Introduction to R for Biologists

Alana Schick

2020-03-13

Contents

Chapter 1

Welcome

Workshop material will be added here as we go along. For each day of the workshop, the basic material will be uploaded and more details will be added based on your questions and problems we encounter. So please ask as many questions are you can to help us in making this workshop better!!

1.1 Workshop Schedule

*Subject to change, depending how things go.

Day 1

Time	Topic
09:00	Welcome and Presentation
09:30	1. Introducing R and RStudio
10:00	2. Installing R packages
10:20	Break
10:35	3. R Basics
12:00	Lunch break
13:00	4. Starting with Data
14:30	Break
14:45	6. Manipulating data in the Tidyverse
16:00	Finish

Day 2

Time	Topic
09:00	6. Tidyverse - cont
09:30	7. Data visualization with ggplot2
10:30	Break
10:45	8. Project - task 1
11:00	9. Project - task 2
12:00	Lunch break
13:00	8. Project - task 3
14:30	Break
14:45	8. Project - task 4
15:30	Review project and open discussion
15:55	Survey!
16:00	Finish

1.2 Important links

- Datacamp: link
- Survey: link

Chapter 2

Pre-workshop

2.1 Installation

R and RStudio are separate downloads and installations. R is the underlying statistical computing environment and RStudio is a graphical integrated development environment (IDE) that makes using R much easier and more interactive. You will need to install R before you install RStudio.

If you already have R and RStudio installed, open RStudio and click on "Help" > "Check for updates". If a new version is available, quit RStudio and download the latest version of RStudio. To check which version of R you are using, start RStudio and check the top of the console to see which version of R you are running. Alternatively, you can type `sessionInfo()`, which will also display the version of R you are running. Go on the CRAN website and check whether a more recent version is available. If so, please download and install it.

If you don't have R and RStudio installed, please follow the instructions for your operating system below.

2.1.1 Windows

- Download R from the CRAN website.
- Run the `.exe` file that was just downloaded
- Go to the RStudio download page
- Under *Installers* select RStudio for Windows
- Double click the file to install it
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

2.1.2 MacOS

- Download R from the CRAN website.
- Select the `.pkg` file for the latest R version
- Double click on the downloaded file to install R
- It is also a good idea to install XQuartz (needed by some packages)
- Go to the RStudio download page
- Under *Installers* select RStudio for MacOS
- Double click the file to install RStudio
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

2.1.3 Linux

- Follow the instructions for your distribution from CRAN, they provide information to get the most recent version of R for common distributions. For most distributions, you could use your package manager (e.g., for Debian/Ubuntu run `sudo apt-get install r-base`, and for Fedora `sudo yum install R`), but we don't recommend this approach as the versions provided by this are usually out of date. In any case, make sure you have at least R 3.3.1.
- Go to the RStudio download page
- Under *Installers* select the version that matches your distribution, and install it with your preferred method (e.g., with Debian/Ubuntu `sudo dpkg -i rstudio-x.yy.zzz-amd64.deb` at the terminal).
- Once it's installed, open RStudio to make sure it works and you don't get any error messages.

2.2 Datacamp Courses

To help familiarize yourself with R, you have been assigned two Datacamp courses to be completed prior to the workshop:

- Introduction to R
- Introduction to the Tidyverse

Please do your best to have these courses completed by the start of the workshop.

Chapter 3

Introduction

3.1 R and the RStudio IDE

Short presentation introducing R and the RStudio IDE. Attempting to convince you how amazing R is. Slides will be uploaded here.

3.2 Why learn R

R does not involve a lot of pointing and clicking - that's a good thing!

With R, the results of your analysis do not depend on remembering a succession of pointing and clicking, but instead on a series of written commands. That means that if (when!) you want to redo your analysis because you collected more data, or you want to run the same analyses on a different dataset, you don't have to remember which button your clicked to obtain your results, you just have to run the script again.

Working with scripts makes the steps you used in your analysis clear, and the code you write can be inspected by someone else for their own work, or to give you feedback.

Working with scripts forces you to have a deeper understanding of what you are doing, and facilitates your learning and comprehension of the methods you use.

R code is great for reproducibility

Reproducibility is when someone else (including your future self) can obtain the same results from the same dataset when using the same analysis.

Short-term goals:

- Are the tables and figures reproducible from the code and data?
- Does the code actually do what you think it does?
- In addition to what was done, is it clear why is was done? (e.g., how were parameter settings chosen?)

Long-term goals:

- Can the code be used for other data?
- Can you extend the code to do other things?

An increasing number of journals and funding agencies expect analyses to be reproducible, so knowing R will give you an edge with those requirements.

R is interdisciplinary and extensible

With 10,000+ packages that can be installed to extend its capabilities, R provides a framework that allows you to combine statistical approaches from many scientific disciplines to best suit the analytical framework you need to analyze your data. For instance, R has packages for image analysis, GIS, time series, population genetics, bioinformatics and a lot more.

R works on data of all shapes and sizes

The skills you learn with R scale easily with the size of your dataset. Whether your dataset has hundreds or millions of lines, it won't make much difference to you.

R is designed for data analysis. It comes with special data structures and data types that make handling of missing data and statistical factors convenient.

R can connect to spreadsheets, databases, and many other data formats, on your computer or on the web.

R produces high-quality graphics

The plotting functionalities in R are endless, and allow you to adjust any aspect of your graph to convey most effectively the message from your data.

R has a large and welcoming community

Thousands of people use R daily. Many of them are willing to help you through mailing lists and websites such as Stack Overflow, or on the RStudio community.

Interacting with R

There are two main ways of interacting with R: using the console or by using script files (plain text files that contain your code).

The console window (in RStudio, the bottom left panel) is the place where R is waiting for you to tell it what to do, and where it will show the results of a command. You can type commands directly into the console, but they will be forgotten when you close the session. It is better to enter the commands in the script editor, and save the script. This way, you have a complete record of what you did, you can easily show others how you did it and you can do it again later on if needed. You can copy-paste into the R console, but the Rstudio script editor allows you to 'send' the current line or the currently selected text to the R console using the `Ctrl-Enter` shortcut.

If R is ready to accept commands, the R console shows a `>` prompt. If it receives a command (by typing, copy-pasting or sent from the script editor using `Ctrl-Enter`), R will try to execute it, and when ready, show the results andcome back with a new `>` prompt to wait for new commands.

If R is still waiting for you to enter more data because it isn't complete yet, the console will show a `+` prompt. It means that you haven't finished entering a complete command. This is because you have not 'closed' a parenthesis or quotation. If you're in Rstudio and this happens, click inside the console window and press `Esc`; this should help you out of trouble.

3.3 Creating an RStudio project

One of the benefits of RStudio is something called an RStudio Project. An RStudio project allows you to more easily:

- Save data, files, variables, packages, etc. related to a specific analysis project
- Restart work where you left off
- Collaborate, especially if you are using version control such as git.

EXERCISE

To create a project:

- Click the `File` menu, click on `New project`.
- Choose `New directory`, then `New project`
- For “directory name” enter `rworkshop`. For “Create project as subdirectory of”, you may leave the default, which is your home directory “~”.
- Click on `Create project`
- Under the `Files` tab on the right of the screen, you should see an RStudio project file, rworkshop.Rproj. All RStudio projects end with the “.Rproj” file extension.

3.4 Creating your first R script

Now that we are ready to start exploring R, we will want to keep a record of the commands we are using. To do this we can create an R script:

EXERCISE

To create an R script:

- Click the `File` menu, click on `New File` and then `R Script`.
- Save your script by clicking the save icon that is in the bar above the first line in the script editor, or click `File` then `Save`.
- In the “Save File” window that opens, name your file`rworkshop_basicscript`.
- The new script rworkshop_basicscript.R should appear under “files” in the output pane. All R scripts end with the “.R” file extension.

Enter the following code in the script and hit run:

```r
# Define vectors
d <- c(1,2,3,4,5,6,7)
e <- 8:14
f <- "Myplot"

# Plot example
plot(d,e,main=f)
```

Organizing your working directory

Using a consistent folder structure across your projects will help keep things organized, and will also make it easy to find/file things in the future. This can be especially helpful when you have multiple projects. In general, you should separate the original data (raw data) from intermediate datasets that you may create for the need of a particular analysis. For instance, you may want to create a `data/` directory within your working directory that stores the raw data, and have a `data_output/` directory for intermediate datasets and `figure_output/` directory for the plots you will generate.

The working directory is an important concept to understand. It is the place from where R will be looking for and saving the files. When you write code for your project, it should refer to files in relation to the root of your working directory and only need files within this structure.

Using RStudio projects makes this easy and ensures that your working directory is set properly. If you need to check it, you can use:

```r
getwd()
```

If for some reason your working directory is not what it should be, you can change it in the RStudio interface by navigating in the file browser where your working directory should be, and clicking on the blue gear icon "More", and select "Set As Working Directory". Alternatively you can use:

```r
setwd("/path/to/working/directory")
```

to reset your working directory.

3.5 How to learn more?

The material we cover during this workshop will give you an initial taste of how you can use R to analyze data for your own research. However, you will need to learn more to do advanced operations. The best way to become proficient and efficient at R, as with any other tool, is to use it to address your actual research questions. As a beginner, it can feel daunting to have to write a script from scratch, and given that many people make their code available online, modifying existing code to suit your purpose might make it easier for you to get started.

The following part of this section is from "Modern Dive Section 2.2.3"

Learning to code/program is very much like learning a foreign language, it can be very daunting and frustrating at first. Such frustrations are very common and it is very normal to feel discouraged as you learn. However just as with learning a foreign language, if you put in the effort and are not afraid to make mistakes, anybody can learn.

Here are a few useful tips to keep in mind as you learn to program:

- Remember that computers are not actually that smart: You may think your computer or smartphone are "smart," but really people spent a lot of time and energy designing them to appear "smart." Rather you have to tell a computer everything it needs to do. Furthermore the instructions you give your computer can't have any mistakes in them, nor can they be ambiguous in any way.

- Take the "copy, paste, and tweak" approach: Especially when learning your first programming language, it is often much easier to taking existing code that you know works and modify it to suit your ends, rather than trying to write new code from scratch. We call this the copy, paste, and tweak approach. So early on, we suggest not trying to write code from memory, but rather take existing examples we have provided you, then copy, paste, and tweak them to suit your goals. Don't be afraid to play around!
- The best way to learn to code is by doing: Rather than learning to code for its own sake, we feel that learning to code goes much smoother when you have a goal in mind or when you are working on a particular project, like analyzing data that you are interested in.
- Practice is key: Just as the only method to improving your foreign language skills is through practice, practice, and practice; so also the only method to improving your coding is through practice, practice, and practice.

Seeking help

One of the fastest ways to get help is to use the RStudio help interface. This panel by default can be found at the lower right hand panel of RStudio. As seen in the screenshot, by typing the word "Mean", RStudio tries to also give a number of suggestions that you might be interested in. The description is then shown in the display window.

If you need help with a specific function, let's say `barplot()`, you can type:

```
?barplot
```

If you just need to remind yourself of the names of the arguments, you can use:

```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

If you are looking for a function to do a particular task, you can use the `help.search()` function, which is called by the double question mark ??. However, this only looks through the installed packages for help pages with a match to your search request

```
??kruskal
```

If you can't find what you are looking for, you can use the rdocumentation.org website that searches through the help files across all packages available.

I am stuck... I get an error message that I don't understand

Start by googling the error message. However, this doesn't always work very well because often, package developers rely on the error catching provided by R. You end up with general error messages that might not be very helpful to diagnose a problem (e.g. "subscript out of bounds"). If the message is very generic, you might also include the name of the function or package you're using in your query.

However, you should check Stack Overflow. Search using the `[r]` tag. Most questions have already been answered, but the challenge is to use the right words in the search to find the answers:

Asking for help

The key to receiving help from someone is for them to rapidly grasp your problem. You should make it as easy as possible to pinpoint where the issue might be.

Try to use the correct words to describe your problem. For instance, a package is not the same thing as a library. Most people will understand what you meant, but others have really strong feelings about the difference in meaning. The key point is that it can make things confusing for people trying to help you. Be as precise as possible when describing your problem.

If possible, try to reduce what doesn't work to a simple *reproducible example*. If you can reproduce the problem using a very small data frame instead of your 50,000 rows and 10,000 columns one, provide the small one with the description of your problem. When appropriate, try to generalize what you are doing so even people who are not in your field can understand the question. For instance instead of using a subset of your real dataset, create a small (3 columns, 5 rows) generic one. For more information on how to write a reproducible example see this article by Hadley Wickham.

Where to ask for help?

- Stack Overflow: if your question hasn't been answered before and is well crafted, chances are you will get an answer in less than 5 min. Remember to follow their guidelines on how to ask a good question. The R-help mailing list: it is read by a lot of people (including most of the R core team), a lot of people post to it, but the tone can be pretty dry, and it is not always very welcoming to new users. If your question is valid, you are likely to get an answer very fast but don't expect that it will come with smiley faces. Also, here more than anywhere else, be sure to use correct vocabulary (otherwise you might get an answer pointing to the misuse of

your words rather than answering your question). You will also have more success if your question is about a base function rather than a specific package.
- The "RStudio Community"
- If your question is about a specific package, see if there is a mailing list for it. Usually it's included in the DESCRIPTION file of the package that can be accessed using packageDescription("name-of-package"). You may also want to try to email the author of the package directly, or open an issue on the code repository (e.g., GitHub).

How to ask for R help useful guidelines

- "How to ask for R help" has useful guidelines
- This blog post by Jon Skeet has quite comprehensive advice on how to ask programming questions.
- The "reprex" package is very helpful to create reproducible examples when asking for help.
    - The rOpensci blog on How to ask questions so they get answered and "video recording" includes a presentation of the reprex package and of its philosophy.

Chapter 4

Installing R Packages

4.1 What are R packages?

R packages extend the functionality of R by providing additional functions, data, and documentation for specific tasks. They are written by a world-wide community of R users and can be downloaded for free from the internet. A good analogy for R packages is they are like apps you can download onto a mobile phone.

So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it doesn't have everything. R packages are the apps you download onto your phone from Apple's App Store or Android's Google Play.

Say you have purchased a new phone and you would like to share a recent photo you have taken on Instagram. You need to:

1. Install the app: Since your phone is new and does not include the Instagram app, you need to download the app from either the App Store or Google Play. You do this once and you're set. You might do this again in the future any time there is an update to the app.

2. Open the app: After you've installed Instagram, you need to open the app.

Once Instagram is open on your phone, you can then proceed to share your photo with your friends. The process is very similar for using an R package. You need to:

1. *Install the package*: Most packages are not installed by default when you install R and RStudio. Thus if you want to use a package for the first time, you need to install it first. Once you've installed a package, you

likely won't install it again unless you want to update it to a newer version.

2. *Load the package*: Packages are not loaded by default when you start RStudio on your computer; you need to load each package you want to use every time you start RStudio.

4.2 Installing R packages from different sources

There are multiple sources of R packages.

1. CRAN

CRAN is the "Comprehensive R Archive Network" - the main repository (or collection) of R packages.

There are three ways to install an R package from CRAN. For example, to install the `tidyverse` metapackage (a package of packages):

1. **Easy way**: In the Files pane of RStudio:
   a. Click on the `Packages` tab
   b. Click on `Install`
   c. Type the name of the package under "Packages (separate multiple with space or comma):" In this case, type `tidyverse`
   d. Click `Install`
2. **Easy way**: In the menu bar of RStudio:
   a. Click on `Tools` and `Install Packages...`
   b. Type the name of the package under "Packages (separate multiple with space or comma):" In this case, type `tidyverse`
   c. Click `Install`
3. **Slightly harder way**: An alternative way to install a package is by typing `install.packages("tidyverse")` in the Console pane of RStudio and hitting enter. Note you must include the quotation marks.

If you want to update an already installed package to a newer version, you need to re-install it by repeating the above steps.

2. Bioconductor

Bioconductor is a repository for R packages that concern computational biology or bioinformatics. To install the `knitr` package from Bioconductor, type the following in the Console of RStudio:

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("knitr")
```

3. Github

Some packages are not hosted on CRAN or Bioconductor, but are instead stored in Github repositories. Sometimes this is because they are very new and have not yet been contributed. To install the `dplyr` package from Github, type the following in the Console of RStudio:

```
install.packages("devtools")
library(devtools)
install_github("hadley/dplyr")
```

EXERCISE

Practice what you have just learned and install the R package `ggplot2`.

4.3 Loading a package

Recall that after you've installed a package, you need to "load" it, in other words open it. We do this by using the `library()` command. For example, to load the `tidyverse` package, run the following code in the Console pane. What do we mean by "run the following code"? Either type or copy & paste the following code into the Console pane and then hit the enter key.

```
library(tidyverse)
```

If after running the above code, a blinking cursor returns next to the `>` "prompt" sign, it means you were successful and the `tidyverse` package is now loaded and ready to use. If however, you get a red "error message" that reads...

```
Error in library(tidyverse) : there is no package called 'tidyverse'
```

it means that you didn't successfully install it. In that case, go back to the previous subsection and trying install it again.

