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

IMPROVING THE REPRODUCIBILITY OF EXPERIMENTAL DATA RECORDING AND PRE-PROCESSIN

Contents

1
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 1.1), 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

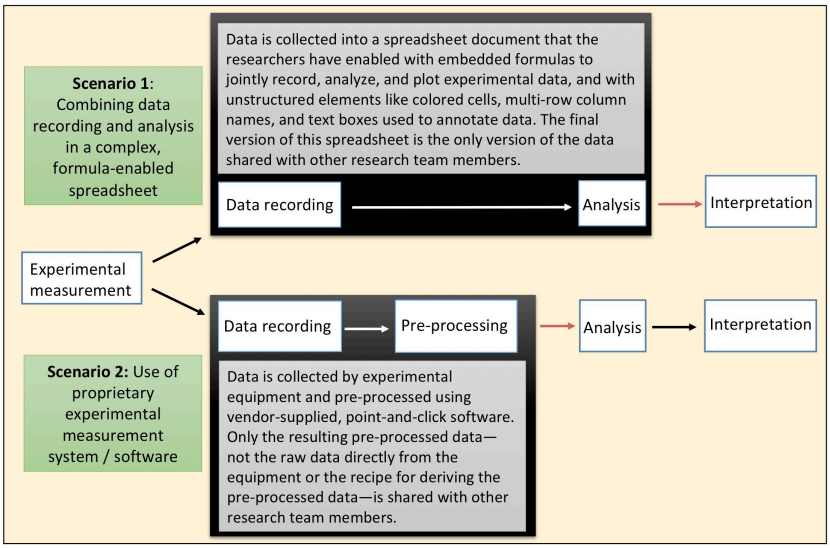


Figure 1.1: Two scenarios where 'black boxes' of non-transparent, non-reproducible data handling exist in research data workflows at the stages of data recording and pre-processing. These create potential points of failure for reproducible research. Red arrows indicate where data is passed to other research 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 choosen 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

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 metaphore 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 scritped programming

lanugages can be perceived as requiring a cognitive load and lengthly 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 lost 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 requires 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 experiemental 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).

"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]

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 defininining 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.

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]

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

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 respository, 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 onotologies 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).

"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]

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, like TextEdit 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

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 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, thing 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.

"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]

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 species 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

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 insitutions. 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.

"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]

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).

"Tabular plain-text data formats are used extensively in computing. The basic format is incrediably 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]

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

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.

"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]

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 taabs 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 ANSCII or Unicode. To extract the data

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 that 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 outputtin 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 outweight 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 Occassions 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 managment 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.

"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, aa tiny, pocket-sized database, built in."
[@ford2015i]

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,

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).

"Without community-level harmonization and interoperability, many community projects risk becoming data silos." [@sansone2012toward]

"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]

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

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

2.2.7 Applied exercise

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; **?**; 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; **?**; 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 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]

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 observation, 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.

"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]

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.

"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]

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 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 measurment (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).

"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]

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 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.

"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]

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" 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.

"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."
[@wickhem2017tidy]

"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]

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 `tidyr` 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 "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`), phytogenetic 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.

2.3.4 Practice quiz

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:

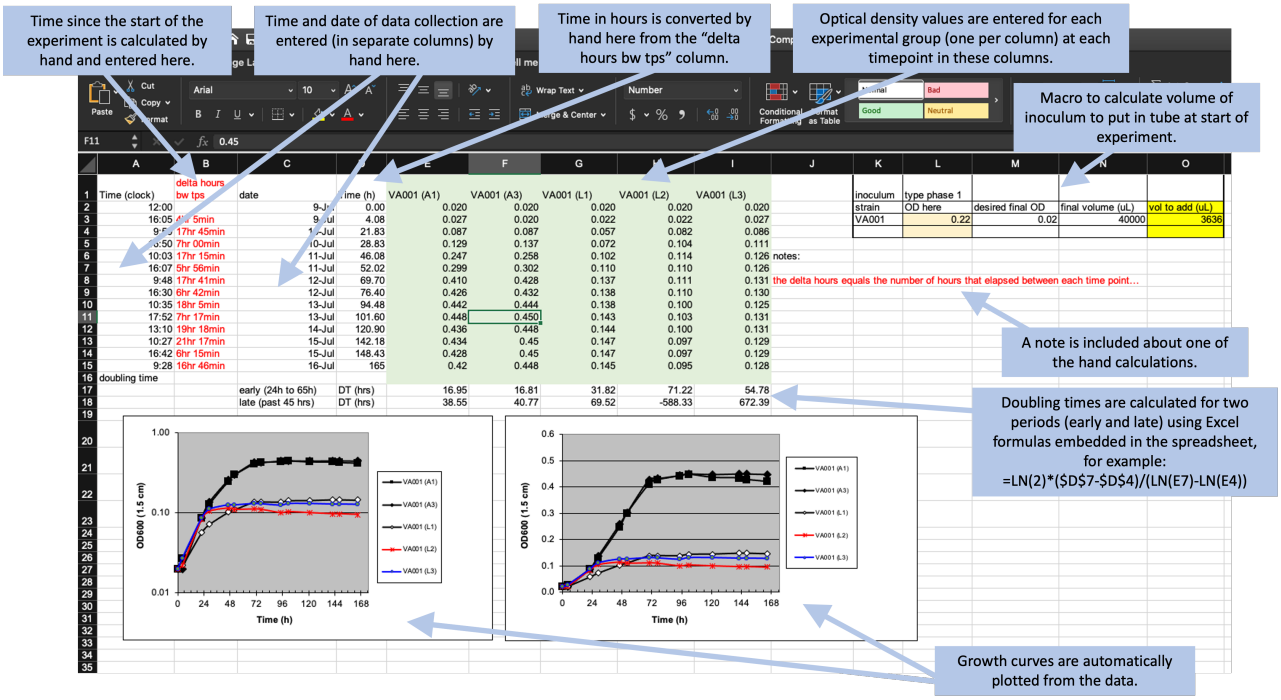


Figure 2.1: Example of an Excel spreadsheet used to record and analyze data for a laboratory experiment. Annotations highlight where data is entered by hand, where calculations are done by hand, and where embedded Excel formulas are used. The figures are created automatically using values in a specified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

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 and 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 spectrophotometer 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,

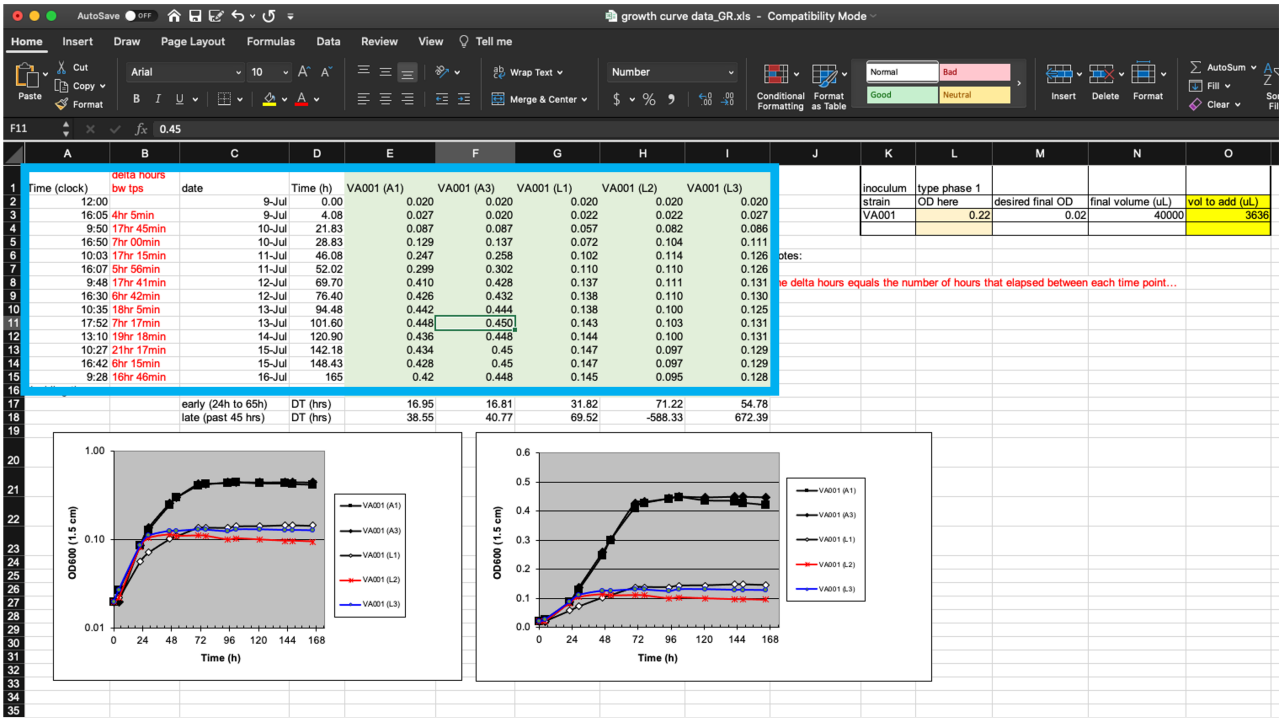
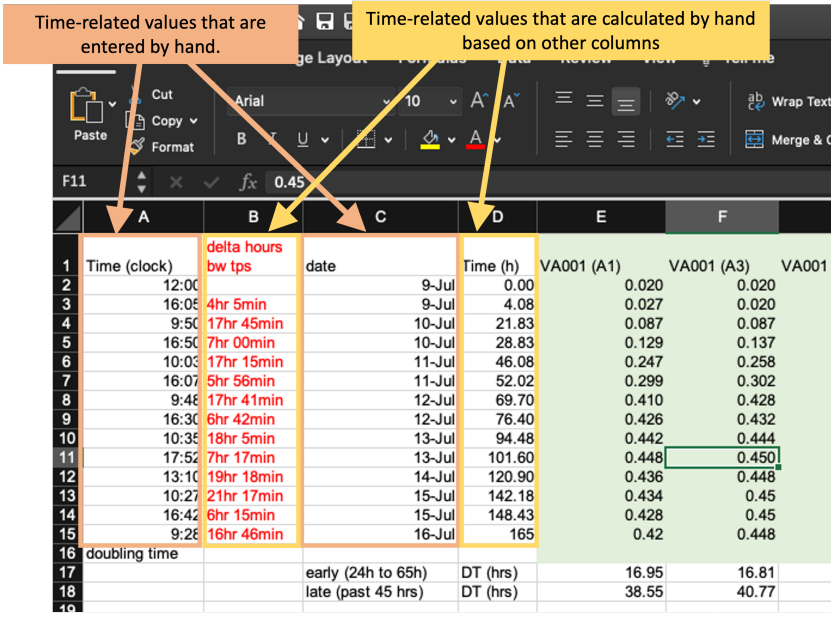


Figure 2.2: Isolating raw data collected in a template from extra elements. The box in this figure highlights the area of the spreadsheet where data are collected. All other elements of the spreadsheet focus on other aims (e.g., summarizing these data, adding notes, macros for experimental design). Those 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.

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 experiment ("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).



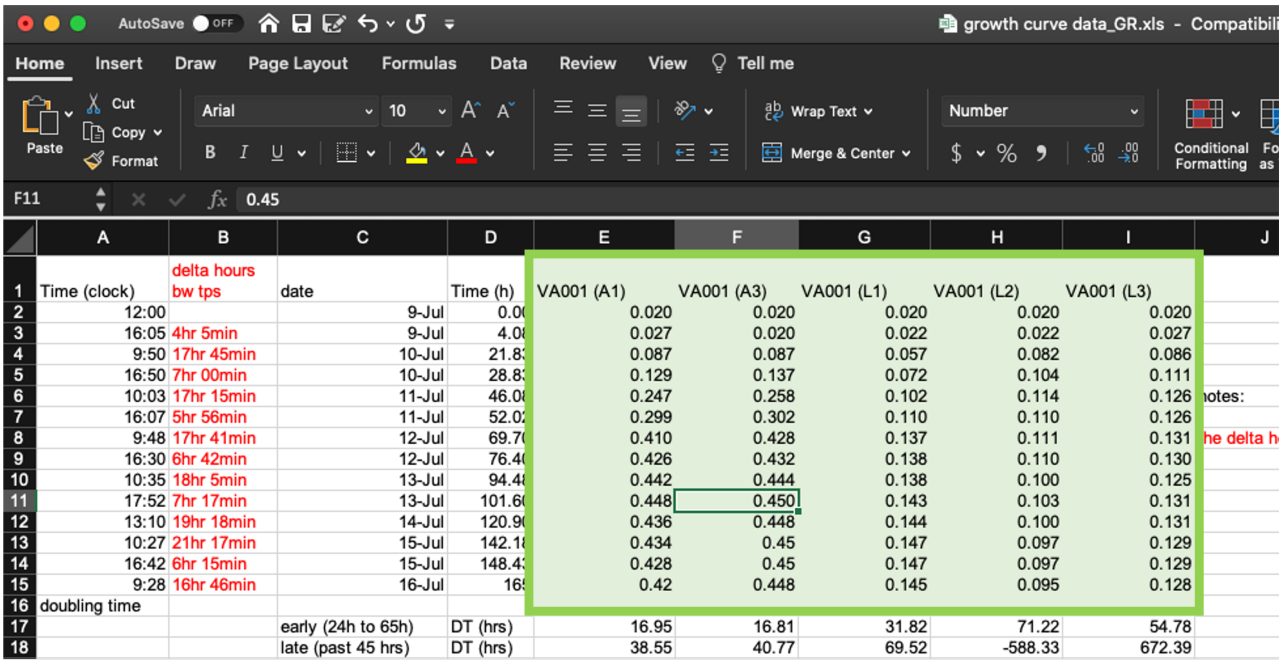
	A	B	C	D	E	F
1	Time (clock)	delta hours bw tps	date	Time (h)	VA001 (A1)	VA001 (A3)
2	12:00		9-Jul	0.00	0.020	0.020
3	16:05	4hr 5min	9-Jul	4.08	0.027	0.020
4	9:50	17hr 45min	10-Jul	21.83	0.087	0.087
5	16:50	7hr 00min	10-Jul	28.83	0.129	0.137
6	10:03	17hr 15min	11-Jul	46.08	0.247	0.258
7	16:07	5hr 56min	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

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 recording 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	0.0	0.020	0.020	0.020	0.020	0.020	
3	16:05	4hr 5min	9-Jul	4.0	0.027	0.020	0.022	0.022	0.027	
4	9:50	17hr 45min	10-Jul	21.8	0.087	0.087	0.057	0.082	0.086	
5	16:50	7hr 00min	10-Jul	28.8	0.129	0.137	0.072	0.104	0.111	
6	10:03	17hr 15min	11-Jul	46.0	0.247	0.258	0.102	0.114	0.126	otes:
7	16:07	5hr 56min	11-Jul	52.0	0.299	0.302	0.110	0.110	0.126	
8	9:48	17hr 41min	12-Jul	69.7	0.410	0.428	0.137	0.111	0.131	he delta h
9	16:30	6hr 42min	12-Jul	76.4	0.426	0.432	0.138	0.110	0.130	
10	10:35	18hr 5min	13-Jul	94.4	0.442	0.444	0.138	0.100	0.125	
11	17:52	7hr 17min	13-Jul	101.6	0.448	0.450	0.143	0.103	0.131	
12	13:10	19hr 18min	14-Jul	120.9	0.436	0.448	0.144	0.100	0.131	
13	10:27	21hr 17min	15-Jul	142.1	0.434	0.45	0.147	0.097	0.129	
14	16:42	6hr 15min	15-Jul	148.4	0.428	0.45	0.147	0.097	0.129	
15	9:28	16hr 46min	16-Jul	16	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	

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.

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 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.

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

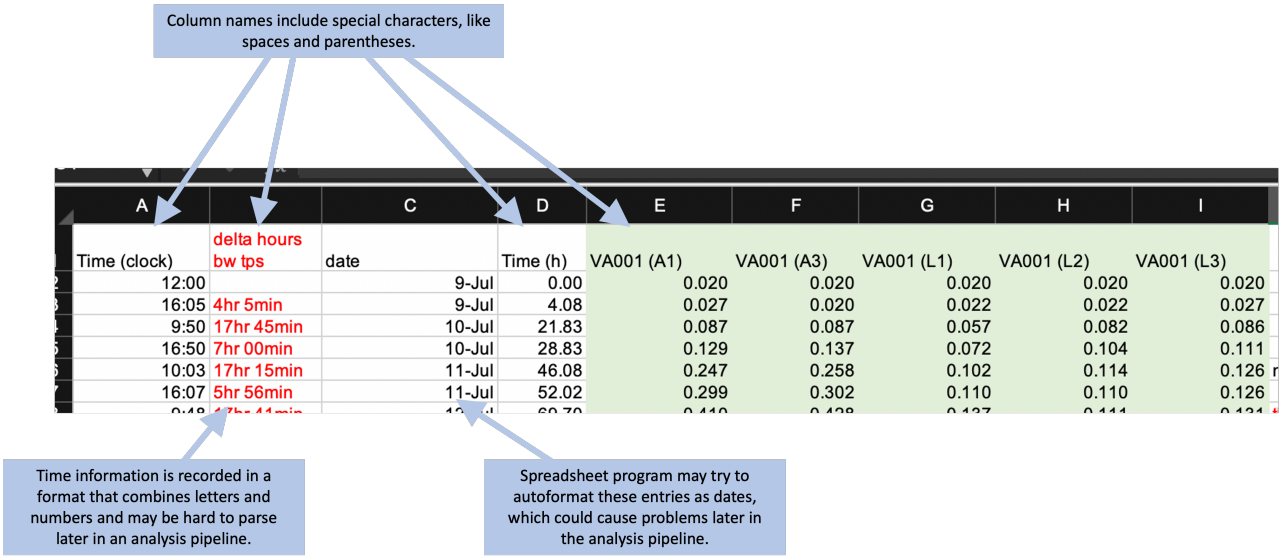


Figure 2.5: Examples of special characters and formatting in the example template that could cause problems later in a data analysis pipeline.

