. What’s more, computers are strong in precisely the ways that humans are weak. Give us a repetitive task, and our mind tends to wander, so we gradually perform it more and more irregularly. Ask us to do something at a precise time or interval, and we space out and forget to do it. … In contrast, computers are clock driven and superb at doing the same thing at the same time, day in and day out.” [Coders, p. 124]

"Larry Wall, the famous coder and linguist who created the Perl programming language, deeply intuited this coderly aversion to repetition. In his book on Perl, he and coauthors wrote that one of the key virtues of a programmer is 'laziness'. It's not that you're too lazy for coding. It's that you're too lazy to do routine things, so it inspires you to automate them." [Coders p. 126]

In scientific research, there are a lot of these repetitive tasks, and as tools for automation continue to develop, there are many opportunities to "automate away" busywork.

"Science often involves repetition of computational tasks such as processing large number of data files in the same way or regenerating figures each time new data are added to an existing analysis. Computers were invented to do these kinds of repetitive tasks but, even today, many scientists type the same commands in over and over again or click the same buttons repeatedly." [wilson2014best]

"Whenever possible, rely on the execution of programs instead of manual procedures to modify data. Such manual procedures are not only inefficient and error-prone, they are also difficult to reproduce." [sandve2013ten]

"Other manual operations like the use of copy and paste between documents should also be avoided. If manual operations cannot be avoided, you should as a minimum note down which data files were modified or moved, and for what purpose." [sandve2013ten]

Statisticians have been doing this for a while for data cleaning analysis tasks. For example, if you need to read in an Excel file into a statistical programming language like R, you could write a few lines of code to do that anew each time you get a new file. However, say you get Excel files over and over that follow the same format—for example, files with the same number of columns, the same names for those columns, and the same type of data. You can write a *script*—a recorded file with a few lines of code, in this case—that reads in the file. You can apply this script to each new file.

This saves you a little bit of time. It also ensures that you do the exact same thing with every file you get. It also means that you can reproduce what you do now to a file in the future. Say, for example, that you are working on a project and you read in a file and conduct an analysis. Your laboratory group sends the paper out for review. Months later, you get back comments from the reviewers, and they are wondering what would happen if you had analyzed the data a bit differently—say, used a different statistical test. If you use a script to read in the data file, then when you re-run it to address the reviewers' comments, you can be sure that you are getting your data into the statistical program in the exact same way you did months ago, and so you're not unintentionally introducing differences in your results because you are doing some small things differently in processing the file.

This idea can extend across the full data analysis you do on a project. You are only saving a little bit of time and effort, maybe, by automating the step where you read the data from a spreadsheet into the statistical program. And

it takes some time to write that script the first time, so it can be tempting to do it fresh each time you need to do it. However, you can also write scripts that will automate cleaning your data. Maybe you want to identify data points with very high (maybe suspect) values for a certain measurement, or remove observations with missing data. You can also write scripts that will automate processing your data—doing things like calculating the time since the start of an experiment based on the recorded sampling time for an observation. Each of these steps might be small, but the time saved really adds up since you typically need to perform many of these steps each time you run a new experiment.

There are many cases in life where you'll need to make the choice between spending some time upfront to make something more efficient, versus doing it more quickly the first time but then having to do it "from scratch" again each following time. For example, say that you're teaching a class, and you need to take attendance for each class period. You could write down the names of each student at the first class and save that, and then the next class write down the name of each student who shows up that day on a separate sheet of paper, and so on for each class meeting. Conversely, you could take some extra time before the first class and create a table or spreadsheet file with every student's name and the date of each class, and then use that to mark attendance. The first method will be quicker the first day, but more time consuming each following time. The second method requires a small initial investment, but with time-saving returns in the following class meetings.

For people who use scripts and computer programs to automate their data-related tasks, it quickly becomes very confusing how anyone who doesn't could argue that they don't because they don't have time to learn how to. The amount of time you end up saving based on your initial investment is just so high if you're working with data, that it would have to take a huge time investment to not be worth it. Plus—the thrill of running something that you've automated! It's a very similar feeling to the feeling you get when a student or postdoc that you've spent a lot of time training has gotten to the point where you can just ask them to run something, and they do, and it means you don't have to.

Here are some of the problems that are solved by automating your small tasks:

1. **It gets done the same way every single time.** Even simple tasks can be done with numerous small modifications. You will probably remember some of those choices and settings and modifications the next time you need to do the same thing, but probably not all of them, and so those choices will not be exact from one time to the next. If the computer is doing it based on a clear set of instructions, it will be.
2. **It gets done more quickly.** Or if not more quickly (some large data might take some time to process), at least the spent time is the computers time, not yours. You can leave the computer to run the script while you get on

with other things that a computer can't do.

3. **Anyone who does it can do it the same way.** Just as you might not do something exactly the same way from one time to the next, one person in a laboratory group is likely to do things at least slightly different than other members of the group. Even with very detailed instructions, few instructions written for humans can be so detailed and precise to ensure that something is done exactly the same way by everyone who follows them. If everyone is given the same computer script to run, however, and they all instruct the computer to run that script, the task will be done in exactly the same way.
4. **It is easier teach new people how to do the task.** Often, with a script to automate a task, you just need to teach someone new to the laboratory group how to get the computer to run a script in a certain language. When you need them to run a new script, the process will be the same. The script encapsulates all the task-specific details, and so the user doesn't need to understand all of them to get something to run. What's more, once you want to teach a new lab member how everything is working, so they can understand the full process, the script provides the exact recipe. You can teach them how to read and understand scripts in that language, and then the scripts you've created to automate tasks serve as a recipe book for everything going on in terms of data analysis for the lab.
5. **You can create tools to share with others.** If you've written a script that's very useful, with a bit more work you can create it into a tool that you can share with other research groups and perhaps publish a paper about. Papers about R software extensions (also called *packages*) and data analysis workflows and pipelines are becoming more and more common in biological contexts.
6. **It's more likely to be done correctly.** Boring, repetitive tasks are easy to mess up. We get so bored with them, that we shift our brains into a less attentive gear when we're working on them. This can lead to small, stupid mistakes, ones at the level of typos but that, with data cleaning and analysis, can have much more serious ramifications.

"We view workflows as a paradigm to: 1) expose non-experts to well-understood end-to-end data analysis processes that have proven successful in challenging domains and represent the state-of-the-art, and 2) allow non-experts to easily experiment with different combinations of data analysis processes, represented as workflows of computations that they can easily reconfigure and that the underlying system can easily manage and execute." [hauder2011making]

"While reuse [of workflows] by other expert scientists saves them time and effort, reuse by non-experts is an enabling matter as in practice they would not be able to carry out the analytical tasks without the help of workflows." [hauder2011making]

"We observed that often steps that could be easily automated were performed manually in an error-prone fashion." [vidger2008supporting]

Biological research is quickly moving where a field where projects often required only simple and straightforward data analysis once the experimental data was collected—with the raw data often published directly in a table in the manuscript—to a field with very complex and lengthy data analysis pipelines between the experiment and the final manuscript. To ensure rigor and clarity in the final research results, as well as to allow others to reproduce the results exactly, the researcher must document all details of the computational data analysis, and this is often missing from papers. RMarkdown documents (and their analogues) can provide all these details unambiguously—with RMarkdown documents, you can even run a command to pull out all the code used within the document, if you'd like to submit that code script as a stand-alone document as a supplement to a manuscript.

"More recently, scientists who are not themselves computational experts are conducting data analysis with a wide range of modular software tools and packages. Users may often combine these tools in unusual or new ways. In biology, scientists are now routinely able to acquire and explore data sets far beyond the scope of manual analysis, including billions of DNA bases, millions of genotypes, and hundreds of thousands of RNA measurements. ... While propelling enormous progress, this increasing and sometimes 'indirect' use of computation poses new challenges for scientific publication and replication. Large datasets are often analyzed many times, with modifications to the methods and parameters, and sometimes even updates of the data, until the final results are produced. The resulting publication often gives only scant attention to the computations details. Some papers have suggested these papers are 'merely the advertisement of scholarship whereas computer programs, input data, parameter values, etc., embody the scholarship itself.' However, the actual code or software 'mashup' that gave rise to the final analysis may be lost or unrecoverable." [mesirov2010accessible]

"Bioinformatic analyses invariably involve shepherding files through a series of transformations, called a pipeline or workflow. Typically, these transformations are done by third-part executable command line software written for Unix-compatible operating systems. The advent of next-generation sequencing (NGS), in which millions of short DNA sequences are used as the source input for interpreting a range of biological phenomena, has intensified the need for robust pipelines. NGS analyses tend to involve steps such as sequence alignment and genomic annotation that are both time-intensive and parameter-heavy."

[leipzig2017review]

**"Rule 7: Always Store Raw Data behind Plots.** From the time a figure is first generated to it being part of a published article, it is often modified several times. In some cases, such modifications are merely visual adjustments to improve readability, or to ensure visual consistency between figures. If raw data behind figures are stored in a systematic manner, so as to allow raw data for a given figure to be easily retrieved, one can simply modify the plotting procedure, instead of having to redo the whole analysis. An additional advantage of this is that if one really wants to read fine values in a figure, one can consult the raw

numbers. ... When plotting is performed using a command-based system like R, it is convenient to also store the code used to make the plot. One can then apply slight modifications to these commands, instead of having to specify the plot from scratch." [sandve2013ten]

#### 4.1.4 Don't repeat your report-writing!

Until a few years ago, statisticians and data analysts frequently automated the data cleaning, processing, and analysis tasks. But that still left the paper and report writing to be done by hand. This process is often repetitive. You would do your analysis and create some tables or figures. You would save these from your statistical program and then paste them into your report or paper draft. If you decided that you needed to change your analysis a bit, or if you got a new set of data to analyze in a similar way, you had to go back to the statistical program, run things again there, save the tables and figure files again, and paste them in the report or paper again to replace the outdated version. If there were numbers from the analysis in the text of the paper, then you had to go back through the text and update all of those with the newer numbers, too.

Do you still write your papers and reports like this? I can tell you that there is now a *much* better way. Computer scientists and other programmers started thinking quite a while ago about how to create documents that combine computer code and text for humans, and to do it in a way where the computer code isn't just a static copy of what someone once told the computer to do, but instead a living, working, executable set of instructions that the computer can run anytime you ask it to.

These ideas first percolated with Donald Knuth, who many consider to be the greatest computer programmer of all time [Bill Gates, for example, has told anyone who reads Dr. Knuth's magnum opus, *The Art of Computer Programming*, to come see him right away about a job]. As Dr. Knuth was writing a book on computer programming, he became frustrated with the quality of the typesetting used in the final book. In a field that requires a lot of mathematical and other symbols incorporated into the text, it takes a bit more to make an attractive book than with simpler text. Dr. Knuth therefore took some time to create a programming for typesetting. (You may have heard of it—if you ever notice that a journal's Instructions to Authors allow authors to submit articles in "LaTeX" or "TeX", that's using a system built off of Donald Knuth's typesetting program.)

And then, once he had that typesetting program, he started thinking about how programmers document their code. When one person does a very small code project, and that one person is the only person who will ever go back to try to modify or understand the code, that person might be able to get away with poor documentation in the code. However, interesting code projects can become enormous, with many collaborators, and it becomes impossible to understand and improve the code if it doesn't include documentation explaining, in human terms, what the code is doing at each step, as well as some overall

documentation explaining how different pieces of the code coordinate to get something big done.

Traditionally, code was documented by including small comments within the code. These comments are located near the code that they explain, and the order of the information in the code files are therefore dominated by the order of the instructions to the computer, not the order that you might explain what's going on to a human. To "read" the code and the documentation, you end up hopscotching through the code, following the code inside one function when it calls another function, for example, to where the code for that second function is defined and then back to the first, and so on. You often follow paths as you get deeper and deeper into helper functions and the helper functions for those functions, that you feel like you're searching through a set of Russian dolls and then coming back up to start on a new set of Russian dolls later down the line.

Donald Knuth realized that, with a good typesetting program that could itself be programmed, you could write your code so that the documentation for humans took precedence, and could be presented in a very clear and attractive final document, rather than hard-to-read computer code with some plain-text comments sprinkled in. Computers don't care what order the code is recorded in—as long as you give them some instructions on how to decipher code in a certain format or order, they can figure out how to use it fine. But human brains are a bit more finicky, and we need clear communication, laid out in a logical and helpful order. Donald Knuth created a paradigm of *literate programming* that interleaved executable code inside explanations written for humans; by making the code executable, it meant that the document was a living guide. When someone changed the program, they did it by changing the documentation—documentation wasn't left as the final, often neglected, step to refine once the "real code" was written (and the "real work" done).

"Programs must be written for people to read, and only incidentally for machines to execute. A great program is a letter from current you to future you or the person who inherits your code. A generous humanistic document."  
[ford2015what]

Well, this was a fantastic idea. It hasn't been universally leveraged, but the projects that do leverage it are much stronger for it. But that's not where the story ends. If you are someone who does a little bit of coding (maybe small scripts to analyze and visualize your data, for example) and a lot of "documenting" of the results, and if you're not planning on doing a lot of large coding projects or creating software tools, it's not immediately how you'd use these literate programming ideas.

Well, there are many people who do a little bit of programming in service to a larger research project. While they are not creating software that needs classical software documentation, they do want to document the results that they get when they run their scripts, and they want to create reports and journal articles to share what they've found. Several people took the ideas behind

literate programming—as it's used to document large software projects—and leveraged it to create tools to automate writing in data-related fields.

[F. Leisch?] was the first to do this with the R programming language, with a tool called “Sweave” (“S”-“weave”, as R builds off of another programming language called “S” and Leisch’s program would “weave” together S / R code and writing). This used Donald Knuth’s typesetting program. It allowed you to write a document for humans (like a report or journal article) and to intersperse bits of code in the paper. You’d put each code piece in the spot in the paper where the text described what was going on or where you wanted the results that it generated for example, if you had a section in the Methods where you talked about removing observations that were outliers, you would add in the code that took out those outliers right there in the paper. And if you had a placed in the Results that talked about how your data were different between two experimental groups, you would add the code that generated the plot to show that right there in the paper.

To tell the computer how to tell between code and writing, you would add a little weird combination of text each time that you wanted to “switch” into code and then another one each time you wanted to switch back into writing for humans. (These combinations were so weird because that guaranteed that it was a combination you would probably never want to type otherwise, so you wouldn’t have a lot of cases of the computer getting confused between whether the combo meant to switch to code or whether it was just something that came up in the regular writing.) You’d send the document, code and writing and all, through R once you had it written up. R would ignore everything that wasn’t code. When it got to the code pieces, it would run them, and if the code created output (like a figure or table), it would “write” that into the document at that point in the text. Then you’d run the document through Donald Knuth’s typesetting program (or an extension of it), and the whole document would get typeset into an attractive final product (often a pdf, although you had some choices on the type of output).

This meant that you got very attractive final documents. It also meant that your data analysis code was well documented—it was “documented” by the very article or report you wrote based on it, because the code was embedded right there in the final product! It also meant that you could save a lot of time if you needed to go back and change some of your code later (or input a different or modified dataset). You just had to change that small piece of code or data input, and then essentially press a button to put everything together again, and the computer would re-write the whole report for you, with every figure and table updated. It even let you write small bits of computer code directly into the written text, in case you need to write something like “this study included 52 subjects”, where the “52” came from you counting up the number of rows in one of your datasets—if you later added three more subjects and re-ran the analysis with the updated dataset, the report would automatically change to read “this study included 55 subjects”.

Leisch's system is still out there, but another has been adopted much more widely, building on it. Yihui Xie started work on a program that tweaked and improved Leisch's Sweave program, creating something called "knitr" ("knit"- "R"—are you noticing a pattern in the names?). Xie's knitr program, along with its extensions, is now widely used for data analysis projects. What's more, it's grown to allow for larger or more diverse writing projects—this book, for example, is written using an extension called "bookdown", and extensions also exist for create blogs that include executable R code ("blogdown") and websites with documentation for R packages ("packagedown").

So now, let's put these two pieces together. We know that programmers love to automate small tasks, and we know that there are tools that can be used to "program" tasks that involve writing and reporting. So what does this mean if you frequently need to write reports that follow a similar pattern and start from similar types of data? If you are thinking like a code, it means that you can move towards automating the writing of those reports.

One of us was once talking to someone who works in a data analysis-heavy field, and she was talking about how much time she spends copying the figures that her team creates, based on a similar analysis of new data that's regularly generated, into PowerPoint presentations. So, for this weeks report, she's creating a presentation that shows the same analysis she showed last week, just with newer data. Cutting and pasting is an enormous waste of time—there are tools to automate this.

#### 4.1.5 Automating reports

First—think through the types of written reports or presentations you've created in the past year or two. Are there any that follow a similar pattern? Any that input the same types of data, but from different experiments, and then report the same types of statistics or plots for them? Are there Excel spreadsheets your lab uses that generate specific tables or plots that you often cut and paste for reports or presentations? Look through your computer file folders or email attachments if you need to—many of these might be small regular reports that are so regular that they don't pop right to mind. If you are creating documents that match any of these conditions, you probably have something ripe for converting to a reusable, automatable template.