EXERCISE

Confirm that you successfully installed the packages `ggplot2`, `dplyr` and `knitr` by loading them.

Package use

One extremely common mistake new R users make when wanting to use particular packages is they forget to "load" them first by using the `library()` command we just saw. Remember: you have to load each package you want to use every time you start RStudio. If you don't first "load" a package, but attempt to use one of its features, you'll see an error message similar to:

```
Error: could not find function
```

R is telling you that you are trying to use a function in a package that has not yet been "loaded". Almost all new users forget do this when starting out, and it is a little annoying to get used to. However, you'll remember with practice.

Chapter 5

R Basics

5.1 Creating objects in R

You can get output from R simply by typing math in the console:

```
3 + 5
```

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

```
12 / 7
```

```
## [1] 1.714286
```

However, to do useful and interesting things, we need to assign values to objects. To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the *assignment* operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** x. For historical reasons, you can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is best practice to always use `<-` for assignments.

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long.

They cannot start with a number (`2x` is not valid, but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`).

There are some names that cannot be used because they are the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `mean`, `data`, `weights`). If in doubt, check the help to see if the name is already in use. It's also best to avoid dots (`.`) within an object name as in `my.dataset`. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them. It is also recommended to use nouns for object names, and verbs for function names.

It's important to be consistent in the styling of your code (where you put spaces, how you name objects, etc.). Using a consistent coding style makes your code clearer to read for your future self and your collaborators. In R, three popular style guides are Google's, Jean Fan's and the tidyverse's. This may seem overwhelming at first. My suggestion is to focus on what makes sense to you, and that you are using naming conventions that will help you to remember what is stored in each object. Try and be consistent with spacing and capitalization (or not) — your future self will thank you.

5.2 Objects vs. variables

What are known as `objects` in `R` are known as `variables` in many other programming languages. In this lesson, the two words are used synonymously. For more information see https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name.

```
weight_kg <- 55 #  doesn't print anything
```

```
(weight_kg <- 55) #  will print the value of `weight_kg`
```

```
## [1] 55
```

```
weight_kg         #  and so does typing the name of the object
```

```
## [1] 55
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

```
## [1] 121
```

We can also change an object's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
## [1] 126.5
```

This means that assigning a value to one object does not change the values of other objects For example, let's store the animal's weight in pounds in a new object, `weight_lb`:

```
weight_lb <- 2.2 * weight_kg
```

and then change weight_kg to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

5.3 Comments

The comment character in `R` is `#`, anything to the right of a `#` in a script will be ignored by R. It is useful to leave notes, and explanations in your scripts. RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard `Ctrl + Shift + C`. If you only want to comment out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press `Ctrl + Shift + C`.

EXERCISE

What are the values stored in each object in the following (do this in your head)?

```
mass <- 50              # mass?
age  <- 122             # age?
mass <- mass * 2.0      # mass?
age  <- age - 22        # age?
mass_index <- mass/age  # mass_index?
```

5.4 Functions and their arguments

Functions are "canned scripts" that automate more complicated sets of commands including operations assignments, etc. Many functions are predefined, or can be made available by importing R packages.

A function usually takes one or more inputs called arguments. Functions often (but not always) return a value. A typical example would be the function `sqrt()`. The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number.

You can think of a function call generally as

```
do_this(to_that)
do_that(to_this, to_that, with_those)
```

Executing a function ('running it') is called calling the function. An couple of examples of function calls are:

```r
print("hello world")
```

```
## [1] "hello world"
```

```r
a <- 9
b <- sqrt(a)
b
```

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

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to the object `b`. This function is very simple, because it takes just one argument.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). Some functions take arguments which may either be specified by the user, or, if left out, take on a default value: these are called options. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round()`.

```r
round(3.14159)
```

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

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value 3. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the round function. We can use `args(round)` to find what arguments it takes, or look at the help for this function using `?round`.

```r
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
?round
```

We see that if we want a different number of digits, we can type `digits = 2` or however many we want.

```
round(3.14159, digits = 2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

```
## [1] 3.14
```

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to then specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

EXERCISE

Compute the golden ratio in a single line of code. One approximation of the golden ratio can be found by taking the sum of 1 and the square root of 5, and dividing by 2. Compute the golden ratio to 3 digits of precision using the `sqrt()` and `round()` functions. Hint: you can place one function inside of another.

5.5 Vectors

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is a colection of values that are all of the same type (numeric, character, etc.). We can assign a series of values to a vector using the `c()` function. For example we can create a vector of animal weights and assign it to a new object `weight_g`:

```
weight_g <- c(50, 60, 65, 82)
weight_g
```

```
## [1] 50 60 65 82
```

A vector can also contain characters:

```r
animals <- c("mouse", "rat", "dog")
animals
```

```
## [1] "mouse" "rat"   "dog"
```

The quotes around "mouse", "rat", etc. are essential here. Without the quotes R will assume objects have been created called `mouse`, `rat` and `dog`. As these objects don't exist in R's memory, there will be an error message.

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```r
length(weight_g)
```

```
## [1] 4
```

```r
length(animals)
```

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

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```r
class(weight_g)
```

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

```r
class(animals)
```

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

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```r
str(weight_g)
```

```
##  num [1:4] 50 60 65 82
```

```r
str(animals)
```

```
##  chr [1:3] "mouse" "rat" "dog"
```

You can use the `c()` function to add other elements to your vector:

```r
weight_g <- c(weight_g, 90) # add to the end of the vector
weight_g <- c(30, weight_g) # add to the beginning of the vector
weight_g
```

```
## [1] 30 50 60 65 82 90
```

In the first line, we take the original vector `weight_g`, add the value 90 to the end of it, and save the result back into `weight_g`. Then we add the value `30` to the beginning, again saving the result back into `weight_g`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

An atomic vector is the simplest R data type and is a linear vector of a single type. Above, we saw 2 of the 6 main atomic vector types that R uses: “character” and “numeric”. These are the basic building blocks that all R objects are built from. The other 4 atomic vector types are:

- “logical” for TRUE and FALSE (the boolean data type)
- “integer” for integer numbers (e.g., 2L, the L indicates to R that it’s an integer)
- “complex” to represent complex numbers with real and imaginary parts (e.g., 1 + 4i) and that’s all we’re going to say about them
- “raw” for bitstreams that we won’t discuss further

Vectors are one of the many data structures that R uses. Other important ones are matrices (`matrix`), factors (`factor`), data frames (`data.frame`), and lists (`list`). The set of packages in the `tidyverse` uses tibbles(`tibble`) which are sort of like souped up data frames. Data frames are analogous to rectangular spreadsheets: they are representations of datasets in R where the rows correspond observations and the columns correspond to variables that describe the observations.

EXERCISE

- We’ve seen that atomic vectors can be of type character, numeric, integer, and logical. But what happens if we try to mix these types in a single vector?

- What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

```r
num_char <- c(1, 2, 3, "a")
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
tricky <- c(1, 2, 3, "4")
```

- Why do you think it happens?

- How many values in `combined_logical` are “TRUE” (as a character) in the following example:

```r
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
combined_logical <- c(num_logical, char_logical)
```

You've probably noticed that objects of different types get converted into a single, shared type within a vector. In R, we call converting objects from one class into another class *coercion.* These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

5.6 Indexing vectors

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
# Get the 2nd value in the animals vector
animals[2]
```

```
## [1] "rat"
```

```
# Get the 1st through 3rd value in the animals vector
animals[1:3]
```

```
## [1] "mouse" "rat"   "dog"
```

```
# Get the 2nd and 4th values
animals[c(2,4)]
```

```
## [1] "rat" "cat"
```

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because *reasons.*

Adding values to vectors

We can add new items to a vector using the `c()` function.

```
animals <- c(animals, "horse", "crab", "monkey")
animals
```

```
## [1] "mouse"  "rat"    "dog"    "cat"
## [5] "horse"  "crab"   "monkey"
```

Replacing values in existing vectors

We can replace a value by using the index of the item to be replaced:

```r
# Change monkey to bird
animals[7] <- "bird"
animals
```

```
## [1] "mouse" "rat"   "dog"   "cat"   "horse"
## [6] "crab"  "bird"
```

5.7 Conditional subsetting

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not:

```r
weight_g <- c(21, 34, 39, 54, 55)
weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 21 39 54
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 50:

```r
weight_g > 50    # will return logicals with TRUE for the indices that meet the condition
```

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

```r
## so we can use this to select only the values above 50
weight_g[weight_g > 50]
```

```
## [1] 54 55
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```r
weight_g[weight_g < 30 | weight_g > 50]
```

```
## [1] 21 54 55
```

```r
weight_g[weight_g >= 30 & weight_g == 21]
```

```
## numeric(0)
```

Here, `<` stands for "less than", `>` for "greater than", `>=` for "greater than or equal to", and `==` for "equal to". The double equal sign `==` is a test for numerical equality between the left and right hand sides, and should not be confused with the single `=` sign, which performs variable assignment (similar to `<-`).

Operator	Description
`<`	less than
`<=`	less than or equal to

Operator	Description
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to
`!x`	not x
`a|b`	a or b
`a&b`	a and b

A common task is to search for certain strings in a vector. One could use the "or" operator | to test for equality to multiple values, but this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
```

```
## [1] "rat" "cat"
```

```
animals %in% c("rat", "cat", "dog", "duck", "goat")
```

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

```
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
```

```
## [1] "rat" "dog" "cat"
```

The `which()` function returns the indices of any item that evaluates as TRUE in our comparison:

```
which(animals == "cat")
```

```
## [1] 4
```

EXERCISE

- Can you figure out why "four" > "five" returns TRUE?

5.8 Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as `NA`.

When doing operations on numbers, most functions will return `NA` if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument na.rm = TRUE to calculate the result while ignoring the missing values.

```
heights <- c(2, 4, 4, NA, 6)
mean(heights)
```

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

```
max(heights)
```

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

```
mean(heights, na.rm = TRUE)
```

```
## [1] 4
```

```
max(heights, na.rm = TRUE)
```

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

If your data include missing values (and that's probably all of us!), you may want to become familiar with the functions `is.na()`, and `na.omit()`. See below for examples.

```
# Extract those elements which are not missing values.
heights[!is.na(heights)]
```

```
## [1] 2 4 4 6
```

```
# Returns the object with incomplete cases removed.
na.omit(heights)
```

```
## [1] 2 4 4 6
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

EXERCISE

1. Using this vector of heights in inches, create a new vector, `heights_no_na`, with the NAs removed.

```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64, 70, 63, 65)
```

2. Use the function median() to calculate the median of the heights vector.
3. Use R to figure out how many people in the set are taller than 67 inches.

Chapter 6

Starting with Data

6.1 Download the data

Make a new RStudio Project and create a `data` directory in your RStudio Project.

You can download the file directly from dropbox using this link. Remember to put this file in a directory that makes sense (ex. in your RStudio Project directory in a `data` directory).

6.2 CSV files

Spreadsheet data is often saved in one of the following formats:

- A *Comma Separated Values* `.csv` file. You can think of a `.csv` file as a bare-bones spreadsheet where:
  - Each line in the file corresponds to one row of data/one observation.
  - Values for each line are separated with commas. In other words, the values of different variables are separated by commas.
  - The first line is often, but not always, a header row indicating the names of the columns/variables.
- An Excel `.xlsx` file. This format is based on Microsoft's proprietary Excel software. As opposed to a bare-bones `.csv` files, `.xlsx` Excel files sometimes contain a lot of meta-data, or put more simply, data about the data. Some examples of spreadsheet meta-data include the use of bold

and italic fonts, colored cells, different column widths, and formula macros.

- A Google Sheets file, which is a "cloud" or online-based way to work with a spreadsheet. Google Sheets allows you to download your data in both comma separated values `.csv` and Excel `.xlsx` formats however: go to the Google Sheets menu bar -> File -> Download as -> Select "Microsoft Excel" or "Comma-separated values."

There are several ways to import data into R. Here, we will use the tools every R installation comes with (ie. "base R") to import a comma-delimited file containing some data. We will need to load the file using a function called `read.csv()`.

EXERCISE

Before using the `read.csv()` function, use R's help feature to answer the following questions:

1. What is the default parameter for 'header' in the read.csv() function?
2. What argument would you have to change to read a file that was delimited by semicolons (;) rather than commas?
3. What argument would you have to change to read file in which numbers used commas for decimal separation (i.e. 1,00)?
4. What argument would you have to change to read in only the first 10,000 rows of a very large file?

6.3 Sidenote: Importing data from Excel

Excel is one of the most common formats, so we need to discuss how to make these files play nicely with R. The simplest way to import data from Excel is to save your Excel file in .csv format. You can then import into R right away. Sometimes you may not be able to do this (imagine you have data in 300 Excel files, are you going to open and export all of them?).

One common R package (a set of code with features you can download and add to your R installation) is the readxl package which can open and import Excel files.

6.4 Import data

Now let's read in some data. We're going to study a population of *Escherichia coli* (designated Ara-3), which were propagated for more than 40,000 generations in a glucose-limited minimal medium. This medium was supplemented with citrate, which the ancestral *E. coli* cannot metabolize in the aerobic conditions of the experiment. Sequencing of the populations at regular time points revealed that spontaneous citrate-using mutants (Cit+) appeared at around 31,000 generations in one of twelve populations. The dataset is stored as a comma separated value (CSV) file. This metadata describes information on the Ara-3 clones and the columns represent:

Column	Description
sample	clone name
generation	generation when sample frozen
clade	based on a parsimony tree
strain	ancestral strain
cit	citrate-using mutant status
run	sequence read archive sample ID
genome_size	size in Mbp

Read in the metadata file:

```
metadata <- read.csv("data/Ecoli_metadata.csv")
```

This statement doesn't produce any output because assignment doesn't display anything. If we want to check that our data has been loaded, we can print the variable's value: `metadata`.

This will output the entire dataset. If we only want to see the top 6 lines of the file to ensure it has been loaded, use the `head()` function:

```
head(metadata)
```

```
##     sample generation   clade strain
## 1   REL606          0     N/A REL606
## 2 REL1166A       2000 unknown REL606
## 3   ZDB409       5000 unknown REL606
## 4   ZDB429      10000      UC REL606
## 5   ZDB446      15000      UC REL606
## 6   ZDB458      20000 (C1,C2) REL606
##       cit       run genome_size
## 1 unknown                  4.62
## 2 unknown SRR098028        4.63
## 3 unknown SRR098281        4.60
## 4 unknown SRR098282        4.59
## 5 unknown SRR098283        4.66
## 6 unknown SRR098284        4.63
```

6.5 What are data frames?

A data frame is the standard way in R to store tabular data. It can also be thought of as a collection of vectors, all of which have the same length. Each vector represents a column and each vector can be a different data type (ex. characters, integers).

The `str()` function is useful to insepct the data types of the columns.

```
str(metadata)
```

```
## 'data.frame':    30 obs. of  7 variables:
##  $ sample     : Factor w/ 30 levels "CZB152","CZB154",..: 7 6 18 19 20 21 22 23 24 2
##  $ generation : int  0 2000 5000 10000 15000 20000 20000 20000 25000 25000 ...
##  $ clade      : Factor w/ 8 levels "(C1,C2)","C1",..: 6 8 8 7 7 1 1 1 2 4 ...
##  $ strain     : Factor w/ 1 level "REL606": 1 1 1 1 1 1 1 1 1 1 ...
##  $ cit        : Factor w/ 3 levels "minus","plus",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ run        : Factor w/ 30 levels "","SRR097977",..: 1 5 22 23 24 25 26 27 28 29
##  $ genome_size: num  4.62 4.63 4.6 4.59 4.66 4.63 4.62 4.61 4.65 4.59 ...
```

EXERCISE

Create a data frame using the `data.frame()` function:

```
# Create the data frame
BMI <- data.frame(gender = c("male", "male", "female"), height = c(152, 171.5, 165), we
# Pring the data frame
BMI
```

```
##   gender height weight age
## 1   male  152.0     81  42
## 2   male  171.5     93  36
## 3 female  165.0     78  26
```

Try the following functions on the BMI data frame:

- `dim()`
- `nrow()`
- `ncol()`
- `summary()`
- `rownames()`
- `colnames()`

What information does each of these functions tell you?

Saving or "writing" a data frame

We can save a data frame to a csv file using the `write.csv()` function. For example, to save the BMI data to bmi.csv:

```r
write.csv(BMI, file = "bmi.csv")
```

Ensure that you have successfully saved this csv file, then feel free to delete it.

6.6 What are factors?

Factors are the final major data structure we will introduce. Factors can be thought of as vectors which are specialized for categorical data. Given R's specialization for statistics, this make sense since categorial and continuous variables usually have different treatments. Sometimes you may want to have data treated as a factor, but in other cases, this may be undesirable.

