R Crash Course for Biologists

Robert I. Colautti

2021-12-05

Contents

Preface

This book is targeted at students and professionals in the biological sciences who wish to develop coding skills for the analysis of biological data using R. The content is based on my teaching experience at the senior undergraduate and early graduate level for a range of topics in Biology and Health Sciences at Queen's University – Environmental Science, Epidemiology, Genomics, Ecology, and Evolution. Many biology students do not receive strong quantitative skills training in math, statistics, or computer science but later realize how valuable these skills can be for investigating biological phenomena. This book tries to demystify mathematical equations and computational algorithms using a hands-on tuturial approach.

Professionals who have little to no coding and quantitative skills may also find this book helpful, as it assumes only minimal training in math and statistics.

To help demonstrate the tremendous value of coding and quantitative skills, I focus on examples drawn from real biological studies. This provides real-world examples of how one can apply programming tools and techniques to curate, analyze, and visualize biological data. These are the areas in which I have researched and published papers – opportunities that were presented to me because of my ability to analyze data in a reproducible and open framework. Thee tools presented here are the very same tools that I use on a regular basis for rigorous, peer-reviewed analysis. A few examples of our datasets, analyses and visualizations using R include:

1. A *Science* paper examining rapid evolution of flowering time: https://doi.org/10.1126/science.1242121
2. A *de novo* genome assembly: https://doi.org/10.1093/g3journal/jkab339
3. A Meta-analysis of evolution of invasive species: https://doi.org/10.1111/mec.13162
4. Tracking COVID-19 ourbreaks using whole-genome sequencing: https://doi.org/10.1038/s41598-021-83355-1

In choosing the content for this book, I tried to focus on everything that I wish I knew when I first started learning to program in R. Many of the functions and packages included here were not available when I started, but have some exceptional functionality. I continue to add new tricks and techniques that I find useful.

0.1 Advice

If this is your first attempt to learn how to code, then it's important to understand HOW to learn to code. It's crucial to understand that you won't learn to code just by reading this text and the examples. You need to take control of your education and actively participate with the examples in this book. Consider that R is a programming *language*...

Question: How did you become fluent in a second language?

- Immerse yourself
- Study, read, listen
- Try something new, fail, correct errors, repeat
- Practice, practice, practice!

Question: How do you become fluent in a programming language?

- Immerse yourself
- Study, read, TYPE!
- Try something new, fail, correct errors, repeat
- Practice, practice, practice!

Here are a few specific tips to become fluent in R:

1. Get organized and PLAN. Plan which pages/lessons you want to focus on for the day/week/month/semester.
2. As your skills develop, you will start to develop a toolbox of coding techniques. Look for opportunities to apply them. Take time to plan out a course of action to help you think about what coding tools you can apply.
3. Set aside large blocks of time (2+ hours), to **immerse yourself** in your coding lessons or project. Turn off your phone, turn off your wifi, and find some place you can work without interruption.
4. Get some good headphones with white noise or instrumental sounds (no lyrics), depending on mood:

```
a. Baroque/Classical
b. Smooth Jazz
c. Electronic (ambient, house, lofi)
e. https://coffitivity.com/
```

4. If you get stuck, Google: "How do I _________ in R". Look for answers from Stack Overflow
5. If you can't figure out what an error means, paste it into Google. Again, look for answers from Stack Overflow
6. Read other people's code carefully to see how they tackle problems. Rarely is there one single 'right' way to code something.
7. When you are starting out, the 'right' way to code is whatever it takes to get the code to do what you want.
8. As you advance, look for ways to do the same thing faster and with fewer lines of code.
9. **EMBRACE FAILURE!** – even after 10+ years of programming experience, most of my algorithms do not work on the first try, and most of my time is spent dealing with error messages and unexpected output.
10. **Read** the documentation for the function or package you are using. Often the repository on R-CRAN.org or Bioconductor will have not only the official documentation but also 'vignettes' or tutorials to guide you through its details.

0.2 Learn By Doing!

As you work through these self-tutorials, don't just read them. I can't stress this enough: type them out in your R (Studio) console and see what the output looks like. The simple act of typing it out will send messages to your brain telling it that this is an important thing to remember and store in there somewhere. If you get an error, even better! Read the error carefully, then compare what you typed to what is in the tutorial. Once you find what is different, you will learn what that error means.

About 70-90% of coding is dealing with errors, and the same is true for learning to code.

In my experience there are three main steps to coding competence.

1. **Utter bewilderment** – reading code is like reading a foreign language. All these letters and symbols are meaningless to you.
2. **Understanding** – you can look at a function and have a decent idea of what it does and how to use it.
3. **Competence** – you can write your own code from scratch, without needing to look up examples.

Don't confuse *understanding* with *competence* – this is a common mistake that students make. It's relatively easy to learn how to understand code that is shown to you, but it's quite another skill to learn the names and parameters of useful functions and apply them in the appropriate circumstances. That doesn't mean you need to memorize every function – though memorization can help. A good strategy for move from understanding to competence is to make the extra time and effort to type out the code that is shown to you, even when you can look at it and understand what it does. Again, the act of typing out the code is what will help to solidify it in your brain.

Chapter 1

Setup

1.1 R

Before you begin these tutorials, you should install the latest version of R: https://cran.r-project.org/

Versions are available for Windows, MacOS and Linux operating systems. Immediately we can see one of the advantages of learning to code in R – we can move code across computing platforms quite easily, as long as R is installed there.

1.2 R Studio

You should also install R Studio: https://rstudio.com/products/rstudio/download/#download

R Studio is an **Integrated Development Environment (IDE). Once you install and open R Studio, run the program for the first time.

You will see several helpful *tabs* – small windows. Several windows have more than one tab at the top, which you can click on to access.

1.2.1 Console

One of the most important tabs is the console. It's usually the main tab that opens when you open R Studio. You'll see a little chevron `>` with a cursor after it. That's the R Console and it's the part that actually runs the R program. The other tabs help with other things, like keeping track of your files, a place to show graphs that you make, a **help** tab to look for information on particular functions, and several other useful tabs that you can read about in the R Studio documentation.

1.2.2 R Script

To run an R script, you can just type functions into the console. However, it is very hard to keep track of everything you do if you only use the console. In R Studio you can click **File–>New File–>R Script**. This will open a new tab window called **Untitled**. This is called a **script**, but it's really just a text file, with a **.R** suffix, that you can use to keep track of your R program. Try typing something into your R script – don't worry for now if it is just some random text. Note that you can **Save** this file.

Nothing happens (yet). To run the script, you have to send the text from the script tab to the console tab. There are a few ways you could do this:

1. Copy and paste manually. This works fine, but there are more effeicient options
2. Highlight the code you want to run and click the **Run** button on the top-right corner of the script tab. The run button sends the highlighted text from the script to the console.
3. If you click the **Run** button without highlighting text, it will send whatever text is on the same *line* as your cursor
4. If you press **Ctl + R** it will do the same thing – this is the shortcut for the **Run** button
5. There are other options if you press the tiny triangle next to the **Run** button, including **Run All**. This is the equivalent of running one line at a time.
6. **Ctl + Shift + R** is a shortcut for **Run All**

1.3 Packages

Packages in R contain functions – small programs that you can run. One really good package is called `tidyverse`. The `tidyverse` package contains a lot of useful functions for working with different types of data, including visualizations. You'll need to make sure you are connected to the internet and that your connection to the internet won't be interrupted during the download.

> WARNING! This may take a long time to run

To install the packages, open R Studio and look for the **Console** tab.

Type this into your console:

```r
install.packages("tidyverse")
```

Chapter 2

Fundamentals

2.1 Introduction

This chapter provides a rapid breakdown of the core functionality of R. There is a lot to cover in a very short time. Remember that you can only learn coding through repetition. Make the extra time and effort to actually type out the code and run it in your console.

It is important that you physically participate and code along with the examples. Type everything out. The physical act of typing into R and troubleshooting any errors you get is a crucial part of the learning process.

2.2 R Basics

Make comments inside your code. Very important (unless you are using R markdown or R notebooks)!

```
# Use hastags to make comments - not read by the R console
# Use other characters and blank lines to improve readability:
# -------------------------
# My first R script
# Today's Date
# -------------------------
# Add a summary description of what the script does
# This script will...
# And annotate individual parts of the script
```

2.2.1 Basic Math

You can do basic mathematical equations in R.

> Yes, type these out!

```
10+2 # add
```

```
## [1] 12
```

```
10-2 # subtract
```

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

```
10*2 # multiply
```

```
## [1] 20
```

```
10/2 # divide
```

```
## [1] 5
```

```
10^2 # exponent
```

```
## [1] 100
```

2.2.2 Functions

R uses **functions**. Each function has a name and is followed by brackets `function()`.

Inside the brackets we can define input values and parameters.

2.2.3 ‘c()

The **concatenate** function `c()` is a very important function in R that is used to group items together.

```
c(1,2,3,5)
```

```
## [1] 1 2 3 5
```

2.2.4 Math Functions

Here are some basic mathematical functions:

```
abs(-10) # absolute value
```

```
## [1] 10
```

```
sqrt(10-1) # square root (with subtraction)
```

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

```
log(10) # natural log
```

```
## [1] 2.302585
```

```
log10(10) # log base 10
```

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

```
exp(1) # power of e
```

```
## [1] 2.718282
```

```
sin(pi/2) # sine function
```

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

```
asin(1) # inverse sine
```

```
## [1] 1.570796
```

```
cos(pi) # cosine
```

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

```
acos(-1) # inverse cosine
```

```
## [1] 3.141593
```

```
tan(0) # tangent
```

```
## [1] 0
```

```
atan(0) # inverse tangent
```

```
## [1] 0
```

2.2.5 Round/Truncate

Rounding and truncating numbers:

```
round(pi,digits=3) # standard rounding to 3 digits
```

```
## [1] 3.142
```

```
floor(pi) # round down to closest whole number
```

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

```
ceiling(pi) # round up to closest whole number
```

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

```
signif(pi,digits=2) # round to keep 2 significant digits
```

```
## [1] 3.1
```

2.2.6 Logic Operators

An **operator** is used to compare values.

```
1 > 2 # greater than
```

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

```r
1 < 2 # less than
```

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

```r
1 <= 2 # less than or equal to
```

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

```r
1 == 1 # equal to
```

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

```r
1 == 2 | 1 == 1 # | means 'OR'
```

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

```r
1 == 2 & 1 == 1 # & means 'AND'
```

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

```r
1 == 1 & 1 == 1
```

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

Note: `!` is a negation/inverse operator

```r
1 != 1 # not equal to
```

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

2.2.7 Group Comparison

Instead of `|`, you can us `%in%` with `c()` to check a large number of values.

```r
1 %in% c(1,2,3,4,5,6,7,8,9,10)
```

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

2.2.8 Random Numbers

It is very easy to generate some random numbers from different distributions. This is very useful for modelling and testing your code. These are covered in more detail in the Distributions Tutorial. But for now, it's just useful to know how to generate different kinds of random numbers.

The most basic random number is a whole number (i.e. no decimal) drawn from a **uniform distribution**, meaning that each number has an equal probability of being selected.

```
runif(n=10, min=0, max=1)
```

```
##  [1] 0.9885135 0.8733742 0.4665132 0.3481788 0.3506248 0.9238958 0.2820634
##  [8] 0.5946945 0.6089696 0.7636489
```

One of the most common random numbers in statistics is a number drawn from a **random, normal distribution** with a given `mean` and `sd` (standard deviation). Rational numbers (i.e. with decimal) can be chosen and numbers closer to the mean are more likely to be chosen.

```
rnorm(10, mean=0, sd=1)
```

```
##  [1]  0.7389776 -1.4592149 -0.8139667 -0.4287346  1.7775109 -0.7333748
##  [7] -0.3866044 -0.4916229  0.4920625  1.2680194
```

Note that som

A **poisson distribution** includes only whole numbers but are include a parameter `lambda`, which is the equivalent of the mean in the normal distribution.

```
rpois(10, lambda=10)
```

```
##  [1] 14 11 15 13 15 10 13  3 12 10
```

The **binomial distribution** is useful for binary outcomes, like a coin toss, coded as 0 or 1. The `size` parameter is the number of events (e.g. number of coin flips), and the `prob` parameter is the probability of getting a 1.

```
rbinom(10,size=1,prob=0.5)
```

```
##  [1] 1 1 0 0 0 1 1 0 1 1
```

```r
rbinom(10,size=10,prob=0.5)
```

```
##  [1] 5 6 6 5 2 5 6 5 5 6
```

Fun fact: random numbers generated by a computer are not truly random. Instead, the numbers involve a calculation from a starting number called a **seed**. The seed might be the current Year/Month/Day/Hour/Minute/Second/Millisecond, which means the 'random' number could be determined by somebody who knows the equation and the time it eas executed.

In practice, computer-generated random numbers are much better than human-generated random numbers.

We can also set the seed number to help with testing and debugging our code.

We can recreate the exact random number using the `set.seed()` function.

Compare these outputs:

```r
runif(5)
```

```
## [1] 0.99911815 0.05911907 0.34001244 0.04761804 0.30843737
```

```r
runif(5)
```

```
## [1] 0.7334727 0.5379420 0.7196811 0.7018300 0.9512855
```

```r
set.seed(3)
runif(5)
```

```
## [1] 0.1680415 0.8075164 0.3849424 0.3277343 0.6021007
```

```r
set.seed(3)
runif(5)
```

```
## [1] 0.1680415 0.8075164 0.3849424 0.3277343 0.6021007
```

```r
set.seed(172834782)
runif(5)
```

```
## [1] 0.13729290 0.18587365 0.01860484 0.88440060 0.21414154
```

```
set.seed(172834782)
runif(5)
```

```
## [1] 0.13729290 0.18587365 0.01860484 0.88440060 0.21414154
```

```
runif(5)
```

```
## [1] 0.19787402 0.84870074 0.27303904 0.12225215 0.08365613
```

2.2.9 Combining objects

Use `c()` to concatenate single objects.

```
Nums<-c(1,2,5)
c(Nums,"string")
```

```
## [1] "1"      "2"      "5"      "string"
```

Use `:` to include a range of numbers.

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
100:90
```

```
##  [1] 100  99  98  97  96  95  94  93  92  91  90
```

```
-1:1
```

```
## [1] -1  0  1
```

Use `cbind()` to bind columns and `rbind` to bind rows.

```
cbind(1:10,10:1)
```

```
##       [,1] [,2]
##  [1,]    1   10
##  [2,]    2    9
##  [3,]    3    8
##  [4,]    4    7
##  [5,]    5    6
##  [6,]    6    5
##  [7,]    7    4
##  [8,]    8    3
##  [9,]    9    2
## [10,]   10    1
```

```
rbind(1:10,10:1)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]   10    9    8    7    6    5    4    3    2     1
```

2.3 Data Types

Programming languages like R use different data types.

It's very important to understand data types in order to properly encode and analyze data in R. It's very common to have errors in statistical analyses caused by the wrong kind of data. For example, if you have 3 experimental groups coded as 1, 2 and 3 then these should be coded and analyzed as **factors** not **numeric** variables.

Type	Example	Description
`string`	`"String"`	Strings are the most common and versatile and can be defined with single `''` or double `""` quotation marks. The downside of strings is that you can't do any kind of equations.
`numeric`	`12.421`	Numeric variables are rational numbers.
`integer`	`12`	Integers are whole numbers and may be positive or negative (no decimal).
`complex`	`0+12.43i`	Complex numbers include real and imaginary numbers.
`boolean`	`T` or `TRUE`	Boolean or **logical** variables are either true or false (Note always capital).

`factors` Factors are a special type of data that may include strings and/or numbers but have a limited number of classes. Factors are often used to code groups in statistical models.

Note that computers cannot store irrational numbers, they have to be rounded to some (tiny) decimal place.

2.4 Objects

R supports **Object-Oriented Programming (OOP)**, which is a programming style that defines and manipulates **objects**

An **object** in R can be a lot of things, but an easy way is to think of a spreadsheet (example Microsoft Excel).

A spreadsheet has columns organized into rows and columns, and may have multiple sheets.

2.4.1 Cells

The most basic object is a single value. For example, a string:

```
X<-"string"
```

> Why no output?

When we wrote: `X<-"string"`

R created the object called **X**, so no output is produced.

There are a few options To see the contents of **X**:

```
print(X)
```

```
## [1] "string"
```

```
paste(X)
```

```
## [1] "string"
```

```
X
```

```
## [1] "string"
```

`print()` Is most generic and versatile for providing feedback while running complex scripts (e.g. during loops, Bash scripts, etc)

`paste()` Converts objects to a string, we'll come back to this.

Generally `print()` or `paste()` are preferred over calling the object directly.

2.4.2 Vector

A vector is a one-dimensional array of cells. This could be a row or column in our spreadsheet example.

Each cell within the vector has an 'address' – a number corresponding to the cell ranging from 1 to N, where N is the number of cells.

The number of cells in a vector is called the **length** of the vector.

All items in a vector must be of the same type. If you mix numbers and text, then the whole vector will be formatted to the simplest type. For example, if you include a string with any other fomat, then the whole vector will be treated as a string:

```
Xvec<-c(1.1829378, X, 1:10, "E", "Computational Biology", 100:90)
Xvec
```

```
##  [1] "1.1829378"             "string"                "1"
##  [4] "2"                     "3"                     "4"
##  [7] "5"                     "6"                     "7"
## [10] "8"                     "9"                     "10"
## [13] "E"                     "Computational Biology" "100"
## [16] "99"                    "98"                    "97"
## [19] "96"                    "95"                    "94"
## [22] "93"                    "92"                    "91"
## [25] "90"
```

> **Protip**: A common problem when importing data to R occurs when a column of numeric data includes at least one text value (e.g. "missing" or "< 1"). R will treat the entire column as text rather than numeric values. Watch for this when working with real data!

2.4.2.1 Subset a vector

Use square brackets `[]` to subset a vector.

```
Xvec[1]
```

```
## [1] "1.1829378"
```

```
Xvec[13]
```

```
## [1] "E"
```

```
Xvec[1:3]
```

```
## [1] "1.1829378" "string"    "1"
```

2.4.3 Matrices

A matrix is a 2-D array of cells, equivalent to one sheet in a spreadsheet program.

```
Xmat<-matrix(Xvec,nrow=6)
```

```
## Warning in matrix(Xvec, nrow = 6): data length [25] is not a sub-multiple or
## multiple of the number of rows [6]
```

```
Xmat
```

```
##      [,1]        [,2] [,3]                    [,4] [,5]
## [1,] "1.1829378" "5"  "E"                     "96" "90"
## [2,] "string"    "6"  "Computational Biology" "95" "1.1829378"
## [3,] "1"         "7"  "100"                   "94" "string"
## [4,] "2"         "8"  "99"                    "93" "1"
## [5,] "3"         "9"  "98"                    "92" "2"
## [6,] "4"         "10" "97"                    "91" "3"
```

2.4.3.1 Subset

Notice the square brackets along the top and left side?

These show the 'address' of each element in the matrix. We can subset with square brackets, just like we did with vectors. Since there are two dimensions, we need to specify two numbers:

`[row,column]`

```
Xmat[1,3]
```

```
## [1] "E"
```

Or leave it blank if you want the whole row or column:

```
Xmat[1,]
```

```
## [1] "1.1829378" "5"         "E"         "96"        "90"
```

```
Xmat[,3]
```

```
## [1] "E"                      "Computational Biology" "100"
## [4] "99"                     "98"                    "97"
```

2.4.4 Tensors

Tensors are the general term for a grid with N dimensions. We've already seen a few different tensors:

Name	Tensor Dimension
Cell	0
Vector	1
Matrix	2
Array	3+

Another common term for tensor is **array**. In R you can build tensors by adding as many dimensions as you need using the `array()` function.

```
Xarray<-array(0, dim=c(3,3,2)) # 3 dimensions
Xarray
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

Notice how 3rd dimension is sliced to print out in 2D.

Higher-order arrays are possible, but a bit tricky to read on a 2-dimensional screen.

```
Xarray<-array(rnorm(64), dim=c(2,2,2,2,2,2)) # 6 dimensions
```

Once you get the hang of it, it's easy to subset. Just think of each dimension, separated by commas.

```
Xarray[1:2,1:2,1,1,1,1]
```

```
##           [,1]       [,2]
## [1,] -1.718987  0.3487603
## [2,]  1.779268 -0.3523615
```

```
Xarray[1:2,1,1,1:2,1,1]
```

```
##           [,1]      [,2]
## [1,] -1.718987 0.8664164
## [2,]  1.779268 1.2394975
```

> Why are these numbers not the same?

Look at the `array[]` function and compare to the 6-D tensor to understand how this works.

2.5 Matrix Algebra

R is pretty handy for matrix calculations that would be very time consuming to do by hand or even in a spreadsheet program.

As an example, let's create some numeric vectors that we can play with:

```
X<-c(1:10)
X
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
Y<-c(1:10*0.5)
Y
```

```
##  [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

2.5.1 Basic operations

Probably the most common calculation is just to go through each element in each vector and multiply them together. For example, if X is a vector of leaf length measurements and Y is a vector of leaf width measurements, then we might want to calculate leaf area by multiplying each length by its corresponding width.

In R we just use the standard multiplication operator `*` on a vector, just like we would do for two individual numbers.

```
X*Y
```

```
##  [1]  0.5  2.0  4.5  8.0 12.5 18.0 24.5 32.0 40.5 50.0
```

Addition, subtraction, division, and exponents are similar.

```
X+Y
```

```
##  [1]  1.5  3.0  4.5  6.0  7.5  9.0 10.5 12.0 13.5 15.0
```

```
X/Y
```

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

```
X^Y
```

```
##  [1] 1.000000e+00 2.000000e+00 5.196152e+00 1.600000e+01 5.590170e+01
##  [6] 2.160000e+02 9.074927e+02 4.096000e+03 1.968300e+04 1.000000e+05
```

Just as we apply operators to vectors, we can also apply functions to vectors. When we do this, the same function is applied to each individual cell of the vector.

```
log(X)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

```
exp(Y)
```

```
##  [1]   1.648721   2.718282   4.481689   7.389056  12.182494  20.085537
##  [7]  33.115452  54.598150  90.017131 148.413159
```

2.5.2 Matrix Algebra

In matrix algebra, we can ‘multiply’ two vectors together in a different way. First, we multiply each pair of cells together, as above, but then we sum the products together (e.g. `X[1]*Y[1]+X[2]*Y[2]...`).

This is matrix multiplication operator `%*%` in R.

```
X%*%Y # Matrix multiplication
```

```
##       [,1]
## [1,] 192.5
```

```
sum(X*Y) == X%*%Y
```

```
##      [,1]
## [1,] TRUE
```

There are a few other important matrix operations:

```
Z<-X[1:4]%o%Y[1:3] # Outer product
Z
```

```
##      [,1] [,2] [,3]
## [1,]  0.5    1  1.5
## [2,]  1.0    2  3.0
## [3,]  1.5    3  4.5
## [4,]  2.0    4  6.0
```

```
t(Z) # Transpose
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  0.5    1  1.5    2
## [2,]  1.0    2  3.0    4
## [3,]  1.5    3  4.5    6
```

```
crossprod(X[1:4],Z) # Cross product
```

```
##      [,1] [,2] [,3]
## [1,]   15   30   45
```

```
crossprod(Z) # Cross product of Z and t(Z)
```

```
##      [,1] [,2] [,3]
## [1,]  7.5   15 22.5
## [2,] 15.0   30 45.0
## [3,] 22.5   45 67.5
```

```
diag(4) # Identity matrix, 4x4 in this case
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```
diag(Z) # Diagonal elements of Z
```

```
## [1] 0.5 2.0 4.5
```

These calculations can get a bit tricky – especially when we move to 2D matrices instead of vectors. You'll want to consult or review a matrix algebra textbook if you are going to apply these. For now, the important thing is just to know that these options are available if you need them in the future.

Operator	Name
`*`	Pairwise multiplicaiton
`%*%`	Matrix multiplication
`%o%`	Outer product
`t()`	Transpose
`crossprod()`	Cross-product
`diag(4)`	Identity of 4x4 matrix
`diag(M)`	Diagonal elements of matrix M

2.5.3 PCA

One popular use-case for matrix calculation is the principal components analysis (PCA). The PCA is covered in more detail in the PCA Tutorial, but it is a method to rescale a bunch of **correlated** vectors (e.g. measurements) so that they can be remapped to an equal number of **independent** PC axes.

PCA is widely used in biology, from community ecology and metagenomics to gene expression and morphometrics. It also has many applications outside of biology. For now, just know that it is easy to run a PCA using the `prcomp()` function. In most cases, we would want to scale the vectors to have a mean of 0 and standard deviation of 1. Equivalently, we can use the `cor=T` parameter to use the correlation matrix in the calculations.

```
prcomp(Z, cor=T)
```

```
## Warning: In prcomp.default(Z, cor = T) :
##  extra argument 'cor' will be disregarded
```

```
## Standard deviations (1, .., p=3):
## [1] 2.415229 0.000000 0.000000
##
## Rotation (n x k) = (3 x 3):
##            PC1        PC2        PC3
## [1,] 0.2672612  0.0000000  0.9636241
## [2,] 0.5345225 -0.8320503 -0.1482499
## [3,] 0.8017837  0.5547002 -0.2223748
```

2.6 Lists

Tensors generally all have the same data type and sub-dimension. For example, if you want to combine two separate 2D matrices into a single tensor (3rd dimension) then the individual matrices have to have the same number of rows and columns, and the same data type.

But often we want to group different types of information together. Think of a record in a database where you may have information about an individual's hight, weight, eye colour, adn maybe a photograph and a set of DNA sequences. This wouldn't fit neatly into a tensor format. Instead, we can use a list.

Lists are useful for mixing data types, and can combine different dimensions cells, vectors, and higher-order arrays.

Each element needs a name:

```
MyList<-list(name="SWC",potpourri=Xvec,numbers=1:10)
MyList
```

```
## $name
## [1] "SWC"
##
## $potpourri
##  [1] "1.1829378"            "string"               "1"
##  [4] "2"                    "3"                    "4"
##  [7] "5"                    "6"                    "7"
## [10] "8"                    "9"                    "10"
## [13] "E"                    "Computational Biology" "100"
## [16] "99"                   "98"                   "97"
## [19] "96"                   "95"                   "94"
## [22] "93"                   "92"                   "91"
## [25] "90"
##
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Important: Many of the statistical functions and other tools in R use list objects to store output. Taking some time now to understand how lists work will help you save a lot of time interpreting statistical output in R.

2.6.1 Subset

There are a few different ways to subset a list object.

```
MyList$numbers # Use $ to subset by name
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
MyList[3] # A 'slice' of MyList
```

```
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
MyList[[3]] # An 'extract' of MyList
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

What's the difference between `[]` and `[[]]`?

Look carefully at the output above; notice how the `[]` includes `$numbers` but the `[[]]` includes only the values? This is important if you want to use the slice:

```
2*MyList[[3]]
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
2*MyList[3]
```

```
## Error in 2 * MyList[3]: non-numeric argument to binary operator
```

2.6.2 Output

> **Protip**: Many analysis functions in R output as lists (e.g. statistical packages).

For example, the output of `prcomp`:

```
prcomp(Z)
```

```
## Standard deviations (1, .., p=3):
## [1] 2.415229 0.000000 0.000000
##
## Rotation (n x k) = (3 x 3):
##            PC1        PC2        PC3
## [1,] 0.2672612  0.0000000  0.9636241
## [2,] 0.5345225 -0.8320503 -0.1482499
## [3,] 0.8017837  0.5547002 -0.2223748
```

```
names(prcomp(Z))
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

```
prcomp(Z)$center
```

```
## [1] 1.25 2.50 3.75
```

```
prcomp(Z)$scale
```

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

2.7 print() and paste()

The `print` function is the go-to function for printing output to the user. The `paste` function is for combining things together.

Paste is a versatile function for manipulating output:

```r
paste("Hello World!") # Basic string
```

```
## [1] "Hello World!"
```

```r
paste("Hello","World!") # Concatenate two strings
```

```
## [1] "Hello World!"
```

Sometimes we need to convert numbers to strings. `paste` is an easy way to do this:

```r
paste(1:10) # Paste numbers as strings
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

```r
paste(1:10)[4]
```

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

Note above how each number is a separate cell in a vector of strings.

Use `as.numeric` to convert strings back to numbers.

```r
as.numeric(paste(1:10)) # Convert back to numbers
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Finally, use the `collapse` parameter to condense a multi-cell vector into a single cell.

```r
paste(1:10,collapse=".")
```

```
## [1] "1.2.3.4.5.6.7.8.9.10"
```

Note what happens if we combine objects of different length?

```
paste("Hello",1:10,sep="-")
```

```
##  [1] "Hello-1"  "Hello-2"  "Hello-3"  "Hello-4"  "Hello-5"  "Hello-6"
##  [7] "Hello-7"  "Hello-8"  "Hello-9"  "Hello-10"
```

It is not uncommon to nest a paste function within a print function when communicating output in a more complex R script.

```
print(paste("Hello",1:10,sep="-"))
```

```
##  [1] "Hello-1"  "Hello-2"  "Hello-3"  "Hello-4"  "Hello-5"  "Hello-6"
##  [7] "Hello-7"  "Hello-8"  "Hello-9"  "Hello-10"
```

This would be useful inside of a `for` loop (see below) where the output of `paste` is not shown on the screen if used inside of a loop, whereas the output of `print` is.

2.7.1 ? for HELP

Whenever you are learning a new function, you should use `?` and carefully read about all the parameters and outputs. The explanations can be a bit technical, which is intimidating at first. But after enough practice you will start to understand more and more of the descriptions.

```
?paste
```

2.8 Data

So far we've done everything within the R environment. If we quit R then everything we have made will be removed from memory and we'll have to start all over. Therefore, it can be useful to save and load data from external files.

2.8.1 Working Directory

The **working directory** is the place where R looks to load or save your files. You can figure out what your current working directory is with the `getwd()` function.

```
getwd()
```

Or you can set a specific working directory. Here's one example:

```
setwd("C:/Users/ColauttiLab/Documents")
```

Did you type out the above line? You should! Remember, going through and typing everything out is one of the most effective ways to learn to code. So do it now

Okay, so you probably have an error unless you are working in Windows and for some reason have a ColauttiLab username on your computer. Now try changing to a different directory on your computer.

If you are a mac user, you can just ignore the `C:` part:

```
setwd("/Users/ColauttiLab/Documents")
```

2.8.1.1 Relative Path

The above examples of `setwd()` us an **absolute** path. You can also use a **relative** path. For example, if we have a folder called `Data` inside our `Documents` folder, and our current working directory is one of the two examples above, we can use a relative path name to set the `Data` folder as the working directory. Before you type this out, you should make a folder called `Data` inside of your current working directory.

```
setwd("./Data")
```

The `.` means "Inside of my current directory" and the `/Data` means "find the Data folder".

now try this:

```
getwd()
setwd("..")
getwd()
```

Compare the working directories. The `..` means "Go to the parent directory."

The neat thing about relative directories is that it makes it easy to share code between Windows, MacOS and Linux/Unix. In fact, these commands come from Linux. You can check out the Linux Crash Course for more detail.

2.8.2 Import

Download This Data File and save in a folder called `Data` inside of your current working directory.

We can use the `read.csv` to read a 'comma-delimited file' and import it into an object called `MyData`. Often we have column names as the first row, so we include the parameter `header=T` to convert the first row to column names.

Data without column names would have data on the first row, so we would want `header=F` or else our first row of data would be treated as column names.

```
MyData<-read.csv("Data/FallopiaData.csv",header=T)
```

A `.csv` file is just a text file with special formatting that can be read into a program like Microsoft Excel or R to translate the text file into a data matrix.

Important: In R, objects created by `read.csv` and other `read.?` functions are special objects called `data.frame` objects.

2.8.3 `data.frame`

A `data.frame` is a special type of 2D matrix with additional indexing information for rows/columns of data.

This format is partly why R is so useful for data analysis.

There are a number of useful functions for inspecting a `data.frame` object.

```
names(MyData) # See column names
```

```
##  [1] "PotNum"       "Scenario"     "Nutrients"    "Taxon"        "Symphytum"
##  [6] "Silene"       "Urtica"       "Geranium"     "Geum"         "All_Natives"
## [11] "Fallopia"     "Total"        "Pct_Fallopia"
```

```
head(MyData) #  Show first six rows of data
```

```
##   PotNum Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 1      1      low       low japon      9.81  36.36  16.08     4.68 0.12
## 2      2      low       low japon      8.64  29.65   5.59     5.75 0.55
## 3      3      low       low japon      2.65  36.03  17.09     5.13 0.09
## 4      5      low       low japon      1.44  21.43  12.39     5.37 0.31
## 5      6      low       low japon      9.15  23.90   5.19     0.00 0.17
## 6      7      low       low japon      6.31  24.40   7.00     9.05 0.97
##   All_Natives Fallopia Total Pct_Fallopia
## 1       67.05     0.01 67.06         0.01
## 2       50.18     0.04 50.22         0.08
## 3       60.99     0.09 61.08         0.15
## 4       40.94     0.77 41.71         1.85
## 5       38.41     3.40 41.81         8.13
## 6       47.73     0.54 48.27         1.12
```

```
tail(MyData) #  Show last six rows of data
```

```
##     PotNum     Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 118    143 fluctuations      high bohem      5.06  12.81  23.82     3.64 0.16
## 119    144 fluctuations      high bohem     19.93  21.07   6.08     2.80 0.43
## 120    145 fluctuations      high bohem      4.89  32.93   6.30     9.64 0.00
## 121    147 fluctuations      high bohem      7.84  31.16  13.61     6.58 0.03
## 122    148 fluctuations      high bohem      4.15  38.70  23.59     5.11 1.36
## 123    149 fluctuations      high bohem      1.72  10.41  23.48     8.51 0.43
##     All_Natives Fallopia Total Pct_Fallopia
## 118       45.49    21.31 66.80        31.90
## 119       50.31     0.00 50.31         0.00
## 120       53.76     2.36 56.12         4.21
## 121       59.22     3.74 62.96         5.94
## 122       72.91     5.89 78.80         7.47
## 123       44.55    19.70 64.25        30.66
```

```
dim(MyData) # Number of dimension (rows columns)
```

```
## [1] 123  13
```

```
nrow(MyData) # Number of rows only
```

```
## [1] 123
```

```
ncol(MyData) # Number of columns only
```

```
## [1] 13
```

One more function is particularly useful for inspecting the **structure** of the data. We can use this to see column headers, types of data contained in each column, and the first few values in each column.

```
str(MyData)
```

```
## 'data.frame':    123 obs. of  13 variables:
##  $ PotNum      : int  1 2 3 5 6 7 8 9 10 11 ...
##  $ Scenario    : chr  "low" "low" "low" "low" ...
##  $ Nutrients   : chr  "low" "low" "low" "low" ...
##  $ Taxon       : chr  "japon" "japon" "japon" "japon" ...
##  $ Symphytum   : num  9.81 8.64 2.65 1.44 9.15 ...
##  $ Silene      : num  36.4 29.6 36 21.4 23.9 ...
##  $ Urtica      : num  16.08 5.59 17.09 12.39 5.19 ...
##  $ Geranium    : num  4.68 5.75 5.13 5.37 0 9.05 3.51 9.64 7.3 6.36 ...
##  $ Geum        : num  0.12 0.55 0.09 0.31 0.17 0.97 0.4 0.01 0.47 0.33 ...
##  $ All_Natives : num  67 50.2 61 40.9 38.4 ...
##  $ Fallopia    : num  0.01 0.04 0.09 0.77 3.4 0.54 2.05 0.26 0 0 ...
##  $ Total       : num  67.1 50.2 61.1 41.7 41.8 ...
##  $ Pct_Fallopia: num  0.01 0.08 0.15 1.85 8.13 1.12 3.7 0.61 0 0 ...
```

> Protip: `str()` is very important for functions that use data.frames including statistical analysis and plotting

Pay careful attention to integer `int` vs numeric `num` vs `factor` columns. These are the data types, and we'll look at these in more detail later.

One common source of error students make when starting to analyzing data is using the wrong data *type*.

Here's an example of data types gone rogue: In an analysis of variance (ANOVA), you want a `factor` as a predictor and a `num` or `int` as a response. But in linear regression you want `int` or `num` as a predictor instead of `factor`. If you code your factor (e.g. treatment) as a number (e.g. 1-4) then R will treat it as an integer when you import the data. When you run a linear model with the `lm` function, you will be running a regression rather than ANOVA! As a result, you will estimate a slope rather than the difference between group means.

2.8.4 Subset

The `data.frame` object can be subset, just like a matrix.

```
MyData[1,] # Returns first row of data.frame
```

```
##   PotNum Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 1      1      low       low japon      9.81  36.36  16.08     4.68 0.12
##   All_Natives Fallopia Total Pct_Fallopia
## 1       67.05     0.01 67.06         0.01
```

```
MyData[1,1] # Returns first value of data.frame
```

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

In addition, you can define the column names.

```
MyData[,"PotNum"] # Returns values in "PotNum" column
```

```
##   [1]   1   2   3   5   6   7   8   9  10  11  12  14  16  17  18  19  20  22
##  [19]  23  24  25  26  28  29  30  31  33  34  35  36  38  39  40  41  42  44
##  [37]  45  47  48  49  50  52  53  54  55  57  58  60  61  62  63  65  66  67
##  [55]  68  69  70  72  73  74  76  77  78  79  80  81  83  84  85  86  87  88
##  [73]  90  91  92  93  94  95  96  97  98 100 101 102 103 104 105 107 108 110
##  [91] 111 112 113 114 116 117 118 119 120 121 122 124 125 126 127 128 129 131
## [109] 132 133 134 135 136 138 139 140 142 143 144 145 147 148 149
```

```
MyData$PotNum # Another way to get the same output
```

```
##   [1]   1   2   3   5   6   7   8   9  10  11  12  14  16  17  18  19  20  22
##  [19]  23  24  25  26  28  29  30  31  33  34  35  36  38  39  40  41  42  44
##  [37]  45  47  48  49  50  52  53  54  55  57  58  60  61  62  63  65  66  67
##  [55]  68  69  70  72  73  74  76  77  78  79  80  81  83  84  85  86  87  88
##  [73]  90  91  92  93  94  95  96  97  98 100 101 102 103 104 105 107 108 110
##  [91] 111 112 113 114 116 117 118 119 120 121 122 124 125 126 127 128 129 131
## [109] 132 133 134 135 136 138 139 140 142 143 144 145 147 148 149
```

We can also subset the data based on particular row values. For example, we can find only the records in the *extreme* treatment scenario.