"Think like Henry Ford; he saw that building cars was a repeatable process and came up with the moving assembly line method, revolutionizing production. You may not be building a physical product, but chances are you are producing something. ... Look for the steps that are nearly identical each time, so you can build your own assembly line." [rose2018dont]

... [Creating a framework for the report]

"Odds are, if you're doing any kind of programming, especially Web programming, you've adopted a framework. Whereas an SDK is an expression of a corporate philosophy, a framework is more like a product pitch. Want to save

time? Tired of writing old code? Curious about the next new thing? You use a graphics framework to build graphical applications, a Web framework to build Web applications, a network framework to build network servers. There are hundreds of frameworks out there; just about every language has one. A popular Web framework is Django, which is used for coding in Python. Instagram was bootstrapped on it. When you sit down for the first time with Django, you run the command ‘startproject’, and it makes a directory with some files and configuration inside. This is your project directory. Now you have access to libraries and services that add to and enhance the standard library.” [ford2015what]

One key advantage of creating a report template is that it optimizes the time of statistical collaborators. It is reasonable for a scientists with a couple of courses worth of statistical training to design and choose the statistical tools for simple and straightforward data analysis. However, especially as the biological data collected in experiments expands in complexity and size, a statistician can recommend techniques and approaches to draw more knowledge from the data and to appropriately handle non-standard features of the data. There is substantial work involved in the design of any data analysis pipeline that goes beyond the very basics. It waste time and resources to recreate this with each new project, time that—in the case of statistical collaborators—could probably be better spent in extending data analysis goals beyond the simplest possible analysis to explore new hypotheses or to add exploratory analysis that could inform the design of future experiments.

“Workflows effectively capture valuable expertise, as they represent how an expert has designed computational steps and combined them into an end-to-end process.” [hauder2011making]

When collaborative work between scientists and statisticians can move towards developing repeatable data analysis scripts and report templates, you will start to think more about common patterns and common questions that you ask across many experiments in your research program, rather than focusing on the immediate needs for a specific project. You can start to think of the data analysis tools that are general purpose for your research lab, develop those into clean, well-running scripts or functions, and then start thinking about more sophisticated questions you want to ask of your data. The statisticians you collaborate will be able to see patterns across your work and help to develop global, and perhaps novel, methods to apply within your research program, rather than piecemeal small solutions to small problems.

“Although foundational knowledge is taught in major universities and colleges, advanced data analytics can only be acquired through hands-on practical training. Only exposure to real-world datasets allows students to learn the importance of preparing and cleansing the data, designing appropriate features, and formulating the data mining task so that the data reveals phenomena of interest. However, the effort required to implement such complex multi-step data analysis systems and experiment with the tradeoffs of different algorithms and feature choices is daunting. For most practical domains, it can take weeks to months for a student

to setup the basic infrastructure, and only those who have access to experts to point them to the right high-level design choices will endeavor on this type of learning. As a result, acquiring practical data analytics skills is out of reach for many students and professionals, posing severe limitations to our ability as a society to take advantage of our vast digital data resources.” [hauder2011making]

“In practice designing an appropriate end-to-end process to prepare and analyze the data plays a much more influential role than using a novel classifier or statistical model.” [hauder2011making]

It is neither quick nor simple to design the data analysis plan and framework for a research experiment. It is not simply naming a statistical test or two. Instead, the data analyst must start by making sure they understand the data, how it was measured, how to decipher the format in which it’s stored, what questions the project is hoping to answer, where there might be problems in the data (and what they would look like), and so on. If a data analyst is helping with a lot of projects using similar types of data to answer similar questions, then he or she should, in theory, need less time for these “framework” types of questions and understanding. However, if data isn’t shared in the same format each time, it will still take overhead to figure out that this is indeed the same type of data and that code from a previous project can be adapted or repurposed.

Let’s think about one area where you likely repeat very similar steps frequently—writing up short reports or slide presentations to share your to-date research results with your research group or colleagues. These probably often follow a similar structure. For example, they may start with a section describing the experimental conditions, and then have a slide showing a table with the raw data (or a simple summary of it, if there’s a lot of data), and then have a figure showing something like the difference in experimental measurements between two experimental groups.

[Figure: Three simple slides for a research update—experimental conditions, table of raw data, boxplots with differences between groups.]

“The cornerstone of using DRY in your work life is the humble template. Whenever you create something, whether it’s an email, a business document, or an infographic, think if there’s something there you could save for future use. The time spent creating a template will save you exponentially more time down the road.” [rose2018dont]

You could start very simply in turning this into a template. You could start by creating a PowerPoint document called “lab\_report\_template.pptx”. It could include three slides, with the titles on each slide of “Experimental conditions”, “Raw data”, and “Bacterial burden by group”, and maybe with some template set to provide general formatting that you like (font, background color, etc.). That’s it. When you need to write a new report, you copy this file, rename the copy, and open it up. Now instead of needing to start from a blank PowerPoint file, you’ve shaved off those first few steps of setting up the pieces of the file you always use.

[Figure: Simplest possible template]

This very simple template won't save you much time—maybe just a minute or so for each report. However, once you can identify other elements that you commonly use in that type of report, you can add more and more of these “common elements” to the template, so that you spend less time repeating yourself with each report. For example, say that you always report the raw data using the same number of columns and the same names for those columns. You could add a table to that slide in your template, with the columns set with appropriate column names. You can always add or delete rows in the table if you need to in your reports, but now each time you create a new report, you save yourself the time it takes to create the table structure and add the column names. Plus, now you've guaranteed that the first table will use the exact same column names every time you give a report! You'll never have to worry about someone wondering if you are using a different model animal because you have a column named “Animal ID” in one report, while your last report had “Mouse ID”, for example. And because you're making a tool that you'll use many times, it becomes worthwhile to take some time double-checking the clean-up, so you're more likely to avoid things like typos in the slide titles or in columns names of tables.

[Figure: Template with a table skeleton added.]

You can do the same thing for written reports or paper manuscripts. For example, most of the papers you like may have the classic scientific paper sections: “Introduction”, “Data and Methods”, “Results”, and “Discussion”. And then, you probably typically include a couple of pages at the beginning for the title page and abstract, and then a section at the end with references and figure captions. Again, you could create a file called “article\_template.docx” with section headings for each of the sections and with space for the title page, abstract, and references. Presumably, you are always an author on papers you're writing, so go ahead and add your name, contact information, and affiliation in the right place on the title page (I bet you have to take the time to do that every time you start a paper—and if you're like me, you have to look up the fax number for your building every time you do). You probably need to mention funding sources on the title page for every paper, too. Do you need to look those grant numbers up every time? Nope! Just put all your current ones in the title page of your template, and then you can just delete those that don't apply when you start a new paper.

[Figure: Simple article template]

Again, you can build on this simple template. Look through the “Data and Methods” section of several of your recent papers. Are there certain elements that you commonly report there? For example, is there a mouse model you use in most of your experiments, that you need to describe? Put it in the template. Again, you can always delete or modify this information if it doesn't apply to a specific paper. But for any information that you find yourself copying and pasting from one paper draft to another, add it to your template. It is so much

more delightful to start work on a paper by *deleting* the details that don't apply than by staring down a blank sheet of paper.

[Quote—Taking away everything that isn't the statue.]

"Most docs you work on will have some sort of repeatable process. For example, when I sit down to write a blog post, I go through the same repeatable steps when setting up my file: Title, Subtitle, Focus Keywords, Links to relevant articles / inspiration, Outline of subheads, Intro / hook, etc. ... Even though it is a well-worn process, I can save time by creating a writing template with these sections already pre-set. Not only does this save time, but it also saves mental energy and helps push me into 'Writing' mode instead of 'Set-up' or 'Research' mode."

[mackay2019dry]

This template idea is so basic, and yet far fewer people use it than would seem to make sense. Maybe it's because it does require some forward thinking about the elements of presentations, reports, and papers that are common across your body of work, not just the details that are pertinent to a specific project. It also does require some time investment, but not much more than adding all these element to a single paper or presentation takes. If you can see the appeal of having a template for the communication output that you create from your research, and if you try it an like it, then you are well on your way to having a programmers mindset. The joy of programming is exactly this kind of joy—a little thinking and time at the start and you have these little tools that do some of your work for you over and over again. In fact, a Python programmer has even written a book whose title captures this intrinsic esprit: "[Automating the Boring Stuff?].

But wait. There's more. Do you always do the same calculations or statistical tests with the data you're getting in? Or at least often enough that it would save time to have a template? There is a way to add this into the template that you create for your presentation, report, or paper.

"Your templates are living documents. If you notice that you're making the same change over and over, that means it's time to update the template itself."

[rose2018dont]

Researchers create and use Excel templates for this purpose. The template may have macros embedded in it to make calculations or create basic graphs. However, spreadsheets—whether created from templates or not—share the limitations discussed in an earlier chapter. What's more, they can't easily be worked into a template that creates a final document to communicate results, whether that's a slide presentation or a a written document. Finally, they are in a binary format that can't clearly be tracked with version control like git.

[R Project templates? Can you create them? Clearly something like that is going on when you start a new package...]

#### 4.1.6 Scripts and automated reports as simple pipelines

Scientific workflows or pipelines have become very popular in many biological research areas. These are meant to meet many of the DRY goals—create a recipe that can be repeated at different times and by different research groups, clearly record each step of an analysis, and automate steps or processes that are repeated across different research projects so they can be completed more efficiently.

There are very sophisticated tools now available for creating biological data analysis pipelines and workflows,[leipzig2017review] including tools like Galaxy and Taverna. Simple code scripts and tools that build on them (like makefiles, RMarkdown documents, and Jupyter Notebooks), however, can be thought of as the simpler (and arguably much more customizable) little sibling of these more sophisticated tools.

“Scripts, written in Unix shell or other scripting languages such as Perl, can be seen as the most basic form of pipeline framework.” [leipzig2017review]

“Naive methods such as shell scripts or batch files can be used to describe scientific workflows.” [mishima2011agile]

Flexibility can be incorporated into scripts, and the tools that build directly off them, through including *variables*, which can be set in different configurations each time the script is run [leipzig2017review].

More complex pipeline systems do have some advantages (although generalizable tools that can be applied to scripts are quickly catching up on most of these). For example, many complex data analysis or processing steps may use open-source software that is under continuing development. If the creators of that software modify it between the time that you submit your first version of an article and the time that you need to submit revisions, and you have updated the version of the package on your computer, the code may no longer run the same way. The same thing can happen if someone else tries to run your code—if they are trying to run it with a more recent version of some of the open-source software used in the code, they may run into problems.

This problem of changes in *dependencies* of the code (software programs, packages, or extensions that the code loads as runs as part of its process) is an important challenge to reproducibility in many areas of science. Pipeline software can improve on simpler scripts by helping limit dependency problems [by ...]. However, R extensions are rapidly being developed that also address this issue. For example, the *packrat* package ...., while [packrat update Nichole was talking about].

“Dependencies refer to upstream files (or tasks) that downstream transformation steps require as input. When a dependency is updated, associated downstream files should be updated as well.” [leipzig2017review]

The tools that we’ve discussed for reproducible and automatable report writing—like RMarkdown and Jupyter Notebooks—build off of a tool for coordinating and conducting a process involving multiple scripts and input files, or a

“build tool”. Among computer programmers, perhaps the most popular build tool is called “make”. This tool allows coders to write a “Makefile” that details the order that scripts should be run in a big process, and what other scripts and inputs they require. With these files, you can re-run a whole project, and do it in the right order, and the only steps that will be re-run are those where something will change based on whatever change you just made to the code or input data.

“To avoid errors and inefficiencies from repeating commands manually, we recommend that scientists use a build tool to automate workflows, e.g., specify the ways in which intermediate data files and final results depend on each other, and on the programs that create them, so that a single command will regenerate anything that needs to be regenerated.” [wilson2014best]

For example, say that you have a large project that starts by inputting data, cleans or processes it using a step that takes a lot of time to run, analyzes the simpler processed data, and then creates some plots and tables based on this analysis. With a makefile, if you want to change the color of the labels on a plot, you can change that code and re-run the Makefile, and the computer will re-make the plots, but not re-run the time-intensive early data processing steps. However, if you update the raw data for the project and re-run the Makefile, the computer will (correctly) run everything from the very beginning, since the updated data needs to be reprocessed, all the way through to creating the final plots and tables.

“A file containing commands for an interactive system is often called a script, though there is really no difference between this and a program. When these scripts are repeatedly used in the same way, or in combination, a workflow management tool can be used. The most widely used tool for this task is probably Make, although many alternatives are now available. All of these allow people to express the dependencies between files, i.e., to say that if A or B has changed, then C needs to be updated using a specific set of commands. These tools have been successfully adopted for scientific workflows as well.” [wilson2014best]

“This experience motivated the creation of a way to encapsulate all aspects of our *in silico* analyses in a manner that would facilitate independent replication by another scientist. Computer and computational scientists refer to this goal as ‘reproducible research’, a coinage attributed to the geophysicist Jon Claerbout in 1990, who imposed the standard of makefiles for construction of all the figures and computational results in papers published by the Stanford Exploration Project. Since that time, other approaches have been proposed, including the ability to insert active scripts within a text document and the use of a markup language that can produce all of the text, figures, code, algorithms, and settings used for the computational research. Although these approaches may accomplish the goal, they are not practical for many nonprogramming experimental scientists using other groups’ or commercial software tools today.” [mesirov2010accessible]

“All science campaigns of sufficient complexity consist of numerous interconnected computational tasks. A workflow in this context is the composition of several such computing tasks.” [deelman2018future]

"Scientific applications can be very complex as software artifacts. They may contain a diverse amalgam of legacy codes, compute-intensive parallel codes, data conversion routines, and remote data extraction and preparation. These individual codes are often stitched together using scripted languages that specify the data and software to be executed, and orchestrate the allocation of computing resources and the movement of data across locations. To manage a particular set of codes, a number of interdependent scripts may be used." [gil2008data]

[Disadvantages of more complex pipeline tools over starting from scripts]

"Unlike command line-based pipeline frameworks ... workbenches allow end-users, typically scientists, to design analyses by linking preconfigured modular tools together, typically using a drag-and-drop graphical interface. Because they require exacting specifications of inputs and outputs, workbenches are intrinsically a subset of configuration-based pipelines." [leipzig2017review]

"Magnificent! Wonderful! So, what's the downside? Well, frameworks lock you into a way of thinking. You can look at a website and, with a trained eye, go, 'Oh, that's a Ruby on Rails site.' Frameworks have an obvious influence on the kind of work developers can do. Some people feel that frameworks make things too easy and that they become a crutch. It's pretty easy to code yourself into a hole, to find yourself trying to force the framework to do something it doesn't want to do. Django, for example, isn't the right tool for building a giant chat application, nor would you want to try competing with Google Docs using a Django backend. You pay a price in speed and control for all that convenience. The problem is really in knowing how much speed, control, and convenience you need." [ford2015what]

"Workbenches and class-based frameworks can be considered heavyweight. There are costs in terms of flexibility and ease of development associated with making a pipeline accessible or fast. Integrating new tools into workbenches clearly increases their audience but, ironically, the developers who are most capable of developing plug-ins for workbenches are the least likely to use them." [leipzig2017review]

"Business workflow management systems emerged in the 1990's and are well accepted in the business community. Scientific workflows differ from business workflows in that rather than coordinating activities between individuals and systems, scientific workflows coordinate data processing activities." [vigder2008supporting]

"The concept of workflows has traditionally been used in the areas of process modelling and coordination in industries. Now the concept is being applied to the computational process including the scientific domain." [mishima2011agile]

"Although bioinformatics-specific pipelines such as bcbio-nextgen and Omics Pipe offer high performance automated analysis, they are not frameworks in the sense they are not easily extensible to integrate new user-defined tools." [leipzig2017review]

Writing a script-based pipeline does require that you or someone in your laboratory group develops some expertise in writing code in a "scripting language" like R or Python. However, the barriers to entry for these languages

continues to come down, and with tools that leverage the ideas of templating and literate programming, it is becoming easier and easier for new R or Python users to learn to use them quickly. For example, one of us teaches a three-credit R Programming class, designed for researchers who have never coded. The students in the class are regularly creating code projects by the end of the class that integrate literate programming tools to weave together code and text and saving these documents within code project directories that include raw data, processed data, and scripts with code definitions for commonly used pieces of code (saved as functions). These are all the skills you'd need to craft an R project template for your research group that can serve as a starting point for each future experiment or project.