Factors are stored as integers, and have labels associated with these unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

```r
str(metadata)
```

```
## 'data.frame':	30 obs. of  7 variables:
##  $ sample     : Factor w/ 30 levels "CZB152","CZB154",..: 7 6 18 19 20 21 22 23 24 25 ...
##  $ generation : int  0 2000 5000 10000 15000 20000 20000 20000 25000 25000 ...
##  $ clade      : Factor w/ 8 levels "(C1,C2)","C1",..: 6 8 8 7 7 1 1 1 2 4 ...
##  $ strain     : Factor w/ 1 level "REL606": 1 1 1 1 1 1 1 1 1 1 ...
##  $ cit        : Factor w/ 3 levels "minus","plus",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ run        : Factor w/ 30 levels "","SRR097977",..: 1 5 22 23 24 25 26 27 28 29 ...
##  $ genome_size: num  4.62 4.63 4.6 4.59 4.66 4.63 4.62 4.61 4.65 4.59 ...
```

Here we can see which columns/variables are factors. When we read in a file, any column that contains text is automatically assumed to be a factor. Once created, factors can only contain a pre-defined set values, known as levels. By default, R always sorts levels in alphabetical order.

For instance, we see that `cit` is a Factor w/ 3 levels, `"minus"`, `"plus"` and `"unknown"`.

The $ sign

To isolate a column or variable of a data frame, we use the `$` operator. This applies to any type of variable. For example:

```
metadata$sample
```

```
##  [1] REL606   REL1166A ZDB409   ZDB429  
##  [5] ZDB446   ZDB458   ZDB464*  ZDB467  
##  [9] ZDB477   ZDB483   ZDB16    ZDB357  
## [13] ZDB199*  ZDB200   ZDB564   ZDB30*  
## [17] ZDB172   ZDB158   ZDB143   CZB199  
## [21] CZB152   CZB154   ZDB83    ZDB87   
## [25] ZDB96    ZDB99    ZDB107   ZDB111  
## [29] REL10979 REL10988
## 30 Levels: CZB152 CZB154 ... ZDB99
```

```
metadata$genome_size
```

```
##  [1] 4.62 4.63 4.60 4.59 4.66 4.63 4.62 4.61
##  [9] 4.65 4.59 4.61 4.62 4.62 4.63 4.74 4.61
## [17] 4.77 4.63 4.79 4.59 4.80 4.76 4.60 4.75
## [25] 4.74 4.61 4.79 4.62 4.78 4.62
```

To determine the number of levels of any factor, use the `nlevels()` function:

```
nlevels(metadata$run)
```

```
## [1] 30
```

```
nlevels(metadata$clade)
```

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

```
nlevels(metadata$strain)
```

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

Let's extract the `cit` column of the metadata to a new object, so we don't end up modifying our original data frame.

```
cit <- metadata$cit
cit
```

```
##  [1] unknown unknown unknown unknown unknown
##  [6] unknown unknown unknown unknown unknown
## [11] unknown unknown minus   minus   plus   
## [16] minus   plus    minus   plus    minus  
## [21] plus    plus    minus   plus    plus   
## [26] minus   plus    minus   plus    minus  
## Levels: minus plus unknown
```

```
str(cit)
```

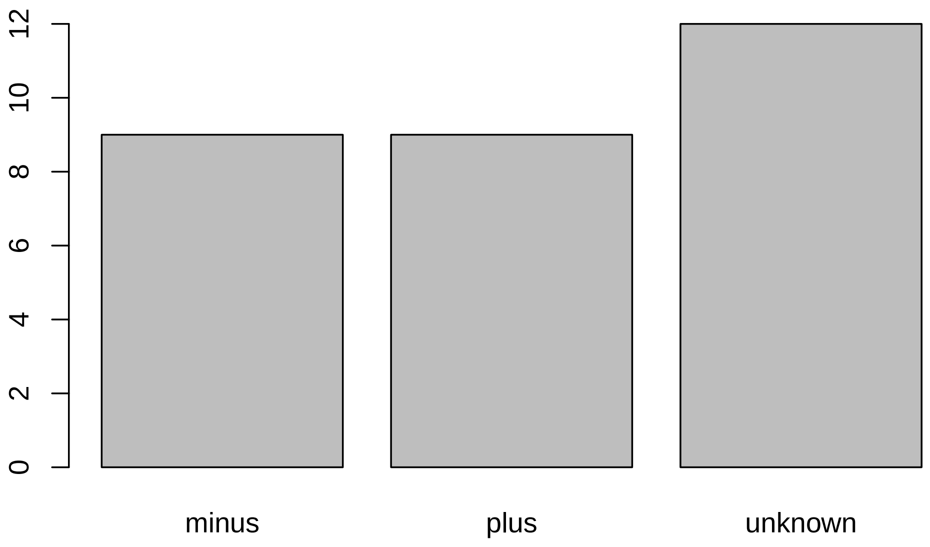
```
##  Factor w/ 3 levels "minus","plus",..: 3 3 3 3 3 3 3 3 3 3 ...
```

For the sake of efficiency, R stores the content of a factor as a vector of integers, which an integer is assigned to each of the possible levels. Recall levels are assigned in alphabetical order. In this case, the first item in our `cit` object is `unknown`, which happens to be the 3rd level of our factor, ordered alphabeticaly.

6.7 Plotting and ordering factors

One of the most common uses for factors will be when you plot categorical values. For example, suppose we want to know how many of our clones had citrate-using mutant status? We could generate a plot:

```
plot(cit)
```



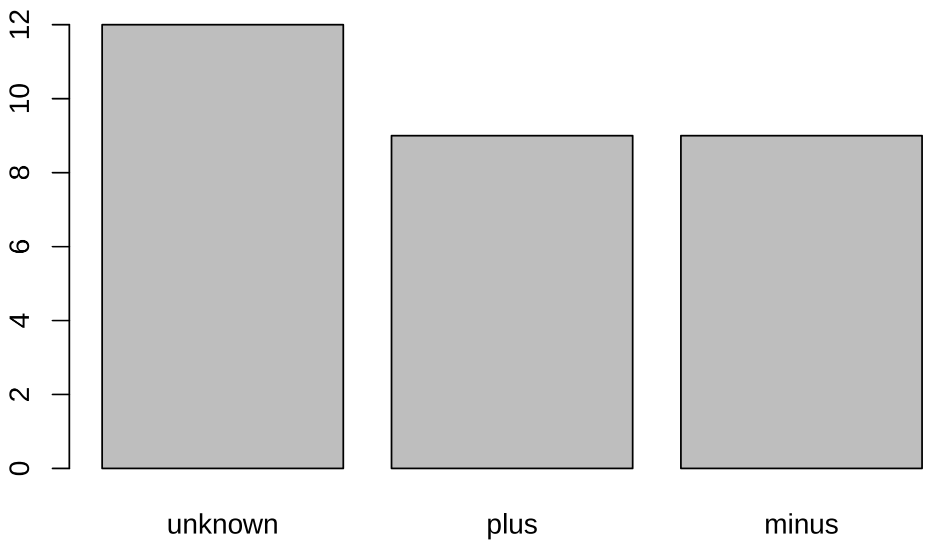
This isn't a particularly pretty example of a plot. But it can be a useful way to get acquainted with your data.

To reorder the levels of a factor:

```
cit <- factor(cit, levels = c("unknown", "plus", "minus"))
cit
```

```
##  [1] unknown unknown unknown unknown unknown
##  [6] unknown unknown unknown unknown unknown
## [11] unknown unknown minus   minus   plus
## [16] minus   plus    minus   plus    minus
## [21] plus    plus    minus   plus    plus
## [26] minus   plus    minus   plus    minus
## Levels: unknown plus minus
```

```
plot(cit)
```



6.8 Subsetting data frames

Subsetting data frames is similar to subsetting vectors with one major difference: date frames are two-dimensional. Therefore, to select a specific value we will use the `[]` notation again, but we will specify more than one value.

EXERCISE

Try the following indices on the metadata data frame:

- `metadata[1,1]`
- `metadata[4,2]`
- `metadata[2,]`
- `metadata[1:4,1]`
- `metadata[,2]`
- `metadata$run`
- `metadata[metadata$cit == "plus",]`

What information does the last line tell you?

You can assign a subset of your data frame to a new object. For example, to create a new data frame of only observations from cit- samples:

```
cit_minus <- metadata[metadata$cit == "plus",]
cit_minus
```

```
##      sample generation clade strain  cit
## 15   ZDB564      31500  Cit+ REL606 plus
## 17   ZDB172      32000  Cit+ REL606 plus
## 19   ZDB143      32500  Cit+ REL606 plus
## 21   CZB152      33000  Cit+ REL606 plus
## 22   CZB154      33000  Cit+ REL606 plus
## 24    ZDB87      34000    C2 REL606 plus
## 25    ZDB96      36000  Cit+ REL606 plus
## 27   ZDB107      38000  Cit+ REL606 plus
## 29 REL10979      40000  Cit+ REL606 plus
##          run genome_size
## 15 SRR098289        4.74
## 17 SRR098042        4.77
## 19 SRR098040        4.79
## 21 SRR097977        4.80
## 22 SRR098026        4.76
## 24 SRR098035        4.75
## 25 SRR098036        4.74
## 27 SRR098038        4.79
## 29 SRR098029        4.78
```

6.9 Other data types

So far, we have looked at three data types: vectors, data frames, and factors. Here we will briefly cover matrices and lists. These two are not as useful but you may come across them. Later on in the workshop we will discuss tibbles, the tidyverse equivalent of the data frame.

Matrices

A matrix is a two-dimensional rectangular dataset. It can be created by using a vector input to the matrix function.

```
m <- matrix( c("a","a","b","c","b","a"), nrow = 2, ncol = 3, byrow = TRUE)
m
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "a"  "b"
## [2,] "c"  "b"  "a"
```

To output the dimensions of a matrix, use the `dim()` function.

```
dim(m)
```

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

R will list the number of rows first and the number of columns second. To list only the number of rows or the number of columns, use the `nrow()` and `ncol()`. The `length()` function will output the total number of elements.

```
nrow(m)
```

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

```
ncol(m)
```

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

```
length(m)
```

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

Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions, and even another list.

```
# Create a list
list1 <- list(c(2,5,3),21.3, BMI)
list1
```

```
## [[1]]
## [1] 2 5 3
##
## [[2]]
## [1] 21.3
##
## [[3]]
##   gender height weight age
## 1   male  152.0     81  42
## 2   male  171.5     93  36
## 3 female  165.0     78  26
```

Chapter 7

Conditional Statements and Loops

7.1 Conditional statements

Decision making is an important part of programming. This can be achieved in R programming using conditional statements such as `if` and `if...else`.

if

The syntax of an if statement is:

```r
if (test_expression) {
  do_this
}
```

```r
x <- 5
if (x > 0) {
  print("positive number")
}
```

```
## [1] "positive number"
```

if...else

The syntax of an if...else statement is:

```r
if (test_expression) {
  do_this
} else {
    do_that
}
```

The else part is optional and is only evaluated if test_expression is FALSE. It is important that the `else` word be in the same line as the closing brace of the `if` statement.

```
x <- 1
if (x > 0){
  print("positive number")
} else {
    print("negative number")
}
```

```
## [1] "positive number"
```

Nested if...else statements

You can have more than two test expressions:

```
if (test_expression1) {
  statement1
} else if (test_expression2) {
    statement2
} else {
      statement4
}
```

```
x <- 0
if (x < 0) {
   print("negative number")
} else if (x > 0) {
   print("positive number")
} else {
   print("zero")
}
```

```
## [1] "zero"
```

EXERCISE

(~5 mins)

Write a simple if...else statement to check if 5 is an odd number and if it is print “I am odd”, otherwise print “I am even”.

7.2 Loops

Conceptually, a loop is a way to repeat a sequence of instructions under certain conditions. They allow you to automate parts of your code that are in need of repetition.

The easiest and most frequently used loop in R is a for loop. Here is a demonstration of using loops.

```r
year <- c(2015,2016,2017,2018)

for(i in 1:length(year)) {
  print(year[i])
}
```

```
## [1] 2015
## [1] 2016
## [1] 2017
## [1] 2018
```

```r
for(i in 1:length(year)) {
  print(paste0("the year is ",year[i]))
}
```

```
## [1] "the year is 2015"
## [1] "the year is 2016"
## [1] "the year is 2017"
## [1] "the year is 2018"
```

EXERCISE

(~10 mins)

Challenging. Using the *E. coli* metadata from the last section, write a `for` loop containing an `if...else` statement to change the genome size into categorical variable with two levels: large and small. Do this by translating the following sentence into R code: for every element in the genome_size variable, if the genome size is greater than the mean, change it to “large”, otherwise, change it to “small”.

Chapter 8

Manipulating Data in the tidyverse

8.1 What is the Tidyverse

science. All packages share an underlying design philosophy, grammar, and data structures. These packages include:

- `dplyr` for data manipulation
- `tibble` for data organizing
- `ggplot2` for data visualization
- `tidyr` for *tidy*-ing your data
- `readr` for reading data into R

When we installed the `tidyverse` package, we installed all of the above packages. This means that all of the specialized functions of these packages are available to use.

8.2 Tibbles

Tibbles are a modern take on data frames. They keep the features that have stood the test of time and drop the features that used to be convenient but are now frustrating (ie. converting character vectors to factors).

Create a tibble

You can create a tibble using the `tibble()` function:

```r
# Create
friends_data <- tibble(
  name = c("Nicolas", "Thierry", "Bernard", "Jerome"),
  age = c(27, 25, 29, 26),
  height = c(180, 170, 185, 169),
  married = c(TRUE, FALSE, TRUE, TRUE)
)
# Print
friends_data
```

```
## # A tibble: 4 x 4
##   name       age height married
##   <chr>    <dbl>  <dbl> <lgl>
## 1 Nicolas     27    180 TRUE
## 2 Thierry     25    170 FALSE
## 3 Bernard     29    185 TRUE
## 4 Jerome      26    169 TRUE
```

Reading in tibbles

Tibbles can be read into R using the `read_csv()` function. Note that the `read_csv()` function is different than the `read.csv()` function. The major difference is that `read_csv()` will store the data as a tibble and `read.csv()` will store the data as a data frame.

EXERCISE

Use the `read_csv()` function to read in the `Ecoli_metadata.csv` file. Assign the data to an object called `metadata2` to avoid writing over the object `metadata`.

How is it different that the `metadata`object?

Data frames versus tibbles

For some applications, you will need to use data frames and for others tibbles. You will find that some older functions don't work on tibbles. They can be easily converted using the functions `as.data.frame()` or `as_tibble()`:

```
head(metadata)
```

```
##     sample generation   clade strain
## 1   REL606          0     N/A REL606
## 2 REL1166A       2000 unknown REL606
## 3   ZDB409       5000 unknown REL606
## 4   ZDB429      10000      UC REL606
## 5   ZDB446      15000      UC REL606
## 6   ZDB458      20000 (C1,C2) REL606
##       cit       run genome_size
## 1 unknown                  4.62
## 2 unknown SRR098028        4.63
## 3 unknown SRR098281        4.60
## 4 unknown SRR098282        4.59
## 5 unknown SRR098283        4.66
## 6 unknown SRR098284        4.63
```

```
metadata_tib <- as_tibble(metadata)
metadata_tib
```

```
## # A tibble: 30 x 7
##    sample generation clade strain cit
##    <fct>       <int> <fct> <fct>  <fct>
##  1 REL606          0 N/A   REL606 unkn~
##  2 REL11~       2000 unkn~ REL606 unkn~
##  3 ZDB409       5000 unkn~ REL606 unkn~
##  4 ZDB429      10000 UC    REL606 unkn~
##  5 ZDB446      15000 UC    REL606 unkn~
##  6 ZDB458      20000 (C1,~ REL606 unkn~
##  7 ZDB46~      20000 (C1,~ REL606 unkn~
##  8 ZDB467      20000 (C1,~ REL606 unkn~
##  9 ZDB477      25000 C1    REL606 unkn~
## 10 ZDB483      25000 C3    REL606 unkn~
## # ... with 20 more rows, and 2 more
## #   variables: run <fct>, genome_size <dbl>
```

```
metadata_df <- as.data.frame(metadata_tib)
head(metadata_df)
```

```
##     sample generation   clade strain
## 1   REL606          0     N/A REL606
## 2 REL1166A       2000 unknown REL606
## 3   ZDB409       5000 unknown REL606
## 4   ZDB429      10000      UC REL606
## 5   ZDB446      15000      UC REL606
## 6   ZDB458      20000 (C1,C2) REL606
##       cit       run genome_size
## 1 unknown                  4.62
## 2 unknown SRR098028        4.63
## 3 unknown SRR098281        4.60
## 4 unknown SRR098282        4.59
## 5 unknown SRR098283        4.66
## 6 unknown SRR098284        4.63
```

As you will see in the following sections, many of the useful data frame functions are the same for tibbles.

