CCESR Intern Hub

CCESR Fellows

Table of contents

Welcome!

This website / HTML book is intended to collect resources for Cedar Creek summer interns doing independent research projects, and present those resources in an easily accessible way.

For now, the focus is primarily on data management, analysis, and using the R programming language.

Much of the content featured is adapted from the work of past CCESR Fellows, including Maggie Anderson, Mariana Cárdenas, and Bea Baselga Cervera. The primary author of the site is Chris Wojan, with Qianna Xu providing the sections on community ecology measures.

This site is structured in different parts, which can be read in any order you choose, depending on your needs / what you already know. Currently, the first part describes data management practices, the second part goes over data analysis in general, the third part describes R-related software and workflows, and the fourth part is intended to give a primer in R coding.

Part I

Data Management

1 The Flow of Data

In any scientific research, data management is a highly important skill. As ecologists, we should strive to record, organize, document, and back up our data in such a way that it will be accessible and useful to both our future selves and other future researchers.

In general, the "flow of data" follows the this general pattern:

1. "Wild" Data - these are our observations, features of the environment we wish to record as…
2. Recorded Data - this is our first record of our observations, with which we ideally will…
    1. Back Up - store copies in an alternative medium / location
    2. Error Check - look over the data for entry errors
    3. Other Processing - ensure proper filenames, data formats, etc., leading to…
3. Processed Data - this is the data that we can use to analyze and answer scientific questions!

The following chapters will go into detail about this flow, with tips and various things to consider.

2 Recording Data

First, let's go over how we can record "wild" data!

2.1 Data Formats

One thing to think about when recording your data is the formatting. First, you generally want to have data arranged in a table such that there is one row at the top that acts as a "header", labeling each column. The rest of the rows should just be values of each of those columns. For example:

Variable 1	Variable 2	Variable 3
1	Blue	45.3
2	Blue	36.9
3	Red	39.1
4	Green	41.2

Data that is formatted like this is easiest for scientific computing environments to interpret. Data like this can be saved as a simple text file, which allows for easier storage, accessibility, and future-proofing.

Generally, you will want to avoid breaking the pattern of a simple table, like repeating header rows, using rows or columns for organization, or merging cells.

In data science there is also a commonly used dichotomy is between "wide" and "long" data formats. You are likely more used to wide data, but long data can be more efficient for scientific computing and analysis. Here is a comparison:

Wide Data

- Easy for humans to read
- Features unique rows for an identifier, with columns describing attributes

Example table of species abundance (attributes) at four research plots (identifiers):

Plot	Species A	Species B
North	3	3
East	0	3
South	1	0
West	2	0

Long Data

- Easy for computers to read
- Features multiple rows for each identifier value, one for each attribute

Example table of the same data from above:

Plot	Species	Count
North	A	3
North	B	3
East	A	0
East	B	3
South	A	1
South	B	0
West	A	2
West	B	0

As you can see, long data take more space, but it is easier to select and compare data in a computing environment when it is in this format. You will see this in action in the R programming section of this website!

2.2 Data Sheet Design

Whether you design a data sheet using long or wide format, you will need to use some sort of computer application, either a spreadsheet program, word processor, or a combination of both. You are probably familiar with many of the options, but here are a few -

Spreadsheet Applications:

- Microsoft Excel - Standalone program, often free with university accounts
- Google Sheets - web-based, free to use with Google account, can be integrated directly with R
- LibreOffice Calc - Standalone program, free and open-source alternative to Excel

Word Processors:

- Microsoft Word - Standalone program, often free with university accounts
- Google Docs - web-based, free to use with Google account
- LibreOffice Writer - Standalone program, free and open-source alternative to Word

Sometimes it can be finicky to design a data sheet directly in a word processor's table functionality. Instead, you can create the basics of your table in a spreadsheet application, and then copy and paste it into the word processor, and then fine-tune the table to your liking.

When designing your sheet/s, wide data will often be more intuitive and easier for recording, but note that there may be more processing steps required when you get to the analysis stage (and if the data are complex, the data manipulation required can also be complicated). On the other hand, recording data in long format, while somewhat tedious, will be efficient for later processing and analysis steps.

In general, wide format for recording data is effective for simple observations when you know exactly how much you are going to record (e.g., a suite of soil attributes for a set of plots). Long format can be effective when the observations you're recording are more complicated and/or you don't know how many observations you will be making in a given plot (e.g., recording each plant species with it's cover class, max height, and disease severity in a set of plots - you might not know how many species you will encounter in each plot!).

Universal Tip: Always include a "Notes" column!

2.3 Data Sheet Medium

When it comes to recording data in the field or in the lab, there are two main styles: using paper data sheets or using electronic spreadsheets/apps. Opinions vary among researchers.

Paper / Physical Media

- Traditional method for ecology research
- Results in having both a hard copy and electronic copy of the data
  - Backups can include photos/scans of the paper data sheets, as well as back-up electronic files
- Less risk of technical glitches
- Need to consider material; write-in-the-rain and pancil often necessary in field settings
- Needs a solid organization scheme to make sure paper sheets don't get lost or misplaced
- Requires data to be entered from paper to computer

  - But this does leave the chance for checking for entry errors between paper and computer

Electronic Media / App-based Recording

- Becoming more popular among large-scale projects (e.g., National Ecological Observatory Network)
- Some projects use propietary software for data recording, but you can use the Google Sheets app
- Data is usually only digital files, but can be printed out for hard copy back-ups
- Somewhat elevated risk for technical glitches / user error
- Can utilize immediate cloud backup
- Phone data entry can be particularly convenient for certain field methods
- Error checking is limited to looking for obvious field entry errors (e.g., impossible attribute values)

3 File Organization

As our recorded data moves into electronic form, either directly from our field/lab observations or via data entry from paper data sheets, it is important to consider how we organize and name our electronic files. It may sound boring, but it is valuable to think about!

Universal Tip: Be consistent!

3.1 Organize

When you are developing an organization scheme, there are many ways you can proceed. Here are a few:

File Organization Strategies

- By Stage - collection materials, raw data, processed data, shared data, etc.
- By Data Type - text, images, spreadsheets, etc.
- By Research Activities - experiments, field observations, interviews, etc.

The best way to organize your files will depend on the project and the researcher, but as long as it is consistent, it is effective.

The strategies listed above generally work best as coarse-scale organization schemes, but one strategy could be used as a subfolder scheme within another strategy.

Date and location generally work best as subfolder organization schemes within one of the above strategies. One exception to this might be when you are doing radically different things among years or locations.

When you are downloading research related files, it is best to move them from the "downloads" folder as soon as possible, or better yet, download directly to the appropriate folder and subfolder. And avoiding downloading important files to the desktop is generally helpful.

Finally, Do not use your folders as the sole description of your files, i.e., use descriptive file names! A file named "plot3" may make sense in your "2023_BioCON" folder, but it will lose context when moved or shared. This brings us to our next section…

3.2 Name

Descriptive file names should be easily scannable and sortable.

Key components of a descriptive file name often include:

- Project name
- Date (YYYYMMDD)
- Type of data
- Location/site
- Researcher Info
- Version

For sorting purposes, it is best to use numerical date format and leading zeroes. Numerical date format is YYYYMMDD, or 20240710 instead of something like 7-10-24. When sorted, everything will be in calendar order. Sorting other date formats can lead to files being out of order, like how 12 comes before 7 in an alphanumeric sort, so December could come before July in your files. Similarly, since 10, 11, 12, etc. come before 2 in an alphanumeric sort, it is good to use leading zeroes depending on the number of numeric IDs you have. For example if you have 12 plots, use one leading zero for plots 1-9 to ensure proper file sorting (e.g., “01”, “02”, etc.).

Universal Tips:

- Don’t use special characters (*@,.^?#!<>)
    - These are incompatible with many file systems
- Be concise, AKA brief but efficiently informative

4 Documentation and Storage

“Your primary collaborator is yourself 6 months from now & your past self doesn’t answer email!”

- Rachel Ainsworth

Taking notes as you conduct research and backing things up are both generally good practice. Here are some things to think about regarding both:

4.1 Document

What to document:

- File-specific info or metadata
  - file naming patterns
  - variable/attribute definitions
- Technical steps
  - data collection
  - processing (what did you include or exclude?)
  - analysis steps
- External data sources
- Software used with version numbers
- Meeting notes
- Organization schemes

Where to document:

- Lab Notebook
  - Can be physical or digital

    - Many scientists use pen for this sort of thing

- Field notebook
    - Generally easier to be physical if taking notes in the field, can be digital if taking notes at end of day
- ReadMe file
    - Good for computer work - data processing and analysis

When to document:

- Set aside ~15 minutes after your work session
- Develop a regular routine during the research process
    - Easier to do than remembering everything at the end
- Can also note things opportunistically as you go
    - jot potentially relevant observations down in a notebook
    - comment out code

4.2 Store

The methods of backing up your data may vary somewhat based on medium, but what remains important universally is to maintain multiple formats of backups, and it is helpful to have one backup in an alternative physical location.

Paper Data Sheets

- take a photo or scan each newly filled data sheet after each day of data collection
- Back up entered data files in cloud storage or some other location

Electronic Data

- Download a dated copy of all electronically entered files after each day of data collection
    - You can write an R script to do this for you with Google Sheets

You may also run into some issues when storing backups. You may run out of storage space, especially on personal cloud storage accounts. However, usually university or organization based accounts have large storage limits. In addition, physical data drives (either hard drives or solid-state drives) can fail, so it's good to have cloud-based storage and/or alternate physical drives (another computer or external drive) as fail-safes. Of course, cloud storage providers can change terms in the near future, so be prepared for adapting your plans.

5 Error Checking and Data Processing

Finally, when you have your data recorded, organized, documented, and backed-up, there are a few things to think about before analysis: checking for errors and late-stage processing steps.

5.1 Error Checking

Your error checking method may vary depending on the medium on which your data were recorded, but a general note is that you can elect to error check everything, or a certain percentage of all the data when data are numerous. Different organizations will follow different practices. Oftentimes personal research projects are small scale enough that full error checking is feasible.

Paper Data

- Cross reference digitally entered data with paper sheet for errors in entry
    - Helpful to have two people
- Note any corrections in a new column
- Hard to check field recording errors, but make any corrections in a different colored writing utensil for clarity

Electronic Data

- Check version control (e.g., Google Drive files) for unusual edits, and roll back where necessary
    - Note when you rolled back a change in a new column

5.2 Data Processing

When you are getting ready to analyze your data, the best file formats are text-based with some sort of delimiter between columns.

The most popular of these file formats is the “comma separated value” file, or .csv. It is small in size, as it is a simple text file where commas denote new columns and line breaks denote new rows. This is easy for a wide variety of computer programs to interpret, and will most likely continue to be efficient for the foreseeable future.

Unlike Excel spreadsheets or Google Sheets, text based format do not support multiple sheets, cell/text colors, or special formatting. As such, do not rely on color or font for describing your electronic data.

You can create .csv files form Excel using the “Save as” command and selecting the csv format, and you can download a Google Sheet as a csv as one of the options. You can also “publish” a Google sheet as a csv on the web to import directly into R, but that is a topic for later in this site (see 15.1.3 Importing Data; From the Web)

Part II

Data Analysis

6 Data Analysis at a Glance

Analyzing your data is usually about transforming long spreadsheets into a form that is relevant to your question/s, and oftentimes including an appropriate statistical approach for inference.

You might use **descriptive statistics**, which is simply *describing* what you observed without presenting every data point, and instead a summary of those data. This can often be helpful in providing a frame of reference to your dataset before looking deeper at trends and comparisons. Alternatively, sometimes descriptive statistics are the main goal - like in surveys of populations and communities (e.g., what is the population size of a certain grass of interest in an old field?). Descriptive statistics include things like the mean and variance, but can also include less commonly used measures like dispersion (which is simply the variance : mean ratio!).

You could also use **inferential statistics**, which is more about using math or simulation techniques to *infer* some conclusion from the shape of your data. This is directly relevant to when you have an ecological question about cause and effect, associations among variables, comparisons among categories, etc. The results of inferential statistics provide a starting point from which to interpret/discuss an answer to your question. Examples include t-tests and linear regression.

When using both of these types of statistics, you should be mindful of **data types**, which are the form that variables take. For example, the height of a tree is number, but the species of a tree is a category. This contrast is obvious, but there are subtle differences that can be important for how you describe, assess, and plot your data. We will go over data types in the very next section!

7 Data Types

As mentioned in the last section, we will now go over different types of data:

7.1 Numeric Data

Any data that can be described with numbers or have quantifiable relationships between values is numeric. But! There are multiple types of numeric data. The most important distinction is **discrete** vs **continuous**.

Discrete numeric data is data where not every value is possible, but you can still quantify specific differences among the possible values - the major example being integer values (1, 2, 3, the rest). Most programming languages will refer to this type as "integer" or "int". Examples might include number of ants on a log.

Continuous numeric data is data where every value is possible! So this is basically all real numbers, including decimals (1.0, 1.1, etc.). Many programming languages will refer to this as simply numeric data, but lower level languages might use "float" or "double". Examples might include the biomass of ants on a log. Note: measures that consist of very large integer values (e.g., when you get up to 4 digits) are approximately continuous.

Other things to consider with numeric data is whether the scale of measurement is bound by any values. For example, the number of or biomass of ants on a log cannot be less than zero. In addition, percentages and proportions are bound by 0 and 100 and 0 and 1 respectively. These limitations can lead to special considerations when performing inferential statistics.

7.2 Categorical Data

Any data for which the values have no specifically quantitative difference among them is categorical. Again there is one majorly important distinction: **nominal** vs **ordinal**.

Nominal data is data where categories have no ranking or order, like the species of ants on a log.

Ordinal data is data where categories have some order, like your top 5 favorite breakfast cereals. But wait! You may be thinking - "isn't this quantitative?" Well yes and no. The difference between ordinal data and discrete numeric data is that you can't really quantify the exact difference between ordinal data values. Say there is a go-kart race between Mario, Luigi, and Peach. The place that each finished would be ordinal, e.g., Peach got 1st and Luigi 2nd, but you wouldn't be able to say how much faster Peach was than Luigi. The time it took for Peach and Luigi each to finish the race would be a numeric variable, and there would be a specific value difference between them.

8 Descriptive Statistics

Now, let's discuss how to describe your data:

8.1 Centrality

You'll often want to describe the central tendency of your data - around where are the values centered?

Mean - the average of the values, or the sum of all values divided by the number of observations

Median - the value at which half of the observations are greater, and the other half are less

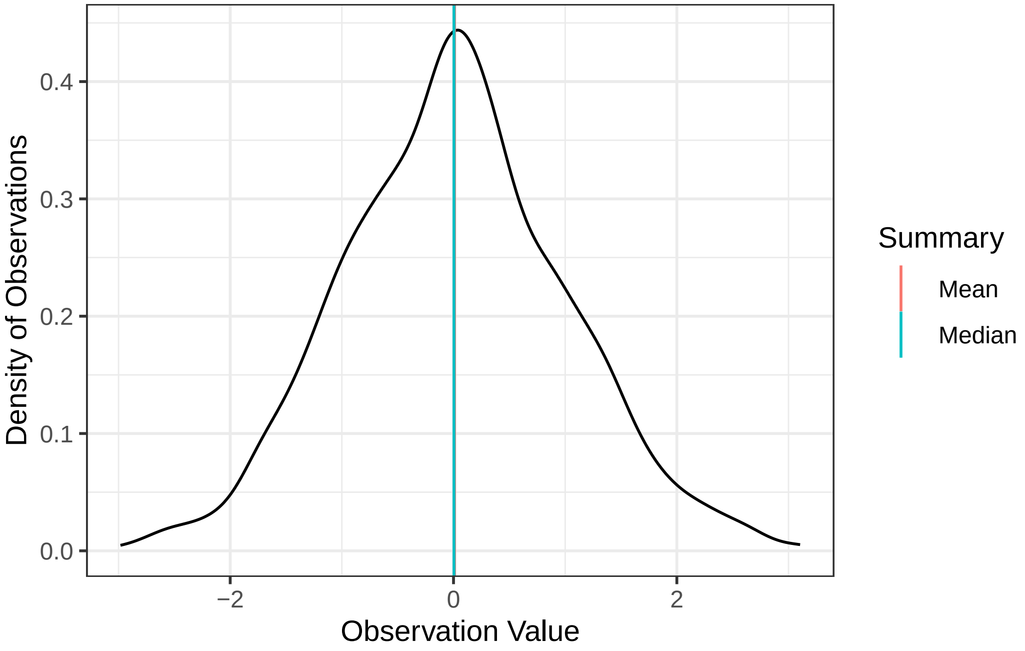
Mode - the most commonly observed value

Usually, the mean is a perfectly adequate descriptor. You can use it on continuous numeric data, discrete numeric data (though the mean value will often be unrealistic), or even ordinal rankings.

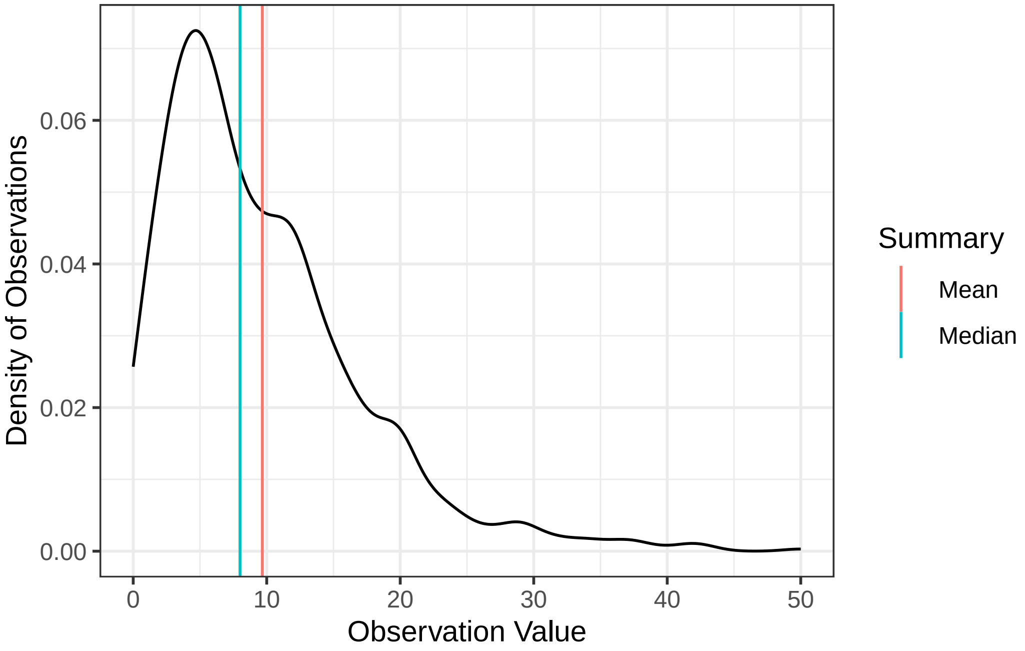
When might you prefer to use the median over the mean?