"Without an easy-to-use graphical editor, developing workflows requires some programming knowledge." [viger2008supporting]

"Scripting languages are programming languages and as a result are inaccessible to any scientists without computing background. Given that a major aspect of scientific research is the assembly of scientific processes, the fact that scientists cannot assemble or modify the applications themselves results in a significant bottleneck." [gil2008data]

"As anyone who's ever shared a networked folder—or organized a physical filing cabinet—knows, without a good shared filing system your office will implode." [ford2015code]

"You can tell how well code is organized from across the room. Or by squinting or zooming out. The shape of code from 20 feet away is incredibly informative. Clean code is idiomatic, as brief as possible, obvious even if it's not heavily documented. Colloquial and friendly." [ford2015code]

"[Wesley Clark] wanted to make the world's first 'personal computer', one that could fit in a single office or laboratory room. No more waiting in line; one scientist would have it all to himself (or, more rarely, herself). Clark wanted specifically to target biologists, since he knew they often needed to crunch data in the middle of an experiment. At that time, if they were using a huge IBM machine, they'd need to stop and wait their turn. If they had a personal computer in their own lab? They could do calculations on the fly, rejiggering their experiment as they went. It would even have its own keyboard and screen, so you could program more quickly: no clumsy punch cards or printouts. It would be a symbiosis of human and machine intelligence. Or, as Wilkes put it, you'd have 'conversational access' to the LINC: You type some code, you see the result quickly. Clark knew he and his team could design the hardware. But he needed Wilkes to help create the computers' operating system that would let the user control the hardware in real time. And it would have to be simple enough that biologists could pick it up with a day or two of training." [Coders, p. 32]

"When they had a rough first prototype [of the LINC] working, Clark tested it on a real-life problem of biological research. He and his colleague Charles Molnar dragged a LINC out to the lab of neurologist Arnold Starr, who had been trying and failing to record the neuroelectric signals cats produce in their brains when they heard a sound. Starr had put an electrode implant into a cat's cortex, but he

couldn't distinguish the precise neuroelectric signal he was looking for. In a few hours, Molnar wrote a program for the LINC that would play a clicking noise out of a speaker, record precisely when the electrode fired, and map on the LINC's screen the average response of the cat to noises. It worked: As data scrolled across the screen, the scientists 'danced a jig right around the equipment'."

[Coders, p. 33]

If you have built a pipeline as an R or Python script, but there is an open source software tool that you need to use that is written in another language, you can write a "wrapper" function that calls that software from within the R or Python process. And chances are good, if that software is a popular tool, that someone else has already written one, so you can just leverage that code or tool. Open-source scripting languages and R and Python "play well with others", and can communicate and run just about anything that you could run at a command line.

"Our approach to dealing with software integration is to wrap applications with Python wrappers." [vigder2008supporting]

The templating process can eventually extend to making small tools as software functions and extensions. For example, if you regularly create a certain type of graph to show your results, you could write a small function in R that encapsulates the common code for creating that. One research group I know of wanted to make sure their figures all had a similar style (font, color for points, etc.), but didn't like the default values, and so wrote a small function that applied their style choices to every plot they made. Once your research group has a collection of these small functions, you can in turn encapsulate them in a R package (which is really just a collection of R functions, plus maybe some data and documentation). This package doesn't have to be shared outside your research group—you can just share it internally, but then everyone can load and use it in their computational work. With the rise of larger datasets in many fields, and the accompanying need to do more and more work on the computer to clean, manage, and analyze the data, more scientists are getting into this mindset that they are not just the "end users" of software tools, but they can dig in and become artisans of small tools themselves, building on the larger structure and heavier lifting made available by the base software package.

"End-user software engineering refers to research dedicated to improving the capability of end-users who need to perform programming or engineering tasks. For many, if not all, of these end-users, the creation and maintenance of software is a secondary activity performed only in service of their real work. This scenario applies to many fields include science. However, there is little research specifically focused on scientists as end-user software engineers." [vigder2008supporting]

John Chambers (one of the creators of R's precursor S, and heavily involved in R development) defines programming as "a language and environment to turn ideas into new tools." [Programming with Data, p. 2]

"Sometimes, it seems that the software we use just sort of sprang into existence, like grass growing on the lawn. But it didn't. It was created by someone who wrote out—in code—a long, painstaking set of instructions telling the computer precisely what to do, step-by-step, to get a job done. There's a sort of priestly class mystery cultivated around the word *algorithm*, but all they consist of are instructions: Do this, then do this, then do this. News Feed [in Facebook] is now an extraordinarily *complicated* algorithm involving some trained machine learning; but it's ultimately still just a list of rules." [Coders, p. 10]

"One of your nonnegotiable rules should be that every person in the lab must keep a clear and detailed laboratory notebook. The business of the lab is results and the communication of those results, and the lab notebook is the all-important documentation of each person's research. There are dozens of reasons to keep a clear and detailed lab notebook and only one—laziness—for not. Whether the work is on an esoteric branch of clam biology or is heading toward a potentially lucrative patent, it makes sense to keep data clear and retrievable for both present and future lab members." (LEIPS, 2010)

"Paper lab notebooks are most commonly seen, valued for their versatility, low expense, and ease of use. The paper notebook type may be determined by the department or institution. Especially at large companies, there may also be a policy that dictates format, daily signatures by supervisors, and lock-up at night. If there is no requirement, everyone in your lab should use the same kind of bound lab notebook, as determined by you. It should have numbered pages, gridlines, and a tough enough binding that it does not fall apart after a few months of rigorous use on the bench." (LEIPS, 2010)

"Electronic lab notebooks (ELNs) may be used to enter, store, and analyze data. Coupled with a sturdy notebook computer, they can be used at the bench for notes and alterations to the protocol. Lab members and collaborators can share data and drawings. Reagents can be organized. Shareware versions are available as well as many stand-alone programs that can be tweaked for your needs by the company. A lab that generates high-throughput, automated, or visual data; collaborates with other labs; or has a high personnel turnover should consider using an ELN (Phillips 2006)." (LEIPS, 2010)

"Laboratory Information Management Systems (LIMSs) are programs that are coupled to a database as well as to lab equipment and facilitate the entry and storage of laboratory data. These systems are expensive and are designed for more large-scale testing and production than for basic research labs. They can be very useful tools: Some can manage protocols, schedule maintenance of lab instruments, receive and process data from multiple instruments, track reagents and samples, print sample labels, do statistical analyses, and be customized to your needs. But although LIMSs can handle a great deal of data analysis, they cannot yet substitute for a lab notebook." (LEIPS, 2010)

"You should demand a level of care with lab notebooks. Everything in it should be understandable not only to the owner, but to you." (LEIPS, 2010)

"Whether you check notebooks, or have a lab member present to you with raw data, or stop at everyone's lab bench a few times a week, you must have a feeling for the quality and results of each person's raw data. It is very easy to make assumptions on the basis of the polished data you see at a research meeting,

but many a lab member has gone astray with over- or misinterpreted data. By keeping an eye on the raw data, you can be ready to comment on the number of repetitions, alternative experiments, or the implications of a minor result.” (LEIPS, 2010)

“In general, data reuse is most possible when: 1) data; 2) metadata (information describing the data); and 3) information about the process of generating those data, such as code, are all provided.” (Goodman et al., 2014)

“So far we have used filenames without ever saying what a legal name is, so it’s time for a couple of rules. First, filenames are limited to 14 characters. Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should avoid characters that might be used with other meanings. … To avoid pitfalls, you would do well to use only letters, numbers, the period and the underscore until you’re familiar with the situation [i.e., characters with pitfalls]. (The period and the underscore are conventionally used to divide filenames into chunks...) Finally, don’t forget that case distinctions matter—junk, Junk, and JUNK are three different names.” (Kernighan and Pike, 1984)

“The [Unix] system distinguishes your file called ‘junk’ from anyone else’s of the same name. The distinction is made by grouping files into *directories*, rather in the way that books are placed on shelves in a library, so files in different directories can have the same name without any conflict. Generally, each user has a personal or *home directory*, sometimes called *login directory*, that contains only the files that belong to him or her. When you log in, you are ‘in’ your home directory. You may change the directory you are working in—often called your working or *current directory*—but your home directory is always the same. Unless you take special action, when you create a new file it is made in your current directory. Since this is initially your home directory, the file is unrelated to a file of the same name that might exist in someone else’s directory. A directory can contain other directories as well as ordinary files … The natural way to picture this organization is as a tree of directories and files. It is possible to move around within this tree, and to find any file in the system by starting at the root of the tree and moving along the proper branches. Conversely, you can start where you are and move toward the root.” (Kernighan and Pike, 1984)

“The name ‘/usr/you/junk’ is called the *pathname* of the file. ‘Pathname’ has an intuitive meaning: it represents the full name of the path from the root through the tree of directories to a particular file. It is a universal rule in the Unix system that wherever you can use an ordinary filename, you can use a pathname.” (Kernighan and Pike, 1984)

“If you work regularly with Mary on information in her directory, you can say ‘I want to work on Mary’s files instead of my own.’ This is done by changing your current directory with the *cd* command… Now when you use a filename (without the ‘/’s) as an argument to *cat* or *pr*, it refers to the file in Mary’s directory. Changing directories doesn’t affect any permissions associated with a file—if you couldn’t access a file from your own directory, changing to another directory won’t alter that fact.” (Kernighan and Pike, 1984)

“It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, if you

want to write a book, you might want to keep all the text in a directory called "book." (Kernighan and Pike, 1984)

"Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it should be divided too, because it is cumbersome to edit large files. Thus you should type the document as a number of files. You might have separate files for each chapter, called 'ch1', 'ch2', etc. ... With a systematic naming convention, you can tell at a glance where a particular file fits into the whole. What if you want to print the whole book? You could say \$ pr ch1.1 ch1.2 ch 1.3 . . . , but you would soon get bored typing filenames and start to make mistakes. This is where filename shorthand comes in. If you say \$ pr ch\* the shell takes the \* to mean 'any string of characters,' so ch\* is a pattern that matches all filenames in the current directory that begin with ch. The shell creates the list, in alphabetical order, and passes the list to pr. The pr command never sees the \*; the pattern match that the shell does in the current directory generates a list of strings that are passed to pr." (Kernighan and Pike, 1984)

"The current directory is an attribute of a process, not a person or a program. ... The notion of a current directory is certainly a notational convenience, because it can save a lot of typing, but its real purpose is organizational. Related files belong together in the same directory. '/usr' is often the top directory of a user file system... '/usr/you' is your login directory, your current directory when you first log in. ... Whenever you embark on a new project, or whenever you have a set of related files ... you could create a new directory with `mkdir` and put the files there." (Kernighan and Pike, 1984)

"Despite their fundamental properties inside the kernel, directories sit in the file system as ordinary files. They can be read as ordinary files. But they can't be created or written as ordinary files—to preserve its sanity and the users' files, the kernel reserves to itself all control over the contents of directories." (Kernighan and Pike, 1984)

"A file has several components: a name, contents, and administrative information such as permissions and modification times. The administrative information is stored in the inode (over the years, the hyphen fell out of 'i-node'), along with essential system data such as how long it is, where on the disc the contents of the file are stored, and so on. ... It is important to understand inodes, not only to appreciate the options on `ls`, but because in a strong sense the inodes *are* the files. All the directory hierarchy does is provide convenient names for files. The system's name for a file is its *i-number*: the number of the inode holding the file's information. ... It is the i-number that is stored in the first two bytes of a directory, before the name. ... The first two bytes in each directory entry are the only connection between the name of a file and its contents. A filename in a directory is therefore called a *link*, because it links a name in the directory hierarchy to the inode, and hence to the data. The same i-number can appear in more than one directory. The `rm` command does not actually remove the inodes; it removes directory entries or links. Only when the last link to a file disappears does the system remove the inode, and hence the file itself. If the i-number in a directory entry is zero, it means that the link has been removed, but not necessarily the contents of the file—there may still be a link somewhere else." (Kernighan and Pike, 1984)

#### 4.1.7 Subsection 2

"The file system is the part of the operating system that makes physical storage media like disks, CDs and DVDs, removable memory devices, and other gadgets look like hierarchies of files and folders. The file system is a great example of the distinction between logical organization and physical implementation; file systems organize and store information on many different kinds of devices, but the operating system presents the same interface for all of them." (Kernighan, 2011)

"A *folder* contains the names of other folders and files; examining a folder will reveal more folders and files. (Unix systems traditionally use the word *directory* instead of *folder*.) The folders provide the organizational structure, while the files hold the actual contents of documents, pictures, music, spreadsheets, web pages, and so on. All the information that your computer holds is stored in the file system and is accessible through it if you poke around. This includes not only your data, but the executable forms of programs (a browser, for example), libraries, device drivers, and the files that make up the operating system itself.

... The file system manages all this information, making it accessible for reading and writing by applications and the rest of the operating system. It coordinates accesses so they are performed efficiently and don't interfere with each other, it keeps track of where data is physically located, and it ensures that the pieces are kept separate so that parts of your email don't mysteriously wind up in your spreadsheets or tax returns." (Kernighan, 2011)

"File system services are available through system calls at the lowest level, usually supplemented by libraries to make common operations easy to program."

(Kernighan, 2011)

"The file system is a wonderful example of how a wide variety of physical systems can be made to present a uniform logical appearance, a hierarchy of folders and files." (Kernighan, 2011)

"A folder is a file that contains information about where folders and files are located. Because information about file contents and organization must be perfectly accurate and consistent, the file system reserves to itself the right to manage and maintain the contents of folders. Users and application programs can only change the folder contents implicitly, by making requests of the file system." (Kernighan, 2011)

"In fact, folders *are* files; there's no difference in how they are stored except that the file system is totally responsible for folder contents, and application programs have no direct way to change them. But otherwise, it's just blocks on the disk, all managed by the same mechanisms." (Kernighan, 2011)

"A folder entry for this [example] file would contain its name, its size of 2,500 bytes, the date and time it was created or changed, and other miscellaneous facts about it (permissions, type, etc., depending on the operating system). All of that information is visible through a program like Explorer or Finder. The folder entry also contains information about where the file is stored on disk—which of the 100 million blocks [on the example computer's hard disk] contain its bytes. There are different ways to manage that location information. The folder entry could contain a list of block numbers; it could refer to a block that itself contains a list of block numbers; or it could contain the number of the first block, which in turn

gives the second block, and so on. ... Blocks need not be physically adjacent on disk, and in fact they typically won't be, at least for large files. A megabyte file will occupy a thousand blocks, and those are likely to be scattered to some degree. The folders and the block lists are themselves stored in blocks..." (Kernighan, 2011)

"When a program wants to access an existing file, the file system has to search for the file starting at the root of the file system hierarchy, looking for each component of the file path name in the corresponding folder. That is, if the file is /Users/bwk/book/book.txt on a Mac, the file system will search the root of the file system for Users, then search within that folder for bwk, then within that folder for book, then within that for book.txt. ... This is a divide-and-conquer strategy, since each component of the path narrows the search to files and folders that lie within that folder; all others are eliminated. Thus multiple files can have the same name for some component; the only requirement is that the full path name be unique. In practice, programs and the operating system keep track of the folder that is currently in use so searches need not start from the root each time, and the system is likely to cache frequently-used folders to speed up operations." (Kernighan, 2011)

"When quitting R, the option is given to save the 'workspace image'. The workspace consists of all values that have been created during a session—all of the data values that have been stored in RAM. The workspace is saved as a file called .Rdata and then R starts up, it checks for such a file in the current working directory and loads it automatically. This provides a simple way of retaining the results of calculations from one R session to the next. However, saving the entire R workspace is not the recommended approach. It is better to save the original data set and R code and re-create results by running the code again." (Murrell, 2009)

"Project directory organization isn't just about being tidy, but is essential to the way by which tasks are automated across large numbers of files" (Buffalo, 2015)

"Naming files and directories on a computer matters more than you may think. In transitioning from a graphical user interface (GUI) based operating system to the Unix command line, many folks bring the bad habit of using spaces in file and directory names. This isn't appropriate in a Unix-based environment, because spaces are used to separate arguments in commands. ... Although Unix doesn't require file extensions, including extensions in file names helps indicate the type of each file. For example, a file named osativa-genes.fasta makes it clear that this is a file of sequences in FASTA format. In contrast, a file named osativa-genes could be a file of gene models, notes on where these *Oryza sativa* genes came from, or sequence data. When in doubt, explicit is always better than implicit when it comes to filenames, documentation, and writing code." (Buffalo, 2015)

"Scripts and analyses often need to refer to other files (such as data) in your project hierarchy. This may require referring to parent directories in your directory's hierarchy ... In these cases, it's important to always use *relative paths* ... rather than *absolute paths* ... As long as your internal project directory structure remains the same, these relative paths will always work. In contrast, absolute paths rely on your particular user account and directory structures details *above* the project directory level (not good). Using absolute paths leaves your work less portable between collaborators and decreases reproducibility." (Buffalo, 2015)

*“Document the origin of all data in your project directory.* You need to keep track of where data was downloaded from, who gave it to you, and any other relevant information. ‘Data’ doesn’t just refer to your project’s experimental data—it’s any data that programs use to create output. This includes files your collaborators send you from their separate analyses, gene annotation tracks, reference genomes, and so on. It’s critical to record this important data about you’re data, or *metadata*. For example, if you downloaded a set of genic regions, record the website’s URL. This seems like an obvious recommendation, but countless times I’ve encountered an analysis step that couldn’t be easily reproduced because someone forgot to record the data’s source.” (Buffalo, 2015)

*“Record data version information.* Many databases have explicit release numbers, version numbers, or names (e.g., TAIR10 version of genome annotation for *Arabidopsis thaliana*, or Wormbase release WS231 for *Caenorhabditis elegans*). It’s important to record all version information in your documentation, including minor version numbers.” (Buffalo, 2015)