8.3 Exploring tibbles

We can explore the contents of a tibble in several ways. By typing the name of the tibble in the console, we can view the first ten rows of a tibble as above, which tells us lots of information about the column types and the number of rows. We can also use the `glimpse()` function:

```
glimpse(metadata_tib)
```

```
## Observations: 30
## Variables: 7
## $ sample      <fct> REL606, REL1166A, Z...
## $ generation  <int> 0, 2000, 5000, 1000...
## $ clade       <fct> "N/A", "unknown", "...
## $ strain      <fct> REL606, REL606, REL...
## $ cit         <fct> unknown, unknown, u...
## $ run         <fct> , SRR098028, SRR098...
## $ genome_size <dbl> 4.62, 4.63, 4.60, 4...
```

As with data frames, we can return a vector containing the values of a variable (column) using the $ sign:

```
metadata_tib$generation
```

```
##  [1]     0  2000  5000 10000 15000 20000
##  [7] 20000 20000 25000 25000 30000 30000
## [13] 31500 31500 31500 32000 32000 32500
## [19] 32500 33000 33000 33000 34000 34000
## [25] 36000 36000 38000 38000 40000 40000
```

Again, similar to data frames, we can use the subsetting operator [] directly on tibbles. A tibble is two-dimensional, so we must pass two arguments to the [] operator; the first indicates the row(s) we require and the second indicates the column(s). To return the value in row 10, column 1:

```
metadata_tib[10,1]
```

```
## # A tibble: 1 x 1
##   sample
##   <fct>
## 1 ZDB483
```

Similarly, to return the values in rows 25 to 30, and columns 1 to 3:

```
metadata_tib[25:30, 1:3]
```

If we leave an index blank, this acts as a wildcard and matches all of the rows or columns:

```
metadata_tib[22,]
```

```
## # A tibble: 1 x 7
##   sample generation clade strain cit   run
##   <fct>       <int> <fct> <fct>  <fct> <fct>
## 1 CZB154      33000 Cit+  REL606 plus  SRR0~
## # ... with 1 more variable:
## #   genome_size <dbl>
```

```
metadata_tib[,1:3]
```

```
## # A tibble: 30 x 3
##    sample   generation clade
##    <fct>         <int> <fct>
##  1 REL606            0 N/A
##  2 REL1166A       2000 unknown
##  3 ZDB409         5000 unknown
##  4 ZDB429        10000 UC
##  5 ZDB446        15000 UC
##  6 ZDB458        20000 (C1,C2)
##  7 ZDB464*       20000 (C1,C2)
##  8 ZDB467        20000 (C1,C2)
##  9 ZDB477        25000 C1
## 10 ZDB483        25000 C3
## # ... with 20 more rows
```

You can also refer to columns by name with quotation marks:

```
metadata_tib[,"sample"]
```

```
## # A tibble: 30 x 1
##    sample
##    <fct>
##  1 REL606
##  2 REL1166A
##  3 ZDB409
##  4 ZDB429
##  5 ZDB446
##  6 ZDB458
##  7 ZDB464*
##  8 ZDB467
##  9 ZDB477
## 10 ZDB483
## # ... with 20 more rows
```

Note that subsetting a tibble using the [] method returns another tibble. In contrast, using the $ sign to extract a variable returns a vector:

```
metadata_tib$sample
```

```
##  [1] REL606   REL1166A ZDB409   ZDB429
##  [5] ZDB446   ZDB458   ZDB464*  ZDB467
##  [9] ZDB477   ZDB483   ZDB16    ZDB357
## [13] ZDB199*  ZDB200   ZDB564   ZDB30*
## [17] ZDB172   ZDB158   ZDB143   CZB199
## [21] CZB152   CZB154   ZDB83    ZDB87
## [25] ZDB96    ZDB99    ZDB107   ZDB111
## [29] REL10979 REL10988
## 30 Levels: CZB152 CZB154 ... ZDB99
```

For more information on tibbles see the tibbles vignette.

8.4 Tidy data

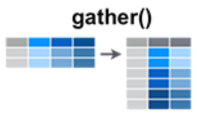
Tidy data is data that's easy to work with: it's easy to manage (with dplyr), visualize (with ggplot2) and model (with R's hundreds of modeling packages). Most importantly, tidy data is data where **each column is a variable** and **each row is an observation**.

Arranging your data in this way makes it easier to work with because you have a consistent way of referring to variables (as column names) and observations (as row indices). When use tidy data and tidy tools, you spend less time worrying about how to feed the output from one function into the input of another, and more time answering your questions about the data.

To tidy messy data, you first identify the variables in your dataset, then use the tools provided by `tidyr` to move them into columns. `tidyr` provides three main functions for tidying your messy data:

- `gather()`
- `separate()`
- `spread()`

`gather()`



The `gather()` function takes multiple columns, and gathers them into key-value pairs: it makes "wide" data longer. Here's an example. In this experiment we've given three people two different drugs and recorded their heart rate:

```
messy <- data.frame(
  name = c("Wilbur", "Petunia", "Gregory"),
  a = c(67, 80, 64),
  b = c(56, 90, 50)
)
messy
```

```
##      name  a  b
## 1  Wilbur 67 56
## 2 Petunia 80 90
## 3 Gregory 64 50
```

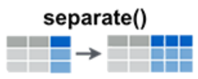
We have three variables (name, drug and heartrate), but only name is currently in a column. We use `gather()` to gather the a and b columns into key-value pairs of drug and heartrate:

```r
messy %>%
  gather(drug, heartrate, a:b)
```

```
##      name drug heartrate
## 1  Wilbur    a        67
## 2 Petunia    a        80
## 3 Gregory    a        64
## 4  Wilbur    b        56
## 5 Petunia    b        90
## 6 Gregory    b        50
```

Now each column is a variable and each row is an observation - tidy! Note: more about the "pipe" operator later.

`separate()`



Sometimes two variables are clumped together in one column. separate() allows you to tease them apart. We have some measurements of how much time people spend on their phones, measured at two locations (work and home), at two times. Each person has been randomly assigned to either treatment or control.

```r
messy
```

```
##   id       trt    work.T1   home.T1
## 1  1 treatment 0.08513597 0.6158293
## 2  2   control 0.22543662 0.4296715
## 3  3   control 0.27453052 0.6516557
## 4  4 treatment 0.27230507 0.5677378
##     work.T2    home.T2
## 1 0.1135090 0.05190332
## 2 0.5959253 0.26417767
## 3 0.3580500 0.39879073
## 4 0.4288094 0.83613414
```

To tidy this data, we first use gather() to turn columns work.T1, home.T1, work.T2 and home.T2 into a key-value pair of key and time:

```r
tidier <- messy %>%
  gather(key, time, -id, -trt)
tidier
```

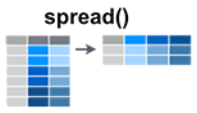
```
##    id       trt     key       time
## 1   1 treatment work.T1 0.08513597
## 2   2   control work.T1 0.22543662
## 3   3   control work.T1 0.27453052
## 4   4 treatment work.T1 0.27230507
## 5   1 treatment home.T1 0.61582931
## 6   2   control home.T1 0.42967153
## 7   3   control home.T1 0.65165567
## 8   4 treatment home.T1 0.56773775
## 9   1 treatment work.T2 0.11350898
## 10  2   control work.T2 0.59592531
## 11  3   control work.T2 0.35804998
## 12  4 treatment work.T2 0.42880942
## 13  1 treatment home.T2 0.05190332
## 14  2   control home.T2 0.26417767
## 15  3   control home.T2 0.39879073
## 16  4 treatment home.T2 0.83613414
```

Next we use separate() to split the key into location and time, using a regular expression to describe the character that separates them:

```
tidy <- tidier %>%
  separate(key, into = c("location", "timepoint"), sep = "\\.")
tidy
```

```
##    id       trt location timepoint
## 1   1 treatment     work        T1
## 2   2   control     work        T1
## 3   3   control     work        T1
## 4   4 treatment     work        T1
## 5   1 treatment     home        T1
## 6   2   control     home        T1
## 7   3   control     home        T1
## 8   4 treatment     home        T1
## 9   1 treatment     work        T2
## 10  2   control     work        T2
## 11  3   control     work        T2
## 12  4 treatment     work        T2
## 13  1 treatment     home        T2
## 14  2   control     home        T2
## 15  3   control     home        T2
## 16  4 treatment     home        T2
##          time
## 1  0.08513597
## 2  0.22543662
## 3  0.27453052
## 4  0.27230507
## 5  0.61582931
## 6  0.42967153
## 7  0.65165567
## 8  0.56773775
## 9  0.11350898
## 10 0.59592531
## 11 0.35804998
## 12 0.42880942
## 13 0.05190332
## 14 0.26417767
## 15 0.39879073
## 16 0.83613414
```

`spread()`



The function `spread()` does the reverse of `gather()`. It takes two columns and spreads them into multiple columns. It produces "wide" data from "long" data. Typically you will want your data in a wide form so you likely won't use this much. See the documentation for more information.

8.5 What is dplyr?

The package `dplyr` is a package that tries to provide easy tools for the most common data manipulation tasks. It is built to work directly with tibbles. The thinking behind it was largely inspired by the package `plyr` which has been in use for some time but suffered from being slow in some cases. `dplyr` addresses this by porting much of the computation to C++. An additional feature is the ability to work with data stored directly in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query returned.

This addresses a common problem with R in that all operations are conducted in memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can have a database of many 100s GB, conduct queries on it directly and pull back just what you need for analysis in R.

8.6 Selecting columns and filtering rows

We're going to learn some of the most common `dplyr` functions: `select()`, `filter()`, `mutate()`, `group_by()`, and `summarize()`. To select columns of a data frame, use `select()`. The first argument to this function is the data frame (tibble), and the subsequent arguments are the columns to keep.

```
select(metadata, sample, clade, cit, genome_size)
```

```
##       sample   clade     cit genome_size
## 1     REL606     N/A unknown        4.62
## 2   REL1166A unknown unknown        4.63
## 3     ZDB409 unknown unknown        4.60
## 4     ZDB429      UC unknown        4.59
## 5     ZDB446      UC unknown        4.66
## 6     ZDB458 (C1,C2) unknown        4.63
## 7    ZDB464* (C1,C2) unknown        4.62
## 8     ZDB467 (C1,C2) unknown        4.61
## 9     ZDB477      C1 unknown        4.65
## 10    ZDB483      C3 unknown        4.59
## 11     ZDB16      C1 unknown        4.61
## 12    ZDB357      C2 unknown        4.62
## 13   ZDB199*      C1   minus        4.62
## 14    ZDB200      C2   minus        4.63
## 15    ZDB564    Cit+    plus        4.74
## 16    ZDB30*      C3   minus        4.61
## 17    ZDB172    Cit+    plus        4.77
## 18    ZDB158      C2   minus        4.63
## 19    ZDB143    Cit+    plus        4.79
## 20    CZB199      C1   minus        4.59
## 21    CZB152    Cit+    plus        4.80
## 22    CZB154    Cit+    plus        4.76
## 23     ZDB83    Cit+   minus        4.60
## 24     ZDB87      C2    plus        4.75
## 25     ZDB96    Cit+    plus        4.74
## 26     ZDB99      C1   minus        4.61
## 27    ZDB107    Cit+    plus        4.79
## 28    ZDB111      C2   minus        4.62
## 29  REL10979    Cit+    plus        4.78
## 30  REL10988      C2   minus        4.62
```

To choose rows, use filter():

```
filter(metadata, cit == "plus")
```

```
##      sample generation clade strain  cit
## 1    ZDB564      31500  Cit+ REL606 plus
## 2   ZDB172      32000  Cit+ REL606 plus
## 3   ZDB143      32500  Cit+ REL606 plus
## 4   CZB152      33000  Cit+ REL606 plus
## 5   CZB154      33000  Cit+ REL606 plus
## 6    ZDB87      34000    C2 REL606 plus
## 7    ZDB96      36000  Cit+ REL606 plus
## 8   ZDB107      38000  Cit+ REL606 plus
## 9 REL10979      40000  Cit+ REL606 plus
##         run genome_size
## 1 SRR098289        4.74
## 2 SRR098042        4.77
## 3 SRR098040        4.79
## 4 SRR097977        4.80
## 5 SRR098026        4.76
## 6 SRR098035        4.75
## 7 SRR098036        4.74
## 8 SRR098038        4.79
## 9 SRR098029        4.78
```

EXERCISE

1. Keep only the rows that are not cit = plus.
2. Keep only the rows for samples taken in the first 30000 generations.

8.7 Pipes

But what if you wanted to `select` and `filter`? There are three ways to do this: use intermediate steps, nested functions, or pipes. With the intermediate steps, you essentially create a temporary data frame and use that as input to the next function. This can clutter up your workspace with lots of objects. You can also nest functions (i.e. one function inside of another). This is handy, but can be difficult to read if too many functions are nested as the process from inside out. The last option, pipes, are a fairly recent addition to R. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to many things to the same data set. Pipes in R look like `%>%` and are made available via the `magrittr` package installed as part of tidyverse. If you're familiar with the Unix shell, you may already have used pipes to pass the output from one command to the next. The concept is the same, except the shell uses the `|` character rather than R's pipe operator `%>%`.

The pipe operator can be tedious to type. In Rstudio pressing `Ctrl + Shift + M` under Windows / Linux will insert the pipe operator. On the mac, use `+ Shift + M`.

```
metadata %>%
  filter(cit == "plus") %>%
  select(sample, generation, clade, cit)
```

```
##      sample generation clade  cit
## 1   ZDB564      31500  Cit+ plus
## 2   ZDB172      32000  Cit+ plus
## 3   ZDB143      32500  Cit+ plus
## 4   CZB152      33000  Cit+ plus
## 5   CZB154      33000  Cit+ plus
## 6    ZDB87      34000    C2 plus
## 7    ZDB96      36000  Cit+ plus
## 8   ZDB107      38000  Cit+ plus
## 9 REL10979      40000  Cit+ plus
```

In the above we use the pipe to send the data set first through `filter`, to keep rows where cit was equal to 'plus', and then through `select` to keep the sample and generation and clade columns. When the data frame is being passed to the `filter()` and `select()` functions through a pipe, we don't need to include it as an argument to these functions anymore.

EXERCISE

Does the order of `filter()` and `select()` above matter? Why or why not?

Answer to yourself or to a neighbour and then confirm your answer by trying it both ways.

Note that the above is the same as the nested version:

```
select(filter(metadata, cit == "plus"), sample, generation, clade)
```

```
##      sample generation clade
## 1   ZDB564      31500  Cit+
## 2   ZDB172      32000  Cit+
## 3   ZDB143      32500  Cit+
## 4   CZB152      33000  Cit+
## 5   CZB154      33000  Cit+
## 6    ZDB87      34000    C2
## 7    ZDB96      36000  Cit+
## 8   ZDB107      38000  Cit+
## 9 REL10979      40000  Cit+
```

If we wanted to create a new object with this smaller version of the data we could do so by assigning it a new name:

```
Ecoli_citplus <- metadata %>%
  filter(cit == "plus") %>%
  select(sample, generation, clade)

Ecoli_citplus
```

```
##     sample generation clade
## 1   ZDB564      31500  Cit+
## 2   ZDB172      32000  Cit+
## 3   ZDB143      32500  Cit+
## 4   CZB152      33000  Cit+
## 5   CZB154      33000  Cit+
## 6    ZDB87      34000    C2
## 7    ZDB96      36000  Cit+
## 8   ZDB107      38000  Cit+
## 9 REL10979      40000  Cit+
```

We can think of the `filter()` and `select()` functions as verbs in the sentence; they do things to the data flowing through the pipeline.

EXERCISE

Using pipes, subset `metadata` to include rows where the clade is 'Cit+' and keep only the columns `sample`, `cit`, and `genome_size`.
How many rows are in that tibble?

8.8 Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions or find the ratio of values in two columns. For this we'll use `mutate()`.

To create a new column of genome size in bp:

```
metadata %>%
  mutate(genome_bp = genome_size *1e6)
```

```
##      sample generation   clade strain
## 1    REL606          0     N/A REL606
## 2  REL1166A       2000 unknown REL606
## 3    ZDB409       5000 unknown REL606
## 4    ZDB429      10000      UC REL606
## 5    ZDB446      15000      UC REL606
## 6    ZDB458      20000 (C1,C2) REL606
## 7   ZDB464*      20000 (C1,C2) REL606
## 8    ZDB467      20000 (C1,C2) REL606
## 9    ZDB477      25000      C1 REL606
## 10   ZDB483      25000      C3 REL606
## 11    ZDB16      30000      C1 REL606
## 12   ZDB357      30000      C2 REL606
## 13  ZDB199*      31500      C1 REL606
## 14   ZDB200      31500      C2 REL606
## 15   ZDB564      31500    Cit+ REL606
## 16   ZDB30*      32000      C3 REL606
## 17   ZDB172      32000    Cit+ REL606
## 18   ZDB158      32500      C2 REL606
## 19   ZDB143      32500    Cit+ REL606
## 20   CZB199      33000      C1 REL606
## 21   CZB152      33000    Cit+ REL606
## 22   CZB154      33000    Cit+ REL606
## 23    ZDB83      34000    Cit+ REL606
## 24    ZDB87      34000      C2 REL606
## 25    ZDB96      36000    Cit+ REL606
## 26    ZDB99      36000      C1 REL606
## 27   ZDB107      38000    Cit+ REL606
## 28   ZDB111      38000      C2 REL606
## 29 REL10979      40000    Cit+ REL606
## 30 REL10988      40000      C2 REL606
##        cit       run genome_size genome_bp
## 1  unknown                  4.62   4620000
## 2  unknown SRR098028        4.63   4630000
## 3  unknown SRR098281        4.60   4600000
## 4  unknown SRR098282        4.59   4590000
## 5  unknown SRR098283        4.66   4660000
## 6  unknown SRR098284        4.63   4630000
## 7  unknown SRR098285        4.62   4620000
## 8  unknown SRR098286        4.61   4610000
## 9  unknown SRR098287        4.65   4650000
## 10 unknown SRR098288        4.59   4590000
## 11 unknown SRR098031        4.61   4610000
## 12 unknown SRR098280        4.62   4620000
## 13   minus SRR098044        4.62   4620000
## 14   minus SRR098279        4.63   4630000
## 15    plus SRR098289        4.74   4740000
## 16   minus SRR098032        4.61   4610000
## 17    plus SRR098042        4.77   4770000
## 18   minus SRR098041        4.63   4630000
## 19    plus SRR098040        4.79   4790000
## 20   minus SRR098027        4.59   4590000
## 21    plus SRR097977        4.80   4800000
## 22    plus SRR098026        4.76   4760000
## 23   minus SRR098034        4.60   4600000
## 24    plus SRR098035        4.75   4750000
## 25    plus SRR098036        4.74   4740000
## 26   minus SRR098037        4.61   4610000
## 27    plus SRR098038        4.79   4790000
## 28   minus SRR098039        4.62   4620000
## 29    plus SRR098029        4.78   4780000
## 30   minus SRR098030        4.62   4620000
```

EXERCISE

Use the `mutate()` function to mutate generation into a fraction from 0 to 1.