When the data is skewed such that there are many small values and a few big values, the mean might be inflated by those large values, and thus overestimate the central tendency in some contexts.

When data is roughly normally distributed, the mean and median are roughly the same:



But when data are skewed, the median may be a better estimate of the central tendency:



8.2 Spread

You also might be interested in how varied your data is, how much it deviates from the central tendency. This can be done with the following:

Variance - how variable is the data? Measured as the average squared difference between observations and the mean:

$$Variance = \frac{\sum(Observation_i - Mean)^2}{NumberofObservations}$$

($\sum$ means “sum of”)

The differences are squared to get rid of negative differences, because other wise everything would cancel out and our variance would be zero!

Standard Deviation - the square root of the variance. This is useful because it is in the same units as the original measurements!

8.3 Other Descriptors

Another descriptor that may prove useful is the **dispersion**, or the variance divided by the mean. This provides an estimate of how skewed the data is - for example, the first plot above has very low dispersion, while the second plot has high dispersion.

8.4 Ecological Community Descriptors

Many of you are interested in describing the species composition of of community. Here are a few common descriptors:

Species Richness - this is just the number of different species present in a given area or sample. Simple comparison of species richness is only possible when the data are “equivalent”, that is, your same sampling effort is the same for each replicate. If you are using quadrats to sample plant species, for example, you should use the same size, as a larger quadrat will generally contain more species.

Diversity indices - If two sites have the same number of species they obviously have the same species richness but there can be large difference in the relative abundance of the species. Various diversity indices can takes into account the richness as well as the relative abundances of each species. The result of a diversity index is a value that increases not only as the number of species increases, but also as the relative abundance becomes more even. The two most commonly used indices are Simpson’s and Shannon.

- **Simpson's Index** - uses the abundance of each species (n) in a sample to determine the proportion that each species contributes to the total. This index varies from 0 to 1, with the higher value indicating a more diverse sample. The formula for the 'classic' Simpson's diversity index (n = the abundance of an individual species/taxon; N = the total abundance):

$$D = 1 - \sum (\frac{n}{N})^2$$

- **Shannon Index** - uses the proportions of the various species but in a slightly different way - using a logarithm. The proportion of each species is not squared but multiplied by the log of the proportion. This index varies from 0 with no upper limit (although values are usually <10). Higher values indicate higher diversity:

$$H = -\sum (\frac{n}{N}) Ln(\frac{n}{N})$$

Species Evenness - this is an index that estimates specifically how evenly distributed species abundances are. E.g., Pielou's Evenness, which ranges from 0 to 1, with 1 meaning that each species has equal numbers.

Diversity at different scales - diversity is often explored at different spatial scales.

- *Alpha* diversity is the smallest scale or measurement unit, most likely from a single sample in a habitat.
- *Gamma* diversity is the largest scale of measurement, where you combine measurements of alpha diversity into one large sample over a large area (e.g. landscape or habitat).
- *Beta* diversity describes the change in diversity between smaller sampling units (at alpha level).

Similarity and dissimilarity - compares samples and allows you to see which are most similar or dissimilar to one another. When looking at communities, analysis of similarity may be more useful in many cases than simply looking at diversity.

- **The Jaccard index of similarity** - often used when you only know the presence or absence of species in samples. The Jaccard index is calculated simply as the number of species that are shared by the two samples divided by the total number of species that occur in both samples combined. The higher the index the more similar the two samples (the index can have values between 0 and 1). In the formula A = number of species in first sample, B = number of species in second sample, and J = number of shared species:

$$Jaccard = \frac{J}{A + B - J}$$

- **The Bray-Curtis dissimilarity index** - a more ‘sensitive’ measure of dissimilarity when abundance data is available. The index is always a number between 0 (two samples are identical) and 1 (two samples don’t share any species), and can be calculated with the following formula:

$$BC_{ij} = 1 - \frac{2C_{ij}}{S_i + S_j}$$

Note: these measures can apply to any taxonomic distinction, e.g., family richness, order diversity, etc.

9 Inferential Statistics

Now, let's think about how to use your data to answer your questions. There are a couple of approaches statisticians use, and we will talk about frequentist statistics, where probabilities are thought of like relative frequencies. There is also Bayesian statistics, which is a bit more complex, so we will skip it for now.

Within frequentist statistics, we can run various tests to see how variables are related, which typically make some assumptions about the data. We can also do something called bootstrapping, which makes no assumptions, but can be simplistic from some perspectives.

In any case, we are hoping to estimate two main values:

Effect Size: How related are two variables? How different are two means? How much does one variable affect another?

p-value: What is the chance of observing data like yours (or something more extreme) if there was no relationship among the variables?

The p-value can be a bit tricky, but know that it **isn't** the probability that there is no relationship. P-values are used to draw conclusions from test results, a traditional guideline is that if the p-value is less than 0.05, the results are "significant." Some statisticians bristle at this arbitrary and binary system, so it's often best to report both the effect size and the actual p-value, so readers can interpret for themselves. The smaller the effect size, the weaker the relationship, the smaller the p-value, the stronger the evidence for the relationship.

Obviously, there are entire classes taught on this stuff (which many of you may have taken or will take!), but we are thinking in just the basics for now.

9.1 Classic Frequentist Tests

Now let's go over some statistical tests! For this section, it can be useful to remind ourselves of the variables involved in a research question:

Independent / Explanatory / Predictor Variable: this is either what you are manipulating in an experiment or what your study is designed to capture variation in (e.g., C02 at BioCON, species richness at BigBio).

Dependent / Response Variable: these are what you measure or observe throughout your study, generally hypothesizing that they will differ among the levels of your independent variable (e.g., aboveground biomass in BioCON or BigBio).

9.1.1 Assumptions

We should mention what these tests generally assume about your data.

First, they assume that your **data are independent**. This just means that no two observations of your data are more related to each other in a way that isn't accounted for by a variable. Say you were comparing mean tree height between two forests - individual tree heights in the same forest would be independent, but two measures of the same tree on different days would be non-independent.

Second, they assume that the **errors are normally distributed**. This is a bit more confusing without a statistical background. An example may be illustrative - in the tree height example above, we assume that the individual tree heights are normally distributed around the mean. Without getting too much into the weeds, if you collect enough data (i.e., 30+ observations), these errors will likely be approximately normally distributed. However, things get dicey when we deal with data that is not continuous like tree height, for example, discrete count data - more on that below.

Third, they assume **homogeneity of variance**. This is another complicated one, but it mean that the variance of the errors doesn't change with the independent variable. In the tree example, we are assuming that the variance of the differences between observed tree heights and the forest mean does not change between forests.

Data that break the first assumption are difficult to deal with outside of accounting for the non-independence factor (which can severely reduce the size of your sample), but failing to meet the second or third assumptions generally leads to transforming data or using alternative tests.

9.1.2 Categorical Predictor/s, Numeric Response

9.1.2.1 Two Predictor Categories

When you are comparing numeric values from two groups, you can use a **t-test** to compare their means. T-tests can be **paired** when each observation in one group is specifically linked to an observation in the other group (e.g., masses of sibling plants in separate treatments) which can be more powerful. When the variance of values in each group is different, you can do a **t-test with unequal variance**.

The effect size here is the difference between means.

9.1.2.2 More Than Two Predictor Categories

If you have more than two groups/categories, you can use a **Analysis of Variance** or **ANOVA**. This will tell you if the means of each group are equivalent, or if there is at least one inequality. You can test for pairwise comparisons among the groups with **Tukey's test**. If you have multiple categorical predictors, you can do **two-way or three-way ANOVAs**. Tests with more than three categorical predictor variables are uncommon and harder to interpret.

The effect sizes are the pairwise difference in means.

9.1.2.3 Ordinal Predictors

When your predictor variable is ordinal, the quick and easy way to analyze it would be to convert the predictor to a numeric integer data type and proceed from there. However this is imprecise…

This section is under construction

9.1.3 Numeric Predictor/s, Numeric Response

9.1.3.1 Simple Association

When all you are interested in is whether two numeric variables are related to one another, not cause and effect, you can do a **correlation test**. **Pearson's correlation** is generally applicable for continuous data. **Spearman's correlation** is good for when you are dealing with data with non-normal distributions, like count data (it also works for ordinal data!).

The effect size here will be a correlation coefficient ranging from -1 to 1, with -1 means an inverse relationship, 0 means no relationship, and 1 mean a direct positive relationship.

9.1.3.2 Cause and Effect

When you are suppose a causal relationship between numeric variables, you can use a **linear regression**. This will use linear algebra or maximum likelihood estimation (don't worry about the finer details here) to find the best fit line that describes the relationship between two variables; where the sum of the squared distances from the observations to the line is minimized. You can also include multiple predictor variables to perform **multiple linear regression** AKA **multivariate linear regression**.

When your response variable is count data, the assumptions of simple linear regression are usually unmet, so you can use generalized forms like a **Poisson regression** or a **Negative Binomial regression**.

The effect sizes here are the parameter coefficients, i.e., how much does the response change for on unit increase in the predictor? Note: these are not straightforward for Poisson and negative binomial regression, so ask your mentor.

9.1.4 Numeric Predictor/s, Categorical Response

9.1.4.1 Binary Response

When your categorical response is only two categories (e.g., presence or absence), you can use a **binomial regression** AKA **logistic regression**. This works similarly to linear regression, but the effect sizes are measured in log odds, which is difficult to interpret, but can be transformed to estimating how the probability of observing one category value instead of the other increases with a variable.

9.1.4.2 Multiple Response Categories

Multinomial regression *(under construction)*

9.1.5 Categorical Predictor/s, Categorical Response

Chi-square test *(under construction)*

9.2 Bootstrapping

One alternative to these classic tests has no assumptions: bootstrapping. Essentially, it involves using the sampled data to simulate more samples, and compare your observations to those simulations.

Empirical Bootstrapping is where you take your actual observations and shuffle which value is associated with which observation. For example, you could take measurements of tree heights from two forests, and randomly assign forest ID to each measurement.

Parametric Bootstrapping is where you summarize your observed data and use it to generate simulated data. For example, you could calculate the mean and variance of tree heights in two forests and then generate simulated forests of trees through random pulls from a normal distribution with the appropriate mean and variance.

With both approaches, you simulate a large number of simulated datasets (1000+), and then calculate whatever you are interested in for each of those simulations, and compare the calculation from the observed data to the distribution of simulated values. For example, if you empirically bootstrap the two forests of tree heights 1000 times, and then calculate difference in means for each you will have 1000 mean difference values. The proportion of those simulated values that are equal to or more extreme than your observed mean difference is your p-value!

Part III

R on your Computer

10 R Itself

R is both a programming language and an application that you can install to your computer.

10.1 R, the Language

R is a programming language designed for statistical computing, and is often the language of choice for scientists. R is also used for data science in some business, tech, and health contexts (but many prefer Python in those areas).

As a programming language it is essentially an expandable collection of functions with syntax to perform tasks, and it could be written in any text editor. However, in order for your computer to interpret the language, it needs some software.

10.2 R, the Software

The R application allows you to run R code on your computer, and comes with a basic "console" window where code is run and output is printed, as well as a basic script editor where you can write code to run.

You can download the application from here:

https://cran.r-project.org/

If you are asked to select a mirror, simply select the nearest one (I believe Iowa State should work).

If you have a Windows machine, it should be fairly straightforward to simply download and install the "base" R from the link.

If you have a Mac, you will want to select the .pkg file that matches your processor type: x-86 for Intel processors (mostly Macs pre-2020), arm64 for Macs with the M1 or M2 chip (most Macs post-2020).

If you are using Linux, you know more than me.

10.3 R Packages

As mentioned above, R is *expandable*. You can add more functionality to R by installing packages. Packages contain more options of code to use to process and analyze data, and also do many other things.

Packages can be installed through writing R code, or by clicking some buttons in RStudio. Then they will live in a directory that was built when you installed R for auxiliary packages.

We will discuss more about installing packages in the R coding section.

11 R Studio

While you can use R with just the basic application, it is much easier and beginner-friendly to use RStudio, which is an integrated development environment or IDE. This is just an application that provides a suite of features to make programming easier for users. In fact, I'm typing this in RStudio *right now*!! Note: you must have the R application installed to use RStudio, as it relies on the R application to interpret R code.

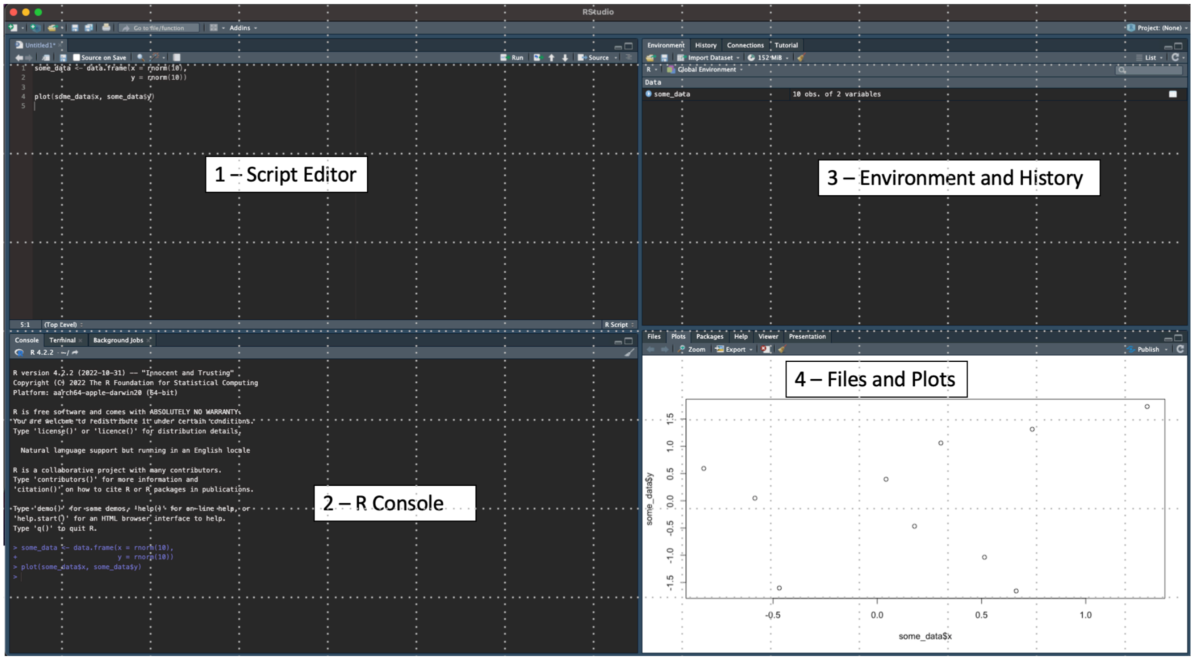
You can download and install RStudio from here:

https://posit.co/download/rstudio-desktop/

Which should be more straightforward than downloading and installing R.

11.1 RStudio at a Glance

If you open up RStudio, you will see something like this:



```
some_data <- data.frame(x = rnorm(10),
                        y = rnorm(10))

plot(some_data$x, some_data$y)
```

```
some_data          10 obs. of 2 variables
```

```
R version 4.2.2 (2022-10-31) -- "Innocent and Trusting"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> some_data <- data.frame(x = rnorm(10),
+                         y = rnorm(10))
> plot(some_data$x, some_data$y)
>
```

1- Script Editor: Here is where you will write code! You can create an R script (a text document to save code in) with the file tab, and write what you need in the resulting script. It is **highly** recommended to use scripts, because then you can save your code for later, and troubleshoot errors easier. From this window, you can highlight code and run it with the "Run" button on top, or with Ctrl + Enter / Cmd +Enter.

2- R Console: Here is where the action happens - code will run here, and text output, warnings and messages will be displayed. You can also type code into the console, but that is only recommended for installing packages, entering credentials, rendering documents, and things of that nature. Don't type your data processing or analysis code into the console, use a script instead! There's also a terminal tab if you ever need to perform shell commands (which is probably unlikely for your project this summer).

3- Environment and History: Here you can find a list of the variables and data you have loaded into your "workspace" or "environment" in the Environment tab. These are objects you can do stuff with with code. You can also click the History tab to see the code you have run thus far.

4- Files and Plots: Here is where any figures you draw will pop up (and you can save them from here as well). There is also a Files tab that allows you to navigate through your file directory (helpful with projects, described below). The Packages tab shows which packages you have installed and loaded (you can also click "Install" at top to easily install new ones!). Finally, the help tab is where you can search for the documentation on any R function.

11.2 R Projects

It is highly recommended to use R Projects when working with RStudio. Projects are essentially just subdirectories in your file folders, but they come with a special .Rproj file that RStudio can read and use. This helps you organize your work, and makes your code more easily portable.

You can create a new projects from the File tab at upper left, or in the project dropdown menu at upper right. You can just create one in a new directory. Then you can select a name and where you want to save it.

There are many different types of projects - this book/website is one!