*“Describe how you downloaded the data.* For example, did you use MySQL to download a set of genes? Or the UCSC Genome Browser? These details can be useful in tracking down issues like when data is different between collaborators.” (Buffalo, 2015)

“Bioinformatics projects involve many subprojects and subanalyses. For example, the quality of raw experimental data should be assessed and poor quality regions removed before running it through bioinformatics tools like aligners or assemblers. ... Even before you get to actually analyzing the sequences, your project directory can get cluttered with intermediate files. Creating directories to logically separate subprojects (e.g., sequencing data quality improvement, aligning, analyzing alignment results, etc.) can simplify complex projects and help keep files organized. It also helps reduce the risk of accidentally clobbering a file with a buggy script, as subdirectories help isolate mishaps. Breaking a project down into subprojects and keeping these in separate subdirectories also makes documenting your work easier; each README pertains to the directory it resides in. Ultimately, you’ll arrive at your own project organization system that works for you; the take-home point is: leverage directories to help stay organized.” (Buffalo, 2015)

“Because lots of daily bioinformatics work involves file processing, programmatically accessing files makes our job easier and eliminates mistakes from mistyping a filename or forgetting a sample. However, our ability to programmatically access files with wildcards (or other methods in R or Python) is only possible when our filenames are consistent. While wildcards are powerful, they’re useless if files are inconsistently named. ... Unfortunately, inconsistent naming is widespread across biology, and is the source of bioinformaticians everywhere. Collectively, bioinformaticians have probably wasted thousands of hours fighting others’ poor naming schemes of files, genes, and in code.” (Buffalo, 2015)

“Another useful trick is to use leading zeros ... when naming files. This is useful because lexicographically sorting files (as ls does) leads to correct ordering. ... Using leading zeros isn’t just useful when naming filenames; this is also the best way to name genes, transcripts, and so on. Projects like Ensembl use this naming scheme in naming their genes (e.g., ENSG00000164256).” (Buffalo, 2015)

"In order to read or write a file, the first thing we need to be able to do is specify which file we want to work with. Any function that works with a file requires a precise description of the name of the file and the location of the file. A filename is just a character value..., but identifying the location of a file can involve a **path**, which describes a location on a persistent storage medium, such as a hard drive." (Murrell, 2009)

"A regular expression consists of a mixture of **literal** characters, which have their normal meaning, and **metacharacters**, which have a special meaning. The combination describes a **pattern** that can be used to find matches amongst text values." (Murrell, 2009)

"A regular expression may be as simple as a literal word, such as `cat`, but regular expressions can also be quite complex and express sophisticated ideas, such as `[a-z]{3,4}[0-9]{3}`, which describes a pattern consisting of either three or four lowercase letters followed by any three digits." (Murrell, 2009)

"... it's important to mind R's working directory. Scripts should *not* use `setwd()` to set their working directory, as this is not portable to other systems (which won't have the same directory structure). For the same reason, use *relative paths* ... when loading in data, and *not* absolute pathers... Also, it's a good idea to indicate (either in comments or a README file) which directory the user should set as their working directory." (Buffalo, 2015)

**"Centralize the location of the raw data files and automate the derivation of intermediate data.** Store the input data on a centralized file server that is professionally backed up. Mark the files as read-only. Have a clear and linear workflow for computing the derived data (e.g., normalized, summarized, transformed, etc.) from the raw files, and store these in a separate directory. Anticipate that this workflow will need to be run several times, and version it. Use the BiocFileCache package to mirror these files on your personal computer. [footnote: A more basic alternative is the `rsync` utility. A popular solution offered by some organizations is based on ownCloud. Commercial options are Dropbox, Google Drive and the like]." (Holmes and Huber, 2018)

"Using an RCS [revision control system] has changed how I work. ... a day's work is no longer a featureless slog toward the summit, but a sequence of small steps. What one feature could I add? What one problem could I fix? Once a step is made and you are sure your code base is in a safe and clean state, commit a revision, and if your next step turns out disastrously, you can fall back to the revision you just committed instead of starting from the beginning." (Klemens, 2014)

With version control, "Our filesystem now has a time dimension. We can query the RCS's repository of file information to see what a file looked like last week and how it changed from then to now. Even without the other powers, I have found that this alone makes me a more confident writer." (Klemens, 2014)

"The most rudimentary means of revision control is via `diff` and `patch`, which are POSIX-standard and therefore most certainly on your system." (Klemens, 2014)

"Git is a C program like any other, and is based on a small set of objects. The key object is the commit object, which is akin to a unified diff file. Given a previous

commit object and some changes from that baseline, a new commit object encapsulates the information. It gets some support from the *index*, which is a list of the changes registered since the last commit object, the primary use of which will be in generating the next commit object. The commit objects link together to form a tree much like any other tree. Each commit object will have (at least) one parent commit object. Stepping up and down the tree is akin to using *patch* and *patch -R* to step among versions.” (Klemens, 2014)

“Having a backup system organized enough that you can delete code with confidence and recover as needed will already make you a better writer.” (Klemens, 2014)

“GitHub issues are a great way to keep track of bugs, tasks, feature requests, and enhancements. While classical issue trackers are primarily intended to be used as bug trackers, in contrast, GitHub issue trackers follow a different philosophy: each tracker has its own section in every repository and can be used to trace bugs, new ideas, and enhancements by using a powerful tagging system. The main objective of issues in GitHub is promoting collaboration and providing context using cross-references. Raising an issue does not require lengthy forms to be completed. It only requires a title and, preferably, at least a short description. Issues have very clear formatting and provide space for anyone with a GitHub account to provide feedback. … Additional elements of issues are (i) color-coded labels that help to categorize and filter issues, (ii) milestones, and (iii) one assignee responsible for working on the issue.” (Perez-Riverol et al., 2016)

“As another illustration of issues and their generic and wide application, we and others used GitHub issues to discuss and comment on changes in manuscripts and address reviewers’ comments.” (Perez-Riverol et al., 2016)

“A good approach is to store at least three copies in at least two geographically distributed locations (e.g., original location such as a desktop computer, an external hard drive, and one or more remote sites) and to adopt a regular schedule for duplicating the data (i.e., backup).” (Michener, 2015)

“One study surveyed neuroscience researchers at a UK institute.” The backup ‘rule of three’ states that for a file to be sufficiently backed up it should be kept in three separate locations using two different types of media with one offsite backup. A lack of an adequate backup solution could mean permanently lost data, effort and time. In this research, more than 82% of the respondents seemed to be unaware of suitable backup procedures to protect their data. Some respondents kept a single backup of work on external hard disks. Others used the Universities local networked servers as their means of backup.” (AlTarawneh and Thorne, 2017)

“Departmental or institutional servers provide an area to store large files such as graphics files as well as e-mail and documents. Such systems will usually have frequent routine backups of all data, often onto optical disks. They might also encrypt the data, which makes it less able to be hacked. This is the most dependable form of long-term storage.” (LEIPS, 2010)

“It’s very important to keep a project notebook containing detailed information about the chronology of your computational work, steps you’ve taken, information about why you’ve made decisions, and of course all pertinent information to

reproduce your work. Some scientists do this in a handwritten notebook, others in Microsoft Word documents. As with README files, bioinformaticians usually like keeping project notebooks in simple plain-text because these can be read, searched, and edited from the command line and across network connections to servers. Plain text is also a future-proof format: plain-text files written in the 1960s are still readable today, whereas files from word processors only 10 years old can be difficult or impossible to open and edit. Additionally, plain text project notebooks can also be put under version control ... While plain-text is easy to write in your text editor, it can be inconvenient for collaborators unfamiliar with the command line to read. A lightweight markup language called *Markdown* is a plain-text format that is easy to read and painlessly incorporated into typed notes, and can also be rendered to HTML or PDF." (Buffalo, 2015)

"Markdown is just plain-text, which means that it's portable and programs to edit and read it will exist. Anyone who's written notes or papers in old versions of word processors is likely familiar with the hassle of trying to share or update out-of-date proprietary formats. For these reasons, Markdown makes for a simple and elegant notebook format." (Buffalo, 2015)

"Information, whether data or computer code, should be organized in such a way that there is only one copy of each important unit of information." (Murrell, 2009)

"A typical encounter with Bioconductor (Box 1) starts with a specific scientific need, for example, differential analysis of gene expression from an RNA-seq experiment. The user identifies the appropriate documented workflow, and because the workflow contains functioning code, the user runs a simple command to install the required packages and replicate the analysis locally. From there, she proceeds to adapt the workflow to her particular problem. To this end, additional documentation is available in the form of package vignettes and manual pages." (Huber et al., 2015a)

**Case study: high-throughput sequencing data analysis.** Analysis of large-scale RNA or DNA sequencing data often begins with aligning reads to a reference genome, which is followed by interpretation of the alignment patterns. Alignment is handled by a variety of tools, whose output typically is delivered as a BAM file. The Bioconductor packages Rsamtools and GenomicAlignments provide a flexible interface for importing and manipulating the data in a BAM file, for instance for quality assessment, visualization, event detection and summarization. The regions of interest in such analyses are genes, transcripts, enhancers or many other types of sequence intervals that can be identified by their genomic coordinates. Bioconductor supports representation and analysis of genomic intervals with a 'Ranges' infrastructure that encompasses data structures, algorithms and utilities including arithmetic functions, set operations and summarization (Fig. 1). It consists of several packages including IRanges, GenomicRanges, GenomicAlignments, GenomicFeatures, VariantAnnotation and rtracklayer. The packages are frequently updated for functionality, performance and usability. The Ranges infrastructure was designed to provide tools that are convenient for end users analyzing data while retaining flexibility to serve as a foundation for the development of more complex and specialized software. We have formalized the data structures to the point that they enable interoperability, but we have also made them adaptable to specific use cases by allowing additional, less formalized userdefined

data components such as application-defined annotation. Workflows can differ vastly depending on the specific goals of the investigation, but a common pattern is reduction of the data to a defined set of ranges in terms of quantitative and qualitative summaries of the alignments at each of the sites. Examples include detecting coverage peaks or concentrations in chromatin immunoprecipitation–sequencing, counting the number of cDNA fragments that match each transcript or exon (RNA-seq) and calling DNA sequence variants (DNA-seq). Such summaries can be stored in an instance of the class `GenomicRanges`.” (Huber et al., 2015a)

“Visualization is essential to genomic data analysis. We distinguish among three main scenarios, each having different requirements. The first is rapid interactive data exploration in ‘discovery mode.’ The second is the recording, reporting and discussion of initial results among research collaborators, often done via web pages with interlinked plots and tool-tips providing interactive functionality. Scripts are often provided alongside to document what was done. The third is graphics for scientific publications and presentations that show essential messages in intuitive and attractive forms. The R environment offers powerful support for all these flavors of visualization—using either the various R graphics devices or HTML5-based visualization interfaces that offer more interactivity—and Bioconductor fully exploits these facilities. Visualization in practice often requires that users perform computations on the data, for instance, data transformation and filtering, summarization and dimension reduction, or fitting of a statistical model. The needed expressivity is not always easy to achieve in a point-and-click interface but is readily realized in a high-level programming language. Moreover, many visualizations, such as heat maps or principal component analysis plots, are linked to mathematical and statistical models—for which access to a scientific computing library is needed.” (Huber et al., 2015a)

” It can be surprisingly difficult to retrace the computational steps performed in a genomics research project. One of the goals of Bioconductor is to help scientists report their analyses in a way that allows exact recreation by a third party of all computations that transform the input data into the results, including figures, tables and numbers. The project’s contributions comprise an emphasis on literate programming vignettes, the BiocStyle and ReportingTools packages, the assembly of experiment data and annotation packages, and the archiving and availability of all previously released packages. ... Full remote reproducibility remains a challenging problem, in particular for computations that require large computing resources or access data through infrastructure that is potentially transient or has restricted access (e.g., the cloud). Nevertheless, many examples of fully reproducible research reports have been produced with Bioconductor.” (Huber et al., 2015a)

“Using Bioconductor requires a willingness to modify and eventually compose scripts in a high-level computer language, to make informed choices between different algorithms and software packages, and to learn enough R to do the unavoidable data wrangling and troubleshooting. Alternative and complementary tools exist; in particular, users may be ready to trade some loss of flexibility, automation or functionality for simpler interaction with the software, such as by running single-purpose tools or using a point-and-click interface. Workflow and data management systems such as Galaxy and Illumina BaseSpace provide a way to assemble and deploy easy-to-use analysis pipelines from components

from different languages and frameworks. The IPython notebook provides an attractive interactive workbook environment. Although its origins are with the Python programming language, it now supports many languages, including R. In practice, many users will find a combination of platforms most productive for them.” (Huber et al., 2015a)

“A lab manual is perhaps the best way to inform new lab members of the ins and outs of the lab and to keep all members updated on protocols and regulations. What could be included in a lab manual? Anything you do not want to explain over and over, anything that will make the lab more functional and that can make life easier for yourself and lab members.” (LEIPS, 2010)

“Most PIs wish the labs were more organized, but it is not a huge priority, that is, until the first student leaves and no one can find a particular cell line in the freezer boxes. Resolutions are made, the crisis passes, and all goes on as before until the next person leaves. Although it is probably inevitable that there will be some confusion when a long-time lab member moves on, an organized lab will not be as affected as an unorganized one.” (LEIPS, 2010)

“LaTeX gives you output documents that look great and have consistent cross-references and citations. Much of your output document is created automatically and much is done behind the scenes. This gives you extra time to think about the ideas you want to present and how to communicate those ideas in an effective way.” (Van Dongen, 2012)

“LaTeX provides state-of-the-art typesetting” (Van Dongen, 2012)

“Many conferences and publishers accept LaTeX. In addition they provide classes and packages that guarantee documents conforming to the required formatting guidelines.” (Van Dongen, 2012)

“LaTeX automatically numbers your chapters, sections, figures, and so on.” (Van Dongen, 2012)

“LaTeX has excellent bibliography support. It supports consistent citations and an automatically generated bibliography with a consistent look and feel. The style of citations and the organisation of the bibliography is configurable.” (Van Dongen, 2012)

“LaTeX is very stable, free, and available on many platforms.” (Van Dongen, 2012)

“LaTeX was written by Leslie Lamport as an extension of Donald Knuth’s TeX program. It consists of a Turing-complete procedural markup language and a typesetting processor. The combination of the two lets you control both the visual presentation as well as the content of your documents.” (Van Dongen, 2012)

“Roughly speaking LaTeX is built on top of TeX. This adds extra functionality to TeX and makes writing your documents much easier.” (Van Dongen, 2012)

“To create a perfect output file and have consistent cross-references and citations, latex also writes information to and reads information from *auxiliary* files. Auxiliary files contain information about page numbers of chapters, sections, tables, figures, and so on. Some auxiliary files are generated by latex itself (e.g.,

aux files). Others are generated by external programs such as bibtex, which is a program that generates information for the bibliography. When an auxiliary file changes then LaTeX may be out of sync. You should rerun latex when this happens.” (Van Dongen, 2012)

“LaTeX is a markup language and document preparation system. It forces you to focus on the content and *not* on the presentation. In a LaTeX program you write the content of your document, you use commands to provide markup and automate tasks, and you import libraries.” (Van Dongen, 2012)

“The main purpose of [LaTeX] commands is to provide markup. For example, to specify the author of the document you write \author{<author name>}. The real strength of LaTeX is that it also is a Turing-complete programming language, which lets you define your own commands. These commands let you do real programming and give you ultimate control over the content and the final visual presentation. You can reuse your commands by putting them in a library.” (Van Dongen, 2012)

“The paragraph is one of the most important basic building blocks of your document. The paragraph formation rules depend on how latex treats spaces, empty lines, and comments. Roughly, the rules are as follows. In its default model, latex treats a sequence of one or more spaces as a single space. The end of the line is the same as a space. However: An empty line acts as an end-of-paragraph specifier...” (Van Dongen, 2012)

### **Including executable code in other languages.**

In your RMarkdown documents, you include executable code in special sections (“chunks”) that are separated from the regular text using a special combination of characters, as described earlier in this module and in the previous module. By default, in Rmarkdown files the code in these chunks are executed using the R programming language. However, you can also include executable code in a number of other programming languages. For example, you could set some code chunks to run Python, others to run Julia, and still others (e.g., bash) to run a shell script.

This can be very helpful if you have steps in your Python that use code in different languages. For example, there may be a module in Python that works well for an early step in your data preprocessing, and then later steps that are easier with general R functions. This presents no problem in creating an RMarkdown data pre-processing protocol, as you can include different steps using different languages.

The program that is used to run the code in a specific chunk is called the “engine” for that chunk [ref—R Markdown def guide]. You can change the engine by changing the combination of characters you use to demarcate the start of executable code. When you are including a chunk of R code, you mark it off starting with the character combination ` ` {r}. You change this to give the engine you would like to use—for example, you would include a chunk of Python code using ` ` {python} [ref—R Markdown def guide]. When your RMarkdown document is rendered, your computer will use the specified

software to run each code chunk. Of course, to run that piece of code, your computer must have that type of software installed and available. For example, if you include a chunk of code that you'd like to run with a Python engine, you must have Python on your computer.