```
levels(MyData$Scenario)
```

```
## NULL
```

```
subset(MyData,Scenario=="extreme") # Subset
```

```
##    PotNum Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 74     91  extreme      high japon      6.77  29.29  17.58     1.20 0.98
## 75     92  extreme      high japon      9.19  16.76   7.32     4.84 0.21
## 76     93  extreme      high japon      6.55  48.19   0.00     9.21 0.05
## 77     94  extreme      high japon      5.22  18.21  21.51     6.53 0.58
## 78     95  extreme      high japon      4.92  25.93  40.33     4.92 0.07
## 79     96  extreme      high japon      9.51   3.15  36.16     10.16 0.62
## 80     97  extreme      high japon     14.98  33.34   2.56      7.38 0.16
## 81     98  extreme      high japon     17.37   4.38   2.73     16.41 0.00
## 82    100  extreme      high japon      7.36  24.90   3.21      7.30 0.46
## 83    101  extreme      high japon     10.54  31.97   7.37      4.62 0.54
## 84    102  extreme      high japon      9.51   0.00  17.56      6.97 0.00
## 85    103  extreme      high japon      6.92  49.38   0.00     10.33 0.42
## 86    104  extreme      high japon     12.42  18.36   4.44      6.42 0.42
## 87    105  extreme      high japon      5.89  37.92   4.71      5.18 0.09
## 88    107  extreme      high bohem      0.00  43.85   8.10      8.18 0.36
## 89    108  extreme      high bohem     10.29  27.47   5.47      0.00 0.40
## 90    110  extreme      high bohem     17.39  24.42  10.71      8.34 0.00
## 91    111  extreme      high bohem     12.59   0.00  41.08      4.81 1.38
## 92    112  extreme      high bohem     15.27   1.66  25.77      8.79 0.50
## 93    113  extreme      high bohem     12.52  21.52   0.00      5.71 1.11
## 94    114  extreme      high bohem     11.33  31.61   6.95     10.14 0.86
## 95    116  extreme      high bohem     14.89   0.00  10.16     10.28 0.18
## 96    117  extreme      high bohem     14.12  24.57  12.59      8.46 0.63
## 97    118  extreme      high bohem     14.21  25.15  12.69      8.06 0.29
## 98    119  extreme      high bohem      0.00  36.79   8.69      3.38 1.20
## 99    120  extreme      high bohem      9.85  15.71  15.56      3.89 1.46
##    All_Natives Fallopia Total Pct_Fallopia
## 74       55.82     1.98 57.80         3.43
## 75       38.32     3.40 41.72         8.15
## 76       64.00     3.44 67.44         5.10
## 77       52.05     6.73 58.78        11.45
## 78       76.17     1.57 77.74         2.02
## 79       59.60     6.08 65.68         9.26
## 80       58.42     0.00 58.42         0.00
## 81       40.89    14.46 55.35        26.12
## 82       43.23     7.05 50.28        14.02
## 83       55.04     0.00 55.04         0.00
## 84       34.04     5.52 39.56        13.95
## 85       67.05     4.02 71.07         5.66
## 86       42.06     3.17 45.23         7.01
## 87       53.79     3.41 57.20         5.96
## 88       60.49     7.21 67.70        10.65
## 89       43.63     8.83 52.46        16.83
## 90       60.86     0.00 60.86         0.00
## 91       59.86     2.33 62.19         3.75
## 92       51.99    13.54 65.53        20.66
## 93       40.86     7.33 48.19        15.21
## 94       60.89     0.00 60.89         0.00
## 95       35.51    12.62 48.13        26.22
## 96       60.37     0.00 60.37         0.00
## 97       60.40     7.46 67.86        10.99
## 98       50.06     6.34 56.40         11.24
## 99       46.47     2.66 49.13          5.41
```

2.8.5 New Columns

It's easy to add new columns to a data frame. For example, to add a new column that is the sum of two others:

```
MyData$Total<-MyData$Symphytum + MyData$Silene + MyData$Urtica
names(MyData)
```

```
##  [1] "PotNum"       "Scenario"     "Nutrients"    "Taxon"        "Symphytum"
##  [6] "Silene"       "Urtica"       "Geranium"     "Geum"         "All_Natives"
## [11] "Fallopia"     "Total"        "Pct_Fallopia"
```

```
print(MyData$Total)
```

```
##   [1] 62.25 43.88 55.77 35.26 38.24 37.71 49.46 32.77 45.76 39.20 49.84 36.28
##  [13] 40.52 35.78 39.26 45.33 38.54 33.27 50.09 48.55 49.25 31.36 36.56 37.21
##  [25] 57.06 48.48 39.32 28.03 47.19 57.63 53.27 37.79 43.56 45.47 56.68 43.88
##  [37] 47.03 47.41 50.64 34.74 54.40 48.46 64.48 49.31 48.97 54.00 55.71 64.35
##  [49] 38.73 41.31 52.31 56.77 50.10 47.22 38.61 59.73 75.59 36.33 53.57 54.94
##  [61] 39.21 45.22 25.39 57.10 38.00 37.51 56.86 43.48 51.86 67.97 16.43 58.13
##  [73] 52.64 53.64 33.27 54.74 44.94 71.18 48.82 50.88 24.48 35.47 49.88 27.07
##  [85] 56.30 35.22 48.52 51.95 43.23 52.52 53.67 42.70 34.04 49.89 25.05 51.28
##  [97] 52.05 45.48 41.12 45.67 38.18 53.30 40.52 36.70 58.78 37.31 14.48 43.19
## [109] 65.25 30.79 54.97 62.96 38.64 37.08 41.39 38.85 39.34 41.69 47.08 44.12
## [121] 52.61 66.44 35.61
```

2.9 Other Functions

There are a few more useful functions for inspecting your data.

2.9.1 `unique`

Find all the unique values within a vector using `unique`.

```
unique(MyData$Nutrients)
```

```
## [1] "low"  "high"
```

2.9.2 `duplicated`

Look at each value in a vector and return a `TRUE` if it is duplicated and `FALSE` if it is unique.

```
duplicated(MyData$Nutrients)
```

```
##   [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [25]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [61]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [73]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [85]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [97]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [109]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [121]  TRUE  TRUE  TRUE
```

This is a good example of a Boolean vector, which can be used to subset your data.

```
MyData$Nutrients[duplicated(MyData$Nutrients)]
```

```
##   [1] "low"  "low"  "low"  "low"  "low"  "low"  "low"  "low"  "low"  "low" 
##  [11] "low"  "low"  "low"  "low"  "low"  "low"  "low"  "low"  "low"  "low" 
##  [21] "low"  "low"  "low"  "low"  "high" "high" "high" "high" "high" "high"
##  [31] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
##  [41] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
##  [51] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
##  [61] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
##  [71] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
##  [81] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
##  [91] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
## [101] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
## [111] "high" "high" "high" "high" "high" "high" "high" "high" "high" "high"
## [121] "high"
```

Note that only the `TRUE` values are returned (duplicated).

2.9.3 aggregate

Quickly calculate means of one vector (Total) for each value of a grouping variable (Nutrients).

```
aggregate(MyData$Total,list(MyData$Nutrients), mean)
```

```
##   Group.1        x
## 1    high 46.51173
## 2     low 42.76800
```

The ~ provides an alternative way to write this function. In R the ~ usually means 'by' and is often used in statistical models. Here we can say aggregate Total 'by' Nutrients grouping.

```
aggregate(Total ~ Nutrients, data=MyData, mean)
```

```
##   Nutrients    Total
## 1      high 46.51173
## 2       low 42.76800
```

The nice thing about doing it this way is that we preserve the column name (Total instead of x).

You can also use this to calculate means across different grouping variables.

```
aggregate(Total ~ Nutrients*Taxon*Scenario, data=MyData, mean)
```

```
##    Nutrients Taxon    Scenario    Total
## 1       high bohem     extreme 45.24833
## 2       high japon     extreme 45.31500
## 3       high bohem fluctuations 43.89545
## 4       high japon fluctuations 44.77692
## 5       high bohem     gradual 45.36923
## 6       high japon     gradual 50.43417
## 7       high bohem        high 52.04273
## 8       high japon        high 45.69417
## 9        low bohem         low 41.75231
## 10       low japon         low 43.86833
```

Note that `mean` is just the `mean()` function in R. We can use other functions, like the standard deviation `sd`:

```
aggregate(Total ~ Nutrients, data=MyData, sd)
```

```
##   Nutrients     Total
## 1      high 11.175885
## 2       low  8.227402
```

2.9.4 tapply

The `tapply` function is a more general way to replicate functions.

```
tapply(MyData$Total, list(MyData$Nutrients), mean) # calculate means
```

```
##     high      low
## 46.51173 42.76800
```

Compare the two outputs with `aggregate` above

2.10 Tidyverse

Most of the methods above for managing and summarizing data are the *classic* or *base R* functions. More recently, the **tidyverse** group of functions was introduced and has a lot of advantages over the classic tools, particularly for complex data management. For example, it is easy to string together multiple steps into a single 'pipe' of data reorganization. The Data Science Tutorial introduces `dplyr` – one of the tools in the `tidyverse`.

2.11 Save

Just as we can load FROM external files, we can save TO external files

```
## Calulate means
NutrientMeans<-tapply(MyData$Total,list(MyData$Nutrients),mean)
## Save means as .csv file
write.csv(NutrientMeans,"MyData_Nutrient_Means.csv")
```

You should see this file in your working directory.

2.12 Flow control

Think of your data analysis as a stream flowing from the raw data at the headwaters and flowing down to the end, exiting as a full analysis with graphics, stats, and interpretation.

There are different ways we can control the flow of the water. The simplest is just to write a sequence of lines of code, with the output of one line of code forming the input of the next. There are several examples of this above:

```
A<-functionA()
B<-functionB(A)
C<-functionC(B)
```

But sometimes we may want to do the same function or analysis only if the input meets certain criteria. Or we may want to reiterate the same analysis multiple times on different inputs. This is where more advanced flow control comes in handy.

To start, let's make up a couple of objects to play with:

```
X<-21
Xvec<-c(1:10,"string")
```

2.12.1 `if`

The `if` statement uses an **operator** (see above) to asses whether the value is `TRUE` or `FALSE`:

```
if(X > 100){ # Is X greater than 100?
  print("X > 100") # If TRUE
} else {
  print("X <= 100") # If FALSE
}
```

```
## [1] "X <= 100"
```

> **NOTE**: A common 'rookie' mistake is to leave out a bracket or use the wrong type.

Use regular brackets for the if function `if()` followed by two sets of curly brackets `{}else{}`.

Break up across multiple lines to improve readability. Note that you don't need an `else{}` part if you just want to 'do nothing' when `FALSE`.

```
if(1 > 0){print ("yup")}
```

```
## [1] "yup"
```

2.12.2 ifelse

The `ifelse` is a more compact version for simple comparisons. The following code does the same as above.

```
ifelse(X > 100, print("X > 100"), print("X <= 100"))
```

```
## [1] "X <= 100"
```

```
## [1] "X <= 100"
```

2.12.3 `for` loop

A loop does the same thing over and over again until some condition is met. In the case of a `for` loop, we set a 'counter' variable and loop through each value of the counter variable. Here are a few examples:

```
# Loop through numbers from 1 to X
for (i in 1:X){
  print(paste(X,i,sep=":"))
}
```

```
## [1] "21:1"
## [1] "21:2"
## [1] "21:3"
## [1] "21:4"
## [1] "21:5"
## [1] "21:6"
## [1] "21:7"
## [1] "21:8"
## [1] "21:9"
## [1] "21:10"
## [1] "21:11"
## [1] "21:12"
## [1] "21:13"
## [1] "21:14"
## [1] "21:15"
## [1] "21:16"
## [1] "21:17"
## [1] "21:18"
## [1] "21:19"
## [1] "21:20"
## [1] "21:21"
```

```r
# Loop through elements of a vector directly
for (i in Xvec){
  print(i)
}
```

```
## [1] "1"
## [1] "2"
## [1] "3"
## [1] "4"
## [1] "5"
## [1] "6"
## [1] "7"
## [1] "8"
## [1] "9"
## [1] "10"
## [1] "string"
```

```r
# Use an index to loop through the elements
for (i in 1:length(Xvec)){
  print(Xvec[i])
}
```

```
## [1] "1"
## [1] "2"
## [1] "3"
## [1] "4"
## [1] "5"
## [1] "6"
## [1] "7"
## [1] "8"
## [1] "9"
## [1] "10"
## [1] "string"
```

Note that in each case there is a vector and we loop through each cell in the vector. The `i` keeps track of the cell value in each iteration of the loop.

Loops can be tricky, and the only way to really learn them is to try to write a bunch. Whenever you find yourself writing similar code more than 2 or 3 times, challenge yourself to try to do it as a loop.

In addition to looping through a vector, it can often be useful to include a counter variable.

One thing to watch out for is what part of the loop you want to update the counter variable. USUALLY it will be at the beginning

```
count1<-1
count10<-1

for(i in 1:10){
  print(paste("count1 =",count1))
  print(paste("count10 =",count10))
  count1<-count1+1
  count10<-count10*10
}
```

```
## [1] "count1 = 1"
## [1] "count10 = 1"
## [1] "count1 = 2"
## [1] "count10 = 10"
## [1] "count1 = 3"
## [1] "count10 = 100"
## [1] "count1 = 4"
## [1] "count10 = 1000"
## [1] "count1 = 5"
## [1] "count10 = 10000"
## [1] "count1 = 6"
## [1] "count10 = 1e+05"
## [1] "count1 = 7"
## [1] "count10 = 1e+06"
## [1] "count1 = 8"
## [1] "count10 = 1e+07"
## [1] "count1 = 9"
## [1] "count10 = 1e+08"
## [1] "count1 = 10"
## [1] "count10 = 1e+09"
```

or at the end.

```
countbefore<-0
countafter<-0
for(i in 1:10){
  countbefore<-countbefore+1
  print(paste("before =",countbefore))
  print(paste("after =",countafter))
  countafter<-countafter+1
}
```

```
## [1] "before = 1"
## [1] "after = 0"
## [1] "before = 2"
## [1] "after = 1"
## [1] "before = 3"
## [1] "after = 2"
## [1] "before = 4"
## [1] "after = 3"
## [1] "before = 5"
## [1] "after = 4"
## [1] "before = 6"
## [1] "after = 5"
## [1] "before = 7"
## [1] "after = 6"
## [1] "before = 8"
## [1] "after = 7"
## [1] "before = 9"
## [1] "after = 8"
## [1] "before = 10"
## [1] "after = 9"
```

Read through the outputs above carefully to make sure you understand how the loops work. When you are confident you understand, then write a new for loop and write down the predicted output. Run the loop to check if you were right.

2.12.4 Nested Loops

Counters are particularly valuable when you have a nested loop, which is just one loop inside of another.

In the example below, we are first looping through a vector of length 3, tracked with `i`. Then **for each i** we do a second loop, tracked by `j`.

This time, try to predict the output BEFORE you run the loop. Write it down, then run the loop to check your answer.

```
LoopCount<-0
for(i in 1:3){
  for(j in 1:4){
    LoopCount<-LoopCount+1
    print(paste("i = ",i))
    print(paste("j = ",j))
    print(paste("Loop =",LoopCount))
  }
}
```

```
## [1] "i =  1"
## [1] "j =  1"
## [1] "Loop = 1"
## [1] "i =  1"
## [1] "j =  2"
## [1] "Loop = 2"
## [1] "i =  1"
## [1] "j =  3"
## [1] "Loop = 3"
## [1] "i =  1"
## [1] "j =  4"
## [1] "Loop = 4"
## [1] "i =  2"
## [1] "j =  1"
## [1] "Loop = 5"
## [1] "i =  2"
## [1] "j =  2"
## [1] "Loop = 6"
## [1] "i =  2"
## [1] "j =  3"
## [1] "Loop = 7"
## [1] "i =  2"
## [1] "j =  4"
## [1] "Loop = 8"
## [1] "i =  3"
## [1] "j =  1"
## [1] "Loop = 9"
## [1] "i =  3"
## [1] "j =  2"
## [1] "Loop = 10"
## [1] "i =  3"
## [1] "j =  3"
## [1] "Loop = 11"
## [1] "i =  3"
## [1] "j =  4"
## [1] "Loop = 12"
```

2.12.5 `while` loop

The `while` is another kind of loop, but instead of looping through a predefined set of variables, we iterate until some condition is met inside of the loop. This is called the **exit condition**.

Often, the `while` loop is used in optimization algorithms, where many calculations are run until some optimum or threshold value is reached.

One common coding error associated with `while` loops is that the exit condition is never reached, causing your computer to run an infinite loop.

Here's a simple while loop, which will continue until `count` is greater than or equal to `X`.

```
count<-0
while (count < X){
  print(count)
  count<-count+1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
```

2.13 Packages

As noted above, **functions** in R use brackets `()` and generally have **input** and **output** objects as well as **parameters** that affect their behaviour.

All of the functions in this tutorial are automatically loaded when you start R. There are many more functions available from developers. For example, our lab developed the `baRcodeR` package for creating unique identifier codes with printable barcodes and data sheets to help with sample management and data collection.

A **package** in R is a set of functions grouped together. For example, the `stats` package is automatically loaded when you run R and contains many useful functions. You can see what package a function belongs to at the beginning of the help file:

```
?cor
```

2.13.1 Installing

Before you can use a new function from a package that isn't installed with R, you first have to install the package on your computer. You only have to do this once per computer. However, it is a good idea to update the package frequently, especially when you update your version of R. This ensures that you are using the most recent version of the package.

Note that **installing** a package just downloads it from an online '**repository**' and saves to your computer.

Packages are installed with `install.packages()`, with the package indicated with single or double quotation marks. You may be asked for a repository, in which case choose one that is geographically close to you.

```
install.packages('baRcoeR')
```

2.13.2 Loading

Once you have installed a package, you can access it two ways.

2.13.2.1 1. Library

You can load the package using the `library()` function, giving you access to functions contained within it:

```
library(baRcodeR)
make_labels()
```

2.13.2.2 2. Function

You can run a function without loading the whole package:

```
baRcodeR::make_labels()
```

This translates to ”Run the `make_labels` function from the `baRcodeR` library.

The first method is more common, especially for the commonly used functions covered in these self-tutorials. However, the second method is convenient if you just want to use one function from a large library that take up a lot of space.

More importantly, some packages have **functions with the same names**. Let’s say you load two packages `pkgA` and `pkgB` that have different functions but both are called `cor`. When you run the `cor` function, R will assume you want the one from the package that was most recently loaded using the `library()` function. However, you can over-ride this with the second method:

```
pkgA::cor()
pkgB::cor()
```

2.13.2.3 Library vs Package

The terms **library** and **package** are often used interchangeably. Technically, the **package** is the collection of functions whereas the **library** is the specific folder where the R packages are stored. A library may contain more than one package.

For the most part, you just need to know that a package and a library are a collection of functions.

2.14 Readable code

It’s important to make your code readable and interpretable by collaborators, peer reviewers, and yourself 6 months from now. There are lots of opinions on this but here are a few basic suggestions:

1. Add documentation to explain what you are doing
2. Add spacing between parameters to improve readability

3. Add spacing on either side of `<-` when making objects
4. Break long functions into multiple lines; add the line break after a `,` or `+`
5. Follow these suggestions for object/column/file names a. Try to keep your names short and concise but meaningful b. Use underscore `_` to improve readability and avoid `.` c. Always start with a letter d. Avoid symbols

Bad	Good
`sum(X,na.rm=T)`	`sum(X, na.rm=T)`
`data.frame(X, Y, ..., data=...)`	`data.frame(X, Y, ..., data=...)`
`X`	`Mass`
`Days.To.First.Flower`	`Flwr_Days` or `FDays`
`10d.Height`	`Ht10d`
`Length*Width`	`LxW`

To take your code to the next level, look into the Tidyverse Style Guide

2.15 TEST yourself:

Are you ready to test your knowledge?

If so, click HERE

Chapter 3

Quick Visualizations

3.1 Overview

Visualizing data is a key step in any analysis. Whether you are just starting to understand the structure of your data or polishing off the perfect figure for publication in 'tabloid' journal like Science or Nature, R provides powerful and flexible graphing tools.

In this tutorial, you will learn how to make quick graphs with the `qplot` function, with the option of some basic formatting customization. In the ggplot Tutorial you will learn some additional tricks and resources for developing your graphing skills even further

By the time you are finished these two self-tutorials, you will have all the resources you need to make **publication-ready graphics**! All you will really need to do is practice and apply what you have learned.

Both `qplot` and `ggplot` come from the `ggplot2` package. Don't ask me what happened to ggplot1... Developed by Hadley Wickham and the same team as R Studio, `ggplot2` is part of the Tidyverse group of helpful R packages.

Once you have mastered these tutorials, you might want to continue to expand your `ggplot` repertoire by reading through additional examples in the `ggplot2` documentation http://docs.ggplot2.org/current/

> WARNING: There is a learning curve!

Learning visualizations in R can feel like a struggle at first, and you may ask yourself: *Is it worth my time?* If you already have experience making figures with graphics programs, you may ask youself: *Why deal with all these coding errors when I can just generate a quick figure in a different program?* There are a few good reasons to invest the time to get over the learning curve and use R for all your graphing needs.

1. You will get much faster with practice
2. You have much more control over every aspect of your figure
3. Your visualization will be **reproducible**, meaning anyone with the data and the code can reproduce every aspect of your figure, from each individual data point down to the specific axis labels.

The third point is worth some extra thought. Everybody makes mistakes, whether you are graphing with R or a point-and-click graphics program. If you make an error in R you will either get an error message telling you, or you will have reproducible code that you can have somebody check. If you make a mistake in a point-and-click program, you may produce a graph that is incorrect and no way to check!

Now consider what happens if you add new data or find a mistake in your original data that needs to be corrected. With a point-and-click program you have to make the graph all over again. With R, you just rerun the code with new input and get the updated figure right away!

3.1.1 Graphical Concepts

There are a few universal graphing concepts that are important to understand in order to create publication-ready graphics in R.

3.1.2 Vector vs. Raster

The first concept is to understand that there are two main **file types** that you can use to save your graphs. Once saved, you can send these to graphics programs, printers, or journal websites.

Raster files save graphs in a 2-dimensional grid of data corresponding to pixels. You are probably quite familiar with this format if you've ever worked with a digital photo or an 8-bit video game. Some popular Raster file types include *JPEG/JPG, PNG, TIFF, and BMP*.

Vector files save information about points, lines and curves. If you've ever drawn a shape in a program like Microsoft Powerpoint or Adobe Illustrator, you might have some sense of how this works. Some popular vector formats include *SVG, PDF, EPS, AI, PS*.

Why does this matter?

In most cases, you should save your visualizations as **vector** files. *SVG* is a good choice, because it can be interpreted by web browsers and it is not proprietary. *PDF* and *EPS* are commonly used by publishers. The reason for this is that *raster* files introduce artifacs when they are scaled. You may have seen some images that look 'pixelated' – this happens when you try to expand the size of a lower-resolution figure.

In contrast, you can expand **vector** images to any size and the lines will always be clean and clear. This is why they are generally preferred for publication. There are a few important exceptions, however.

1. **Photographs** – Photographs captured by a camera are saved in the *Raster* format and cannot be converted to vector without significant loss of information
2. **Grid Data** – Raster data are convenient for plotting data that occurs in a grid. This may include spatial data that is broken up into a geographic grid. However, even in these cases, you may often want to use the vector format so that the overlapping geographical features (e.g. borders, lakes, rivers) are not pixellated.
3. **Large Data** – With some large data applications a graph may have many millions or billions of data points or lines. In these cases, the **vector** file would be too big to use in publication (e.g. several gigabytes). In this case you might opt for a high-resolution *Raster* file. On the other hand, you could graph your data using a density grid with colours corresponding to the density of points. In this case, you could use the *vector* format to maintain clean lines for the graph axes and labels.

The bottom line: you generally want to save your graphics as *SVG* or *PDF* files if you plan to publish them.

3.1.3 Resolution vs Dimension

In cases where you do need to use raster images in a publication, pay careful attention to the image's **Pixel Dimension**. We are used to thinking about resolution – for example a 2 megapixel camera is better than a 1 megapixel camera. Or 200 dpi (dots per inch) is better than 100 dpi. But it's not just the **resolution** that matters, the image **size** also determines the quality of the final image. The **size** and **resolution** of an image jointly determine its **Pixel Dimension**.

For example, an image with 200 dpi that is 1 x 3 inches will have the same pixel dimension of an image with 100 dpi that is 2 x 6 inches. These images will look exactly the same if they are printed at the same size. That's because the pixel dimension determines how an image looks on a page, and both of these have the same pixel dimension: 200 x 600 pixels.

3.1.4 Screen vs Print

Another important consideration is whether your figures are intended for a computer screen or printed page (or both). Each pixel of your screen has tiny lights that determine the specific colour that is reproduced. The **emission** of different wavelengths from your screen produces the different colours. In contrast, printed images get their colour from combinatios of ink, which **absorb** different wavelengths of colour. As a result, some colours that look fine on your screen do not reproduce well in print. In print, the intensity of colours are limited by the intensity of the Cyan, Magenta and Yellow inks that are used to reproduce the images. This is called CMYK printing.

Some programs like Adobe Illustrator have options to limit the colours on your screen to only display colours that can be reproduced with CMYK printing.

3.1.5 Accessibility

There are two other important considerations for colouring your figures. First, remember that a significant portion of the population is unable to see certain colours. In addition, many scientists and students may print out your paper in black-and-white. You should try to choose colours that can be discerned in these situations.

The `viridis` package is a good tool for this. See: https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html

3.1.6 Slides

Some of the above information is covered and expanded in the Introductory Slides for examples of graphical output from R, using `ggplot2` and other graphing functions. These slides cover:

1. Graphing examples
2. Graphical concepts
3. `ggplot` grammar
4. anatomy of a `qplot`/`ggplot` graph

3.2 Getting Started

Install the `ggplot2` package using `install.packages("ggplot2")`. Load it with the `library` function whenever you want to include a function from the `ggplot2` library in your code..

```
library(ggplot2)
```

The `ggplot2` package includes two main functions, quickplot `qplot()` for fast graphs and the `ggplot()` function for more detailed, customizable graphs.

3.2.1 Data setup

We will again be working with the `FallopiaData.csv` dataset, which can be downloaded here: https://colauttilab.github.io/RCrashCourse/FallopiaData.csv, and saved to a folder called `Data` inside your project folder. You can find your currentworking folder with the `getwd()` function. This was covered in the [R Fundamentals Tutorial].

Now load the *.csv* file into R as a `data.frame` object:

```
MyData<-read.csv("./Data/FallopiaData.csv", header=T)
```

Alternatively, you can load the file right from the internet:

```
MyData<-read.csv(
  "https://colauttilab.github.io/RCrashCourse/FallopiaData.csv")
```

This dataset comes from the research group of Dr. Oliver Bossdorf at the University of Tuebingen in Tuebingen, Germany – a wonderful little town on the Neckar River. The data come from a plant competition experiment involving two invasive species from the genus *Fallopia*.

```
str(MyData)
```

```
## 'data.frame':    123 obs. of  13 variables:
##  $ PotNum      : int  1 2 3 5 6 7 8 9 10 11 ...
##  $ Scenario    : chr  "low" "low" "low" "low" ...
##  $ Nutrients   : chr  "low" "low" "low" "low" ...
##  $ Taxon       : chr  "japon" "japon" "japon" "japon" ...
##  $ Symphytum   : num  9.81 8.64 2.65 1.44 9.15 ...
##  $ Silene      : num  36.4 29.6 36 21.4 23.9 ...
##  $ Urtica      : num  16.08 5.59 17.09 12.39 5.19 ...
##  $ Geranium    : num  4.68 5.75 5.13 5.37 0 9.05 3.51 9.64 7.3 6.36 ...
##  $ Geum        : num  0.12 0.55 0.09 0.31 0.17 0.97 0.4 0.01 0.47 0.33 ...
##  $ All_Natives : num  67 50.2 61 40.9 38.4 ...
##  $ Fallopia    : num  0.01 0.04 0.09 0.77 3.4 0.54 2.05 0.26 0 0 ...
##  $ Total       : num  67.1 50.2 61.1 41.7 41.8 ...
##  $ Pct_Fallopia: num  0.01 0.08 0.15 1.85 8.13 1.12 3.7 0.61 0 0 ...
```

The first 4 columns give information about the pot and treatments. The rest give biomass measurements.

3.3 Basic Graphs

Think back to the R Fundamentals Tutorial, and you will hopefully recall the different data types. For graphing purposes, there are two main types of data: **categorical** and **continuous** putting these together in different combinations with `plot` gives us different graphs. One nice thing is that we don't have to specify which graphs to make; `qplot` will decide which type of graph based on the type of data. Of course, we can also specify a plot type if we want something different.

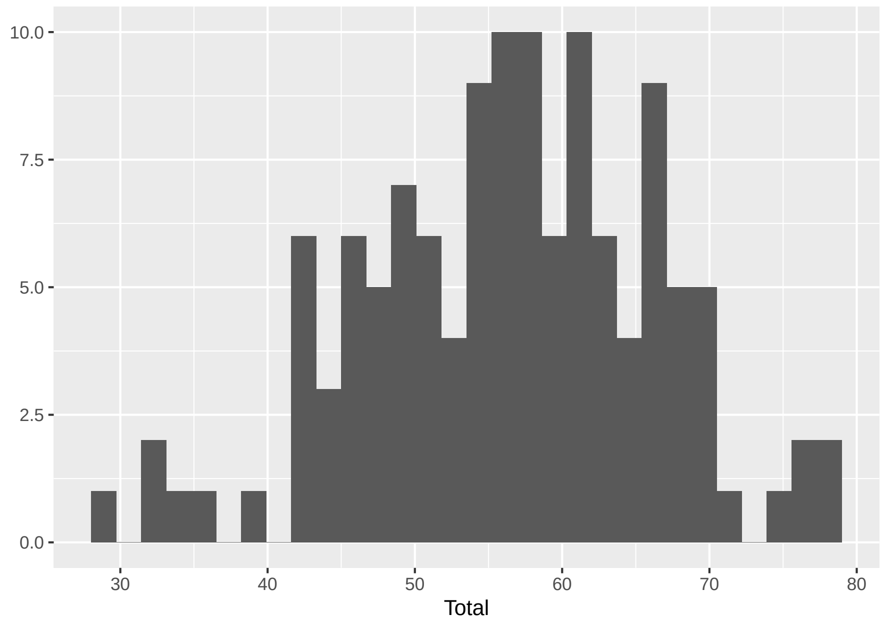
3.3.1 One continuous

If we input one continous variable, `qplot` will produce a frequency histogram by default.

Histogram

```
qplot(x=Total, data=MyData)
```

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

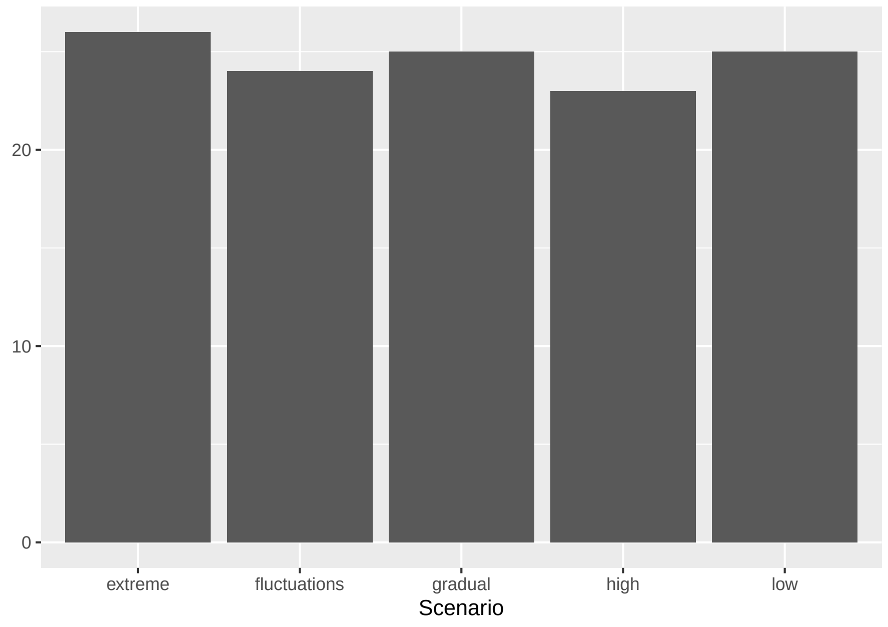


3.3.2 One categorical

If we input one categorical variable, `qplot` will produce a bar graph showing counts for each category.

Bar Graph

```
qplot(x=Scenario, data=MyData)
```

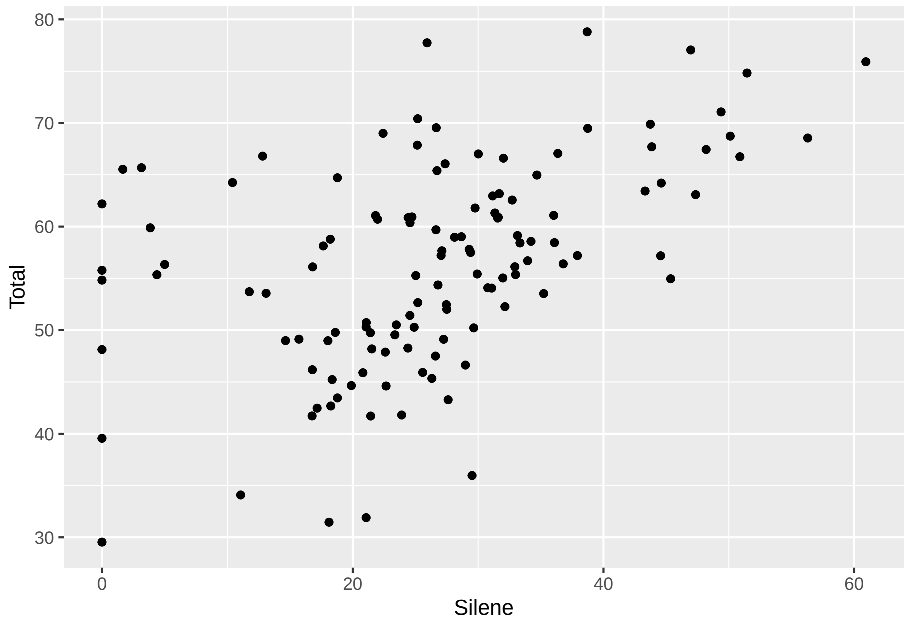


3.3.3 Two continuous

If we input two continuous variables, we get the classic bivariate plot a.k.a. scatterplot.

Bivariate Plot

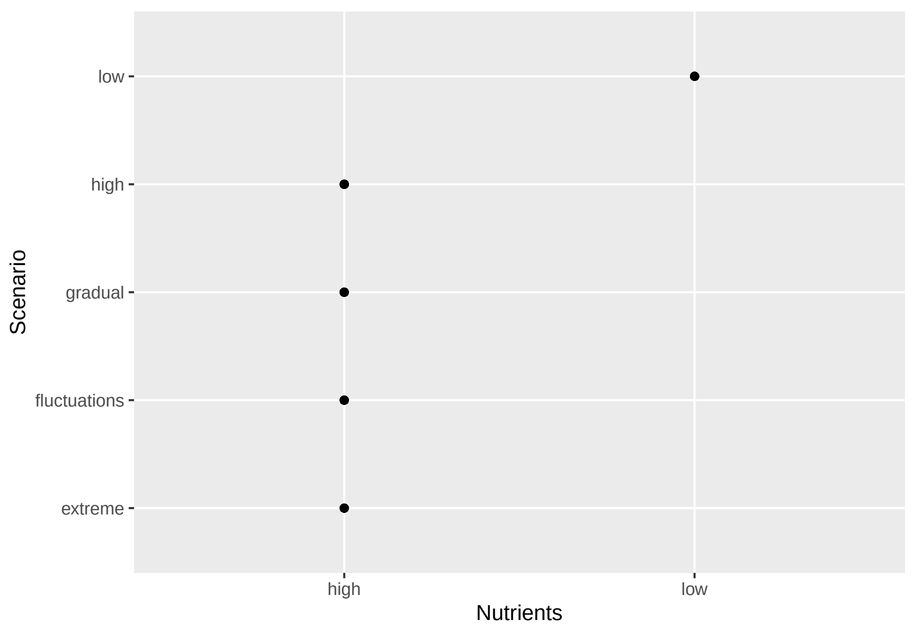
```
qplot(x=Silene, y=Total, data=MyData)
```



3.3.4 Two categorical

Plotting two categorical variables just plots overlapping dots, and it's not so useful except if you want to check for representation of particular variables.

```
qplot(x=Nutrients, y=Scenario, data=MyData)
```



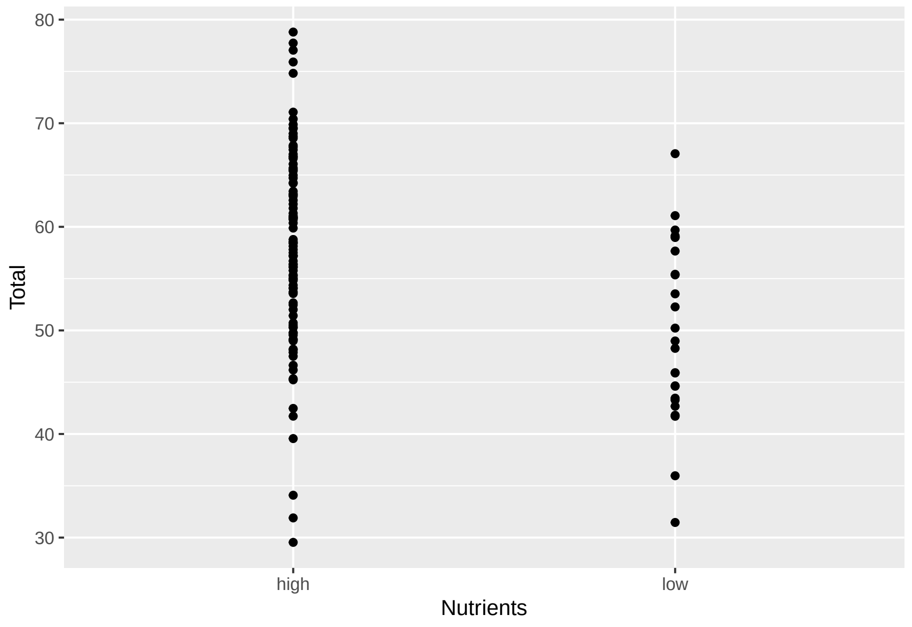
In this case we can see that there is only one type of “Low” Nutrient treatment, but 4 types of “High” treatments. In other words, all of the rows with “Low” nutrient treatment also have the “Low” scenario, and NONE of the rows with “Low” in the scenario column have “Low” in the Nutrients column.

3.3.5 Categorical by continuous

If we input a combination of categorical and continuous variables, we see get a categorical scatterplot showing the spread of points along the Y or X axis.

Categorical scatter plots

```
qplot(x=Nutrients, y=Total, data=MyData)
```



That's it! That's all you need to start exploring your data! Load your data frame, and plot different combinations of variables to look at the distribution of values or the relationship between your variables.

Of course, you might also want to make a few quick tweaks to the look of the graph.

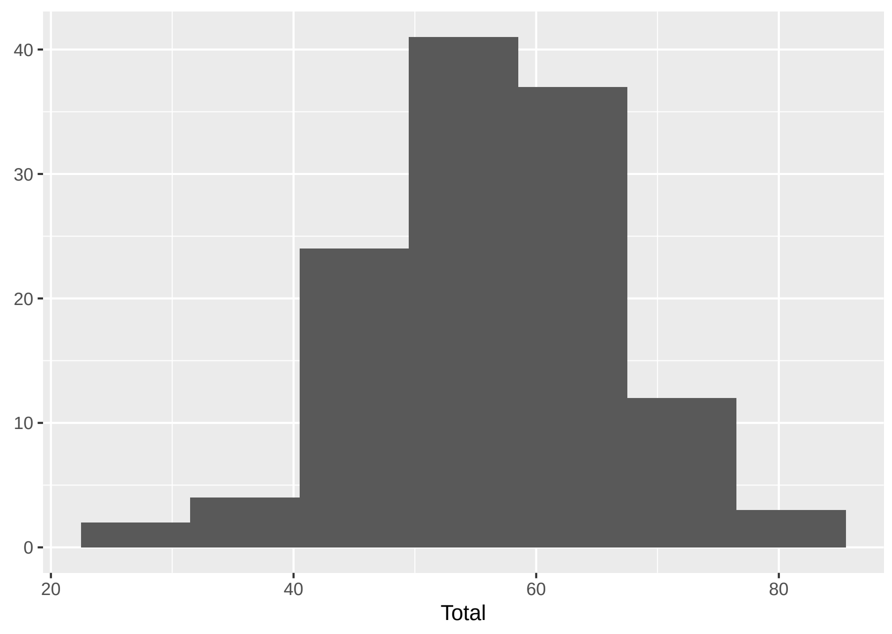
3.4 Basic customization

There are a number of parameters available with `qplot()` to quickly customize a few of the basic features of your graphs. Most of these will work with `ggplot()` as well, though the syntax or context is a bit different in some cases. Refer back to these when you work through the ggplot Tutorial.

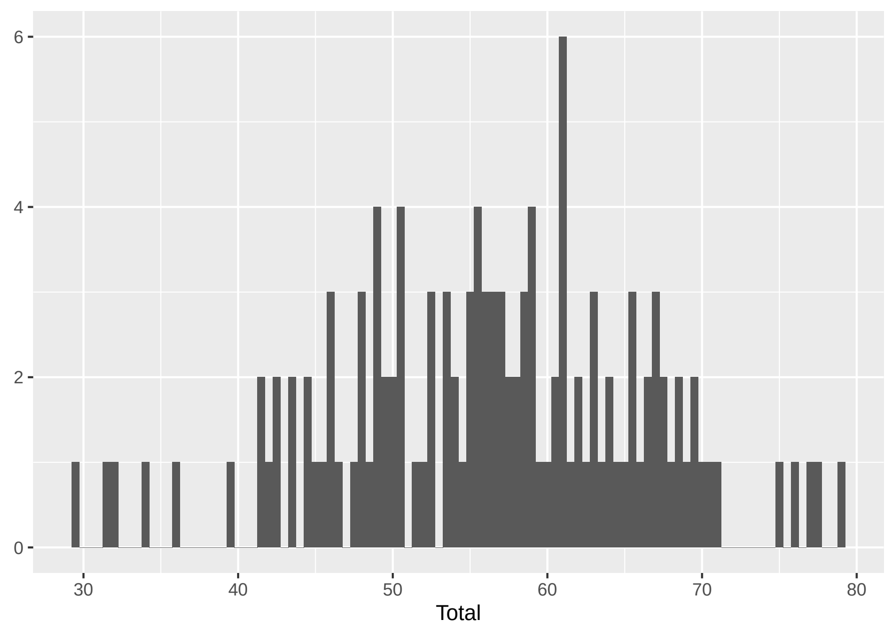
3.4.1 binwidth

Use this with the histogram graph to alter the size of the 'bins' along the x-axis.