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,

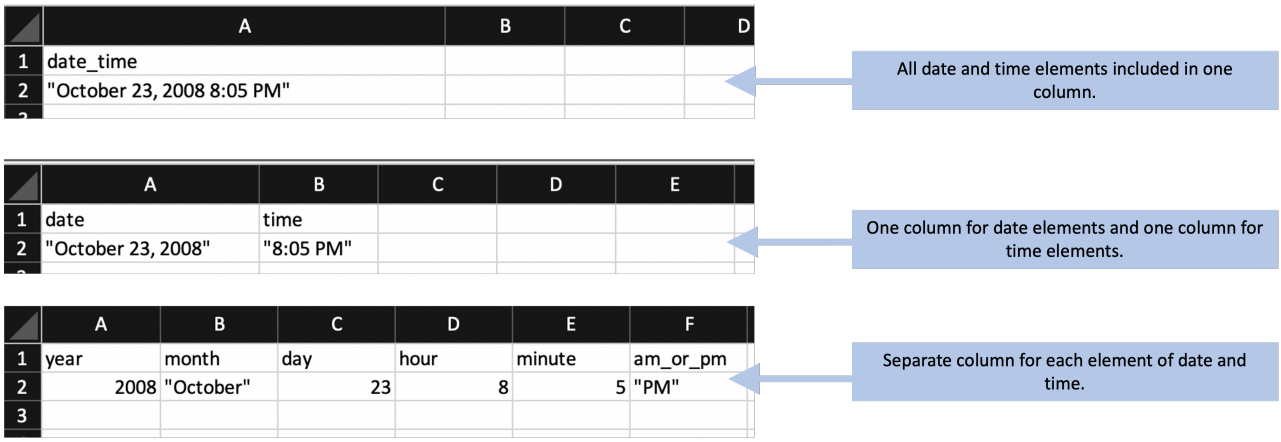


Figure 2.6: Examples of special characters and formatting in the example template that could cause problems later in a data analysis pipeline.

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 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 easy 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).

[Long versus wide formats.]

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 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

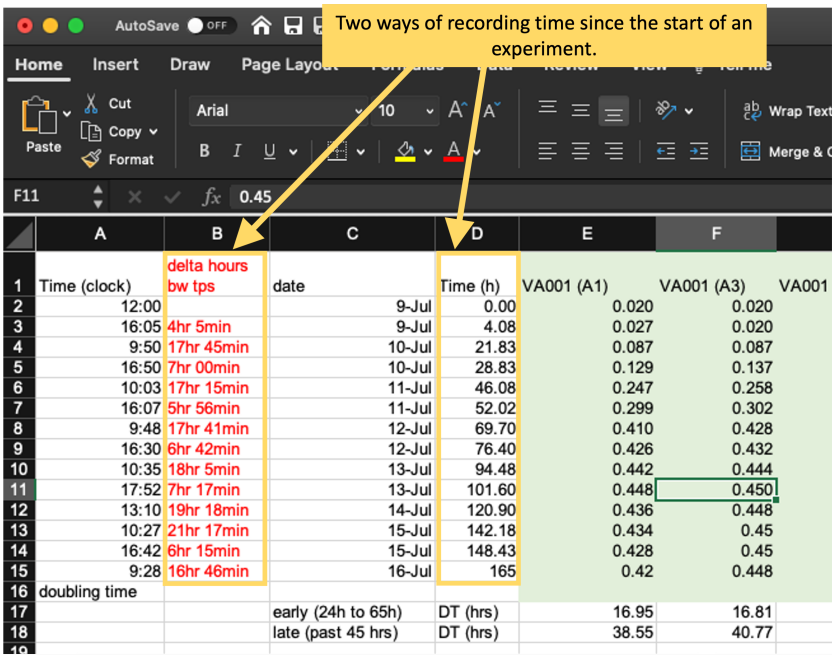
can change it to "Optical_density" without making it much more difficult for a person to understand. As with other choices in designing a data collection template, these choices about column names can be a balance between making the template easy for researchers to use in the laboratory and easy for the statistical program to parse later in the pipeline. For example, statistical programs 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 2.7 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



	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	

Figure 2.7: 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.

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 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 stings, 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.

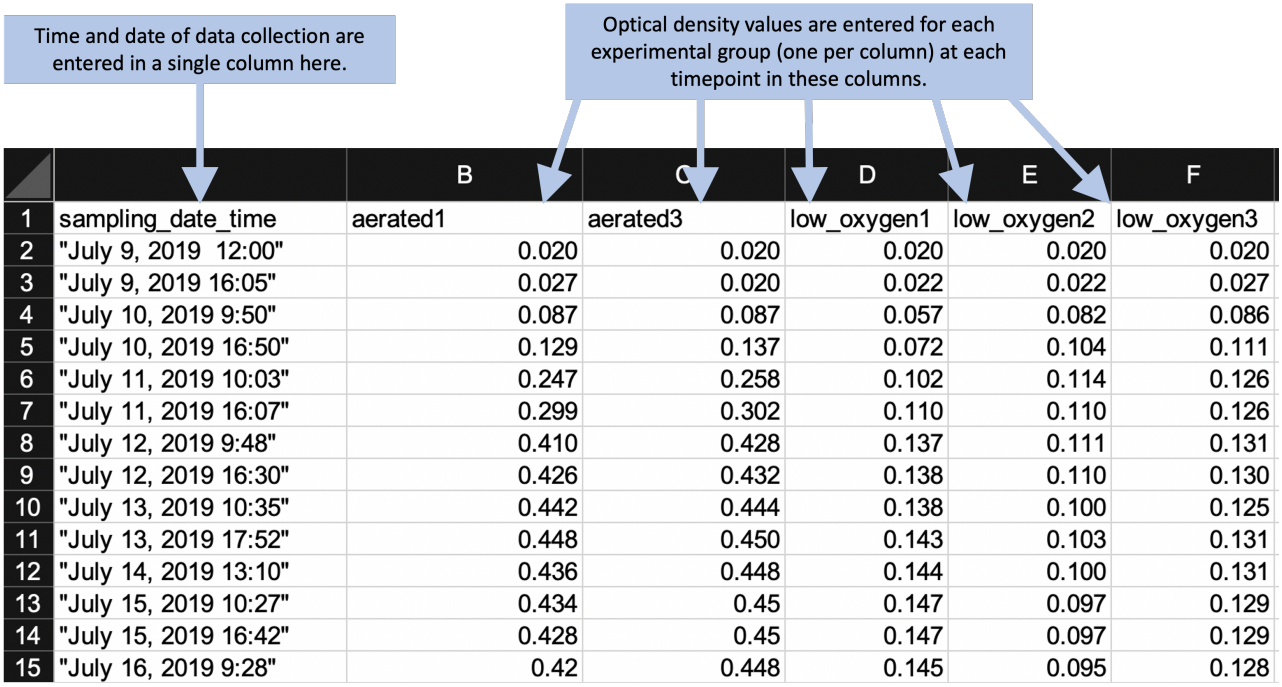
For example, you could design column names and column entries so that you will be able to take advantage of statistical programming tools based on something called regular expressions. "Regular expressions" refers to patterns in character strings (which are just strings of one or more characters, like

"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 current 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 2.9 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 2.8. 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



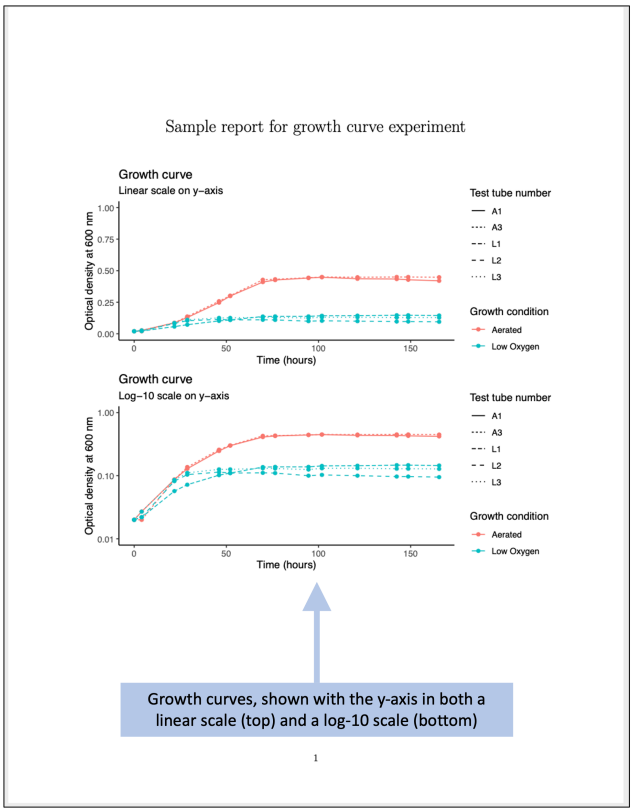
		B	C	D	E	F
1	sampling_date_time	aerated1	aerated3	low_oxygen1	low_oxygen2	low_oxygen3
2	"July 9, 2019 12:00"	0.020	0.020	0.020	0.020	0.020
3	"July 9, 2019 16:05"	0.027	0.020	0.022	0.022	0.027
4	"July 10, 2019 9:50"	0.087	0.087	0.057	0.082	0.086
5	"July 10, 2019 16:50"	0.129	0.137	0.072	0.104	0.111
6	"July 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 2.8: Example of an 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.

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.

The report shown in Figure 2.9 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 2.9. 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



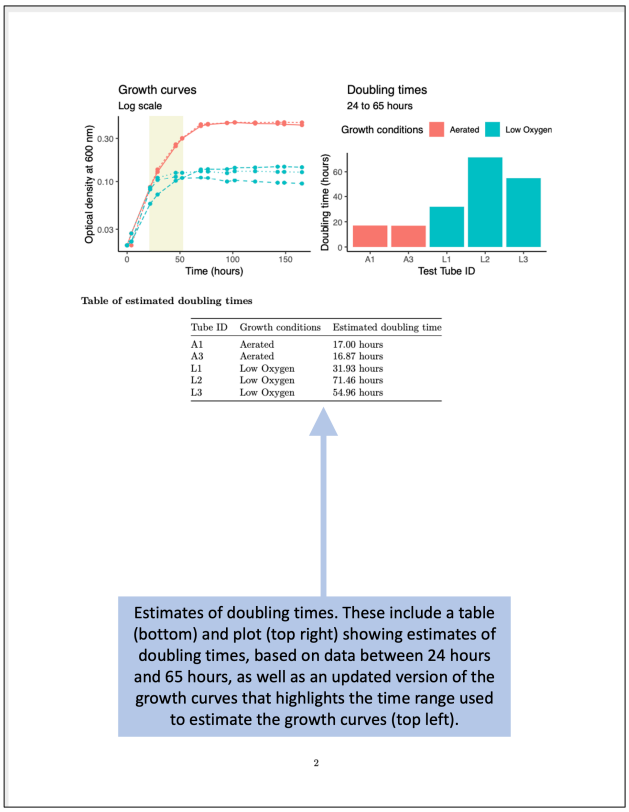


Figure 2.9: Examples of an automated report that can be created to quickly generate summaries and estimates of the data collected in the simplified data collection template for the example experiment.

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 2.9, 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 use to calculate the doubling time for this experiment.

The report shown in Figure 2.9 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 2.10 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 2.10) 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

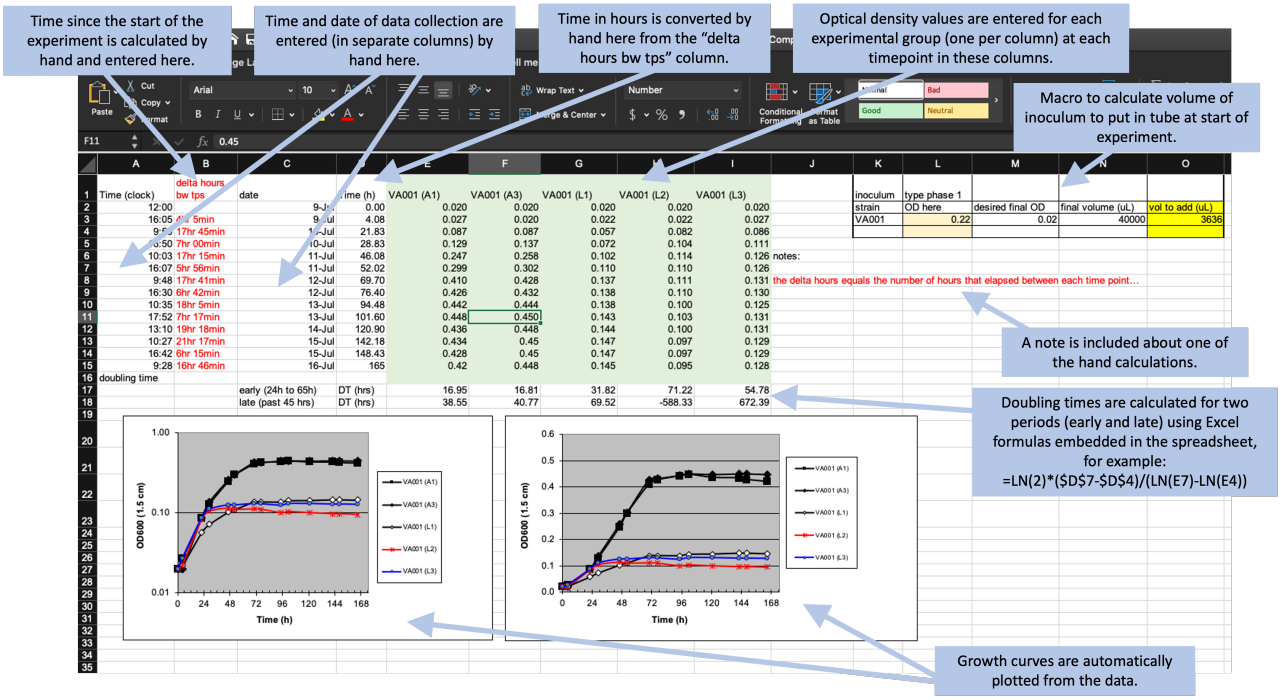


Figure 2.10: Example of an Excel spreadsheet used to record and analyze data for a laboratory experiment. Annotations highlight where data is entered by hand, where calculations are done by hand, and where embedded Excel formulas are used. The figures are created automatically using values in a specified column.

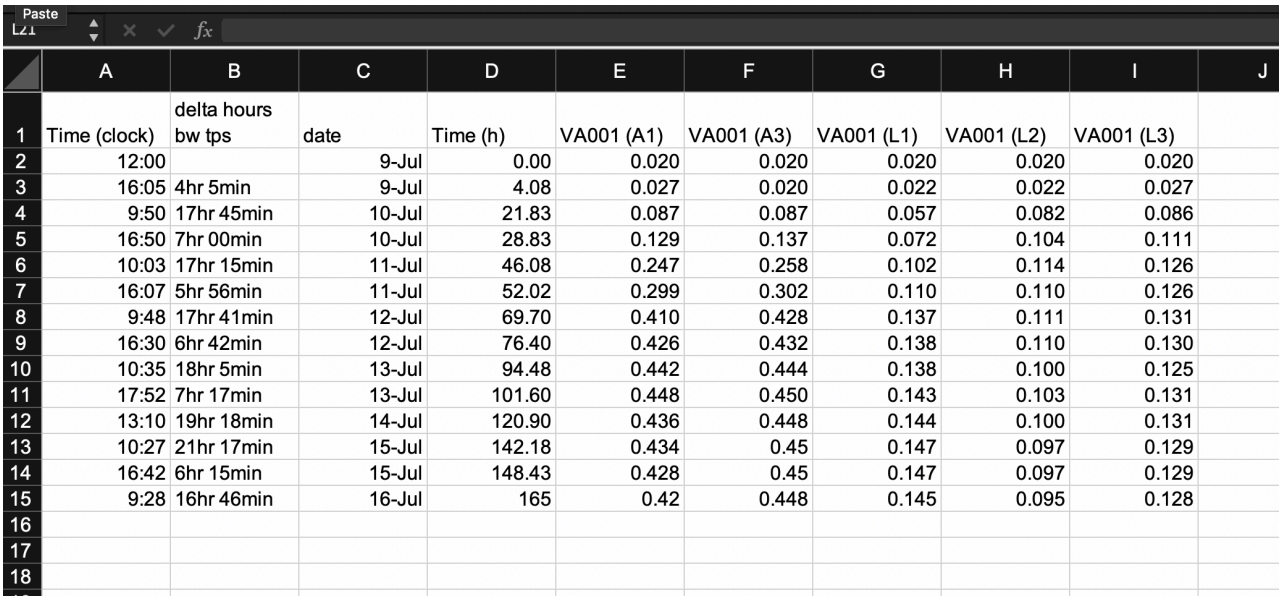
R or Python—at best, they will be ignored by the program. They can even complicate reading in the cells with measurements (rows 1–15 of columns A–I), as most statistical programs will try to read in all the non-empty cells of a spreadsheet unless directed otherwise.

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 2.11.

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 2.11 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.

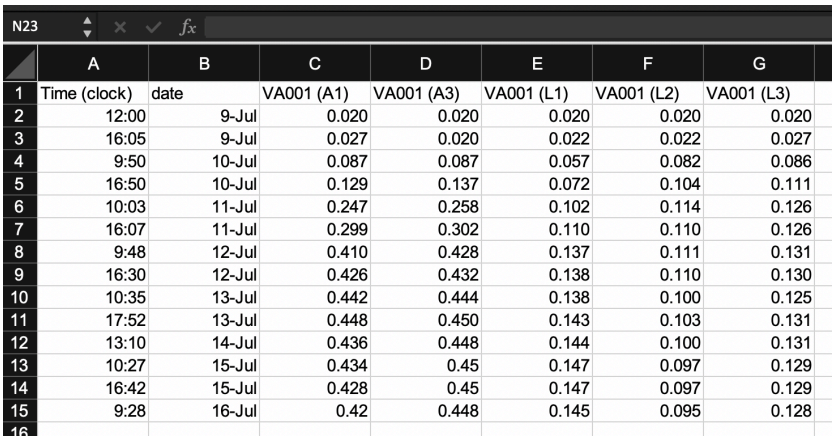
The time since the start of the experiment can easily be calculated later in the analysis pipeline, once you read the data into a statistical program like R. By delaying this step, you can both simplify the data collection template (requiring



	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										

Figure 2.11: 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.

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 2.12 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 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 2.12: 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 2.12.

A

	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

B

	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

C

	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

Figure 2.13: 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

Panel A (an exact repeat of the template shown in Figure 2.12) 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 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 2.14 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.