While you can use many different software programs as the engine for each code chunk, there are a few limitations with some programs. For many open-source software programs, the results from running a chunk of code with that engine will be available for later code chunks that also use that engine to use as an input [ref—R Markdown def guide]. This is not the case, however, for most of the available engines. For example, if you use the SAS software program as the engine for one of your code chunks, the output from running that code will not be available to input to later code in the document.

#### **Caching code results.**

Some code can take a while to run, particularly if it is processing very large datasets. By default, RMarkdown will re-run all code in the document every time you render it. This is usually the best set-up, since it allows you to confirm that the code is all executing as desired each time the code is rendered. However, if you have steps that take a long time, this can make it so the RMarkdown document takes a long time to render each time you render it.

To help with this problem, RMarkdown has a system that allows you to cache results from some or all code chunks in the document. This is a really nice system—it will check the inputs to that part of the code each time the document is run. If those inputs have changed, it will take the time to re-run that piece of code, to use the updated inputs. However, if the inputs have not changed since the last time the document was rendered, then the last results for that chunk of code will be pulled from memory and used, without re-running the code in that chunk. This saves time most of the times that you render the document, while taking the time to re-run the code when necessary, because the inputs have changed and so the outputs may be different.

There are some downsides to caching. For example, caching can increase the storage space it takes to save Rmarkdown work, as intermediate results are saved. However, if some of your code is very time-intensive to run, it may make sense to look into caching options with Rmarkdown. For more on caching with Rmarkdown documents, see this section of the *R Markdown Cookbook* [ref].

#### **Outputting to other formats.**

You can use RMarkdown to create documents other than traditional reports. Scientists might find the outputs of presentations and posters particularly useful.

RMarkdown has allowed a pdf slide output for a long time. This output leverages the “beamer” format from LaTeX. You can create a series of presentation slides in RMarkdown, using Markdown to specify formatting, and then the document will be rendered to pdf slides. These slides can be shown using pdf viewer software, like Adobe Acrobat, set either to full screen or to

the presentation option. More recently, capability has been added to RMarkdown that allows you to create PowerPoint slides. Again, you will start from an RMarkdown document, using Markdown syntax to do things like divide content into separate slides. Regardless of the output format you choose (pdf slides or PowerPoint), the code to generate figures and tables in the presentation can be included directly in the RMarkdown file, so it is re-run with the latest data each time you render the presentation.

It is also possible to use RMarkdown to create scientific posters, although this is a bit less common and there are fewer tutorial resources with instructions on doing this. To find out more about creating scientific posters with Rmarkdown, you can start by looking at the documentation for some R packages that have been created for this process. Two include the `posterdown` package [ref], with documentation available at <https://reposehub.com/python/miscellaneous/brentthorne-posterdown.html>, and the `pagedown` package [ref], with documentation available at <https://github.com/rstudio/pagedown>. There are also some blog posts available where researchers describe how they created a poster with Rmarkdown; one thorough one is “How to make a poster in R” by Wei Yang Tham, available at <https://wytham.rbind.io/post/making-a-poster-in-r/>.

This idea of customizing Rmarkdown documents has evolved in another useful way through the idea of Rmarkdown templates. These are templates that are customized—often very highly customized—while allowing you to write the content using Rmarkdown. One area where these templates can be very useful to scientists is with article templates that are customized for specific scientific journals. A number of scientific journals have created LaTeX templates that can be used when writing drafts to submit to the journal. These templates produce a draft that is nicely formatted, following all the journal’s guidelines for submission, and in some cases formatted as the final article would be for the journal. These templates have existed for a long time, particularly for journals in fields in which LaTeX is commonly used for document formatting, including physics and statistics. However, the templates traditionally required you to use LaTeX, which is a complex markup language with a high threshold for learning to use it.

Now, many of these article templates have been wrapped within an Rmarkdown template, allowing you to leverage them while writing all the content in Rmarkdown syntax, and allowing you to include executable code directly in the draft. An example of the first page of an article created in Rmarkdown using one of these article templates is shown in Figure 3.8.5.

These Rmarkdown templates are typically available through R packages, which you can install on your computer in the same way you would install any R package (i.e., with the `install.packages` function). Many journal article templates are available through the `rticles` package [ref], including the template used to create the manuscript shown in Figure 3.8.5. You can find more information about the `rticles` package on its GitHub page, at

<https://github.com/rstudio/rticles>. There is also a section in the book *R Markdown: A Definitive Guide* [ref] on writing manuscripts for scientific journals using Rmarkdown, available online at <https://bookdown.org/yihui/rmarkdown/rticles-templates.html>.

As a similar idea, you can created parameterized RMarkdown documents. These are a simple way to create a kind of template for reports in your laboratory. You can create these is a similar way to regular RMarkdown documents, but they include an area where you can change some inputs each time you render the document. There is a section on parameterized reports in *R Markdown: A Definitive Guide* [ref].

You can also use RMarkdown to create much larger outputs, compared to simpler reports and protocols. RMarkdown can now be used to create very large and dynamic documents, including online books (which can also be rendered to pdf versions suitable for printing), dashboard-style websites, and blogs. Once members of your research group are familiar with the basics of RMarkdown, you may want to explore using it to create these more complex outputs. The book format divides content into chapters and special sections like appendices and references. It includes a table of contents based on weblinks, so readers can easily navigate the content. It uses a book format as its base that allows readers to do things like change the font size and search the book text for keywords. The book containing these modules is one example of using bookdown. If you would like to explore using bookdown to create online books based on Rmarkdown files, there are a number of resources available. There is an online book available with extensive instructions on using this package, available at <https://bookdown.org/yihui/bookdown/>. There is also a helpful website with more details on this package, <https://bookdown.org/>. The website include a gallery of example books created with bookdown <https://bookdown.org/home/archive/>, which you can use to explore the types of books that can be created.

You can also use Rmarkdown documents to create webpages, with pages included for blogs. This format allows you to create a very attractive website that includes a blog section, where you can write and regularly post new blogs, keeping the site dynamic. It is a nice entry point to developing and maintaining a website for people who are learning to code in R but otherwise haven't done much coding, as you can do all the steps within RStudio. There are templates for these blogs that are appropriate for creating personal or research group websites for academics. These websites can be created to highlight the research and people in your research lab. You can encourage students and postdocs to create personal sites, to raise the profile of their research. In the past, we have even used one as a central, unifying spot for a group study, with students contributing blog posts as their graded assignment (<https://kind-neumann-789611.netlify.app/>). To learn how to create websites with blogs, you can check the book *blogdown: Creating Websites with R Markdown* [ref], which is available both in print and free online at

<https://bookdown.org/yihui/blogdown/>. This process takes a bit of work to initially get the website set up, but then allows for easy and straightforward maintenance.

Finally, a simpler way to make basic web content with RMarkdown is through their *flexdashboard* format. This format creates a smaller website that is focused on sharing data results—you can see a gallery of examples at <https://rmarkdown.rstudio.com/flexdashboard/examples.html>. This format is excellent for creating a webpage that allows users to view complex, and potentially interactive, results from data you've collected. It can be particularly helpful for groups that need to quickly communicate regularly updated data to viewers. During the COVID-19 pandemic, for example, many public health departments maintained dashboard-style websites to share evolving data on COVID-19 in the community. Using RMarkdown in this case has the key advantage of allowing you to easily update the dashboard webpage as you get new or updated data, since it is easy to re-run any data processing, analysis, and visualization code in the document. To learn how to use RMarkdown to create dashboard websites, you can check out RStudio's *flexdashboard* site at <https://rmarkdown.rstudio.com/flexdashboard/index.html>. There is also guidance available in one of the chapters of *R Markdown: The Definitive Guide* [ref]: <https://bookdown.org/yihui/rmarkdown/dashboards.html>.

### **More complex formatting.**

As mentioned earlier, Markdown is a fairly simple markup language. Occasionally, this simplicity means that you might not be able to create fancier formatting that you might desire. There is a method that allows you to work around this constraint in RMarkdown.

In RMarkdown documents, when you need more complex formatting, you can shift into a more complex markup language for part of the document. Markup languages like LaTeX and HTML are much more expressive than Markdown, with many more formatting choices possible. For example, there is functionality within LaTeX and HTML to create much more complex tables than in Markdown. However, there is a downside—when you include formatting specified in these more complex markup languages, you will limit the output formats that you can render the document to. For example, if you include LaTeX formatting within an RMarkdown document, you must output the document to PDF, while if you include HTML, you must output to an HTML file. Conversely, if you stick with the simpler formatting available through the Markdown syntax, you can easily switch the output format for your document among several choices.

The *R Markdown Cookbook* [ref] includes chapters on how to customize RMarkdown output through LaTeX (<https://bookdown.org/yihui/rmarkdown-cookbook/latex-output.html>) and HTML (<https://bookdown.org/yihui/rmarkdown-cookbook/html-output.html>). These customizations can include creating custom formats for the entire document (for example, you can customize the appearance of a whole HTML

document by customizing the CSS style file for the document). They can also include smaller-level customizations, like changing the citation style that is used in conjunction with a BibTeX file by adding to the preamble for LaTeX output.

One area of customization that is particularly useful and simple to implement is with customized tables. The Markdown syntax can create very simple tables, but does not allow the creation of more complex tables. There is an R package called `kableExtra` [ref] that allows you to create very attractive and complex tables in RMarkdown documents.

This package leverages more of the power of underlying markup languages, rather than the simpler Markdown language. If you remember, Markdown is pretty easy to learn because it has a somewhat limited set of special characters and special markings that you can use to specify formatting in your output document. This basic set of functionality is often all you need, but for complex table formatting, you will need more. There is much more available in the deeper markup languages that you can use specifically to render pdf documents (software derived from TeX) and the one that you can use specifically to render HTML (the HTML markup language). As a result, you will need to create RMarkdown files that are customized to a single output format (pdf or HTML) to take advantage of this package.

You can install this package the same as any other R package from CRAN, using `install.packages`. You will need to use then need to use `library("kableExtra")` within your RMarkdown document before you use functions from the package. The `kableExtra` package is extensively documented through two vignettes that come with package, one if the output will be in pdf ([https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome\\_table\\_in\\_pdf.pdf](https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_pdf.pdf)) and one if it will be in HTML ([https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome\\_table\\_in\\_html.html](https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html)). There is also information on using `kableExtra` available through *R Markdown Cookbook* [ref]: <https://bookdown.org/yihui/rmarkdown-cookbook/kableextra.html>.

“WordPerfect was always the best word processor. Because it allowed for insight into its very structure. You could hit a certain key combination and suddenly the screen would split and you’d reveal the codes, the bolds and italics, and so forth, that would define your text when it was printed. It was beloved of legal secretaries and journalists alike. Because when you work with words, at the practical, everyday level, the ability to look under the hood is essential. Words are not simple. And WordPerfect acknowledged that. Microsoft Word did not. Microsoft kept insisting that what you saw on your screen was the way things were, and if your fonts just kept sort of randomly changing, well, you must have wanted it that way. Then along came HTML, and what I remember most was that sense of being back inside the file. Sure, HTML was a typographic nightmare, a bunch of unjustified Times New Roman in 12 pt on screens with chiclet-size pixels, but under the hood you could see all the pieces. Just like WordPerfect. That transparency was a wonderful thing, and it renewed computing for me.”  
(Ford, 2014)

"TeX was created by Donald E. Knuth, a professor at Stanford University who has achieved international renown as a mathematician and computer scientist. Knuth also has an aesthetic sense uncommon in his field, and his work output is truly phenomenal. TeX is a happy byproduct of Knuth's mammoth enterprise, *The Art of Computer Programming*. This series of reference books, designed to cover the whole gamut of programming concepts and techniques, is a *sine qua non* for all computer scientists." (Seroul, 2012)

"Roughly speaking, text processors fall into two categories: (1) WYSIWYG systems: what you see is what you get. You see on the screen at all times what the printed document will look like, and what you type has immediate effect on the appearance of the document. (2) markup systems, where you type your text interspersed with formatting instructions, but don't see their effect right away. You must run a program to examine the resulting image, whether on paper or on the screen. In computer science jargon, markup systems must compile the source file you type. WYSIWYG systems have the obvious advantage of immediate feedback, but they are not very precise: what is acceptable at a resolution of 300 dots per inch, for an ephemeral publication such as a newsletter or flier, is no longer so for a book that will be phototypeset at high resolution. The human eye is extraordinarily sensitive: you can be bothered by the appearance of a text without being able to pinpoint why, just as you can tell when someone plays the wrong note in an orchestra, without being able to identify the CULprit. One quickly learns in typesetting that the beauty, legibility and comfortable reading of a text depend on minute details: each element must be placed exactly right, within thousandths of an inch. For this type of work, the advantage of immediate feedback vanishes: fine details of spacing, alignment, and so on are much too small to be discernible at the screen's relatively low resolution, and even if it such were not the case, it would still be a monumental chore to find the right place for everything by hand. For this reason it is not surprising that in the world of professional typesetting markup systems are preferred. They automate the task of finding the right place for each character with great precision. Naturally, this approach is less attractive for beginners, since one can't see the results as one types, and must develop a feeling for what the system will do. But nowadays, you can have the best of both worlds by using a markup system with a WYSIWYG front end; we'll talk about such front ends for TEX later on. TEX was developed in the late seventies and early eighties, before WYSIWYG systems were widespread. But were it to be redesigned now, it would still be a markup language. To give you an idea of the precision with which TEX operates: the internal unit it uses for its calculations is about a hundred times smaller than the wavelength of visible light! (That's right, a hundred times.) In other words, any round-off error introduced in the calculations is invisible to the naked eye." (Seroul, 2012)

"You should be sure to understand the difference between a text editor and a text processor. A text processor is a text editor together with formatting software that allows you to switch fonts, do double columns, indent, and so on. A text editor puts your text in a file on disk, and displays a portion of it on the screen. It doesn't format your text at all. We insist on the difference because those accustomed to WYSIWYG systems are often not aware of it: they only know text processors. Where can you find a text editor? Just about everywhere. Every text processor includes a text editor which you can use. But if you use your text processor as a text editor, be sure to save your file using a 'save ASCII' or 'save text only' option, so that the text processor's own formatting commands

are stripped off. If you give **TEX** a file created without this precaution, you'll get garbage, because **TEX** cannot digest your text processor's commands." (Seroul, 2012)

"TeX enabled authors to encode their precise intent into their manuscripts: This block of text is a computer program, while this word is a keyword in that program. The language it used, called TeX markup, formalized the slow, error-prone communication that is normally carried out with the printer over repeated galley proofs." (Apte, 2019)

"The idea of writing markup inside text wasn't especially novel; it has been used from 1970's runoff (the UNIX family of printer-preparation utilities) to today's HTML tags. TeX was new in that it captured key concepts necessary for realistic typesetting and formalized them." (Apte, 2019)

"With these higher-level commands, the free TeX engine, and the LaTeX book, the use of TeX exploded. The macro file has since evolved and changed names, but authors still typically run the program called latex or its variants. Hence, most people who write TeX manuscripts know the program as LaTeX and the commands they use as LaTeX commands." (Apte, 2019)

"The effect of LaTeX on scientific and technical publishing has been profound. Precise typesetting is critical, particularly for conveying concepts using chemical and mathematical formulas, algorithms, and similar constructs. The sheer volume of papers, journals, books, and other publications generated in the modern world is far beyond the throughput possible via manual typesetting. And TeX enables automation without losing precision. Thanks to LaTeX, book authors can generate camera-ready copy on their own. Most academic and journal publishers accept article manuscripts written in LaTeX, and there's even an open archive maintained by Cornell University where authors of papers in physics, chemistry, and other disciplines can directly submit their LaTeX manuscripts for open viewing. Over 10,000 manuscripts are submitted to this archive every month from all over the world." (Apte, 2019)

"For many users, a practical difficulty with typesetting using TeX is preparing the manuscripts. When TeX was first developed, technical authors were accustomed to using plain-text editors like WordStar, vi, or Emacs with a computer keyboard. The idea of marking up their text with commands and running the manuscript through a typesetting engine felt natural to them. Today's typesetters, particularly desktop publishers, have a different mental model. They expect to see the output in graphical form and then to visually make edits with a mouse and keyboard, as they would in any WYSIWYG program. They might not be too picky about the quality of the output, but they appreciate design capabilities, such as the ability to flow text around curved outlines. Many print products are now produced with tools like Microsoft Word for this very reason. TeX authors cannot do the same work as easily." (Apte, 2019)

"Poor documentation can lead to irreproducibility and serious errors. There's a vast amount of lurking complexity in bioinformatics work: complex workflows, multiple files, countless program parameters, and different software versions. The best way to prevent this complexity from causing problems is to document everything extensively. Documentation also makes your life easier when you need to go back and rerun an analysis, write detailed methods about your steps for a paper, or find the origin of some data in a directory." (Buffalo, 2015)

"Scatterplots are useful for visualizing treatment-response comparisons ..., associations between variables ..., or paired data (e.g., a disease biomarker in several patients before and after treatment)." (Holmes and Huber, 2018)