```
qplot(x=Total, data=MyData, binwidth=9)
```



```
qplot(x=Total, data=MyData, binwidth=0.5)
```



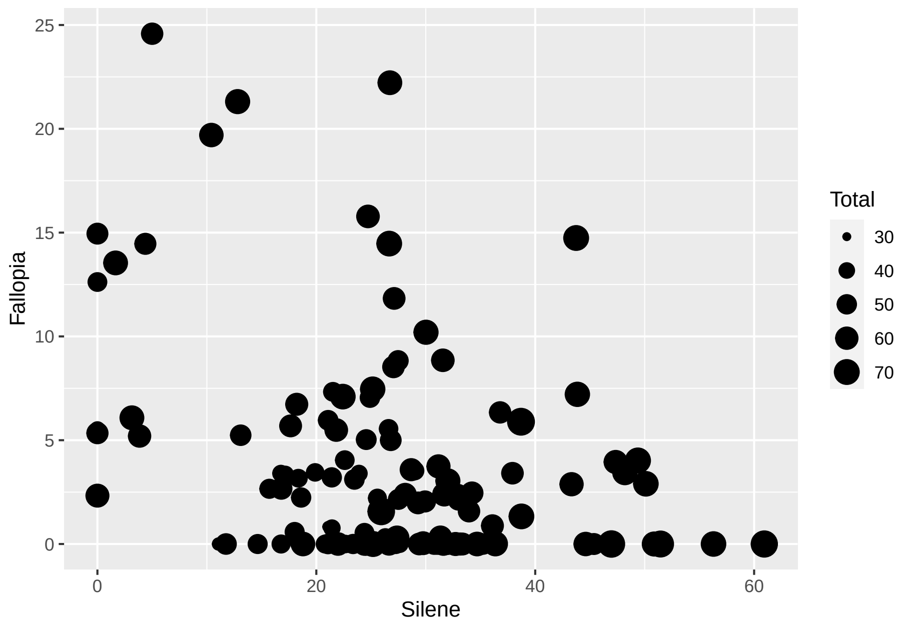
3.4.2 `size`

This controls the size of points (usually) or sometimes lines, depending on the context. It's also important to understand that many parameters in plotting can be inerpreted in two ways

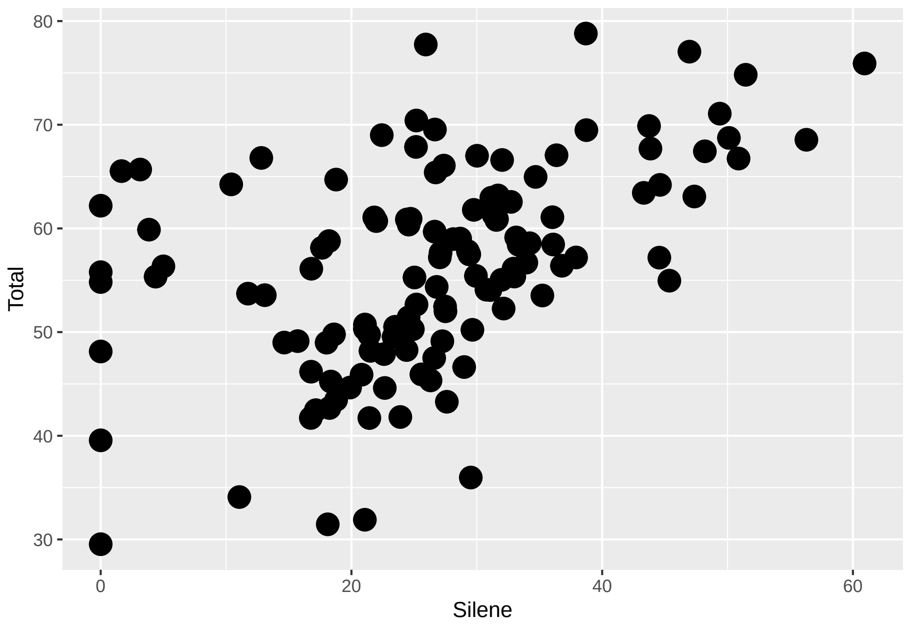
1. As a vector (e.g. scale this colour/size/shape based on a column of data)
2. As a value (e.g. I want this exact colour/size/shape)

In `qplot` the second option usually requires the **identity** function `I`. Compare these two examples.

```
qplot(x=Silene, y=Fallopia, data=MyData, size=Total)
```



```
qplot(x=Silene, y=Total, data=MyData, size=I(5))
```



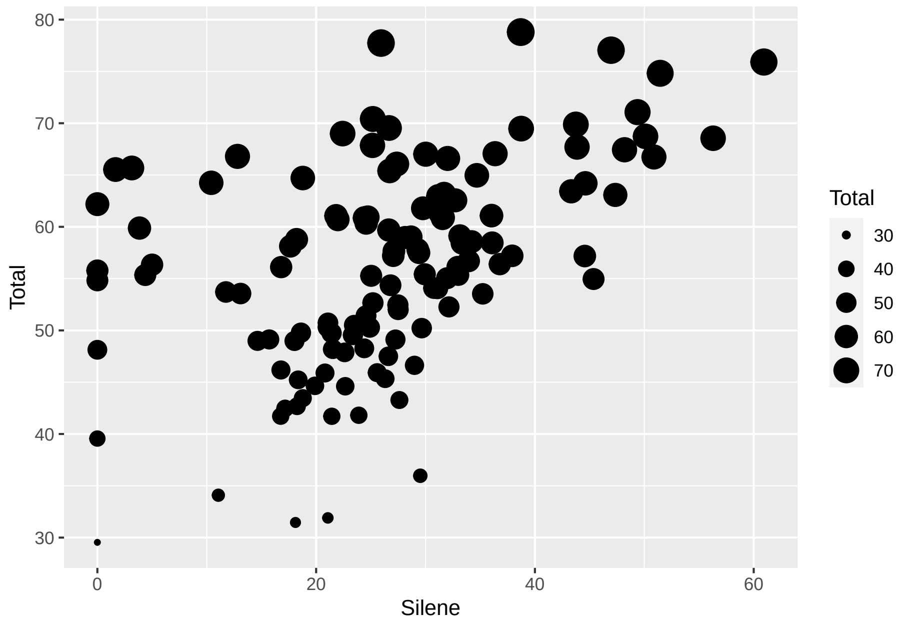
NOTE the use of the identity function: `I()` for constant. What happens if you just put a 5 in there without the identity function? Do it and see if you can figure out the problem.

Try using different numbers for comparison (e.g., 2, 5 and 10). How does the size change when you include the `I()` function vs when you just put the number in?

The explanation is a bit tricky, but imagine if you added a column to your data frame and put in a number. Then for size you put the column name. R would treat this as a variable and scale the size of the symbol to the value in this column. However, since all the rows have the same value, then they all have the same scaled size.

Now try using a different column for scaling:

```
qplot(x=Silene, y=Total, data=MyData, size=Total)
```



You can see in the legend how the point size scales with the column called *Total*

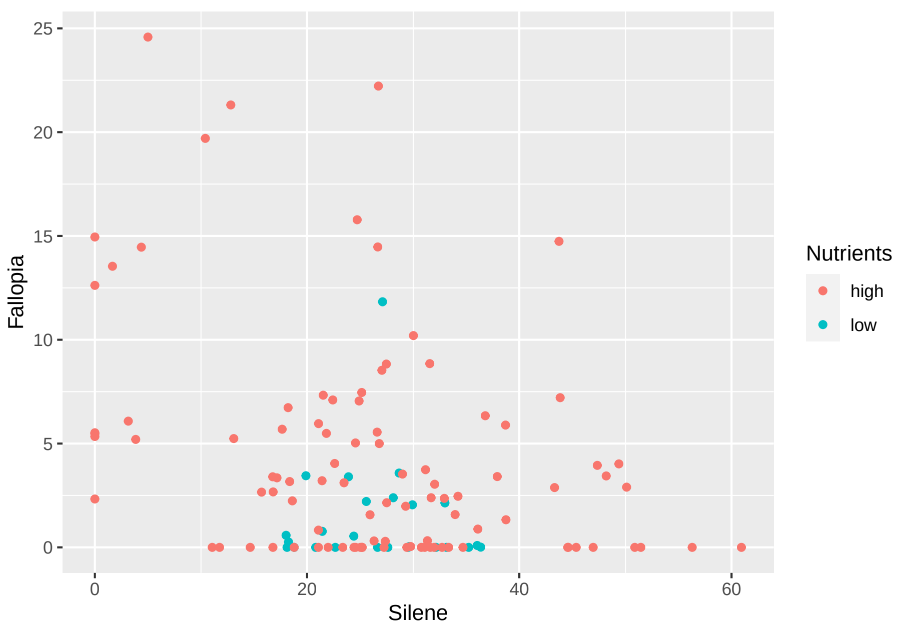
On the other hand, the identify function `I()` tells R: *I want to use this exact point size.*

3.4.3 `colour or color`

Another nice feature of `qplot` and `ggplot` is that you can use different spelling. You can use colour or color to colour add colour to your graphs.

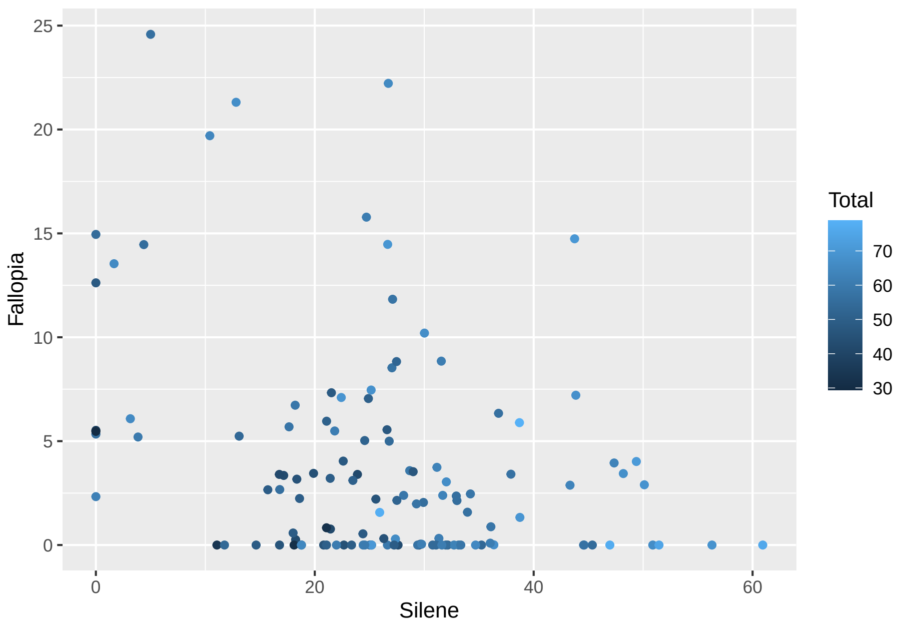
For example, you can colours points based on a factor…

```
qplot(x=Silene, y=Fallopia, data=MyData, colour=Nutrients)
```



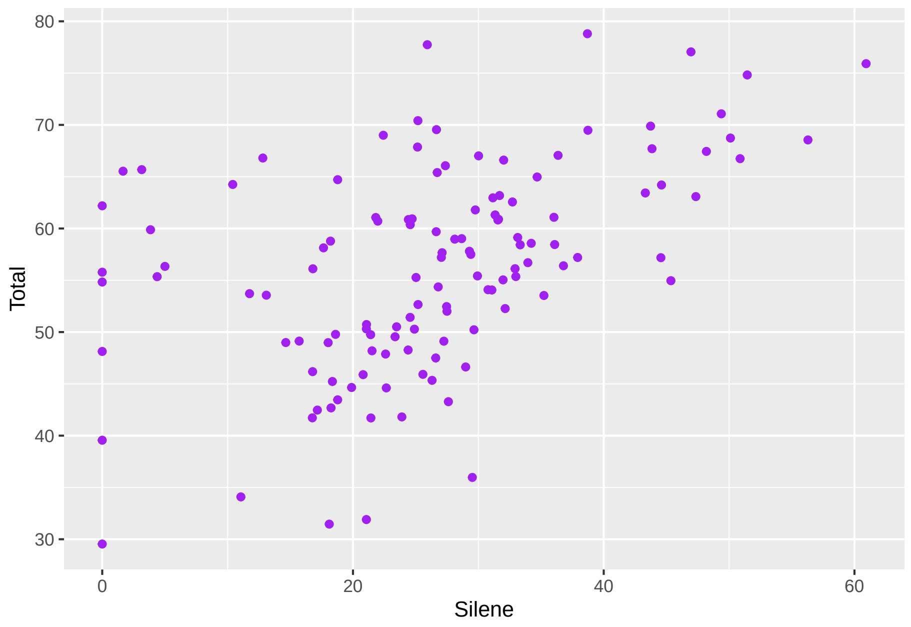
… or a continuous variable.

```
qplot(x=Silene, y=Fallopia, data=MyData, colour=Total)
```



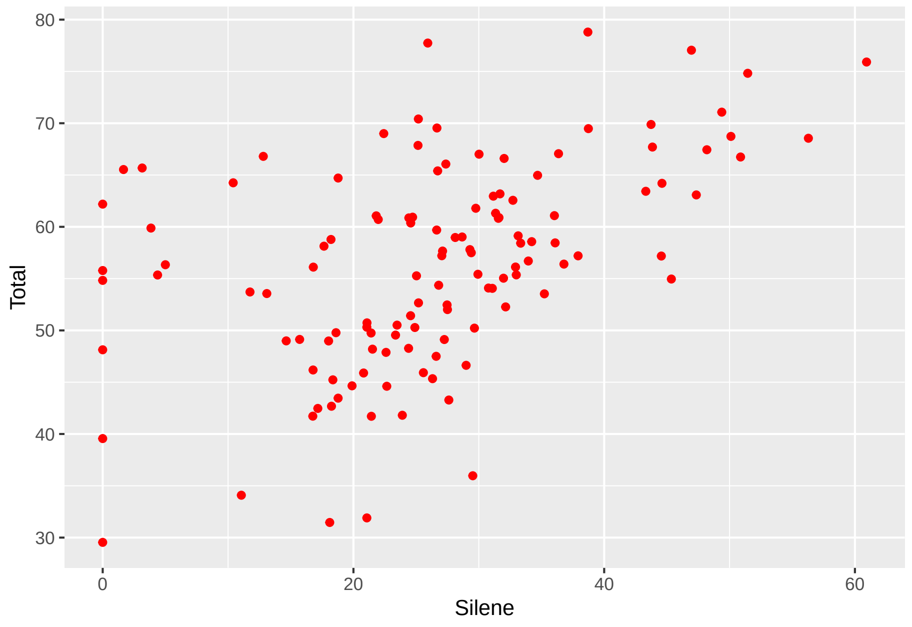
Or use the identity function `I()` again to set a specific colour.

```
qplot(x=Silene, y=Total, data=MyData, colour=I("purple"))
```



Not all colours are available as strings, but you can make just about any colour with the `rgb()` function. It takes three values corresponding to the intensity of red, green and blue light, respectively.

```
qplot(x=Silene, y=Total, data=MyData, colour=I(rgb(1,0,0)))
```

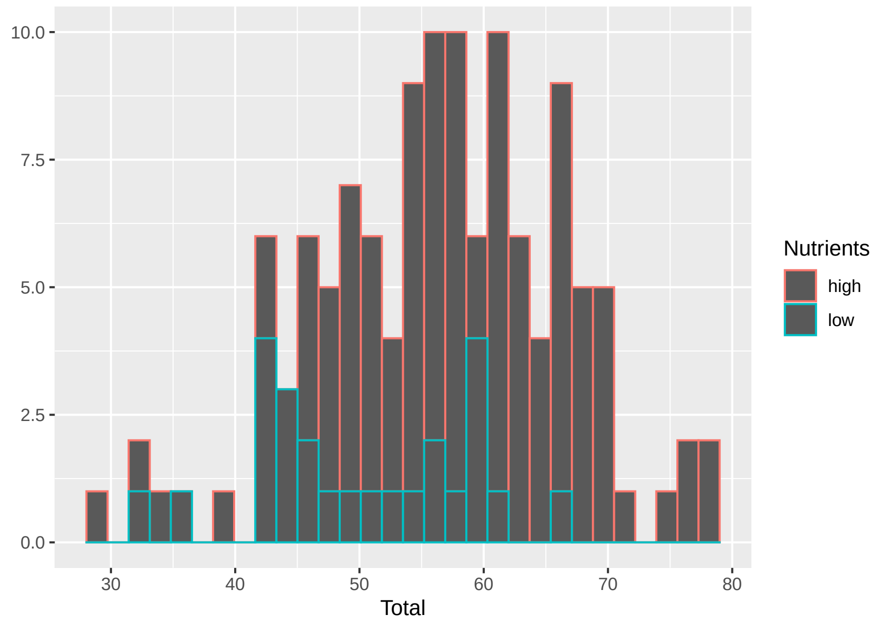


3.4.3.1 Histogram

Note what happens when we use the `colour` parameter for a histrogram

```
qplot(x=Total, data=MyData, group=Nutrients, colour=Nutrients)
```

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



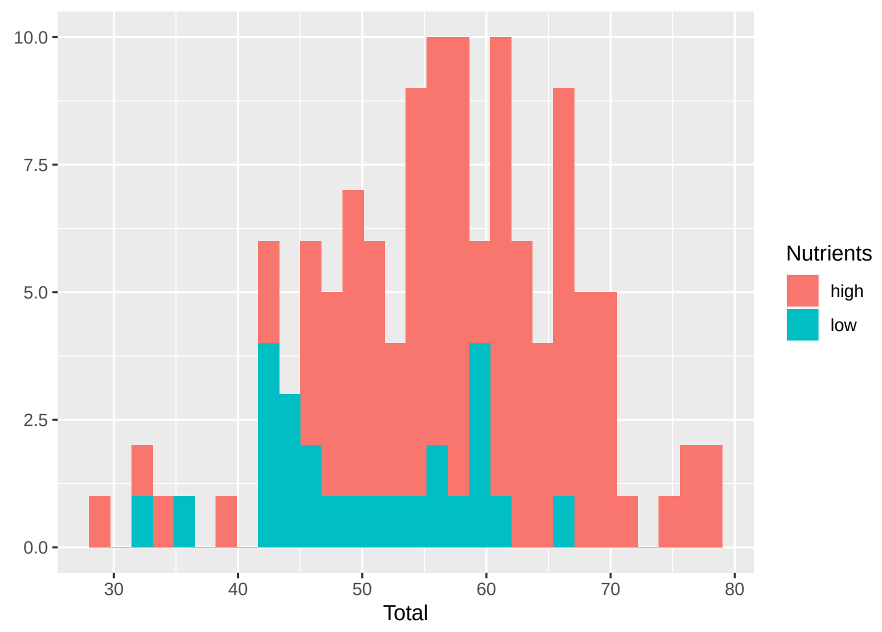
The coloured outlines might be useful but we usually will want the inside coloured.

3.4.4 `fill`

This parameter is used for boxes and other shapes that have a separate outline (`colour=`) and interior (`fill=`).

```
qplot(x=Total, data=MyData, group=Nutrients, fill=Nutrients)
```

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

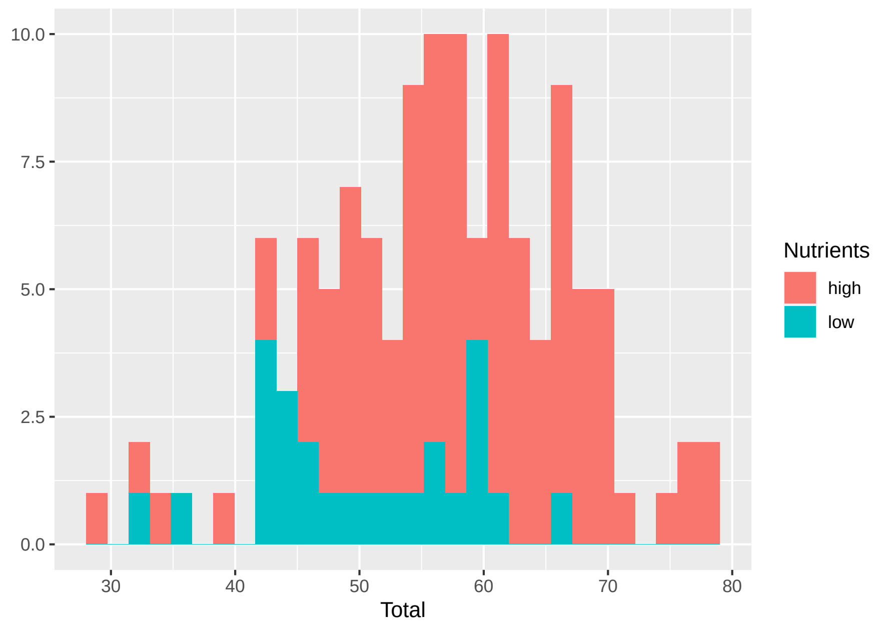


3.4.5 `posit`

Use this to adjust the position, usually for histograms or bar graphs. For example, if we want to compare histograms on the same graph:

```
qplot(x=Total, data=MyData, group=Nutrients, fill=Nutrients)
```

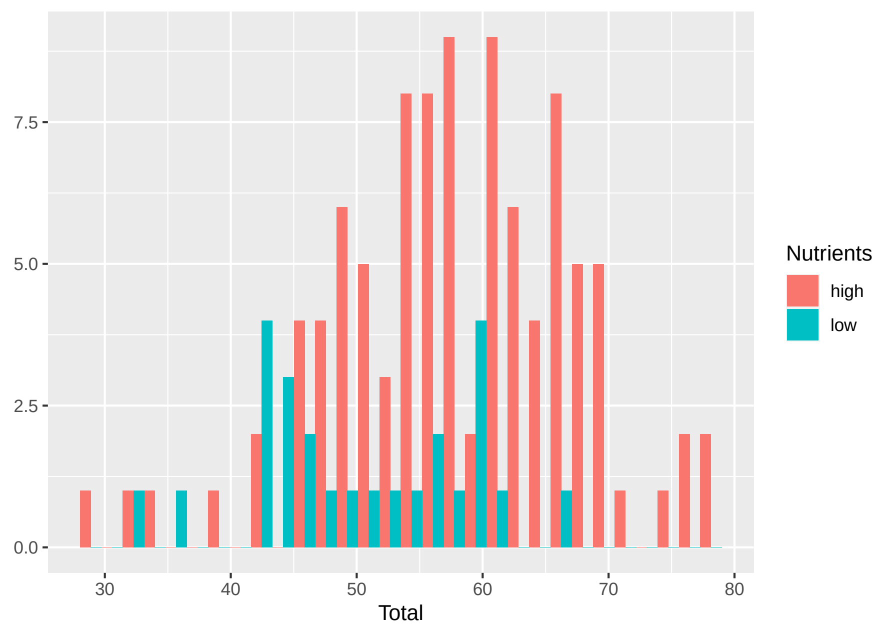
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



It's hard to interpret that graph, so we can shift the position using `dodge`.

```
qplot(x=Total, data=MyData, group=Nutrients, fill=Nutrients,
      posit="dodge")
```

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

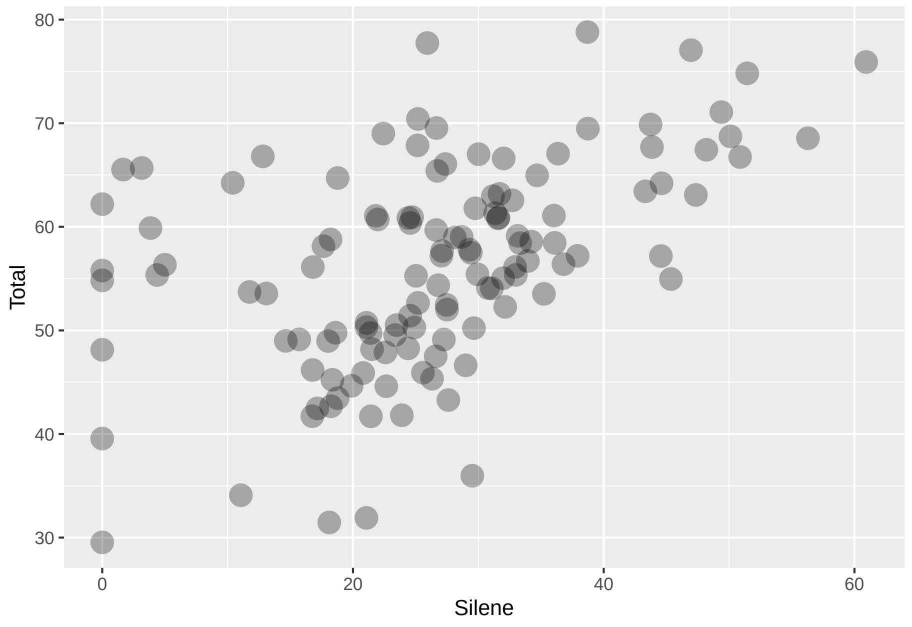


3.4.6 `alpha`

Think of `alpha` as a measure of opacity, ranging from 0 to 1 with 1 being the default, solid point or line and 0 being invisible.

This is particularly useful for visualizing overlapping points.

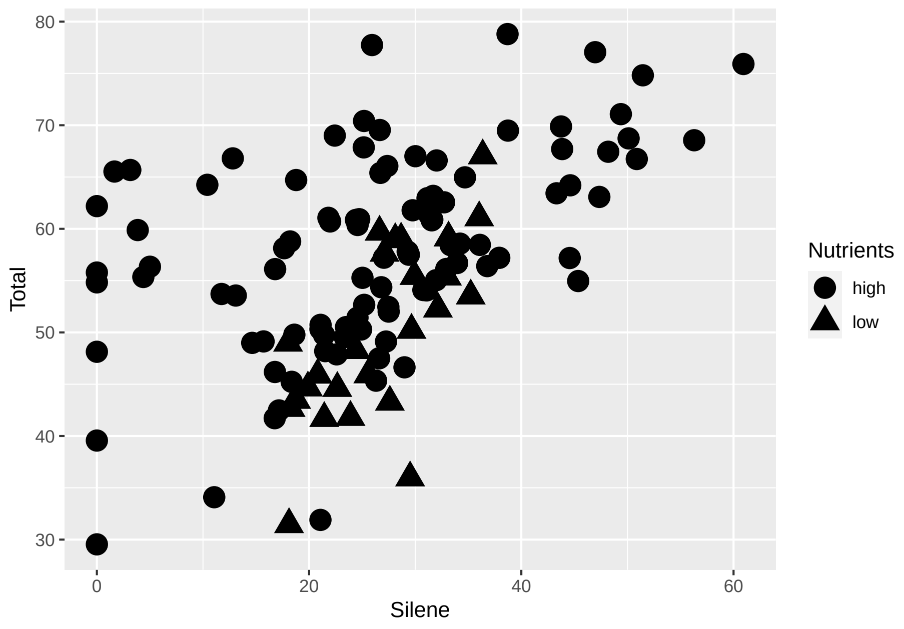
```
qplot(x=Silene, y=Total, data=MyData, size=I(5), alpha=I(0.3))
```



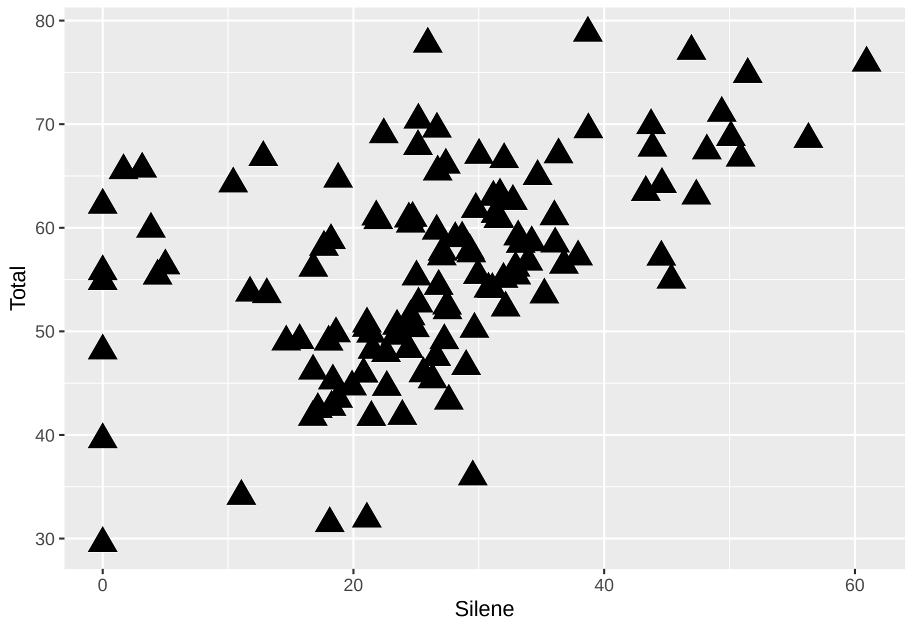
3.4.7 shape

You can also change the shape of your points, again using a column of data or the identity `I()` function.

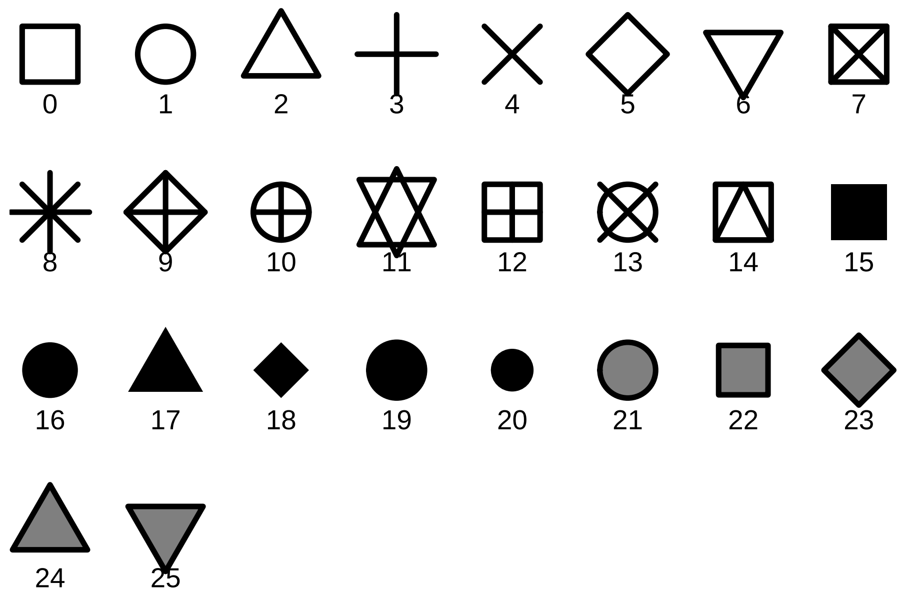
```
qplot(x=Silene, y=Total, data=MyData, size=I(5), shape=Nutrients)
```



```
qplot(x=Silene, y=Total, data=MyData, size=I(5), shape=I(17))
```



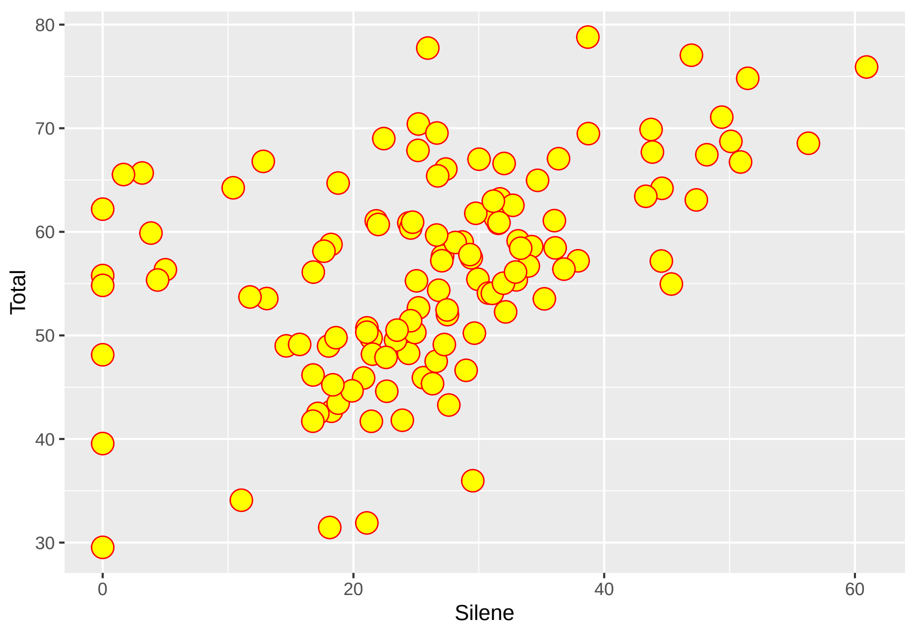
There are a number of different shapes available, by specifying a number from 0 through 25.



Note that the shapes with grey in the above figure can be coloured with `fill=` paremeter, while all of the black parts (lines and fill) can be coloured with the `colour=` parameter.

You can use `fill` and `colour` to customize these separately.

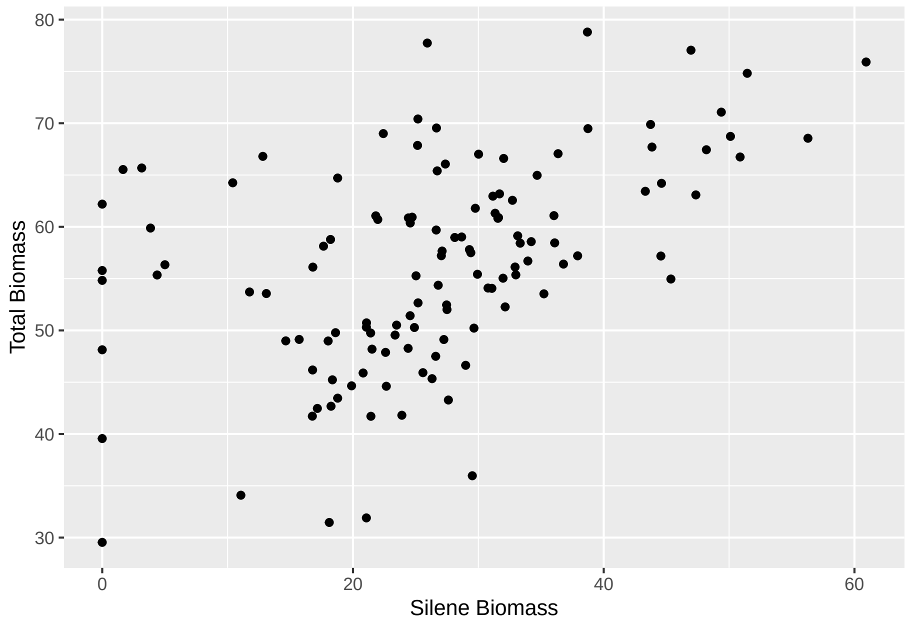
```
qplot(x=Silene, y=Total, data=MyData, size=I(5), shape=I(21),
      fill=I("yellow"), colour=I("red"))
```



3.4.8 `lab`, `xlab`, and `ylab`

Use these to customize your axis labels.

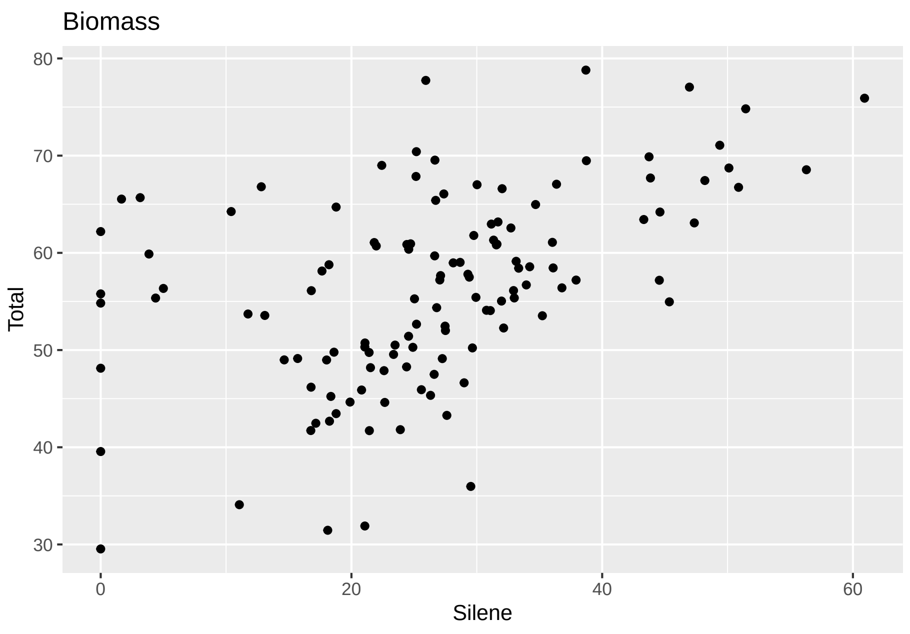
```
qplot(x=Silene, y=Total, data=MyData,
      xlab="Silene Biomass", ylab="Total Biomass")
```



3.4.9 `main`

This will add a title to your plot. Usually you wouldn't use this for a figure intended for publication, but this can be useful for reports, websites, presentations, supplementary material, appendices, etc.