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 2.14, for example, the column names include spaces (for example, in "Date and time"), as well as parenthese (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,"aerated1" would work well, but "1–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

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

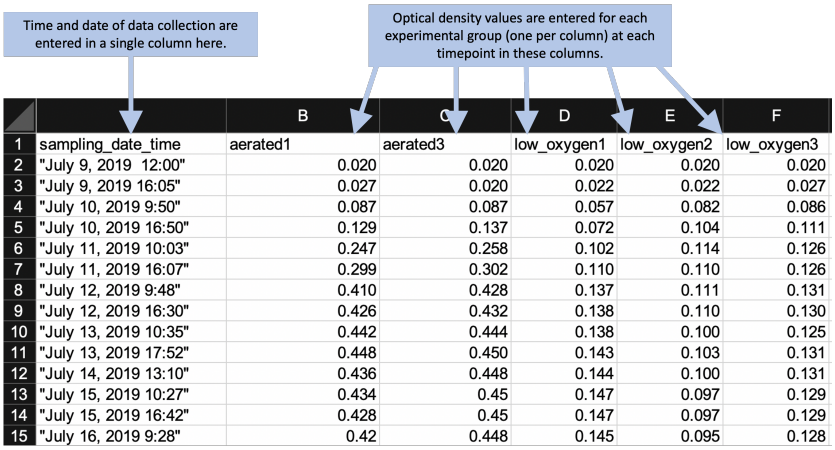
	A	B	C	D	E	F
1	Date and time	VA001 (A1)	VA001 (A3)	VA001 (L1)	VA001 (L2)	VA001 (L3)
2	"July 9, 2019 12:00"	0.020	0.020	0.020	0.020	0.020
3	"July 9, 2019 16:05"	0.027	0.020	0.022	0.022	0.027
4	"July 10, 2019 9:50"	0.087	0.087	0.057	0.082	0.086
5	"July 10, 2019 16:50"	0.129	0.137	0.072	0.104	0.111
6	"July 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

Figure 2.14: 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.

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 2.14) 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 2.15 shows the template for the example experiment after the col-

umn 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.



	A	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

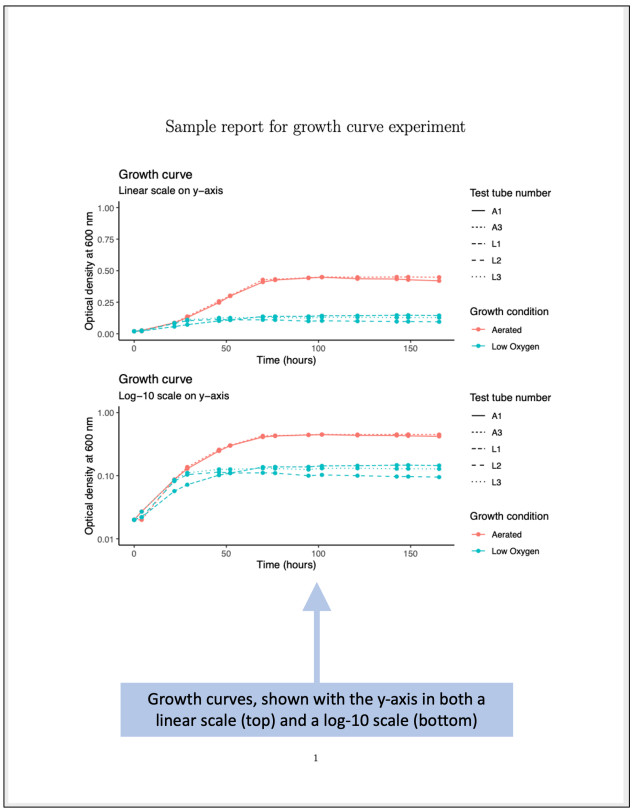
Figure 2.15: Example of an simpler format that can be used to record and analyze data for the same laboratory experiment as the previous figure. Annotations highlight where data is entered by hand. No calculations are conducted or figures created—these are all done later, using a code script.

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 2.16 shows an example of a report template that could be created to pair with the data collection template shown in Figure 2.15.

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'll walk through how you could create the report template shown in Figure 2.16. We will be using the programming language R, including RMarkdown, which is R's main tool for creating reproducible reports. If you have never used R or RMarkdown before, you'll find it useful to learn some more about their use to help in creating these types of reproducible reports from collected data. 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/`.

Older text



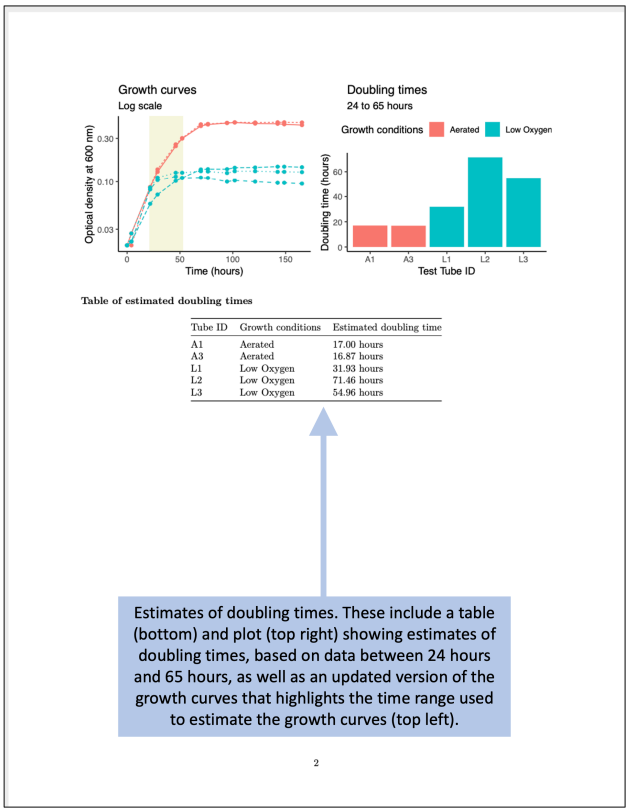


Figure 2.16: Examples of an automated report that can be created to quickly generate summaries and estimates of the data collected in the simplified data collection template for the example experiment.

2.5.6 Example—Data on rate of bacterial growth

The first set of data are from a study on the growth of *Mycobacterium tuberculosis*. The goal of this study was to compare growth yield and doubling time of *Mycobacterium tuberculosis* grown in rich medium under two assay conditions. One set of cultures were grown in tubes with a low culture volume relative to a large air head space to allow free oxygen exchange. A second set of cultures were grown in tubes filled to near capacity, resulting in limited air head space which has been shown elsewhere to limit oxygen availability over time. The caps on both sets of cultures were sealed to restrict air exchange during the study.

Some background information is helpful in understanding these example data, especially if you have not conducted this type of experiment. The increase in the cell size and cell mass during the development of an organism is termed growth. It is the unique characteristics of all organisms. The organism must require certain basic parameters for their energy generation and cellular biosynthesis. The growth of the organism is affected by both physical and nutritional factors. There are multiple methods by which growth can be measured, but the use of closed tissue culture tubes and a spectrophotometer to track increases in optical density (absorbance at 600 nm) over time offers several advantages: 1) it is less subject to technical error and contamination, 2) read out is fast and simple, 3) growth as measure by increased absorbance (turbidity) is directly proportional to increases in cell mass. There are four distinct phases of bacterial growth. Lag phase, log (exponential phase), stationary phase, death phase. From these data, bacterial generation times (doubling time) during the exponential growth phase can be calculated.

$$\text{Doubling time} = \frac{log(2)(t_1 - t_2)}{log(OD_{t_1} - log(OD_{t_2}))}$$

where t_1 and t_2 are two time points and OD_{t_1} and OD_{t_2} are the optical densities at the two time points (all logs are natural in this case).

An excel-based workbook (Figure 2.10) was created to allow the student performing the work 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. Columns were added to allow the student to track the time (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). Absorbance data for each sampling timepoint were listed in Columns E-F (high oxygen conditions; VA001 A1, A3) or columns G-I (limited oxygen conditions; VA001 L1, L2, L3).

What the researchers found appealing about the format of this Excel sheet was the ease with which the student 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 student to see tangible differences between the two assay conditions. This was also somewhat problematic as the equation to calculate doubling time was based on anchored time points built into the original spreadsheet resulting in two different results that were not properly linked to the correct data time points.

Data that are saved in a format like that shown in Figure 2.10, however, are hard to read in for a statistical program like R, Perl, or Python. In this format, 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. The spreadsheet also includes notes, automated figures, and cells where an embedded formula runs calculations behind the scenes.

Instead of this format, we can design a simpler format to collect the data. We'll remove all figures and calculations, and instead save those to perform in a code script. Figure **??** shows an example of a simpler format for collecting the same data. In this case, all the "extras" have been stripped out—this only has spaces for recording times points and the observed optical density at those time points. In later chapters, we'll show how a code script can be used to input these data into R and then perform calculations and create figures. By separating out the steps of data recording from data analysis, you can ensure that all steps of analysis are clearly spelled out (and can be easily reproduced with other similar data) through a code script. Note that you can still collect the data in this simpler format using a spreadsheet program, if you'd like—Figure 2.15 shows the data collection set up to be recorded in a spreadsheet program, for example. Within the spreadsheet, you can choose to save the data in a plain text format (a csv [comma-separated value] file, for example).

In this new data collection format, the data are not completely "tidy". This is because there is still some information included in the column names that we might want to use for analysis and plotting—namely, the different experimental group names (e.g., "aerated1", "low_oxygen1"). However, there is a balance in creating data collection spreadsheets. They should be in a format that is easy to read into an interactive programming environment like R, as well as in a format that will be easy to convert to a truly "tidy" format once they are read in. However, it's okay to balance these needs with aims to make the data collection spreadsheet easy for a researcher to use.

The example shown in Figure 2.15 is designed to be easy to use when collecting data. All data points for a single collection time are grouped together on a single row. When a researchers collects data for one time point, he or she can easy confirm visually that all the experimental groups have been measured for that time point. This format still makes it easy to read the data into an interactive programming environment, however, since they are in a clear two-dimensional format, with column names in the first row and values in the remaining rows. The removal of extraneous elements—like embedded formu-

las, the results of hand calculations or automated calculations, and annotations through notes or colored highlighting—remove barriers when reading the data into more sophisticated software. Once the data are read into R, there can be converted into a truly tidy data format with just a few command calls.

The following code shows an example of how easy it is to read data into R in the simplified format shown in Figure 2.15. It also shows how a few lines of code can then be used to convert the data into a truly "tidy" format, and how easily sophisticated plots can then be made with the data.

```
library("tidyverse")
library("readxl")

# Read data into R from the simplified data collection template
growth_curve <- read_excel("data/growth_curve_data_in_excel (1)/growth curve data_GR.xls",
                           sheet = "simplified_template")

# Example of data
growth_curve
```

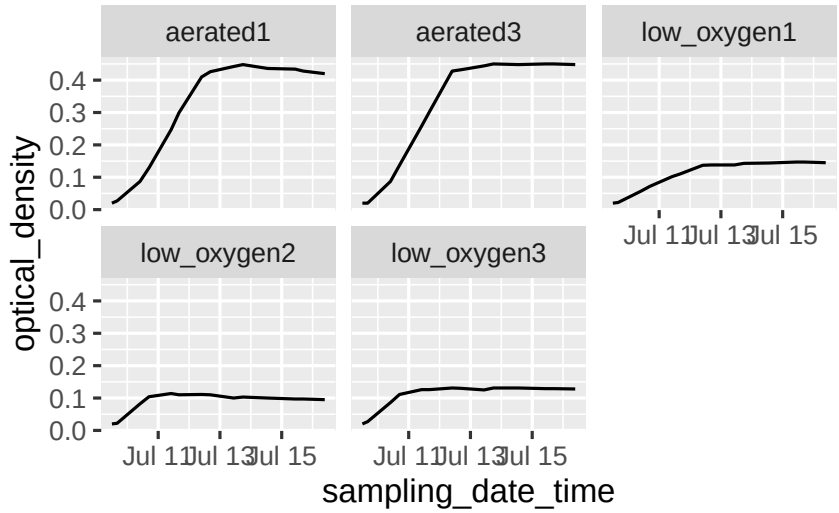
```
## # A tibble: 14 x 6
##    sampling_date_time  aerated1 aerated3 low_oxygen1 low_oxygen2 low_oxygen3
##    <dttm>                 <dbl>    <dbl>       <dbl>       <dbl>       <dbl>
##  1 2019-07-09 12:00:00    0.02     0.02        0.02        0.02        0.02
##  2 2019-07-09 16:05:00    0.027    0.02        0.022       0.022       0.027
##  3 2019-07-10 09:50:00    0.087    0.087       0.057       0.082       0.086
##  4 2019-07-10 16:50:00    0.129    0.137       0.072       0.104       0.111
##  5 2019-07-11 10:03:00    0.247    0.258       0.102       0.114       0.126
##  6 2019-07-11 16:07:00    0.299    0.302       0.11        0.11        0.126
##  7 2019-07-12 09:48:00    0.41     0.428       0.137       0.111       0.131
##  8 2019-07-12 16:30:00    0.426    0.432       0.138       0.11        0.13
##  9 2019-07-13 10:35:00    0.442    0.444       0.138       0.1         0.125
## 10 2019-07-13 17:52:00    0.448    0.45        0.143       0.103       0.131
## 11 2019-07-14 13:10:00    0.436    0.448       0.144       0.1         0.131
## 12 2019-07-15 10:27:00    0.434    0.45        0.147       0.097       0.129
## 13 2019-07-15 16:42:00    0.428    0.45        0.147       0.097       0.129
## 14 2019-07-16 09:28:00    0.42     0.448       0.145       0.095       0.128
```

```
# Convert to a fully tidy format
growth_curve <- growth_curve %>%
  pivot_longer(-sampling_date_time,
               names_to = "experimental_group",
               values_to = "optical_density")

# How the data look after this transformation
growth_curve
```

```
## # A tibble: 70 x 3
##    sampling_date_time  experimental_group optical_density
##    <dttm>              <chr>                         <dbl>
##  1 2019-07-09 12:00:00 aerated1                      0.02
##  2 2019-07-09 12:00:00 aerated3                      0.02
##  3 2019-07-09 12:00:00 low_oxygen1                   0.02
##  4 2019-07-09 12:00:00 low_oxygen2                   0.02
##  5 2019-07-09 12:00:00 low_oxygen3                   0.02
##  6 2019-07-09 16:05:00 aerated1                      0.027
##  7 2019-07-09 16:05:00 aerated3                      0.02
##  8 2019-07-09 16:05:00 low_oxygen1                   0.022
##  9 2019-07-09 16:05:00 low_oxygen2                   0.022
## 10 2019-07-09 16:05:00 low_oxygen3                   0.027
## # ... with 60 more rows
```

```
# Example of how easily sophisticated plots can be created with data in this format
growth_curve %>%
  ggplot(aes(x = sampling_date_time, y = optical_density)) +
  geom_line() +
  facet_wrap(~ experimental_group)
```



In later chapters, we'll discuss R's "tidyverse", a collection of tools within R that facilitate analyzing and visualizing data once they've been read into R. Here, we only aim to give an example of how little R code is needed to create useful output from the data, with the only requirement for gaining this power being that the data need to be collected in a format that is "tidy" or close enough to easily read into R.

2.5.7 Example—Data on bacteria colony forming units

2.5.8 Example—Data from multiple related experiments

2.5.9 Issues with these data sets

1. Issues related to using a spread sheet program
   - Embedded macros
   - Use of color to encode information
2. Issues related to non-structured / non-two-dimensional data
   - Added summary row
   - Multiple tables in one sheet
   - One cell value is meant to represent values for all rows below, until next non-missing row
3. Issues with data being non-"tidy"

2.5.10 Final "tidy" examples

2.5.11 Options for recording tidy data

Spreadsheet program.

Spreadsheet-like interface in R.

2.5.12 Examples of how "tidy" data can be easily analyzed / visualized

2.5.13 Discussion questions

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

…

For the next few modules, we'll use an example from a set of real immunology experiments. This was a a collection of studies with similar designs and similar goals—all are aiming to test candidate treatments for tuberculosis.

Each study tested one or more controls (e.g., saline solution, isofluorane, INH) and one or more treatments. Part of the novelty of this experiment was in testing some treatments that were monotherapies versus some that were combinations of several 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*. 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.

2.7 Details of the data from the set of example studies

Let's start by taking a closer look at the set of studies that we'll use as a motivating example in this module. Figure [x] provides an example of the data that were collected for one of the studies. Each treatment is coded with a numeric ID (e.g., a 0 in the `rx_group` column indicates one of the control treatments). A different spreadsheet contains metadata on each of the treatment regimes (Figure [x]). For each treatment that was tested in this study, bacterial load was measured and recorded for each mouse given the treatment, with (in this study in the set) between five and eight mice given each treatment.

One other piece of information was recorded in the laboratory for this study. Each week of treatment, all the mice under a given treatment were weighed as a group, and these weights were recorded (Figure [x]). These values can be divided by the number of mice in that treatment group to determine the average weight per mouse each week of treatment. These data were recorded within the same data collection spreadsheet as the data on bacterial loads, but in a different sheet of that spreadsheet.

This same data collection process we conducted for 19 different studies. All the data can be collected using the same data collection template. A separate spreadsheet was used to record metadata about the set of experiments. This included details on each treatment, using numeric treatment IDs that correspond with those used in the data collection template shown in Figure [x].

…

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.

2.8 Creating an organized directory for project files

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

2.9 Making the project directory and R Project

Old text

2.9.1 Organizing project files through the file system

One of the most amazing parts of how modern computers work is their file directory systems. [More on these.]

It is useful to leverage this system to organize all the files related to a project. These include 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.

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 this type of version control—this is one example of this more dynamic way of sharing files within a directory.

2.9.2 Organizing files within a project 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.

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 "raw_data", and consistently call it "raw_data", to store data directly from observations or directly output from laboratory equipment. You may want to include subdirectories in that "raw_data" subdirectory for each type of data—maybe a "cfu" subdirectory, for example, with results from plating data to count colony forming units, and another called "flow" for output from a flow cytometer. By using the same structure and the same subdirectory names, you will find that code scripts are easier to reuse from one project to another. Again, most scripting languages allow you to leverage 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.

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 a immunology research laboratory.

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.

2.9.3 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 chosing "File" -> "New Project" in RStudio's menu. This gives you the option to create a project from scratch or to make an existing directory and 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, includuing … Also, when you open one of these Projects in RStudio, it will move your working directory into that projects 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 on 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. The next module gives step-by-step directions for making a directory an RStudio Project, and also how to create you own RStudio Project template to quickly create a new directory for project files each time you start

a new research project.

[Visual—project directory as a *mise en place* for cooking—everything you need for the analysis, plus the recipe for someone to repeat later.]

[Reference: The Usual Suspects—you'll typically have the same types of data files, analysis, types of figures, etc., come up again and again for different research projects. Leverage tools to improve efficiency when working with these "usual suspects". The first time you follow a protocol that is new to you, or the first time you cook a recipe, it takes much longer and much more thought than it should as you do it over and over—there are some recipes where I only use the cookbook now to figure out the oven temperature or the exact measurement of an ingredient. These tools will help you streamline your project file organization and move towards reuse of modular tools and ideas (e.g., remembering how to make a vinaigrette and applying that regardless of the type of salad) across projects.]

[Analogies for moving to do things more programatically—Tom Sawyer outsourcing the fence painting, sorcerer's apprentice (all the mops, plus some difficulties when you first start, before you get the hang of it).]