"Sometimes we want to show the relationships between more than two variables. Obvious choices for including additional dimensions are plot symbol shapes and colors. ... Another way to show additional dimensions of the data is to show multiple plots that result from repeatedly subsetting (or 'slicing') the data based on one (or more) of the variables, so that we can visualize each part separately. This is called faceting and it enables us to visualize data in up to four or five dimensions. So we can, for instance, investigate whether the observed patterns among the other variables are the same or different across the range of the faceting variable." (Holmes and Huber, 2018)

"You can add an enormous amount of information and expressivity by making your plots interactive. ... The package `ggvis` is an attempt to extend the good features of `ggplot2` into the realm of interactive graphics. In contrast to `ggplot2`, which produces graphics into R's traditional graphics devices (PDF, PNG, etc.), `ggvis` builds upon a JavaScript infrastructure called Vega, and its plots are intended to be viewed in an HTML browser." (Holmes and Huber, 2018)

"Heatmaps are a powerful way of visualizing large, matrix-like datasets and providing a quick overview of the patterns that might be in the data. There are a number of heatmap drawing functions in R; one that is convenient and produces good-looking output is the function `pheatmap` from the eponymous package." (Holmes and Huber, 2018)

"Plots in which most points are huddled in one area, with much of the available spaces sparsely populated, are difficult to read. If the histogram of the marginal distribution of a variable has a sharp peak and then long tails to one or both sides, transforming the data can be helpful. ... The plots in this chapter that involve microarray data use the logarithmic transformation [footnote: 'We used it implicitly, since the data in the `ExpressionSet` object `x` already came log-transformed']—not only in scatterplots... for the x- and y-coordinates but also ... for the color scale that represents the expression fold change. The logarithm transformation is attractive because it has a definite meaning—a move up or down by the same amount on a log-transformed scale corresponds to the same multiplicative change on the original scale:  $\log(ax) = \log a + \log x$ . Sometimes, however, the logarithm is not good enough, for instance when the data include zero or negative values, or when even on the logarithmic scale the data distribution is highly uneven." (Holmes and Huber, 2018)

"To visualize genomic data, in addition to the general principles we have discussed in this chapter, there are some specific considerations. The data are usually associated with genomic coordinates. In fact, genomic coordinates offer a great organizing principle for the integration of genomic data. ... The main challenge of genomic data visualization is the size of the genomes. We need visualization at multiple scales, from whole genome down to the nucleotide level. It should be easy to zoom in and out, and we may need different visualization strategies for the different size scales. It can be convenient to visualize biological molecules (genomes, genes, transcripts, proteins) in a linear manner, although their embedding in the physical world can matter (a great deal)." (Holmes and Huber, 2018)

"Visualizing the data, either 'raw' or along the various steps of processing, summarization, and inference, is one of the most important activities in applied statistics and, indeed, in science. It sometimes gets short shrift in textbooks since there is not much deductive theory. However, there are many good (and bad) practices, and once you pay attention to it, you will quickly see whether a certain graphic is effective in conveying its message, or what choices you could make to create powerful and aesthetically attractive data visualizations." (Holmes and Huber, 2018)

#### 4.1.8 Applied exercise

"Most scholarly works have citations and a bibliography or reference section. ... The purpose of the bibliography is to provide details of the works that are cited in the text. We shall refer to cited works as *references*. ... The bibliography entries are listed as . The of a reference is also used when the work is cited in the text. The lists the relevant information about the work. ... Even within a single work there may be different styles of citations. *Parenthetical citations* are usually formed by putting one or several citation labels inside square brackets or parentheses. However, there are also other forms of citations that are derived from information in the citation label. ... The \bibliographystyle command tells LaTeX which style to use for the bibliography [e.g., labels as numbers, labels as names and years]. The bibliography style called

, is defined in the file

.bst. ... for citations you use the \cite command. The argument of the \cite command is the logical label of the work you cite." (Van Dongen, 2012)

"Since bibliographies are important and since it's easy to get them wrong, some of the work related to the creation of the bibliography has been automated. This is done by BibTeX. The BibTeX tool requires an external human-readable bibliography database. The database may be viewed as a collection of records. The record defines the title of the work, the author(s) of the work, the year of publication, and so on. The record also defines the logical label of the work. This is the label you use when you \cite the work. The advantage of using BibTeX is that you provide the information of the entries in the bibliography and that BibTeX generates the text for the bibliography. This guarantees consistency and ease of mind. Furthermore, the BibTeX database is reusable and you may find BibTeX descriptions of many scholarly works on the web. ... Generating the bibliography with BibTeX is a multi-stage process, which requires an external program called bibtex. The bibtex program is to BibTeX what the latex program is to LaTeX." (Van Dongen, 2012)

"You specify the name of the BibTeX database and the location of the bibliography with the \bibliography command. The argument of the command is the basename of the BibTeX database. So if you use \bibliography{<db>}, then your database is .bib." (Van Dongen, 2012)

"There are several problems with the basic LaTeX citation mechanism. The biblatex package overcomes many of these. It provides a more flexible citation mechanism and lets you configure your own citation style." (Van Dongen, 2012)

"The biblatex package distinguishes between *parenthetical* and *textual* citations." (Van Dongen, 2012)

"This chapter is an introduction to presenting pictures that are stored in external files. Historically, this was an important mechanism for importing pictures. ... The `figure` environment is used to present pictures, diagrams, and graphs. The environment creates a *floating* environment. *Floating* environments don't allow pagebreaks and they may 'float' to a convenient location in the output document. This mechanism gives LaTeX more freedom to choose better page breaks for the remaining text. ... However, it should be noted that you can also force the typesetting of a floating environment at the 'current' position in the output file. ... LaTeX gives some control over the placement of floating figures, of floating tables, and other floats. For figures the placement is controlled with an optional argument of the `figure` environment. The same mechanism is used for the `table` environment... The optional argument, which controls the placement, may contain any combination of the letters `t` [top of the page], `b` [bottom of a page], `p` [separate page with no text], `h` [approximately at the current position], and `H` [at the current position] ... The default value for the optional argument is `tbp`. LaTeX parses the letters in the optional argument from left to right and puts the figure at the position corresponding to the first letter for which it thinks the position is 'reasonable'. Good positions are the top of the page, the bottom of the page, or a page with floats only, because these positions do not disturb the running text too much." (Van Dongen, 2012)

"We have already seen how to present information in `tabular` environments. The `table` environment creates a *floating* table. As with the `figure` environment, this puts the body of the environment in a numbered table, which may be put in a different place in the document than where it's actually defined. The table placement is controlled with an optional argument. This optional argument works as with the optional argument of the `figure` environment." (Van Dongen, 2012)

"LaTeX's basic support for mathematics is limited, which is why the American Mathematical Society (AMS) provides a package called `amsmath`, which redefines some existing commands and environments and provides additional commands and environments for mathematical typesetting." (Van Dongen, 2012)

"LaTeX has three basic modes that determine how it typesets its input. These modes are: **text mode** In this mode the output does not have mathematical content and is typeset as text. ... **ordinary math mode** In this mode the output has mathematical content and is typeset in the running text. Ordinary math mode is more commonly referred to as inline math mode. **display math mode** In this mode the output has mathematical content and is typeset in a display." (Van Dongen, 2012)

"The `$` operator switches from text mode to ordinary math mode and back... The mathematical expressions in the output are typeset in the running text." (Van Dongen, 2012)

"The `equation` environment is for typesetting a *single* numbered displayed equation. It is one of the most commonly used environments for typesetting display math material." (Van Dongen, 2012)

LaTeX math typesetting includes (Van Dongen, 2012):

- subscripts and superscripts

- Greek letters
- text in formulae
- aligned sets of equations
- delimiters (parentheses, brackets)
- fractions
- sums, products, integration, differentiation
- square roots
- relational symbols (less than or equal to, greater than or equal to symbols)
- arrows
- matrices
- accents, hats, dots, overlines

"This chapter introduces the `beamer` class, which is widely used for computer presentations. Some people call such presentations *powerpoint presentations*. The `beamer` class is seamlessly integrated with the `tikz` package and lets you present *incremental presentations*, which are presentations that incrementally add text and graphics to a page of the presentation." (Van Dongen, 2012)

"In May 1977, Donald Knuth of Stanford University started work on the text-processing system that is now known as 'TeX' and METAFONT. ... In the early 1990s, Donald Knuth officially announced that TeX would not undergo any further development in the interest of stability. Perhaps unsurprisingly, the 1990s saw a flowering of experimental projects that extended TeX in various directions; many of these are coming to fruition in the early 21st century..." (Mittelbach et al., 2004)

The development of TeX from its birth as one of Don's 'personal productivity tools' (created simply to ensure the rapid completion and typographic quality of his then-current work on *The Art of Computer Programming*) was largely influenced and nourished by the American Mathematical Society on behalf of UL research mathematicians." (Mittelbach et al., 2004)

"The most obviously important files in any LaTeX-based documentation project are the *input source files*. Typically, there will be a master file that uses other subsidiary files... These files most often have the extension `.tex`... they are commonly known as 'plain text files' since they can be prepared with a basic text editor. Often, external graphical images are included in the typeset document utilizing the `graphics` interface... LaTeX also needs several files containing structure and layout definitions: *class* files with the extension `.cls`; *option* files with the extension `.clo`; *package* files with the extension `.sty`... Many of these are provided by the base system set-up, but others may be supplied by individual users." (Mittelbach et al., 2004)

"One of the ideas behind LaTeX is the separation between layout and structure (as far as possible), which allows the user to concentrate on content rather than having to worry about layout issues." (Mittelbach et al., 2004)

"Commands placed between `\documentclass` and `\begin{document}` are in the so-called *document preamble*. All style parameters must be defined in this preamble, either in package or class files or directly in the document before the `\begin{document}` command, which sets the values for some of the global parameters." (Mittelbach et al., 2004)

"Basic LaTeX offers excellent mathematical typesetting capabilities for straightforward documents. However, when complex displayed equations or more advanced mathematical constructs are heavily used, something more is needed. Although it is possible to define new commands or environments to ease the burden of typing in formulas, this is not the best solution. The American Mathematical Society (AMS) provides a major package, `amsmath`, which makes the preparation of mathematical documents much less time-consuming and more consistent." (Mittelbach et al., 2004)

"Citations are cross-references to bibliographical information outside the current document, such as to publications containing further information on a subject and source information about used quotations. ... There are numerous ways to compile bibliographies and reference lists. They can be prepared manually, if necessary, but usually they are automatically generated from a database containing bibliographical information..." (Mittelbach et al., 2004)

"There are four common methods of referring to sources: the 'short-title', 'author-date', 'author-number', and 'number-only' systems." (Mittelbach et al., 2004)

"The standard LaTeX environment for generating a list of references or a bibliography is called `thebibliography`. In its default implementation it automatically generates an appropriate heading and implements a vertical list structure in which every publication is represented as a separate item." (Mittelbach et al., 2004)

"Inside a document, publications are cited by referring to the `cite-key` argument of the `\bibitem` command. For this purpose LaTeX offers the `\cite` command, which takes such a key as its argument. It can, in fact, take a comma-separated list of such keys and offers an optional argument to specify additional information such as page or chapter numbers." (Mittelbach et al., 2004)

"The BibTeX program gathers all citation keys used in a document, looks them up in a bibliographical database, and generates a complete `thebibliography` environment that can be loaded by LaTeX in a subsequent run." (Mittelbach et al., 2004)

"... the basic structure of a BibTeX entry consists of three parts: 1. A publication *entry type* (e.g., 'book', 'article', 'inproceedings', 'phdthesis'). 2. A *user-chosen keyword* identifying the publication. If you want to reference the entry in your document, then the argument `cite-key` of the `\cite` command should be identical (also in case) to this keyword. 3. A series of *fields* consisting of a field identifier with its data between quotes or curly braces (e.g., 'author', 'journal', and 'title')." (Mittelbach et al., 2004)

"BibTeX entries are read by BibTeX in the bibliography database (the `.bib` file), and the formatting of the entries is controlled by an associated bibliography style (the `.bst` file), which contains a set of instructions written in a stack-based language." (Mittelbach et al., 2004)

"The BibTeX program was designed by Oren Patashnik to provide a flexible solution to the problem of automatically generating bibliography lists conforming to different layout styles. It automatically detects the citation requests in a LaTeX document (by scanning its `.aux` file or files), selects the needed bibliographical

information from one or more specified databases, and formats it according to a specified layout style. Its output is a file containing the bibliography listing as LaTeX code that will be automatically loaded and used by LaTeX on the following run.” (Mittelbach et al., 2004)

“A BibTeX database is a plain text (ASCII) file that contains bibliographical entries internally structured as keyword/value pairs. ... Each entry in a BibTeX database consists of three main parts: a *type* specifier, followed by a *key*, and finally the *data* for the entry itself. The *type* describes the general nature of the entry (e.g., whether it is an article, book, or some other publication). The *key* is used in the interface to LaTeX; it is the string that you have to place in the argument of a \cite command when referencing that particular entry. The *data* part consists of a series of *field entries* depending on the *type*), which can have one of two forms... The comma is the field separator. Spaces surrounding the equals sign or the comma are ignored. Inside the text part of a field (enclosed in a pair of double quotes or a pair of braces) you can have any string of characters, but braces must be matched. The quotes or braces can be omitted for text consisting entirely of numbers (like the year field...). ... BibTeX ignores the case of the letters for the entry type, key, and field names. You must, however, be careful with the key. LaTeX honors the case of the keys specifically as the argument for a \cite command, so the key for a given bibliographic entry must match the one specified in the LaTeX file.” (Mittelbach et al., 2004)

“Various organizations and individuals have developed style files for BibTeX that correspond to the house style of particular journals or editing houses.” (Mittelbach et al., 2004)

“The document format ‘R Markdown’ was first introduced in the knitr package in early 2012. The idea was to embed code chunks (of R or other languages) in Markdown documents. In fact, knitr supported several authoring languages from the beginning in addition to Markdown, including LaTeX, HTML, AsciiDoc, reStructuredText, and Textile. Looking back over the five years, it seems to be fair to say that Markdown has become the most popular document format, which is what we expected. The simplicity of Markdown clearly stands out among these document formats.” (Xie et al., 2018)

“However, the original version of Markdown invented by John Gruber was often found overly simple and not suitable to write highly technical documents. For example, there was no syntax for tables, footnotes, math expressions, or citations. Fortunately, John MacFarlane created a wonderful package named Pandoc to convert Markdown documents (and many other types of documents) to a large variety of output formats. More importantly, the Markdown syntax was significantly enriched. Now we can write more types of elements with Markdown while still enjoying its simplicity.” (Xie et al., 2018)

“In a nutshell, R Markdown stands on the shoulders of knitr and Pandoc. The former executes the computer code embedded in Markdown, and converts R Markdown to Markdown. The latter renders Markdown to the output format you want (such as PDF, HTML, Word, and so on.).” (Xie et al., 2018)

“R Markdown may not be the right format for you if you find these elements not enough for your writing: paragraphs, (section) headers, block quotations, code blocks, (numbered and unnumbered) lists, horizontal rules, tables, inline

formatting (emphasis, strikeout, superscripts, subscripts, verbatim, and small caps text), LaTeX math expressions, equations, links, images, footnotes, citations, theorems, proofs, and examples. We believe this list of elements suffice for most technical and non-technical documents.” (Xie et al., 2018)

“Please do not underestimate the customizability of R Markdown because of the simplicity of its syntax. In particular, Pandoc templates can be surprisingly powerful, as long as you understand the underlying technologies such as LaTeX and CSS, and are willing to invest time in the appearance of your output documents (reports, books, presentations, and / or websites).” (Xie et al., 2018)

“R Markdown documents are often portable in the sense that they can be compiled to multiple types of output formats. Again, this is mainly due to the simplified syntax of the authoring language, Markdown. The simpler the elements in your document are, the more likely that the document can be converted to different formats. Similarly, if you heavily tailor R Markdown to a specific output format (e.g., LaTeX), you are likely to lose the portability, because not all features in one format work in another format.” (Xie et al., 2018)

“Last but not least, your computing results will be more likely to be reproducibly if you use R Markdown (or other knitr-based source documents), compared to the manual cut-and-paste approach. This is because the results are dynamically generated from computer source code. If anything goes wrong or needs to be updated, you can simply fix or update the source code, compile the document again, and the results will automatically be updated. You can enjoy reproducibility and convenience at the same time.” (Xie et al., 2018)

“There is a fundamental assumption underneath R Markdown that users should be aware of: we assume it suffices that only a limited number of features are supported in Markdown. By ‘features’, we mean the types of elements you can create with native Markdown. The limitation is a great feature, not a bug.” (Xie et al., 2018)

“R Markdown provides an authoring framework for data science. You can use a single R Markdown file to both: save and execute code, and; generate high quality reports that can be shared with an audience. R Markdown was designed for easier reproducibility, since both the computer code and narratives are in the same document, and results are automatically generated from the source code.” (Xie et al., 2018)

“There are three basic components of an R Markdown document: the metadata, text, and code. The metadata is written between the pair of three dashes ---. The syntax for the metadata is YAML (YAML Ain’t Markup Language), so sometimes it’s also called the YAML metadata or the YAML frontmatter. Before it bites you hard, we want to warn you in advance that indentation matters in YAML, so do not forget to indent the sub-fields of a top field properly. ... The body of a document follows the metadata. The syntax for text (also known as prose or narratives) is Markdown... There are two types of computer code... a code chunk starts with three backticks ... and ends with three backticks... An inline R code expression starts with `r and ends with a backtick.” (Xie et al., 2018)