```
qplot(x=Silene, y=Total, data=MyData, main="Biomass")
```



3.5 Themes and Geoms

Themes determine the look and 'feel' of your graphs, while Geoms determine the geometry – how your data are physically mapped to the graphing space. In both `qplot` and `ggplot`, themes are added with a separate function linked to the graph by using the plus sign `+`. Geoms are defined by the `geom=` paremeter inside the `qplot()` function or with a specific `geom_FUNCTION()` linked to the main `ggplot()` function by using the plus sign.

These are easy to understand with a few examples.

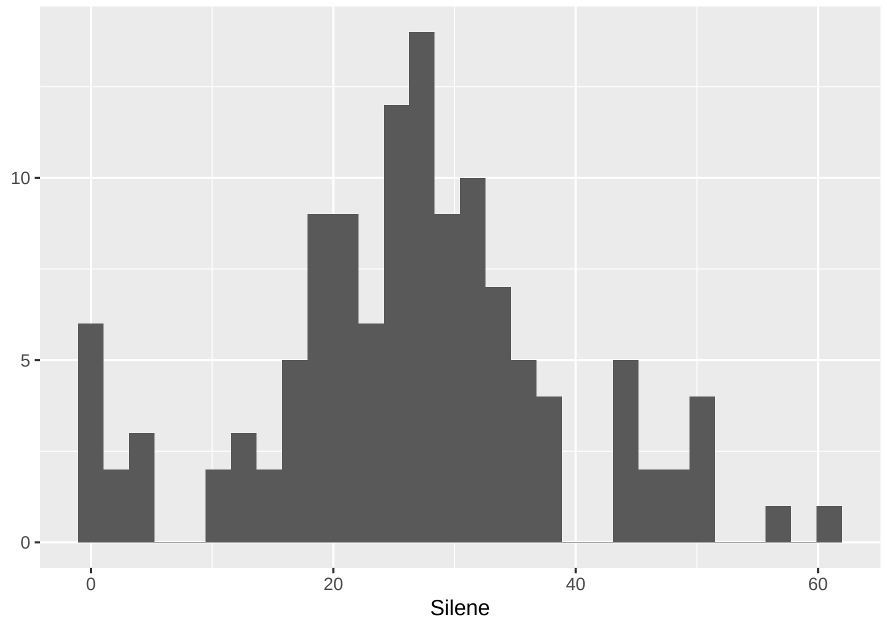
3.5.1 `+ theme_NAME()`

There are a number of available themes, defined by changing the *NAME* part.

Default theme:

```
qplot(x=Silene,data=MyData) + theme_grey()
```

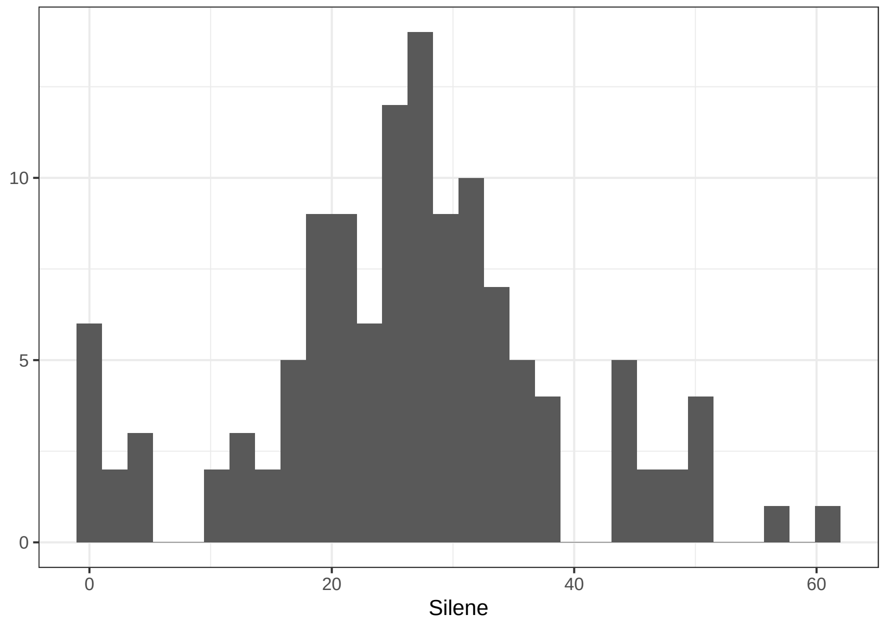
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



A cleaner theme with better contrast:

```r
qplot(x=Silene,data=MyData) + theme_bw()
```

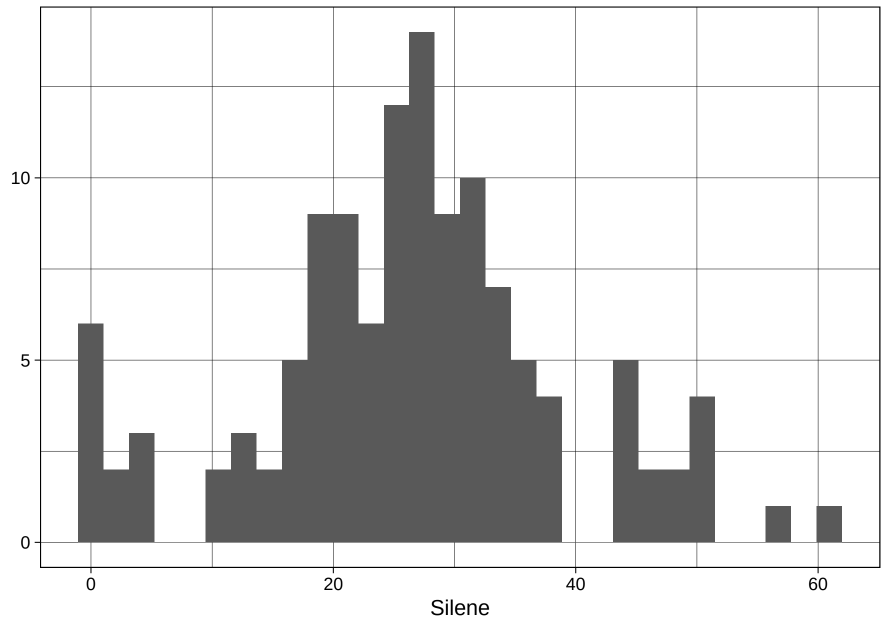
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Thicker grid lines:

```
qplot(x=Silene,data=MyData) + theme_linedraw()
```

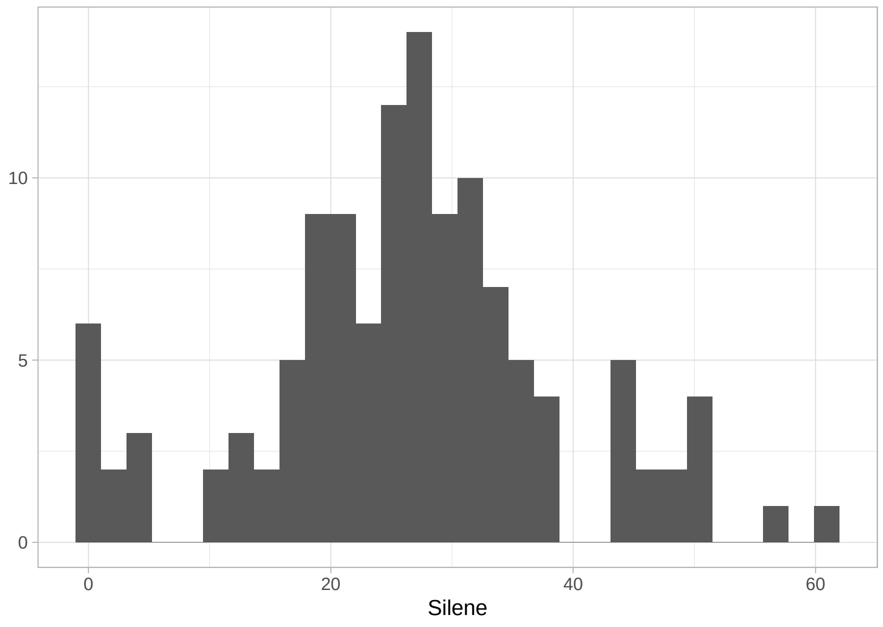
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Fainter border and axis values

```
qplot(x=Silene,data=MyData) + theme_light()
```

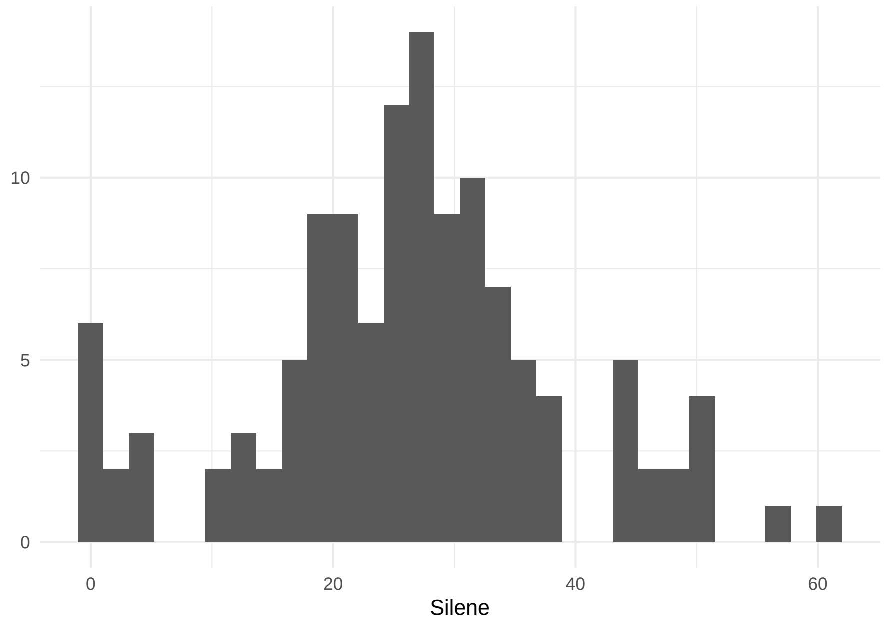
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



No borders at all

```
qplot(x=Silene,data=MyData) + theme_minimal()
```

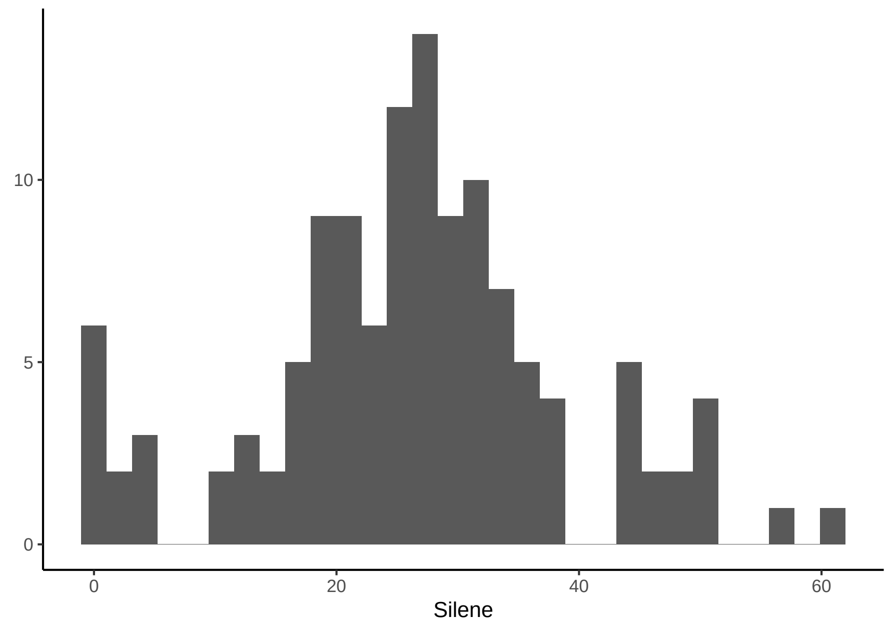
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



A minimal theme, closest to what you would see in a published paper

```
qplot(x=Silene,data=MyData) + theme_classic() # x and y lines only, no tick marks
```

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



These can be further customized. Or you can create a completely new theme.

3.5.2 Custom Theme

For example, here is a simplified and cleaner version of `theme_classic` but with bigger axis labels taht are more suitable for figures in presentation or publication:

```
# Clean theme for presentations & publications used in the Colautti Lab
theme_pub <- function (base_size = 12, base_family = "") {
  theme_classic(base_size = base_size, base_family = base_family) %+replace%
    theme(
      axis.text = element_text(colour = "black"),
      axis.title.x = element_text(size=18),
      axis.text.x = element_text(size=12),
      axis.title.y = element_text(size=18,angle=90),
      axis.text.y = element_text(size=12),
      axis.ticks = element_blank(),
      panel.background = element_rect(fill="white"),
      panel.border = element_blank(),
      plot.title=element_text(face="bold", size=24),
      legend.position="none"
    )
}
```

To use this theme, you could just copy-and-paste the above function into your R code.

Alternatively, you could save it as a separate `.R` file and thean load it with the `source()` function.

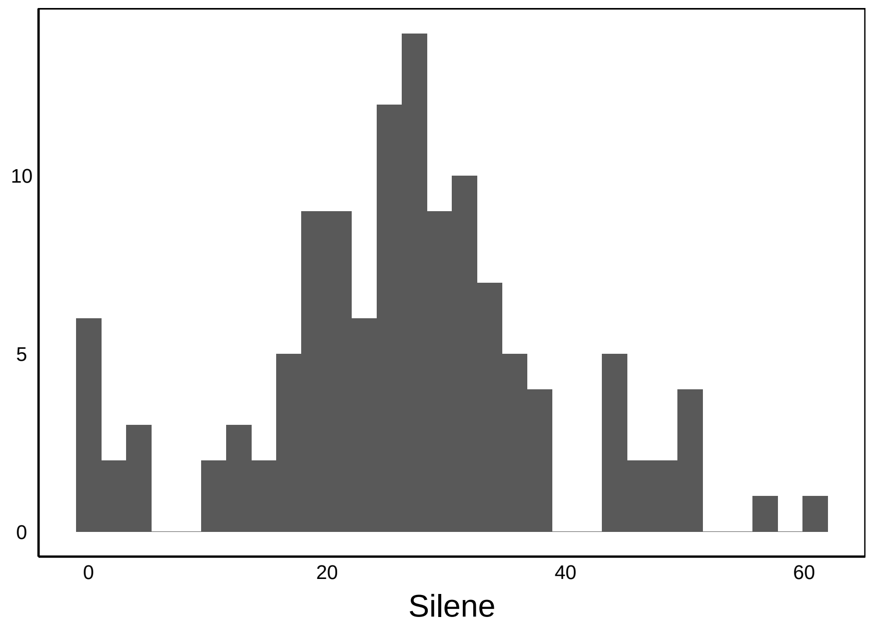
A third, even easier option, is to load the version of this code that is available online:

```
source("http://bit.ly/theme_pub")
```

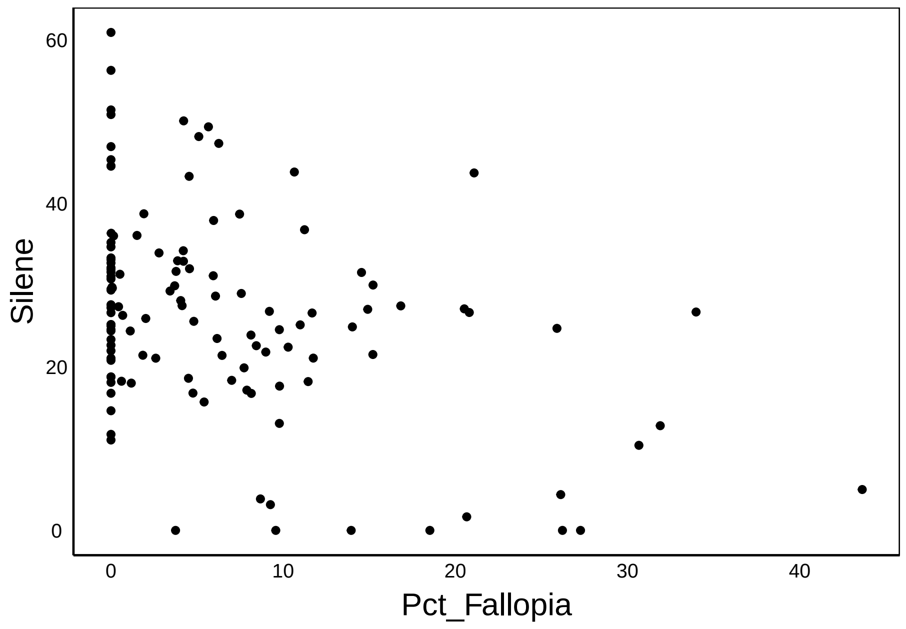
The theme is called `theme_pub` (pub is short for publication):

```
qplot(x=Silene,data=MyData) + theme_pub()
```

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



```
qplot(x=Pct_Fallopia,y=Silene,data=MyData) + theme_pub()
```

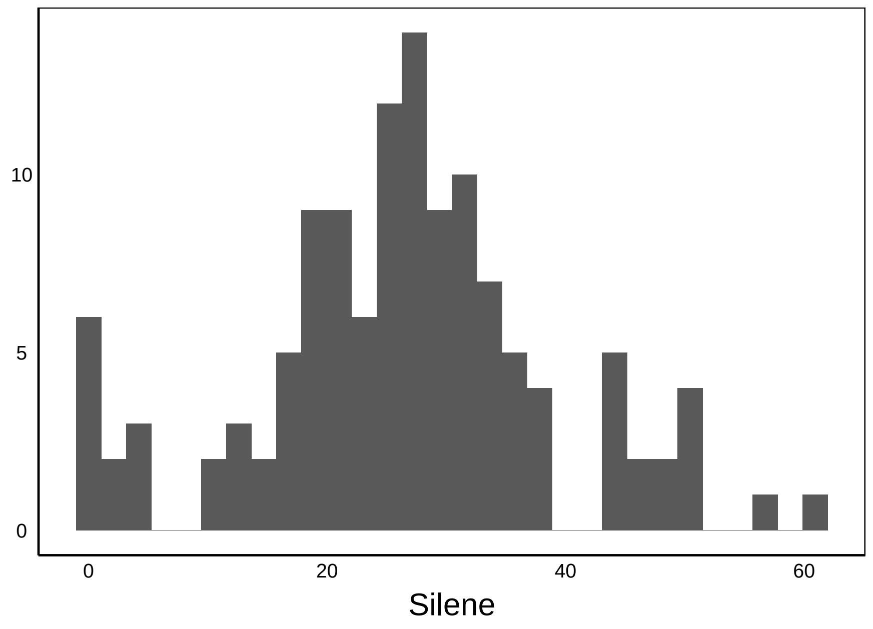


3.5.3 theme_set

If you want to use the same theme throughout your code, you can use this function.

```
theme_set(theme_pub())
qplot(x=Silene,data=MyData)
```

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

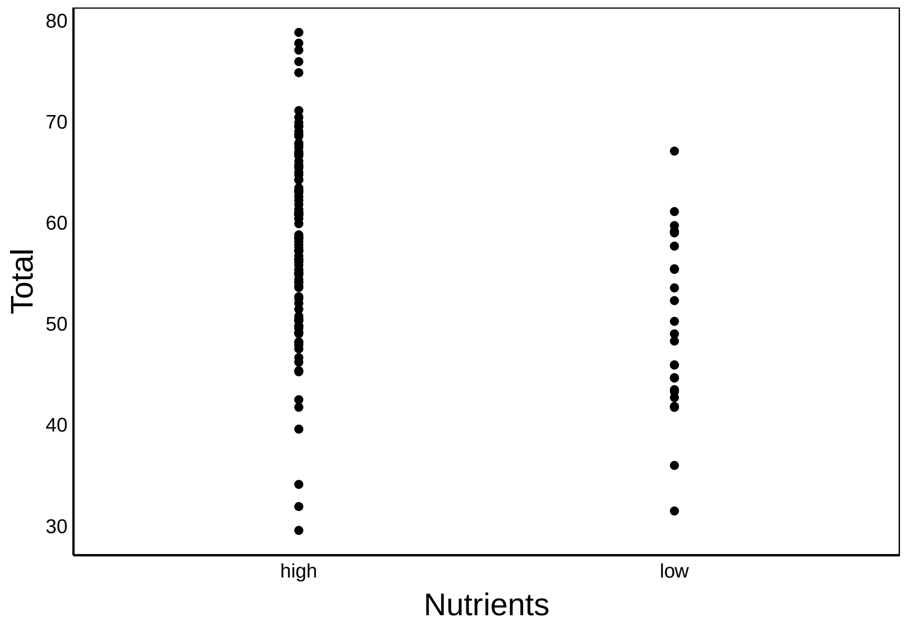


3.5.4 `geom`

In addition to the default genomes that `qplot` uses for different data types, there are some other options available.

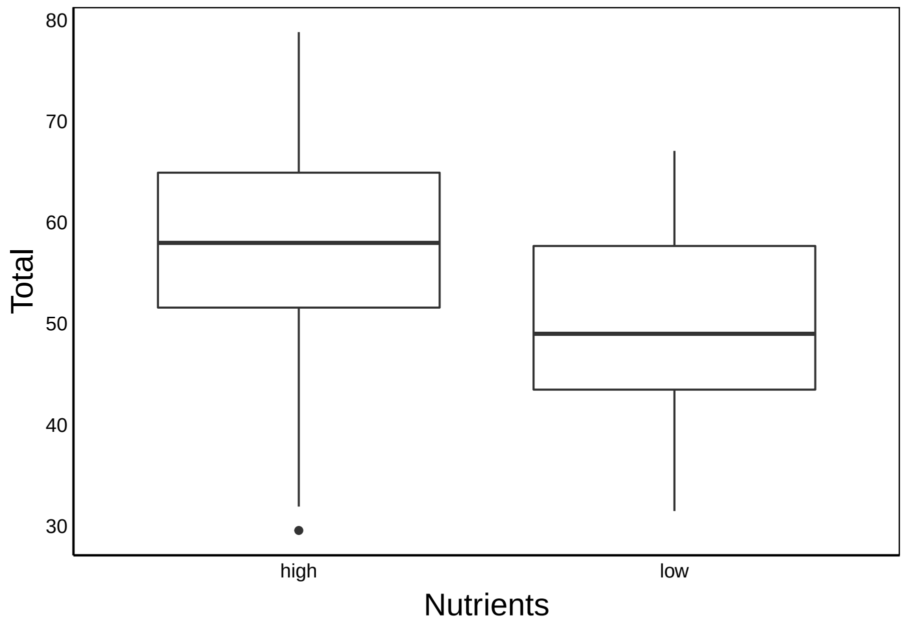
For example, the scatterplot doesn't give a good sense of the spread of the data.

```
qplot(x=Nutrients,y=Total,data=MyData)
```



But the `boxplot` geom shows the median (line), 1st and 3rd quartiles (boxes), range of most observations (whiskers), and potential outliers (dots)

```
qplot(x=Nutrients,y=Total,data=MyData,geom="boxplot")
```

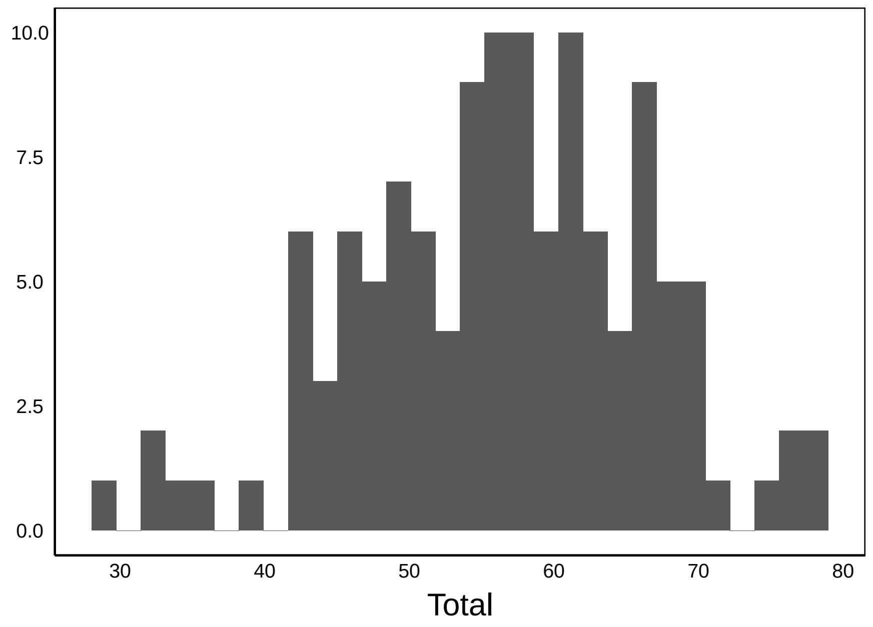


Another useful example is for histograms with many ‘bins’ which can be smoothed into a `density` curve rather than plotting bar graphs.

For example, the bar graph from above

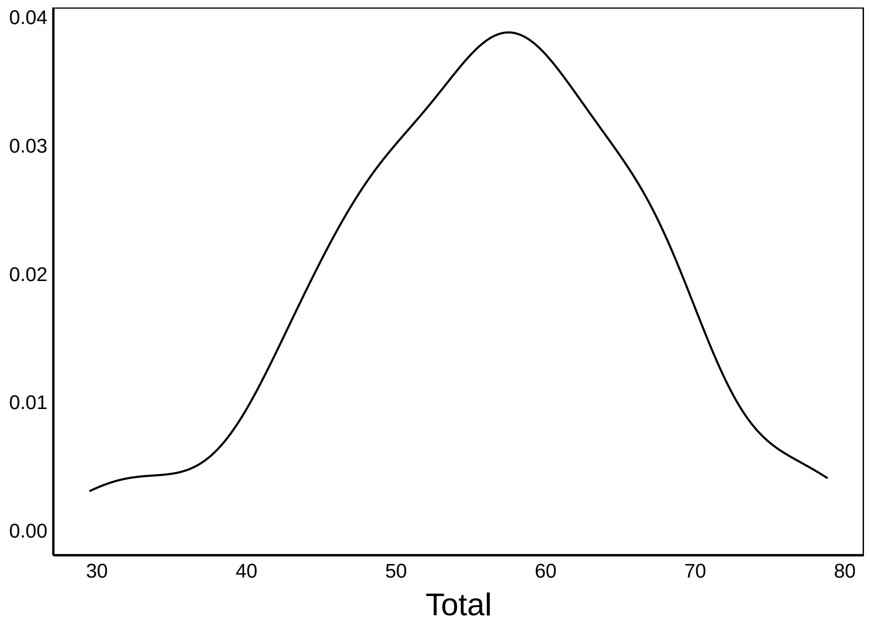
```
qplot(Total,data=MyData)
```

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



reformatted as a `density` geom.

```
qplot(Total,data=MyData,geom="density")
```



See the `ggplot2` website for a complete list of geoms with examples:

https://ggplot2.tidyverse.org/reference/

3.6 Multiple graphs

It is often handy to plot separate graphs for different categories of a grouping variable. This can be done with `facets` in `qplot`.

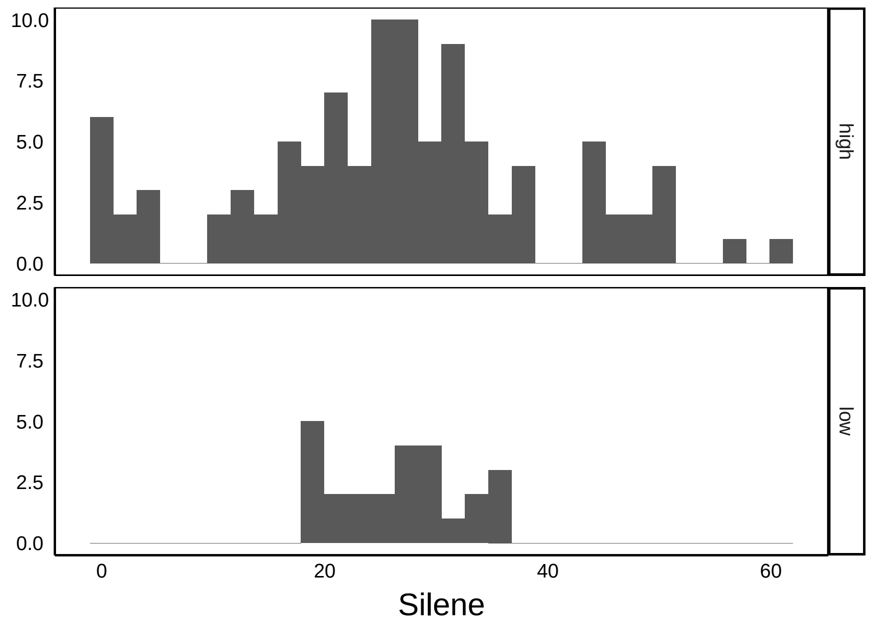
3.6.1 `facets`

Facets have the general form `VERTICAL ~ HORIZONTAL`. Use the period `.` to indicate ‘all data’ r ‘do not separate my data’.

Vertical stacking

```
qplot(x=Silene, data=MyData, facets=Nutrients~.)
```

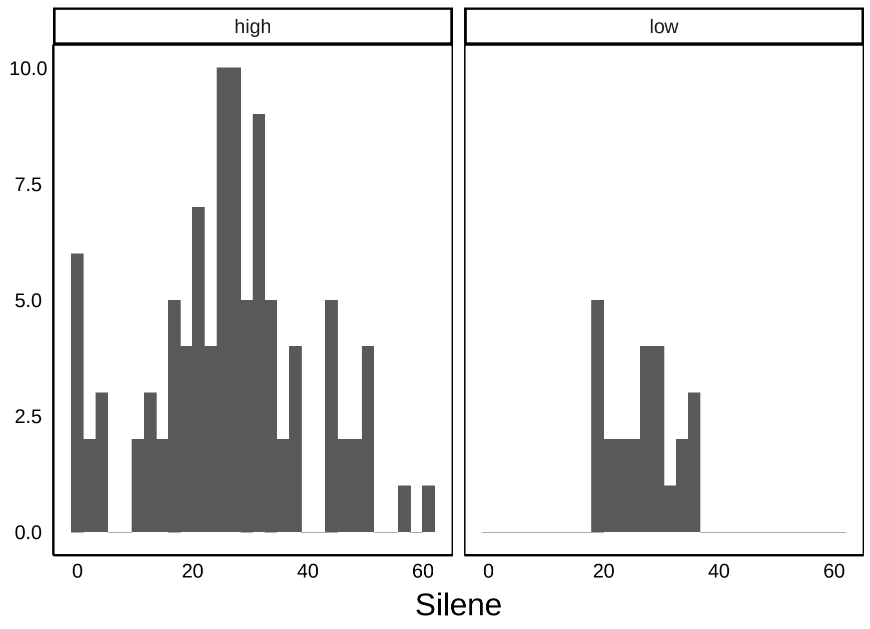
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Horizontal stacking

```
qplot(x=Silene, data=MyData, facets=.~Nutrients)
```

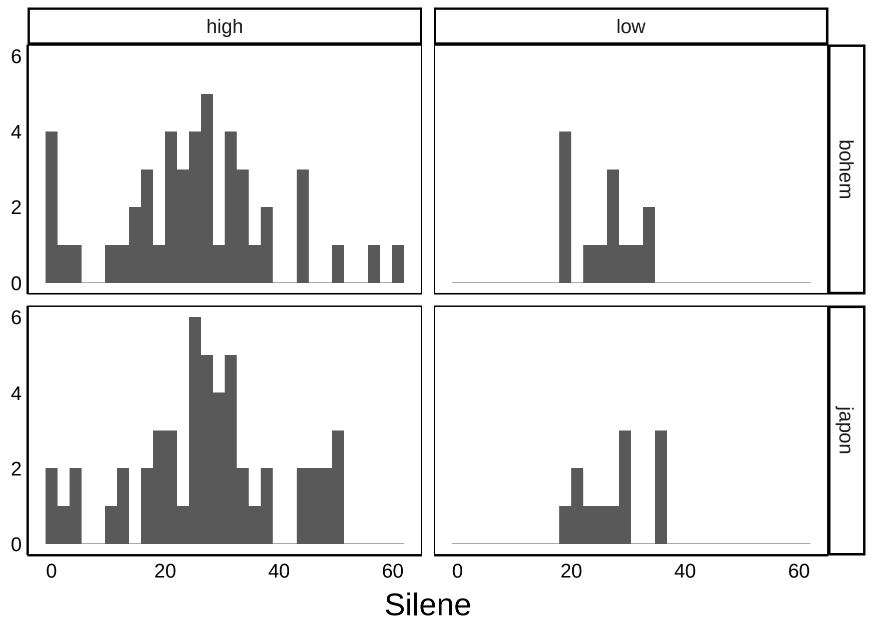
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Both (grid)

```r
qplot(x=Silene, data=MyData, facets=Taxon~Nutrients)
```

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



3.7 Save output

Saving your graphs to an external file requires three important steps:

1. Open a file using a function like `pdf` or `svg` for the **vector** format, or `png` for the **raster** format.
2. Run the code to produce the graph, just like above. However, instead of seeing a graph in your R interface, you will not see anything because the graph is being sent to the file.
3. Close the file! Do this with the `dev.off()` function.

Important: If you don't close the file, it is unusable.

Failing to close the file is a common source of error when generating graphs. If you are having problems with graphing outputs, try running the `dev.off()` function a few times to make sure you close any files that are 'hanging' open.

Here's an example code for making a `pdf` output of a graph.

```
pdf("SileneHist.pdf") # 1. Open
  qplot(x=Silene, data=MyData, facets=Taxon ~ Nutrients) # 2. Write
dev.off() # 3. Close
```

Note how the qplot command does not open in the plots window when you run this. This is because the info is sent to *SileneHist.pdf* file instead of the graphing area in your R console (e.g. *plots* tab in R Studio)

3.8 Practice

Graphing may seem slow and tedious at first, but the more you practice, the faster you will be able to produce meaningful visualizations.

Don't be afraid to try new things. Try mixing up components and see what happens. At worst you will just get an error message.

Once you have a good understanding of these basics, you can see how to build more advanced plots with `ggplot()`.

Chapter 4

Advanced Visualizations

4.1 Overview

Before following this tutorial, you should be familiar with the qplot() tutorial, and you should have lots of practice making graphs with different formatting options using `qplot`.

In this self-tutorial, we look at some more advanced options for visualizations by using the `ggplot` function. The 1`ggplot` function adds even more flexibiliity and possibilities to our visualizations.

The ggplot cheat sheet is a downloadable pdf that provides a good summary and quick-reference guide for advanced graphics.

4.2 Getting Started

We'll load the ggplot2 library and set a custom theme as described in the qplot Tutorial