If you want to backup your work with version control or collaborate with others using git and GitHub, you will need to use projects. (Well, technically you don't need to, but you'd be doing many things manually).

12 Optional: Git and Github

If you are interested in:

1. Backing up your code using a version control system that allows you to roll back changes and monitor incremental progress

   and/or

2. Sharing your code and collaborating with others

You may like to try using git (a program for your computer) and GitHub (a website that hosts code projects).

We won't go into detail here, but Jenny Bryan's excellent introduction and tutorial on the topic can be found here:

https://happygitwithr.com/

Part IV

R Programming

13 The Basics

13.1 Intro

I'm sure those of you reading this come from a wide variety of backgrounds regarding computer programming - some of you may be very familiar with it, others total novices. Some of you may love computing, others might hate it. If you're apprehensive about learning R, or if you find yourself struggling with it - don't worry! Scientific computing presents a challenge at some point to everyone who does it. Just remember a few things:

1. Everyone makes mistakes.
2. Don't be afraid to ask questions!
3. Don't compare yourself to others, compare the "you" of today to the "you" of yesterday.
4. Everyone is constantly learning new things, including those who seem like experts.

That said, learning a programming language is a little like learning a human language, except there's a much smaller vocabulary and the grammar is *very* strict. And where human language has parts of speech like nouns and verbs, R has a certain syntax as well. Some of the main components of the R language are **operators**, **functions**, **arguments**, and **data**.

13.2 Operators

Operators are short symbols that tell the computer to do certain simple things. You are already familiar with many operators - the **math operators** like +, -, *, and /. R at its simplest is a calculator:

```r
## This is block of R code! Anything that starts with # is a comment, and doesn't run.

## adding
2 + 2
```

```
[1] 4
```

```
## subtracting
5 - 4
```

```
[1] 1
```

```
## multiplying
3 * 3
```

```
[1] 9
```

```
## dividing
6 / 2
```

```
[1] 3
```

There are a couple other math operators too:

```
## exponentiate with ^
3^2
```

```
[1] 9
```

```
## find the remainder with the modulus, %%
10 %% 3
```

```
[1] 1
```

```
## perform integer division with %/%
10 %/% 3
```

```
[1] 3
```

But math operators aren't the only type! There are also the closely related **comparison operators**, which will return TRUE or FALSE instead of calculated numbers:

```
## equals, ==
2 + 2 == 4
```

```
[1] TRUE
```

```
## does not equal, !=
2 + 2 != 4
```

```
[1] FALSE
```

```
## greater than, >
5 > 4
```

```
[1] TRUE
```

```
## less than, <
5 < 4
```

```
[1] FALSE
```

There are also greater than or equal to (>=) and less than or equal to (<=).

You can combine comparisons with **logical operators** - and (&), or (|), and not (!):

```
## and: are both true?
(3 > 2) & (4 > 3)
```

```
[1] TRUE
```

```
## or: is at least one true?
(2 == 1) | (4 < 3)
```

```
[1] FALSE
```

```
## not: is this false?
!(2 == 1)
```

```
[1] TRUE
```

There are few other important operators, but they will make more sense once we talk about the other parts of R.

13.3 Functions

Functions are words (though not necessarily real words) or letters that instruct the computer to perform more complicated tasks. They generally are followed by parentheses ().

```
## here's a function that returns the current date
Sys.Date()
```

```
[1] "2024-07-31"
```

```
## and here is a function that returns the date with the time
Sys.time()
```

```
[1] "2024-07-31 11:25:27 CDT"
```

No you may be thinking - “this is pretty basic” and “what are the parentheses for?”, which brings use to arguments!

13.4 Arguments

Arguments are values or objects that go inside the parentheses of functions to specify what you want the function to do. This is what gives functions their power. Arguments are separated inside a function by commas.

```
## the sum function can sum many numbers
sum(1,2,3,4,5)
```

```
[1] 15
```

In the function above, each number is acting as an argument. In this case, the arguments don’t have names. Oftentimes a function’s arguments will be explicitly named, and to specify what you want those arguments to be, you use the = operator.

```
## this function pulls values randomly from a normal distribution specified in the arguments
## n specifies how many numbers to return, and mean and sd specify shape of the distribution
rnorm(n = 10, mean = 5, sd = 1)
```

```
 [1] 4.471736 4.747955 4.944738 6.976598 3.899062 4.158518 5.562331 5.741905
 [9] 3.990851 5.845077
```

Operators are actually a special type of function that can be used with syntax that is more intuitive for them. You can also use them in the same way as most functions by surrounding them with back ticks, ‘.

```
## here we use the + operator in a much more confusing context
`+`(2, 2)
```

```
[1] 4
```

```
## it is equivalent to
2 + 2
```

```
[1] 4
```

13.5 Data

We are using the word data here to broadly encompass **values** (like the numbers we were using above, both with operators and as arguments), **variables** (stored values), and **data structures** (organized collections of values).

13.5.1 Values

Values are much like the data types we discuss in the data analysis section. In fact, the different types of values R can deal with are called data types as well!

In R, values can be numeric, character, or logical (among other, more specific types).

```
## numeric values are numbers!
2
```

```
[1] 2
```

```
2.5
```

```
[1] 2.5
```

```
## character values are letters, words, phrases (often referred to as "strings")
"a"
```

```
[1] "a"
```

```
"apple"
```

```
[1] "apple"
```

```
"there is a worm in my apple"
```

```
[1] "there is a worm in my apple"
```

```
## note: character values or strings must be surrounded by "" or '' for R to interpret them

## Logical values are TRUE or FALSE (you've seen these above)
TRUE
```

```
[1] TRUE
```

```
FALSE
```

```
[1] FALSE
```

There are other types of values too: missing values (NA and NaN), infinite values (Inf and -Inf), and something that indicates empty (NULL).

13.5.2 Variables

Variables are named values that are stored in the "environment", or the workspace that R can access to perform its tasks. In order to store a value as a variable, you need to use a special kind of operator called an **assignment operator** (<- or =). As I mentioned variables have names, which are unquoted text.

```
## store 2 as a variable called x
x <- 2

## R returns no output here because you're just storing a value
## but you can return the value by calling the variable
x
```

```
[1] 2
```

```
## store 3 as a variable called y
y <- 3

## you use variables with operators
x + y
```

```
[1] 5
```

```
## store a character value
string <- "hello"

## math doesn't work on strings
```

Technically, you can use = in place of <-. This is why the equals operator is ==. I generally use <- to prevent any confusion between assignment and comparison.

13.5.2.1 Naming Rules

Variables have rules about how they can be named:

1. No special symbols other than _ and .
2. You can't start with a number or _.
3. They can't be special words that R interprets differently. You can enter ?Reserved in your console to see a list.

13.5.3 Data Structures

Data structures are collections of values with some sort of organization, and also saved in the environment. Plot twist: the variables above are the simplest data structure, the **scalar**, which is just a single value.

The next data structure is the **vector**, which is a collection of values of the *same data type*. We can store them much like variables.

```
## we use another operator, :, to create a sequence of integers from 1 to 5
my_vector <- 1:5

my_vector
```

```
[1] 1 2 3 4 5
```

```
## you can also create vectors with the combine function, c()
my_other_vector <- c("a", "b", "c")

my_other_vector
```

```
[1] "a" "b" "c"
```

The next data structure is called a **list**. A list is a collection of values like a vector, but they can be of any data type, or data structure. You can have a list of numeric values and character values, a list of vectors, or even a lists of lists! Every other complex data structure is technically a list with special attributes and/or rules.

```
## you can create lists with the list function
my_list <- list("a", 1, 2:4)

my_list
```

```
[[1]]
[1] "a"

[[2]]
[1] 1

[[3]]
[1] 2 3 4
```

```
## can also use the combine function, but it will default to a vector when data types are th
my_other_list <- c("b", 2)
```

Finally, the most common special type of list you will use is the **data frame**. A data frame is a list of vectors that are arranged in a table, much like an excel spreadsheet. Each of the vectors will be named as a column, and all must be the same length. The position of a value in a vector is its row in the data frame.

```
## we can make a data frame with the data.frame function
my_data <- data.frame(letter = c("a","b","c"), # each column has a name
                      number = c(1, 2, 3),
                      vowel = c(TRUE, FALSE, FALSE))

my_data
```

```
  letter number vowel
1      a      1  TRUE
2      b      2 FALSE
3      c      3 FALSE
```

Next, we will extend these concepts a bit further!

14 Next Steps

Now that we have the basic “parts of speech” of R down, we can move toward what we can do with them.

What follows are a few unconnected topics that will prove as useful background to working with data in the later sections.

14.1 Packages

Packages are collections of R functions that people write to make tasks easier. One of the strengths of R is that countless programmers have taken the time to assemble functions of use in their respective fields, and shared them with the world. For example the “vegan” package contains a number of functions geared towards community ecology, like calculating diversity indices. You could calculate a diversity index with just the base R, but it would be more difficult and take longer.

You can install packages in at least two ways:

1. You can use the following code, with the package names in quotes (this is one of the few times where using the console is recommended, because you only need to install a package once):

   ```
   install.packages("PACKAGE NAME HERE")
   ```

2. Or you can use the packages tab in RStudio. In the lower right panel, there should be a packages tab in between “Plots” and “Help”. Once there, there is an “Install” button. When clicked a window will appear allowing you to search for packages to install.

But Installing packages does not make them automatically accessible to you. When R boots up, it only loads its base functionality by default, so you have to load any packages that you want to use for a given R session. You can do this with the following code (with the package name not in quotes):

```
library(PACKAGE NAME)
```

The code for loading packages should be saved in your r script, because it will need to be done every time you open R.

There is a family of packages that is very popular called the "tidyverse." The aim of the tidyverse is to make data manipulation and visualization streamlined and efficient. Some people are very opinionated about whether you should use the tidyverse or base R, but in my opinion, it's mostly personal preference. If you only want to dip into R and don't plan to use it much in the future, you may as well just pick up the specific functions you need to use and not worry about much else. If you'd like to continually use R for data analysis, but don't plan on getting deep into it, getting a handle on the tidyverse may be a good idea. If you want to really get into R, I would recommend learning how to do things in base R (as well as tidyverse functions).

You can install the tidyverse suite with:

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

More information on this suite of packages can be found here:

https://www.tidyverse.org/

Note that when you install the tidyverse, it instalss a large suite of packages, but when you load the tidyverse with the "library" function, it only loads a smaller subset of "core" packages by default. Thus, if you are looking to use a specific tidyverse package with a more niche purpose, you may need to load it separately with another call of the "library" function.

14.2 Subsetting

In the last section we introduced data structures. Now let's talk about what you can do with them.

14.2.1 Vectors

The individual elements of a vector can be accessed with bracket operators - [and]. You can refer to an element by its index, or its numeric place in the sequence of elements (e.g., the 1st, the 10th, etc.). It's important to not here that R starts counting at 1, while many other programming language start counting at 0 (e.g., Python). This is another thing that people are opinionated about, and if you put your mind to it, *you can be too*! Anyway, here are some examples:

```r
## let's create a vector of the first five letters of the alphabet
my_vector <- c("a","b","c","d","e")
my_vector
```

```
[1] "a" "b" "c" "d" "e"
```

```
## now let's return the 5th element
my_vector[5]
```

```
[1] "e"
```

```
## we can return multiple elements with c()
my_vector[c(2,4)]
```

```
[1] "b" "d"
```

```
## or as a series with :
my_vector[2:4]
```

```
[1] "b" "c" "d"
```

You can also use negative numbers to exclude values from what's returned:

```
## lose the last element
my_vector[-5]
```

```
[1] "a" "b" "c" "d"
```

```
## everything but the last element
my_vector[-1:-4]
```

```
[1] "e"
```

14.2.2 Lists

Subsetting vectors is fairly straightforward, but subsetting lists can be tricky. Since lists have multiple levels of organization, they use both the [] operators and the [[]] operators. Single brackets give you the list element, and double brackets give you *what the list element contains*. Let's demonstrate:

```r
## create a list
my_list <- list(c("a","b","c"), "d", "e")
my_list
```

```
[[1]]
[1] "a" "b" "c"

[[2]]
[1] "d"

[[3]]
[1] "e"
```

```r
## grab the first list element
my_list[1]
```

```
[[1]]
[1] "a" "b" "c"
```

```r
## grab what's conatined in the first list element (in this case a vector)
my_list[[1]]
```

```
[1] "a" "b" "c"
```

```r
## another example with a scalar
my_list[2]
```

```
[[1]]
[1] "d"
```

```r
my_list[[2]]
```

```
[1] "d"
```

```r
## you can also subset what you have subsetted:
my_list[[1]][1]
```

```
[1] "a"
```

```
## but if you try subsetting a list element, it won't work the same way
my_list[1][1]
```

```
[[1]]
[1] "a" "b" "c"
```

```
## this is because [] returns the list element as a list of length 1, therefore [1] gives yo
```

This distinction can be difficult to understand, but don't worry! It takes time. The best analogy I've seen is from Hadley Wickham here:

https://adv-r.hadley.nz/subsetting.html#subset-single

You can think of a list as a train, every list element is a train car, and each has its own contents. Single brackets give you the train car/s, and double brackets gives you what's inside a single train car. And even a single train car can be another train (or a list). Also note:

```
## you can grab multiple list elements with []; this give a list with two elements
my_list[1:2]
```

```
[[1]]
[1] "a" "b" "c"

[[2]]
[1] "d"
```

```
## list elements can be named and indexed by their name as well
named_list <- list(first = 1:3, second = 10)
named_list
```

```
$first
[1] 1 2 3

$second
[1] 10
```

```
named_list["first"]
```

```
$first
[1] 1 2 3
```

14.2.3 Data Frames

14.2.3.1 Base

Subsetting data frames is a little easier to get a handle on, you just need to think in two dimensions. When using single brackets to subset data frames, you need to specify the index of the row and the column separately and in that order. You separate each index number by a comma inside the brackets. Check it out:

```
## create data frame
my_data <- data.frame(letter = c("a","b","c"), # each column has a name
                      number = c(1, 2, 3),
                      vowel = c(TRUE, FALSE, FALSE))
my_data
```

```
  letter number vowel
1      a      1  TRUE
2      b      2 FALSE
3      c      3 FALSE
```

```
## grab the element in the 2nd row, 1st column
my_data[2,1]
```

```
[1] "b"
```

```
## you can also grab a whole row or column by leaving onse side of the comma blank
my_data[2,]
```

```
  letter number vowel
2      b      2 FALSE
```

```
my_data[,1]
```

```
[1] "a" "b" "c"
```

```
## (subsetting a row gives you a data frame, subsetting a column gives you a vector)
```

But data frames also have named columns! Let's use that to our advantage. You can specify a column's name instead of its index in brackets, like for a list, or you can use the $ operator.

```
## subsetting by name in brackets
my_data[,"vowel"]
```

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

```
## subsetting by name with $ (notice no quotes)
my_data$vowel
```

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

```
## the downside of $ is that you can't grab more than one column like with brackets
my_data[,c("letter", "vowel")]
```

```
  letter vowel
1      a  TRUE
2      b FALSE
3      c FALSE
```

```
## subsetting multiple columns gives you a data.frame

## you can use $ with named lists too
named_list$first
```

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

```
## you can mix subsetting operators if you ever need to
my_data$vowel[1]
```

```
[1] TRUE
```

```
my_data[1,]$vowel
```

```
[1] TRUE
```

You can also use brackets to select rows by value, not index. You just need to use some comparison operator in a statement that resolves as TRUE or FALSE.

```
## grab the consonant rows
my_data[my_data$vowel == FALSE,]
```

```
  letter number vowel
2      b      2 FALSE
3      c      3 FALSE
```

```
## grab the rows before the third
my_data[my_data$number < 3,]
```

```
  letter number vowel
1      a      1  TRUE
2      b      2 FALSE
```

```
## you can combine criteria
my_data[my_data$number < 3 & my_data$vowel == FALSE,]
```

```
  letter number vowel
2      b      2 FALSE
```

14.2.3.2 tidyverse

Now the reason that we talked about packages in between data structures and subsetting is because the tidyverse (specifically, the dplyr package) has more functions for subsetting: filter and select. Filter works much like grabbing rows by value, and select works like grabbing columns by name. Let's look at some examples:

```
## load the tidyverse
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

```r
## filter for consonants
filter(.data = my_data, vowel == FALSE)
```

```
  letter number vowel
1      b      2 FALSE
2      c      3 FALSE
```

```r
## select letter related columns
select(.data = my_data, letter, vowel)
```

```
  letter vowel
1      a  TRUE
2      b FALSE
3      c FALSE
```

```r
## you can also exclude columns
select(.data = my_data, !number)
```

```
  letter vowel
1      a  TRUE
2      b FALSE
3      c FALSE
```

```r
## note: selecting a single column will give a data frame, not a vector
select(.data = my_data, number)
```

```
  number
1      1
2      2
3      3
```

```r
## another tidyverse/dplyr function, pull, will give just a vector
pull(.data = my_data, number)
```

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

As you can see, filter, select and pull are versatile, consistent and powerful. However, they lack one key ability: assignment. You can use brackets and $s to assign things (which are base R operators):

```
## assign a new value to a data element (NA means missing value)
my_data[3,2] <- NA
my_data
```

```
  letter number vowel
1      a      1  TRUE
2      b      2 FALSE
3      c     NA FALSE
```

```
## create a whole new column with $ (vector must be of same length as the number of rows)
my_data$new_column <- c("some", "new", "data")
my_data
```

```
  letter number vowel new_column
1      a      1  TRUE       some
2      b      2 FALSE        new
3      c     NA FALSE       data
```

15 Importing Data

The previous chapters focused on R's functionality, but you might not be feeling any closer to working with *your* data in R. We'll transition to that now! The first step is getting your data into your R environment, so that you can use it with R's functions.

15.1 Reading Data

Importing data into R is often referred to as reading data, as that is what the computer is doing, it's reading the contents of a file (usually a text file). Most ecologists and data scientists work with a text file called a Comma Separated Value file, or csv. This is a small file that's easy for computers to read where each column is separated by a column, and each row by a new line. You can save excel files as csv from the "Save As…" menu, and you can specify csv as the type when downloading a Google sheet.

15.1.1 From Your Computer

So when you have files you want to read locally on your computer, the first thing you need to think about is what's called the "working directory". The working directory is the folder on your computer where R will look for files when prompted, and also where it will save output.

You can check your current working directory:

```
## return current working directory
getwd()
```

```
[1] "/Users/kit/Documents/UMN/Research/cedar_creek_projects/ccesr_intern_hub"
```

You can also set your working directory manually

```
## change working directory
setwd("some/different/folder")
```

Or, in RStudio, you can click the Session dropdown menu at the top of the window, then “Set Working Directory”, then “Choose Directory.”

If you use an R Project (**highly recommended**), you don’t have to worry as much about this. If you have a project open, the working directory will be automatically set to the folder that contains the .Rproj file that is created when you create a project. See Section 11.2 for more info!

When you’re in an R Project, or have a csv you want in your working directory, you can read it into your environment like so:

```
## read data
my_data <- read.csv("the name of your file in quotes", header = TRUE)
```

The read.csv function creates a data frame from the csv you specify, and then the <- assigns it to “my_data.” The “header = TRUE” argument tells R to interpret the first line of the csv as the column names.

15.1.2 tidyverse Function

The readr package in the tidyverse family also has its own data reading functions.