[File extensions give an idea of the power of consistent file names. While some operating systems don't require these, by naming all the files that should be opened with, for example, Word ".docx", the operating system can easily do a targeted search that looks for files with certain key words in the name while limiting the search only to Word files. You can leverage this same power yourself, and in a way that's more customized to your project or typical research approach, by using consistent conventions to name your files.]

2.9.4 Subsection 1

One study 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)

> "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 om shelves in a library, so files in different directories can have the same name without any conflict. Generally, each user haas 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 aa 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 modifications 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)

2.9.5 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 differet 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 you 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 currenlty 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)

"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." (Buffalo, 2015)

"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)

"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." (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 you directory's hierarcy … 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 you 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 ocuntless 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 USCS 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 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 programatically means doing it through code rather than manually, using a method that can effortlessly scale to multiple objects (e.g., files). Programatically referring to multiple files is easier and safer than typing them all out (because it's less error prone.)" (Buffalo, 2015)

"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 programatically using the Unix shell and other programming languages." (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 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 outdate 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." (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 profesionally 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)

2.9.6 Practice quiz

2.10 Creating 'Project' templates

Researchers can use RStudio's 'Projects' can 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.

2.10.1 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.

2.10.2 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 specificially 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 2.17.

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

Figure 2.17: 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'.

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 2.17, for example, one of the template options is for a "Book Project using bookdown", available because the `bookdown` R package has been installed locally.

Your research group can create your own Project templates. 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 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)

2.10.3 Discussion questions

2.11 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

2.11.1 Subsection 1

2.11.2 Subsection 2

2.11.3 Applied exercise

2.12 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, while maintaining the power to 'backtrack' to previous versions, messages included to explain changes. 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

2.12.1 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.

> "Tracking all that detail is just the sort of thing computers are good at and humans are not." (Raymond, 2003)

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 and 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, provided *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 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, 2018; 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.

> "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 bo back to the March version of my code and I can recreate it'." (Perkel, 2018)

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 occassionally 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 members 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 scienctific projects is *git*. This program was created by Linus Torvalds, who also created the Linux operating system, 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 can helps in collaborating on file-based projects, including *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, all of which are discussed in the next chapter as tools that 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).

> "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." (Target, 2018)

2.12.2 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).

> "Handwritten lab notebooks are usually chaotic and always unsearchable." (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 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.

> "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." (Kwok, 2018)

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.

> "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." (Tippmann, 2014)

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 addititions and substractions 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 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 let's 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 DOI 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, 2018). 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 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)

> "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*." (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." (Perez-Riverol et al., 2016)

2.12.3 Discussion questions

> "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)

2.13 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*)

2.13.1 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 attachements 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)

> "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." (Raymond, 2003)

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

> "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." (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.

> Resources like GitHub are "essential for collaborative software projects because they enable the organization and sharing of programming tasks between different remote contributors." (Perez-Riverol et al., 2016)

2.13.2 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, resources for learning the details of 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 of 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 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.

> "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." (Perez-Riverol et al., 2016)

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).

> "[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 guarnateeing that reversion to a known-good state will always be easy." (Raymond, 2003)

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, tools we discuss in other parts of this book.

> "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." (Grune, 1986)

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 occassionally 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 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 thing of this as just merging the *change*s 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 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.

> "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." (Blischak et al., 2016)

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.

> "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?" (Raymond, 2003)

In recent years, some complementary tools have been developed that make the process of collaborating together using version control software easier.

Other tools can helps in collaborating on file-based projects, including *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, all of which are 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. Thethe 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 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.

> "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." (Perez-Riverol et al., 2016)

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, 2018) 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.

> "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 each 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." (Irving, 2011)

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.

> "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 diaster 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." (LEIPS, 2010)

2.13.3 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 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.

2.13.4 Discussion questions

2.14 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

2.14.1 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 lanugage 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).

2.14.2 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 managemment.

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 2.18), 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

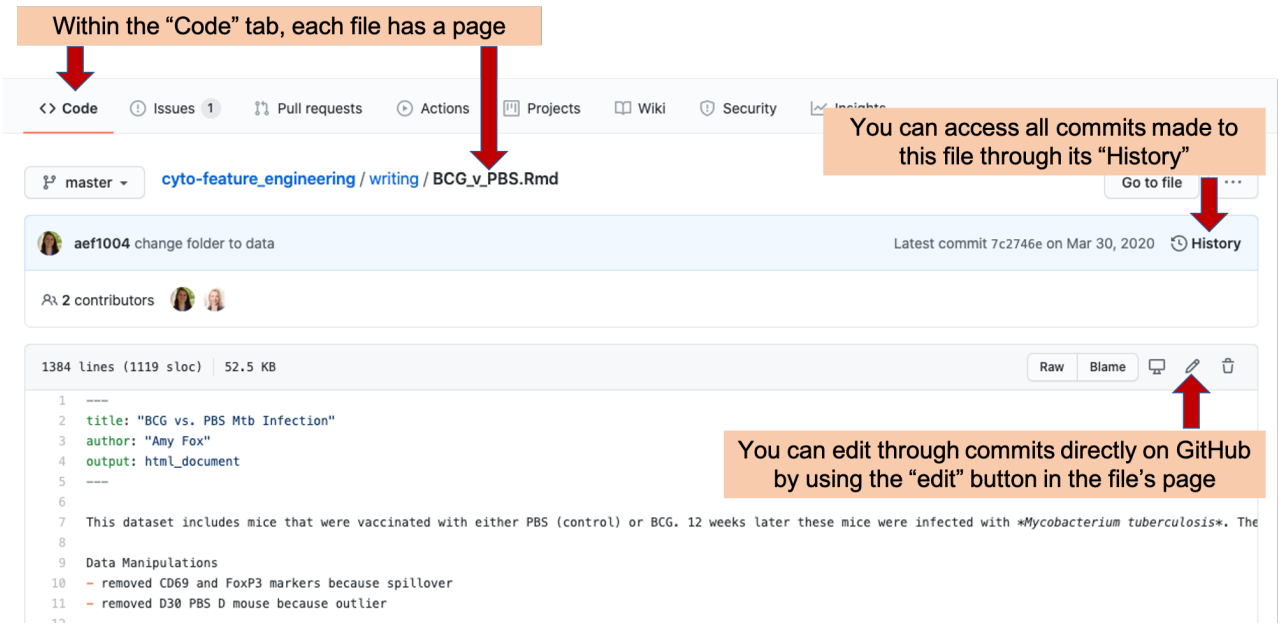


Figure 2.18: Example of a file page within a GitHub repository. Each file in a repository has its own page. On this page, you can see the history of changes made to the file by looking at ’History’. You can also make a commit an edit directly in GitHub by clicking on the ’Edit’ icon.

is an “Edit” icon (Figure 2.18). By clicking on this, you will get to a page where you can directly edit the file (Figure 2.19). 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 2.19).

You can see the full history of changes that have been made to each file in the project (Figure 2.20). 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 2.20, it will take you to a page providing information on the changes made with that commit (Figure 2.21). 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, 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.

Issues

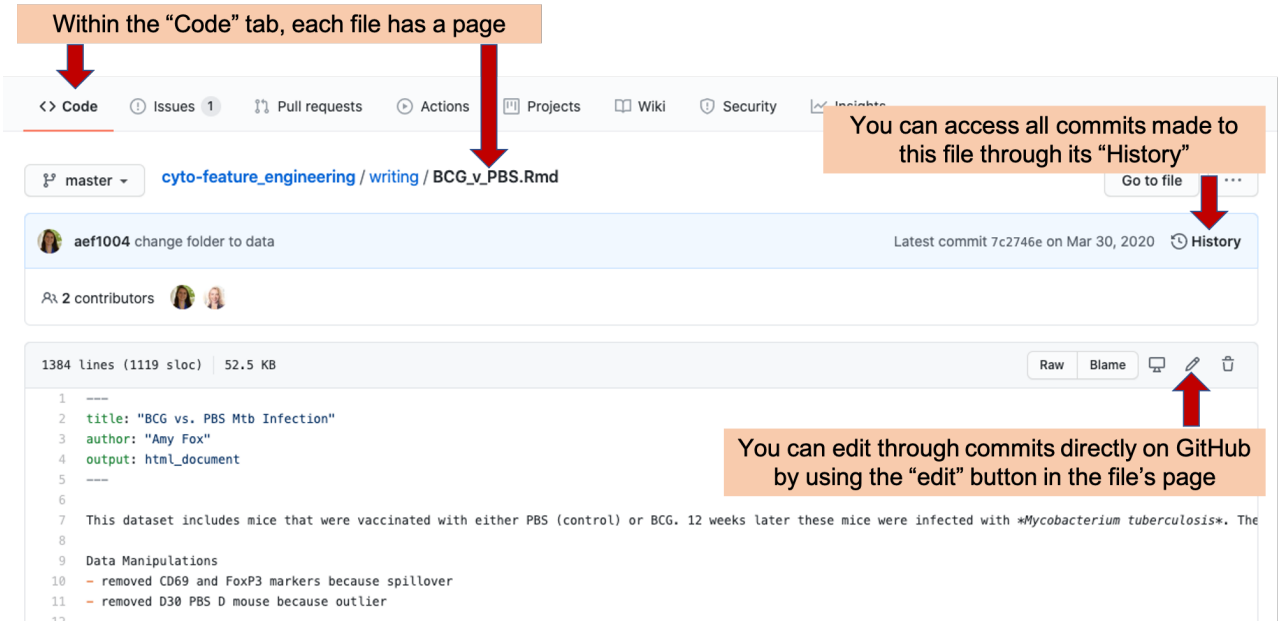


Figure 2.19: Committing changes directly in GitHub. When you click on the ’Edit’ button in a file’s GitHub page (see previous figure), it will take you to a page where you can edit the file directly. You save the changes a commit message

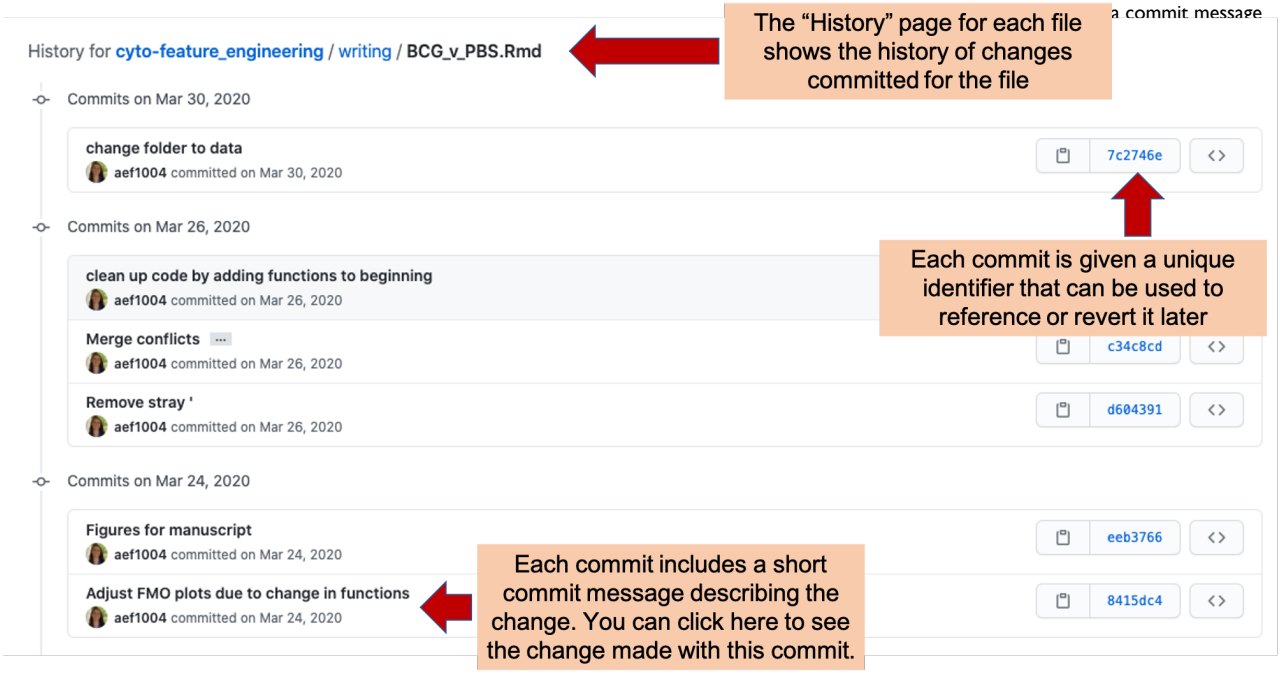


Figure 2.20: Commit history in GitHub. Each file in a repository has a ’History’ page, where you can explore each change commited for the file. Each commit has a unique identifier and commit message describing the change. You can click on the entry for any of these commits to see the changes made to the file with the commit (see next figure).

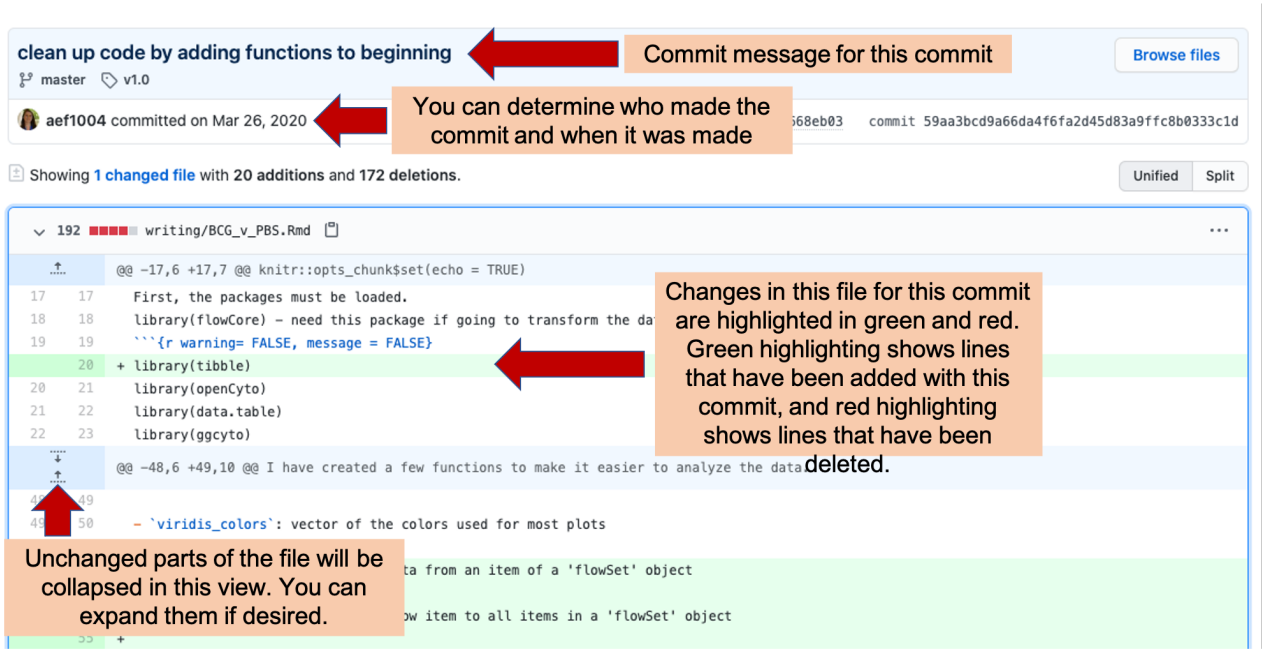


Figure 2.21: Commit history in GitHub. Each commit has its own page, where you can explore what changes were made with the commit, who made them, and when they were committed.

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 2.22 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 2.23 for an example of this button).

On the page for a specific issue (e.g., Figure 2.23), 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 not fully resolved.

The Issues tracker page includes some more advanced functionality, as well (Figure 2.24). For example, you can "assign" an issue to one of more team members, indicating that they are responsible for resolving that issue. You can also tag each issue with one of more labels, allowing you to group issues into

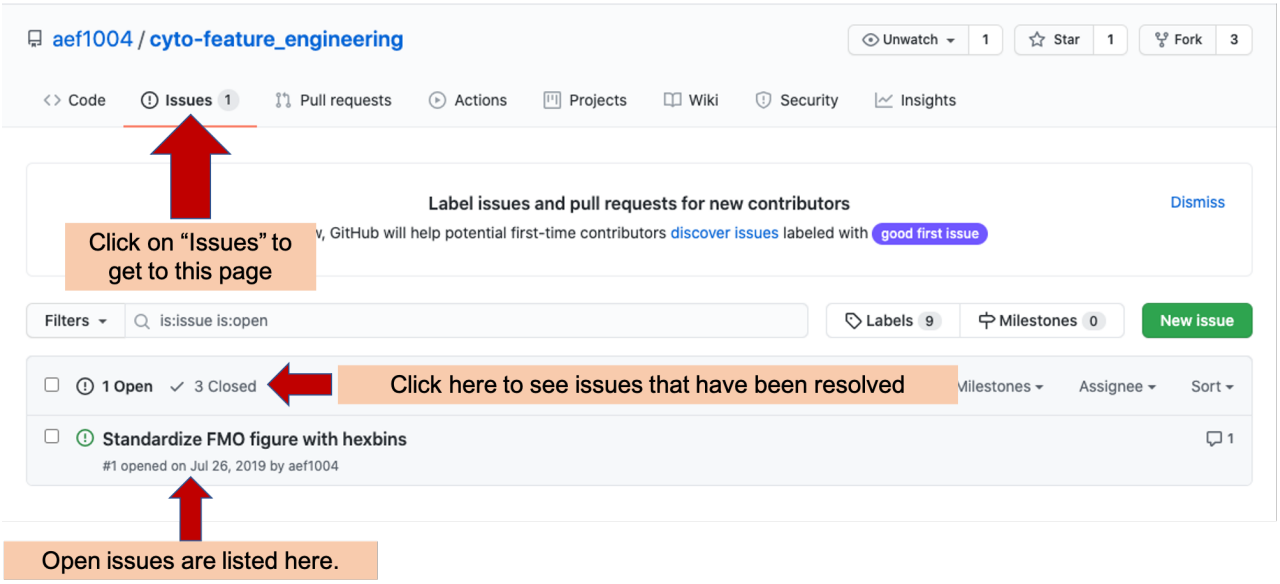


Figure 2.22: Issues tracker page for an example GitHub repository. Arrows highlight the tab to click to get to the Issues tracker page in a repository, as well as where to