```
library(ggplot2)
source("http://bit.ly/theme_pub")
theme_set(theme_pub())
```

The `source` function loads an external file, in this case from the internet. The file is just a .R file with a custom function defining different aspects of the graph (e.g. text size, line width, etc.) You can open the link in a web browser or download and open in a text editor to see the file.

The `theme_set()` command sets our custom theme (`theme_pub`) as the default plotting theme. Since the theme is a function in R, we need the extra brackets: `theme_pub()`

4.3 Rules of thumb

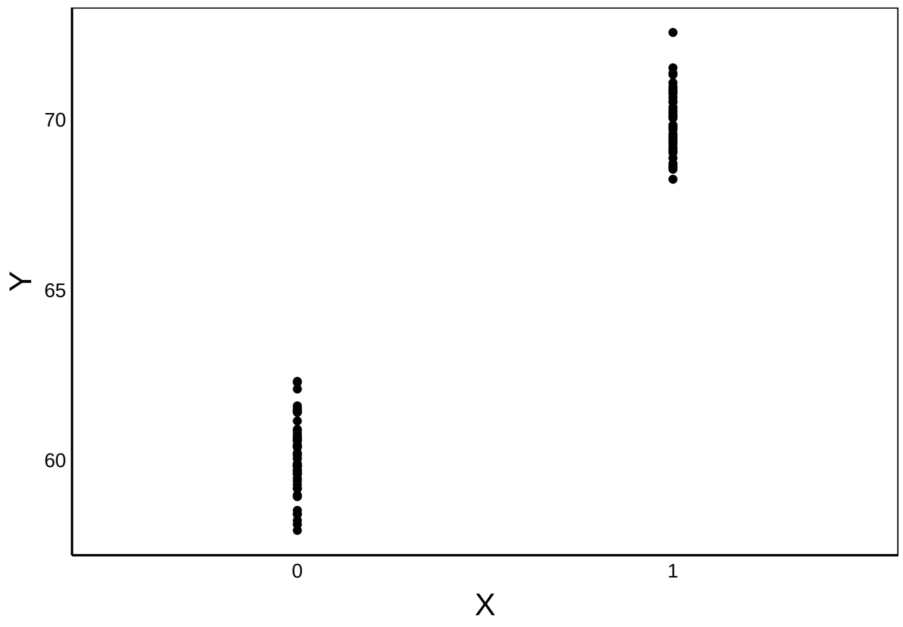
Standards of practice for published graphs in professional journals can vary depending on format and discipline, but there are a number of useful 'rules of thumb' to keep in mind. These are not hard and fast rules but helpful for new researchers who aren't sure how or where to start.

4.3.1 1. Minimize 'ink'

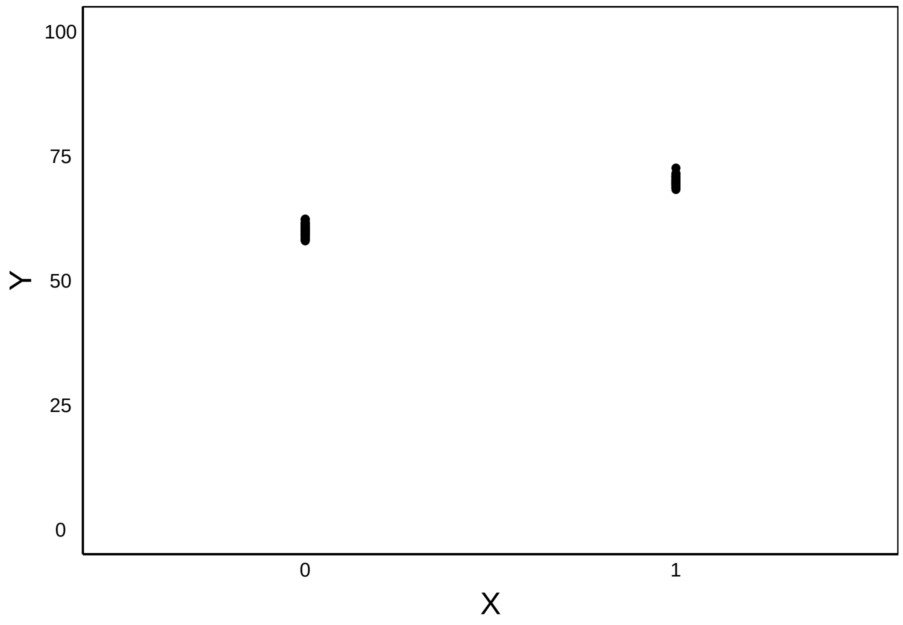
In the old days, when most papers were actually printed and mailed to journal subscribers, black ink was expensive and printing in colour was very expensive. Printing is still expensive but of course most research articles are available online where there is no additional cost to colour or extra ink. However, the concept of minimizing ink (or pixels) can go a long way toward keeping a graph free from clutter and unnecessary distraction.

4.3.2 2. Use space wisely

Empty space is not necessarily bad, but ask yourself if it is necessary and what you want the reader to take away. Consider the next two graphs:



Above: Y-axis scaled to the data



Above: Y-axis scaled between 0 and 100

What are the benefits/drawbacks of scaling the axes? When might you choose to use one over ther other?

4.3.3 3. Choose a colour palette

Colour has three basic components

a. Hue – the amount of red vs green vs blue light
b. Saturation – how vivid the colour is
c. Brightness – the amount of white (vs black) in the colour

In R these can be easily defined with the `rgb()` function. For example:

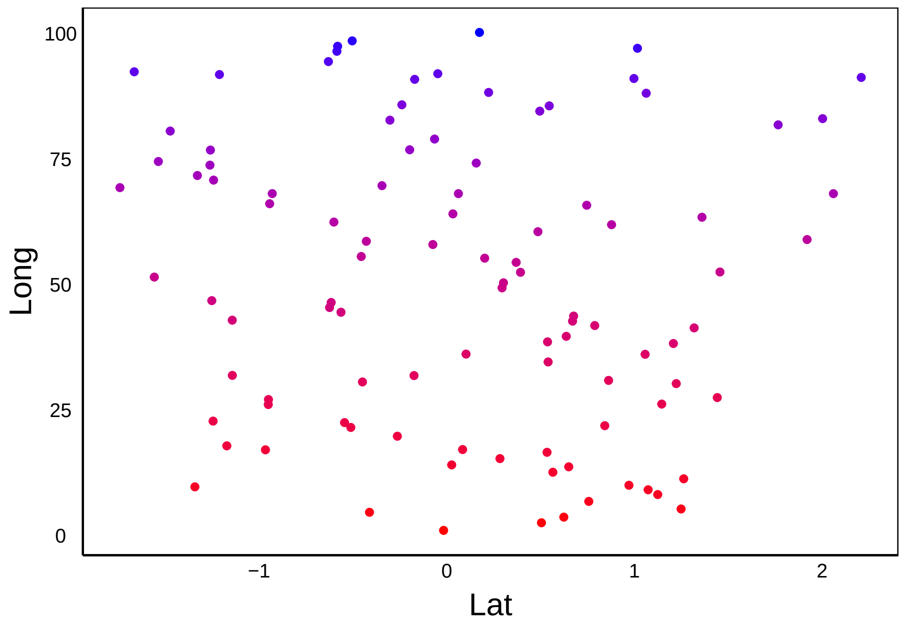
`rgb(1,0,0)` – a saturated red `rgb(0.1,0,0)` – a dark red (low brightness, low saturation) `rgb(1,0.9,0.9)` – a light red (high brightness, low saturation)

Don't underestimate the impact of choosing a good colour palette, especially for presentations. Colour theory can get a bit overwhelming but here are a few good websites to help:

- Quickly generate your own palette using coolors
- Use a colour wheel to find complementary colours using Adobe
- Browse some pre-made palettes or create one from a picture colorfavs

4.3.4 4. Colours have meaning

What's wrong with this graph?

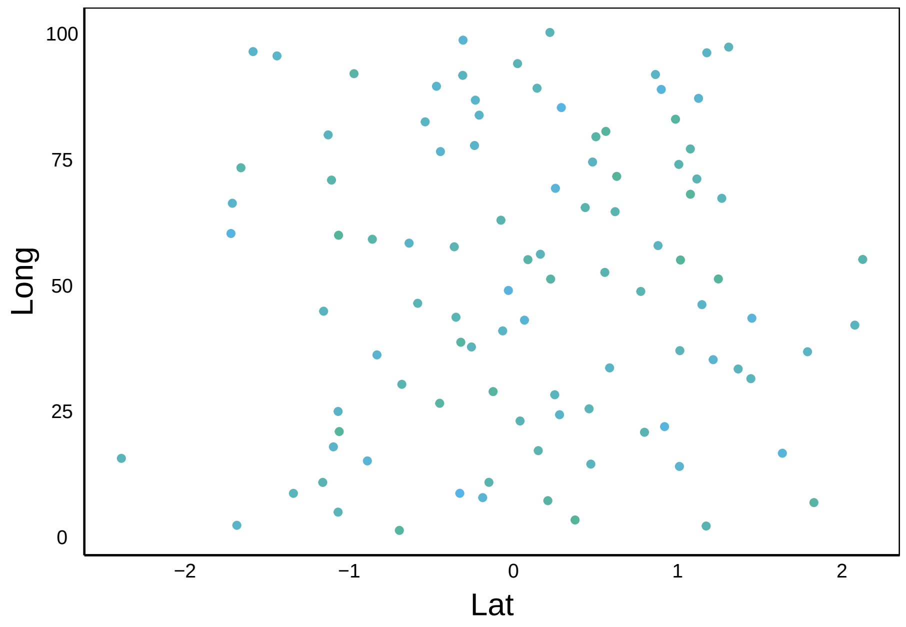


Humans naturally associate colours with particular feelings. Be mindful of these associations when choosing a colour palette

Another important consideration is that not everyone sees colour the same way. About 5% to 10% of the population has colour blindness. In order to make colour graphs readable to everyone, you can use different

4.3.5 5. Maximize contrast

Colours that are too similar will be hard to distinguish



4.3.6 6. Keep relevant information

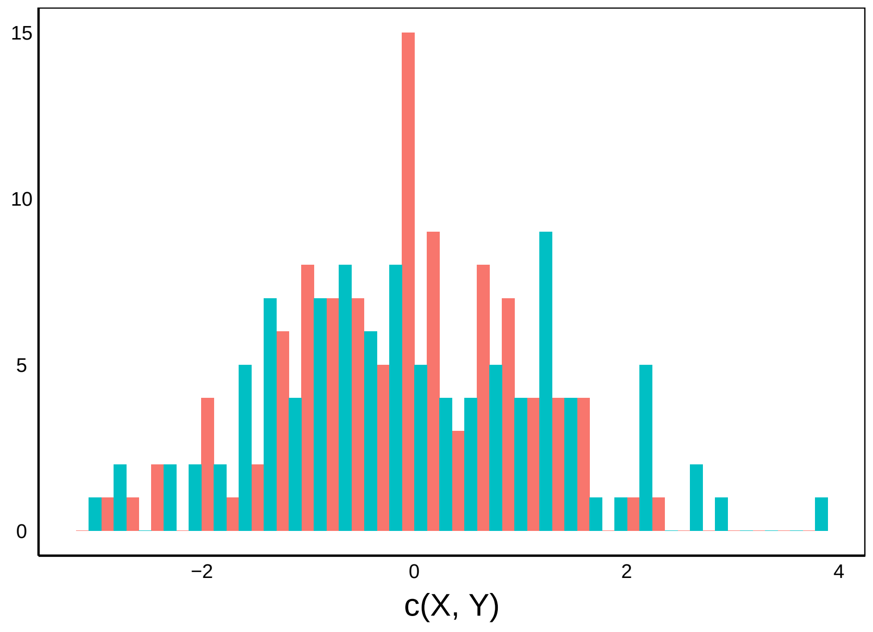
Make sure to include proper axis **labels** (i.e. names) and **tick marks** (i.e. numbers or categories showing the different values). These labels, along with the figure caption, should act as a stand-alone unit. The reader should be able to understand the figure without having to read through the rest of the paper.

4.3.7 7. Choose the right graph

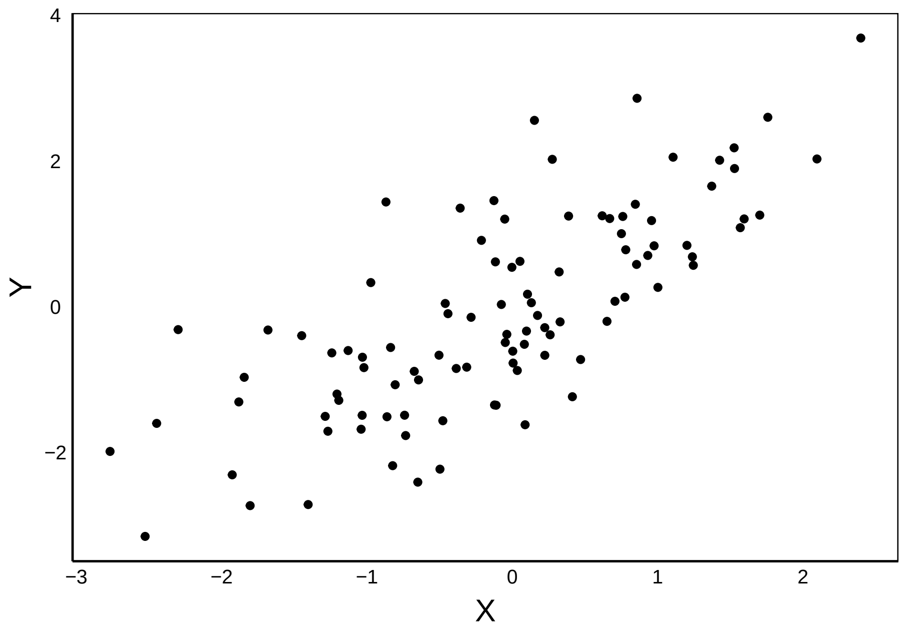
Often the same data can be presented in different ways but some are easier to interpret than others. Think carefully about the story you want to present and the main ideas you want your reader to get from your figures. Look at these two graphs that show the same data and see which one is more intuitive

```r
X<-rnorm(100)
Y<-X+rnorm(100)
qplot(c(X,Y), fill=c(rep("X",100), rep("Y",100)), posit="dodge")
```

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



```
qplot(X,Y)
```



4.4 Example

To get to know ggplot better, let's do a step-by-step example of a figure published in a paper by Colautti & Lau in the journal Molecular Ecology (2015)

4.4.1 Setup

4.4.1.1 Import data

Download selection dataset from Colautti & Lau (2015), published in the journal **Molecular Ecology**: https://doi.org/10.1111/mec.13162

The paper is a meta-analysis and review of evolution occurring during biological invasions. in this tutorial, we'll recreate Figure 2, which shows the result of a meta-analysis of selection gradients (β) and selection differentials (s). First, we'll just recreate the top panel, and then we'll look at ways to make more advanced multi-panel graphs like this one.

The data from the paper are archived on Dryad: https://datadryad.org/stash/dataset/doi:10.5061/dryad.gt678

You could download the zip file and look for the file called `Selection_Data.csv` and save it to your working directory.

Or you can just download directly from this website:

```r
SelData<-read.csv(
  "https://colauttilab.github.io/RCrashCourse/Selection_Data.csv")
```

We are also going to change the names from the file to make them a bit more intuitive and easier to work with in R.

```r
names(SelData)<-c("Collector", "Author", "Year", "Journal",
                  "Vol", "Species", "Native", "N",
                  "Fitness.measure", "Trait", "s",
                  "s.SE", "s.P", "B", "B.SE", "B.P")
```

4.4.2 Inspect

Let's take a quick look at the data

```r
head(SelData)
```

```
##            Collector                Author Year            Journal
## 1 KingsolverDiamond Alatalo and Lundberg 1986          Evolution
## 2 KingsolverDiamond Alatalo and Lundberg 1986          Evolution
## 3 KingsolverDiamond Alatalo and Lundberg 1986          Evolution
## 4 KingsolverDiamond       Alatalo et al. 1990 American Naturalist
## 5 KingsolverDiamond       Alatalo et al. 1990 American Naturalist
## 6 KingsolverDiamond       Alatalo et al. 1990 American Naturalist
##             Vol            Species Native    N       Fitness.measure
## 1     40:574-583  Ficedula hypoleuca    yes  641   male mating success
## 2     40:574-583  Ficedula hypoleuca    yes  713 female mating success
## 3     40:574-583  Ficedula hypoleuca    yes 1705              survival
## 4 135(3):464-471 Ficedula albicollis    yes <NA>              survival
## 5 135(3):464-471 Ficedula albicollis    yes <NA>              survival
## 6 135(3):464-471 Ficedula albicollis    yes <NA>              survival
##           Trait     s s.SE s.P     B B.SE B.P
## 1 tarsus length -0.01       ns    NA
## 2 tarsus length  0.01      sig    NA
## 3 tarsus length  0.04       ns    NA
## 4  tarus length  0.02       ns -0.06
## 5  tarus length  0.08       ns -0.01
## 6  tarus length  0.19      sig  0.01
```

It's worth taking some time to look at this to understand how a meta-analysis works. The **collector** column indicates the paper that the data came from. The **Author** indicates the author(s) of the original paper that reported the data. The **Year**, **Journal**, and **Vol** give information about the publication that the data came from originally.

We can see above the collector `KingsolverDiamond`, which represents a paper from Kingsolver and Diamond that was itself a meta-analysis of natural selection. Most of the studies came from this meta-analysis, but a few of the more recent papers were added by grad students, denoted by initials:

```
unique(SelData$Collector)
```

```
## [1] "KingsolverDiamond" "JAL"               "DJW"
## [4] "CPT"
```

Species is the study species, and **Native** is its stauts as a binary yes/no variable. **N** is the sample size and **Fitness.measure** is the specific trait that defines fitness. **Trait** is the trait on which selection was measured. Finally, s is the **selection differential** and β is the **selection gradient**. Note that these are slopes in units of relative fitness per trait standard deviation. This is explained in more detail below.

4.4.3 Absolute Value

In this analysis, we are interested in the magnitude but not the direction of natural selection. In other words we would want to treat a slope of -4 the same as a slope of +4 because they have the same magnitude. Therefore, we can replace the s column with $|s|$

```
SelData$s<-abs(SelData$s)
```

We'll also add a couple of columns with random variables that we can use later to explore additional plotting options.

First, a column of values sampled from a z-distribution (Gaussian distribution with mean = 0 and sd = 1)

```
SelData$Rpoint<-rnorm(nrow(SelData))
```

Second, a columnn of 1 and 0 sampled randomly with equal frequency (p = 0.5)

```
SelData$Rgroup<-sample(c(0,1), nrow(SelData), replace=T)
```

4.4.4 Missing values

Note the missing data (denoted NA)

```
print(SelData$s)
```

We can subset to remove mising data

```
SelData<-SelData[!is.na(SelData$s),]
```

Recall from the R Fundamentals Tutorial that `!` means 'not' or 'invert'

Alternatively, we could use `filter` from dplyr

```
library(dplyr)
SelData<-SelData %>%
  filter(!is.na(s))
```

There is also has a convenient `drop_na` function in the `tidyr` package

```
library(tidyr)
SelData<-SelData %>%
  drop_na(s)
```

4.5 Measuring Selection

A simple analysis of phenotypic selection as proposed by Lande & Arnold (1983) is just a linear model with **relative fitness** on the y-axis and the **standardized trait value** on the x-axis.

4.5.1 Relative Fitness

Fitness can be measured in many ways, such as survival or lifetime seed or egg production. Use `unique(SelData$Fitness.measure)` to see the list of specific fitness measures used in these studies.

relative fitness is just the observed or **absolute fitness** divided by the mean. **Absolute fitness** is usually denoted by the capital letter W and **relative fitness** is usually represented by a lower-case w or omega ω. Expressing this in mathematical terms:

$$\omega = W_i/\bar{W}$$

4.5.2 Trait Value

A **Trait Value** is literally just the measured trait. Use `unique(SelData$Trait)` to see the list of specific traits that were measured in these studies. The **Standardized Trait Value** is the traits z-score. See the Distributions Tutorial for more information about z-scores. Since traits have different metrics, they are hard to compare: e.g. days to flower, egg biomass, foraging intensity, aggression. But standardizing traits to z-scores puts them all on the same scale for comparison. Specifically, the scale is in standard deviations from the mean.

4.5.3 s vs β

Selection differentials (s) and selection gradients (β) measure selection using linear models but represent slightly different measurements. Linear models are covered in the Linear Models Tutorial.

Both models use relative fitness (ω) as the response variable (Y).

Selection differentials (s) measure selection on only a single trait, ignoring all other traits. In theory, the response to selection is a simple function of the genetic correlation between a trait and fitness. Some of this theory, with examples in R, is covered in the Population Genetics Tutorials.

Fitness differences among individuals can depend on a lot of things – genetic variation for the trait itself, but also environmental effects on the trait as well as effects on other traits that are under selection and correlated with the trait of interest.

Selection gradients (β) measure selection on a trait of interest while also accounting for selection on other correlated traits. This is done via a **multiple regression** – a linear model with multiple predictors.

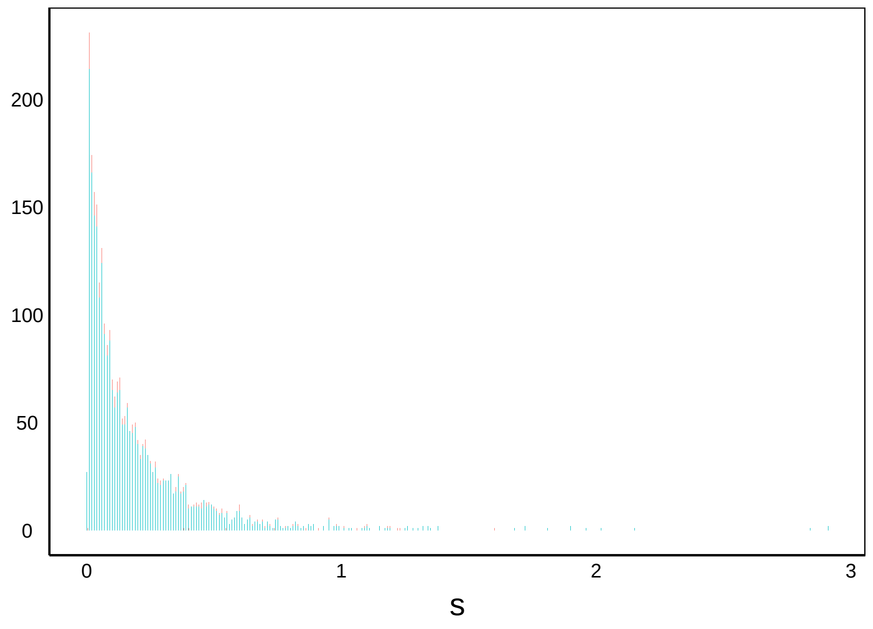
4.6 `ggplot` vs `qplot`

We can create the same graph using qplot and ggplot, just the syntax changes.

4.6.1 Histogram

Compare this `qplot`

```
BarPlot<-qplot(s, data=SelData, fill=Native, geom="bar")
print(BarPlot)
```



with this `ggplot`

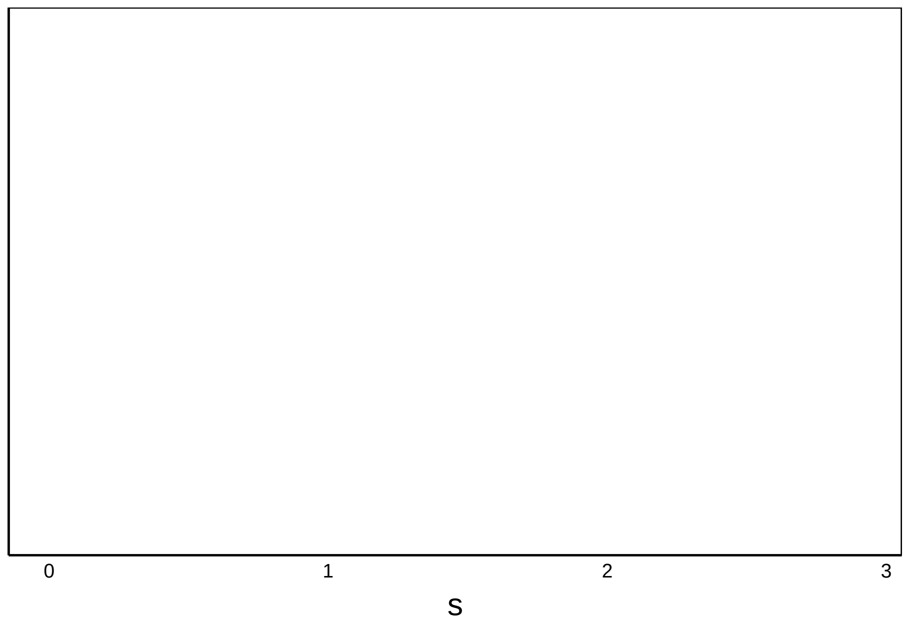
```
BarPlot <-ggplot(aes(s, fill=Native), data=SelData)
```

4.6.2 aes

Note the use of the aesthetic funciton `aes()`. This defines the data that we want to use for our `ggplot` graph. We will see how we do this by adding layers to our plot, similar to the way old-timey cartoons were made by layering multiple clear pages of cellphane with painting on them. The `aes` function inside of the `ggplot` function defines that data that will be shared among all of the layers. In addition, we can have separate `aes` functions inside different `geom_` layers that define and restrict the plotting data to that specific layer.

Let's look at the `ggplot` object so far:

```
print(BarPlot)
```



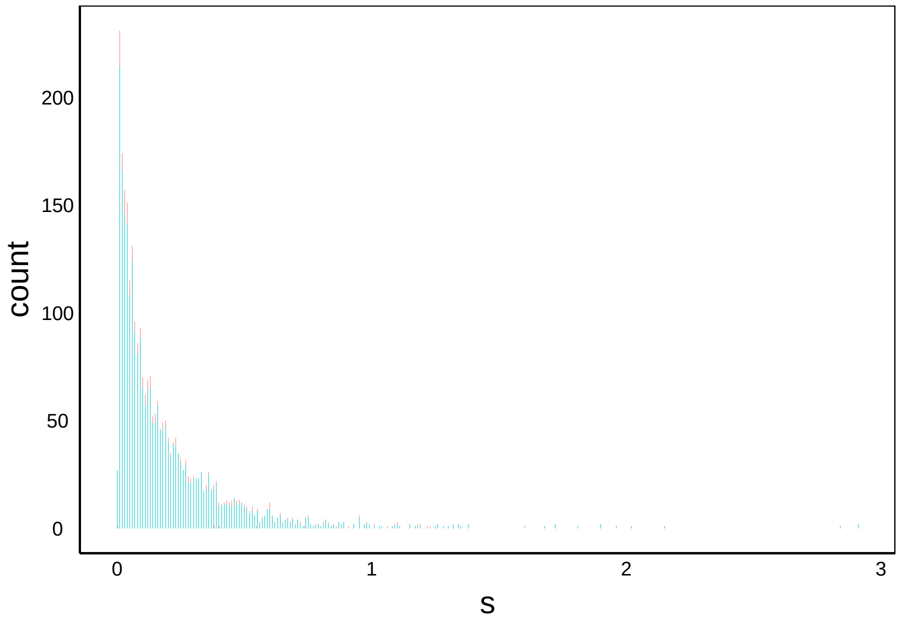
No data! This is one key difference between `qplot` and `ggplot`. The former has default `geoms` that it applies depending on the type of input data. We can modify geoms with the `geom='NAME'` parameter in `qplot`, where 'NAME' is the specific name of the geom we want to use.

In `ggplot` we have to explictly define the geoms using `geom_NAME` where name is the specific name of the geom – the same text that would go in quotation marks for the geom in `qplot`. The advantage of using `ggplot` is that its easier to creat multiple, overlapping layers with different geoms from types data sources.

4.6.3 Layers

So far, we have only loaded in the data info for plotting. We have to specify which geom(s) we want.

```
BarPlot<- BarPlot + geom_bar()
BarPlot
```



Explore the components of our BarPlot object:

```
summary(BarPlot)
```

```
## data: Collector, Author, Year, Journal, Vol, Species, Native, N,
##   Fitness.measure, Trait, s, s.SE, s.P, B, B.SE, B.P, Rpoint, Rgroup
##   [2766x18]
## mapping:  x = ~s, fill = ~Native
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##     compute_layout: function
##     draw_back: function
##     draw_front: function
##     draw_labels: function
##     draw_panels: function
##     finish_data: function
##     init_scales: function
##     map_data: function
##     params: list
##     setup_data: function
##     setup_params: function
##     shrink: TRUE
##     train_scales: function
##     vars: function
##     super:  <ggproto object: Class FacetNull, Facet, gg>
## -----------------------------------
## geom_bar: width = NULL, na.rm = FALSE, orientation = NA
## stat_count: width = NULL, na.rm = FALSE, orientation = NA
## position_stack
```

This shows us the columns of data available, the mapping for our x and y axes, and our fill colours. It also shows some of the functions and parameters used to generate the graph. At the bottom we see parameters for `geom_bar` and `stat_count`. Note that there are more parameters than the ones we defined. These are the **default parameters**.

4.6.4 `stat_NAME`

If geoms are the geometry of the shapes in the plt, stats are the statistics or mathematical functions that create the geoms. In the above case, the bars in `geom_bar` are created by counting the number of observations in each bin. The `stat_count` function is responsible for this calculation, and it is called by default when we use the `geom_bar` function.

Just as we can change the geometry of the plotted shapes with `geom_NAME`, we can define different functions for generating shapes with `stat_NAME`. Luckily, there is a default stat for each geom, so we don't have to choose it unless we want something other than the default.

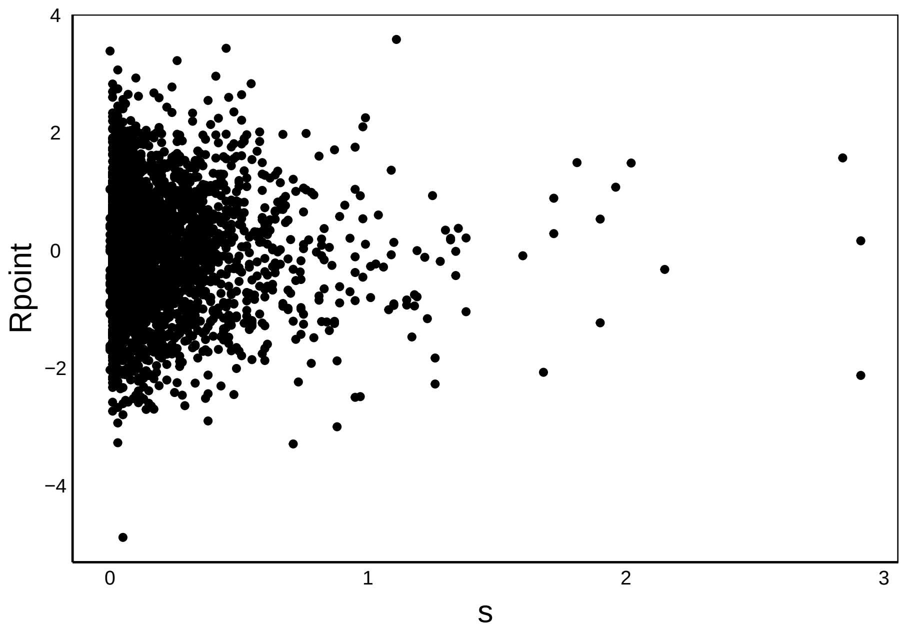
For more information on the parameters and stast of `geom_bar()`

```
?geom_bar
```

4.6.5 Bivariate geom

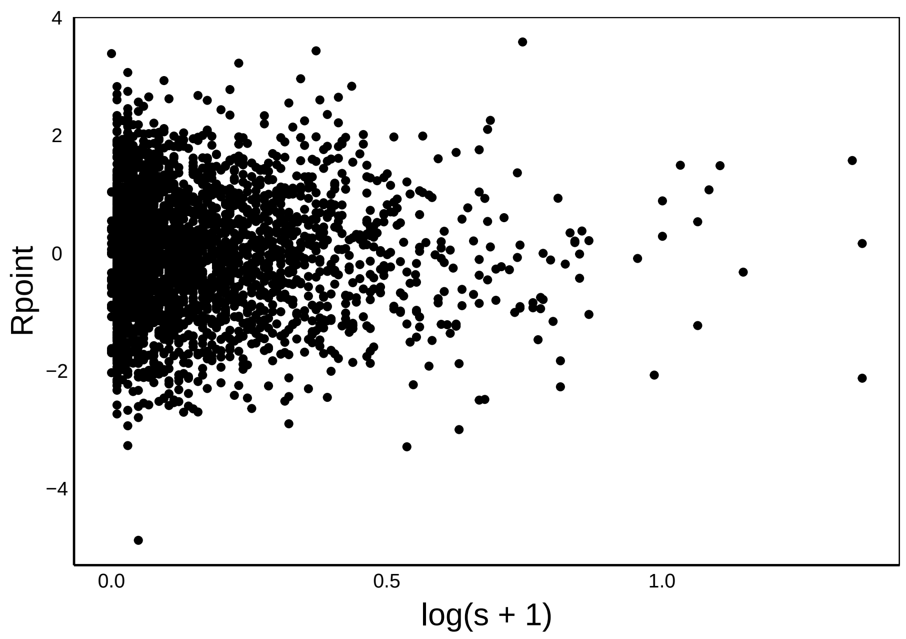
Let's explroe a few more plotting options. Here we'll use the random normal values we generated above so that we can make a bivariate plot:

```
BivPlot<-ggplot(data=SelData, aes(x=s, y=Rpoint)) +
  geom_point()
print(BivPlot)
```



Notice how the points are all clustered to the left. This looks like a classic log-normal variable, so let's log-transform *s* (x-axis)

```
BivPlot<-ggplot(data=SelData, aes(x=log(s+1), y=Rpoint)) +
  geom_point()
print(BivPlot)
```



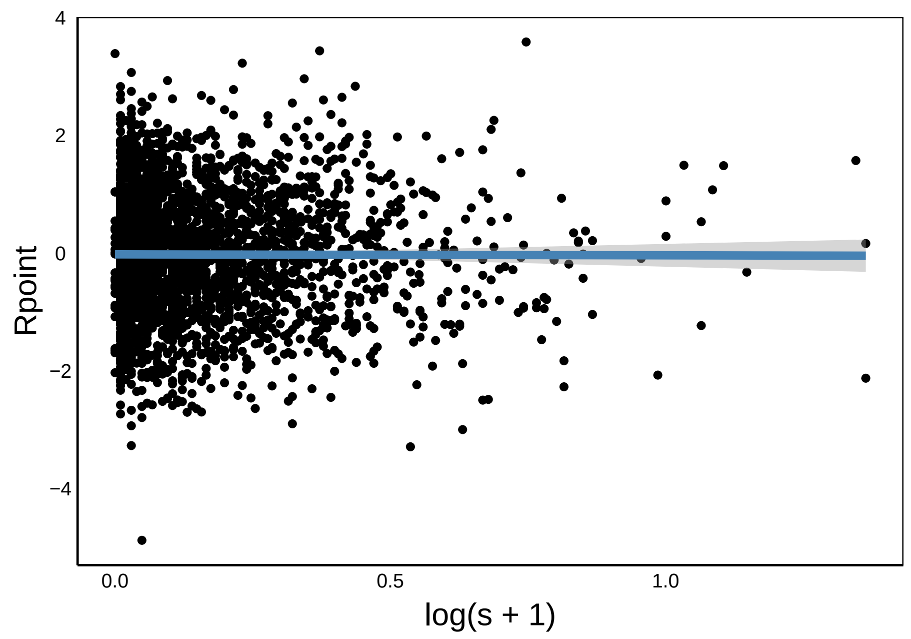
4.6.6 `geom_smooth`

A really handy feature of `ggplot` is the `geom_smooth` function with several options for calculating and plotting a statistical model to the observations.

Here's a simple linear regression slope:

```
BivPlot +
  geom_smooth(method="lm", colour="steelblue", size=2)
```

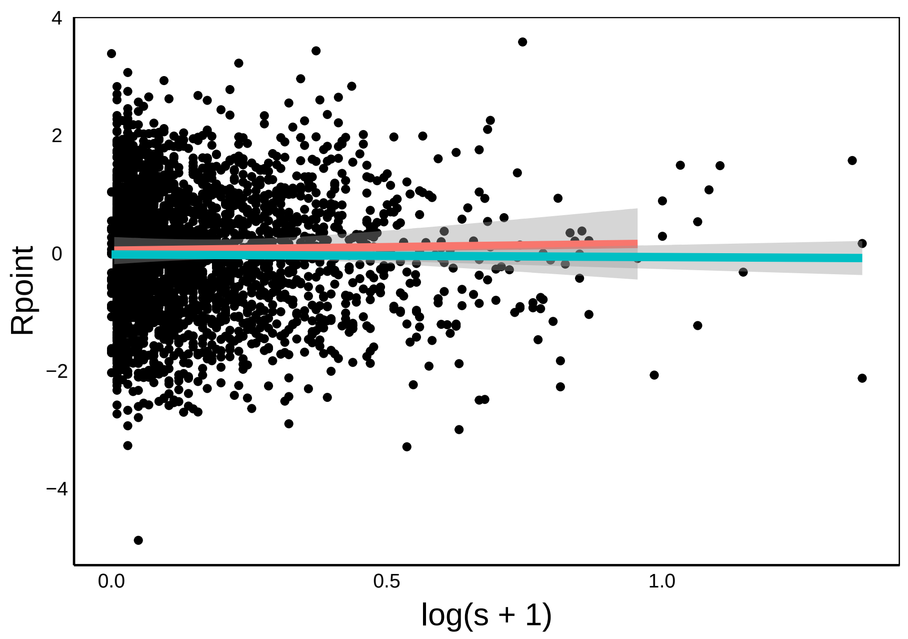
```
## `geom_smooth()` using formula 'y ~ x'
```



We can use a grouping variable to add separate regression lines for each group

```
BivPlot +
  geom_smooth(method="lm" ,size=2, aes(group=Native, colour=Native))
```

```
## `geom_smooth()` using formula 'y ~ x'
```



4.7 Full ggplot

Now that we've done a bit of exploration, let's try to recreate the selection histograms from Colautti & Lau:

1. Create separate data for native vs. introduced species
2. Use a bootstrap to estimate non-parametric mean and 95% confidence intervals
3. Plot all of the components on a single graph

4.7.1 Separate data

Since this is a relatively simple resampling model, we'll use two separate vectors to keep track of each interation: one for native and one for non-native.

```
NatSVals<-SelData$s[SelData$Native=="yes"]
IntSVals<-SelData$s[SelData$Native=="no"]
```

An alternative would be to set up a data frame and keep track of values as separate columns, with a different row for each iteration.

4.7.2 Bootstrap

The graph includes a bootstrap model to estimate the mean and variance for each group (native=yes vs no). Bootstrapping is covered in the Bootstrapping and Randomization Tutorial. The example below is not the most efficient approach but it applies the flow control concepts covered in the R Fundamentals Tutorial

4.7.2.1 Data Setup

First we define the number of iterations and set up two objects to hold the data from our bootstrap loops.

```
IterN<-100 # Number of iterations
NatSims<-{} # Dummy objects to hold output
IntSims<-{}
```

4.7.2.2 `for` loop

Here we apply our `for` loop. in each round, we:

1. Sample, with replacement and calculate average
2. Store average in `NatSims` (native species) or `IntSims` (non-native species)

```
for (i in 1:IterN){
  NatSims[i]<-mean(sample(
    NatSVals, length(NatSVals), replace=T))
  IntSims[i]<-mean(sample(
    IntSVals, length(IntSVals), replace=T))
}
```

Note in the above code we use 'nested' functions. First we sample, then calculate the mean.

4.7.2.3 95% CI

Confidence Intervals (CI) are calculated from the bootstrap output.

First, sort the datea from low to high

```
NatSims<-sort(NatSims)
IntSims<-sort(IntSims)
```

Each of the objects contains a number of values equal to our `Iter` variable defined above. Now we identify the lower 2.5% and upper 97.5% values in each vector. For example, with 1000 iterations our 2.5% would be the 25th value in the sorted vector and the upper 97.5% would be the 975th value in the sorted vector.

We use this number to index the vector with square brackets. We make sure to round to a whole number since we can't have a fractional cell position.

```
CIs<-c(sort(NatSims)[round(IterN*0.025,0)], # Lower 2.5%
       sort(NatSims)[round(IterN*0.975,0)], # Upper 97.5%
       sort(IntSims)[round(IterN*0.025,0)], # Lower 2.5%
       sort(IntSims)[round(IterN*0.975,0)]) # Upper 97.5%
```

Note that the output is a vector of length 4:

```
print(CIs)
```

```
## [1] 0.1830531 0.2011817 0.2116054 0.2996865
```

4.7.3 Plot data

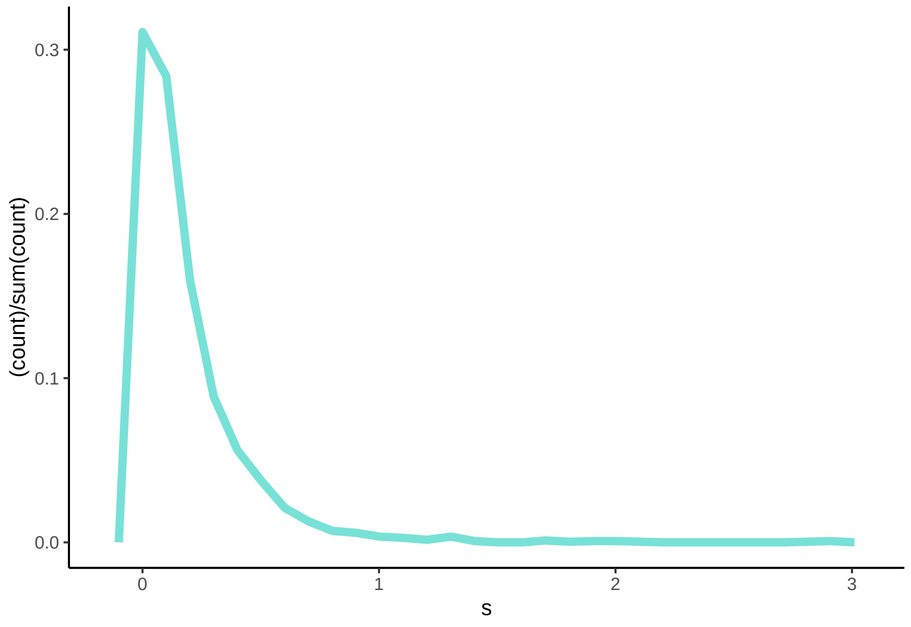
We'll combine the separate bootstrap vectors into a single data.frame object to make it easier to incorporate into our `ggplot` functions.