“It is fine for humans to err (in computing), as long as the source code is readily available.” (Xie et al., 2018)

“The idea [of R Markdown] should be simple enough: interweave narratives with code in a document, knit the document to dynamically generate results from the code, and you will get a report. This idea was not generated by R Markdown, but came from an early programming paradigm called ‘Literate Programming’ (Knuth, 1984).” (Xie et al., 2018)

“The text in an R Markdown document is written with the Markdown syntax. Precisely speaking, it is Pandoc’s Mardown. There are many flavors of Markdown invented by different people, and Pandoc’s flavor is the most comprehensive one to our knowledge.” (Xie et al., 2018)

Markdown syntax allows: inline formatting (e.g., italics, subscript, superscript), hyperlinks, including images from external files, footnotes, bibliographical referencing, section headers, ... (Xie et al., 2018)

“There are multiple ways to insert citations [in Markdown], and we recommend that you use BibTeX databases, because they work better when the output format is LaTeX / PDF. ... The key idea is that when you have a BibTeX database (a plain-text file with the conventional filename extension .bib) that contains entries like ... you may add a field named bibliography to the YAML metadata, and set its value to the path of the BibTeX file. Then in Markdown, you may use @R-base ... to reference the BibTeX entry. Pandoc will automatically generate a list of references in the end of the document.” (Xie et al., 2018)

“In general, you’d better leave at least one empty line between adjacent but different elements, e.g., a header and a paragraph. This is to avoid ambiguity to the Markdown renderer.” (Xie et al., 2018)

“Inline LaTeX equations can be written in a pair of dollar signs using the LaTeX syntax...” (Xie et al., 2018)

“There are a lot of things you can do in a code chunk: you can produce text output, tables, or graphics. You have fine control over all these output via chunk options, which can be provided inside the curly braces. ... There are a large number of chunk options in knitr document at <https://yihui.name/knitr/options>. We list a subset of them below [eval, echo, results, collapse, warning, message, error, include, cache, fig.width, fig.height, out.width, out.height, fig.align, dev, fig.cap].” (Xie et al., 2018)

“Chunk options in knitr can be surprisingly powerful. For example, you can create animations from a series of plots in a code chunk.” (Xie et al., 2018)

“If caching is enabled the same code chunk will not be evaluated the next time the document is compiled (if the code chunk was not modified), which can save you time. However, I want to honestly remind you of the two hard problems in computer science (via Phil Karlton): naming things, and cache invalidation. Caching can be handy but also tricky sometimes.” (Xie et al., 2018)

“PDF documents are generated through the LaTeX files generated from R Markdown. A highly surprising fact to LaTeX beginners is that figures float by default: even if you generate a plot in a code chunk on the first page, the whole figure environment may float to the next page. This is just how LaTeX works by default. It has a tendency to float figures to the top or bottom of pages. Although it can

be annoying and distracting, we recommend that you refrain from playing the ‘Whac-A-Mole’ game in the beginning of your writing, i.e., desperately trying to position figures ‘correctly’ while they seem to be always dodging you. You may wish to fine-tune the positions once the content is complete using the `fig.pos` chunk option.” (Xie et al., 2018)

“Formatting tables can be a very complicated task, especially when certain cells span more than one column or row. It is even more complicated when you have to consider different output formats. For example, it is difficult to make a complex table work for both PDF and HTML output.” (Xie et al., 2018)

“A less well-known fact about R Markdown is that many other languages are also supported, such as Python, Julia, C++, and SQL. The support comes from the `knitr` package, which has provided a large number of *language engines*. Language engines are essentially functions registered in the object `knitr::knit_engine`. ... To use a different language engine, you can change the language name in the chunk header from `r` to the engine name... For engines that rely on external interpreters such as `python`, `perl`, and `ruby`, the default interpreters are obtained from `Sys.which()`, i.e., using the interpreter found via the environment variable `PATH` of the system. If you want to use an alternative interpreter, you may specify its path in the chunk option `engine.path`. ... Most engines will execute each code chunk in a separate new session (via a `system()` call in R), which means objects created in memory in a previous code chunk will not be directly available to latter code chunks. ... Currently, the only exceptions are `r`, `python`, and `julia`. Only these engines execute code in the same session throughout the document. To clarify, all `r` code chunks are executed in the same R session, all `python` code chunks are executed in the same Python session, and so on, by *the R session and the Python session are independent.*” (Xie et al., 2018)

“You can also write Shell scripts in R Markdown, if your system can run them (the executable `bash` or `sh` should exist). Usually this is not a problem for Linux or macOS users. It is not impossible for Windows users to run Shell scripts, but you will have to install additional software (such as Cygwin or the Linux Subsystem).” (Xie et al., 2018)

“R Markdown documents can also generate interactive content. There are two types of interactive R Markdown documents: you can use the HTML Widgets framework, or the Shiny framework (or both). ... The HTML Widgets framework is implemented in the R package `htmlwidgets`, interfacing JavaScript libraries that create interactive applications, such as interactive graphics and tables. ... The `shiny` package builds interactive web apps powered by R. ... You may use Shiny to run any R code that you like in response to user actions. Since web browsers cannot execute R code, Shiny interactions occur on the server side and rely on a live R session. By comparison, HTML widgets do not require a live R session to support them, because the interactivity comes from the client side (via JavaScript in the web browser).” (Xie et al., 2018)

“By default, MathJax scripts are included in HTML documents for rendering LaTeX and MathML equations.” (Xie et al., 2018)

“Within R Markdown documents that generate PDF output, you can use raw LaTeX, and even define LaTeX macros. See Pandoc’s documentation on the `raw_tex` extension for details.” (Xie et al., 2018)

“By default, citations are processed through pandoc-citeproc, which works for all output formats. For PDF output, sometimes it is better to use LaTeX packages to process citations, such as natbib or biblatex. To use one of these packages, just set the option citation\_package [in the YAML].” (Xie et al., 2018)

“Dashboards are particularly common in business-style reports. They can be used to highlight brief and key summaries of a report. The layout of a dashboard is often grid-based, with components arranged in boxes of various sizes.” (Xie et al., 2018)

“Most R Markdown applications are single documents. That is you, have a single R Markdown source document, and it generates a single output file. However, it is also possible to work with multiple Rmd documents in a project, and organize them in a meaningful way (e.g., pages can reference each other). Currently there are two major ways to build multiple Rmd documents: blogdown for building websites, and bookdown for authoring books.” (Xie et al., 2018)

“With blogdown, you can write a blog post or a general page in an Rmd document, or a plain Markdown document. These source documents will be built into a static webset, which is essentially a folder containing static HTML files and associated assets (such as images and CSS files). You can publish this folder to any web server as a website. Because it is only a single folder, it can be easy to maintain. ... Because the website is generated from R Markdown, the content is more likely to be reproducible, and also easier to maintain (no cut-and-paste results). Using Markdown means your content could be more portable in the sense that you may convert your pages to PDF or other formats in the future, and you are not tied to the default HTML format.” (Xie et al., 2018)

“Academic journals often have strict guidelines on the formatting for submitted articles. As of today, few journals directly support R Markdown submissions, but many support the LaTeX format. While you can convert R Markdown to LaTeX, different journals have different typesetting requirements and LaTeX styles, and it may be slow and frustrating for all authors who want to use R Markdown to figure out the technical details about how to properly convert a paper based on R Markdown to a LaTeX document that meets the journal requirements. The rrticles package is designed to simplify the creation of documents that conform to submission standards. A suite of custom R Markdown templates for popular journals is provided... Understanding of LaTeX is recommended, but not essential, to use this package. R Markdown templates may sometimes inevitably contain LaTeX code, but usually we can use the simpler Markdown and knit syntax to produce elements like figures, tables, and math equations...” (Xie et al., 2018)

The rrticles package includes templates for the Elsevier and Springer family of journals.

“A century ago society had no way to combat TB, save for limiting its spread by sequestering affected individuals in sanatoriums. Back then TB, often called ‘consumption’, was widespread even in places that today have a relatively low incidence of the scourge, such as North America and Europe. Scientists began to gain on the disease in 1921, when a vaccine made by French immunologists Albert Calmette and Camille Guerin, both at the Pasteur Institute in Paris, first entered

into public use. (Initially believed to protect against both adult and childhood forms of the disease, the BCG vaccine, as it is known, was later shown through an extensive series of tests to confer consistent protection against only severe childhood forms.)" (Barry and Cheung, 2009)

"Preventing TB infection in the first place is, of course, better than treating people after they have become sick. To that end, efforts to create a vaccine that confers better protection against the disease than does the BCG vaccine are under way. Some developers are trying to improve the existing vaccine; others are attempting to make entirely new ones. But for the moment, the work is mostly doomed to trial and error because we do not understand why the current vaccine does not work nor how to predict what will work without testing candidates in humans." (Barry and Cheung, 2009)

"In other diseases for which vaccines are available, surviving an initial infection provides immunity to future infection. In TB, however, initial infection does not offer any such protection. A vaccine that is based simply on an attenuated version of TB therefore will not work." (Barry and Cheung, 2009)

"About a third of the global population harbors a latent TB infection until something—such as stress or another illness—reactivates the bugs, leading both the bacteria and the body's own immune response to attack lung tissue, setting transmission to other individuals in motion." (Lehrman, 2013)

"Once inside the body, the TB germ actually does not do very much. It is the body's own attempts to rid itself of the infection that causes the most damage. For example, the white blood cells of the immune system create the cavities in the lungs where TB gets walled off." (Lehrman, 2013)



# 5

## Bibliography

- (2021). RStudio Project Templates. [https://rstudio.github.io/rstudio-extensions/rstudio\\_project\\_templates.html](https://rstudio.github.io/rstudio-extensions/rstudio_project_templates.html). Accessed: 2021-03-03.
- Al-JunDi, A. and SAkkA, S. (2016). Protocol writing in clinical research. *Journal of clinical and diagnostic research: JCDR*, 10(11):ZE10.
- AlTarawneh, G. and Thorne, S. (2017). A pilot study exploring spreadsheet risk in scientific research. *arXiv preprint arXiv:1703.09785*.
- Altman, D. G. and Bland, J. M. (1997). Statistical notes: Units of analysis. *BMJ*, 314:1874.
- Altschul, S., Demchak, B., Durbin, R., Gentleman, R., Krzywinski, M., Li, H., Nekrutenko, A., Robinson, J., Rasband, W., Taylor, J., et al. (2013). The anatomy of successful computational biology software. *Nature biotechnology*, 31(10):894–897.
- Anderson, N. R., Lee, E. S., Brockenbrough, J. S., Minie, M. E., Fuller, S., Brinkley, J., and Tarczy-Hornoch, P. (2007). Issues in biomedical research data management and analysis: needs and barriers. *Journal of the American Medical Informatics Association*, 14(4):478–488.
- Apte, P. (2019). The lingua franca of latex. *Increment*.
- Arnold, T. (2017). A Tidy Data Model for Natural Language Processing using cleanNLP. *The R Journal*, 9(2):248–267.
- Bacher, R., Chu, L.-F., Leng, N., Gasch, A. P., Thomson, J. A., Stewart, R. M., Newton, M., and Kendziora, C. (2017). Scnorm: robust normalization of single-cell rna-seq data. *Nature methods*, 14(6):584–586.
- Baker, M. (2015). Blame it on the antibodies. *Nature*, 521(7552):274.
- Baker, M. (2016). Quality time. *Nature*, 529(7587):456.

- Barga, R., Howe, B., Beck, D., Bowers, S., Dobyns, W., Haynes, W., Higdon, R., Howard, C., Roth, C., Stewart, E., et al. (2011). Bioinformatics and data-intensive scientific discovery in the beginning of the 21st century. *Omics: a journal of integrative biology*, 15(4):199–201.
- Barnett, D., Walker, B., Landay, A., and Denny, T. N. (2008). Cd4 immunophenotyping in hiv infection. *Nature Reviews Microbiology*, 6(11):S7–S15.
- Barry, C. E. and Cheung, M. S. (2009). New tactics against tuberculosis. *Scientific American*, 300(3):62–69.
- Bass, A. J., Robinson, D. G., Lianoglou, S., Nelson, E., Storey, J. D., and with contributions from Laurent Gatto (2020). *biobroom: Turn Bioconductor objects into tidy data frames*. R package version 1.20.0.
- Baumer, B. (2015). A data science course for undergraduates: Thinking with data. *The American Statistician*, 69(4):334–342.
- Baumer, B. S., Kaplan, D. T., and Horton, N. J. (2017). *Modern Data Science with R*. CRC Press, Boca Raton.
- Ben-David, A. and Davidson, C. E. (2014). Estimation method for serial dilution experiments. *Journal of microbiological methods*, 107:214–221.
- Benoist, C. and Hacohen, N. (2011). Flow cytometry, amped up. *Science*, 332(6030):677–678.
- Birch, D., Lyford-Smith, D., and Guo, Y. (2018). The future of spreadsheets in the big data era. *arXiv preprint arXiv:1801.10231*.
- Blischak, J. D., Davenport, E. R., and Wilson, G. (2016). A quick introduction to version control with git and github. *PLoS computational biology*, 12(1).
- Brazma, A., Krestyaninova, M., and Sarkans, U. (2006). Standards for systems biology. *Nature Reviews Genetics*, 7(8):593.
- Brennecke, P., Anders, S., Kim, J. K., Kołodziejczyk, A. A., Zhang, X., Proserpio, V., Baying, B., Benes, V., Teichmann, S. A., Marioni, J. C., et al. (2013). Accounting for technical noise in single-cell rna-seq experiments. *Nature methods*, 10(11):1093–1095.
- Broman, K. W. and Woo, K. H. (2018). Data organization in spreadsheets. *The American Statistician*, 72(1):2–10.
- Brown, Z. (2018). A git origin story. *Linux Journal*.
- Bryan, J. (2018). Excuse me, do you have a moment to talk about version control? *The American Statistician*, 72(1):20–27.
- Bryan, J. and Wickham, H. (2017). Data science: A three ring circus or a big tent? *Journal of Computational and Graphical Statistics*, 26(4):784–785.

- Buffalo, V. (2015). *Bioinformatics data skills: Reproducible and robust research with open source tools.* " O'Reilly Media, Inc.".
- Burns, P. (2011). *The R inferno.* Lulu. com.
- Butler, D. (2005). Electronic notebooks: A new leaf. *Nature*, 436(7047):20–22.
- Callaway, E. (2016). The visualizations transforming biology. *Nature*, 535(7610):187–188.
- Campbell-Kelly, M. (2007). Number crunching without programming: The evolution of spreadsheet usability. *IEEE Annals of the History of Computing*, 29(3):6–19.
- Chambers, J. (2006). How s4 methods work. Technical report, Technical report.
- Chatfield, C. (1995). *Problem solving: a statistician's guide.* CRC Press.
- Chen, Y., McCarthy, D., Robinson, M., and Smyth, G. K. (2014). edger: differential expression analysis of digital gene expression data user's guide. *Bioconductor User's Guide.* Available online: <http://www.bioconductor.org/packages/release/bioc/vignettes/edgeR/inst/doc/edgeRUsersGuide.pdf> (accessed on 15 February 2021).
- Creeth, R. (1985). Microcomputer spreadsheets: their uses and abuses. *Journal of Accountancy (pre-1986)*, 159(000006):90.
- Edwards, P. N., Mayernik, M. S., Batcheller, A. L., Bowker, G. C., and Borgman, C. L. (2011). Science friction: Data, metadata, and collaboration. *Social Studies of Science*, 41(5):667–690.
- Ellis, S. E. and Leek, J. T. (2018). How to share data for collaboration. *The American Statistician*, 72(1):53–57.
- Ford, P. (2014). On file formats, very briefly. *The Manual.*
- Ford, P. (2015). I dreamed of a perfect database. *New Republic.*
- Fox, A., Dutt, T. S., Karger, B., Rojas, M., Obregón-Henao, A., Anderson, G. B., and Henao-Tamayo, M. (2020). Cyto-feature engineering: A pipeline for flow cytometry analysis to uncover immune populations and associations with disease. *Scientific reports*, 10(1):1–12.
- Garraway, L. (2017). Remember why we work on cancer. *Nature*, 543(7647):613–615.
- Gatto, L. (2013). Msnbbase development.
- Gentleman, R. (2008). *R programming for bioinformatics.* CRC Press.