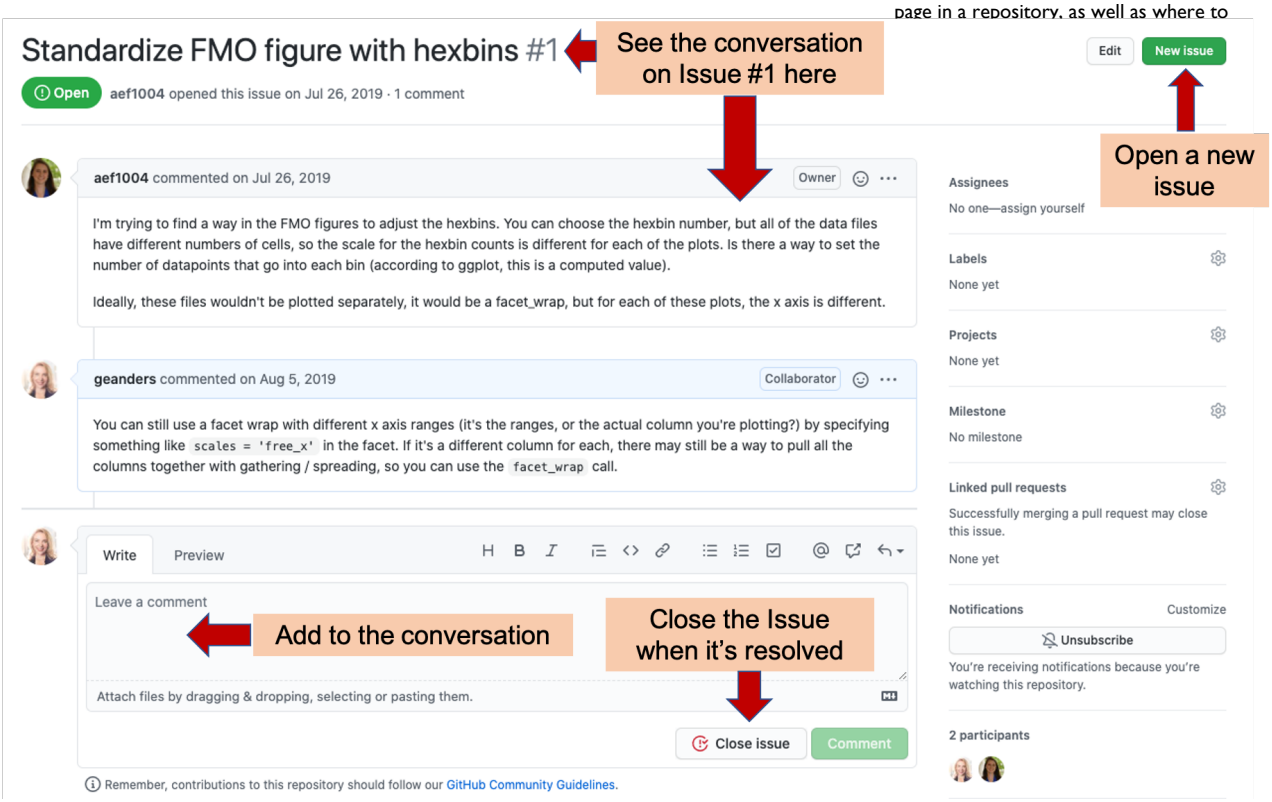


Figure 2.23: Conversation about an Issue on Issues tracker page of an example GitHub repository. In this example, you can see how GitHub Issues trackers allow you to discuss how to resolve an issue across your team. From this page, you can read the current conversation about Issue #1 of the repository 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

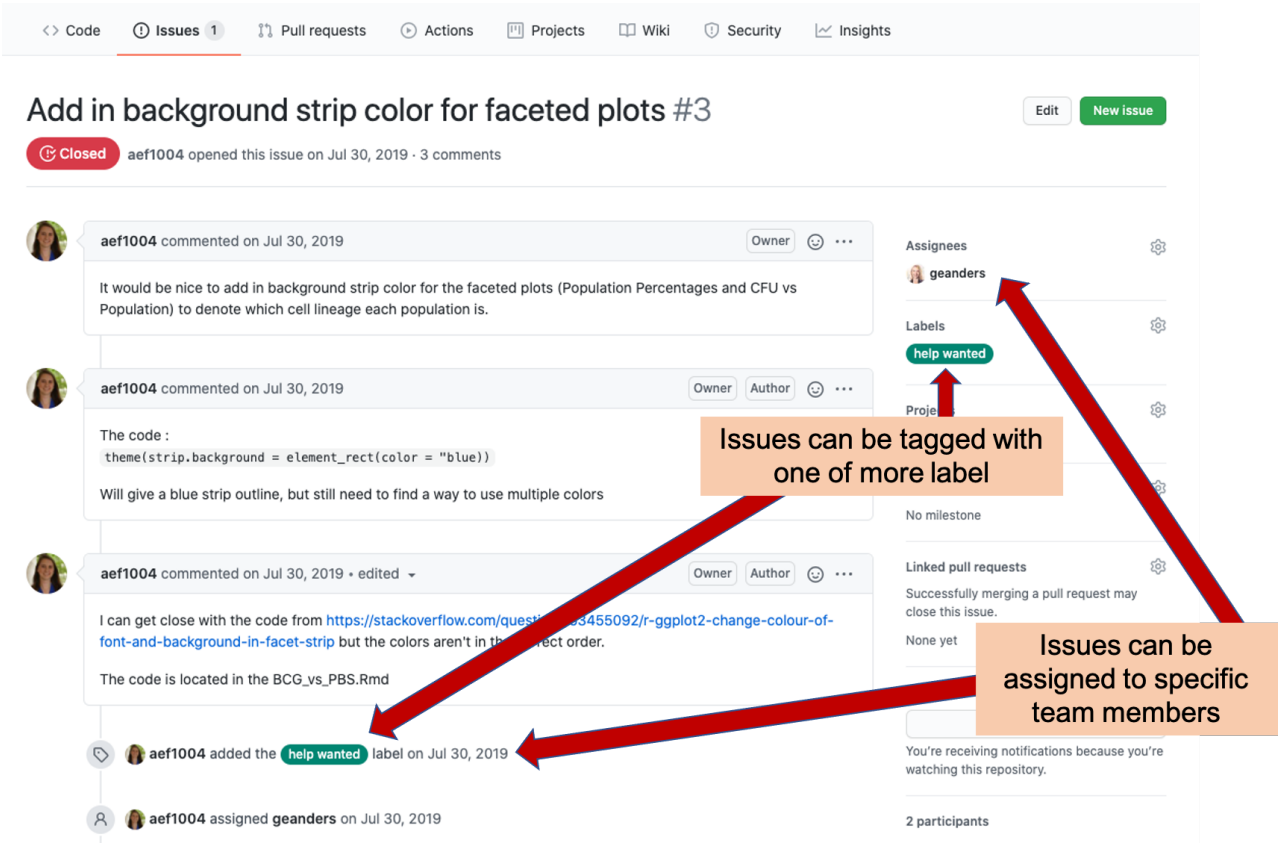


Figure 2.24: Labeling and assigning Issues. The GitHub Issues tracker allows you to assign each issue to one or more team members, clarifying that they will take the lead in resolving the issue. It also allows you to tag each issue with one or more labels, so you can easily navigate to issues of a specific type or identify the category of a specific issue.

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

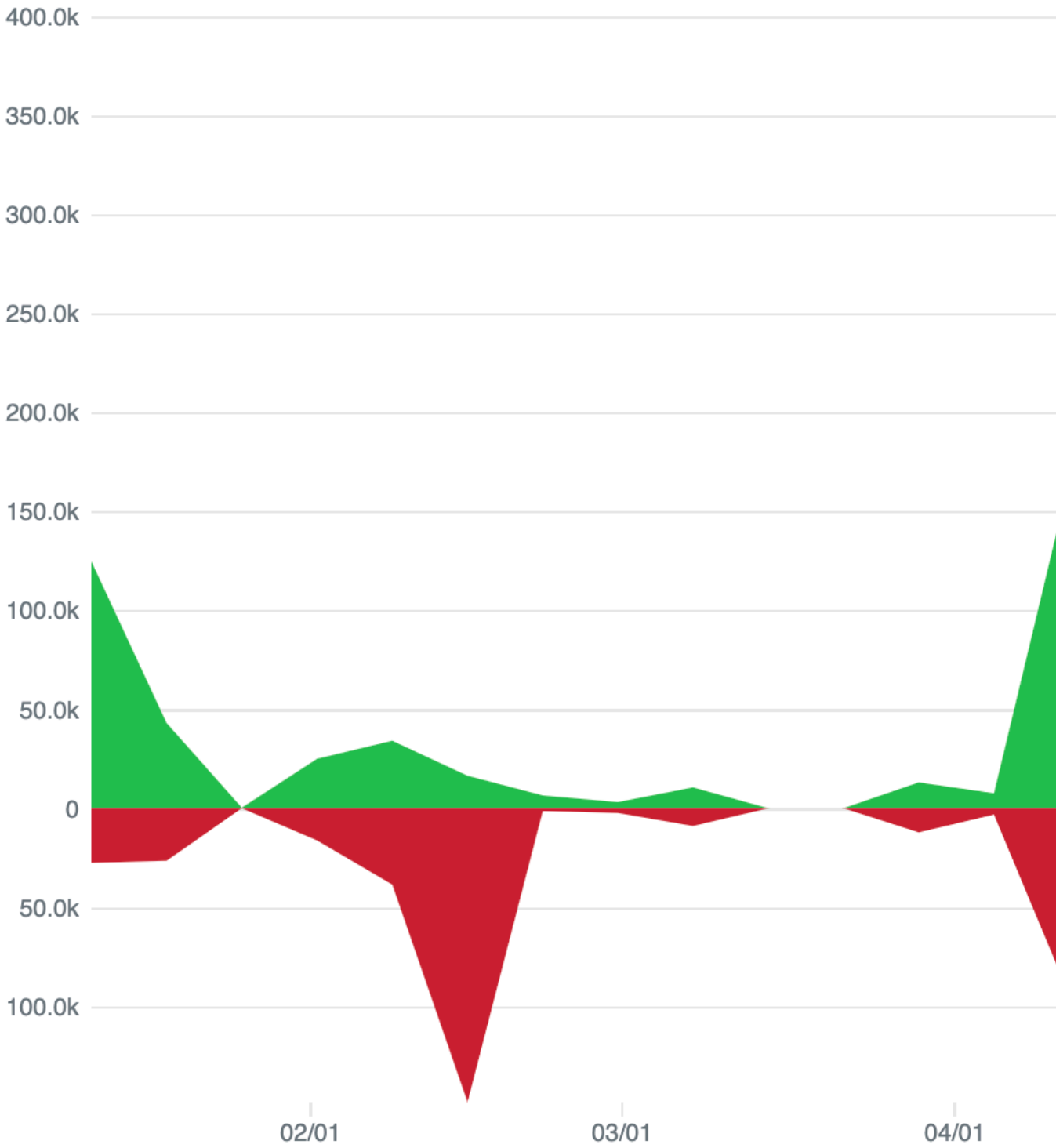
Repositories include functionality for inviting team members, assigning roles, and otherwise managing access to the repository. First, a repository can be either public or private. For a public repository, anyone will be able to see the full contents of the repository through GitHub. You can also set 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 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

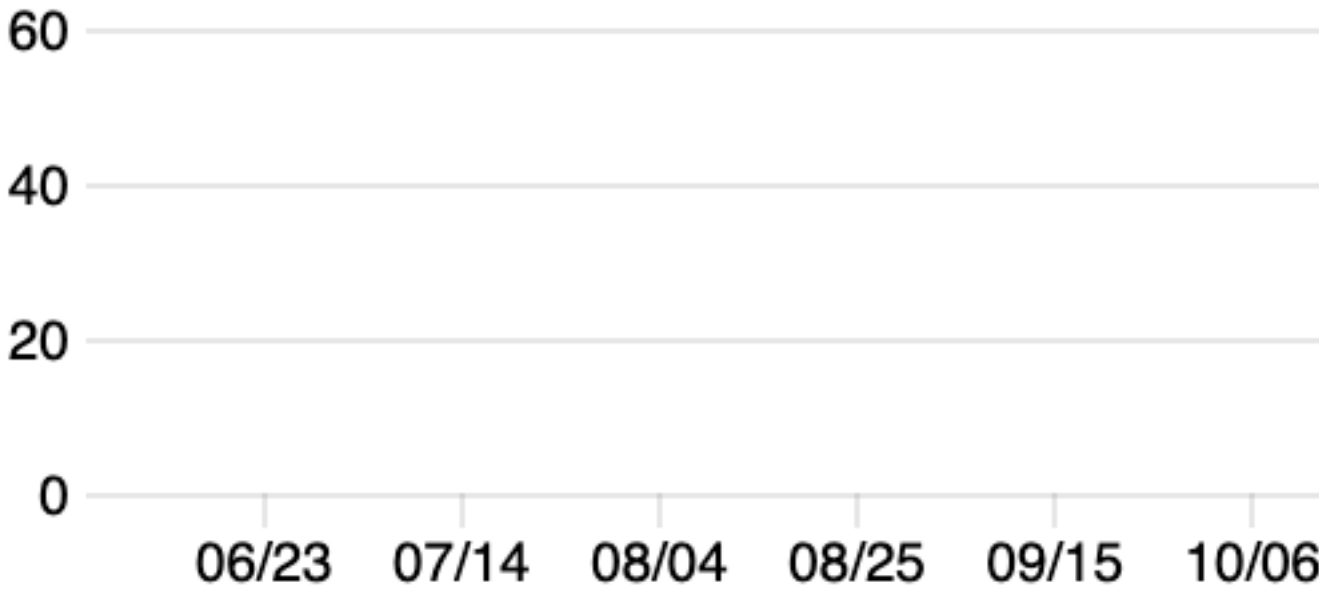
Each GitHub repository also provides an "Insights" page, which lets you see who is contributing to the project and, as well when and how much they have contributed, as tracked by the commits they've made.

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]).


400.0k
350.0k
300.0k
250.0k
200.0k
150.0k
100.0k
50.0k
0
50.0k
100.0k
02/01
03/01
04/01

During periods when the research team is collecting data, you would expect a lot more additions that 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 delections 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 web-site 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.

2.14.3 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 preprocessing, 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 repostiory 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)

2.14.4 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 What is pre-processing?

Some data collected through laboratory experiments is very straightforward and requires little or no pre-processing before it's used in analysis. For example, [example]. Other data may require some minimal pre-processing. 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.

This step of pre-processing data can become much more complex with data that was collected using complex equipment, like a flow cytometer or a mass spectrometer. 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. For data collected from a flow cytometer, preprocessing may include steps to disentangle the florescence from different markers to ensure that the read for one marker isn't inflated by spillover florescence from a different marker.

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.]

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.

3.1.2 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.

3.1.3 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 be-

comes 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 tranform 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.

3.1.4 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 predictible 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.]

3.1.5 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 advntge of a new tech-

nique 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 sorcere'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 system 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 bioinformatices 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 lanuage 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 lanugage 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 statitical 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 explicityly 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 fo R and any R pakcages 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 incrediably 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 pakcage 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 availablility 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 subpopulation 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, 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 taxa-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 mode 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 substition 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 dataframe 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 truely 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 = log2(n + 1) or more generally y = log2(n + n0)." (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 becuase 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 depthy 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 proproational 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 identially distributed (and, hence, homoscedastic) noise." (Holmes and Huber, 2018)

"We distinguis 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 specificiations 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 profesionally 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 experession 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 PC1 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)

3.1.6 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 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.

3.2.2 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.

3.2.3 Process of building a code script

A code script is essentially a recipe for cleaning and analyzing data.

3.2.4 Applied exercise

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 Limitations of object-oriented programming

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 premade 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.

3.3.2 The "tidyverse" 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.

The centralizing principal of the tidyverse approach is the format in which data is stored throughout "tidyverse" coding—the tidy dataframe. We've described this data type, including its rules and advantages, in an earlier module of this book. 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. 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.

Sometimes, you'll see that data in a tidyverse approach are stored in a special type of dataframe called a "tibble"—this isn't very different from a dataframe, and in fact is a special type of dataframe. 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 dataframe. By default, it will only print the first few lines. By contrast, a dataframe 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 dataframe and the data type of each column in the data.

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.

3.3.3 How to "tidyverse"

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 now widely taught, both in in-person courses at universities and through a variety of online resources. 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/`).

[More on other resources for learning the tidyverse.]

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.

3.3.4 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 languagae (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 thata systems made up of many simple pieces that can be roped 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 Unit (iOS and Andriod), 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 eremged 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 lear, 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 (discs, 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 compiles, editors, command languages, programs for copying and printing files, and so on. Still more broadly, 'Unix' may even include programs develpoed 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)

3.3.5 Subsection 2

3.3.6 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

3.4.1 Introduction

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 pipelin 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.

3.4.2 Data structures and the dataframe

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.

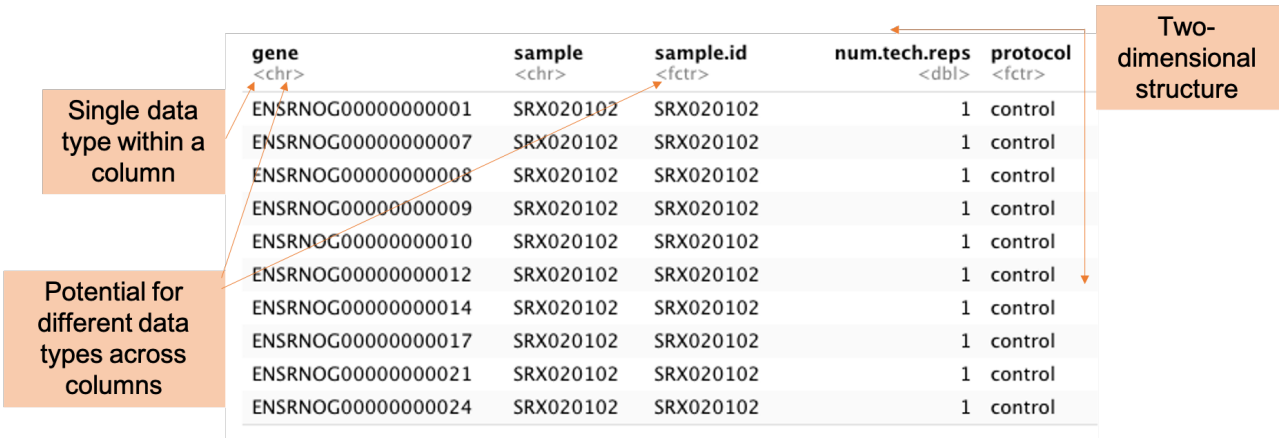


Figure 3.1: An example of the dataframe data structure. This data structure is the most frequently used data structure within the tidyverse approach, and its use is in fact a defining element of the approach.

Dataframes like this 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.

By 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 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.

3.4.3 More complex data structures 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 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 measure-

ment 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 pre-processing 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.