```
HistData<-data.frame(s=SelData$s,Native=SelData$Native)
```

Now we can add layers to the plot. We'll print out each layer as we go, so that we can see what each layer does. The coding is a bit complex here, so don't worry if it's hard to follow everything. The key thing to understand here is how the different geoms contribute to the final plot as individual 'layers'.

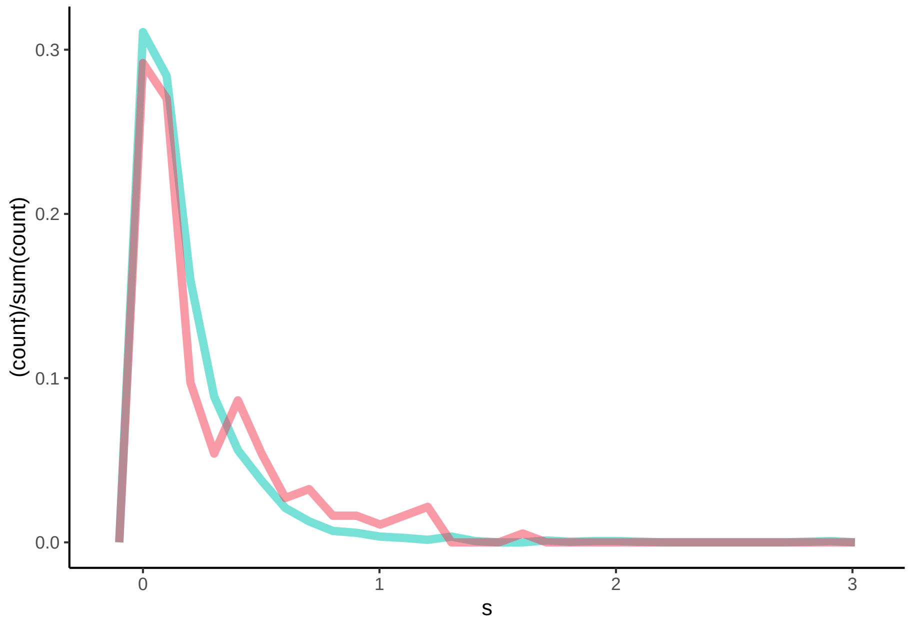
```
p <- ggplot() + theme_classic()
p <- p + geom_freqpoly(data=HistData[HistData$Native=="yes",],
                       aes(s,y=(..count..)/sum(..count..)),
                       alpha = 0.6,colour="#1fcebd",size=2)
print(p) # native species histogram
```

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



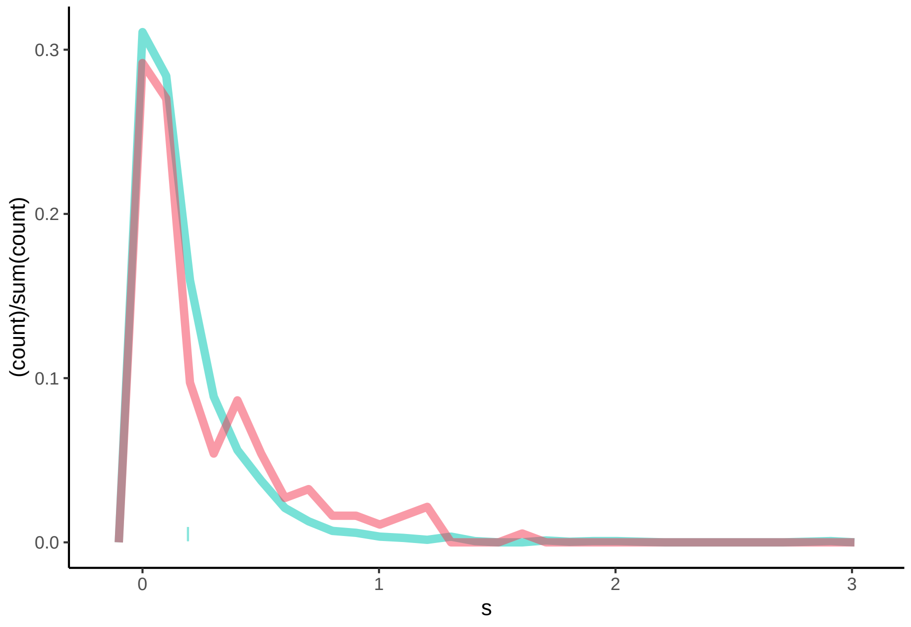
```r
p <- p + geom_freqpoly(data=HistData[HistData$Native=="no",],
                       aes(s,y=(..count..)/sum(..count..)),
                       alpha = 0.5,colour="#f53751",size=2)
print(p) # introduced species histogram
```

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



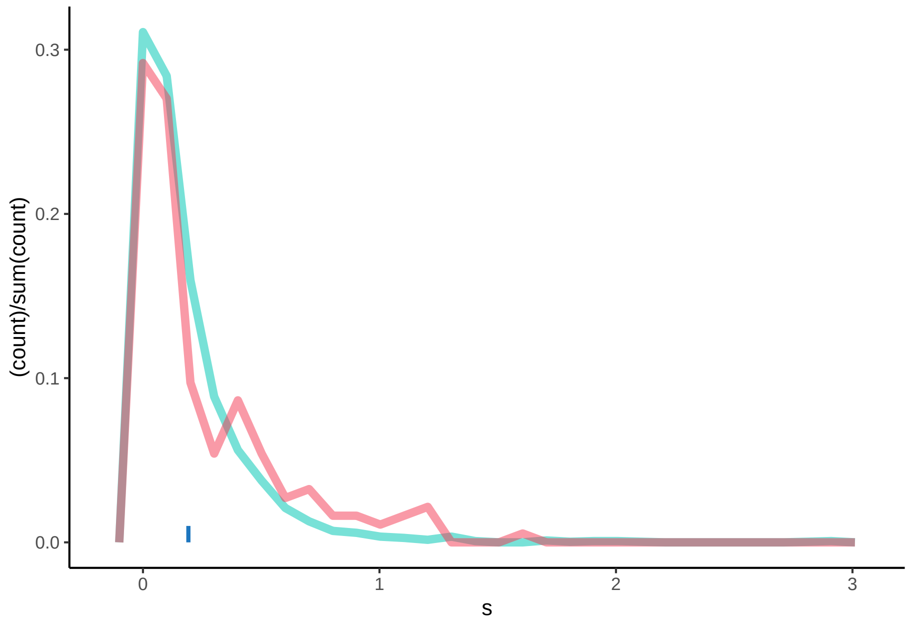
```
p <- p + geom_rect(aes(xmin=CIs[1],xmax=CIs[2],ymin=0,ymax=0.01),
                   colour="white",fill="#1fcebd88")
print(p) # native species 95% CI bar
```

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



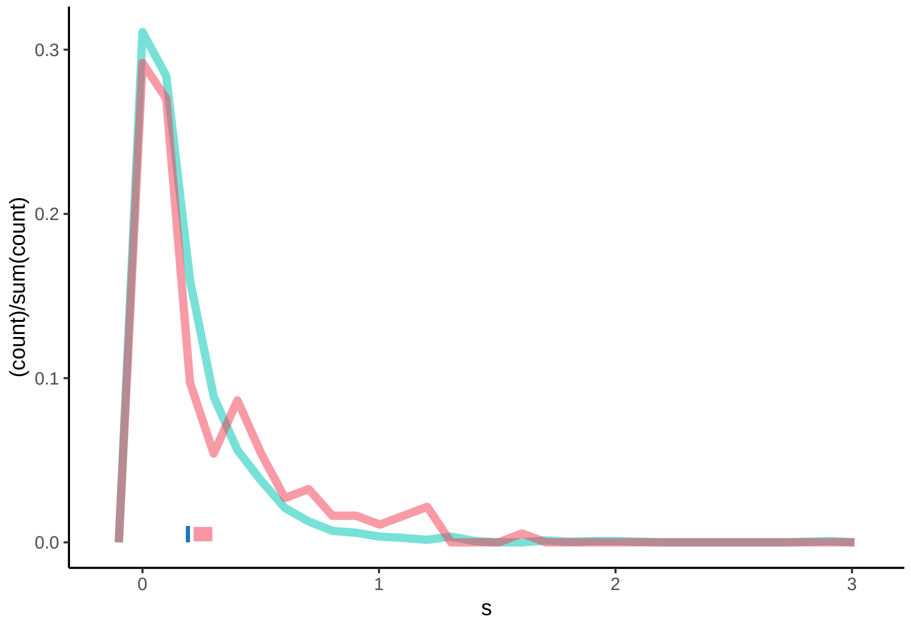
```
p <- p + geom_line(aes(x=mean(NatSims),y=c(0,0.01)),
                   colour="#1d76bf",size=1)
print(p) # native species bootstrap mean
```

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



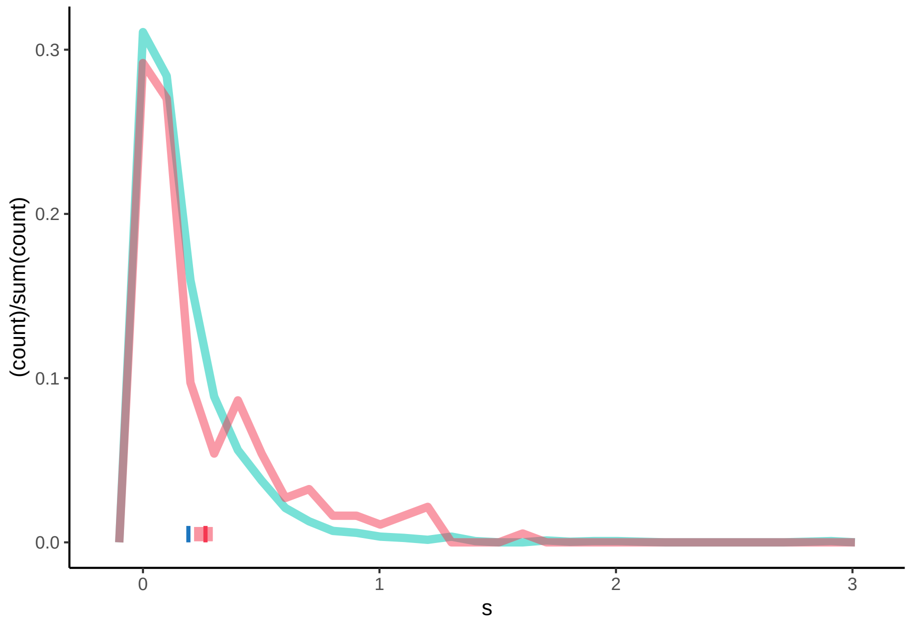
```
p <- p + geom_rect(aes(xmin=CIs[3],xmax=CIs[4],ymin=0,ymax=0.01),
                   colour="white",fill="#f5375188")
print(p) # introduced species 95% CI bar
```

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



```
p <- p + geom_line(aes(x=mean(IntSims),y=c(0,0.01)),
                   colour="#f53751",size=1)
print(p) # introduced species bootstrap mean
```

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



```
p <- p + ylab("Frequency") +
  scale_x_continuous(limits = c(0, 1.5))
print(p) # labels added, truncated x-axis
```

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

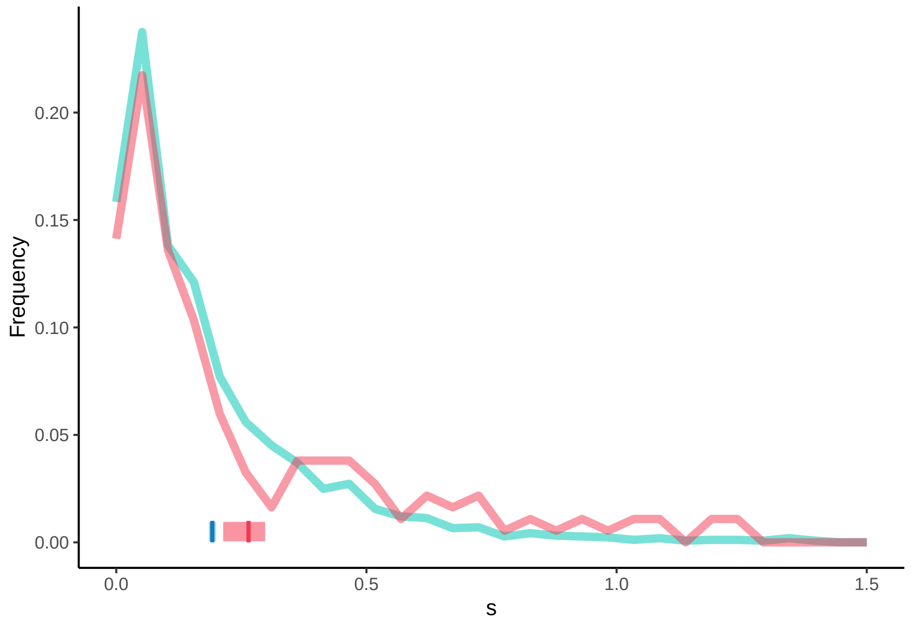
```
## Warning: Removed 12 rows containing non-finite values (stat_bin).
```

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

```
## Warning: Removed 1 rows containing non-finite values (stat_bin).
```

```
## Warning: Removed 2 row(s) containing missing values (geom_path).

## Warning: Removed 2 row(s) containing missing values (geom_path).
```



Another important point to note is that case we leave the `ggplot()` function empty because each geom uses different data. The data for each geom is defined by separate `aes` functions inside of each geom.

4.8 Multi-graph

In the qplot Tutorial we saw a quick and easy way to make multiple graphs for different groups using the `facets` parameter.

4.8.1 `facets`

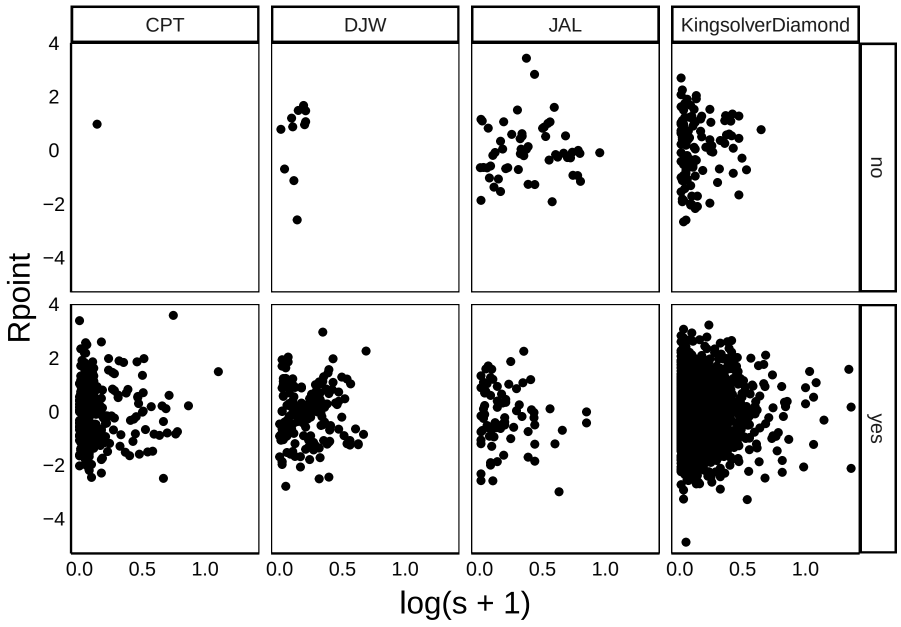
We can also use facets with `gpplot`, just using a slightly different syntax giving a couple of options:

1. `facet_grid` lets us define a grid and set the vertical and horizontal variables
2. `facet_wrap` is a convenient option if only have one categorical variable but many categories

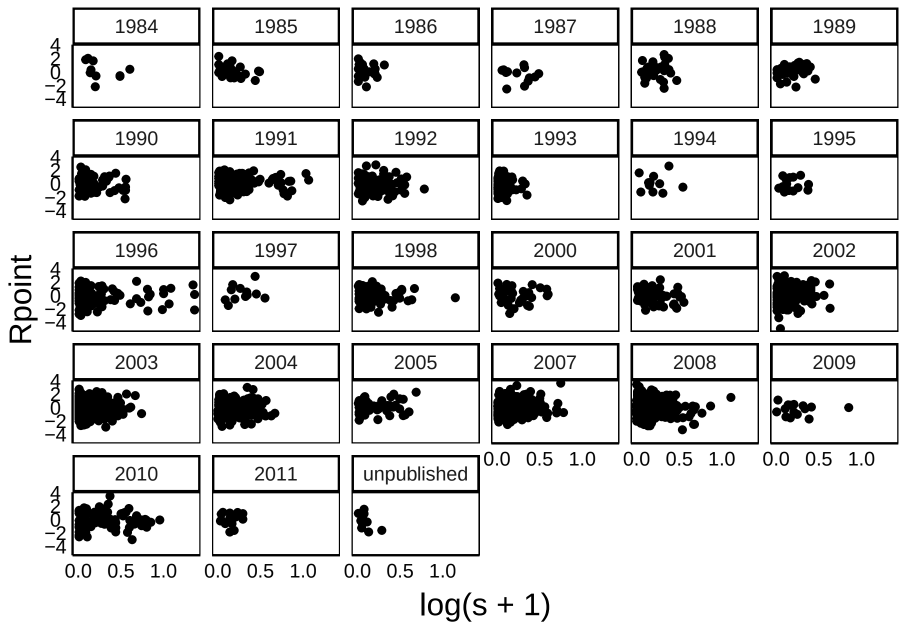
NOTE: one little tricky part of facets with `ggplot` is that it uses `vars` instead of the `aes` function to indicate which categorical variables from the original dataset should be used to subset the graphs.

Returning to the `BivPlot` example above:

```
BivPlot<-ggplot(data=SelData, aes(x=log(s+1), y=Rpoint)) +
  geom_point()
BivPlot + facet_grid(rows=vars(Native), cols=vars(Collector))
```



```
BivPlot<-ggplot(data=SelData, aes(x=log(s+1), y=Rpoint)) +
  geom_point()
BivPlot + facet_wrap(vars(Year))
```



4.8.2 `grid.extra` package

Facets produce graphs with the same x- and y-axes. We might call these 'homogenous' plots because they use the same axes and they are all the same size. For publications though, we might want to includ 'heterogeneous' plots with different axes and different sizes. The `gridExtra` package provides options for this.

Install 'gridExtra' with `install.packages("gridExtra")`

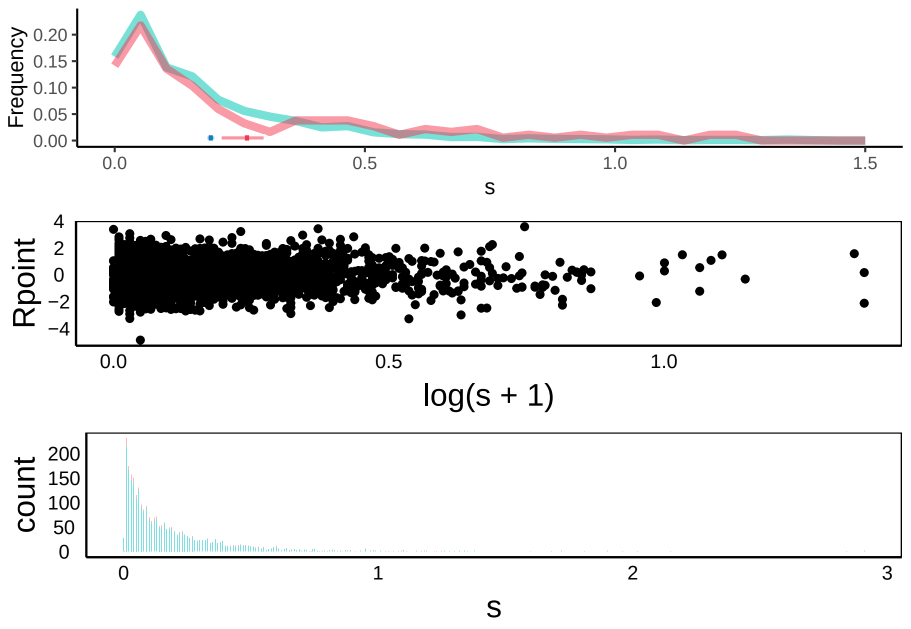
```
library(gridExtra)
```

4.8.2.1 `grid.arrange()`

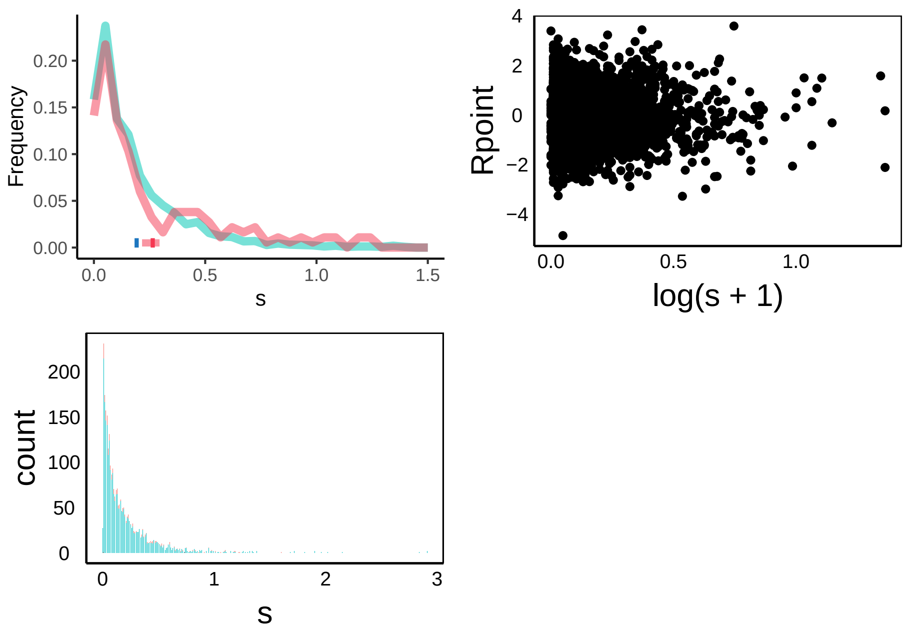
Use this to combine heterogeous ggplot objects into a single multi-panel plot.

Note that this will print ghe graphs graphs down rows, then across columns, from top left to bottom right. You can use `nrow` and `ncol` to control the layout in a grid format.

```
HistPlot<-p # Make a more meaningful name
grid.arrange(HistPlot,BivPlot,BarPlot,ncol=1)
```



```r
grid.arrange(HistPlot,BivPlot,BarPlot,nrow=2)
```



Note: You might get some warnings based on missing values or wrong binwidth. You will also see some weird things with different text sizes in the graphs. Normally, you would want to fix these for a final published figure but here we are just focused on showing what is possible with the layouts.

4.8.3 `grid` package

What if we want to have graphs of different sizes? Or what if we want one figure to be inside another? We can make some even more advanced graphs using the `grid` package.

```
library(grid)
```

First, we set up the plotting area with `grid.newpage`

```
grid.newpage() # Open a new page on grid device
```

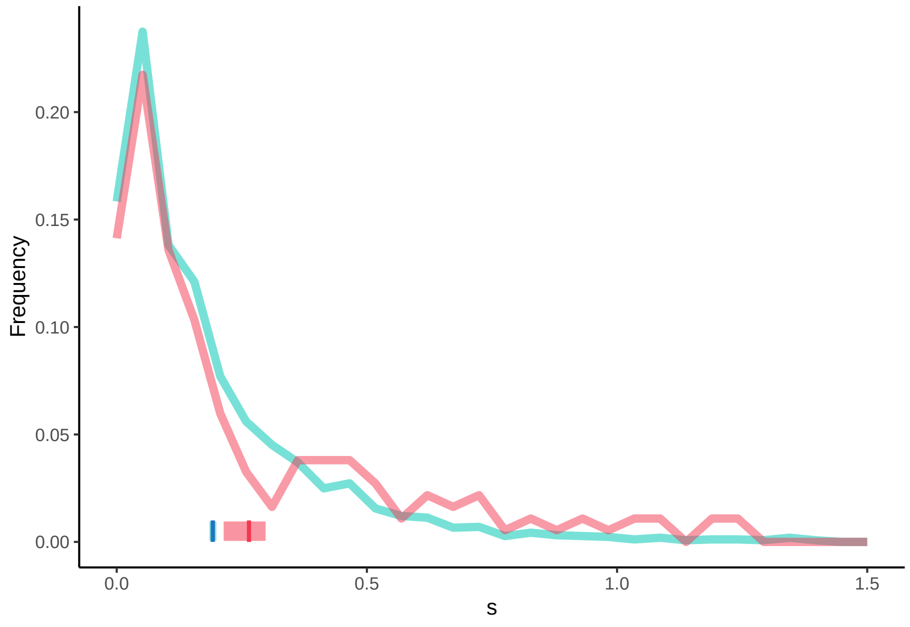
To insert a new graph on top (or inside) the current graph, we use `pushViewport` to set up an imaginary plotting grid. In this case, imagine breaking up the plotting space into 3 rows by 2 columns.

```
pushViewport(viewport(layout = grid.layout(3, 2)))
```

Finally, we print each `ggplot` object, specifying which grid(s) to plot in.

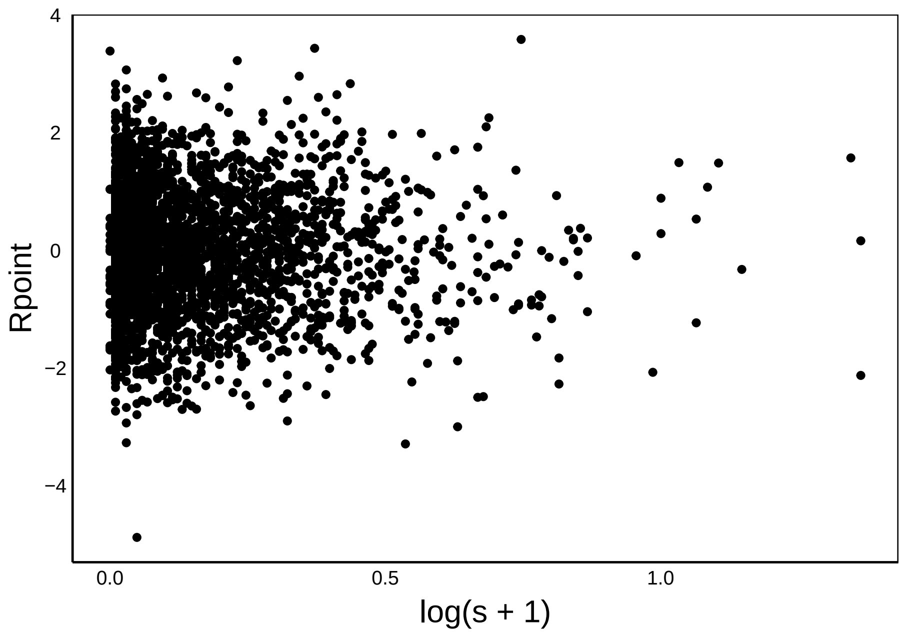
Add the first figure in row 3 and across columns 1:2

```
print(HistPlot, vp = viewport(layout.pos.row = 3,
                              layout.pos.col = 1:2))
```



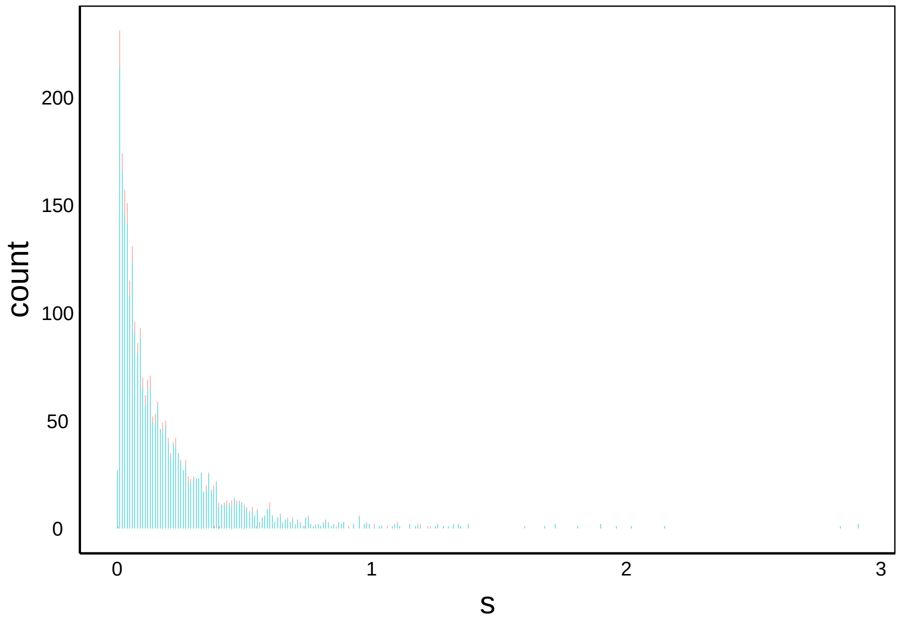
Add the next figure across rows 1 and 2 of column 1

```
print(BivPlot, vp = viewport(layout.pos.row = 1:2,
                             layout.pos.col = 1))
```



Add the final figure across rows 1 and 2 of column 2

```
print(BarPlot, vp = viewport(layout.pos.row = 1:2,
                             layout.pos.col = 2))
```

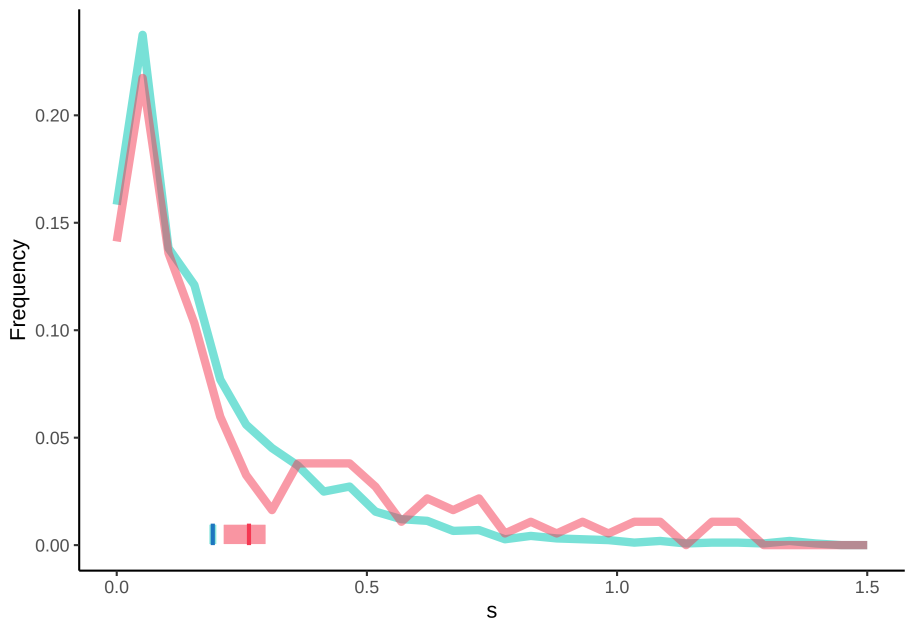


4.8.3.1 Inset

We can also use `pushViewport` to set up a grid for plotting on top of an existing graph or image. This can be used to generate a figure with an inset.

First generate the 'background' plot. Note that you could alternatively load an image here to place in the background.

```
HistPlot
```

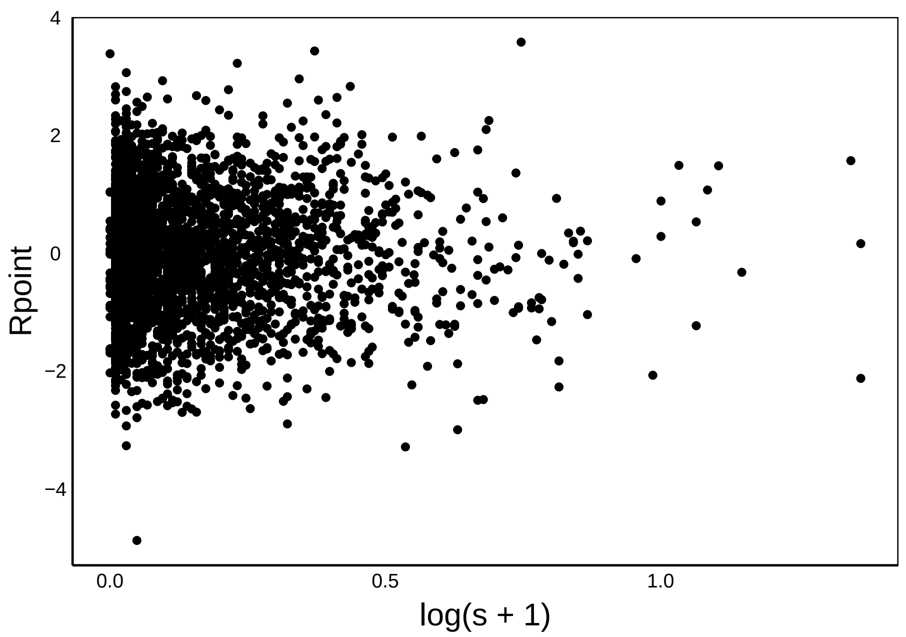


Next, overlay an invisible 4x4 grid layout (number of cells, will determine size/location of graph)

```
pushViewport(viewport(layout = grid.layout(4, 4)))
```

Finally, add the graph. In this case we want it only in the top two rows and the right-most two columns – i.e. the top-right corner.

```
print(BivPlot, vp = viewport(layout.pos.row = 1:2, layout.pos.col = 3:4))
```



4.9 Further Reading

The 2009 book *ggplog2: Elegant Graphics for Data Analysis* by Hadly Wickham is the definitive guide to all things ggplot.

A physical copy is published by Springer http://link.springer.com/book/10.1007%2F978-0-387-98141-3

And there is a free ebook version: https://ggplot2-book.org/

Chapter 5

Regular Expressions

5.1 Overview

Regular Expressions, also known as **regex** and **regexp** is a syntax that allows for coders to run complex find-and-replace functions. I didn't learn regular expressions until I was a postdoc working at Duke University, but I wish I had learned this a lot earlier! This remains one of the most useful programming tools I have ever used. It is absolutely essential for working with any kind of large text files or large datasets.

A lot of programming tools in biology use input text files that require very specific formatting (e.g. .txt, .csv, .fasta, .nex). Sometimes, you might need to reorganize or recode data in a large text file. This can be a big pain that can lead to errors if you try to do everything manually. But regular expressions can automate the process.

Here's one example. As a PhD student I co-founded a project called the **Global Garlic Mustard Field Survey (GGMFS)** with collaborator Dr. Oliver Bossdorf at the University of Tuebingen – yes the same Dr. Bossdorf mentioned in the qplot Tutorial. We were fortunate to have over 100 collaborators across Europe and North America who helped to collect samples for the project. Details of the project were published in the Journal **Neobiota**: https://neobiota.pensoft.net/article/1270/ but one BIG problem is the way that each of these 100+ collaborators entered their data online. For example, latitudes and longitudes were entered in a variety of different formats. Regular expressions allowed me to write a small program to automatically convert all of these different formats to a common, decimal format that we could use for the analysis. This saved a huge amount of time and prevented errors that could have been introduced if we tried to edit these values by hand.

Often when you work with large datasets, you will need to automate some of your error correction, and regular expressions can be a big help here. For example, imagine a simple field where people were simply asked a simple yes or no question. You might find a variety of inputs such as: "YES, Y, yes, and Yes". These all mean the same thing but if you try to analyze it, R will treat these as different categories. Here again, regular expressions can be used to quickly change all the different examples to a common "Y" or even a boolean variable `TRUE`.

One final example, is pattern searching. This is common for the analysis of DNA, proteins or other large strings of data. You may want to find a particular sequence of data, possibly with a few variable sites: e.g. TCTA or TCAA or TCGA. This is another area where regular expressions can help.

5.1.1 Universal

Regular expressions are a universal language that extends to many other programming languages, including **C/C#/C++**, **Python**, **Unix/Linux**, and **Perl**. We focus here on R but most of the syntax is mantained across programming languages.

WARNING!

There is a very steep learning curve here, and the only way to really learn this is to drown yourself in examples. There are lots of exercises you can do for practice online. You should also try to apply these whenever you can.

5.2 Functions

There are four main functions that use regular expressions in R.

`grep()` and `grepl()` are equivalent to 'find' in your favorite word processor

They have the general form: `grep("find this pattern", in.this.object)`

`grep()` outputs a vector with all the address locations (i.e. numbers) that match. Thus the output length is equal to the number of matches.

`grepl()` outputs a vector of `TRUE` (match) and `FALSE` (no match). Thus, the output length is equal to the length of the input object.

`sub()` and `gsub()` are equivalent to 'find and replace'

They have the general form: `grep("find this pattern", "and replace with this", in.this.object)`

`sub()` replaces only the first match, whereas `gsub()` replaces all of the matches.

Some specific examples are provided below to help you understand these similarities and differences.

There are two other more advanced functions in R. These aren't covered in this tutorial, but may be of use once you are more comortable with the above functions.

`regexpr()` provides more detailed info about the first match

`gregexpr()` provides more detailed results about all matches

5.2.1 Examples

Some examples can help to understand the differences among the four main functions. Let's start with a simple data frame of species names.

```
Species<-c("petiolata", "verticillatus", "salicaria", "minor")
print(Species)
```

```
## [1] "petiolata"     "verticillatus" "salicaria"     "minor"
```

5.2.2 `grep()`

This returns cell addresses matching query. Note the vector length compared to the input vector.

```
grep("a",Species)
```

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

5.2.3 `grepl()`

This returns a vector of `TRUE` (match) and `FALSE` (no match). Comapre this output with `grep()`.

```
grepl("a",Species)
```

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

5.2.4 `sub()`

This replaces the first match (in each cell)

```
sub("l","L",Species)
```

```
## [1] "petioLata"     "verticiLlatus" "saLicaria"     "minor"
```

5.2.5 gsub()

This replaces all matches (in each cell). Compere this output to `sub()`.

```
gsub("l","L",Species)
```

```
## [1] "petioLata"     "verticiLLatus" "saLicaria"     "minor"
```

5.3 Wildcards

5.3.1 \ escape character

The backslash is a special character. It's called the 'escape' character because it is used to 'escapes' the interpretation of the next character that occurs after it. This is easier to understand by example, as shown below.

5.3.2 \\ in R

In the introduction, we discussed the universality of **regular expressions** in the sense that a similar syntax is used by many different programming langagues. But now here is one exception. In R, the double-escape is usually needed, whereas other programming languages typically use just one. The reason is a bit meta – it's because we are running regular expressions within R object. So the first \ is used to escape special characters in R, applying it to the second \, which is itself the special character that needs to be escaped to pass through the function. The second slash is followed by the 'escaped' character. Some examples are provided below.

If that isn't clear. Just remember that you need two backslashes instead of one.

5.3.3 `\\w`

Instead of finding the letter `w`, the `\\w` is a **wildcard** character that represents any letter or digits.

```
gsub("w","X","...which 1-100 words get replaced?")
```

```
## [1] "...Xhich 1-100 Xords get replaced?"
```

```
gsub("\\w","X","...which 1-100 words get replaced?")
```

```
## [1] "...XXXXX X-XXX XXXXX XXX XXXXXXXX?"
```

5.3.4 `\\W`

This is the inverse of `\\w` find a character that is NOT a letter or number.

```
gsub("\\W","X","...which 1-100  words get replaced?")
```

```
## [1] "XXXwhichX1X100XXwordsXgetXreplacedX"
```

5.3.5 `\\s`

This represents a space

```
gsub("\\s","X","...which 1-100  words get replaced?")
```

```
## [1] "...whichX1-100XXwordsXgetXreplaced?"
```

5.3.6 `\\t`

This is a tab character. A lot of data files stored as text are tab-delimited (.tsv) as well as comma-delimited (.csv)

```
gsub("\\t","X","...which 1-100  words get replaced?")
```

```
## [1] "...which 1-100  words get replaced?"
```

5.3.7 `\\d`

Digit characters

```
gsub("\\d","X","...which 1-100  words get replaced?")
```

```
## [1] "...which X-XXX  words get replaced?"
```

5.3.8 \\D

Non-digit characters

```
gsub("\\D","X","...which 1-100  words get replaced?")
```

```
## [1] "XXXXXXXXX1X100XXXXXXXXXXXXXXXXXXXXXX"
```

5.4 New Lines

There are two special characters that indicate new lines in a text file.

5.4.1 \\r

This is the 'carriage return' special character

5.4.2 \\n

This is the 'newline' special character

5.4.3 Big Problem

One or two of these is/are generated when you press the 'enter' key while writing a text file. These also add a source of headache and confusion when working with text files because:

> Macs/Unix and PC/Windows use different standards!

Unix/Mac files – lines usually end with `\\n` only

Windows/DOS files – lines usually end with `\\r\\n`

This can cause problems when moving text files between Windows/DOS and Mac/Unix machines, particularly with older operating systems or when working on remote computers that use very basic Linux software.

5.5 Special characters

In addition to special characters that use the escape `\\`, there are a number of other special characters that don't use the escape, but have a special meaning.

Note that if you want to search for the characters below you would have to use the escape character. E.g. `\\.` if you wanted to search for a period `.`.

5.5.1 `|`

This is sometimes called the **pipe** character, and it simply means 'or'. For example, we can search for `w or e`.

```
gsub("w|e","X","...which 1-100  words get replaced?")
```

```
## [1] "...Xhich 1-100  Xords gXt rXplacXd?"
```

5.5.2 `.`

This means any character except new line. This includes all of the `\\w` characters but also other characters like puncutation marks.

```
gsub(".","X","...which 1-100  words get replaced?")
```

```
## [1] "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
```

So how to search for a period `.`? As noted above, we have to use the escape character

```
gsub("\\.","X","...which 1-100  words get replaced?")
```

```
## [1] "XXXwhich 1-100  words get replaced?"
```

5.5.3 `*, ?, +, {}`

These special characters refer to details about the kind of search that we are trying to conduct. Look at these examples carefully, and remember that `sub` replaces the first match while `gsub` replaces all of the matches.