- Ghosh, S., Matsuoka, Y., Asai, Y., Hsin, K.-Y., and Kitano, H. (2011). Software for systems biology: from tools to integrated platforms. *Nature Reviews Genetics*, 12(12):821.
- Gibb, B. C. (2014). Reproducibility. *Nature chemistry*, 6(8):653–654.
- Giles, J. (2012). The digital lab: lab-management software and electronic notebooks are here—and this time, it's more than just talk. *Nature*, 481(7382):430–432.
- Gillespie, C. and Lovelace, R. (2016). *Efficient R programming: a practical guide to smarter programming.* " O'Reilly Media, Inc.".
- Goodman, A., Pepe, A., Blocker, A. W., Borgman, C. L., Cranmer, K., Crosas, M., Di Stefano, R., Gil, Y., Groth, P., Hedstrom, M., et al. (2014). Ten simple rules for the care and feeding of scientific data.
- Hafemeister, C. and Satija, R. (2019). Normalization and variance stabilization of single-cell rna-seq data using regularized negative binomial regression. *Genome biology*, 20(1):1–15.
- Hammer, P., Banck, M. S., Amberg, R., Wang, C., Petznick, G., Luo, S., Khreb-tukova, I., Schroth, G. P., Beyerlein, P., and Beutler, A. S. (2010). mrna-seq with agnostic splice site discovery for nervous system transcriptomics tested in chronic pain. *Genome research*, 20(6):847–860.
- Hamming, R. R. (1997). *The Art of doing science and engineering: Learning to learn.* CRC Press.
- Haque, A., Engel, J., Teichmann, S. A., and Lönnberg, T. (2017). A practical guide to single-cell rna-sequencing for biomedical research and clinical applications. *Genome medicine*, 9(1):1–12.
- Hermans, F., Jansen, B., Roy, S., Aivaloglou, E., Swidan, A., and Hoepelman, D. (2016). Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 56–65. IEEE.
- Hermans, F. and Murphy-Hill, E. (2015). Enron's spreadsheets and related emails: A dataset and analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 7–16. IEEE.
- Hicks, S. C. and Irizarry, R. A. (2017). A guide to teaching data science. *The American Statistician*, In Press.
- Hicks, S. C., Liu, R., Ni, Y., Purdom, E., and Risso, D. (2021). mbkmeans: Fast clustering for single cell data using mini-batch k-means. *PLOS Computational Biology*, 17(1):e1008625.

- Holmes, S. and Huber, W. (2018). *Modern statistics for modern biology*. Cambridge University Press.
- Hsieh, T., Ma, K., and Chao, A. (2016). iNEXT: an R package for rarefaction and extrapolation of species diversity (Hill numbers). *Methods in Ecology and Evolution*, 7(12):1451–1456.
- Huber, W., Carey, V. J., Gentleman, R., Anders, S., Carlson, M., Carvalho, B. S., Bravo, H. C., Davis, S., Gatto, L., Girke, T., et al. (2015a). Orchestrating high-throughput genomic analysis with bioconductor. *Nature methods*, 12(2):115.
- Huber, W., Carey, V. J., Gentleman, R., Anders, S., Carlson, M., Carvalho, B. S., Bravo, H. C., Davis, S., Gatto, L., Girke, T., Gottardo, R., Hahne, F., Hansen, K. D., Irizarry, R. A., Lawrence, M., Love, M. I., MacDonald, J., Obenchain, V., Ole's, A. K., Pag'es, H., Reyes, A., Shannon, P., Smyth, G. K., Tenenbaum, D., Waldron, L., and Morgan, M. (2015b). Orchestrating high-throughput genomic analysis with Bioconductor. *Nature Methods*, 12(2):115–121.
- Hunt, A., Thomas, D., and Cunningham, W. (2000). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional.
- Ilicic, T., Kim, J. K., Kolodziejczyk, A. A., Bagger, F. O., McCarthy, D. J., Marioni, J. C., and Teichmann, S. A. (2016). Classification of low quality cells from single-cell rna-seq data. *Genome biology*, 17(1):1–15.
- Irizarry, R. A. and Love, M. I. (2016). *Data Analysis for the Life Sciences with R*. Chapman and Hall.
- Irving, F. (2011). Astonishments, ten, in the history of version control.
- Janssens, J. (2014). *Data Science at the Command Line: Facing the Future with Time-tested Tools*. " O'Reilly Media, Inc.".
- Johnson, S. (2011). *Where good ideas come from: The natural history of innovation*. Penguin.
- Kaplan, D. (2018). Teaching stats for data science. *The American Statistician*, 72(1):89–96.
- Keller, S., Korkmaz, G., Orr, M., Schroeder, A., and Shipp, S. (2017). The evolution of data quality: Understanding the transdisciplinary origins of data quality concepts and approaches. *Annu. Rev. Stat. Appl.*, 4:85–108.
- Kernighan, B. W. (2011). *D is for Digital: What a well-informed person should know about computers and communications*. CreateSpace Independent Publishing Platform.
- Kernighan, B. W. and Pike, R. (1984). *The UNIX programming environment*, volume 270. Prentice-Hall Englewood Cliffs, NJ.

- Klemens, B. (2014). *21st century C: C tips from the new school.* " O'Reilly Media, Inc.".
- Krishnan, S., Haas, D., Franklin, M. J., and Wu, E. (2016). Towards reliable interactive data cleaning: A user survey and recommendations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 9. ACM.
- Kwok, R. (2018). Lab notebooks go digital. *Nature*, 560(7717):269–270.
- Lakhani, S. R. and Ashworth, A. (2001). Microarray and histopathological analysis of tumours: the future and the past? *Nature Reviews Cancer*, 1(2):151–157.
- Lawrence, M. and Morgan, M. (2014). Scalable genomics with r and bioconductor. *Statistical science: a review journal of the Institute of Mathematical Statistics*, 29(2):214.
- Lehrman, S. (2013). The diabolical genius of an ancient scourge. *Scientific American*, 309(1):80–85.
- LEIPS, J. (2010). At the helm: Leading your laboratory. *Genetics Research*, 92(4):325–326.
- Levy, S. (1984). A spreadsheet way of knowledge. *Harpers*, 269:58–64.
- Lithgow, G. J., Driscoll, M., and Phillips, P. (2017). A long journey to reproducible results. *Nature*, 548(7668):387–388.
- Lowndes, J. S. S., Best, B. D., Scarborough, C., Afflerbach, J. C., Frazier, M. R., O'Hara, C. C., Jiang, N., and Halpern, B. S. (2017). Our path to better science in less time using open data science tools. *Nature Ecology & Evolution*, 1(6):160.
- Lynch, C. (2008). Big data: How do your data grow? *Nature*, 455(7209):28.
- Lytal, N., Ran, D., and An, L. (2020). Normalization methods on single-cell rna-seq data: an empirical survey. *Frontiers in genetics*, 11:41.
- Maecker, H. T., McCoy, J. P., and Nussenblatt, R. (2012). Standardizing immunophenotyping for the human immunology project. *Nature Reviews Immunology*, 12(3):191–200.
- Majumder, E. L.-W., Billings, E. M., Benton, H. P., Martin, R. L., Palermo, A., Guijas, C., Rinschen, M. M., Domingo-Almenara, X., Montenegro-Burke, J. R., Tagtow, B. A., et al. (2021). Cognitive analysis of metabolomics data for systems biology. *Nature Protocols*, 16(3):1376–1418.
- Mak, H. C. (2011). John storey. *Nature Biotechnology*, 29(4):331–333.
- Marwick, B., Boettiger, C., and Mullen, L. (2018). Packaging data analytical work reproducibly using R (and friends). *The American Statistician*, 72(1):80–88.

- Mascarelli, A. (2014). Research tools: Jump off the page. *Nature*, 507(7493):523–525.
- McCarthy, D. J., Campbell, K. R., Lun, A. T., and Wills, Q. F. (2017). Scater: pre-processing, quality control, normalization and visualization of single-cell rna-seq data in r. *Bioinformatics*, 33(8):1179–1186.
- McCullough, B. (2001). Does microsoft fix errors in excel? In *Proceedings of the 2001 joint statistical meetings*.
- McCullough, B. D. (1999). Assessing the reliability of statistical software: Part ii. *The American Statistician*, 53(2):149–159.
- McCullough, B. D. and Heiser, D. A. (2008). On the accuracy of statistical procedures in microsoft excel 2007. *Computational Statistics & Data Analysis*, 52(10):4570–4578.
- McCullough, B. D. and Wilson, B. (1999). On the accuracy of statistical procedures in microsoft excel 97. *Computational Statistics & Data Analysis*, 31(1):27–37.
- McCullough, B. D. and Wilson, B. (2002). On the accuracy of statistical procedures in microsoft excel 2000 and excel xp. *Computational Statistics & Data Analysis*, 40(4):713–721.
- McCullough, B. D. and Wilson, B. (2005). On the accuracy of statistical procedures in microsoft excel 2003. *Computational Statistics & Data Analysis*, 49(4):1244–1252.
- McMurdie, P. J. and Holmes, S. (2013). phyloseq: an R package for reproducible interactive analysis and graphics of microbiome census data. *PLoS one*, 8(4):e61217.
- McNamara, A. (2016). On the state of computing in statistics education: Tools for learning and for doing. *arXiv preprint arXiv:1610.00984*.
- Mélard, G. (2014). On the accuracy of statistical procedures in microsoft excel 2010. *Computational statistics*, 29(5):1095–1128.
- Metz, C. (2015). How github conquered google, microsoft, and everyone else.
- Michener, W. K. (2015). Ten simple rules for creating a good data management plan. *PLoS computational biology*, 11(10):e1004525.
- Mittelbach, F., Goossens, M., Braams, J., Carlisle, D., and Rowley, C. (2004). *The LATEX companion*. Addison-Wesley Professional.
- Murrell, P. (2009). *Introduction to data technologies*. Chapman and Hall/CRC.

- Myneni, S. and Patel, V. L. (2010). Organization of biomedical data for collaborative scientific research: A research information management system. *International journal of information management*, 30(3):256–264.
- Nardi, B. A. and Miller, J. R. (1990). The spreadsheet interface: A basis for end user programming. In *Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*, pages 977–983. North-Holland Publishing Co.
- Nash, J. (2006). Spreadsheets in statistical practice—another look. *The American Statistician*, 60(3):287–289.
- Neff, E. P. (2021). On the past, present, and future of in vivo science. *Lab animal*, 50(10):273–276.
- Nekrutenko, A. and Taylor, J. (2012). Next-generation sequencing data interpretation: enhancing reproducibility and accessibility. *Nature Reviews Genetics*, 13(9):667–672.
- Pawlak, A., van Gelder, C. W., Nenadic, A., Palagi, P. M., Korpelainen, E., Lijnzaad, P., Marek, D., Sansone, S.-A., Hancock, J., and Goble, C. (2017). Developing a strategy for computational lab skills training through Software and Data Carpentry: Experiences from the ELIXIR Pilot action. *F1000Research*, 6:ELIXIR–1040.
- Peng, R. (2018). Teaching r to new users—from tapply to the tidyverse.
- Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., da Veiga Leprevost, F., Fufezan, C., Ternent, T., Eglen, S. J., Katz, D. S., et al. (2016). Ten simple rules for taking advantage of git and github. *PLoS computational biology*, 12(7).
- Perkel, J. (2018a). Git: The reproducibility tool scientists love to hate.
- Perkel, J. M. (2011). Coding your way out of a problem.
- Perkel, J. M. (2017). Single-cell sequencing made simple. *Nature*, 547(7661):125–126.
- Perkel, J. M. (2018b). The future of scientific figures. *Nature*, 554(7690):133–134.
- Powell, K. (2012). A lab app for that. *Nature*, 484(7395):553–555.
- Powell, S. G., Baker, K. R., and Lawson, B. (2009). Errors in operational spreadsheets: A review of the state of the art. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–8. IEEE.
- Quintelier, K., Couckuyt, A., Emmaneel, A., Aerts, J., Saeys, Y., and Van Gassen, S. (2021). Analyzing high-dimensional cytometry data using flowsom. *Nature Protocols*, 16(8):3775–3801.

- Raphael, M. P., Sheehan, P. E., and Vora, G. J. (2020). A controlled trial for reproducibility.
- Raymond, E. (2009). Understanding version-control systems (draft).
- Raymond, E. S. (2003). *The art of Unix programming*. Addison-Wesley Professional.
- Robinson, D. (2014). broom: An r package for converting statistical analysis objects into tidy data frames. *arXiv preprint arXiv:1412.3565*.
- Robinson, D. (2017). Teach the tidyverse to beginners.
- Robinson, M. D., McCarthy, D. J., and Smyth, G. K. (2010). edger: a bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26(1):139–140.
- Ross, Z., Wickham, H., and Robinson, D. (2017). Declutter your R workflow with tidy tools. *PeerJ Preprints*, 5:e3180v1.
- Sansone, S.-A., Rocca-Serra, P., Field, D., Maguire, E., Taylor, C., Hofmann, O., Fang, H., Neumann, S., Tong, W., Amaral-Zettler, L., et al. (2012). Toward interoperable bioscience data. *Nature genetics*, 44(2):121.
- Savage, A. (2020). *Every tool's a hammer: life is what you make it*. Atria Books.
- Schadt, E. E., Linderman, M. D., Sorenson, J., Lee, L., and Nolan, G. P. (2010). Computational solutions to large-scale data management and analysis. *Nature reviews genetics*, 11(9):647.
- Schrode, N., Seah, C., Deans, P. M., Hoffman, G., and Brennand, K. J. (2021). Analysis framework and experimental design for evaluating synergy-driving gene expression. *Nature Protocols*, 16(2):812–840.
- Seder, R. A., Darrah, P. A., and Roederer, M. (2008). T-cell quality in memory and protection: implications for vaccine design. *Nature Reviews Immunology*, 8(4):247–258.
- Sedgwick, P. (2014). Unit of observation and unit of analysis. *BMJ*, 348:g3840.
- Seroul, R. (2012). *A Beginner's Book of TEX*. Springer Science & Business Media.
- Silge, J. and Robinson, D. (2016). tidytext: Text mining and analysis using tidy data principles in R. *The Journal of Open Source Software*, 1(3).
- Silge, J. and Robinson, D. (2017). *Text Mining with R: A Tidy Approach*. O'Reilly Media, Sebastopol.
- Smith, C. A. (2013). Lc/ms preprocessing and analysis with xcms. *Documentation of Bioconductor xcms package*.

- Spraul, V. A. (2012). *Think like a programmer: an introduction to creative problem solving*. no starch press.
- Standar, J. and Dalla Valle, L. (2017). On enthusing students about big data and social media visualization and analysis using R, RStudio, and RMarkdown. *Journal of Statistics Education*, 25(2):60–67.
- Stark, P. B. (2018). Before reproducibility must come preproducibility. *Nature*, 557(7706):613–614.
- Steen, H. and Mann, M. (2004). The abc's (and xyz's) of peptide sequencing. *Nature reviews Molecular cell biology*, 5(9):699–711.
- Target, S. (2018). Version control before git with cvs.
- Teixeira, R. and Amaral, V. (2016). On the emergence of patterns for spreadsheets data arrangements. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 333–345. Springer.
- Topaloglou, T., Davidson, S. B., Jagadish, H., Markowitz, V. M., Steeg, E. W., and Tyers, M. (2004). Biological data management: Research, practice and opportunities. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1233–1236. VLDB Endowment.
- Tyner, S., Briatte, F., and Hofmann, H. (2017). Network Visualization with ggplot2. *The R Journal*, 9(1):27–59.
- U.S. Department of Health and Human Services, National Institutes of Health (2016). NIH-Wide Strategic Plan, Fiscal Years 2016-2020: Turning Discovery Into Health. Accessed: 2018-06-24.
- U.S. Department of Health and Human Services, National Institutes of Health (2018). NIH Strategic Plan for Data Science. Accessed: 2018-06-24.
- Van Dongen, M. R. (2012). *LATEX and Friends*. Springer Science & Business Media.
- Welsh, E. A., Stewart, P. A., Kuenzi, B. M., and Eschrich, J. A. (2017). Escape excel: A tool for preventing gene symbol and accession conversion errors. *PloS one*, 12(9):e0185207.
- Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.
- Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer, New York, 2nd edition.
- Wickham, H. (2017). The tidy tools manifesto.
- Wickham, H. (2019). *Advanced R, Second Edition*. CRC press.
- Wickham, H. and Grolemund, G. (2016). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media.

- Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., et al. (2016). The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3.
- Willekens, F. (2013). Chronological objects in demographic research. *Demographic research*, 28:649–680.
- Wilson, G. (2014). Software Carpentry: lessons learned. *F1000Research*, 3:62–62.
- Winchester, C. (2018). Give every paper a read for reproducibility. *Nature*, 557(7706):281–282.
- Xie, Y., Allaire, J. J., and Grolemund, G. (2018). *R Markdown: The definitive guide*. CRC Press.
- Xie, Y., Dervieux, C., and Riederer, E. (2020). *R Markdown Cookbook*. Chapman and Hall/CRC.
- Yin, T., Cook, D., and Lawrence, M. (2012). ggbio: an R package for extending the grammar of graphics for genomic data. *Genome Biology*, 13(8):R77.
- Zeeberg, B. R., Riss, J., Kane, D. W., Bussey, K. J., Uchio, E., Linehan, W. M., Barrett, J. C., and Weinstein, J. N. (2004). Mistaken identifiers: gene name errors can be introduced inadvertently when using Excel in bioinformatics. *BMC Bioinformatics*, 5(1):80.
- Ziemann, M., Eren, Y., and El-Osta, A. (2016). Gene name errors are widespread in the scientific literature. *Genome biology*, 17(1):177.