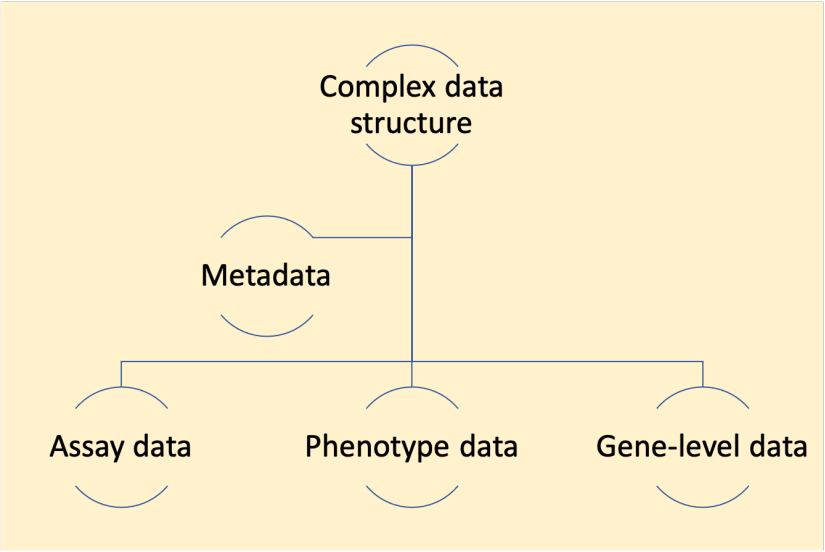


Figure 3.2: An example of different components of data that may need to be stored in a data structure that is more complex than a dataframe. Data from a genomic experiment may include data from several levels, including metadata that describes the entire experiment, as well as assay data, phenotype data, 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.

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.

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 store 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 exper-

iment. 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]

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 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

"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]

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?]

"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 vectors are implicitly and silently copied to achieve copy-on-write semantics, contributing to high memory usage and poor performance."
[@lawrence2014scalable]

"Our ultimate goal is to process and summarize a large data set in its entirety, and iteration enables this by limiting the resource commitment at a given point in time. Limiting resource consumption generalizes beyond iteration and is a fundamental technique for computing with big data. In many cases, it may render iteration unnecessary. Two effective approaches for being frugal with data are restriction and compression. Restriction means controlling which data are loaded and lets us avoid wasting resources on irrelevant or excessive data. Compression helps by representing the same data with fewer resources."

3.4.4 Limitations to complex data structures

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 dataframe 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.

"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]

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.

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 characteris-

tics 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.

3.4.5 Complex versus simple structures for storing data

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 spectometry 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.

By contrast to these more complex data formats, 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]

As you learn R, you will almost certainly learn how to create and work with these more general data formats, 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.

3.4.6 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 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 dataframe, 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.

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 *type*s 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* (*bi*nary di*gits*). 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.

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 equipement 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's 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 mecahnisms 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 you 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 differentlyin [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 different: `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)

3.5.2 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 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 thing 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. … Techinally, 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 componenet 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 used 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 audient. 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., Summarized Experiment, 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)

3.5.3 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 amenible 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). …

3.5.4 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 everyoen 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 inot 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 explonentially 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 that 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 coule 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 paramter 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 indes, `[[1]]`, followed by the contents of this component…The second component of the list starts with the component index `[[2]]` followed by the contenets 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 ofter 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 reatures 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 elemeents 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()` sill 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 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 lenth 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 langauge 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 wehre programmers more familiar with other languages fail to make appropriate use of the available functionality. … There are slight differents between subsetting of vectors, arrays, lists, data.frames, and enviroments 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 fo 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 tiem), 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 moerately 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 *shap*e 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 *heterogenesou*); 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 withstr()'s second argument,max.level. By default,max.levelisNA`', 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)$entrezId returns the NCBI Entrez Gene identifiers of the features, and se$tissue 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 rhdf5—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, rectagular 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 coordinated. 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'etre 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 lables 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 approapriate 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 anlayses 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. It's 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 and example of an awkward way of combining bach 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 informaiton 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 example 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 `validObject` 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 revlevant to each of your observations handy within a dataframe—a structure than 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 as 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 allow 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 accomplished 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 based 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 (renetion 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). j

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.all_vars"
## [4] "print.anova"       "print.ansi_string" "print.ansi_style"
```

```
## [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)

3.5.5 Subsection 2

3.5.6 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 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 workflow 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.

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 com-

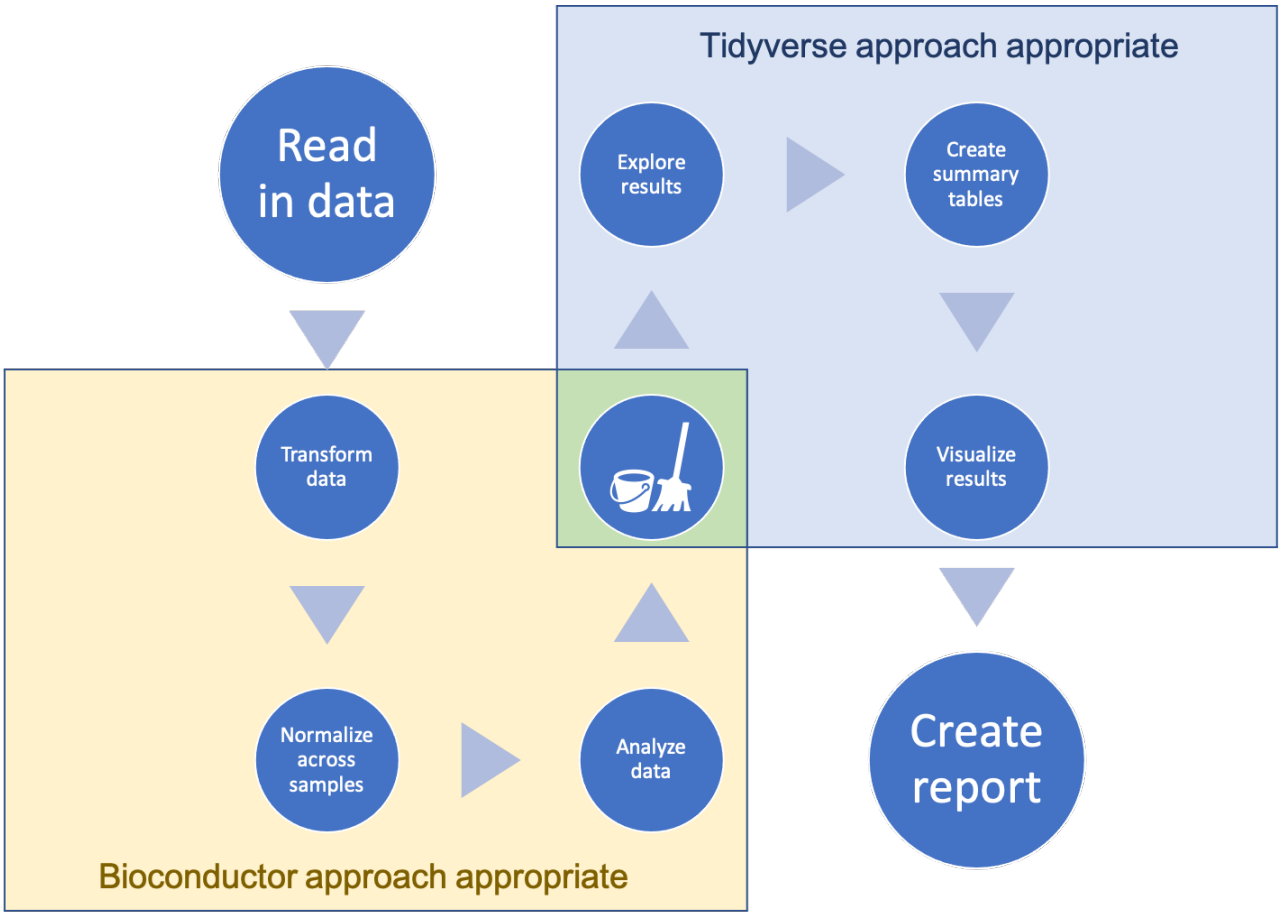


Figure 3.3: An overview of a workflow that moves from a Bioconductor approach—for pre-processing of the data—through to a tidyverse approach one pre-processing has created smaller, simpler data that can be reasonably stored in a dataframe structure.

mon bioconductor classes and into tidy dataframes. this package drawers 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 tiny 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.

3.6.2 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 …
- `DGEExact` objects, which are used …
- [limma objects]
- [ExpressoinSet 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 dataframe, 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 `DGEExact` 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 ENSRNOG00000000001 SRX020102     2
##  2 ENSRNOG00000000007 SRX020102     4
##  3 ENSRNOG00000000008 SRX020102     0
##  4 ENSRNOG00000000009 SRX020102     0
##  5 ENSRNOG00000000010 SRX020102    19
##  6 ENSRNOG00000000012 SRX020102     7
##  7 ENSRNOG00000000014 SRX020102     0
##  8 ENSRNOG00000000017 SRX020102     4
##  9 ENSRNOG00000000021 SRX020102     7
## 10 ENSRNOG00000000024 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] "DGEExact"
## attr(,"package")
## [1] "edgeR"

tidy(et)

## # A tibble: 29,516 x 4
##    gene               estimate  logCPM    p.value
```

```
##    <chr>                 <dbl>   <dbl>      <dbl>
##  1 ENSRNOG00000000001   2.65    1.49   0.00000131
##  2 ENSRNOG00000000007  -0.409  -0.226  1
##  3 ENSRNOG00000000008   2.22   -0.407  0.129
##  4 ENSRNOG00000000009   0      -1.31   1
##  5 ENSRNOG00000000010   0.0331  1.79   1
##  6 ENSRNOG00000000012  -3.39    0.0794 0.00375
##  7 ENSRNOG00000000014   3.65   -0.854  0.252
##  8 ENSRNOG00000000017   2.42    1.11   0.0000638
##  9 ENSRNOG00000000021  -2.02    0.211  0.0373
## 10 ENSRNOG00000000024   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
## ENSRNOG00000000001         2         4        18
## ENSRNOG00000000007         4         1         3
## ENSRNOG00000000008         0         1         4
## ENSRNOG00000000009         0         0         0
## ENSRNOG00000000010        19        10        19
## ENSRNOG00000000012         7         5         1
## ENSRNOG00000000014         0         0         2
## ENSRNOG00000000017         4         1        12
## ENSRNOG00000000021         7         5         2
## ENSRNOG00000000024        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.

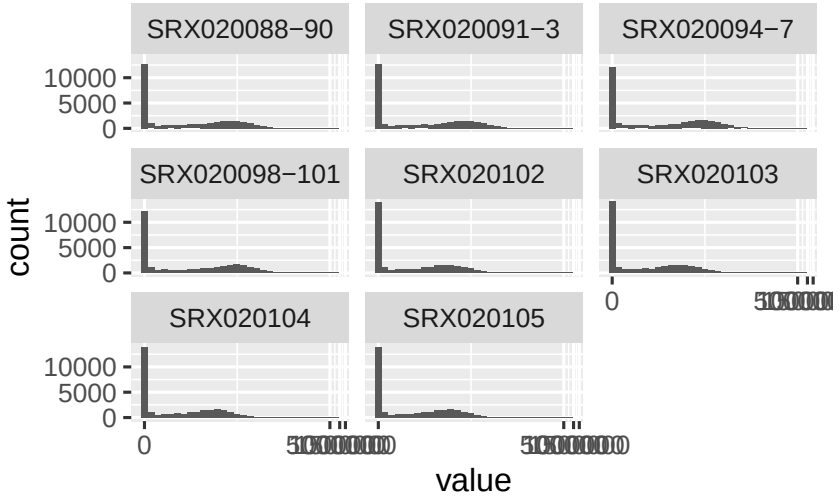
These data are structured and stored in such a way that they have some of the characteristics that can make data difficult to work with the data using tidyverse tools. For example, they store gene identifiers in the rownames, rather than in a separate column where they can be easily accessed when using 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 ENSRNOG00000000001 SRX020102     2
##  2 ENSRNOG00000000007 SRX020102     4
##  3 ENSRNOG00000000008 SRX020102     0
##  4 ENSRNOG00000000009 SRX020102     0
##  5 ENSRNOG00000000010 SRX020102    19
##  6 ENSRNOG00000000012 SRX020102     7
##  7 ENSRNOG00000000014 SRX020102     0
##  8 ENSRNOG00000000017 SRX020102     4
##  9 ENSRNOG00000000021 SRX020102     7
## 10 ENSRNOG00000000024 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`.
```

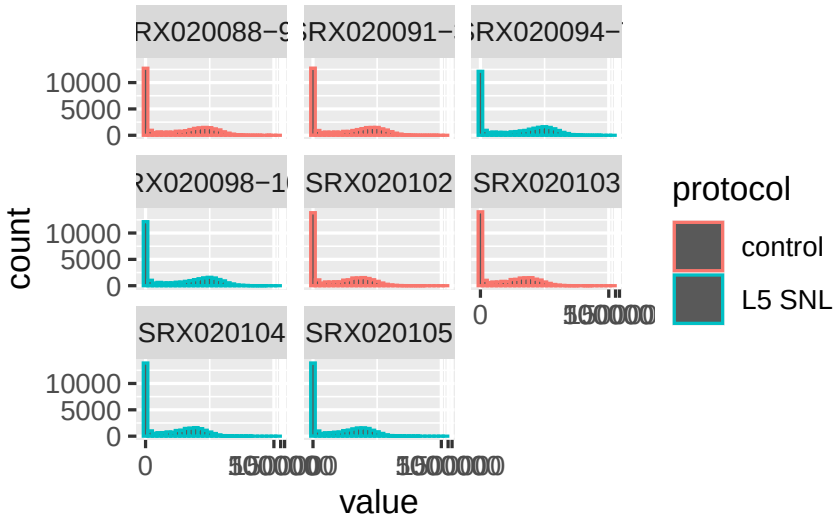


You can also incorporate data that are stored in the `phenoData` slot of the `ExpressionSet` object by specifying `addPheno = TRUE`:

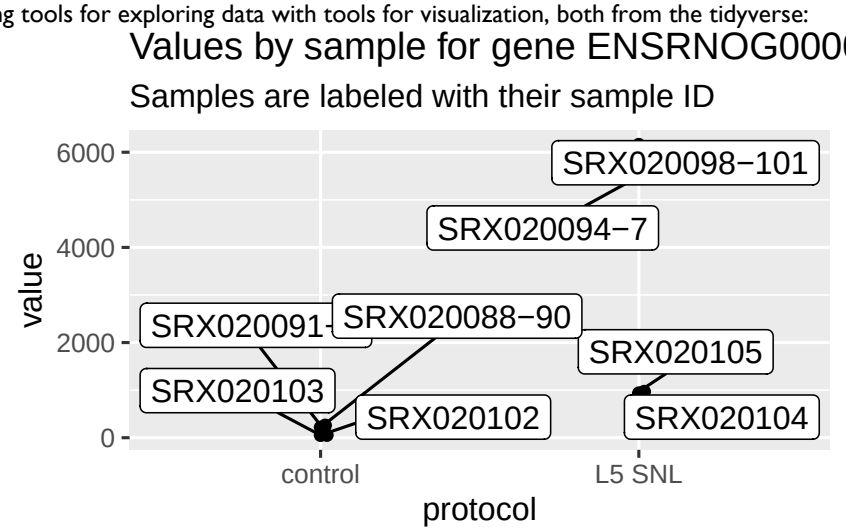
```
## # A tibble: 236,128 x 8
##    gene               sample sample.id num.tech.reps protocol strain Time  value
##    <chr>              <chr>  <fct>             <dbl> <fct>    <fct>  <fct> <int>
##  1 ENSRNOG00000000001 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     2
##  2 ENSRNOG00000000007 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     4
##  3 ENSRNOG00000000008 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     0
##  4 ENSRNOG00000000009 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     0
##  5 ENSRNOG00000000010 SRX02~ SRX020102             1 control  Sprag~ 2 mo~    19
##  6 ENSRNOG00000000012 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     7
##  7 ENSRNOG00000000014 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     0
##  8 ENSRNOG00000000017 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     4
##  9 ENSRNOG00000000021 SRX02~ SRX020102             1 control  Sprag~ 2 mo~     7
## 10 ENSRNOG00000000024 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:



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 `DGEExact` class.

```
## [1] "DGEExact"
## attr(,"package")
## [1] "edgeR"
```