```
## load tidyverse
library(tidyverse)

## read data (assumes header by default)
my_data <- read_csv("name of your data in quotes")
```

These functions are pretty similar, with one exception: read.csv gives you a data frame, but read_csv gives you a “tibble.” What is a tibble? It’s another special type of list, much like a data frame, but with a few differences. It was designed to work more consistently with tidyverse functions. One important difference between data frames and tibbles that when you subset an individual column with the brackets ([]), data frames will give you vectors, and tibble will give single column tibbles. This has caused me confusion when writing functions, but you may not run into it.

15.1.3 From The Web

You can also read files directly from the web. If you have your data in Google sheets, you can create a URL for R to import it directly. Simply go to the File menu, click “Share” and then “Publish to web”. In the box that pops up, you will need to select the file type as “.csv”, not web page. Then save the URL that it gives you!

For demonstration, I've created a few data sheets that you too can import into R by copying the following code:

```
## put the url of the data in quotes
fake_mammals <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vQ9mfx88nM33PC6WpI

fake_insects <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vTOsnHMdsxzzzkxt_J
```

These two files will be used throughout the next chapters. The first is some made-up data of some mammal captures at 6 sites across forest and savanna habitats (with mass and parasite info), and the second is made up sweep-netting data from the same sites.

15.1.4 Other File Types

Now there may be times when you want or need to import data that aren't .csv files. You may come across .tsv files, which are "tab-separated values" files. These are also in a text based format, but use tabs instead of commas to delimit columns. In Base R, the "read.delim" function will read .tsv files by default. The tidyverse readr package has "read_tsv" for this purpose. You can also use "read.delim" or readr's "read_delim" to read a text file with any sort of delimiter between columns, like dashes, underscores, dots, etc. The "read.delim" function can do this by setting the "sep" argument, and the "read_delim"function can do it by setting the "delim" argument.

You may also want to import Microsoft Excel files (.xls or .xlsx). Base R doesn't have any functionality for this, but the tidyverse's readxl package does: the "read_excel" function. Do note however that you may need to specify the sheet you want! Also, the readxl package is not in the "core" tidyverse, so you will have to specifically load it in R.

More info on the tidyverse's readr and readxl can be found here:

https://readr.tidyverse.org/

https://readxl.tidyverse.org/

You can also import Google Sheets with the googlesheets4 package, but that is is a bit trickier given that you'll need to specify your Google credentials and work with Google Drive folder hierarchy and file IDs. The googlesheets4 package, as well as the related googledrive package, are both in the tidyverse family, but not in the "core", so need to be specifically loaded. More info here:

https://googlesheets4.tidyverse.org/

https://googledrive.tidyverse.org/

15.2 Checking Data

Now that you have data, you will want to look at it!

15.2.1 The Whole Table

You can look at a whole data frame by clicking on its name in the “Environment” pane in RStudio (upper right), or with the View() function:

```
View(fake_mammals)
```

You can also just look at parts:

```
## check top 6 rows
head(fake_mammals)
```

```
  site site_type            species mass_g tick_count helminth_mass_g
1    a    forest White-footed mouse     20          0             512
2    a    forest White-footed mouse     24         10             365
3    a    forest White-footed mouse     23          2               0
4    a    forest White-footed mouse     19          0             608
5    a    forest White-footed mouse     25         12             109
6    a    forest        Deer mouse      22          3             456
```

```
## check bottom 6 rows
tail(fake_mammals)
```

```
   site site_type            species mass_g tick_count helminth_mass_mg
43    f   savanna White-footed mouse     21          0              408
44    f   savanna White-footed mouse     25          1              197
45    f   savanna White-footed mouse     24          0              152
46    f   savanna         Deer mouse     20          0              508
47    f   savanna         Deer mouse     22          2              496
48    f   savanna        Meadow vole     23         NA               56
```

You can also take a look at the structure of the data with str(), which will tell you how many rows (observations) and how many columns (variables), as well as the type of each column.

```
## check structure
str(fake_mammals)
```

```
'data.frame':	48 obs. of  6 variables:
 $ site            : chr  "a" "a" "a" "a" ...
 $ site_type       : chr  "forest" "forest" "forest" "forest" ...
 $ species         : chr  "White-footed mouse" "White-footed mouse" "White-footed mouse" "Whi
 $ mass_g          : int  20 24 23 19 25 22 22 21 23 20 ...
 $ tick_count      : int  0 10 2 0 12 3 2 0 NA NA ...
 $ helminth_mass_mg: int  512 365 0 608 109 456 521 432 20 129 ...
```

15.2.2 Individual Columns

You can also take a look at individual columns with the $ operator, and get quick summaries with summary():

```
## summarize mammal masses
summary(fake_mammals$mass_g)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  17.00   20.75   23.00   22.60   25.00   28.00 
```

```
## summarize helminth masses
summary(fake_mammals$helminth_mass_mg)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    0.0   119.5   341.5   306.5   501.2   713.0 
```

Note that for the second summary (helminth mass), it tells you how many NA's, or missing values, there are.

15.2.3 Factors: The Pseudo Data Type

Unless you specify, csv reading functions will assume the data type of each column in a data sheet (numeric, character, etc.). Any categorical variable will be considered a character type generally. But the way character types are stored in computer memory does not lend itself well to statistical analysis. To remedy this, R has a special data type called the factor for categorical data. A factor is made up of two parts - the levels, which are stored to the computer as integers, and the labels, which are character strings that we can read as the category names.

In our mammal data, we probably want mammal species, site type, and site all to be factors. We can convert them with the as.factor function!

```
## convert species to factor
fake_mammals$species <- as.factor(fake_mammals$species)
## convert site type to factor
fake_mammals$site_type <- as.factor(fake_mammals$site_type)
## convert site to a factor
fake_mammals$site <- as.factor(fake_mammals$site)

## check it out!
summary(fake_mammals$species)
```

```
        Deer mouse        Meadow vole White-footed mouse
                15                  9                 24
```

```
## also do for insects
fake_insects$site <- as.factor(fake_insects$site)
fake_insects$site_type <- as.factor(fake_insects$site_type)
```

There are also similar functions for converting data types to numeric (as.numeric()) and character (as.character()).

16 Wrangling Data

Now that we have our data in, let's play with it!

Setup:

```
## read data
fake_mammals <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vQ9mfx88nM33PC6WpI

fake_insects <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vTOsnHMdsxzzzkxt_J

## convert species to factor
fake_mammals$species <- as.factor(fake_mammals$species)
## convert site type to factor
fake_mammals$site_type <- as.factor(fake_mammals$site_type)
## convert site to a factor
fake_mammals$site <- as.factor(fake_mammals$site)

## also do for insects
fake_insects$site <- as.factor(fake_insects$site)
fake_insects$site_type <- as.factor(fake_insects$site_type)
```

16.1 Adding Columns

One simple thing you may want to do is add columns to your data, which may be calculations from existing columns.

16.1.1 Base

In base R, we have already kinda done this. You can assign something to a new column with the $ and <- operators.

For the insect data, let's say we wanted to calculate average temperature at a given site based on the recorded high (temp_hi) and low (temp_low):

```r
## calculate mean temp
fake_insects$temp_mean <- (fake_insects$temp_hi + fake_insects$temp_lo)/2

## check it out
fake_insects$temp_mean
```

```
[1] 21.25 23.25 19.25 26.75 24.00 28.00
```

16.1.2 tidyverse

In the tidyverse, adding new columns is done with the mutate function:

```r
## load tidyverse
library(tidyverse)

## mutate a new column
fake_insects <- mutate(.data = fake_insects, ## specify data
                       temp_mean_mutated = (temp_hi + temp_lo)/2) ## calculate new column

## this column should be the same for all six rows (a TRUE should be returned for each)
fake_insects$temp_mean == fake_insects$temp_mean_mutated
```

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

As you can see, using mutate mean you have to write the name of data frame fewer times.

16.2 Pivoting / Reshaping

You also may need to transform your data between the wide and long formats (see 2.1 Data Formats). I find that the pivot functions from tidyr in the tidyverse are easier to use, so we will go over those. But by all means, if you prefer base, go for it! I'm just less familiar with using base R for this.

16.2.1 Wide to Long

The insect data is partly in wide format: we have a column for each order of insect, where the count is implicitly the values in the cells. It will be easier to work with if we make one "count" column and one "order" column. We can do this with the pivot_longer function:

```
## lengthen the order count data
long_insects <- pivot_longer(data = fake_insects,
                             cols = c(hymenoptera, lepidoptera, coleoptera, diptera,
                                      odonata, hemiptera, orthoptera, ephemeroptera,
                                      tricoptera, plecoptera),
                             names_to = "order",
                             values_to = "count"
                             )

## check it out
head(long_insects)
```

```
# A tibble: 6 x 8
  site  site_type temp_hi temp_lo temp_mean temp_mean_mutated order       count
  <fct> <fct>       <dbl>   <dbl>     <dbl>             <dbl> <chr>       <int>
1 a     forest       23.5      19      21.2              21.2 hymenoptera     9
2 a     forest       23.5      19      21.2              21.2 lepidoptera     3
3 a     forest       23.5      19      21.2              21.2 coleoptera     16
4 a     forest       23.5      19      21.2              21.2 diptera        29
5 a     forest       23.5      19      21.2              21.2 odonata         4
6 a     forest       23.5      19      21.2              21.2 hemiptera      10
```

Now we have multiple rows for each site, one for each order! You may not believe me, but this will make things easier down the line.

For reference, the cols argument specifies which columns you want to pivot. The names_to argument names the column that will contain the pivoted column names, and the values_to argument names the column in which the cell values will be placed.

16.2.2 Long to Wide

You'll occasionally want to turn long to wide as well. pivot_wider works for this

```
## widen our long data
wide_insects <- pivot_wider(data = long_insects,
                            names_from = "order",
                            values_from = "count")
```

This is the inverse of what we just did - we made a column for each value in the “order” column given to the names_from argument, the values of which are pulled from the “count” column given to the values_from argument.

16.2.3 Other Data Reformatting Functions

There are other functions that can be used to reformat data between long and wide, but some are outdated. You may come across some of them in blogs of code examples.

The base R function for transforming data between wide and long format is “reshape”, from the “stats” package. It’s just one function that can transform data in both directions. More info here:

https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/reshape

There are also previous versions of the “pivot” functions, such as

- “cast” and “melt”, from the “reshape” package
- “dcast”, “acast”, and “melt”, from the “reshape2” package
- “gather” and “spread”, from older versions of the “tidyr” package

All of the above still work, but have been superseded by the pivot functions.

Personally, I prefer to use the pivot functions because they are named as exactly what they do and they work nicely with other tidyverse functions. The reshape function from base R works fine too, but may have some slight differences. I recommend avoiding using the older cast, melt, gather, or spread functions.

16.3 String Manipulation

String manipulation may sound like playing cat’s cradle or fingerstyle guitar, but in the context of R programming, it refers to working with text data. As you may recall, values with the character data type in R can also be called “strings”. Essentially, strings ar strings of characters (alphanumeric, symbols, etc.).

In ecological and environmental data, we often encounter strings as names of sites or plots, as labels for categorical data (which we transform into factors), or as notes on observations.

R has many functions for working with strings - combining them, splitting them, searching through them, transforming them, padding them with leading zeroes, etc. We will focus on a few simple ones that might be useful for typical ecological data workflows. Base R features several valuable functions that can do these tasks. In addition, the "stringr" package in the tidyverse features a suite of consistently formatted functions that do similar things (stringr is part of the core tidyverse, so is loaded with the "library(tidyverse)" command). If you are getting deep into pattern recognition, string transformation, etc., the stringr package is probably your best bet, but for the simple things we're about to go over, base and tidverse functions both work fine.

Note: From a broad perspective, R is not the language of choice when it comes to text data, as it was designed primarily for statistics. In the realms of bioinformatics (genomes are really long strings!) and language processing (the written word is made of strings!), Python is often used. But for most ecological purposes, R does just fine.

16.3.1 Combining Strings

Often times you may want to combine several columns of your data into an "ID" type variable to describe observations from a certain plot in a certain site on a particular date. The base R "paste" function and then less well-named "str_c" function from the stringr package can do this easily.

```r
## first let's make some fake data to work with
## this will include sites , plots, and years of observation
fake_observations <- data.frame(site = c(1, 1, 2, 2, 1, 1, 2, 2),
                                plot = c("a", "b", "a", "b",
                                         "a", "b", "a", "b"),
                                year = c(2023, 2023, 2023, 2023,
                                         2024, 2024, 2024, 2024))

## now let's add an ID column by combining these variables

## first with paste
## we specify we want each variable to be separated by an underscore
## with the "sep" argument
fake_observations$id <- paste(fake_observations$site,
                              fake_observations$plot,
                              fake_observations$year,
                              sep = "_")

## now look at the result
fake_observations$id
```

```
[1] "1_a_2023" "1_b_2023" "2_a_2023" "2_b_2023" "1_a_2024" "1_b_2024" "2_a_2024"
[8] "2_b_2024"
```

```
## now let's try with str_c
## this time we use a hyphen in the "sep" argument
fake_observations <- mutate(fake_observations,
                            id2 = str_c(site, plot, year, sep = "-"))

## and we can look at this as well (using pull from tidyverse this time)
pull(fake_observations, id2)
```

```
[1] "1-a-2023" "1-b-2023" "2-a-2023" "2-b-2023" "1-a-2024" "1-b-2024" "2-a-2024"
[8] "2-b-2024"
```

Note that for both functions we first present the variables we want to combine, and then specify how we want each to be separated in the resulting string. Also note that “site” and “year” are numeric data in the original data frame, but can be combined into strings all the same.

The paste function and str_c function behave similarly in most simple cases, but have slightly different rules when it comes to dealing with missing values (NAs) and when input vectors have different lengths.

16.3.2 Converting String Case

Sometimes when entering ecological data, character strings might end up with inconsistent capitalization. As a result, when converting a categorical variable into a factor, you can end up with multiple levels for the same category. For example, if you have four plots at a site- “north”, “east”, “south”, and “west”, and some entries of the “north” plot are written as “North”, the resulting factor would have both a “north” level and a “North” level. Converting your strings all to one case can avoid this.

You can convert to upper case with:

- “toupper”, the base R function
- “str_to_upper”, the stringr function

And you can convert to lower case with:

- “tolower”, the base R function
- “str_to_lower”, the stringr function

Here is an example:

```r
## create a vector of character strings to work with
plots <- c("north", "North", "east", "east",
           "South", "South", "West", "west")

## convert to factor
plot_factor <- as.factor(plots)

## check out the factor
summary(plot_factor)
```

```
 east north North South  west  West
    2     1     1     2     1     1
```

```r
## oh no, there are six levels instead of four!

## let's fix it with "tolower"
plots_lower <- tolower(plots)

## and factorize...
lower_plot_factor <- as.factor(plots_lower)

summary(lower_plot_factor)
```

```
 east north south  west
    2     2     2     2
```

```r
## much better!
```

For more information see the reference site for the stringr package:

https://stringr.tidyverse.org/

And this article comparing stringr and base R string functions:

https://stringr.tidyverse.org/articles/from-base.html#mutate-strings

17 Summarizing Data

Now let’s describe our data!

Setup:

```
## load tidyverse
library(tidyverse)

## read data
fake_mammals <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vQ9mfx88nM33PC6WpI

fake_insects <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vT0snHMdsxzzzkxt_J

## convert species to factor
fake_mammals$species <- as.factor(fake_mammals$species)
## convert site type to factor
fake_mammals$site_type <- as.factor(fake_mammals$site_type)
## convert site to a factor
fake_mammals$site <- as.factor(fake_mammals$site)

## also do for insects
fake_insects$site <- as.factor(fake_insects$site)
fake_insects$site_type <- as.factor(fake_insects$site_type)

## lengthen the order count data
long_insects <- pivot_longer(data = fake_insects,
                             cols = c(hymenoptera, lepidoptera, coleoptera, diptera,
                                      odonata, hemiptera, orthoptera, ephemeroptera,
                                      tricoptera, plecoptera),
                             names_to = "order",
                             values_to = "count"
                             )
```

17.1 Describing With Summaries

You'll often want to give simple, illustrative information about the data you collected. The tidyverse is great for this! (base R has "aggregate", which also works, but we won't go into it here).

17.1.1 tidyverse

The package dplyr in the tidyverse has two wonderful functions: group_by() and summarize(). You can also use the British spelling, "summarise()", but I use a z because *it's what the founders would have wanted.*

Before we use these however, we need to introduce a very useful operator, the pipe: %>%. This operator directs data into the first argument of a function, which allows you to chain functions together efficiently. Let's try an example with the filter and select subsetting functions (see 14.2.3 Next Steps; Subsetting; Data Frames):

```
## grab only the forest sites from the insect data
forest_sites <- fake_insects %>% ## take fake_insects and pipe it into filter...
  filter(site_type == "forest") %>% ## filter only forest rows, pipe into select
  select(site) ## select only the site column
## the whole pipe chain is assigned to "forest_sites"

forest_sites
```

```
  site
1    a
2    b
3    c
```

```
## this is the same as
forest_rows <- filter(fake_insects, site_type == "forest")
forest_sites <- select(forest_rows, site)
```

Note: base R also has a pipe operator, |>. It's newer and mostly the same as %>%, so I just haven't transitioned.

Now, let's try with group_by() and summarize()! Let's say you wanted the total number of insects caught at each site (be sure to have pivoted your insect data as described in the setup above!):

```
## summarize total insect catch
insect_counts <- long_insects %>%
  group_by(site) %>% ## group observations
  summarize(total_insects = sum(count)) ## sum all insects

insect_counts
```

```
# A tibble: 6 x 2
  site  total_insects
  <fct>         <int>
1 a                84
2 b                87
3 c               136
4 d                60
5 e                61
6 f                51
```

As you can see, the summarize function works a bit like the mutate function, in that you create a new column.

Note that group_by doesn't visibly change your data, but it changes some attributes that the computer can see when it runs the summarize function. If you forgot which sites are in which type of habitat, you could also include that variable in the group_by arguments (since it doesn't subdivide the sites, it won't change the calculation).