```
sub("\\w","X","...which 1-100 words get replaced?")
```

```
## [1] "...Xhich 1-100 words get replaced?"
```

```
gsub("\\w","X","...which 1-100 words get replaced?")
```

```
## [1] "...XXXXX X-XXX XXXXX XXX XXXXXXXX?"
```

Now let's apply some of these special characters to see how they work.

5.5.3.1 `+`

Finds 1 or more matches

```
sub("\\w+","X","...which 1-100 words get replaced?")
```

```
## [1] "...X 1-100 words get replaced?"
```

```
gsub("\\w+","X","...which 1-100 words get replaced?")
```

```
## [1] "...X X-X X X X?"
```

Compare this match to the one above. Notice how we have replaced groups of letters instead of single letters. The algorithm works like this:

1. Start at the left and move to the right, one character at a time
2. Check if the character is a letter or number (`\\w`).
3. If NO, move to the next character
4. If YES, check the next character. If it is also a `\\w` then go to the next character. Repeat until the next character is not `\\w`, and replace the entire string of characters.

When run in the `sub` command, it does the above and then stops. When run with the `gsub` command, it continues to the next character, and then starts over.

5.5.4 `*`

This is a 'greedy' search (match the largest possible)

```
sub("\\w*","X","...which 1-100 words get replaced?")
```

```
## [1] "X...which 1-100 words get replaced?"
```

```
gsub("\\w*","X","...which 1-100 words get replaced?")
```

```
## [1] "X.X.X.X X-X X X X?X"
```

In the `sub` command, it detects a `.` as the first character, indicating no match. It replaces the 'null' or `0` match at the beginning, which has the effect of adding a character. In the `gsub` command it repeats this before each `.` until it finds the letter `w`. Then it finds a group of `\\w` matches, replacing with a single star. Then a space, which is skipped, then a `-`, which is another null match, prompting another insert.

5.5.5 ?

This means 'match zero or one time'

```
sub("\\w?","X","...which 1-100 words get replaced?")
```

```
## [1] "X...which 1-100 words get replaced?"
```

```
gsub("\\w?","X","...which 1-100 words get replaced?")
```

```
## [1] "X.X.X.XXXXX X-XXX XXXXX XXX XXXXXXXX?X"
```

Compare this to the `*` above. It behaves in a similar way, except it is not 'greedy' – in the second example, each letter is replaced instead of entire words.

5.5.6 +?

This is the 'lazy' version of `+` – note in particular the difference in `sub` which replaces on the the first letter here but the whole word when `+` is used alone. In the `gsub` example we end up replacing every letter instead of whole words. Remember, `sub` runs the algorithm once and then stops, while `gsub` repeats until it reaches the end of the line.

```
sub("\\w+?","X","...which 1-100 words get replaced?")
```

```
## [1] "...Xhich 1-100 words get replaced?"
```

```
gsub("\\w+?","X","...which 1-100 words get replaced?")
```

```
## [1] "...XXXXX X-XXX XXXXX XXX XXXXXXXX?"
```

5.5.7 `*?`

Similarly, we can combine these characters for the 'lazy' version of `*`

```
sub("\\w*?","X","...which 1-100 words get replaced?")
```

```
## [1] "X...which 1-100 words get replaced?"
```

```
gsub("\\w*?","X","...which 1-100 words get replaced?")
```

```
## [1] "X.X.X.XwXhXiXcXhX X1X-X1X0X0X XwXoXrXdXsX XgXeXtX XrXeXpXlXaXcXeXdX?X"
```

Try using `+*`. Why do you get an error message?

5.5.8 `{}`

Curly brackets are used to specify a number of matches, expanding on the options even futher.

5.5.9 `{n,m}`

Find between n to m matches

```
gsub("\\w{3,4}","X","...which 1-100 words get replaced?")
```

```
## [1] "...Xh 1-X Xs X XX?"
```

5.5.10 `{n}`

Find exactly n matches

```
gsub("\\w{3}","X","...which 1-100 words get replaced?")
```

```
## [1] "...Xch 1-X Xds X XXed?"
```

5.5.11 {n,}

Fomd n or more matches

```
gsub("\\w{4,}","X","...which 1-100 words get replaced?")
```

```
## [1] "...X 1-100 X get X?"
```

5.5.12 {}?

As above, we cau use `?` for the 'lazy' versions of these searches

```
gsub("\\w{4,}?","X","...which 1-100 words get replaced?")
```

```
## [1] "...Xh 1-100 Xs get XX?"
```

5.6 []

Square brackets allow us to find 'any' of the values listed within them. We can also use the dash `-` to specify a range of numbers or letters.

```
gsub("[aceihw-z]","X","...which 1-100 words get replaced?")
```

```
## [1] "...XXXXX 1-100 Xords gXt rXplXXXd?"
```

In the above example, we search for 1 of any of the listed letters: a, c, e, i h, w, x, y, z. Note that x and y are included in the `x-z` statement.

What if we want to find 1 or more of these characters in a row?

```
gsub("[aceihw-z]+","X","...which 1-100 words get replaced?")
```

```
## [1] "...X 1-100 Xords gXt rXplXd?"
```

5.7 ^ and $

Use these characters to specify searches at the start ^ or end $ of the input string.

5.7.1 ^

Find species starting with the letter *s*

```
grep("^s",Species)
```

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

> IMPORTANT: ^ Also means 'negate' when used within `[]`

Find species containing any letter other than *s*

```
grep("[^a]",Species)
```

```
## [1] 1 2 3 4
```

Replace every letter except *s*

```
gsub("[^a]","X",Species)
```

```
## [1] "XXXXXXaXa"     "XXXXXXXXXaXXX" "XaXXXaXXa"     "XXXXX"
```

5.7.2 $

Find species ending with *a*

```
grep("a$",Species)
```

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

5.8 ()

Regular parentheses are used to ‘capture’ text, which can then be specified in the replacement string using `\\1`. Or you can capture multiple pieces of text and reorganize them by using the corresponding number – `\\1` for the first set of`()`, `\\2` for the second set of `()`, etc. Some examples should help.

Replace each word with its first letter

```
gsub("(\\w)\\w+","\\1",
     "...which 1-100 words get replaced?")
```

```
## [1] "...w 1-1 w g r?"
```

Pull out only the numbers and reverse their order

```
gsub(".*([0-9]+)-([0-9]+).*","\\2-\\1",
     "...which 1-100 words get replaced?")
```

```
## [1] "100-1"
```

Reverse first two letters of each word

```
gsub("(\\w)(\\w)(\\w+)","\\2\\1\\3",
     "...which 1-100 words get replaced?")
```

```
## [1] "...hwich 1-010 owrds egt erplaced?"
```

5.9 Scraping

Scraping is a method for collecting data from online sources. In R, we can use the functions `readLines` and `curl()`, both from the `curl` library, to ‘scrape’ data from websites. Websites with the `.html` extension are a special kind of text file.

We can use regular expressions to pull out text from the website. Here’s an example where we will scrape a record for the Green Fluorescent Protein (GFP) from the Protein Data Bank (PDB). Note that this is a file with the extension `.pdb` but this is a human-readable text file that can be opened in any text editor

First, we’ll import the text into an R object.

```
library(curl)
```

```
## Using libcurl 7.54.0 with LibreSSL/2.6.5
```

You will have to use `install.packages("curl")` to download this package to your computer. You only need to do this once but you will have to use `library(curl)` whenever you want to use the commands, as explained in the R Fundamentals Tutorial

Now we can download a file to play with.

```
Prot<-readLines(curl("http://www.rcsb.org/pdb/files/1ema.pdb"))
```

> HINT: Download this link to your computer and open with a text file.

This hint is a simple trick to understand what kind of file(s) you are working with.

This is a tab-delimited file, which we could import as a data frame using `read.delim` but we'll keep it this way to see how we can use regular expressions.

The `Prot` object we have made is a simple vector of strings, with each cell corresponding to a different row of text:

```
length(Prot)
```

```
## [1] 2363
```

```
Prot[grep("TITLE",Prot)]
```

```
## [1] "TITLE     GREEN FLUORESCENT PROTEIN FROM AEQUOREA VICTORIA
```

We can pull out the amino acid sequences, which are rows that start with the word 'ATOM'

```
AAseq<-Prot[grep("^ATOM",Prot)]
length(AAseq)
```

```
## [1] 1717
```

```r
AAseq[1]
```

```
## [1] "ATOM      1  N   SER A   2      28.888   9.409  52.301  1.00 85.05           N  "
```

> Try to isolate the 3-letter amino acid code

There are lots of possibilities. Take the time to try a few.

Here's one good option, since we know it's a tab-delimited file with the amino acid in the 4th column:

```r
gsub("ATOM\\t\\w+\\t\\w+\\t(\\w+).*","\\1",AAseq[1])
```

```
## [1] "ATOM      1  N   SER A   2      28.888   9.409  52.301  1.00 85.05           N  "
```

That didn't work. Sometimes the 'tabs' are actually just multiple 'spaces'

```r
AAchain<-gsub("ATOM\\s+\\w+\\s+\\w+\\s+(\\w+).*","\\1",AAseq)
AAchain[1:100]
```

```
##   [1] "SER" "SER" "SER" "SER" "SER" "SER" "LYS" "LYS" "LYS" "LYS" "LYS" "LYS"
##  [13] "LYS" "LYS" "LYS" "GLY" "GLY" "GLY" "GLY" "GLU" "GLU" "GLU" "GLU" "GLU"
##  [25] "GLU" "GLU" "GLU" "GLU" "GLU" "GLU" "GLU" "GLU" "GLU" "GLU" "LEU" "LEU"
##  [37] "LEU" "LEU" "LEU" "LEU" "LEU" "LEU" "PHE" "PHE" "PHE" "PHE" "PHE" "PHE"
##  [49] "PHE" "PHE" "PHE" "PHE" "PHE" "THR" "THR" "THR" "THR" "THR" "THR" "THR"
##  [61] "GLY" "GLY" "GLY" "GLY" "VAL" "VAL" "VAL" "VAL" "VAL" "VAL" "VAL" "VAL"
##  [73] "VAL" "VAL" "VAL" "VAL" "VAL" "VAL" "PRO" "PRO" "PRO" "PRO" "PRO" "PRO"
##  [85] "PRO" "ILE" "ILE" "ILE" "ILE" "ILE" "ILE" "ILE" "ILE" "LEU" "LEU" "LEU"
##  [97] "LEU" "LEU" "LEU" "LEU"
```

5.10 Examples

Let's try practicing with a couple of examples

5.10.1 Transect Data

Regular expressions are also useful with data objects

Imagine you have a repeated measures design. 3 transects (A-C) and 3 positions along each transect (1-3)

```
Transect<-data.frame(Species=letters[1:20],
                     A1=rnorm(20), A2=rnorm(20), A3=rnorm(20),
                     B1=rnorm(20), B2=rnorm(20) ,B3=rnorm(20),
                     C1=rnorm(20), C2=rnorm(20), C3=rnorm(20))
head(Transect)
```

```
##   Species         A1         A2          A3         B1          B2         B3
## 1       a  1.0026367  1.6763781  0.08235899 -2.1279464 -0.53942667  0.8172936
## 2       b  1.6367613  0.2777369 -0.23968309  0.3301162 -0.05800561 -1.3504984
## 3       c -0.1255452 -0.2403649 -1.04368071  0.8118806 -0.66886234 -0.1145576
## 4       d -0.1310659  0.7937417  0.82442696  1.1981370 -1.04644921  1.5674981
## 5       e -0.6411110 -2.9705401 -0.61354242 -0.9798944 -0.62512442 -0.8194804
## 6       f -2.4289272  0.7656483 -0.42568322 -0.8025643  0.17078836 -1.8826243
##           C1           C2         C3
## 1 -1.1362409 -1.155217413 -0.1988261
## 2  0.6382336 -0.872748615 -0.2992873
## 3 -0.7252746 -1.403068356  0.8186320
## 4  0.4539147  0.363032769 -0.2500811
## 5 -0.3036914 -0.280217416 -0.5303418
## 6 -0.7584407  0.003397663  0.8867261
```

> TIP: the object `letters` contains lower-case letter, while `LETTERS` contains upper case.

5.10.1.1 Challenge

Use your knowledged gained above with substting data outlined in the R Fundamentals Tutorial to do the following:

1. Subset only the columns that have an "A" in their name
2. Subset the data for species "D"

Take the time to do this on your own. It will take you a while and you will make a lot of mistakes. That's all part of the learning process. The longer you struglle, the faster you will learn.

Now here is a more challenging example:

5.10.2 Genbank

Here is a line of code to import DNA from genbank. (The one line is broken up into three physical lines to make it easier to read)

```
Lythrum_18S<-scan(
  "https://colauttilab.github.io/RCrashCourse/sequence.gb",
  what="character",sep="\n")
```

This is the sequence of the 18S subunit from the ribosome gene of *Lythrum salicaria* (from Genbank)

```
print(Lythrum_18S)
```

```
##  [1] "LOCUS       AF206955                1740 bp    DNA     linear   PLN 18-APR-2003"
##  [2] "DEFINITION  Lythrum salicaria 18S ribosomal RNA gene, complete sequence."
##  [3] "ACCESSION   AF206955"
##  [4] "VERSION     AF206955.1"
##  [5] "KEYWORDS    ."
##  [6] "SOURCE      Lythrum salicaria"
##  [7] "  ORGANISM  Lythrum salicaria"
##  [8] "            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;"
##  [9] "            Spermatophyta; Magnoliopsida; eudicotyledons; Gunneridae;"
## [10] "            Pentapetalae; rosids; malvids; Myrtales; Lythraceae; Lythrum."
## [11] "REFERENCE   1  (bases 1 to 1740)"
## [12] "  AUTHORS   Soltis,P.S., Soltis,D.E. and Chase,M.W."
## [13] "  TITLE     Direct Submission"
## [14] "  JOURNAL   Submitted (19-NOV-1999) School of Biological Sciences, Washington"
## [15] "            State University, Pullman, WA 99164-4236, USA"
## [16] "FEATURES             Location/Qualifiers"
## [17] "     source          1..1740"
## [18] "                     /organism=\"Lythrum salicaria\""
## [19] "                     /mol_type=\"genomic DNA\""
## [20] "                     /db_xref=\"taxon:13129\""
## [21] "                     /note=\"Lythrum salicaria L.\""
## [22] "     rRNA            1..1740"
## [23] "                     /product=\"18S ribosomal RNA\""
## [24] "ORIGIN      "
## [25] "        1 gtcatatgct tgtctcaaag attaagccat gcatgtgtaa gtatgaacaa attcagactg"
## [26] "       61 tgaaactgcg aatggctcat taaatcagtt atagtttgtt tgatggtatc tgctactcgg"
## [27] "      121 ataaccgtag taattctaga gctaatacgt gcaacaaacc ccgacttctg gaagggacgc"
## [28] "      181 atttattaga taaaaggtcg acgcgggctt tgcccgatgc tctgatgatt catgataact"
## [29] "      241 tgacggatcg cacggccatc gtgccggcga cgcatcattc aaatttctgc cctatcaact"
## [30] "      301 ttcgatggta ggatagtggc ctaccatggt gtttacgggt aacggagaat tagggttcga"
## [31] "      361 ttccggagag ggagcctgag aaacggctac cacatccaag gaaggcagca ggcgcgcaaa"
## [32] "      421 ttacccaatc ctgacacggg gaggtagtga caataaataa caatactggg ctctttgagt"
## [33] "      481 ctggtaattg gaatgagtac aatctaaatc ccttaacgag gatccattgg agggcaagtc"
## [34] "      541 tggtgccagc agccgcggta attccagctc caatagcgta tatttaagtt gttgcagtta"
## [35] "      601 aaaagctcgt agttggacct tgggttgggt cgaccggtcc gcctttggtg tgcaccgatc"
## [36] "      661 ggctcgtccc ttctaccggc gatgcgcgcc tggccttaat tggccgggtc gttcctccgg"
## [37] "      721 tgctgttact ttgaagaaat tagagtgctc aaagcaagca ttagctatga atacattagc"
## [38] "      781 atgggataac attataggat tccgatccta ttatgttggc cttcgggatc ggagtaatga"
## [39] "      841 ttaacaggga cagtcggggg cattcgtatt tcatagtcag aggtgaaatt cttggattta"
## [40] "      901 tgaaagacga acaactgcga aagcatttgc caaggatgtt ttcattaatc aagaacgaaa"
## [41] "      961 gttgggggct cgaagacgat cagataccgt cctagtctca accataaacg atgccgacca"
## [42] "     1021 gggatcagcg aatgttactt ttaggacttc gctggcacct tatgagaaat caaagttttt"
## [43] "     1081 gggttccggg gggagtatgg tcgcaaggct gaaacttaaa ggaattgacg gaagggcacc"
## [44] "     1141 accaggagtg gagcctgcgg cttaatttga ctcaacacgg ggaaacttac caggtccaga"
## [45] "     1201 catagtaagg attgacagac tgagagctct ttcttgattc tatgggtggt ggtgcatggc"
## [46] "     1261 cgttcttagt tggtggagcg atttgtctgg ttaattccgt taacgaacga gacctcagcc"
## [47] "     1321 tgctaactag ctatgtggag gtacacctcc acggccagct tcttagaggg actatggccg"
## [48] "     1381 cttaggccaa ggaagtttga ggcaataaca ggtctgtgat gcccttagat gttctgggcc"
## [49] "     1441 gcacgcgcgc tacactgatg tattcaacga gtctatagcc ttggccgaca ggcccgggta"
## [50] "     1501 atctttgaaa tttcatcgtg atggggatag atcattgcaa ttgttggtct tcaacgagga"
## [51] "     1561 attcctagta agcgcgagtc atcagctcgc gttgactacg tccctgccct ttgtacacac"
## [52] "     1621 cgcccgtcgc tcctaccgat tgaatggtcc ggtgaaatgt tcggatcgcg gcgacgtggg"
## [53] "     1681 cgcttcgtcg ccgacgacgt cgcgagaagt ccattgaacc ttatcattta gaggaaggag"
## [54] "//"
```

Notice that each line is read in as a separate cell in a vector, with sequences beginning with a number ending with 1. We can take advantage of this to extract just the sequence data

5.10.2.1 Challenge

Before proceeding try to do the following:

1. Isolate only the rows containing DNA sequences. This should include

```
a. Removing all of the characters that are not a, t, g, or c.
b. Combining separate cells/lines into a single string. You can do this with using the
```

2. Convert lower-case to upper-case. To do this, you can use `gsub("([actg])","\\U\\1",Seq,perl=T` The `\\U\\\1` means 'paste brackets as upper-case, and is only available as a **Perl** command, which is accessible in gsub with the `perl=T` parameter.
3. Replace start codons (ATG) with "–>START–>ATG"
4. Replace stop codons (TAA or TAG or TGA) with TAA or TAG or TGA followed by ">–STOP–|"

Take the time to stuggle with this and try different combinations until you find a way through. The more you struggle, the faster you will learn.

A cool thing about regular expressions is that there is rarely a single right answer, especially for complicated problems. When you are ready, Continue on to see one possible solution.

5.11 Solutions

5.11.1 Transects

You want to look at only transect A for the first 3 species:

```
Transect[1:3,grep("A",names(Transect))]
```

```
##           A1        A2          A3
## 1  1.0026367  1.6763781  0.08235899
## 2  1.6367613  0.2777369 -0.23968309
## 3 -0.1255452 -0.2403649 -1.04368071
```

Subset the data for species "D":

```
Transect[grep("1",names(Transect)),]
```

```
##   Species        A1         A2         A3         B1          B2         B3
## 2       b  1.636761  0.2777369 -0.2396831  0.3301162 -0.05800561 -1.3504984
## 5       e -0.641111 -2.9705401 -0.6135424 -0.9798944 -0.62512442 -0.8194804
## 8       h -1.019354 -0.4900937 -1.8032210 -2.2482622  0.02246712 -0.7517218
##           C1         C2         C3
## 2  0.6382336 -0.8727486 -0.2992873
## 5 -0.3036914 -0.2802174 -0.5303418
## 8  0.5613055  0.8620439 -0.1722671
```

5.11.2 Genbank

Use `.*` with `()` to delete everything before the DNA sequence

```
Seq<-gsub(".*(1 [gatc])","",Lythrum_18S)
paste(Seq,collapse="")
```

```
## [1] "LOCUS       AF206955                1740 bp    DNA     linear   PLN 18-APR-2003DEFINITION
```

Use the `.*` and space with `+` to eliminate all text before the sequence :

```
Seq<-gsub(".*ORIGIN +","",paste(Seq,collapse=""))
Seq
```

```
## [1] "tcatatgct tgtctcaaag attaagccat gcatgtgtaa gtatgaacaa attcagactggaaactgcg aatggctcat taaa
```

Elimminate spaces and the two `//` at the end

```
Seq<-gsub(" |//","",Seq)
Seq
```

```
## [1] "tcatatgcttgtctcaaagattaagccatgcatgtgtaagtatgaacaaattcagactggaaactgcgaatggctcat
```

Capital letters look nicer, but requires a PERL qualifier `\\U` that is not standard in R

```
Seq<-gsub("([actg])","\\U\\1",Seq,perl=T)
print(Seq)
```

```
## [1] "TCATATGCTTGTCTCAAAGATTAAGCCATGCATGTGTAAGTATGAACAAATTCAGACTGGAAACTGCGAATGGCTCAT
```

Look for start codons?

```
gsub("ATG","-->START-->ATG",Seq)
```

```
## [1] "TCAT-->START-->ATGCTTGTCTCAAAGATTAAGCC-->START-->ATGC-->START-->ATGTGTAAGT-->S
```

Look for stop codons?

```
gsub("(TAA|TAG|TGA)","\\1>--STOP--|",Seq)
```

```
## [1] "TCATATGCTTGTCTCAAAGATTAA>--STOP--|GCCATGCATGTGTAA>--STOP--|GTATGA>--STOP--|ACA
```

5.12 More Exercises

Here are some more exercises to practice your skils. No solutions are given for these, you will have to solve them on your own. Note that they may find their way onto a test, assignment or quiz.

5.12.1 1. Consider a vector of email addresses scraped from the internet:

- robert 'dot' colautti 'at' queensu 'dot' ca
- chris.eckert[at]queensu.ca
- lonnie.aarssen at queensu.ca

Use regular expressions to convert all email addresses to the standard format: name@queensu.ca

5.12.2 2. Create a random sequence of DNA:

```
My.Seq<-sample(c("A","T","G","C"),1000,replace=T)
```

```
* Replace T with U
* Find all start codons (AUG) and stop codons (UAA, UAG, UGA)
* Find all open reading frames (hint: consider each sequence beginning with AUG and ending with a
* Count the length of bp for all open reading frames
```

5.12.3 3. More online examples

http://regex.sketchengine.co.uk/extra_regexps.html

5.12.4 4. Regex Golf

Have fun! LINK

Chapter 6

Data Science Intro

6.1 Setup

If you don't have it installed, then run `install.packages("dplyr")` – it can take a while to download and install.

6.2 Introduction to Data Science

Data Science is a relatively new field of study that merges **computer science** and **statistics** to answer questions in other domains (e.g. business, medicine, biology, psychology). Data Science as a discipline has grown in popularity in response to the rapid rate of increase in data collection and publication.

Data Science often involves 'Big Data', which doesn't have a strict quantitative definition but will usually have one or more of the following characteristics:

1. High **Volume** – large file sizes with lots of observations.
2. Wide **Variety** – lots of different types
3. High **Velocity** – accumulating at a high rate
4. Compromised **Veracity** – variable quality that must be dealt otherwise downstream analyses will be compromised.

> What are some examples of 'big data' in Biology?

Medical records, remote sensing data (e.g. climate stations, satellite images), and 'omics data are good examples of 'big data' in biology.

In biology, it can be helpful to think of Data Science as a continuous life-cycle with multiple stages:

6.2.1 Data Science Life-Cycle

1. **Hypothesize** – Make initial observations of about the natural world, or insights from other data, that lead to testable hypotheses. Your core biology training is crucial here.
2. **Collect** – This may involve taking measurements yourself, manually entering data that is not yet in electronic format, requesting data from authors of published studies, or importing data from online sources. Collecting data is a crucial step that is often done poorly. Some tips on this are provided in a paper by Wu et al
3. **Correct** – Investigate the data for quality assurance, to identify and fix potential errors. Start to visualize the data to look for outliers or nonsensical relationships (or lack thereof).
4. **Explore** – Try to understand the data, where they come from, and potential limitations on their use. Continue visualizing data; this may cause you to modify your hypotheses slightly.
5. **Model** – Now that hypotheses are clearly defined, apply statistical tests of their validity.
6. **Report** – Use visualizations along with the results of your statistical tests to summarize your findings.
7. **Repeat** – Return to step 1.

In this tutorial, we focus mainly on coding in R for steps 2, 3, and 6. Step 5 requires a firm understanding of statistics. Step 4 is covered in the tutorials on qplot and ggplot. Step 1 is everything covered in a typical degree in the biological sciences.

Data collection and management are crucial steps in the Data Science Life-Cycle. Read the paper by Wu et al. called *baRcodeR with PyTrackDat: Open-source labelling and tracking of biological samples for repeatable science.* Pay particular attention to the '*Data Standards*' section. The *baRcodeR* and *PyTrackDat* programs and their application to current projects may also be of interest.

6.2.2 Data Science in R

The book R for Data Science by Hadley Wickham & Garrett Grolemund is an excellent resource using R in Data Science projects.

> TIP: In general, any book by Hadley Wickham that you come across is worth reading if you want to be proficient in R.

Here we'll be focusing on the `dplyr` packages from Wickham et al.

6.3 2D Data Wrangling

The `dplyr` library in R has many useful features for importing and reorganizing your data for steps 2, 3 and 4 in the Data Science Life-Cycle outlined above. Don't forget to install the `dplyr` library and load it into memory.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

> Note: This error message just informs you that `dplyr` uses function or parameter names that are the same as other base or stats packages in R. These base/stats functions are 'masked' meaning that when you run one (e.g. `filter`) then R will run the `dplyr` version rather than the stats version.

We'll work with our FallopiaData.csv dataset, and remind ourselves of the structure of the data

```
Fallo<-read.csv(
  "https://colauttilab.github.io/RCrashCourse/FallopiaData.csv")
str(Fallo)
```

```
## 'data.frame':    123 obs. of  13 variables:
##  $ PotNum      : int  1 2 3 5 6 7 8 9 10 11 ...
##  $ Scenario    : chr  "low" "low" "low" "low" ...
##  $ Nutrients   : chr  "low" "low" "low" "low" ...
##  $ Taxon       : chr  "japon" "japon" "japon" "japon" ...
##  $ Symphytum   : num  9.81 8.64 2.65 1.44 9.15 ...
##  $ Silene      : num  36.4 29.6 36 21.4 23.9 ...
##  $ Urtica      : num  16.08 5.59 17.09 12.39 5.19 ...
##  $ Geranium    : num  4.68 5.75 5.13 5.37 0 9.05 3.51 9.64 7.3 6.36 ...
##  $ Geum        : num  0.12 0.55 0.09 0.31 0.17 0.97 0.4 0.01 0.47 0.33 ...
##  $ All_Natives : num  67 50.2 61 40.9 38.4 ...
##  $ Fallopia    : num  0.01 0.04 0.09 0.77 3.4 0.54 2.05 0.26 0 0 ...
##  $ Total       : num  67.1 50.2 61.1 41.7 41.8 ...
##  $ Pct_Fallopia: num  0.01 0.08 0.15 1.85 8.13 1.12 3.7 0.61 0 0 ...
```

This file is an example of a 2-dimensional data set, which is common in biology. 2D datasets have the familiar row x column layout used by spreadsheet programs like Microsoft Excel or Google Sheets. There are some exceptions, but data in this format should typically follows 3 rules:

1. Each cell contains a single value
2. Each variable must have its own column
3. Each observation must have its own row

Making sure your data are arranged this way will usually make it much easier to work with.

6.3.1 `filter()`

Let's subset observations based on value

```
Pot1<-filter(Fallo,PotNum==1)
head(Pot1)
```

```
##   PotNum Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 1      1      low       low japon      9.81  36.36  16.08     4.68 0.12
##   All_Natives Fallopia Total Pct_Fallopia
## 1       67.05     0.01 67.06         0.01
```

6.3.2 `rename()`

It's possible to change the names of columns in your data. In base R you can use the `names()` function with the square bracket index `[]`:

```
X<-Fallo
names(X)
```

```
##  [1] "PotNum"       "Scenario"     "Nutrients"    "Taxon"        "Symphytum"
##  [6] "Silene"       "Urtica"       "Geranium"     "Geum"         "All_Natives"
## [11] "Fallopia"     "Total"        "Pct_Fallopia"
```

```
names(X)[12]<-"Total_Biomass"
names(X)
```

```
##  [1] "PotNum"        "Scenario"      "Nutrients"     "Taxon"
##  [5] "Symphytum"     "Silene"        "Urtica"        "Geranium"
##  [9] "Geum"          "All_Natives"   "Fallopia"      "Total_Biomass"
## [13] "Pct_Fallopia"
```

```
X<-rename(Fallo, Total_Biomass = Total)
names(X)
```

```
##  [1] "PotNum"        "Scenario"      "Nutrients"     "Taxon"
##  [5] "Symphytum"     "Silene"        "Urtica"        "Geranium"
##  [9] "Geum"          "All_Natives"   "Fallopia"      "Total_Biomass"
## [13] "Pct_Fallopia"
```

There is also a simple `dplyr` function to do this:

6.3.3 `arrange()`

Use the `arrange()` function to sort the *rows* of your data based on the *columns* of your data. For example, let's re-arrange our FallopiaData.csv dataset based on Taxon (a string denoting the species of Fallopia used) and Total (a float denoting the total biomass in each pot).

```
X<-arrange(Fallo, Taxon, Total)
head(X)
```

```
##   PotNum Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 1     26      low       low bohem     13.25  18.11   0.00     0.00 0.10
## 2     17      low       low bohem      4.90  29.52   1.36     0.00 0.19
## 3     80  gradual      high bohem     11.92  17.16   8.92     0.94 0.18
## 4     18      low       low bohem      3.51  27.61   8.14     3.81 0.21
## 5     28      low       low bohem     10.59  18.78   7.19     6.73 0.17
## 6     22      low       low bohem      0.76  22.66   9.85    10.60 0.74
##   All_Natives Fallopia Total Pct_Fallopia
## 1       31.46     0.00 31.46         0.00
## 2       35.97     0.00 35.97         0.00
## 3       39.12     3.35 42.47         7.89
## 4       43.28     0.00 43.28         0.00
## 5       43.46     0.00 43.46         0.00
## 6       44.61     0.00 44.61         0.00
```

use the `desc()` function with `arrange()` to change to descending order

```
X<-arrange(Fallo, Taxon, desc(Total))
head(X)
```

```
##   PotNum     Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 1    148 fluctuations      high bohem      4.15  38.70  23.59     5.11 1.36
## 2     86      gradual      high bohem      2.93  60.93   4.11     6.67 1.27
## 3     60         high      high bohem      7.77  51.45   5.13    10.10 0.37
## 4     85      gradual      high bohem     10.19  26.66  15.01     3.07 0.14
## 5     53         high      high bohem      7.05  56.29   1.14     4.07 0.00
## 6    118      extreme      high bohem     14.21  25.15  12.69     8.06 0.29
##   All_Natives Fallopia Total Pct_Fallopia
## 1       72.91     5.89 78.80         7.47
## 2       75.91     0.00 75.91         0.00
## 3       74.82     0.00 74.82         0.00
## 4       55.07    14.47 69.54        20.81
## 5       68.55     0.00 68.55         0.00
## 6       60.40     7.46 67.86        10.99
```

6.3.4 `select()`

The `select()` function can be used to select a subset of columns (i.e. variables) from your data.

Suppose we only want to look at total biomass, but keep all the treatment columns:

```
X<-select(Fallo, PotNum, Scenario, Nutrients, Taxon, Total)
head(X)
```

```
##   PotNum Scenario Nutrients Taxon Total
## 1      1      low       low japon 67.06
## 2      2      low       low japon 50.22
## 3      3      low       low japon 61.08
## 4      5      low       low japon 41.71
## 5      6      low       low japon 41.81
## 6      7      low       low japon 48.27
```

You can also use the colon `:` to select a range of columns:

```
X<-select(Fallo, PotNum:Taxon, Total)
head(X)
```

```
##   PotNum Scenario Nutrients Taxon Total
## 1      1      low       low japon 67.06
## 2      2      low       low japon 50.22
## 3      3      low       low japon 61.08
## 4      5      low       low japon 41.71
## 5      6      low       low japon 41.81
## 6      7      low       low japon 48.27
```

Exclude columns with -

```
X<-select(Fallo, -PotNum:Taxon, -Total)
```

```
## Warning in x:y: numerical expression has 12 elements: only the first used
```

> Oops, what generated that error? Take a careful look at the error message and see if you can figure it out.

The problem is we are using the range of columns between PotNum and Taxon, but in one case we are excluding and the other we are including. We need to keep both the same:

```
X<-select(Fallo, -PotNum:-Taxon, Total)
head(X)
```

```
##   Symphytum Silene Urtica Geranium Geum All_Natives Fallopia Total Pct_Fallopia
## 1      9.81  36.36  16.08     4.68 0.12       67.05     0.01 67.06         0.01
## 2      8.64  29.65   5.59     5.75 0.55       50.18     0.04 50.22         0.08
## 3      2.65  36.03  17.09     5.13 0.09       60.99     0.09 61.08         0.15
## 4      1.44  21.43  12.39     5.37 0.31       40.94     0.77 41.71         1.85
## 5      9.15  23.90   5.19     0.00 0.17       38.41     3.40 41.81         8.13
## 6      6.31  24.40   7.00     9.05 0.97       47.73     0.54 48.27         1.12
```

Or a bit more clear:

```
X<-select(Fallo, -(PotNum:Taxon), Total)
head(X)
```

```
##   Symphytum Silene Urtica Geranium Geum All_Natives Fallopia Total Pct_Fallopia
## 1      9.81  36.36  16.08     4.68 0.12       67.05     0.01 67.06         0.01
## 2      8.64  29.65   5.59     5.75 0.55       50.18     0.04 50.22         0.08
## 3      2.65  36.03  17.09     5.13 0.09       60.99     0.09 61.08         0.15
## 4      1.44  21.43  12.39     5.37 0.31       40.94     0.77 41.71         1.85
## 5      9.15  23.90   5.19     0.00 0.17       38.41     3.40 41.81         8.13
## 6      6.31  24.40   7.00     9.05 0.97       47.73     0.54 48.27         1.12
```

6.3.5 everything()

Use the `everything()` function with `select()` to rearrange your columns without losing any:

```
X<-select(Fallo, Taxon, Scenario, Nutrients, PotNum, Pct_Fallopia, everything())
head(X)
```

```
##   Taxon Scenario Nutrients PotNum Pct_Fallopia Symphytum Silene Urtica Geranium
## 1 japon      low       low      1         0.01      9.81  36.36  16.08     4.68
## 2 japon      low       low      2         0.08      8.64  29.65   5.59     5.75
## 3 japon      low       low      3         0.15      2.65  36.03  17.09     5.13
## 4 japon      low       low      5         1.85      1.44  21.43  12.39     5.37
## 5 japon      low       low      6         8.13      9.15  23.90   5.19     0.00
## 6 japon      low       low      7         1.12      6.31  24.40   7.00     9.05
##   Geum All_Natives Fallopia Total
## 1 0.12       67.05     0.01 67.06
## 2 0.55       50.18     0.04 50.22
## 3 0.09       60.99     0.09 61.08
## 4 0.31       40.94     0.77 41.71
## 5 0.17       38.41     3.40 41.81
## 6 0.97       47.73     0.54 48.27
```

6.3.6 mutate()

Suppose we want to make a new column (variable) to our data.frame object (dataset) that is the sum of biomass of Urtica and Geranium only. In base R you would use `$`:

```
X<-Fallo
X$UrtSil<-X$Urtica+X$Silene
```

In the dplyr package you can use mutate

```
X<-mutate(Fallo, UrtSil = Urtica + Silene)
head(X)
```

```
##   PotNum Scenario Nutrients Taxon Symphytum Silene Urtica Geranium Geum
## 1      1      low       low japon      9.81  36.36  16.08     4.68 0.12
## 2      2      low       low japon      8.64  29.65   5.59     5.75 0.55
## 3      3      low       low japon      2.65  36.03  17.09     5.13 0.09
## 4      5      low       low japon      1.44  21.43  12.39     5.37 0.31
## 5      6      low       low japon      9.15  23.90   5.19     0.00 0.17
## 6      7      low       low japon      6.31  24.40   7.00     9.05 0.97
##   All_Natives Fallopia Total Pct_Fallopia UrtSil
## 1       67.05     0.01 67.06         0.01  52.44
## 2       50.18     0.04 50.22         0.08  35.24
## 3       60.99     0.09 61.08         0.15  53.12
## 4       40.94     0.77 41.71         1.85  33.82
## 5       38.41     3.40 41.81         8.13  29.09
## 6       47.73     0.54 48.27         1.12  31.40
```

This is a lot more readable, especially when you have complicated equations or you want to add lots of new columns.

> What if you only wanted to retain the new columns and delete everything else? Try it.

Which functions did you use?

6.3.7 `transmute()`

You can also use `transmute()` instead of `mutate()` + `select()`

```
X<-transmute(Fallo, UrtSil = Urtica + Silene)
head(X)
```

```
##   UrtSil
## 1  52.44
## 2  35.24
## 3  53.12
## 4  33.82
## 5  29.09
## 6  31.40
```

6.3.8 `summarize()` + `group_by()`

This can be useful for quickly summarizing your data, for example to find the mean or standard deviation based on a particular treatment or group.