The `DGEExact` 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
## ENSRNOG00000000001  2.64635814  1.49216267 1.309933e-06
## ENSRNOG00000000007 -0.40869816 -0.22616605 1.000000e+00
## ENSRNOG00000000008  2.22296029 -0.40665547 1.288756e-01
## ENSRNOG00000000009  0.00000000 -1.31347471 1.000000e+00
## ENSRNOG00000000010  0.03307909  1.79448965 1.000000e+00
## ENSRNOG00000000012 -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 experssion 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 ENSRNOG00000000001   2.65    1.49   0.00000131
##  2 ENSRNOG00000000007  -0.409  -0.226  1
##  3 ENSRNOG00000000008   2.22   -0.407  0.129
##  4 ENSRNOG00000000009   0      -1.31   1
##  5 ENSRNOG00000000010   0.0331  1.79   1
##  6 ENSRNOG00000000012  -3.39    0.0794 0.00375
##  7 ENSRNOG00000000014   3.65   -0.854  0.252
##  8 ENSRNOG00000000017   2.42    1.11   0.0000638
##  9 ENSRNOG00000000021  -2.02    0.211  0.0373
## 10 ENSRNOG00000000024   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
```

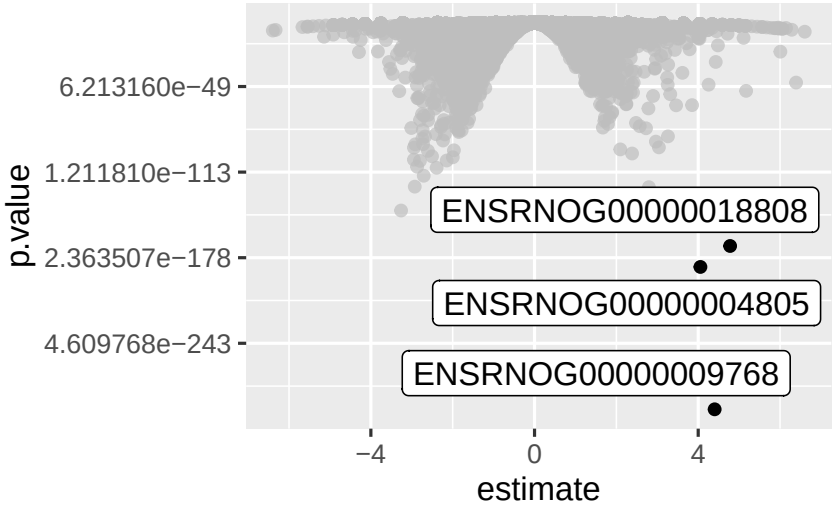
Other exploratory analysis will also be straightforward with the data using tidyverse tools, now that they are in a "tidy" format.

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`.

3.6.3 The `ggbio` package

3.6.4 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 inputing 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 cominatoric 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 primatives, 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 faceting 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 faceting by sequence (chromosome), because when the x axis is the chromosomal coordinate, it is not sensible to plto 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 grammalr, 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)

3.6.5 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.

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]

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 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 created 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.

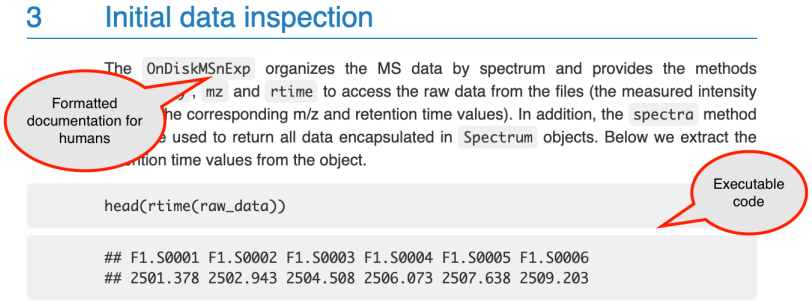


Figure 3.4: An example of a knitted document. This shows a section of the online vignette for the 'xcms' package from Bioconductor. The two types of content are highlighted: formatted text for humans to read, and executable computer code.

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.

"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]

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 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 plan 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 `rtime` 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-rtime, message = FALSE }
head(rtime(raw_data))
```
```

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 'xcms' package from Bioconductor. The full plain text file used for the vignette can be viewed on GitHub [here](https://github.com/sneumann/xcms/blob/master/vignettes/xc

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 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 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 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:

The combination that indicates the start of executable code will vary de-

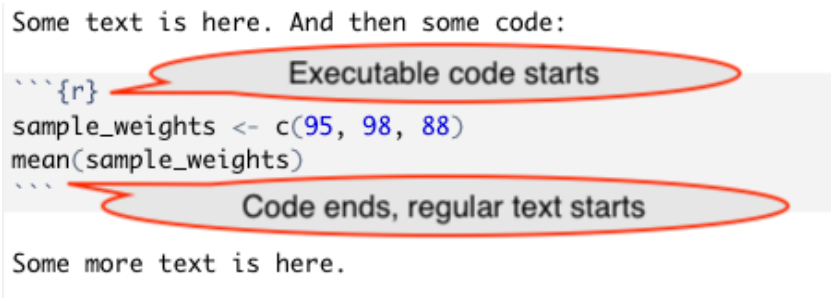


Figure 3.6: An example of how special combinations of characters are used to demarcate code in an RMarkdown file. The color formatting here is applied automatically by RStudio; all the text in this example is written in plain text.

pending 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

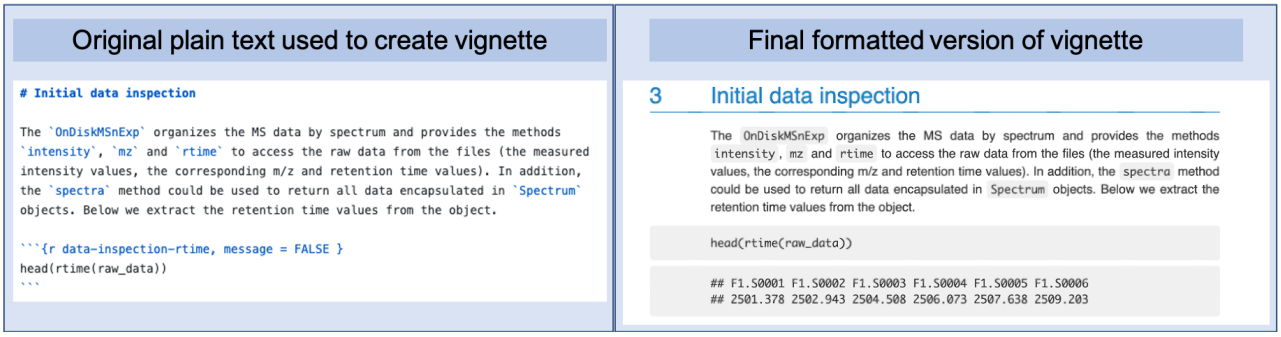


Figure 3.7: The original plain text for a knitted document and the final output, side by side. These examples are from the 'xcms' package vignette, a package available on Bioconductor. The left part of the figure shows the plain text that was written to create the output, which is shown in the left part 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 charaters, using the Markdown language syntax.

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 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.

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

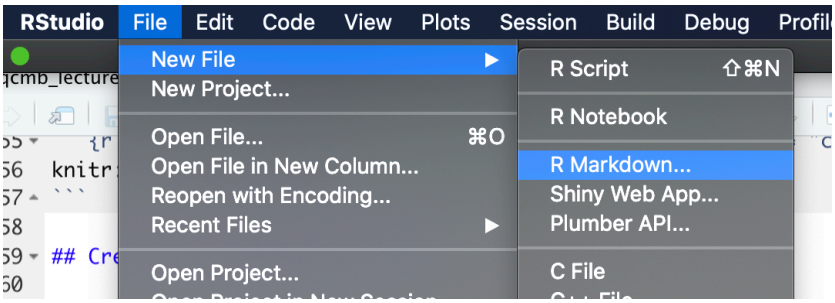


Figure 3.8: RStudio pull-down menus to help you navigate to open a new Rmarkdown file.

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.

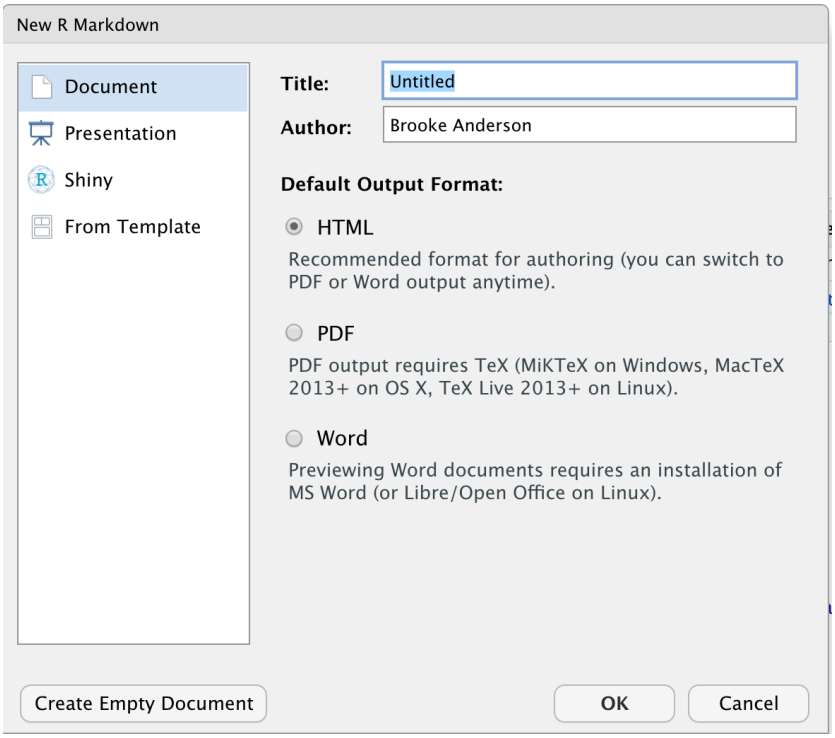


Figure 3.9: Options available when you create a new Rmarkdown file in RStudio. You can specify information that will go into the document's preamble, including the title and authors and the format that the document will be output to (HTML, Word, or PDF).

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.

```
---
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.

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.

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).

You will also notice, after you first render the document, that your working 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

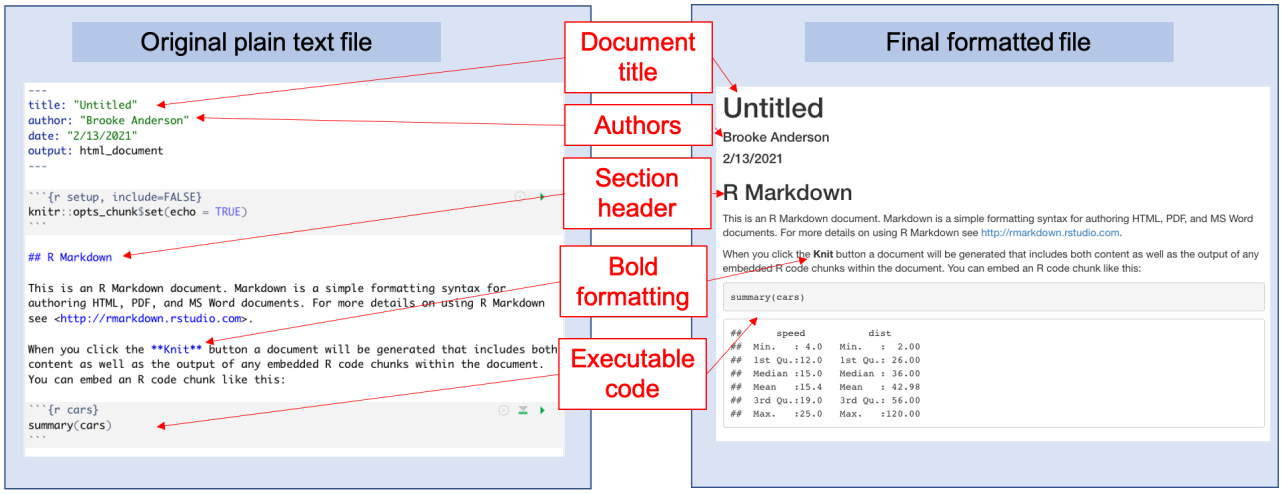


Figure 3.11: Example of the template Rmarkdown document that you will see when you create a new Rmarkdown file in RStudio. You can explore this template and try rendering (knitting) it. Once you are familiar with how this example works, you can edit the text and code to adapt it for your own document.

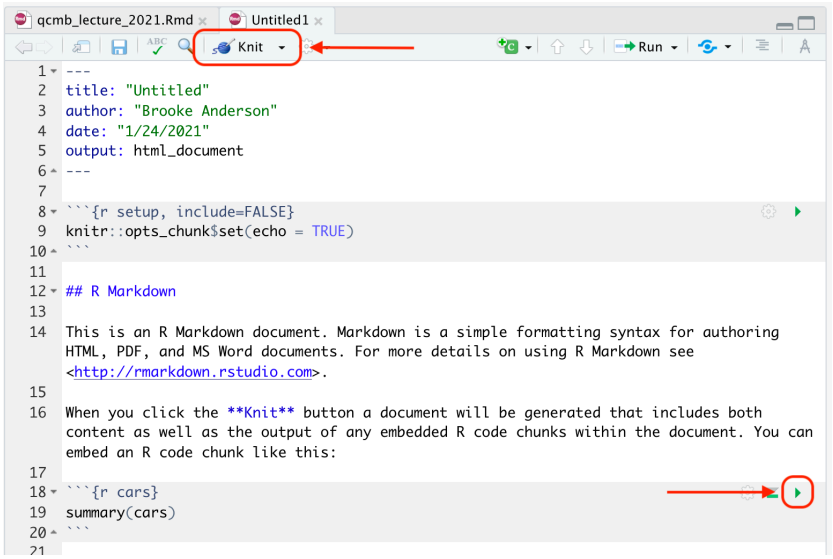


Figure 3.12: Example of the template Rmarkdown document, highlighting buttons in RStudio that you can use to facilitate working with the document. The 'knit' button, highlighted at the top of the figure, will render the entire document. The green arrow, highlighted lower in the figure within a code chunk, can be used to run the code in that specific code chunk.

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 preprocessing protocol, you are aiming to show yourself and others how the preprocessing 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{\'o}n-Henao, Andr{\'e}s 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.

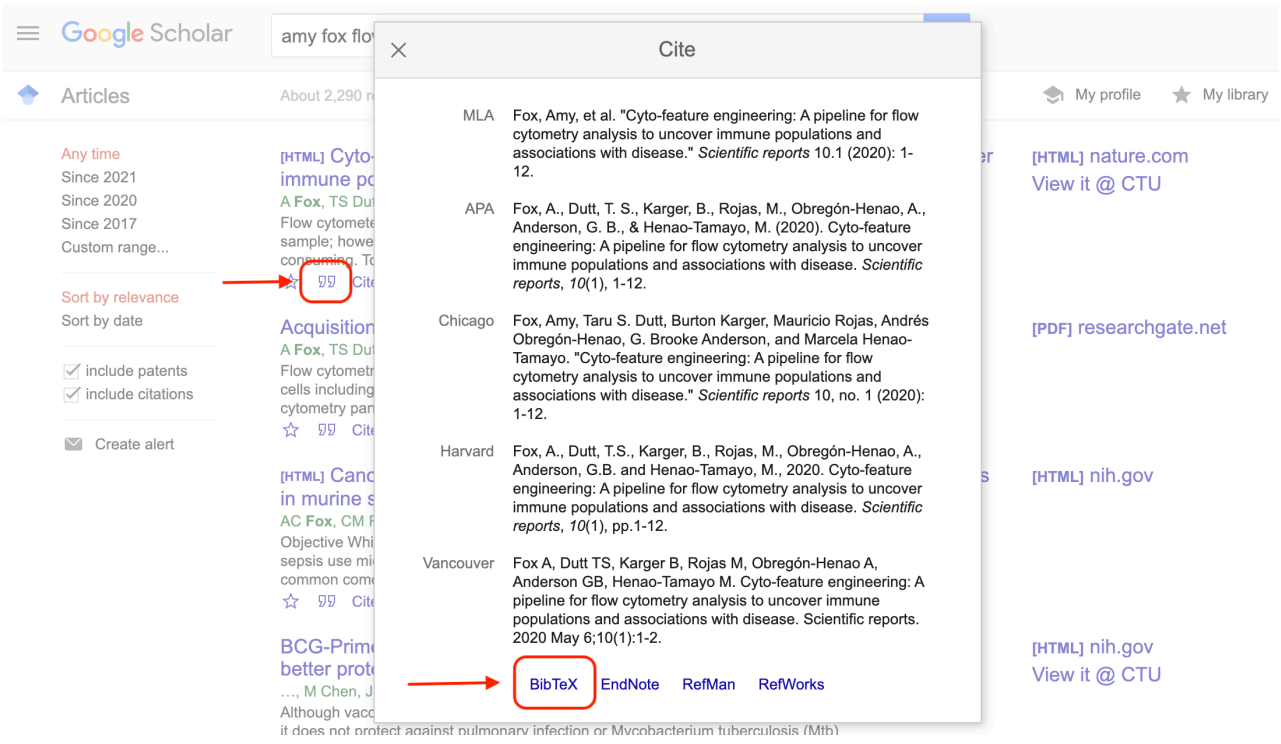


Figure 3.13: Example of using Google Scholar to get bibliographical information for a BibTeX file. When you look up an article on Google Scholar, there is an option (the quotation mark icon under the article listing) to open a pop-up window with bibliographical information. At the bottom of this pop-up 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 you 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.

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, repectively. 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.

Figure 3.14: Example of a website created using blogdown, leveraging the Rmarkdown framework.

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.

Ten simple rules for finding and selecting R packages

Caroline J. Wendt [1 , 2] , G. Brooke Anderson [3 *]

1 Department of Statistics, Colorado State University, Fort Collins, Colorado, United States of America
2 Department of Mathematics, Colorado State University, Fort Collins, Colorado, United States of America
3 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 templat]{Example of a manuscript written in Rmarkdown using a templat. 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 `rticles` package. The full article, including the Rmarkdown input and final pdf, are available on GitHub at `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`) and one if it will be in 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 dillutions, 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 bactieral 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 complete 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 preprocessing. 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 preprocess, 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 preprocess, 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 preprocessing 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 awkard 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).

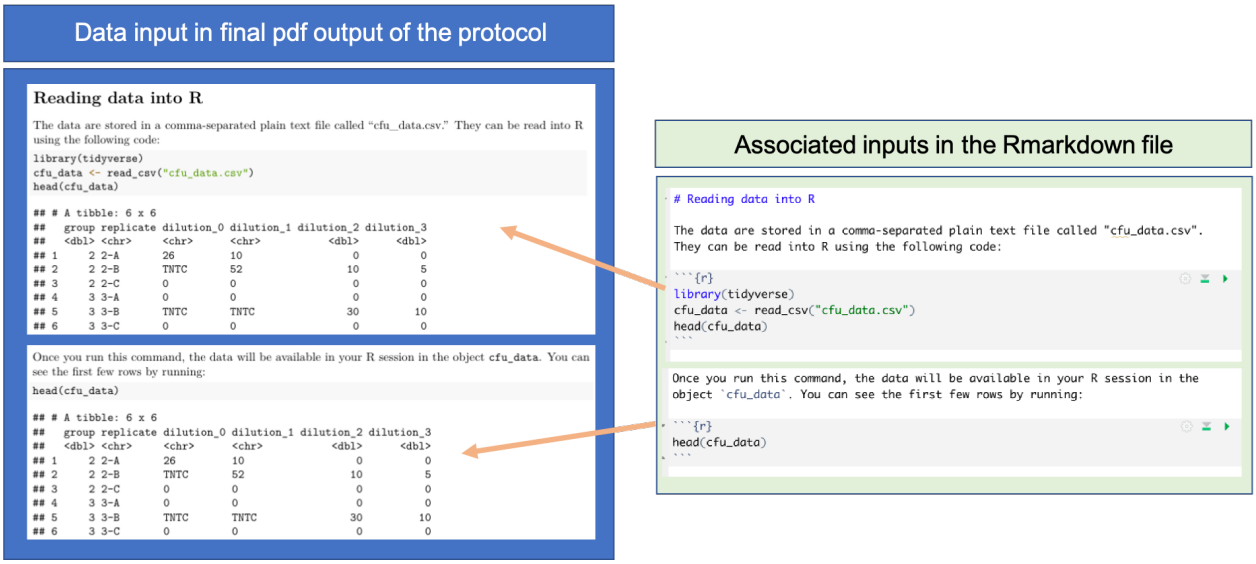


Figure 3.15: Providing details on input data in the pre-processing protocol. Once you have an example data file for the type of data that will be input for the protocol, you can add a section that provides the code to read the data into R. You can also add code that will show the first few rows of the 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.