```
insect_counts <- long_insects %>%
  group_by(site, site_type) %>%
  summarize(total_insects = sum(count))
```

You can also calculate means and variances! You can use the mean, var, and sd functions. Let's try for each order across all sites:

```
order_summary <- long_insects %>%
  group_by(order) %>%
  summarize(count_mean = mean(count), ## you can do multiple sumaries at once
            count_var = var(count),
            count_sd = sd(count))

head(order_summary)
```

```
# A tibble: 6 x 4
  order          count_mean count_var count_sd
  <chr>               <dbl>     <dbl>    <dbl>
1 coleoptera          15.8      43.4      6.59
2 diptera             17       163.      12.8
3 ephemeroptera        5.33    171.      13.1
4 hemiptera            6.17     42.6      6.52
5 hymenoptera          8.5       9.1      3.02
6 lepidoptera          5.67      5.87     2.42
```

You could also do this separately by site type:

```
orders_by_habitat <- long_insects %>%
  group_by(site_type, order) %>%
  summarize(count_mean = mean(count), ## you can do multiple sumaries at once
            count_var = var(count),
            count_sd = sd(count))
```

```
`summarise()` has grouped output by 'site_type'. You can override using the
`.groups` argument.
```

```
head(orders_by_habitat)
```

```
# A tibble: 6 x 5
# Groups:   site_type [1]
  site_type order          count_mean count_var count_sd
  <fct>     <chr>               <dbl>     <dbl>    <dbl>
1 forest    coleoptera          20.7      17.3      4.16
2 forest    diptera             28.3       9.33     3.06
3 forest    ephemeroptera       10.7     341.      18.5
4 forest    hemiptera           11.7      14.3      3.79
5 forest    hymenoptera         10.3       2.33     1.53
6 forest    lepidoptera          4.33      2.33     1.53
```

17.1.2 Saving Summaries

Finally, you can save summarized output with base R's write.csv() or write_csv() from the tidyverse:

```
## save summary to your working/project directory
## first argument is data, second argument is filename
write.csv(orders_by_habitat, "order_summary")
```

When saving output, it can be often helpful to add a timestamp to the saved file name, so you can easily identify when you created it and sort among versions. This is another time when we can use the string combining functions we learned about earlier! (see 16.3.1 String Manipulation, Combining Strings)

```
## first let's save a timestamp string with "Sys.time"
## I like to format it with the "format" function
## this code results in "YYYYMMDD_HHMMSS" format
## (don't worry about the specifics, but feel free to use this code)
timestamp <- format(Sys.time(), format = "%Y%m%d_%H%M%S")

## then we can use it in our data writing step:
write.csv(orders_by_habitat, paste("order_summary", timestamp, sep = "_"))
```

17.1.3 Making Pretty Tables

The “gt” package is good for this (quick tutorial under construction....)

https://gt.rstudio.com/

17.2 Community Ecology

Averages and variances are all well and good but what about ecological measures? Let’s try calculating the measures discussed in 8.4 Ecological Community Descriptors!

17.2.1 Richness

You may be interested in how many insect orders are represented in each site.

Let’s do it in a pipe chain!

```
## order presence
order_richness_site <- long_insects %>%
  mutate(presence = as.numeric(count > 0)) %>% ## create binary presence column
  group_by(site, site_type) %>%
  summarize(order_richness = sum(presence))
```

```
`summarise()` has grouped output by 'site'. You can override using the
`.groups` argument.
```

```
order_richness_site
```

```
# A tibble: 6 x 3
# Groups:   site [6]
  site  site_type order_richness
  <fct> <fct>              <dbl>
1 a     forest                 8
2 b     forest                 7
3 c     forest                 9
4 d     savanna                6
5 e     savanna                7
6 f     savanna                6
```

I calculated the presence column by checking if each value is positive (> 0), which returns a logical TRUE or FALSE, and then if you convert a logical variable to a numeric variable, TRUEs become 1s and FALSEs become 0s. Nifty!

17.2.2 Diversity

The *vegan* (short for *veg*etation *an*alysis) package contains many useful functions that are widely used in analysis of community data, including *diversity( )*, which will calculate Simpson's and Shannon's diversity indices.

You need to get the *vegan* package installed to start with.

```
## load vegan
library(vegan)
```

```
Loading required package: permute
```

```
Loading required package: lattice
```

```
This is vegan 2.6-6.1
```

Now let's calculate Simpson's diversity index with the insect community data:

```
# select community ID and insect order (note: this can be any taxonomic distinction, e.g., sp
insect_community <- fake_insects[, c(1,5:14)] %>%
  column_to_rownames("site")  # use first column (sites) for row names

str(insect_community) # a data frame with observations of 10 insect orders at 6 sites
```

```
'data.frame':   6 obs. of  10 variables:
 $ hymenoptera  : int  9 10 12 9 8 3
 $ lepidoptera  : int  3 6 4 10 5 6
 $ coleoptera   : int  16 22 24 12 15 6
 $ diptera      : int  29 25 31 6 9 2
 $ odonata      : int  4 6 3 2 4 7
 $ hemiptera    : int  10 9 16 0 2 0
 $ orthoptera   : int  12 9 13 21 18 27
 $ ephemeroptera: int  0 0 32 0 0 0
 $ tricoptera   : int  1 0 0 0 0 0
 $ plecoptera   : int  0 0 1 0 0 0
```

```
simpson <- diversity(insect_community, index = "simpson") # compute Simpson's 1-D Index of Di
simpson # closer to 1 = more diverse
```

```
        a         b         c         d         e         f 
0.7947846 0.8093539 0.8293685 0.7761111 0.8013975 0.6682045 
```

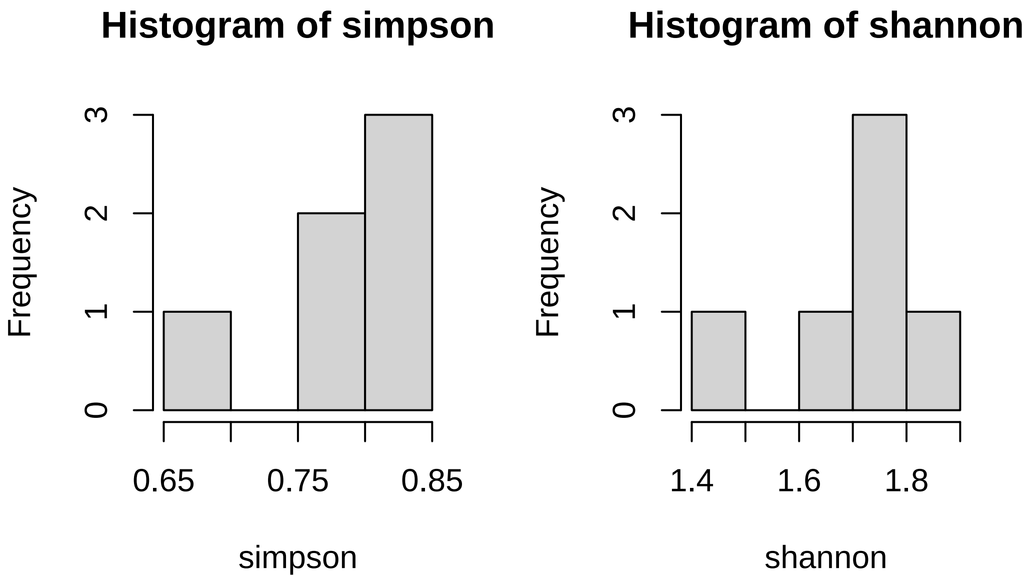
The same *diversity( )* can be used to compute the Shannon Index (and many other diversity indices) by changing the 'index' argument:

```
shannon <- diversity(insect_community) # note that Shannon is the default, therefore you don
shannon # Typically ranging from 1.5-3.4, higher = more diverse
```

```
       a        b        c        d        e        f 
1.770415 1.792899 1.897978 1.616152 1.749617 1.406488 
```

Let's compare the two diversity indices:

```
par(mfrow = c(1, 2)) # use par to generate panels with 1 row of 2 panels
hist(simpson)
hist(shannon)
```



The “hist()” function above provides a quick way to visualize the frequency of values for a given variable or index with histograms. We will learn more about data visualization a little later!

17.2.3 (Dis)similarity

Dissimilarity analysis is a good way to explore variability in community composition between different sites. The *vegan* package also contains powerful functions for calculating (dis)similarity indices. For example, the *vegdist( )* function computes pairwise dissimilarity indices, which are used to quantify the compositional dissimilarity (distance) between two different sites. To use this function, your data is expected to be one where the rows are the sites/samples and the columns are the taxonomic groups (e.g. species, orders, functional groups).

Let’s calculate two most commonly used dissimilarity indices, Bray-Curtis and Jaccard indices, for the insect community data:

```
## Bray-Curtis dissimilarity index
bray <- vegdist(insect_community, method = "bray") # "bray" is the default
bray # values are bounded between 0 and 1, where 0 = same composition, 1 = maximally dissimil
```

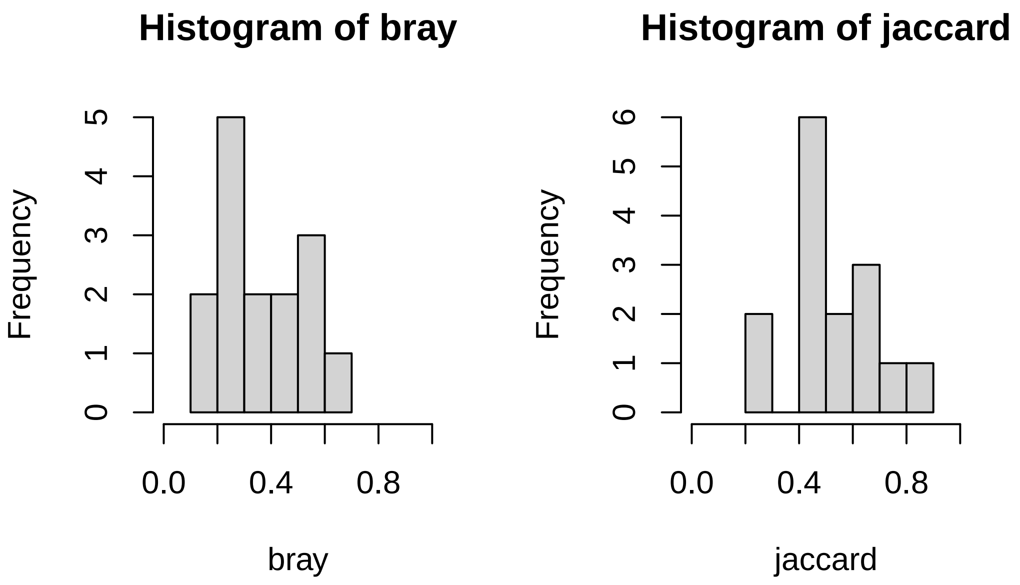
```
     a    b    c    d    e
b 0.1228070
```

```
c 0.2545455 0.2645740
d 0.3888889 0.4013605 0.5306122
e 0.2689655 0.2972973 0.4517766 0.1570248
f 0.5555556 0.5362319 0.6684492 0.2792793 0.3214286
```

```
## Jaccard dissimilarity index
jaccard <- vegdist(insect_community, method = "jaccard")
jaccard
```

```
          a         b         c         d         e
b 0.2187500
c 0.4057971 0.4184397
d 0.5600000 0.5728155 0.6933333
e 0.4239130 0.4583333 0.6223776 0.2714286
f 0.7142857 0.6981132 0.8012821 0.4366197 0.4864865
```

```
## Compare the two dissimilarity indices
par(mfrow = c(1, 2))
hist(bray, xlim = range(0.0,1.0))
hist(jaccard, xlim = range(0.0,1.0))
```



17.2.4 Potential Next Steps

The next steps would be to do some sort of cluster analysis, such as principal components analysis (PCA) or non-metric multidimensional scaling (NMDS), to see where community associations exist. However, we are only sticking to the basics for now.

Also, the ‘BiodiversityR’ package contains some useful routines and utilities for examination of community data.

17.3 Related Topic: Joining Data

Sometimes with summaries, you will want to connect them to other pieces of data. Here we have some insect counts by site, and some mammal data by site. Let’s connect them! We can use the “merge” function from base R or the “join” functions from the tidyverse.

With merge:

```
## grab only the site and total columns from insect_counts
## this prevent doubling the site_type column
merged_data <- merge(insect_counts[,c("site", "total_insects")], fake_mammals, by = "site")

## look at the new column in your data
str(merged_data)
```

```
'data.frame':   48 obs. of  7 variables:
 $ site            : Factor w/ 6 levels "a","b","c","d",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ total_insects   : int  84 84 84 84 84 84 84 84 84 84 ...
 $ site_type       : Factor w/ 2 levels "forest","savanna": 1 1 1 1 1 1 1 1 1 1 ...
 $ species         : Factor w/ 3 levels "Deer mouse","Meadow vole",..: 3 3 3 3 3 1 1 1 2 2 .
 $ mass_g          : int  20 24 23 19 25 22 22 21 23 20 ...
 $ tick_count      : int  0 10 2 0 12 3 2 0 NA NA ...
 $ helminth_mass_mg: int  512 365 0 608 109 456 521 432 20 129 ...
```

With join:

```
## there are different join functions for different contexts
## left_join keeps every row from the first data frame and adds any matching rows from the
## second. it works in most cases
## inner_join and full_join can also be useful
joined_data <- insect_counts %>%
  select(total_insects, site) %>%
```

```r
  left_join(fake_mammals, by = "site")

## look at it
str(joined_data)
```

```
tibble [48 x 7] (S3: tbl_df/tbl/data.frame)
 $ total_insects   : int [1:48] 84 84 84 84 84 84 84 84 84 84 ...
 $ site            : Factor w/ 6 levels "a","b","c","d",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ site_type       : Factor w/ 2 levels "forest","savanna": 1 1 1 1 1 1 1 1 1 1 ...
 $ species         : Factor w/ 3 levels "Deer mouse","Meadow vole",..: 3 3 3 3 3 1 1 1 2 2 .
 $ mass_g          : int [1:48] 20 24 23 19 25 22 22 21 23 20 ...
 $ tick_count      : int [1:48] 0 10 2 0 12 3 2 0 NA NA ...
 $ helminth_mass_mg: int [1:48] 512 365 0 608 109 456 521 432 20 129 ...
```

18 Analyzing Data

Now let’s draw some conclusions about our data, and maybe answer questions!

Setup:

```r
## load tidyverse
library(tidyverse)

## read data
fake_mammals <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vQ9mfx88nM33PC6WpI

fake_insects <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vT0snHMdsxzzzkxt_J

## convert species to factor
fake_mammals$species <- as.factor(fake_mammals$species)
## convert site type to factor
fake_mammals$site_type <- as.factor(fake_mammals$site_type)
## convert site to a factor
fake_mammals$site <- as.factor(fake_mammals$site)

## also do for insects
fake_insects$site <- as.factor(fake_insects$site)
fake_insects$site_type <- as.factor(fake_insects$site_type)

## lengthen the order count data
long_insects <- pivot_longer(data = fake_insects,
                             cols = c(hymenoptera, lepidoptera, coleoptera, diptera,
                                      odonata, hemiptera, orthoptera, ephemeroptera,
                                      tricoptera, plecoptera),
                             names_to = "order",
                             values_to = "count"
                             )
```

18.1 Making Comparisons

First off, let's just do some simple comparisons.

18.1.1 t-tests

Let's say we want to compare two groups, like the number of insects caught in forests and savannas. We already created a summary of this in the last chapter:

```
## sum insects by site
insect_counts <- long_insects %>%
  group_by(site, site_type) %>%
  summarize(total_insects = sum(count))
```

```
`summarise()` has grouped output by 'site'. You can override using the
`.groups` argument.
```

```
insect_counts
```

```
# A tibble: 6 x 3
# Groups:   site [6]
  site  site_type total_insects
  <fct> <fct>             <int>
1 a     forest               84
2 b     forest               87
3 c     forest              136
4 d     savanna              60
5 e     savanna              61
6 f     savanna              51
```

It seems that there may be a difference! So let's run a t-test to test for a difference in means between two groups (see 9.1.2.1 Two Predictor Categories). Most stats functions in R can use the formula operator, ~. This allows us to connect our dependent variable (insect count in this case) as a function of our independent variable (site habitat type): total_insects ~ site_type.