8.9 Split-apply-combine data analysis and the summarize() function

Many data analysis tasks can be approached using the “split-apply-combine” paradigm: split the data into groups, apply some analysis to each group, and then combine the results. `dplyr` makes this very easy through the use of the `group_by()` function, which splits the data into groups. When the data is grouped in this way `summarize()` can be used to collapse each group into a single-row summary. `summarize()` does this by applying an aggregation or summary function to each group. For example, if we wanted to group by citrate-using mutant status and find the number of rows of data for each status, we would do:

```
metadata %>%
  group_by(cit) %>%
  summarize(n())
```

```
## # A tibble: 3 x 2
##   cit     `n()`
##   <fct>   <int>
## 1 minus       9
## 2 plus        9
## 3 unknown    12
```

Here the summary function used was `n()` to find the count for each group. We can also apply many other functions to individual columns to get other summary statistics. For example, in the R base package we can use built-in functions like `mean`, `median`, `min`, and `max`. By default, all R functions operating on vectors that contains missing data will return `NA`. It’s a way to make sure that users know they have missing data, and make a conscious decision on how to deal with it. When dealing with simple statistics like the `mean`, the easiest way to ignore `NA` (the missing data) is to use `na.rm=TRUE` (rm stands for remove).

EXERCISE

Use the `group_by()` and `summarize()` to view mean `genome_size` by mutant status (ie. cit).

You can group by multiple columns too:

```
metadata %>%
  group_by(cit, clade) %>%
  summarize(mean_size = mean(genome_size, na.rm = TRUE))
```

```
## # A tibble: 13 x 3
## # Groups:   cit [3]
##    cit     clade   mean_size
##    <fct>   <fct>       <dbl>
##  1 minus   C1           4.61
##  2 minus   C2           4.62
##  3 minus   C3           4.61
##  4 minus   Cit+         4.6
##  5 plus    C2           4.75
##  6 plus    Cit+         4.77
##  7 unknown (C1,C2)      4.62
##  8 unknown C1           4.63
##  9 unknown C2           4.62
## 10 unknown C3           4.59
## 11 unknown N/A          4.62
## 12 unknown UC           4.62
## 13 unknown unknown      4.62
```

Looks like for one of these clones, the clade is missing. We could then discard those rows using filter():

```
metadata %>%
  group_by(cit, clade) %>%
  summarize(mean_size = mean(genome_size, na.rm = TRUE)) %>%
  filter(!is.na(clade))
```

```
## # A tibble: 13 x 3
## # Groups:   cit [3]
##    cit     clade   mean_size
##    <fct>   <fct>       <dbl>
##  1 minus   C1           4.61
##  2 minus   C2           4.62
##  3 minus   C3           4.61
##  4 minus   Cit+         4.6
##  5 plus    C2           4.75
##  6 plus    Cit+         4.77
##  7 unknown (C1,C2)      4.62
##  8 unknown C1            4.63
##  9 unknown C2            4.62
## 10 unknown C3            4.59
## 11 unknown N/A           4.62
## 12 unknown UC            4.62
## 13 unknown unknown       4.62
```

You can also summarize multiple variables at the same time:

```
metadata %>%
  group_by(cit, clade) %>%
  summarize(mean_size = mean(genome_size, na.rm = TRUE),
            min_generation = min(generation))
```

```
## # A tibble: 13 x 4
## # Groups:   cit [3]
##    cit     clade   mean_size min_generation
##    <fct>   <fct>       <dbl>          <int>
##  1 minus   C1           4.61          31500
##  2 minus   C2           4.62          31500
##  3 minus   C3           4.61          32000
##  4 minus   Cit+         4.6           34000
##  5 plus    C2           4.75          34000
##  6 plus    Cit+         4.77          31500
##  7 unknown (C1,C2)      4.62          20000
##  8 unknown C1           4.63          25000
##  9 unknown C2           4.62          30000
## 10 unknown C3           4.59          25000
## 11 unknown N/A          4.62              0
## 12 unknown UC           4.62          10000
## 13 unknown unknown      4.62           2000
```

For a summary of the tidyr and dplyr functions, see this Handy dplyr cheatsheet.

EXERCISE

Create a tibble containing each unique clade (removing the samples with unknown clades) and the rank of it's mean genome size. (note that ranking genome size will not sort the table; the row order will be unchanged. You can use the `arrange()` function to sort the table).

There are several functions for ranking observations, which handle tied values differently. For this exercise it doesn't matter which function you choose. Use the help options to find a ranking function.

8.10 Other great resources

- Data Wrangling tutorial - an excellent four part tutorial covering selecting data, filtering data, summarising and transforming your data.
- R for Data Science
- Data wrangling with R and RStudio - 55 minute webinar from RStudio

Chapter 9

Data Visualization with ggplot2

9.1 Data

This section uses a different data set from the same experiment - the LTEE (long-term evolution experiment). This data was pulished in: Tempo and mode of genome evolution in a 50,000-generation experiment Tenaillon et al 2016.

Download the data onto your computer from this dropbox link and move it into the `data` directory of your RStudio project.

EXERCISE

Read the data into R. Call this data `variants`. Ensure that you have read the data in by calling `head(variants)`.

You can further investigate the structure of the data frame using the `str()` function:

```
str(variants)
```

```
## 'data.frame':    801 obs. of  29 variables:
##  $ sample_id    : Factor w/ 3 levels "SRR2584863","SRR2584866",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ CHROM        : Factor w/ 1 level "CP000819.1": 1 1 1 1 1 1 1 1 1 1 ...
##  $ POS          : int  9972 263235 281923 433359 473901 648692 1331794 1733343 2103887 2333538
##  $ ID           : logi  NA NA NA NA NA NA ...
##  $ REF          : Factor w/ 59 levels "A","ACAGCCAGCCAGCCAGCCAGCCAGCCAGCCAG",..: 49 33 33 30 2
##  $ ALT          : Factor w/ 57 levels "A","AC","ACAGCCAGCCAGCCAGCCAGCCAGCCAGCCAGCCAG",..: 31 4
##  $ QUAL         : num  91 85 217 64 228 210 178 225 56 167 ...
##  $ FILTER       : logi  NA NA NA NA NA NA ...
##  $ INDEL        : logi  FALSE FALSE FALSE TRUE TRUE FALSE ...
##  $ IDV          : int  NA NA NA 12 9 NA NA NA 2 7 ...
##  $ IMF          : num  NA NA NA 1 0.9 ...
##  $ DP           : int  4 6 10 12 10 10 8 11 3 7 ...
##  $ VDB          : num  0.0257 0.0961 0.7741 0.4777 0.6595 ...
##  $ RPB          : num  NA 1 NA NA NA NA NA NA NA NA ...
##  $ MQB          : num  NA 1 NA NA NA NA NA NA NA NA ...
##  $ BQB          : num  NA 1 NA NA NA NA NA NA NA NA ...
##  $ MQSB         : num  NA NA 0.975 1 0.916 ...
##  $ SGB          : num  -0.556 -0.591 -0.662 -0.676 -0.662 ...
##  $ MQ0F         : num  0 0.167 0 0 0 ...
##  $ ICB          : logi  NA NA NA NA NA NA ...
##  $ HOB          : logi  NA NA NA NA NA NA ...
##  $ AC           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ AN           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ DP4          : Factor w/ 217 levels "0,0,0,2","0,0,0,3",..: 3 132 73 141 176 104
##  $ MQ           : int  60 33 60 60 60 60 60 60 60 60 ...
##  $ Indiv        : Factor w/ 3 levels "/home/dcuser/dc_workshop/results/bam/SRR25848
##  $ gt_PL        : Factor w/ 206 levels "100,0","103,0",..: 16 10 134 198 142 127 93
##  $ gt_GT        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ gt_GT_alleles: Factor w/ 57 levels "A","AC","ACAGCCAGCCAGCCAGCCAGCCAGCCAGCCAGCCAC
```

EXERCISE

Take a few minutes to familiarize yourself with this dataset. There are a lot of variables (how many?), so only worry about the ones listed below.

Column	Description
sample_id	sample ID
CHROM	contig location where the variation occurs
POS	position within the contig where the variation occurs
REF	reference genotype (forward strand)
ALT	sample genotype (forward strand)
QUAL	Phred-scaled probablity that the observed variant exists at this site (higher is better)
INDEL	whether the variant is an indel
IDV	length of indel
IMF	maximum fraction of reads supporting an indel
DP	the depth per allele by sample and coverage
MQ	mapping quality
Indiv	name of file

9.2 Plotting with ggplot2

We start by loading the `ggplot2` package:

```
library(ggplot2)
```

`ggplot2` is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatter plot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

`ggplot2` functions like data in the 'long' format, i.e., a column for every dimension, and a row for every observation. Well-structured data will save you lots of time when making figures with `ggplot2`.

Graphics are built step by step by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

To build a ggplot, we will use the following basic template that can be used for different types of plots:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) +  <GEOM_FUNCTION>()
```

1. Specify which data set to use for the plot using the `data` argument:

```
ggplot(data = variants)
```

2. Define a "mapping" (using the aesthetic (`aes`) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g. as x/y positions or characteristics such as size, shape, color, etc:

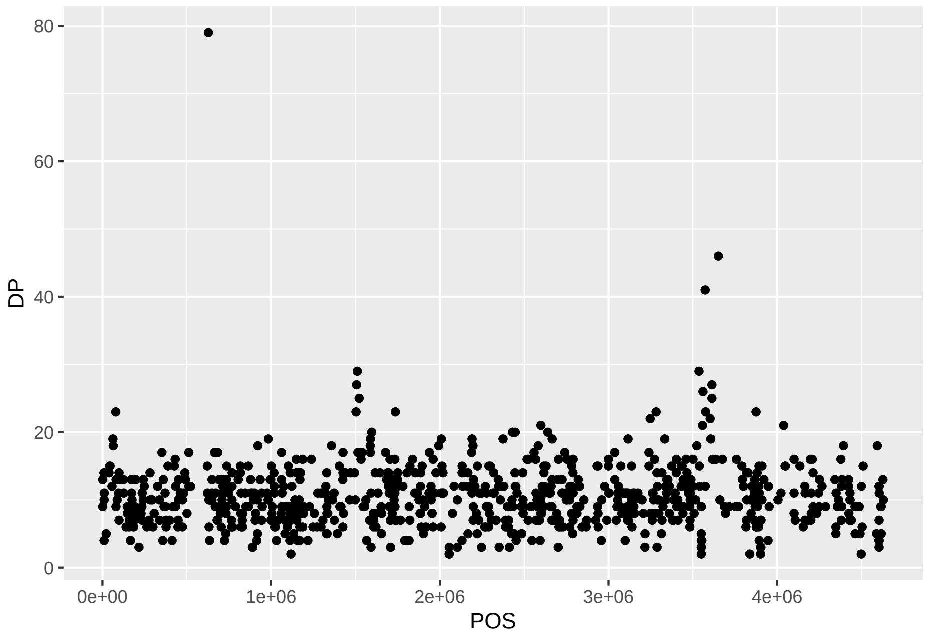
```
ggplot(data = variants, aes(x = POS, y = DP))
```

3. Add "geoms" – graphical representations of the data in the plot (points, lines, bars). `ggplot2` offers many different geoms; we will use some common ones today, including:

- `geom_point()` for scatter plots, dot plots, etc.
- `geom_boxplot()` for boxplots.
- `geom_histogram()` for histograms.
- `geom_barplot()` for barplots.
- `geom_line()` for trend lines, time series, etc.

To add a geom to the plot use the `+` operator. Because we have two continuous variables, let's use `geom_point()` first:

```
ggplot(data = variants, aes(x = POS, y = DP)) +
  geom_point()
```



The `+` in the `ggplot2` package is particularly useful because it allows you to modify existing `ggplot` objects. This means you can easily set up plot templates and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```r
# Assign plot to a variable
coverage_plot <- ggplot(data = variants, aes(x = POS, y = DP))

# Draw the plot
coverage_plot +
    geom_point()
```

Notes:

- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x- and y-axis mapping you set up in `aes()`.
- You can also specify mappings for a given geom independently of the mappings defined globally in the `ggplot()` function.
- The `+` sign used to add new layers must be placed at the end of the line containing the previous layer. If, instead, the `+` sign is added at the beginning of the line containing the new layer, `ggplot2` will not add the new layer and will return an error message.

```r
# This is the correct syntax for adding layers
coverage_plot +
  geom_point()
```

```r
# This will not add the new layer and will return an error message
coverage_plot
  + geom_point()
```

Principles of effective display

SOURCE: (Whitlock & Schluter, The Analysis of Biological Data)[http://whitlockschluter.zoology.ubc.ca/]

We will follow these metrics to create and evaluate figures:
1. Show the data
2. Make patterns in the data easy to see
3. Represent magnitudes honestly
4. Draw graphical elements clearly, minimizing clutter

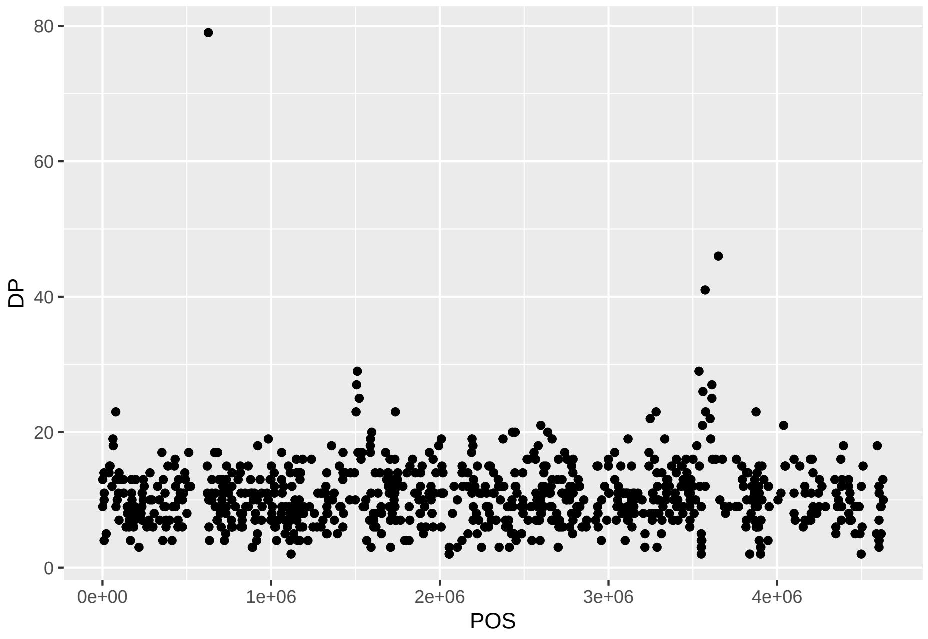
EXERCISE

Create a scatter plot (using the `geom_point()` function for quality (`QUAL`) versus coverage depth (`DP`).