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:

```
## # 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 left 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.

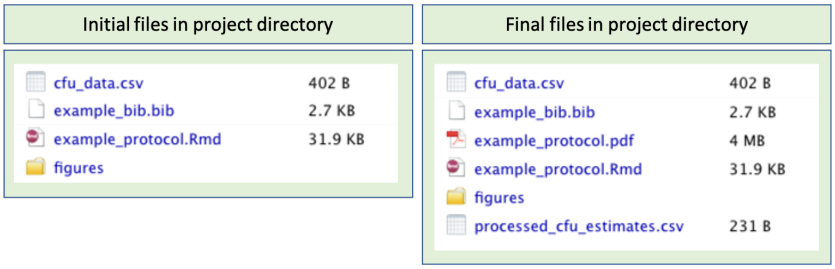


Figure 3.16: Example of files in the project directory for a data pre-processing protocol. On the left are the files initially included in the project directory for the example protocol for this module. These include a file with the input data (cfu_data.csv), a BibTeX file with bibliographical information 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.

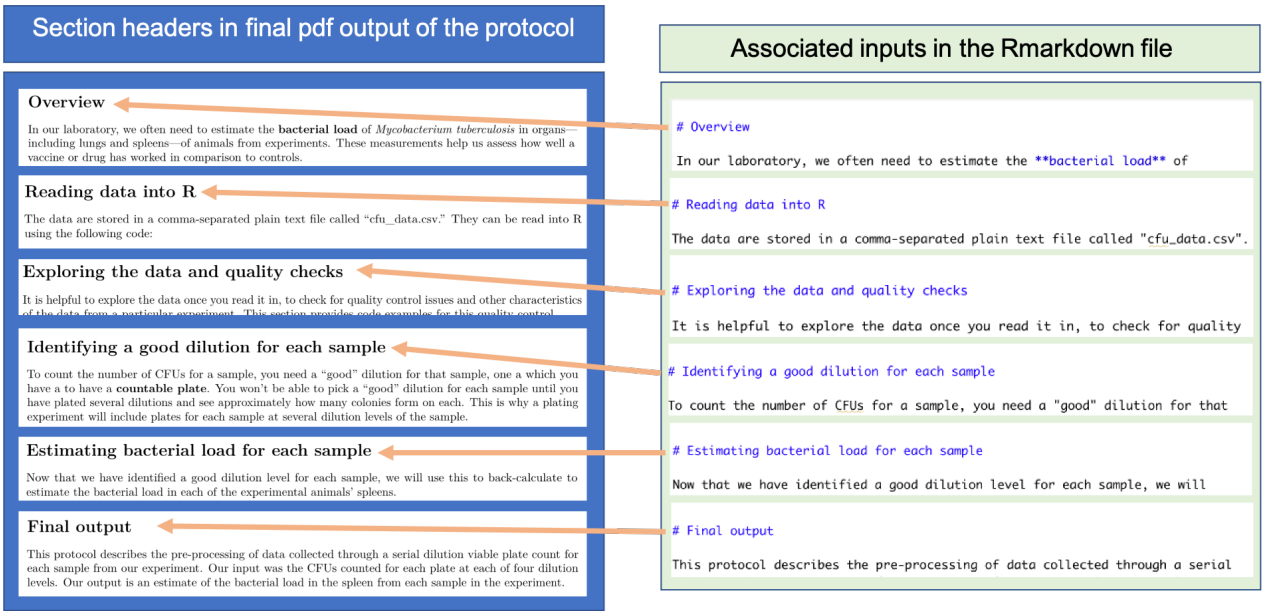


Figure 3.17: Dividing an RMarkdown data pre-processing protocol into sections. This shows an example of creating section headers in a data pre-processing protocol created with RMarkdown, showing section headers in the example pre-procotcol for this module.

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 in 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

finer-level details about what the code is doing.

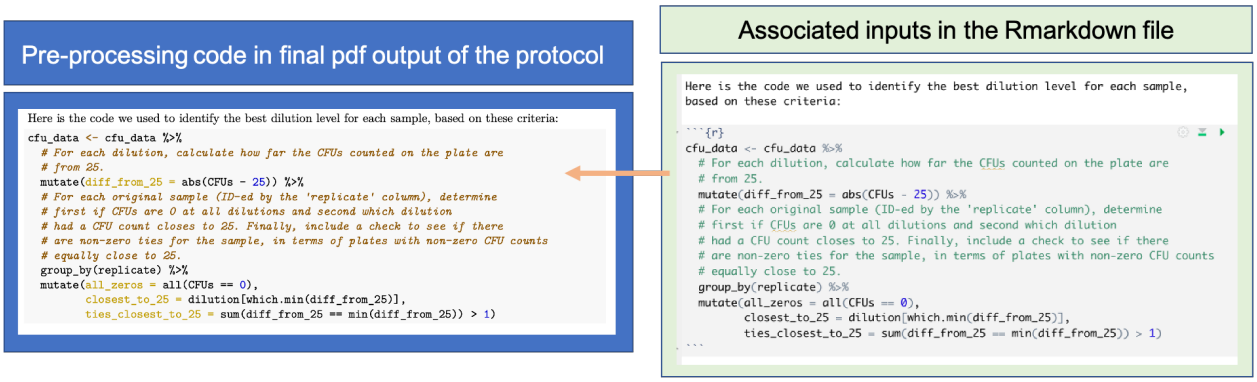


Figure 3.18: Example of including code in a data pre-processing protocol created with RMarkdown. This figure shows how code can be included in the RMarkdown file for a pre-processing protocol (right), and the corresponding output in the final pdf of the protocol (left), for the code to identify a 'good' dilution for counting CFUs for each sample. Code comments are included to provide finer-level details on the code.

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

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.

"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]

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.

"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]

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

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.

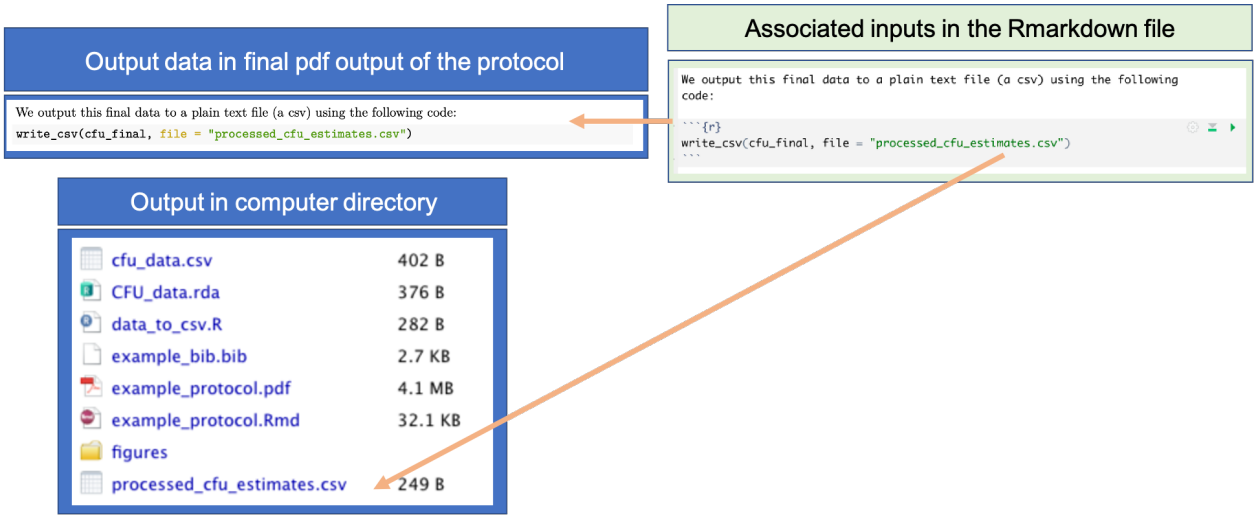


Figure 3.19: Example of using code in pre-processing protocol to output the final, pre-processed data that will be used in further analysis for the research project. This example comes from the example protocol for this module, showing both the executable code included in the RMarkdown file for the protocol (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.

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 your 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

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.

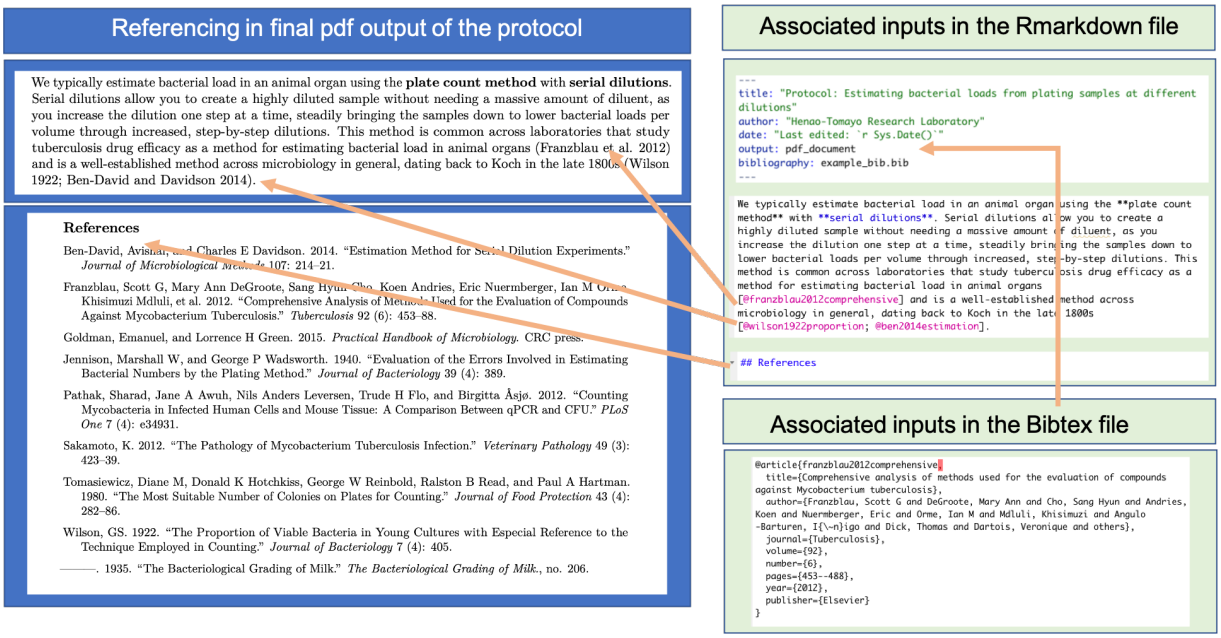


Figure 3.20: Including references in a data pre-processing protocol created with RMarkdown. RMarkdown has a built-in referencing system that you can use, based on the BibTeX system for LaTeX. This figure shows examples from the example protocol for this module of the elements used for referencing. You 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 you 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.

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

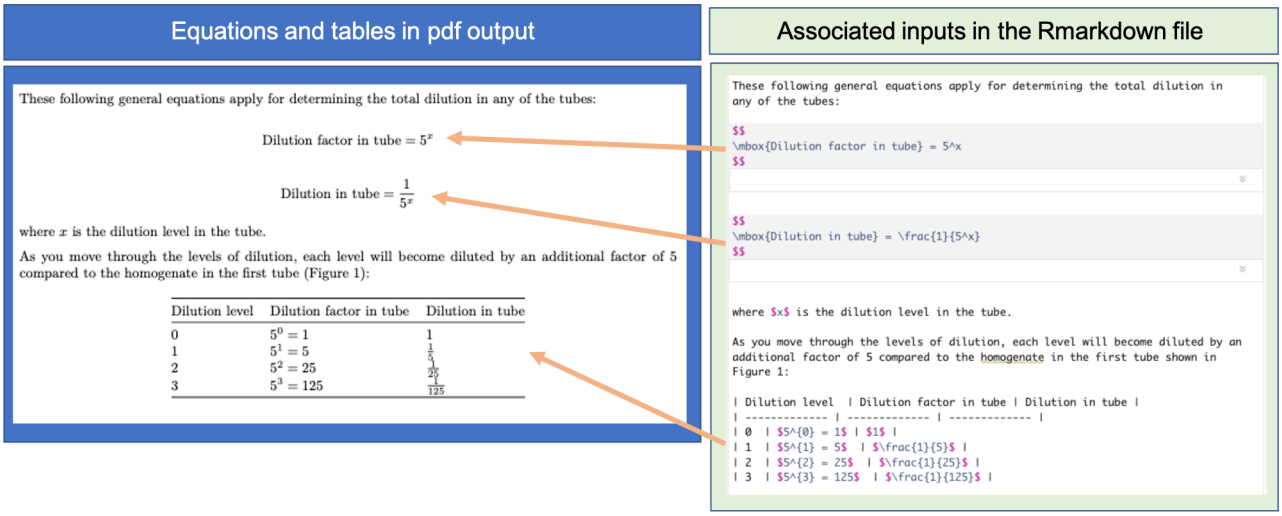


Figure 3.21: Example of including tables and equations in an RMarkdown data pre-processing protocol.

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.

3.10 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?

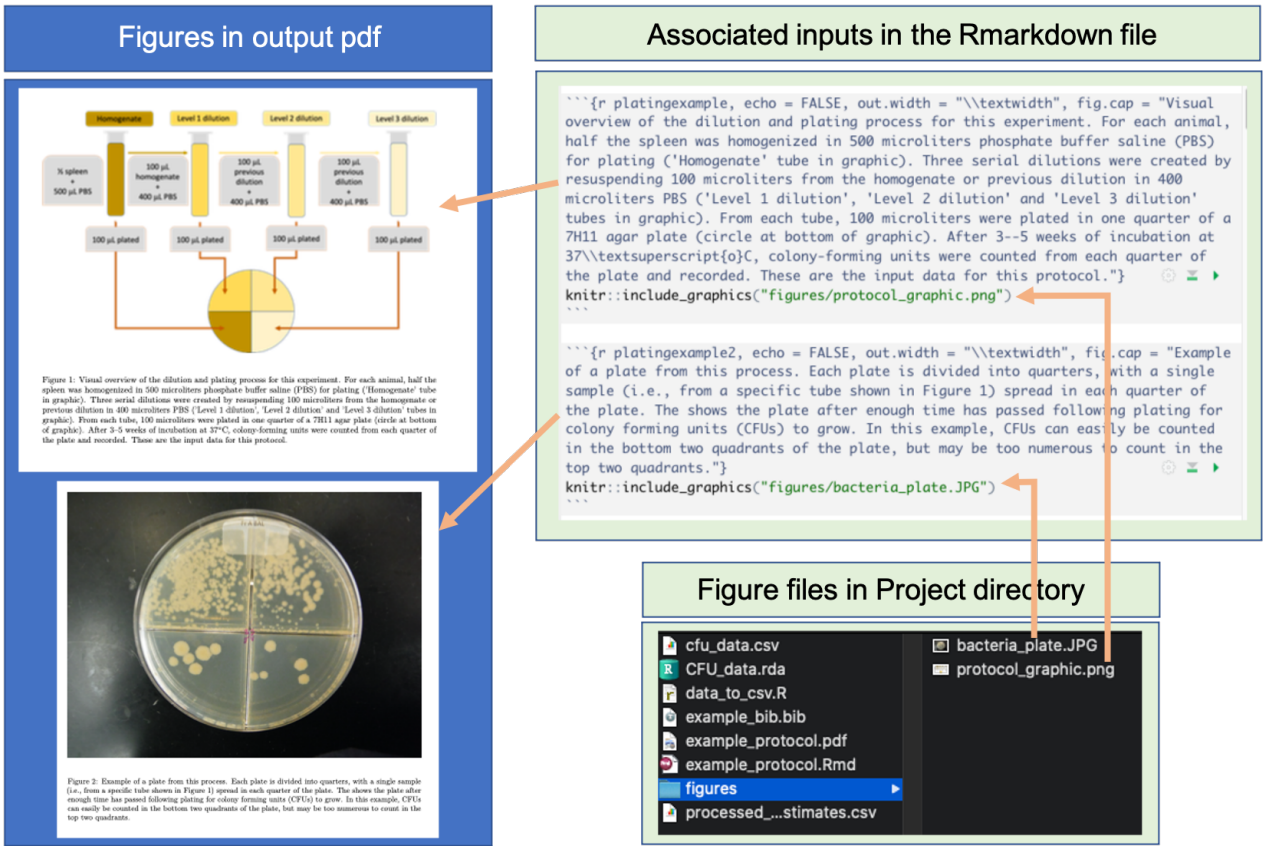


Figure 3.22: Example of including figures from image files in an RMarkdown data pre-processing protocol.

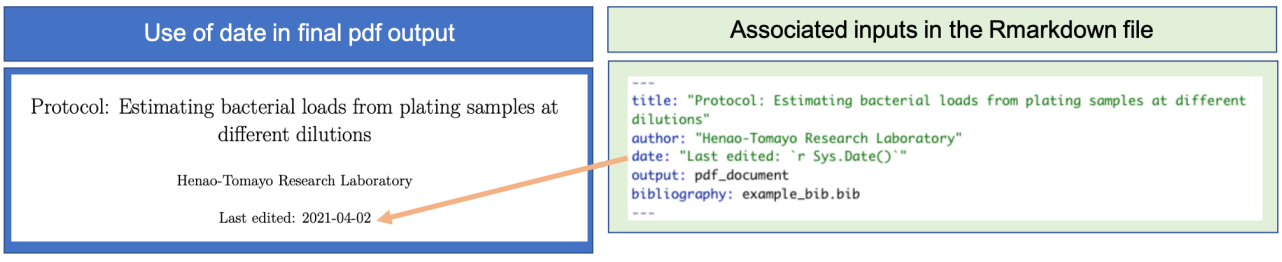


Figure 3.23: Example of using more advanced RMarkdown functionality within a data pre-processing protocol. In this example, R code is incorporated into the YAML of the document to include the date that the document was last rendered, marking this on the pdf output as the *Last edited* date of the protocol.

4

References

5
Bibliography

(2021). RStudio Project Templates. `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.

Arnold, T. (2017). A Tidy Data Model for Natural Language Processing using cleanNLP. *The R Journal*, 9(2):248–267.

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.

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.

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.

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.

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.

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.

Gatto, L. (2013). Msnbase 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.

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.

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.

Grune, D. (1986). Concurrent verions system, a method for independent cooperation. *Technical Report, Vrije Universiteit*.

Hammer, P., Banck, M. S., Amberg, R., Wang, C., Petznick, G., Luo, S., Khrebtukova, 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.

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.

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.".

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.

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.

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.

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.

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.

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.

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.

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 Interational 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.

Pawlik, 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. (2018). Git: The reproducibility tool scientists love to hate.

Perkel, J. M. (2011). Coding your way out of a problem.

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.

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.

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.

Sedgwick, P. (2014). Unit of observation and unit of analysis. *BMJ*, 348:g3840.

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.

Stander, 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.

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.

Tippmann, S. (2014). My digital toolbox: Nuclear engineer katy huff on version-control systems. *Nature News*.

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.

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.

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.