```
## run the t test
habitat_comparison <- t.test(formula = insect_counts$total_insects ~ insect_counts$site_type)

## check the output
habitat_comparison
```

```
	Welch Two Sample t-test

data:  insect_counts$total_insects by insect_counts$site_type
t = 2.6235, df = 2.1422, p-value = 0.1116
alternative hypothesis: true difference in means between group forest and group savanna is n
95 percent confidence interval:
 -24.30218 114.30218
sample estimates:
 mean in group forest mean in group savanna 
            102.33333              57.33333 
```

If we look at the ouput, it look like the forest mean was 102.333, and the savanna mean was 57.333, for a mean difference or effect size of 45. The p-value, or how strong the evidence for a relationship is, is 0.1116. This is higher than the traditional threshold for significance, likely because we have a very small sample size (6 total).

Note: if you check the help for the t.test function (run ?t.test), you can find arguments for paired t-tests (paired) and unequal variances among groups (var.equal).

18.1.2 ANOVA

What if we have more than two categories, and we want to see if any two categories have different means? Let us return to the mammal data and compare the mass of helminths (parasitic worms) in different mammal species. We can run an analysis of variance (see 9.1.2.2 More Than Two Predictor Categories).

```r
## run the anova
helminth_comparison <- aov(fake_mammals$helminth_mass_mg ~ fake_mammals$species)

## check the output (now with the summary function)
summary(helminth_comparison)
```

```
                     Df  Sum Sq Mean Sq F value  Pr(>F)    
fake_mammals$species  2  719721  359861   10.59 0.00017 ***
Residuals            45 1529035   33979                    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Hey, that's a small p-value (0.00017)! That means we have strong evidence that there is at least one difference among the pairs of species, either between white-footed mice and deer mice, white-footed mice and meadow voles, or deer mice and meadow voles. We can use a Tukey's test to find out more:

```
## run tukey on the anova output
helminth_tukey <- TukeyHSD(helminth_comparison)

## check it out
helminth_tukey
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = fake_mammals$helminth_mass_mg ~ fake_mammals$species)

$`fake_mammals$species`
                                    diff        lwr        upr     p adj
Meadow vole-Deer mouse         -356.2222 -544.58911 -167.85534 0.0001061
White-footed mouse-Deer mouse  -111.3333 -258.37719   35.71052 0.1698916
White-footed mouse-Meadow vole  244.8889   70.26811  419.50966 0.0039996
```

At the bottom here we can see the pairwise comparisons. The two mouse species differ in helminth mass by ~111mg, but the difference is not significant. Meadow voles have a significantly different mean helminth mass from both mouse species. So in the data I made up, voles have less helminth mass than mice.

18.2 Assessing Relationships

But what if you're not dealing with categorical comparisons? Then we can check for numerical associations.

18.2.1 Correlation

We can look for simple associations without cause and effect with correlations (see 9.1.3.1 Simple Association). Mice seem to have high helminth loads, so let's check for a correlation between their body mass and helminth mass:

```
## create a subset of only mouse data
## I use the %in% operator to specify that species should be found in a specified vector
## AKA, it could be white-footed mouse OR deer mouse
mouse_data <- filter(fake_mammals, species %in% c("White-footed mouse", "Deer mouse"))

## run correlation with two variables (no formula here)
cor.test(mouse_data$helminth_mass_mg, mouse_data$mass_g)
```

```

    Pearson's product-moment correlation

data:  mouse_data$helminth_mass_mg and mouse_data$mass_g
t = -10.941, df = 37, p-value = 3.741e-13
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.9323983 -0.7711347
sample estimates:
       cor
-0.8740017
```

Here we get an effect size of -0.874 (correlation coefficient), and a p-value of 3.741e-13, which means 3.741 x 10^{-13}, or «<0.001. This means there is a strong negative relationship observed between mouse mass and helminth mass, and we have very strong evidence for it.

18.2.2 Linear Regression

If we want to infer cause and effect we can use linear regression (see 9.1.3.2 Cause and Effect). Let's say we want to know if the number of insects at a site is predictive of mammal mass at a site. Mammal mass is a continuous numeric variable, so a basic linear regression should work. First let's join the two data frames like wed did in the last chapter:

```
## join our data
mammals_insects <- insect_counts %>%
  select(total_insects, site) %>%
  left_join(fake_mammals, by = "site")

## regress mammal mass on total insects with lm function
## this time I'm specifying the data frame with the data argument
## then I don't have to write it twice
mass_model <- lm(mass_g ~ total_insects, data = mammals_insects)
```

```
## look at the output with summary again
summary(mass_model)
```

```
Call:
lm(formula = mass_g ~ total_insects, data = mammals_insects)

Residuals:
    Min      1Q  Median      3Q     Max 
-5.6018 -1.5936  0.0221  2.0487  4.3982 

Coefficients:
              Estimate Std. Error t value Pr(>|t|)    
(Intercept)   20.15860    1.09484  18.412   <2e-16 ***
total_insects  0.02808    0.01184   2.372   0.0219 *  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.549 on 46 degrees of freedom
Multiple R-squared:  0.109,	Adjusted R-squared:  0.08959 
F-statistic: 5.625 on 1 and 46 DF,  p-value: 0.02195
```

If we look at the coefficient table, we can see that the total insect term has an estimate of 0.028, which is our effect size. For every added insect to a plot, the expected average mass of the mammal community goes up by 0.028g. Connected to that effect size is a p-value of 0.0219, which means we have strong evidence for the relationship.

18.2.3 Binomial Regression

If your response variable is binary (e.g., presence/absence), you can use a binomial regression with the glm() function. Let's test if mammal mass effects the probability of having ticks attached (tick_count).

```
## first make a presence absence variable for ticks
fake_mammals$tick_presence <- as.numeric(fake_mammals$tick_count > 0)

## now do the regression, with the binomial "family"
tick_pres_model <- glm(tick_presence ~ mass_g, data = fake_mammals, family = "binomial")

## check it
summary(tick_pres_model)
```

```
Call:
glm(formula = tick_presence ~ mass_g, family = "binomial", data = fake_mammals)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -22.3584     6.7603  -3.307 0.000942 ***
mass_g        0.9857     0.2950   3.341 0.000835 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 54.040  on 38  degrees of freedom
Residual deviance: 28.181  on 37  degrees of freedom
  (9 observations deleted due to missingness)
AIC: 32.181

Number of Fisher Scoring iterations: 5
```

If we look at this like we looked at the linear regression, the mass_g term has a very small p-value meaning strong evidence for a relationship. It also has an effect size of 0.9857, meaning that the chance of having a tick increases with body mass. However, the units are in log odds, which are hard to interpret. The reason for this is some stats theory that is beyond the scope of this book/site.

18.2.4 Poisson / Negative Binomial Regression

Sometimes your response variable will be neither continuous like mass, nor binary like presence/absence. Oftentimes in ecology you have counts: discrete, numeric data AKA integers. In this case, you can use the glm() function to conduct a Poisson regression, or the glm.nb() function (from the MASS package) to conduct a Negative Binomial Regression. Both regressions deal with count data, but the Poisson regression assumes the the mean count is equal to the variance of counts, while the negative binomial regression estimates another parameter to account for difference between the mean count and the variance of counts. Negative binomial regression is good for when the variance of counts is much higher than the mean count, which occurs when there are many small counts and only a few high counts.

We can try these methods on our mammal tick data, testing if larger mammals have more ticks. First, let's try a Poisson regression:

```
## we use the glm function, but with the "poisson" family
tick_pois <- glm(tick_count ~ mass_g, data = fake_mammals, family = "poisson")

## check output
summary(tick_pois)
```

```

Call:
glm(formula = tick_count ~ mass_g, family = "poisson", data = fake_mammals)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) -9.95948    1.17732  -8.459   <2e-16 ***
mass_g       0.45938    0.04656   9.866   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 212.537  on 38  degrees of freedom
Residual deviance:  89.572  on 37  degrees of freedom
  (9 observations deleted due to missingness)
AIC: 160.98

Number of Fisher Scoring iterations: 5
```

The coefficient table shows a positive estimate, 0.45938, for the effect size of mass on tick count. This means that for each one unit increase in mammal mass, the expected average tick count goes up by about half a tick. And the associated p-value is very low, suggesting very strong evidence for this relationship.

However, tick counts on mammals happen to be a variable that often follows a negative binomial distribution (i.e., the variance exceeds the mean). We can see this:

```
## check the mean tick count
## setting na.rm to true ignores the missing tick_count values for the voles
mean(fake_mammals$tick_count, na.rm = TRUE)
```

```
[1] 2.974359
```

```r
## check the variance
var(fake_mammals$tick_count, na.rm = TRUE)
```

```
[1] 17.92038
```

As you can see the variance is ~17.9, while the mean is only ~2.97. Again, a Poisson regression would assume these to be equal, and in this case that assumption would appear to be unmet. There are statistical tests to assess whether the variance and mean are significantly different (see the "dispersiontest" function), but most of the time a researcher's background knowledge will guide the choice between Poisson and negative binomial regression. As alluded to above, there is actually a great number of research papers that show tick counts on hosts to be overdispersed (i.e. variance » mean), so we have good reason to use a negative binomial regression instead based on the scientific literature. We can try it below:

```r
## now we are using the glm.nb function from the MASS package
## (which comes with base R, but isn't loaded by default)
tick_nb <- MASS::glm.nb(tick_count ~ mass_g, data = fake_mammals)

## check output
summary(tick_nb)
```

```

Call:
MASS::glm.nb(formula = tick_count ~ mass_g, data = fake_mammals,
    init.theta = 1.734170089, link = log)

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -13.56153    2.26978  -5.975 2.30e-09 ***
mass_g        0.60711    0.09297   6.530 6.58e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for Negative Binomial(1.7342) family taken to be 1)

    Null deviance: 92.933  on 38  degrees of freedom
Residual deviance: 36.362  on 37  degrees of freedom
  (9 observations deleted due to missingness)
AIC: 137.53

Number of Fisher Scoring iterations: 1
```

```
              Theta:  1.734
          Std. Err.:  0.824

 2 x log-likelihood:  -131.529
```

Here we see a qualitatively similar result - mammal mass has a significantly positive effect on tick count, as the p-value is very low, and the effect size estimate is ~0.607. The exact p-value and estimate are slightly different to the Poisson regression output, of course. We also get an estimate of something called "Theta" at 1.734, which is the parameter that describes the difference between the mean and variance of the response variable (tick count).

Given that previous scientific research points towards a negative binomial regression being more appropriate in this case, our glm.nb results should be more sound.

Note: in the R code above, we use the "MASS::" prefix before the glm.nb function. This is a way to use a function from a package without loading the entire package (the general form being "package::function()"). The MASS package features some functions with identical names to functions in the tidyverse (e.g., select), so we avoid confusing R by not loading MASS.

18.3 Multivariate Analysis

You can of course use multiple explanatory variables in your analyses. For example, when we regressed mammal mass on insect count, we ignored mammal species. We could include it like so:

```r
## multiple regression
multi_mod <- lm(mass_g ~ total_insects + species, data = mammals_insects)

summary(multi_mod)
```

```

Call:
lm(formula = mass_g ~ total_insects + species, data = mammals_insects)

Residuals:
    Min      1Q  Median      3Q     Max 
-5.1418 -1.1856  0.1335  1.8338  5.1527 

Coefficients:
                          Estimate Std. Error t value Pr(>|t|)    
```

```
(Intercept)               19.10561    1.36855  13.960  < 2e-16 ***
total_insects              0.03490    0.01272   2.744  0.00875 **
speciesMeadow vole        -0.29446    1.08017  -0.273  0.78644
speciesWhite-footed mouse  1.02943    0.86932   1.184  0.24270
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.545 on 44 degrees of freedom
Multiple R-squared:  0.1505,    Adjusted R-squared:  0.09254
F-statistic: 2.598 on 3 and 44 DF,  p-value: 0.06421
```

Now we have multiple terms, and since species is categorical, the effect sizes and p-values are based on comparisons to a reference level (deer mouse in this case because it is first alphabetically).

We can look at the overall significance of species by running an ANOVA with aov, and summarizing the output:

```r
## we can use the model object in our aov function to save time, it will take the formula
summary(aov(multi_mod))
```

```
              Df Sum Sq Mean Sq F value Pr(>F)
total_insects  1  36.55   36.55   5.643 0.0219 *
species        2  13.93    6.96   1.075 0.3501
Residuals     44 285.00    6.48
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Looks like there is not much evidence of an effect of species.

You may be confused a bit by this code, but essentially t.tests, ANOVAs, and regressions are all “linear models”, and we are specifying them and looking at them differently with different R functions. To learn more, I recommend taking stats classes!

Finally, you may be wondering which variable to include in your analyses. Model selection is another thing you would learn in stats, but the tl;dr could be: What is your question? Use those variables.

19 Visualizing Data

Now for what most consider the fun part, visualizing patterns in your data!

Setup:

```
## load tidyverse
library(tidyverse)

## read data
fake_mammals <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vQ9mfx88nM33PC6WpI

fake_insects <- read.csv("https://docs.google.com/spreadsheets/d/e/2PACX-1vT0snHMdsxzzzkxt_J

## convert species to factor
fake_mammals$species <- as.factor(fake_mammals$species)
## convert site type to factor
fake_mammals$site_type <- as.factor(fake_mammals$site_type)
## convert site to a factor
fake_mammals$site <- as.factor(fake_mammals$site)

## also do for insects
fake_insects$site <- as.factor(fake_insects$site)
fake_insects$site_type <- as.factor(fake_insects$site_type)

## lengthen the order count data
long_insects <- pivot_longer(data = fake_insects,
                             cols = c(hymenoptera, lepidoptera, coleoptera, diptera,
                                      odonata, hemiptera, orthoptera, ephemeroptera,
                                      tricoptera, plecoptera),
                             names_to = "order",
                             values_to = "count"
                             )

## sum insects by site
insect_counts <- long_insects %>%
  group_by(site, site_type) %>%
  summarize(total_insects = sum(count))

## join data
mammals_insects <- insect_counts %>%
  select(total_insects, site) %>%
  left_join(fake_mammals, by = "site")
```

19.1 General Notes on Data Visualization

There are a few things to keep in mind in general when creating figures, even outside of R:

Usually, **figures should stand alone**. This means that your figure can speak for itself, even without a caption. This means that axes and legends are clearly labelled, and trends are emphasized. It can also be helpful to annotate statistical output onto plots themselves.

When you can, **show your actual data**, instead of summary stats. Generally, when it's not too messy, seeing all the data points is more informative to the audience. For example, you could plot a comparison of means with a point for each mean, but you could show more if you plot every point behind those means.

Finally, **remember accessibility**. Make color schemes appropriate for color-blindness, and make text large.

Note: for simpler code demonstration purposes, the figures that follow will not always necessarily meet these criteria.

19.2 The tidyverse's ggplot2

When it comes to visualizing things in R, there are many methods. You can use the base R functions for plotting (plot, hist, lines, etc.), but I'm not super adept with them. Instead I'll be walking you through using ggplot2, a package in the tidyverse family that is incredibly popular for data visualization. There is a special syntax that may take some getting used to though.

Essentially, you create a ggplot "object" (which is another special type of list with unique attributes), and then you pipe it through a series of ggplot functions to add components, themes, labels, etc. However, ggplot2 is older than the %>% pipe we have used, so it uses an old and deprecated pipe operator: +. R automatically knows to interpret + differently with ggplot objects and functions.

Here is an example of code creating a ggplot figure:

```r
## first create the ggplot object
## you need to specify your data in the data argument
## then there is a special set of arguments called aesthetic arguments
## (bound by the aes() sub-function)
## these specify what variables will inform aesthetics of your figure
## (e.g., axes, color, fills, sizes, etc.)
ggplot(data = your_data, aes(x = variable1, y = variable2, color = variable3)) +
  geom_point(size = 2) + ## then you add geometry, this "geom" is for a scatterplot
  labs(x = "Variable 1") + ## then you can add other things like labels
  scale_color_manual(values = c("red", "blue")) + ## or specify scales
  theme(axis.text = element_text(size = 12)) ## finally you can modify parts of the theme, l
```

It may seem complicated at first, but if you start small and work yourself up, you'll be chaining together code to draw beautiful figures in no time!

19.3 Figure Types

Now we'll go over how to make some common figure types, based on your analyses.

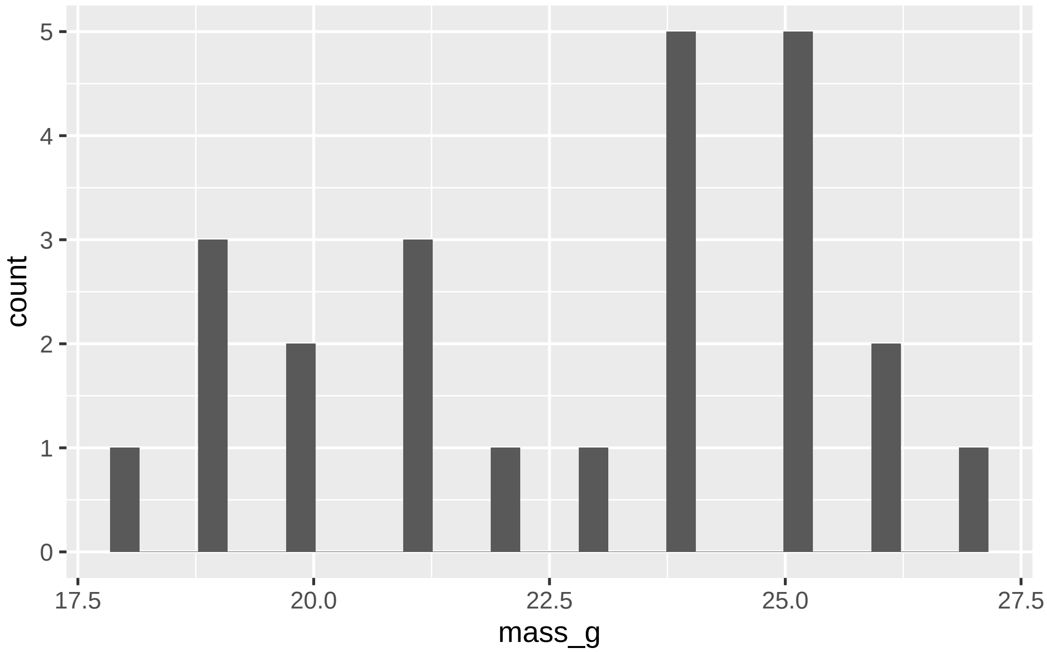
19.3.1 One Variable: Continuous

If you want to show the distribution of a single variable, you could use a histogram or a density plot.

For demonstration, let's make a plots of white-footed mouse masses.

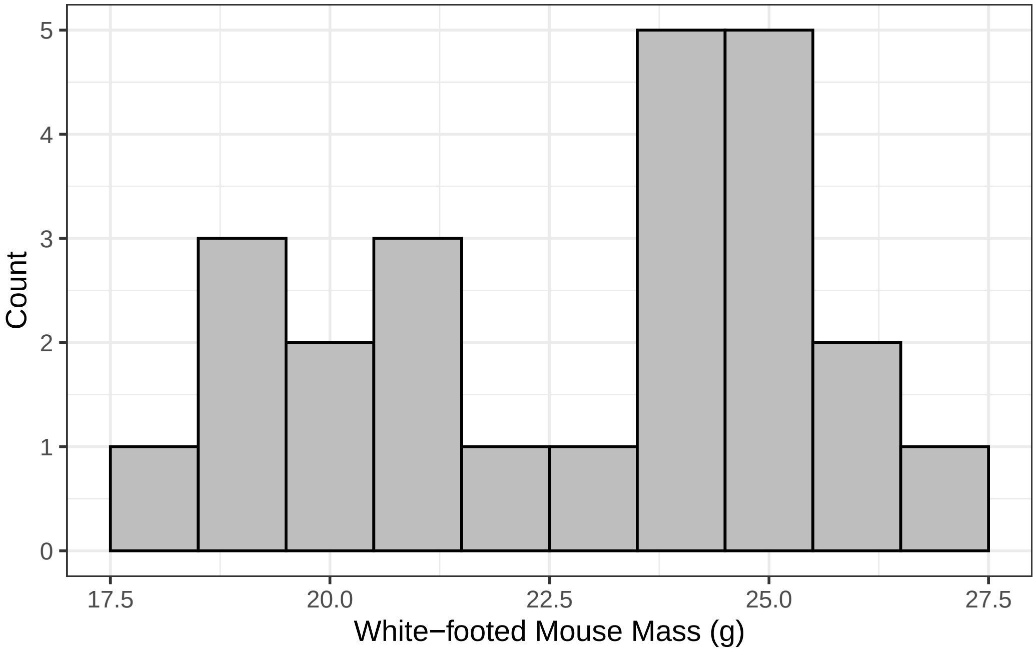
```r
## create a data frame of only white-footed mice
wf_mice <- filter(fake_mammals, species == "White-footed mouse")

## make a ggplot, use wf_mice data, and specify mass as the x variable
ggplot(data = wf_mice, aes(x = mass_g)) +
  geom_histogram()
```

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



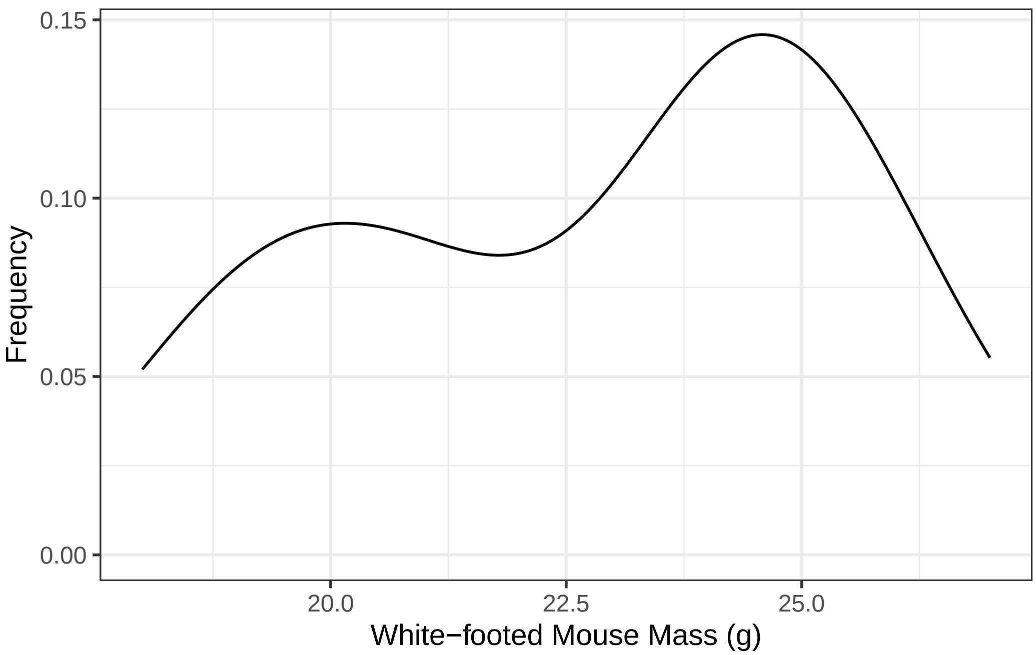
There, a simple histogram. Now let's play with how it looks:

```
## make a ggplot, use wf_mice data, and specify mass as the x variable
ggplot(data = wf_mice, aes(x = mass_g)) +
  ## give a wider binwidth to the histogram, and make it grey bars with black outlines
  geom_histogram(binwidth = 1, fill = "grey", color = "black") +
  labs(x = "White-footed Mouse Mass (g)", y = "Count") + ## nicer labels
  theme_bw() ## my favorite simple theme
```



Cool! We could also look at this as a density plot! This will give more of a smooth line

```
## make a ggplot, use wf_mice data, and specify mass as the x variable
ggplot(data = wf_mice, aes(x = mass_g)) +
  geom_density() + ## create density plot
  labs(x = "White-footed Mouse Mass (g)", y = "Frequency") + ## nicer labels
  theme_bw() ## my favorite simple theme
```

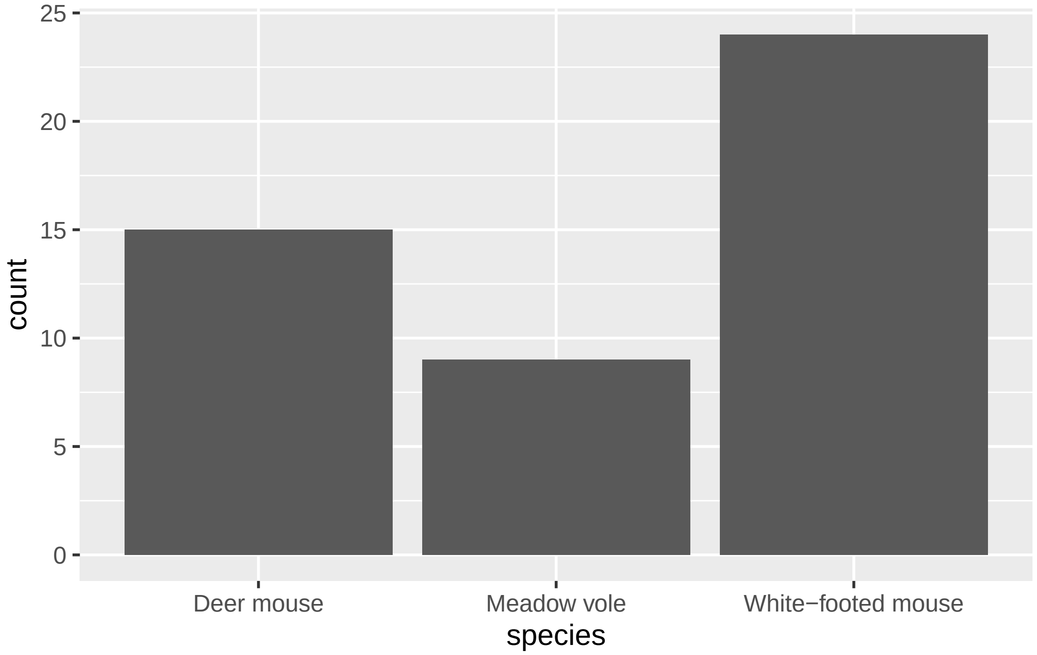


19.3.2 One Variable: Categorical

If you want to show how many observations are in each category, you can use a bar plot.

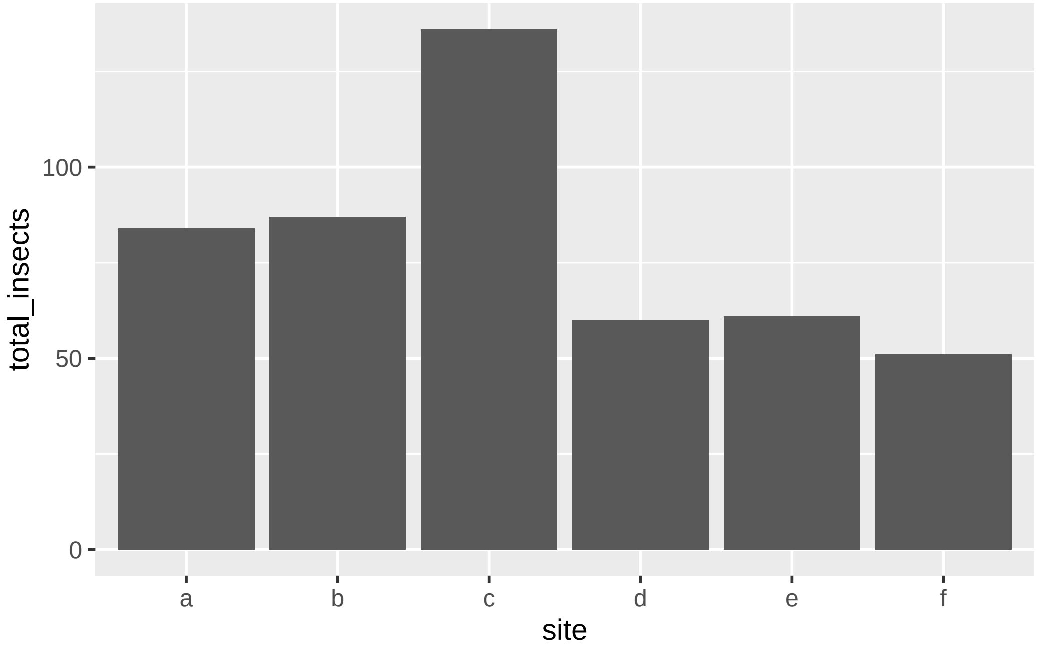
In this demo, let’s make a bar plot of how many of each mammal species were caught.

```
## specify x as species
ggplot(data = fake_mammals, aes(x = species)) +
  geom_bar() ## make bar plot
```



The geom_bar function will count up all the observations of each species level to inform its bars. Thus, it is assuming you are giving it long data. Another closely related function is geom_col, which just makes a bar as tall as a number value in the data. For example, let's make a bar plot of how many insect were caught at each site.

```
## need to specify two variables this time, one for the category, one for the count value
ggplot(data = insect_counts, aes(x = site, y = total_insects)) +
  geom_col()
```



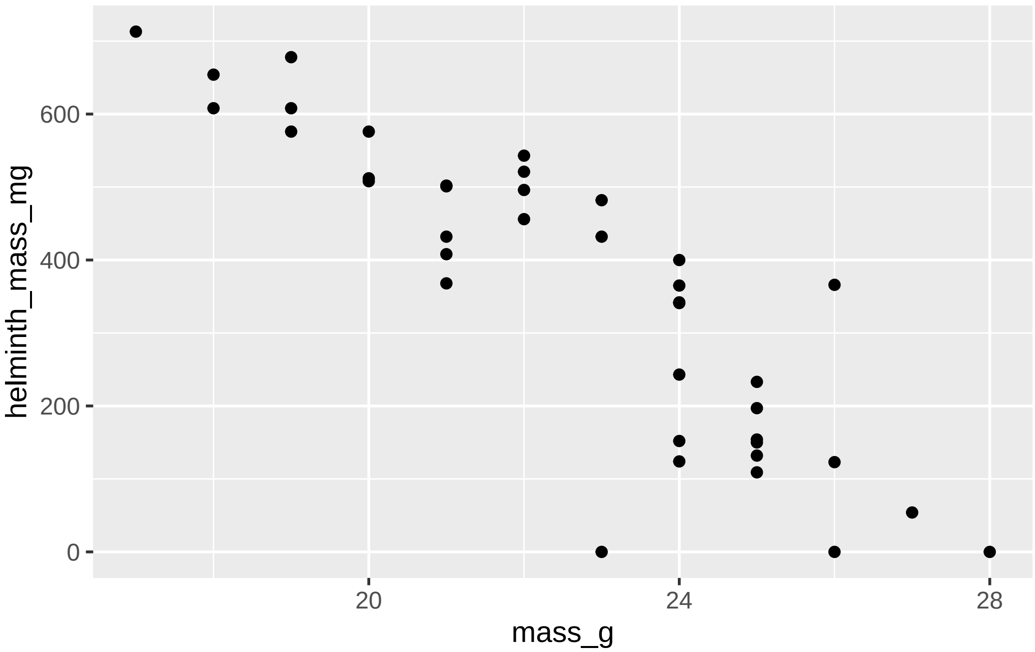
As you can see, your data format will determine whether you should use geom_col or geom_bar. Note: bar plots are generally only best-suited for counts among categories, when you're dealing with measured variables, there are better options below.

19.3.3 Two Variables: Both Continuous

If you are showing the relationship between two continuous variables, scatterplots with or without lines are usually the best way to go.

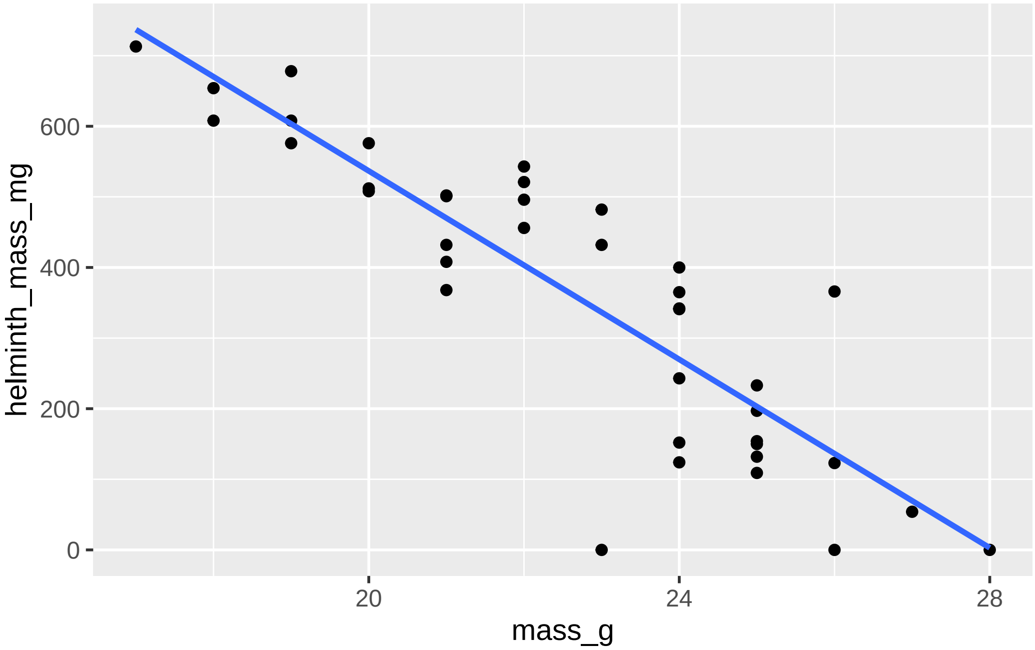
Let's try it out with the mammal data on body mass and helminth mass in mice:

```
## filter for mouse data
mouse_data <- filter(fake_mammals, species %in% c("White-footed mouse", "Deer mouse"))

## create ggplot with your two continuous variables as x and y
ggplot(data = mouse_data, aes(x = mass_g, y = helminth_mass_mg)) +
  geom_point() ## create scatterplot
```