Building plots iteratively

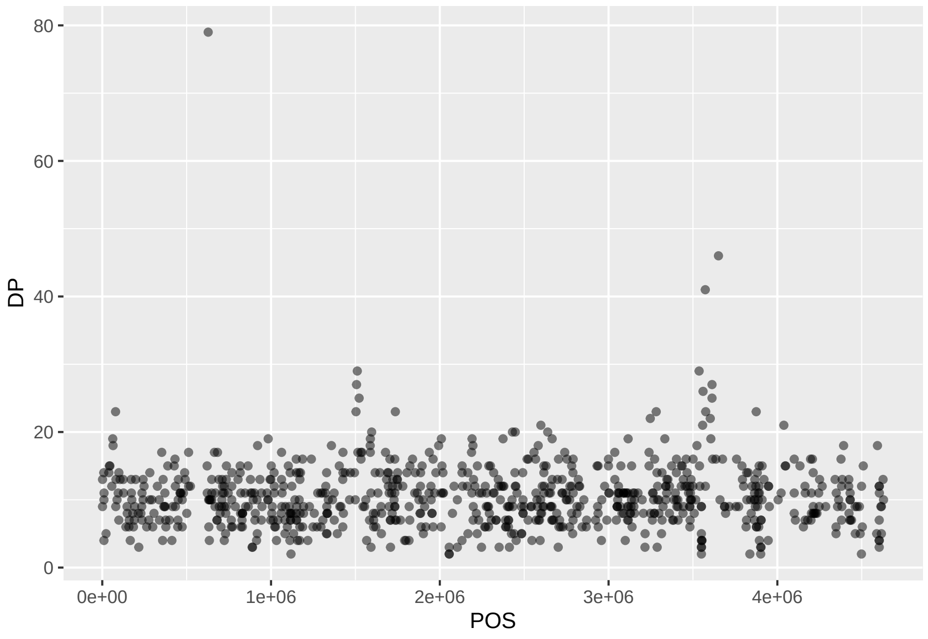
Building plots with `ggplot2` is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```r
ggplot(data = variants, aes(x = POS, y = DP)) +
  geom_point()
```



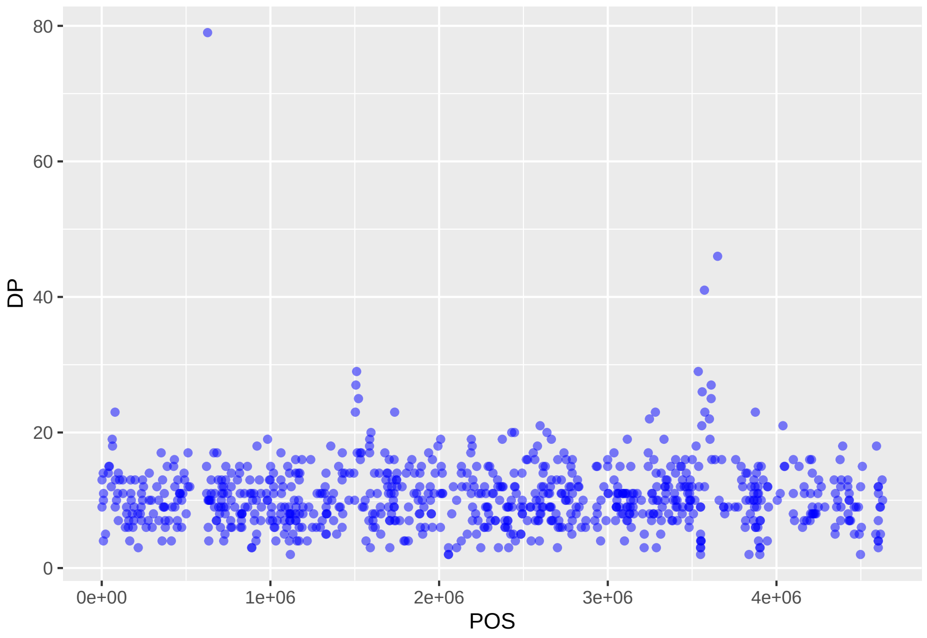
Then, we start modifying this plot to extract more information from it. For instance, we can add transparency (`alpha`) to avoid overplotting:

```r
ggplot(data = variants, aes(x = POS, y = DP)) +
    geom_point(alpha = 0.5)
```



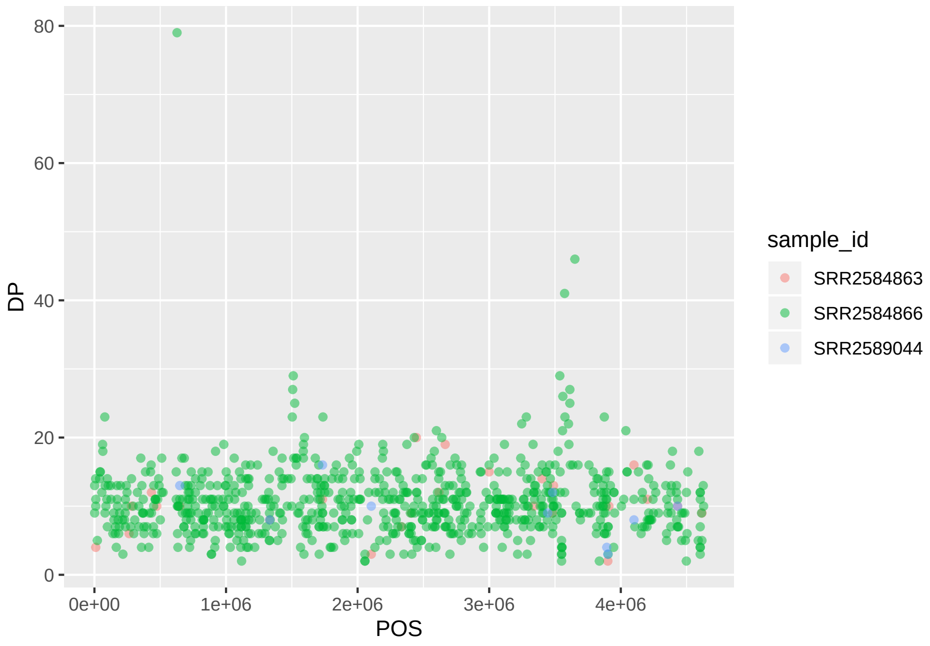
We can also add colors for all the points:

```
ggplot(data = variants, aes(x = POS, y = DP)) +
  geom_point(alpha = 0.5, color = "blue")
```



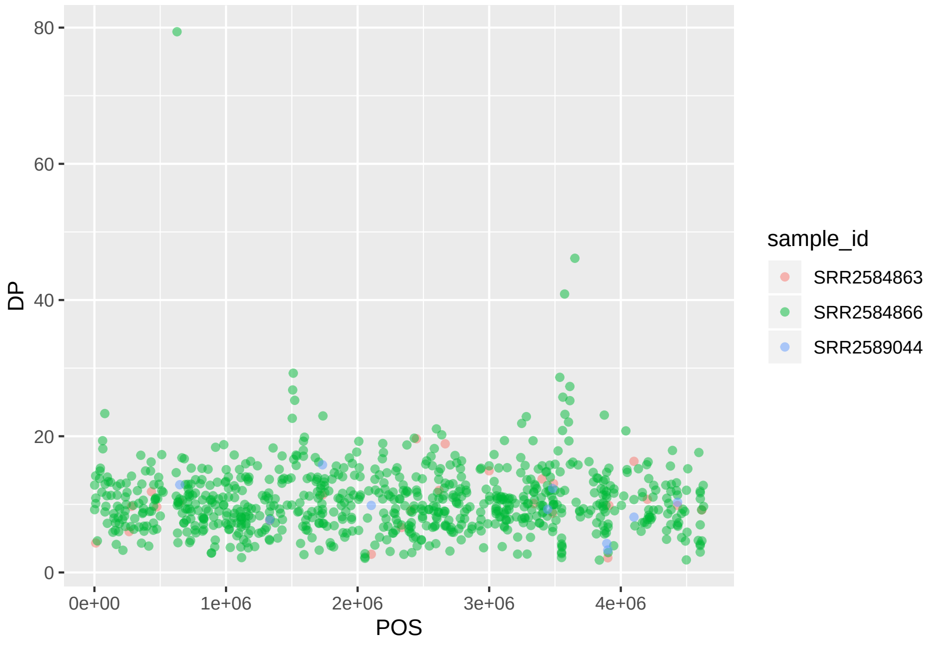
Or to color each species in the plot differently, you could use a vector as an input to the argument color. `ggplot2` will provide a different color corresponding to different values in the vector. Here is an example where we color with `sample_id`:

```
ggplot(data = variants, aes(x = POS, y = DP, color = sample_id)) +
  geom_point(alpha = 0.5)
```



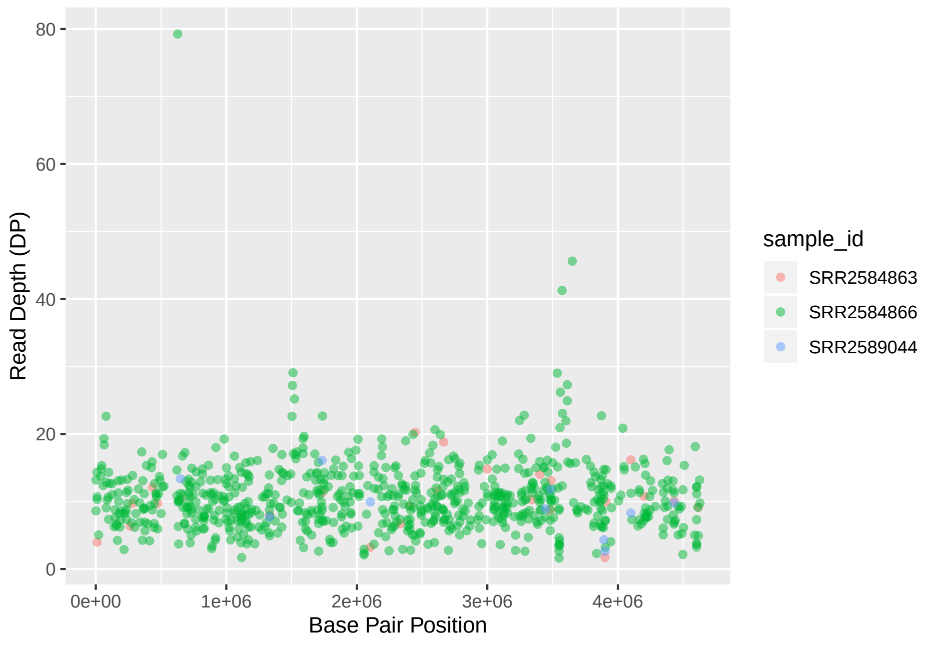
Notice that we can change the geom layer and colors will be still determined by `sample_id`:

```r
ggplot(data = variants, aes(x = POS, y = DP, color = sample_id)) +
  geom_jitter(alpha = 0.5)
```



To make our plot more readable, we can add axis labels:

```r
ggplot(data = variants, aes(x = POS, y = DP, color = sample_id)) +
  geom_jitter(alpha = 0.5) +
  labs(x = "Base Pair Position",
       y = "Read Depth (DP)")
```



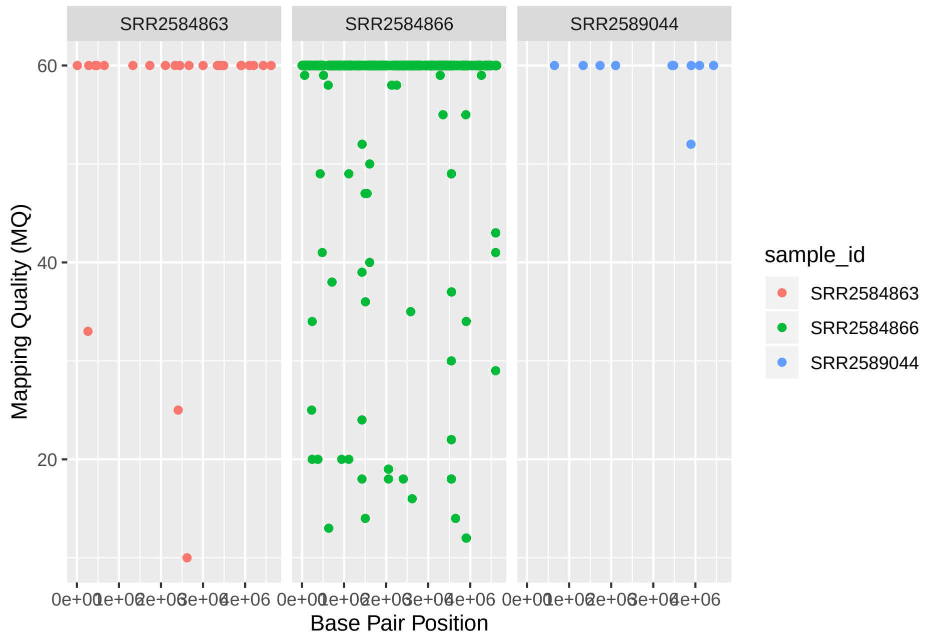
EXERCISE

Create a scatter plot of mapping quality (`MQ`) over position (`POS`) with the samples showing in different colors. Make sure to give your plot relevant axis labels.

9.3 Faceting

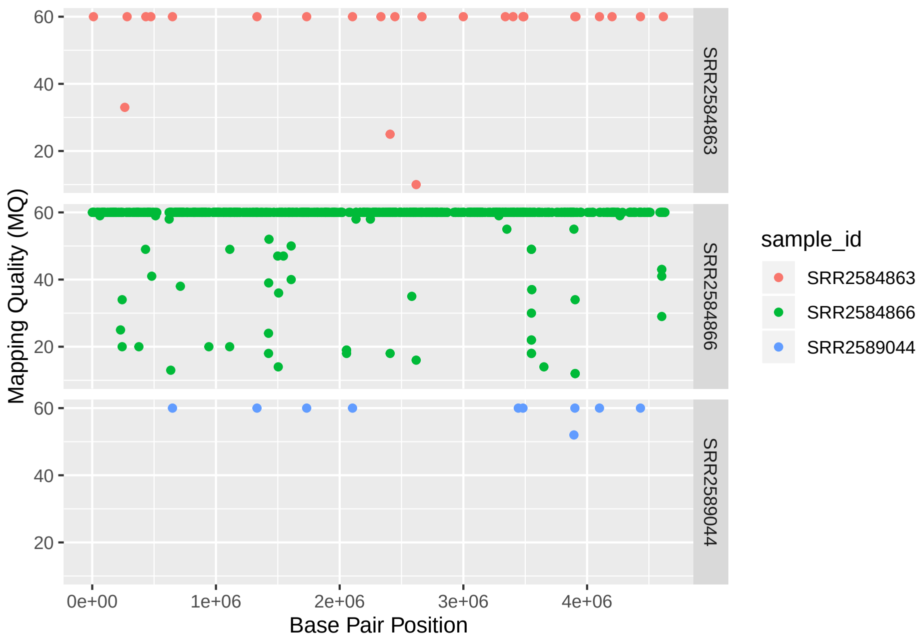
`ggplot2` has a special technique called faceting that allows the user to split one plot into multiple plots based on a factor included in the dataset. We will use it to split our mapping quality plot into three panels, one for each sample.

```r
ggplot(data = variants, aes(x = POS, y = MQ, color = sample_id)) +
 geom_point() +
 labs(x = "Base Pair Position",
      y = "Mapping Quality (MQ)") +
 facet_grid(. ~ sample_id)
```



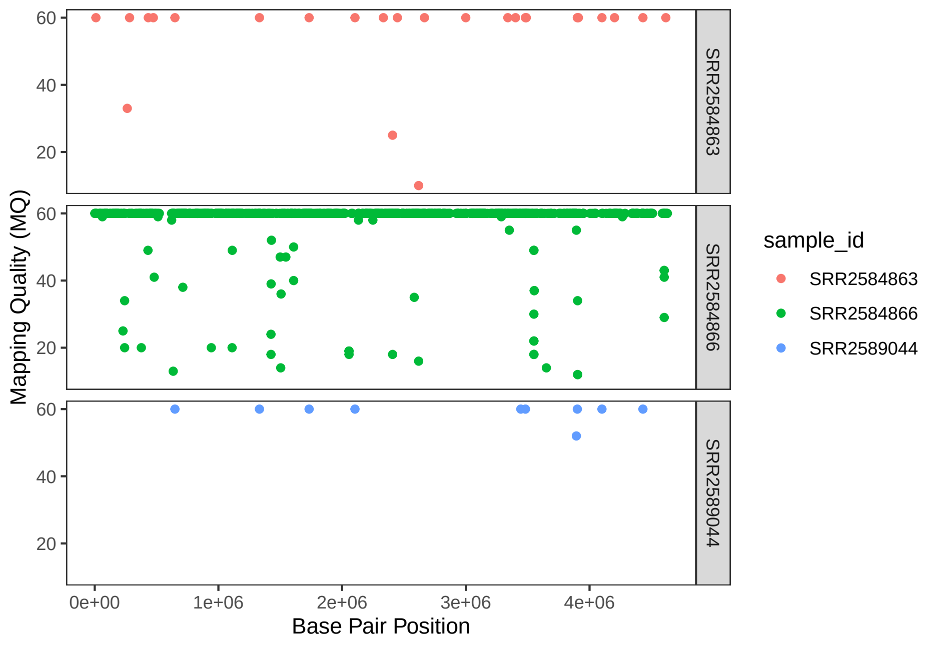
This looks ok, but it would be easier to read if the plot facets were stacked vertically rather than horizontally. The `facet_grid` geometry allows you to explicitly specify how you want your plots to be arranged via formula notation (`rows ~ columns`); a `.` can be used as a placeholder that indicates only one row or column).