```
TrtGrp<-group_by(Fallo,Taxon,Scenario,Nutrients)
summarize(TrtGrp, Mean=mean(Total), SD=sd(Total))
```

```
## `summarise()` has grouped output by 'Taxon', 'Scenario'. You can override using the `.groups`
## # A tibble: 10 x 5
## # Groups:   Taxon, Scenario [10]
##    Taxon Scenario     Nutrients  Mean    SD
##    <chr> <chr>        <chr>     <dbl> <dbl>
##  1 bohem extreme      high       58.3  7.34
##  2 bohem fluctuations high       58.4  9.20
##  3 bohem gradual      high       57.5  9.34
##  4 bohem high         high       60.3  8.68
##  5 bohem low          low        48.0  8.86
##  6 japon extreme      high       57.2 10.9
##  7 japon fluctuations high       56.4 13.7
##  8 japon gradual      high       59.7  9.57
##  9 japon high         high       56.4  8.20
## 10 japon low          low        52.0  8.29
```

6.3.9 Weighted Mean

In our dataset, the **Taxon** column shows which of two species of *Fallopia* were used in the competition experiments. We might want to take the mean total biomass for each of the two *Fallopia* species:

```
X<-group_by(Fallo,Taxon)
summarize(X, Mean=mean(Total), SD=sd(Total))
```

```
## # A tibble: 2 x 3
##   Taxon  Mean    SD
##   <chr> <dbl> <dbl>
## 1 bohem  56.3  9.54
## 2 japon  56.4 10.4
```

However, there are other factors in our experiment that may affect biomass. The *Nutrients* column tells us whether pots received high or low nutrients, and this also affects biomass:

```
X<-group_by(Fallo,Nutrients)
summarize(X, Mean=mean(Total), SD=sd(Total))
```

```
## # A tibble: 2 x 3
##   Nutrients  Mean    SD
##   <chr>     <dbl> <dbl>
## 1 high       58.0  9.61
## 2 low        49.9  8.66
```

Now imagine if our sampling design is 'unbalanced'. For example, maybe we had some plant mortality or lost some plants to a tornado. If one of the two species in the *Taxon* column had more high-nutrient pots, then it would have a higher mean. BUT, the higher mean is not an effect of the Taxon, but is simply due to the unbalanced nature of the design. We can simulate this effect by re-shuffling the species names:

```
RFallo<-Fallo
set.seed(256)
RFallo$Taxon<-rbinom(nrow(RFallo),size=1,prob=0.7)

X<-group_by(RFallo,Taxon)
summarize(X, Mean=mean(Total))
```

```
## # A tibble: 2 x 2
##   Taxon  Mean
##   <int> <dbl>
## 1     0  56.1
## 2     1  56.5
```

To fix this problem, we may want to take a *weighted mean*:

```
X1<-group_by(RFallo,Taxon,Scenario,Nutrients)
Y1<-summarize(X1,Mean=mean(Total))
```

```
## `summarise()` has grouped output by 'Taxon', 'Scenario'. You can override using the `.groups`
```

```
X2<-group_by(Y1,Taxon,Scenario)
Y2<-summarize(X2,Mean=mean(Mean))
```

```
## `summarise()` has grouped output by 'Taxon'. You can override using the `.groups` argument.
```

```
X3<-group_by(Y2,Taxon)
Y3<-summarize(X3, Mean=mean(Mean))
arrange(Y3,desc(Mean))
```

```
## # A tibble: 2 x 2
##   Taxon  Mean
##   <int> <dbl>
## 1     1  56.6
## 2     0  55.8
```

6.3.10 %>% (pipe)

The combination %>% is called 'pipe' for short. Be careful though – in Unix, 'pipe' is slightly different and uses the vertical line (often shift-backslash): |

The pipe is useful to combine operations without creating a whole bunch of new objects. This can save on memory use.

For example, we can re-write the weighted mean example using pipes:

```
RFallo %>%
  group_by(Taxon,Scenario,Nutrients) %>%
  summarize(Mean=mean(Total)) %>%
  group_by(Taxon,Scenario) %>%
  summarize(Mean=mean(Mean)) %>%
  group_by(Taxon) %>%
  summarize(Mean=mean(Mean)) %>%
  arrange(desc(Mean))
```

```
## `summarise()` has grouped output by 'Taxon', 'Scenario'. You can override using the

## `summarise()` has grouped output by 'Taxon'. You can override using the `.groups` a

## # A tibble: 2 x 2
##   Taxon  Mean
##   <int> <dbl>
## 1     1  56.6
## 2     0  55.8
```

This also avoids potential for bugs in our program. Imagine if we mis-spelled 'Taxon' in our second line by accidentialy pressing 's' along with 'x'. Compare the output:

```
X<-group_by(Fallo,Taxon,Scenario,Nutrients)
X<-group_by(X,Tasxon,Scenario)
```

```
## Error: Must group by variables found in `.data`.
## * Column `Tasxon` is not found.
```

```
X<-group_by(X,Taxon)
X<-summarize(X, Mean=mean(Total), SD=sd(Total))
arrange(X,desc(Mean))
```

```
## # A tibble: 2 x 3
##   Taxon  Mean    SD
##   <chr> <dbl> <dbl>
## 1 japon  56.4 10.4
## 2 bohem  56.3  9.54
```

```
Fallo %>%
  group_by(Taxon,Scenario,Nutrients) %>%
  group_by(Tasxon,Scenario) %>%
  group_by(Taxon) %>%
  summarize(Mean=mean(Total), SD=sd(Total)) %>%
  arrange(desc(Mean))
```

```
## Error: Must group by variables found in `.data`.
## * Column `Tasxon` is not found.
```

In both cases we get an error, but in one case we still calculate the means and sd of the two species.

> A bug that produces no output is much less dangerous than an error that gives an output. Why?

6.4 Missing Data

So far we have worked on a pristine data set that has already been edited for errors. More often datasets will contain missing values.

6.4.1 `NA` and `na.rm()`

The R language uses a special object `NA` to denote missing data.

```
Vec<-c(1,2,3,NA,5,6,7)
Vec
```

```
## [1]  1  2  3 NA  5  6  7
```

When a function is run on a vector or other object containing NA, the function will often return NA or give an error message:

```
mean(Vec)
```

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

This is by design, because it is not always clear what NA means. Many functions in R include an na.rm parameter that is set to FALSE by default. Setting it to true tells the function to ignore the NA

```
mean(Vec, na.rm=T)
```

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

6.4.2 NA vs 0

A common mistake students make is to put 0 for missing data. This can be a big problem when analyzing the data since the calculations are very different.

```
Vec1<-c(1,2,3,NA,5,6,7)
mean(Vec1, na.rm=T)
```

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

```
Vec2<-c(1,2,3,0,5,6,7)
mean(Vec2, na.rm=T)
```

```
## [1] 3.428571
```

6.4.3 is.na()

In large datasets you might want to check for missing values. Let's simulate this in our *FallopiaData.csv* dataset.

To set up a test dataset, randomly select 10 rows and replace the value for 'Total' with NA. The `sample` function is a

```
X<-round(runif(10,min=1,max=nrow(Fallo)),0)
Fallo$Total[X]<-NA
Fallo$Total
```

```
##   [1] 67.06 50.22 61.08 41.71 41.81 48.27 55.42 42.68    NA 45.89 59.02 57.66
##  [13] 48.98 35.97 43.28 52.27 45.92 44.61 59.13 58.97 55.36 31.46 43.46 44.65
##  [25] 59.69 60.82 57.21 34.09 58.57 66.74 63.18    NA 54.09 55.27 61.31 53.56
##  [37] 52.66 64.71 61.06 45.34 64.20 57.50 68.55 49.55 56.70 54.06 66.60 74.82
##  [49] 53.71 49.75 58.45 66.06 67.01 70.41    NA 63.43 77.05 47.50 61.79 54.96
##  [61] 48.99 52.01    NA 57.18 42.47 46.18 62.56 54.36 69.54 75.91 56.34 64.97
##  [73] 60.71 57.80 41.72 67.44 58.78    NA    NA 58.42 55.35    NA 55.04 39.56
##  [85] 71.07 45.23 57.20 67.70 52.46 60.86    NA 65.53 48.19 60.89 48.13 60.37
##  [97] 67.86 56.40 49.13 56.11 49.78 69.00 65.40 50.73 63.08 60.93    NA 49.12
## [109] 68.73 31.90 69.88 69.48 47.88 51.42 58.13 50.51 54.83 66.80 50.31 56.12
## [121] 62.96 78.80 64.25
```

Use `is.na()` to check for missing values:

```
is.na(Fallo$Total)
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
##  [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [49] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [61] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [73] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE
##  [85] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [97] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [121] FALSE FALSE FALSE
```

Note that the output is a vector of True/False. Each cell corresponds to a value of 'Total' with TRUE indicating missing values. This is an example of a boolean variable, which has some handy properties in R.

First, we can use it as an index. For example, let's see which pots have missing 'Total' values:

```
Missing<-is.na(Fallo$Total)
Fallo$PotNum[Missing]
```

```
## [1]  10  39  68  78  95  96 100 111 129
```

Another handy trick to count missing values is:

```
sum(is.na(Fallo$Total))
```

```
## [1] 9
```

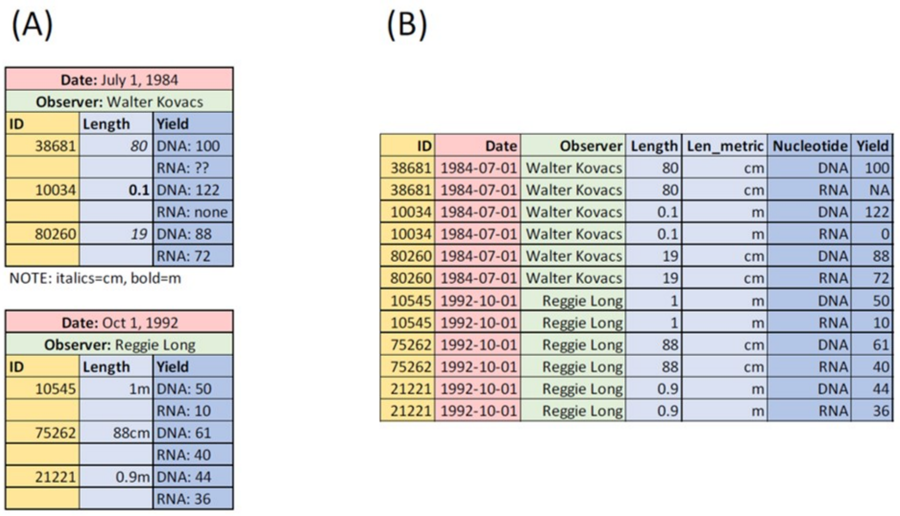
This takes advantage of the fact that the boolean TRUE/FALSE variable is equivalent to the binary 1/0 values.

6.5 Naughty Data

Naughty data contain the same information as a standard row x column (i.e. 2-dimensional) dataset but break the rules outlined above:

1. Each cell contains a single value
2. Each variable must have its own column
3. Each observation must have its own row

Examples of Naughty Data are shown in Figure 2A of the Wu et al. manuscript. :


(A)

Date: July 1, 1984		
Observer: Walter Kovacs		
ID	**Length**	**Yield**
38681	*80*	DNA: 100
		RNA: ??
10034	**0.1**	DNA: 122
		RNA: none
80260	*19*	DNA: 88
		RNA: 72

NOTE: italics=cm, bold=m

Date: Oct 1, 1992		
Observer: Reggie Long		
ID	**Length**	**Yield**
10545	1m	DNA: 50
		RNA: 10
75262	88cm	DNA: 61
		RNA: 40
21221	0.9m	DNA: 44
		RNA: 36

(B)

ID	Date	Observer	Length	Len_metric	Nucleotide	Yield
38681	1984-07-01	Walter Kovacs	80	cm	DNA	100
38681	1984-07-01	Walter Kovacs	80	cm	RNA	NA
10034	1984-07-01	Walter Kovacs	0.1	m	DNA	122
10034	1984-07-01	Walter Kovacs	0.1	m	RNA	0
80260	1984-07-01	Walter Kovacs	19	cm	DNA	88
80260	1984-07-01	Walter Kovacs	19	cm	RNA	72
10545	1992-10-01	Reggie Long	1	m	DNA	50
10545	1992-10-01	Reggie Long	1	m	RNA	10
75262	1992-10-01	Reggie Long	88	cm	DNA	61
75262	1992-10-01	Reggie Long	88	cm	RNA	40
21221	1992-10-01	Reggie Long	0.9	m	DNA	44
21221	1992-10-01	Reggie Long	0.9	m	RNA	36

Figure 2. Example of (A) common errors in data management and (B) corresponding rearrangement of the same data to simplify reproducible data wrangling and analysis. Note that colours are added to show link between data in A and B and do not appear in the final text-based file (e.g. TXT, CSV, TSV).

Naughty data can be very time consuming to fix, but regular expressions can make this a bit easier (see the Regex tutorial).

Chapter 7

Advanced R

7.1 Introduction

By now you have mastered the fundamentals of base R, visualizations, and data science!

In this tutorial, we will cover some a few of the more advanced but extremely useful topics.

7.2 Getting Started

Before beginning this tutorial, make sure you have installed these packages:

```r
install.packages('rmarkdown')
install.packages('dplyr')
install.packages('knitr')
```

7.3 R Markdown

R Markdown is a powerful format for quickly making high-quality reports of your analysis. You can embed code and all kinds of output, including graphs, and output them to a Word Document, PDF or website. In fact, all of our tutorial webpages are written in R Markdown, including this one!

Here we'll cover just the basics, but a complete guide to R Markdown is available online from Yihui Xie, J. J. Allaire and Garrett Grolemund. You can also check out the R Markdown documents that we use to make our tutorial websites on our GitHub Pages (the website files have .html extension and the R Markdown files have the same name with .Rmd extensions):

- Main Colautti Lab Resources Website and GitHub Repository
- R Tutorials and GitHub Repository

7.3.1 Cheat Sheet

There is a very handy 2-page 'cheat sheet' that you can print out to help you remember some of the main commands. I use R Markdown for all kinds of documents – including course tutorials like this one – so I have the 2-page cheat sheet printed out and taped to my wall next to my computer.

You can also access cheat sheets for R Markdown and several others R Studio also includes a number of cheat sheets under the *Help* menu: `Help -> Cheatsheets`

7.3.2 Create

In RStudio: `File-> New-> R Markdown`

Choose `Document` from the left-hand side menu

Make sure `html` is selected

Then click `OK`

Very few elements are needed for a basic markdown file, and these are provided when you create a new file.

7.3.3 YAML Header

This is generated automatically when you make a new .Rmd file in RStudio. Depending on what options you choose, it might look something like this:

```
---
title: "Untitled"
author: "Robert I. Colautti"
date: "January 20, 2019"
output: html_document
---
```

There are other options available for YAML, and you can includes a separate `_output.yml` to set other aspects of the layoug.

7.3.4 Markdown

R Markdown is based on the markdown language, which was created as a quick and easy way to encode formatted websites in a simple text document.

R Markdown has a few additions, including the ability to easily incorporate R code, graphs, and equations.

7.3.5 Basic elements:

7.3.5.1 Plain text

Plain text is converted into paragraph format.

To start a new paragraph, press *enter* twice, so to skip a line in the .Rmd file.

7.3.5.2 Formatted text

You can format text with `*` or `_`

`*italics*` or `_italics_`: *italics*

`**bold**` or `__bold__`: **bold**

Use greater-than sign for block quotes, eg. `> TIP: quote`

> TIP: quote

7.3.6 Headers

Add headers with up to 6x `#` – more headers = subheadings:

`# Header 1`

`## Sub-Header = Header 2`

`### Sub-Sub Header = Header 3`

`#### Sub-Sub-Sub Header = Header 4`

7.3.7 Other Elements

`superscript^2` superscript^2

`--` for short–dash: –

`---` for long — dash: —

7.3.8 Links

Links have a special format. The text you want the user to see goes in square brackets, followed immediately by the file or html link in regular brackets, with no space in between:

`[Colautti Lab Website](https://colauttilab.github.io/)`:

Colautti Lab Website

You can also use this to link a file in the same project folder:

`[Linked jpeg file](./ColauttiLabLogo.png)`:

Linked jpeg file

7.3.9 Images

Or you can embed the image directly by adding an exclamation point. You can leave the linked text blank or keep it to use as a caption.

`![Linked jpeg file](./ColauttiLabLogo.png)`:

Figure 7.1: Linked jpeg file

7.3.10 Lists and tables

Lists are easy to create, simply start a line with `*` or `+` for *unordered* lists or a number for *ordered* lists. Add tab characters for sub-lists:

```
+ Unordered list item 1
* Item 2
    + sub item 2.1
    * sub item 2.2
* Item 3
```

- Unordered list item 1

- Item 2
    - sub item 2.1
    - sub item 2.2
- Item 3

```
1. Ordered list item 1
2. Item 2
    + sub item 2.1
    * sub item 2.2
3. Item 3
```

1. Ordered list item 1

2. Item 2

    - sub item 2.1

    - sub item 2.2

3. Item 3

The fun thing about ordered lists is the numbers you use don't really matter – R Markdown will automatically start at 1 and increase for each item.

```
1. Ordered list item 1
1. Item 2
    + sub item 2.1
    * sub item 2.2
1. Item 3
```

1. Ordered list item 1

2. Item 2

    - sub item 2.1

    - sub item 2.2

3. Item 3

Tables are added using vertical pipe `|` to denote columns, and a line of horizontal dashes to separate the title of the table, and dashes with pipes to separate the header row from the rest of the table. For example, this code:

```
Tables
-----------------------

Date  | Length   | Width
------|----------|------
09/09/09 | 14 | 27
10/09/09 | 15 | 29
11/09/09 | 16 | 31
```

Produces this output:

7.4 Tables

Date	Length	Width
09/09/09	14	27
10/09/09	15	29
11/09/09	16	31

7.4.1 Embed R Code

Embed R code inline using the back-tick ‘ character: `embedded code`

Note that the back-tick is not the single quotation mark. It's often on the same key as ~ on North American keyboards.

You can add larger blocks of code (multiple lines) using three back ticks ‘ and r in curly brackets. Then add three more tick marks after the code chunk:

```
#      ```{r}
#       <<your code goes here>>
#      ```
```

`Ctl-Alt-i` is a nice shortcut in R Studio for adding code chunks quickly

7.4.1.1 Code Chunk Names

You can name your code chunks, which becomes useful when making custom packages or other knitr uses. The name is added after the `r` separated only by spaces. The name cannot contain spaces. E.g. “‘`{r code-chunk-name, eval=F}`

7.4.1.2 Suppress code

You can use different options for your R code chunks, as shown on the cheatsheet. Three main ones are:

- `eval=F` – show the code but don't run it.
- `include=F` – run the code but don't show it and and don't produce any output, plots, messages or warnings.
- `echo=F` – don't show the code but run it and include any output, plots, messages and warnings.

7.4.2 Dynamic tables

Making tables from data is a bit more complicated. For example, if we wanted to summarize the `FallopiaData.csv` data, we could read in the file and then summarize with dplyr as we did in the Data Science Tutorial:

```
library(dplyr)

Fallo<-read.csv(
  "https://colauttilab.github.io/RCrashCourse/FallopiaData.csv")

SumTable<-Fallo %>%
  group_by(Taxon,Scenario,Nutrients) %>%
  summarize(Mean=mean(Total), SD=sd(Total)) %>%
  arrange(desc(Mean))

print(SumTable)
```

```
## # A tibble: 10 x 5
## # Groups:   Taxon, Scenario [10]
##    Taxon Scenario     Nutrients  Mean    SD
##    <chr> <chr>        <chr>     <dbl> <dbl>
##  1 bohem high         high       60.3  8.68
##  2 japon gradual      high       59.7  9.57
##  3 bohem fluctuations high       58.4  9.20
##  4 bohem extreme      high       58.3  7.34
##  5 bohem gradual      high       57.5  9.34
##  6 japon extreme      high       57.2 10.9
##  7 japon high         high       56.4  8.20
##  8 japon fluctuations high       56.4 13.7
##  9 japon low          low        52.0  8.29
## 10 bohem low          low        48.0  8.86
```

Table 7.2: Summary Table

Taxon	Scenario	Nutrients	Mean	SD
bohem	high	high	60.28091	8.677075
japon	gradual	high	59.72917	9.565376
bohem	fluctuations	high	58.36455	9.202334
bohem	extreme	high	58.30917	7.337015
bohem	gradual	high	57.46154	9.338311
japon	extreme	high	57.23643	10.903133
japon	high	high	56.44833	8.204091
japon	fluctuations	high	56.43692	13.724906
japon	low	low	52.02917	8.287938
bohem	low	low	47.98077	8.862164

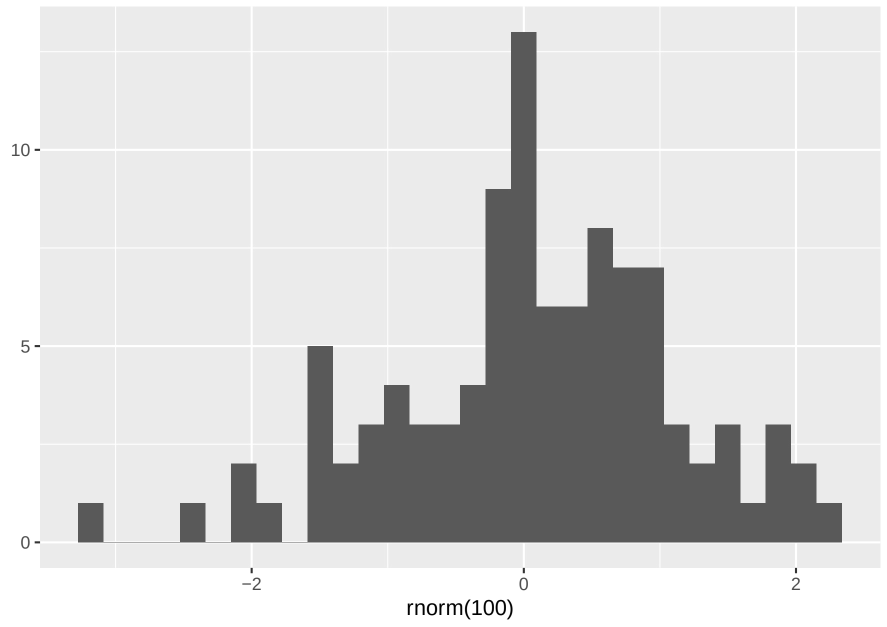
The output is legible but not very attractive for a final report. To make it look better, we can use the `kable` function from the `knitr` package:

```r
library(knitr)
kable(SumTable, caption = "Summary Table")
```

7.4.3 Embed Graphs

Use R code to embed graphs.

```r
qplot(rnorm(100))
```



7.4.4 Options for header

```
output:
  html_document: # Add options for html output
    toc: true # Add table of contents (TOC)
    number_sections: true # Add section numbers
    toc_float: # Have TOC floating at the side
      collapsed: false # Expand subsections
```

7.4.5 Content as tabs

```
## Quarterly Results {.tabset}

### By Product

(Product tab content)

### By Region

(Region tab content)
```

Looks like this:

7.4.6 Quarterly Results

7.4.6.1 By Product

(Product tab content)

7.4.6.2 By Region

(Region tab content)

7.4.7 Equations

Insert equations using LaTeX. Here is a handy cheat sheet

Use single dollar signs for in-line equations, like $Y ~ X$, which will print as $Y X$

Use double dollar signs on a new line for full-line equations, for example:

```
$$\idotsint_V \mu(u_1,\dots,u_k) \,du_1 \dots du_k$$
```

will produce:

$$\int \cdots \int_V \mu(u_1, \dots, u_k) \, du_1 \dots du_k$$

and

```
$$sum_{n=1}^{\infty} 2^{-n} = 1$$
```

will produce:

$$\sum_{n=1}^{\infty} 2^{-n} = 1$$

Notice the use of special characters with the backslash `\`, along with subscripts `_` and superscripts `_` with the super/subscripted text in curly brackets `{}`

7.5 Custom Functions

7.5.1 General form:

```
functionName<-function(var1=Default1,var2=Default2){
  ## Meat and potatoes script
}
```

7.5.2 Example function

User inputs two objects; the function outputs a list of functions applied to the inputs

```
my.function<-function(var1=0,var2=0){
  # You can make new variables within a function
  add<-var1+var2
  subt<-var1-var2
  mult<-var1*var2
  div<-var1/var2
  outlist<-list(input1=var1, input2=var2,
                addition=add, subtraction=subt,
                multiplication=mult, division=div)
  # So far, everything is contained within the function.
  # Use return() to generate output
  return(outlist)
}
```

Note: Nothing output when the funciton is run. This just loads the function into memory.

7.5.2.1 Run the function

```
my.function(var1=10,var2=0.1)
```

```
## $input1
## [1] 10
##
## $input2
## [1] 0.1
##
```

```
## $addition
## [1] 10.1
##
## $subtraction
## [1] 9.9
##
## $multiplication
## [1] 1
##
## $division
## [1] 100
```

```
my.function(var1=c(1:10),var2=c(10:1))
```

```
## $input1
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $input2
##  [1] 10  9  8  7  6  5  4  3  2  1
##
## $addition
##  [1] 11 11 11 11 11 11 11 11 11 11
##
## $subtraction
##  [1] -9 -7 -5 -3 -1  1  3  5  7  9
##
## $multiplication
##  [1] 10 18 24 28 30 30 28 24 18 10
##
## $division
##  [1]  0.1000000  0.2222222  0.3750000  0.5714286  0.8333333  1.2000000
##  [7]  1.7500000  2.6666667  4.5000000 10.0000000
```

Protip #1:

Consider annotating long or complex script with `cat()` to help with troubleshooting. The `cat` function is similar to `print` but lets you print directly to screen rather than passing through a data object. Note that `\n` is a 'new line' character; try removing them and compare the output.

```
my.function<-function(var1=0,var2=0,verbose=FALSE){
  cat("\nInput variables:\nvar1 =", var1,"\nvar2 =", var2,"\n")
  cat("\nCalculating functions...\n")
  cat("\nAdding...\n")
  add<-var1+var2

  cat("\nSubtracting...\n")

  subt<-var1-var2

  cat("\nMultiplying...\n")

  mult<-var1*var2

  cat("\nDividing...\n")

  div<-var1/var2

  cat("\nGenerating output...\n\n")

  outlist<-list(input1=var1, input2=var2,
                addition=add, subtraction=subt,
                multiplication=mult, division=div)

  return(outlist)
}

## Run
my.function(var1=10,var2=0.1)
```

```
##
## Input variables:
## var1 = 10
## var2 = 0.1
##
## Calculating functions...
##
## Adding...
##
## Subtracting...
##
## Multiplying...
##
## Dividing...
##
## Generating output...

## $input1
## [1] 10
##
## $input2
## [1] 0.1
##
## $addition
## [1] 10.1
##
## $subtraction
## [1] 9.9
##
## $multiplication
## [1] 1
##
## $division
## [1] 100
```

Better yet, make it an option:

```
my.function<-function(var1=0,var2=0,verbose=FALSE){
  if (verbose==T){
    cat("\nInput variables:\nvar1 =", var1,"\nvar2 =", var2,"\n")
    cat("\nCalculating functions...\n")
    cat("\nAdding...\n")
  }

  add<-var1+var2

  if (verbose==T){
    cat("\nSubtracting...\n")
  }

  subt<-var1-var2

  if (verbose==T){
    cat("\nMultiplying...\n")
  }

  mult<-var1*var2

  if (verbose==T){
    cat("\nDividing...\n")
  }

  div<-var1/var2
  if (verbose==T){
    cat("\nGenerating output...\n")
  }

  outlist<-list(input1=var1, input2=var2,
                addition=add, subtraction=subt,
                multiplication=mult, division=div)

  return(outlist)
}

# Run
my.function(var1=10,var2=0.1,verbose=FALSE)
```

```
## $input1
## [1] 10
##
## $input2
## [1] 0.1
##
## $addition
## [1] 10.1
##
## $subtraction
## [1] 9.9
##
## $multiplication
## [1] 1
##
## $division
## [1] 100
```

```
my.function(var1=10,var2=0.1,verbose=TRUE)
```

```
##
## Input variables:
## var1 = 10
## var2 = 0.1
##
## Calculating functions...
##
## Adding...
##
## Subtracting...
```

```
##
## Multiplying...
##
## Dividing...
##
## Generating output...
```

```
## $input1
## [1] 10
##
## $input2
## [1] 0.1
##
## $addition
## [1] 10.1
##
## $subtraction
## [1] 9.9
##
## $multiplication
## [1] 1
##
## $division
## [1] 100
```

Protip #2:

If you have a custom function, theme, script, etc., that you use repeatedly:

1. Save in a separate file

```
* e.g. make new "myfunction.R" file containing just my.function
```

2. Load using `source("PathName.FileName.R")`

```
* e.g. `source("C:/Users/Colautti/Documents/RFunctions/myfunction.R")` if save
```

Protip #3:

We have already been using functions that somebody else wrote in R.

To see 'under the hood' type a function without the bracktes:

```
my.function
```

```
## function(var1=0,var2=0,verbose=FALSE){
##   if (verbose==T){
##     cat("\nInput variables:\nvar1 =", var1,"\nvar2 =", var2,"\n")
##     cat("\nCalculating functions...\n")
##     cat("\nAdding...\n")
##   }
##
##   add<-var1+var2
##
##   if (verbose==T){
##     cat("\nSubtracting...\n")
##   }
##
##   subt<-var1-var2
##
##   if (verbose==T){
##     cat("\nMultiplying...\n")
##   }
##
##   mult<-var1*var2
##
##   if (verbose==T){
##     cat("\nDividing...\n")
##   }
##
##   div<-var1/var2
##
##   if (verbose==T){
##     cat("\nGenerating output...\n")
##   }
##
##   outlist<-list(input1=var1, input2=var2,
##                 addition=add, subtraction=subt,
##                 multiplication=mult, division=div)
##
##   return(outlist)
## }
## <bytecode: 0x7f89571d3388>
```

```
library(ggplot2)
qplot
```

```
## function (x, y, ..., data, facets = NULL, margins = FALSE, geom = "auto",
##     xlim = c(NA, NA), ylim = c(NA, NA), log = "", main = NULL,
##     xlab = NULL, ylab = NULL, asp = NA, stat = NULL, position = NULL)
## {
##     caller_env <- parent.frame()
##     if (!missing(stat))
##         warn("`stat` is deprecated")
##     if (!missing(position))
##         warn("`position` is deprecated")
##     if (!is.character(geom)) {
##         abort("`geom` must be a character vector")
##     }
##     exprs <- enquos(x = x, y = y, ...)
##     is_missing <- vapply(exprs, quo_is_missing, logical(1))
##     is_constant <- (!names(exprs) %in% ggplot_global$all_aesthetics) |
##         vapply(exprs, quo_is_call, logical(1), name = "I")
##     mapping <- new_aes(exprs[!is_missing & !is_constant], env = parent.frame())
##     consts <- exprs[is_constant]
##     aes_names <- names(mapping)
##     mapping <- rename_aes(mapping)
##     if (is.null(xlab)) {
##         if (quo_is_missing(exprs$x)) {
##             xlab <- ""
##         }
##         else {
##             xlab <- as_label(exprs$x)
##         }
##     }
##     if (is.null(ylab)) {
##         if (quo_is_missing(exprs$y)) {
##             ylab <- ""
##         }
##         else {
##             ylab <- as_label(exprs$y)
##         }
##     }
##     if (missing(data)) {
##         data <- new_data_frame()
##         facetvars <- all.vars(facets)
##         facetvars <- facetvars[facetvars != "."]
##         names(facetvars) <- facetvars
##         facetsdf <- as.data.frame(mget(facetvars, envir = caller_env))
##         if (nrow(facetsdf))
##             data <- facetsdf
##     }
##     if ("auto" %in% geom) {
##         if ("sample" %in% aes_names) {
```

```
##             geom[geom == "auto"] <- "qq"
##         }
##         else if (missing(y)) {
##             x <- eval_tidy(mapping$x, data, caller_env)
##             if (is.discrete(x)) {
##                 geom[geom == "auto"] <- "bar"
##             }
##             else {
##                 geom[geom == "auto"] <- "histogram"
##             }
##             if (is.null(ylab))
##                 ylab <- "count"
##         }
##         else {
##             if (missing(x)) {
##                 mapping$x <- quo(seq_along(!!mapping$y))
##             }
##             geom[geom == "auto"] <- "point"
##         }
##     }
##     p <- ggplot(data, mapping, environment = caller_env)
##     if (is.null(facets)) {
##         p <- p + facet_null()
##     }
##     else if (is.formula(facets) && length(facets) == 2) {
##         p <- p + facet_wrap(facets)
##     }
##     else {
##         p <- p + facet_grid(facets = deparse(facets), margins = margins)
##     }
##     if (!is.null(main))
##         p <- p + ggtitle(main)
##     for (g in geom) {
##         params <- lapply(consts, eval_tidy)
##         p <- p + do.call(paste0("geom_", g), params)
##     }
##     logv <- function(var) var %in% strsplit(log, "")[[1]]
##     if (logv("x"))
##         p <- p + scale_x_log10()
##     if (logv("y"))
##         p <- p + scale_y_log10()
##     if (!is.na(asp))
##         p <- p + theme(aspect.ratio = asp)
##     if (!missing(xlab))
##         p <- p + xlab(xlab)
##     if (!missing(ylab))
##         p <- p + ylab(ylab)
##     if (!missing(xlim) && !all(is.na(xlim)))
##         p <- p + xlim(xlim)
##     if (!missing(ylim) && !all(is.na(ylim)))
##         p <- p + ylim(ylim)
##     p
## }
## <bytecode: 0x7f896658d040>
## <environment: namespace:ggplot2>
```

7.6 Custom R Package

Most of the general content can be found in Hadley Wickham's R Packages book. It goes into detail on almost everything you would need to know to make a package.

For a quick tutorial, see Hilary Parker's post on a "cat" function.

Install packages first and then read on.

```
install.packages(c("devtools", "roxygen2", "testthat", "knitr"))
```

Tutorial objectives:

1) Make a basic package in RStudio and make 1 function.
2) Make documentation for the function.
3) Installing the package and input checking.

7.6.1 Introduction

When should you use a function vs write a package?

Start with a piece of code, and be sure to add a comment to explain what the code does.

```
# take x, square it and add one to it
y <- x^2 + 1
```

If you are going to use that piece of code multiple times, it's easier to make it into a function and call it, rather than copying and pasting the same cod multiple times. (See functions tutorial LINK)

```r
square_plus<-function(x){
  # take x, square it and add one to it
  y <-x^2 + 1
  return(y)
}
square_plus(2)
```

```
## [1] 5
```

```r
square_plus(4)
```

```
## [1] 17
```

```r
square_plus(1:10)
```

```
##  [1]   2   5  10  17  26  37  50  65  82 101
```

If you want to use the function across many scripts, you can save the function in its own .R file, and then load it into each script

```
source("/Path/To/CustomScript/Script.R")
```

When you have many functions that you use frequently, you might want to make your own R package so that you can load all the functions easily and quickly. This also makes it easy to share the functions.

```
square_plus<-function(x){
  # take x, square it and add one to it
  y <- x^2 + 1
  return(y)
}

cube_plus<-function(x){
  # take x, cube it and add one to it
  y <-x^3 + 1
  return(y)
}

quartic_plus<-function(x){
  # raise x to the power of 4 and add one to it
  y <-x^4 + 1
  return(y)
}
```

7.6.2 Getting started

R Studio makes it easy to create your own packages for R. Once you have installed devtools (see above), create a new R package in R Studio

```
File -> New Project -> New Directory -> R Package
```

You can also use the "R Package" option but delete the NAMESPACE file as it will be automatically generated later. Give the package a name and then click create.

RStudio should load and there will be a file structure with several files and two folders, "R" and "man".

The "R" folder is for code, and there is a hello.R file in it. Save all of your custom functions here.

The "man" folder is for manual pages, the documents that show up when you use the ? for help `?some_function`.

7.6.3 Add functions

We are going to make a function to get public references from the Crossref API. Crossref is one of the organizations for Digital Object Identifiers and is frequently the one used for scientific journals. Crossref has "metadata" on digital objects such as type of object, author, dates etc etc.

We can access this information through the Crossref page.

For example, the link below shows the result of a search for the title of a paper by Primack and Miller-Rushing (2011).

https://search.crossref.org/?q=Broadening+the+study+of+phenology+and+climate+change

The DOI for the first paper (by Primack & Miller-Rushing) is "10.1111/j.1469-8137.2011.03773.x"

Clicking on the "Actions" button for this paper, and then "Metadata as JSON", brings up a json file including citation information, and also citations for the papers referenced in the paper.

There's been times where we read a paper and then go through the references of the paper, especially for literature reviews/meta-analyses. Automating the extraction of references from a paper of interest might be useful.

We can access the json file in R using the doi and using the Crossref api (documentation LINK).

Doing this will give us a list of the citations:

```r
# download jsonlite to parse json files
library(jsonlite)
url<-"https://api.crossref.org/works/10.1111/j.1469-8137.2011.03773.x"
result<-fromJSON(url)
```

`result` is a list containing a variety of information

```r
names(result)
```

```
## [1] "status"          "message-type"    "message-version" "message"
```

`result$message$reference` is a data frame of citations containing 17 references that we can extract this.

```r
references<-as.data.frame(result$message$reference)
```

This can be easily writen to csv or other formats.

But we can also make this a function for any DOI of interest.

```
get_work_references<-function(DOI){
  url<-paste0("https://api.crossref.org/works/",DOI)
  result<-fromJSON(url)
  return(as.data.frame(result$message$reference))
}
```

We can save the script above in the R folder to make it part of the package.

Of course, we would want to add more functions in order to make this a useful package.

If we want to run the function, we can use the `source()` function as described above. But as a package this still lack two important pieces:

1. Documentation for this function (and any other functions we add)
2. A library that would let us load all functions using the `library()` function

7.6.4 Adding documentation

The first piece of documentation is the DESCRIPTION file. There are several fields to fill in for this.

1. The **package name** is already filled automatically.

2. We should add a **title** (ie. This Package Gets References).
3. Change the `Author` to `Authors@R` and add yourself as the author and creator.

```
Authors@R: person("First Name", "Last Name", email="email@email.com", role=c("aut", cr

# Two authors
Authors@R: c(person("First Name", "Last Name", email="email@email.com", role=c("aut", 
             person("Second person name", "second person last name", email="email@emai
```

4. Write a description:

   Interfacing with Crossref's API to get citation information using DOI. This package uses jsonlite and contains only one function. etc…

5. Choose one of the public licenses such as GPL-3, MIT etc. (see Wikipedia)

Save the DESCRIPTION file.

7.6.5 Add a Manual

The `roxygen2` package can be used to make manuals for R packages. This greatly simplifies the writing process, which otherwise would be written in LaTeX. The `roxygen2` package allows us to make comments directly in the script, and then `roxygen2` automatically generates the manual pages from these comments.

First, take out any library(*) commands and use packagename::function() for any functions from other libraries. Read R Packages - R code for more details on why.

When writing your comments, follow this logic:

- Roxygen2 commands start with `#'`.
- The first line is automatically the title field and should cover only one line.
- The next text paragraph goes into the description. The usage field is automatically generated.
- Use `@param` tags for arguments. (Only 1 in this case).
- Use the next line to write a longer description.
- Use `@return` to write what is expected output and `@example` to write example code that will be run when creating the man page.
- We also want to use a `@export` tag so that the function will be available for use when the library is loaded.
- We need to add details such as description, useage, arguments.
- The script would look something like this:

```
#' Takes a DOI and returns references for the object.
#'
#' This function queries the Crossref API to obtain a data frame of references for the DOI. We us
#'
#' @param DOI String. Digital object identifier.
#'
#' @return data frame of references.
#' @example
#' get_work_references("10.1111/j.1469-8137.2011.03773.x")
#' @export

get_work_references<-function(DOI){
  url<-paste0("https://api.crossref.org/works/",DOI)
  result<-jsonlite::fromJSON(url)
  return(as.data.frame(result$message$reference))
}
```

Save the file and use `devtools::document()`.

We will now have a NAMESPACE file, and a new file within the man folder. The NAMESPACE file shows the function we have which will be available in the environment when the library is loaded.

Open `get_work_references.Rd` and then click preview to see how it looks.

However, our man page is a bit dull, and lacks the links most pages have. We have to add the links using code. For example, linking the paste0 function will be `\code{\link[base]{paste0}}`.

Use `document()` again. Now the functions are in monospace font. The actual links only appear when the package is built.

We can use the "CHECK" button on the "Build Pane" to check for any issues in the package.

We did not import the jsonlite package. To do this, go back to the DESCRIPTION file and add:

```
      jsonlite
```

Another CHECK will tell you that the package `curl` is required. Add this to the imports as well.

7.6.6 Installing the package

Once you pass the check, click "Install and Restart" to install the package. The package should be in your "Packages" pane.

Doing `?get_work_references` will bring up the help page with working links. We can successfully run the example. But if input is not a character, the function doesn't work.

7.6.7 Input checking

You cannot account for every possible scenario where the function doesn't work. Or there are certain variables you know have to be in a specific form.

You can add checks for inputs within the function.

For example, the DOI should be a character string. We can add a test for the input and stop the function with an error if the input isn't a character string.

```
get_work_references<-function(DOI){
  if (!is.character(DOI)) stop(" 'DOI' must be a character string")
  url<-paste0("https://api.crossref.org/works/",DOI)
  result<-jsonlite::fromJSON(url)
  return(as.data.frame(result$message$reference))
}
```

Reinstall and you have a working package!

7.7 Make it public

Before going public:

1. Create a public GitHub repository and push your project
2. Flesh out documentation and meta-data
3. Error checking with `devtools` library * `spell_check()` * `check_rhub()` – use rhub to check for errors. More about RHub. * `check_win_release()` – check for errors on Windows with latest release version of R * `check_win_devel()` – check for errors on Windows with latest pre-release of R * `release()` – release to the world!
4. Tag the latest release on GitHub

For details on this and more: http://r-pkgs.had.co.nz/release.html

7.7.1 References:

Hadley Wickham, 'R packages' http://r-pkgs.had.co.nz/

Hilary Parker, 'Writing an R package from scratch', https://hilaryparker.com/2014/04/29/writing-an-r-package-from-scratch/