```r
ggplot(data = variants, aes(x = POS, y = MQ, color = sample_id)) +
 geom_point() +
 labs(x = "Base Pair Position",
      y = "Mapping Quality (MQ)") +
 facet_grid(sample_id ~ .)
```



Usually plots with white background look more readable when printed. We can set the background to white using the function `theme_bw()`. Additionally, you can remove the grid:

```
ggplot(data = variants, aes(x = POS, y = MQ, color = sample_id)) +
  geom_point() +
  labs(x = "Base Pair Position",
       y = "Mapping Quality (MQ)") +
  facet_grid(sample_id ~ .) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



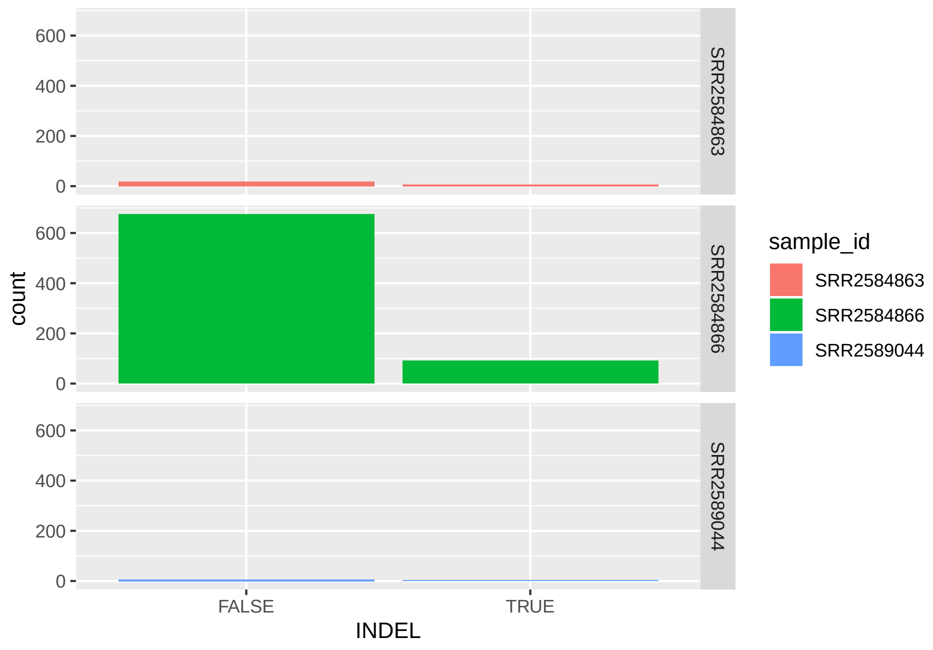
EXERCISE

Use what you just learned to create a scatter plot of PHRED scaled quality (`QUAL`) over position (`POS`) with the samples showing in different colors. Make sure to give your plot relevant axis labels.

9.4 Barplots

We can create barplots using the `geom_bar` geom. Let's make a barplot showing the number of variants for each sample that are indels.

```r
ggplot(data = variants, aes(x = INDEL, fill = sample_id)) +
  geom_bar() +
  facet_grid(sample_id ~ .)
```



EXERCISE

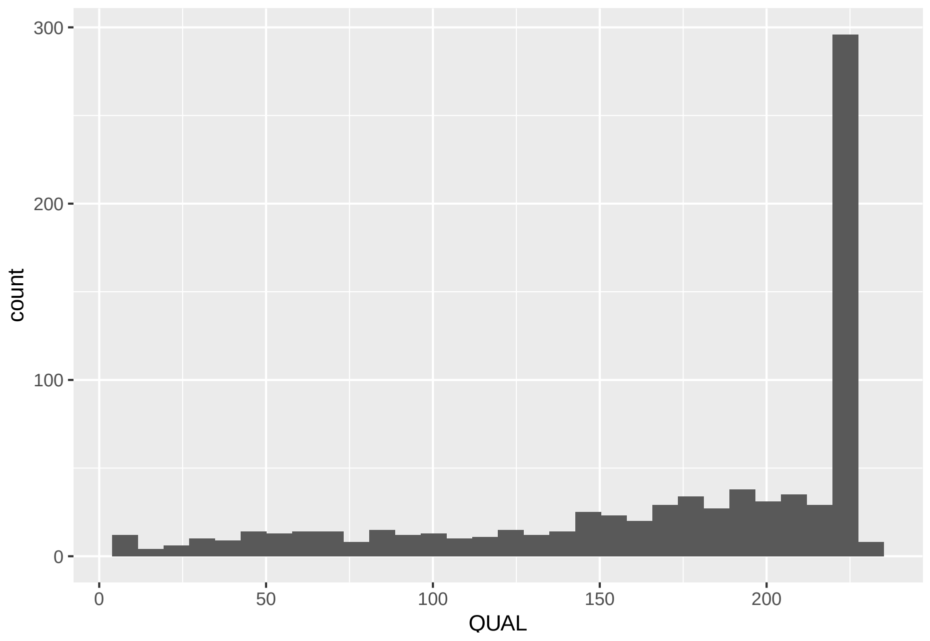
Since we already have the sample_id labels on the individual plot facets, we don't need the legend. Use the help file for `geom_bar` and any other online resources you want to use to remove the legend from the plot.

9.5 Histograms

Sometimes it can be useful to plot a single variable at a time. Usually this is for exploratory purposes - to get a feel for a variable. To do this, use the `geom_histogram()` function. For example, to look at the distribution of Qualities:

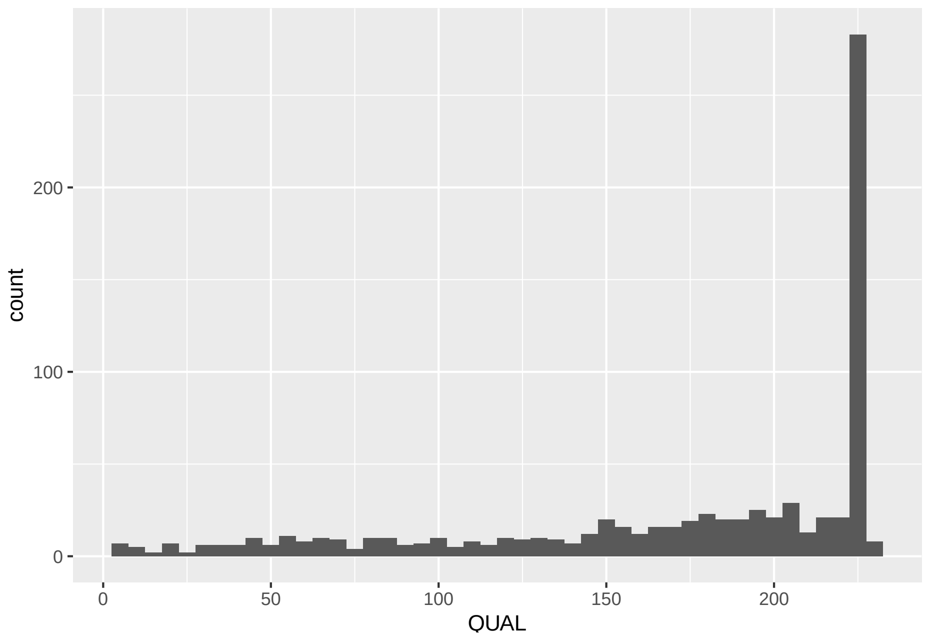
```r
ggplot(data = variants, aes(QUAL)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick
## better value with `binwidth`.
```



R is giving us a warning. We can choose to ignore this if we are able to see what we want in the plot. Otherwise, we can change the `binwidth` by using the binwidth argument:

```
ggplot(data = variants, aes(QUAL)) +
  geom_histogram(binwidth = 5)
```



EXERCISE

Create a plot that shows the distribution of Read Depth (DP) for each sample separately.

Bonus challenge. Create this plot for only two of the three samples IDs - the two with many fewer variants.

9.6 Boxplots

When there are a large number of values for a certain variable (like for one of the samples above), boxplots can be a useful way to display summary statistics like the median, and "spread" of a variable.

The Five Number Summary

The five number summary gives a quick look at the features of numerical variables. It consists of the variables:

- minimum
- 1st quartile
- median
- 3rd quartile

- maximum

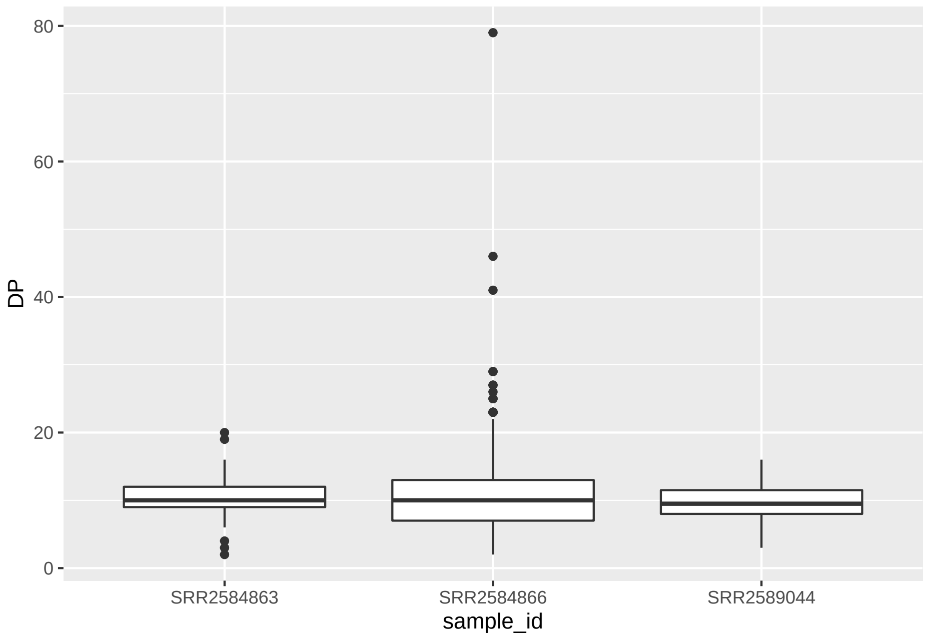
QUANTILES: The *pth* percentile of a data set sorted from smallest to largest is the value such that *p* percent of the data are at or below this value. The quartiles are special percentiles; the 1st quartile is the 25th percentile, and the 3rd quartile is the 75th percentile. The median is also a quartile – it is the 50th percentile.

Within these five numbers is a lot of useful data!

- the median gives a measure of the center of the data
- the minimum and maximum give the range of the data
- the 1st and 3rd quartiles give a sense of the spread of the data, especially when compared to the minimum, maximum, and median

To create a boxplot, use the `geom_boxplot()` function:

```
ggplot(data = variants, aes(x = sample_id, y = DP)) +
  geom_boxplot()
```



EXERCISE

1. Change the colour of each box in the above plot to match the sample_id colours in the plots above.
2. Log transform the y-axis.

Bonus challenge: Change the name of the legend to "Sample ID".

9.7 Themes

In addition to `theme_bw()`, which changes the plot background to white, `ggplot2` comes with several other themes which can be useful to quickly change the look of your visualization. The complete list of themes is available at https://ggplot2.tidyverse.org/reference/ggtheme.html. `theme_minimal()` and `theme_light()` are popular, and `theme_void()` can be useful as a starting point to create a new hand-crafted theme.

The `ggthemes` package provides a wide variety of options (including an Excel 2003 theme). The `ggplot2` extensions website provides a list of packages that extend the capabilities of `ggplot2`, including additional themes.

Chapter 10

Project

Introduction

The data we're going to work with comes from a yeast microarray experiment (Brauer, 2008) where the authors measured gene expression in 36 chemostat cultures where growth was limited by one of six nutrients (glucose, ammonium, sulfate, phosphate, uracil, or leucine) at 6 different growth rates.

The R world is full of niche tools designed for a specific problem or data type. This is particularly true for Bioconductor and genomics analysis. RNA-seq is a good example where all data import, analysis, and even plotting is done with custom packages and functions designed for gene expression data. Analysis of microbiome data with phyloseq is another example. Now this is not a bad thing and can enable simple, rapid analysis. Yet the point of learning a programming language like R is so you can take back control of your analysis and put the tools to work for your specific problem.

For this project we're going to take some genomic data and pull it out of the context that it's usually analyzed in. Bioconductor has a plethora of tools for analyzing microarray data and the `limma` package is by far the most comprehensive and useful. However, for our project we're going to take microarray data that has already been processed and instead use it with data analysis tools that we've been learning.

The take-away from this project to make the tools work for you instead of you working for them. Data, genomic or not, is still data and we can use the tools available to us in R to gain valuable insights.

Task 0: Download the data

You can find the data for download from dropbox here.

Task 1: Data import and examination

The first task of almost all analysis is to get the data into R and spend a little time understanding it's structure. Although you may be tempted to rush into an analysis it pays to take a bit of time to understand what you are working with and do some cleaning before you dive in too deep. This is important help you avoid costly mistakes and wasted time downstream.

The data file you'll be using is `Brauer2008_DataSet1.csv` and although it is pre-processed it will still need some work to get it ready to use.

Your goal is get the data loaded into R and answer the following questions:

1. How many rows and columns are there?
2. What do the rows and columns represent?
3. On first glimpse do the column datatypes look correct?
4. What about column names?
5. Is this data tidy? Why or why not?

As you look through the data start thinking about what steps you'll need to get it into a workable state.

Task 1 Solution

1. How many rows and columns are there?

```r
library(tidyverse)

raw <- read.csv("data/Brauer2008_DataSet1.csv")
#head(raw)
dim(raw)
```

```
## [1] 5537   40
```

There are 5537 rows and 40 columns.

2. What do the rows and columns represent?

Rows are genes and columns are a mix of different variables including gene names and expression values.

3. On first glimpse do the column datatypes look correct?

Yes, it seems that R has guessed correctly on what the values should be.

4. What about column names?

The columns that have the gene expression values don't look right. Something is up with that period in there.

5. Is this data tidy? Why or why not?

It's not. Each observation does not form a row and each variable does not form a column.

Task 2: Data Tidying

Never underestimate the amount of time and effort it will take to tame a wild dataset into something usable. Even if someone has already processed and "cleaned" the data before you (such as our current dataset) it is likely that you will need to spend some time fiddling with it for your particular needs (or for a particular function you want to use).

You've learned about the tidy data concepts, now it's time to put them into practice. The first and most important step is decide what, if any tidying needs to be done. Recall that:

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

Questions to answer:

1. Which column contains multiple variables? Split it up so each column represents a single variable.
2. The `GWEIGHT`, `GID`, and `YORF` columns we also won't need so you can get rid of those.
3. What's wrong with the column names? How can you fix that?
4. Check again: do you have any columns that have multiple variables? Be sure to fix any before you move on.

Try to code all the cleaning steps as a single pipeline. Remember the `separate()` and `gather()` functions from the tidyr package.

Bonus: Your cleaning steps may have left some variables with extra white space at the end. Using `dplyr::mutate_at()` and a certain function from the `stringr` package see if you can clean these up.

Task 2 Solution

1. Which column contains multiple variables? Split it up so each column represents a single variable.

We need to tell `separate()` to split on the `||` character using the `sep` argument. However, the `|` character is actually a special character for something called regular expressions, which we don't need to worry about right now. That's why we need to use the double backslash, which "escapes" the `|` character and tells `separate()` to interpret it like a normal character. If we used `sep = "||"` this would give us the wrong result.

```
clean <- raw %>%
  separate(NAME, c("gene_name", "BP", "MP", "gene_id", "number"), sep = "\\|\\|")
```

2. The `GWEIGHT`, `GID`, and `YORF` columns we also won't need so you can get rid of those.

That's easy with `select()`.

```
clean <- raw %>%
  separate(NAME, c("gene_name", "BP", "MP", "gene_id", "number"), sep = "\\|\\|") %>%
  select(-GWEIGHT, -GID, -YORF)
```

3. What's wrong with the column names? How can you fix that?

They're variables not names and tidy data principles tell us that each variable should be a column. And these are not just not just 1 variable - both nutrient and growth rate are present. We can fix this with `gather()`.

```
clean <- raw %>%
  separate(NAME, c("gene_name", "BP", "MP", "gene_id", "number"), sep = "\\|\\|") %>%
  select(-GWEIGHT, -GID, -YORF) %>%
  gather(variable, expression, G0.05:U0.3)
```

4. Check again: do you have any columns that have multipe variables? Be sure to fix any before you move on.

Why yes, it seems that our `variable` column is comprised of both nutrient and growth rate. We'll have to use `separate()` again. The trick here is how to specify the correct place to split the variable. We want the first letter in one column (nutrient) and the the remaining part in a second column (growth_rate). If you read the help for the `separate()` function you'll see that if you provide a number to the `sep` argument it will split at this position. This works great for us. There are other ways to split this variable, however, these require a bit more knowledge of regular expressions. And always go with the simplist soluion you can find.

```
clean <- raw %>%
  separate(NAME, c("gene_name", "BP", "MP", "gene_id", "number"), sep = "\\|\\|") %>%
  select(-GWEIGHT, -GID, -YORF) %>%
  gather(variable, expression, G0.05:U0.3) %>%
  separate(variable, c("nutrient", "growth_rate"), sep = 1)
```

```
head(clean$BP)
```

```
## [1] " ER to Golgi transport "
## [2] " biological process unknown "
## [3] " proteolysis and peptidolysis "
## [4] " mRNA polyadenylylation* "
## [5] " vesicle fusion* "
## [6] " biological process unknown "
```

Bonus

```
clean <- raw %>%
  separate(NAME, c("gene_name", "BP", "MP", "gene_id", "number"), sep = "\\|\\|") %>%
  select(-GWEIGHT, -GID, -YORF) %>%
  gather(variable, expression, G0.05:U0.3) %>%
  separate(variable, c("nutrient", "growth_rate"), sep = 1, convert = TRUE) %>%
  mutate_at(vars(gene_name:gene_id), str_trim)
head(clean)
```

```
##   gene_name                           BP
## 1      SFB2        ER to Golgi transport
## 2              biological process unknown
## 3      QRI7 proteolysis and peptidolysis
## 4      CFT2      mRNA polyadenylylation*
## 5      SSO2              vesicle fusion*
## 6      PSP2   biological process unknown
##                               MP gene_id
## 1    molecular function unknown YNL049C
## 2    molecular function unknown YNL095C
## 3 metalloendopeptidase activity YDL104C
## 4                  RNA binding YLR115W
## 5              t-SNARE activity YMR183C
## 6    molecular function unknown YML017W
##      number nutrient growth_rate expression
## 1  1082129        G        0.05      -0.24
## 2  1086222        G        0.05       0.28
## 3  1085955        G        0.05      -0.02
## 4  1081958        G        0.05      -0.33
## 5  1081214        G        0.05       0.05
## 6  1083036        G        0.05      -0.69
```

Task 3: Plots

Whew! Alright, that's done, but now you might asking why spend all that time cleaning up the data. Hopefully you should be able to answer that question by now but moving on to making some plots will help make it even more clear.

In a typical analysis you wouldn't necessarily know which gene(s) to start looking at and would want to start with some more exploratory analysis. However, to practice our plotting and analysis let's focus on a few pre-picked genes and pathways.

1. Filter your now cleaned data to contain only the gene LEU1.
2. Make a plot of the expression of this gene at various growth rates. Use colour to distinguish the different nutrients.
3. What are some conclusions you can make from this data?
4. What biological process is LEU1 classified as?
5. Filter your cleaned data for all genes that are in the same biological process as LEU!.
6. Make a plot of these genes, the same as you had for LEU1 but faceted by gene.
7. Do any of the other genes follow the same pattern as LEU1?

8. Customize your plot a bit by changing the colour palette to one of the RColorBrewer palettes. Explore some of the other themes available (`theme_bw()` is particularly nice). Fix the x-axis and y-axis labels so they look nicer.

Bonus: To increase readability of the plot it can helpful to change those nutrients from letters to full words. Although there are ways to do this directly in ggplot the easiest is to change it in the original data.

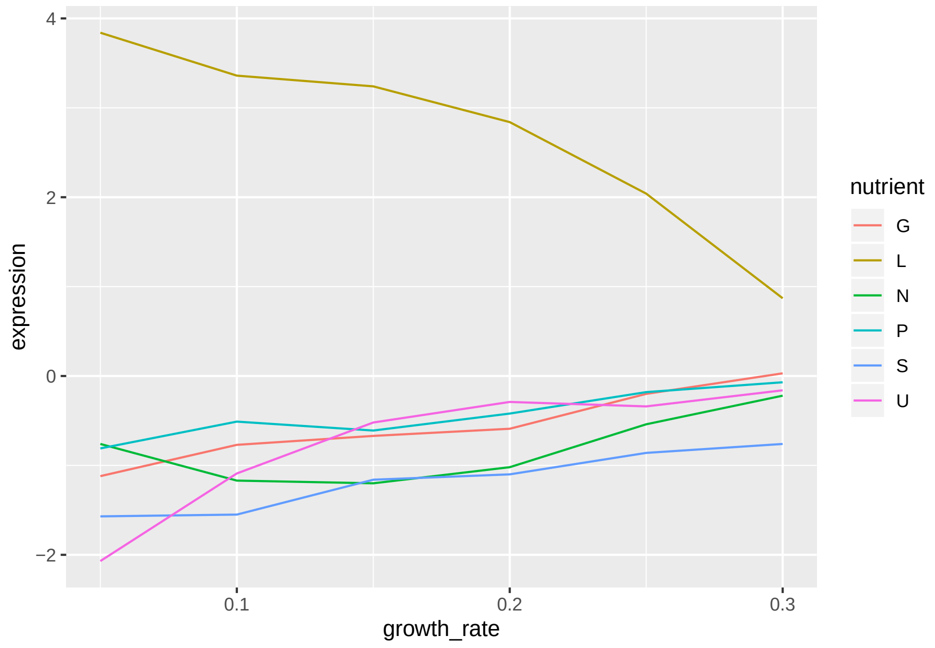
Task 3 Solution

1. Filter your now cleaned data to contain only the gene LEU1.

```
leu<- clean %>%
  filter(gene_name == "LEU1")
```

2. Make a plot of the expression of this gene at various growth rates. Use colour to distinguish the different nutrients.

```
clean %>%
  filter(gene_name == "LEU1") %>%
  ggplot(aes(x = growth_rate, y = expression)) +
    geom_line(aes(colour = nutrient))
```



3. What are some conclusions you can make from this data?

- When leucine is limiting the expression of LEU1 is higher than when any of the other nutrients are limiting.
- As growth rate increase the expresison of LEU1 decreases but only in leucine limiting conditions. In all other conditions it increases as growth rate increases.

4. What biological process is LEU1 classified as?

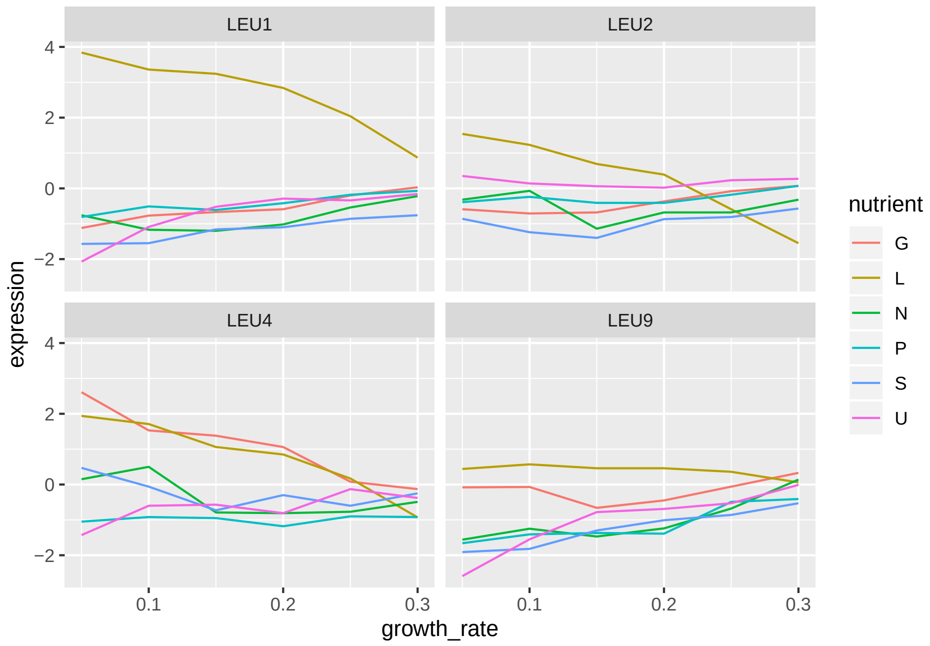
leucine biosynthesis

5. Filter your cleaned data for all genes that are in the same biological process as LEU1.

```
lbs<- clean %>%
  filter(BP == "leucine biosynthesis")
```

6. Make a plot of these genes, the same as you had for LEU1 but faceted by gene.

```
clean %>%
  filter(BP == "leucine biosynthesis") %>%
  ggplot(aes(x = growth_rate, y = expression)) +
    geom_line(aes(colour = nutrient)) +
    facet_wrap(~gene_name)
```

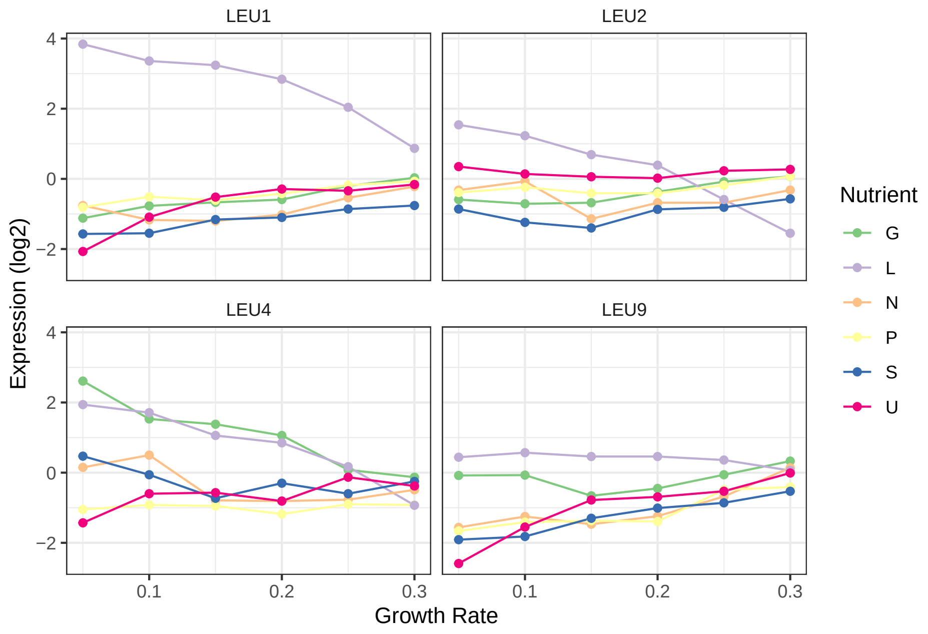


7. Do any of the other genes follow the same pattern as LEU1?

LEU2 seems to but the expression difference in leucine limiting conditions is not quite so high.

8. Make it pretty

```
clean %>%
  filter(BP == "leucine biosynthesis") %>%
  ggplot(aes(x = growth_rate, y = expression, colour = nutrient)) +
    geom_line() +
    geom_point() +
    facet_wrap(~gene_name) +
    scale_color_brewer("Nutrient", palette = "Accent") +
    theme_bw() +
    labs(x = "Growth Rate", y = "Expression (log2)") +
    theme(strip.background = element_blank())
```

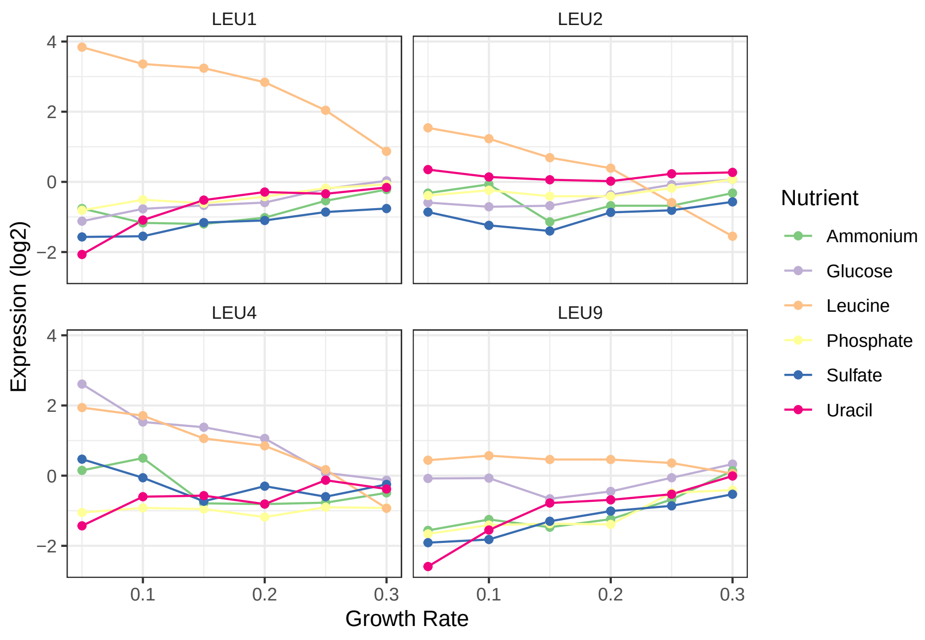


Bonus: Changing nutrient names

```
nutrient_key = c(
  G = "Glucose",
  N = "Ammonium",
  P = "Phosphate",
  S = "Sulfate",
  U = "Uracil",
  L = "Leucine"
)
nutrient_key["P"]
```

```
##           P
## "Phosphate"
```

```
clean %>%
  mutate(nutrient_names = nutrient_key[nutrient]) %>%
    filter(BP == "leucine biosynthesis") %>%
    ggplot(aes(x = growth_rate, y = expression, colour = nutrient_names)) +
    geom_line() +
    geom_point() +
    facet_wrap(~gene_name) +
    scale_color_brewer("Nutrient", palette = "Accent") +
    theme_bw() +
    labs(x = "Growth Rate", y = "Expression (log2)") +
    theme(strip.background = element_blank())
```



Task 4: Heatmap

Heatmaps are quite popular but generally useless visualisations used frequently in the genomic sciences. They can show a large amount of information in a small space, however, this very feature often renders them useless in terms of interpretability. Yet, for some applications they can be useful and let's face it, your supervisor propbably wants one so it's a good skill to have.

Heatmaps are very straight forward to make with ggplot using `geom_tile()` however, one of the more useful features of heatmap is the row and column ordering usually done with hierarchical clustering. This is non-trivial (but not impossible) to implement with ggplot so instead we're going to look at a package specifically designed for these types of heatmaps. There are as many heatmap packages for R as there are heatmaps and you may end up using different packages for different purposes. However, the `pheatmap` package has a nice balance of features, speed, and ease of use and is the one we'll work with today, although you are free to look at others for your own needs.

To this point much of your training has focused on the concepts of tidy data analysis, which are very powerful and you will come to rely on for much of your work. However, for genomics and other bioinformatics analysis you will often find yourself having to go between the *tidyverse* and other custom formats and datatypes specific to another package or the more traditional R datatypes, particularly the matrix format. In addition to learning how to make a nice heatmap the other main goal of this task is to help you become familar with moving back and forth between different data structures.

Questions to answer:

1. Start by installing the **pheatmap** package from CRAN. Have a look at the help page for the `pheatmap()` function. Which argument specifies the data? What format does it need?
2. Using what you learned from the help function get the gene expression data into a format that `pheatmap()` can use. Think about what you want your heatmap to look like - what are the columns and rows?
3. Once you've got data in the correct format start by making a simple heatmap with default parameters. Do you like it? Is there anything wrong with it? What would change from a data interpretation perspective? What about from aethetics perspective?

Task 4 Solution

1. Start by installing the **pheatmap** package from CRAN. Have a look at the help page for the `pheatamp()` function. Which argument specifies the data? What format does it need?

```
#install.packages("pheatmap")
library(pheatmap)
?pheatmap
```

Looks like `pheatmap()` needs a numeric matix given with the `mat` arguments, which is the first and only required arugment.

2. Using what you learned from the help function get the gene expression data into a format that `pheatmap()` can use. Think about what you want your heatmap to look like - what are the columns and rows?

Here we're going to go back to our pre-tidied data. This doesn't negate the work we did to tidy it nor does it mean the the data is now 'dirty' or bad somehow. Tidy data is standardized but many R tools still don't work with data in this format and so we must adapt.

```
#head(raw)
hm_mat <- raw %>%
  select(-GID, -YORF, -NAME, -GWEIGHT) %>%
  as.matrix()

class(hm_mat)
```

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

```
typeof(hm_mat)
```

```
## [1] "double"
```

3. Once you've got data in the correct format start by making a simple heatmap with default parameters. Do you like it? Is there anything wrong with it? What would change from a data interpretation perspective? What about from aethetics perspective?

```
pheatmap(hm_mat)
```

Overall this actually looks pretty good. Because these data have already been normalized and scaled we don't need to do much here but there are few tweaks that may help.

- The default is to order both the rows and columns, however, because this is a timecourse experiment we may not want to order the columns. In fact if we look at the raw data we see that the columns are already ordered by time.
- The colours are better than the typical red/green (which you should NEVER use, BTW) but still leave something to be desired. This could be changed in a final version of the plot. Note that we have a diverging colour scale with values ranging from -6 to +6. This is becuase these data are mean-centred. Blue (negative) values are below the mean for that gene and red values (positive) are above the mean for that gene. Yellow is the mean expression for that gene. Mean-centering is a handy way to show fold-changes, but should always be displayed with a diverging colour scale.
- To help with visualizing the different nutrients, we can add gaps to the heatmap to make a column for each nutrient.

```
library(RColorBrewer)
pheatmap(hm_mat,
         cluster_cols = FALSE,
         color = colorRampPalette(rev(brewer.pal(n = 7, name = "RdBu")))(100),
         breaks = seq(-5, 5, length.out = 101),
         gaps_col = c(6,12,18,24,30)
)
```

Chapter 11

Summary