```
## with trendline
ggplot(data = mouse_data, aes(x = mass_g, y = helminth_mass_mg)) +
  geom_point() +  ## create scatterplot
  ## create a trendline; method = "lm" makes it a straight line, se specifys whether there a
  geom_smooth(method = "lm", se = FALSE)
```

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

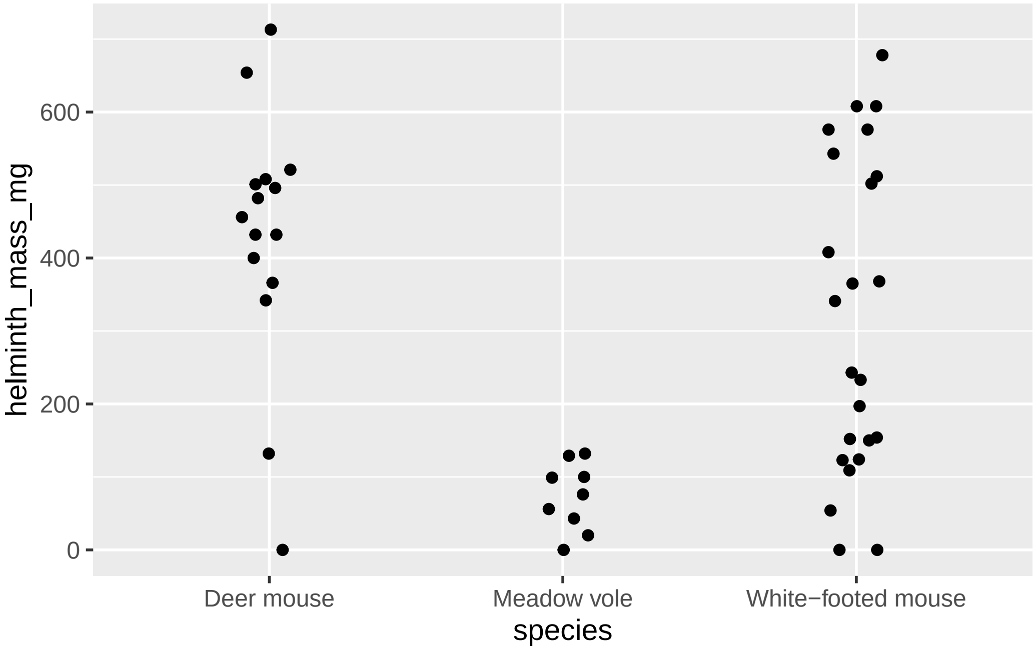


19.3.4 Two Variables: One Continuous, One Categorical

Believe it or not, when one of your variables is categorical, a scatterplot is still appropriate. Why not a bar plot? Because scatterplots show all of your data!

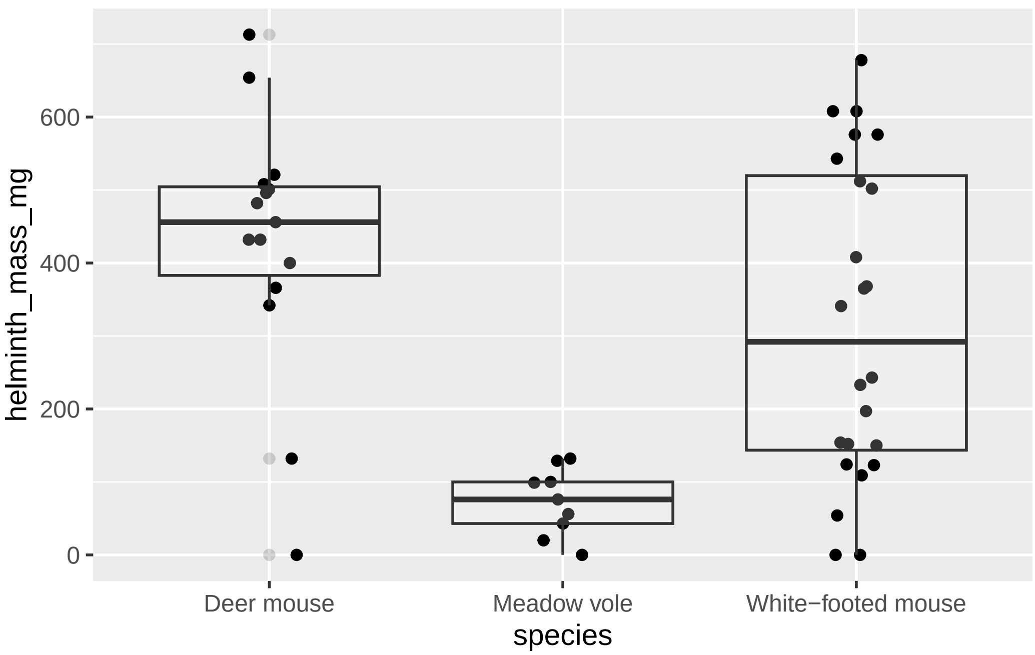
Let's demonstrate with the mammal data by comparing helminth mass among species.

```
## create ggplot with your two variables as x and y
ggplot(data = fake_mammals, aes(x = species, y = helminth_mass_mg)) +
  geom_jitter(width = 0.1, height = 0) ## create points that are "jittered" a bit along the x
```



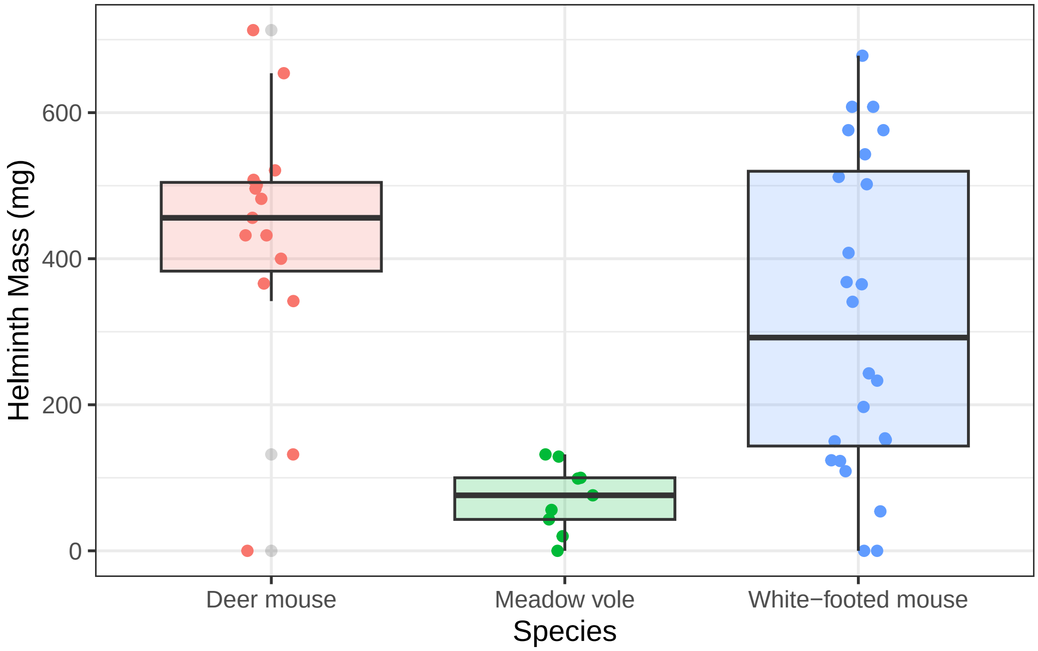
In this plot, we use geom_jitter to make the point spread a bit around each categorical X value so that you can see them better (but we specify height = 0 so as not to mess with the mass information). Instead of a mean helminth mass given by a bar plot, we can see the spread of each set of datapoints, including outliers or lack thereof. Still it's often nice to add some structure to these plots, which can be geom_boxplot or geom_violin (among others). Here is an example:

```
## create ggplot with your two variables as x and y
ggplot(data = fake_mammals, aes(x = species, y = helminth_mass_mg)) +
  geom_jitter(width = 0.1, height = 0) + ## create points that are "jittered" a bit along th
  geom_boxplot(alpha = 0.2) ## create boxplot at 20% transparency with alpha
```



We could also make this plot even clearer by adding color:

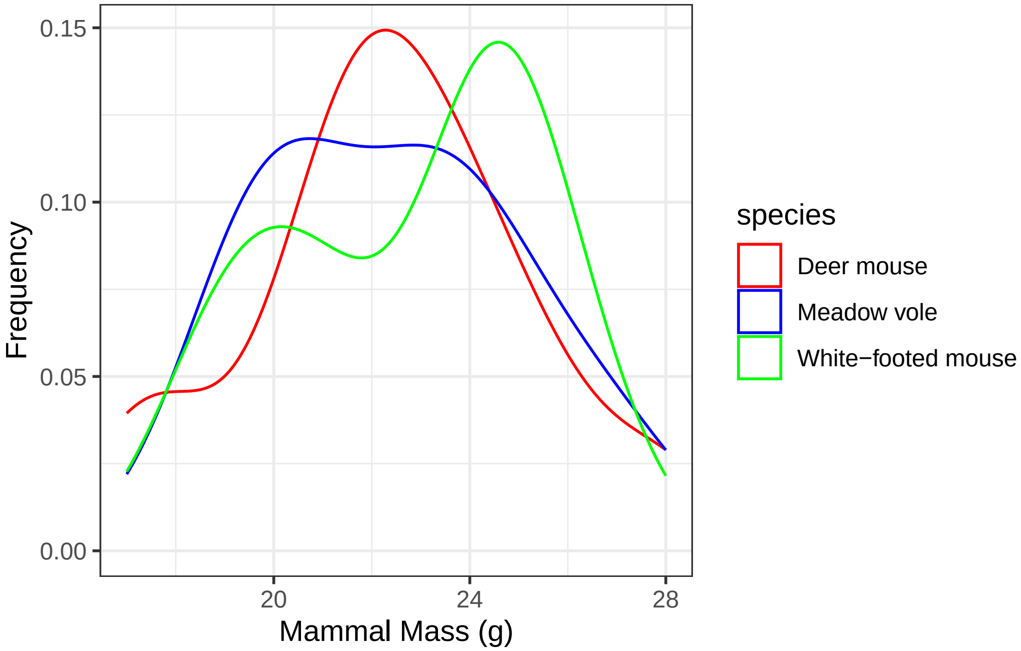
```
## create ggplot with your two variables as x and y
ggplot(data = fake_mammals, aes(x = species, y = helminth_mass_mg)) +
  geom_jitter(aes(color = species), width = 0.1, height = 0) + ## you can put aes() inside g
  geom_boxplot(aes(fill = species), alpha = 0.2) +
  labs(x = "Species", y = "Helminth Mass (mg)") +
  theme_bw() +
  theme(legend.position = "none") ## legend is redundant here, so we can hide it
```



19.3.5 Non-Axis Variables

You can also use other aesthetics to represent variables in your data. For example, you could use color to show the density plots of mammal masses among species. And you can modify the colors with scale functions:

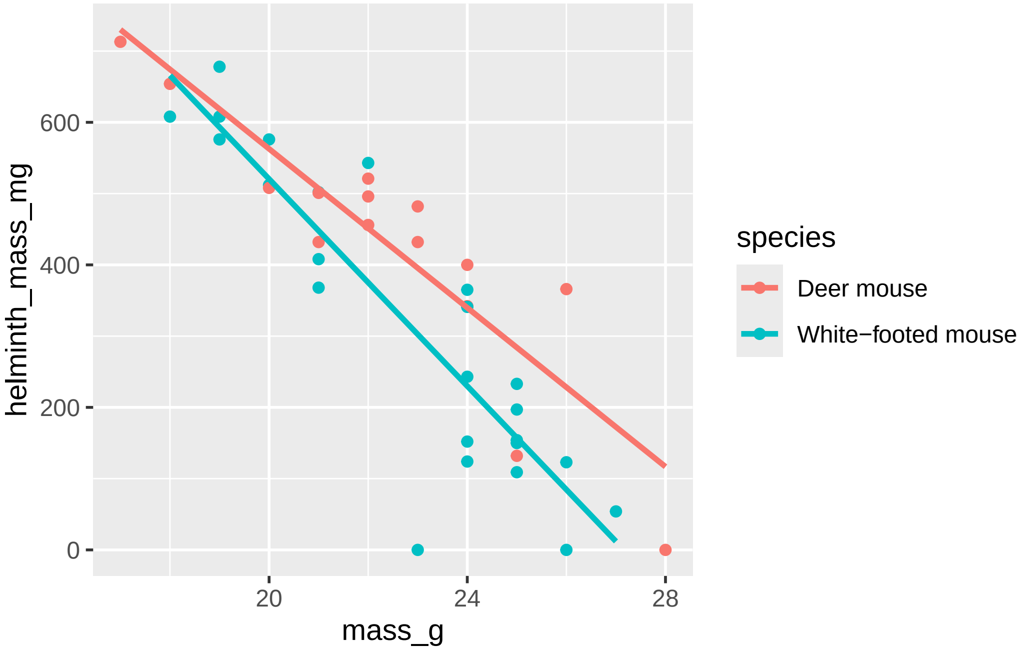
```
## make a ggplot, use wf_mice data, specify mass as the x variable and species as color
ggplot(data = fake_mammals, aes(x = mass_g, color = species)) +
  geom_density() + ## create density plot
  scale_color_manual(values = c("red", "blue", "green")) + ## set my own colors
  labs(x = "Mammal Mass (g)", y = "Frequency") + ## nicer labels
  theme_bw() ## my favorite simple theme
```



Similarly, you can add a third variable to a two variable figure. Take the helminth mass by mammal body mass figure from above:

```
ggplot(data = mouse_data, aes(x = mass_g, y = helminth_mass_mg, color = species)) +
  geom_point() +  ## create scatterplot
  geom_smooth(method = "lm", se = FALSE)
```

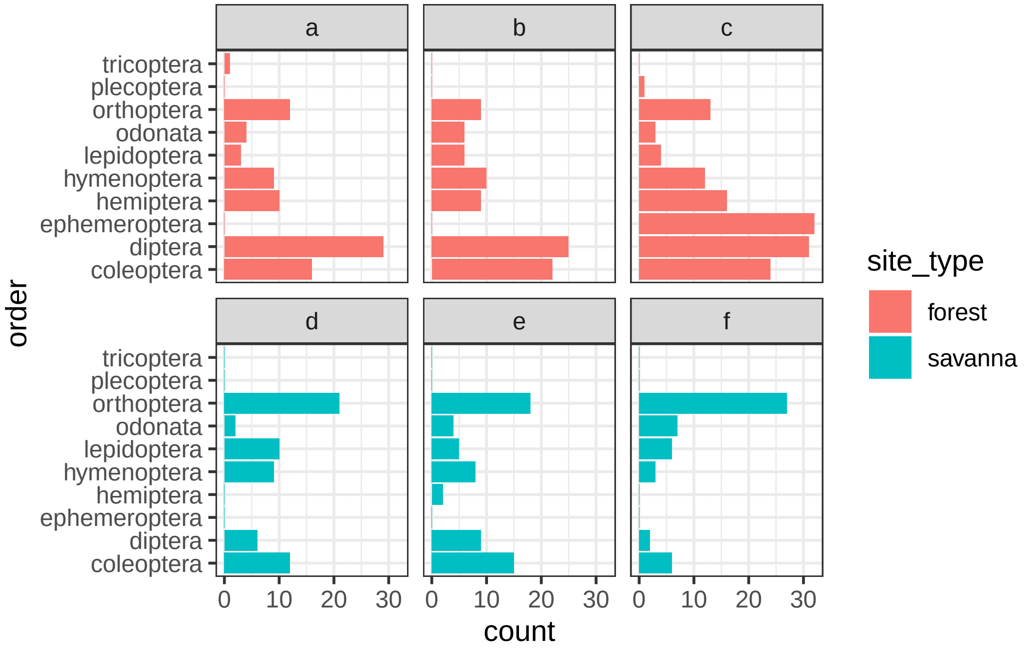
```
`geom_smooth()` using formula = 'y ~ x'
```



Color isn't the only way to show variables outside of axes, you can also use point shape, size, linetype, etc. In addition, you can split data among plot panels or "facets", with facet_wrap() or facet_grid().

Let's demonstrate with the long insect data, showing the insect communities for each site:

```r
## specify order as y variable to show labels better
ggplot(data = long_insects, aes(y = order, x = count, fill = site_type)) +
  geom_col() +
  facet_wrap(vars(site), nrow = 2) + ## specify site variable, two rows to separate habitats
  theme_bw()
```

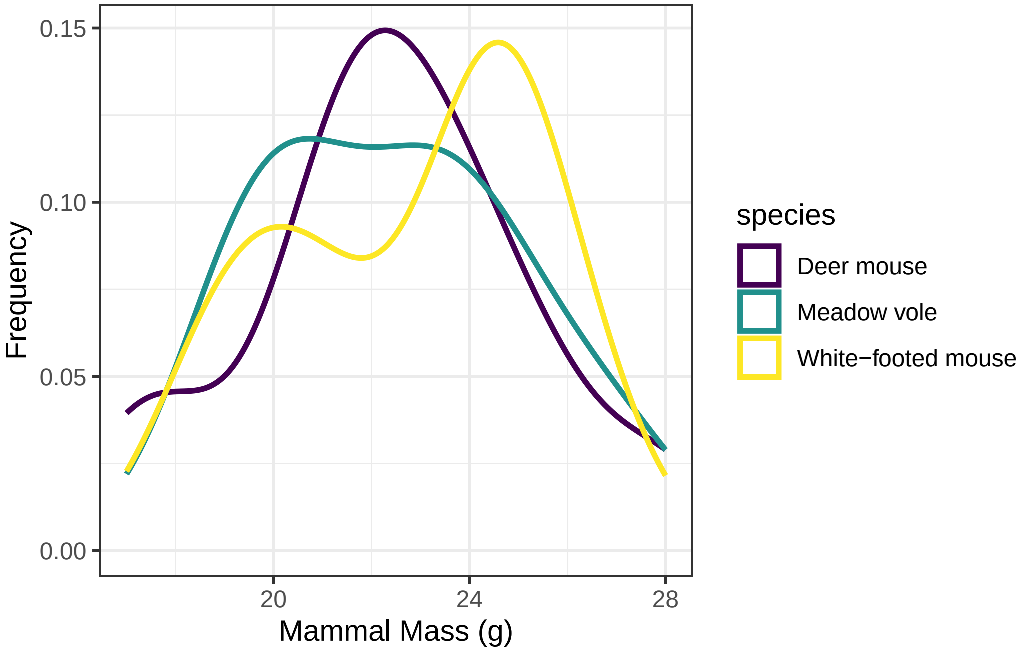


19.3.6 Colorblind Safe Colors

ggplot2 has colorblind-safe color schemes available from the tidyverse-related package viridis.

For example:

```
## make a ggplot, use wf_mice data, specify mass as the x variable and species as color
ggplot(data = fake_mammals, aes(x = mass_g, color = species)) +
  geom_density(linewidth = 1) + ## create density plot, wider lines
  scale_color_viridis_d() + ## set viridis discrete colors
  labs(x = "Mammal Mass (g)", y = "Frequency") + ## nicer labels
  theme_bw() ## my favorite simple theme
```



See the following link for more info:

https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html

19.4 Saving Figures

Much like your data summaries, you likely will want to save them. You can save figures from the RStudio user interface, i.e., the "Export" button in the "Plots" pane. With the Export button, you can save as an image, and specify the dimensions. This will generally work fine for sharing figures via email or for slide presentations.

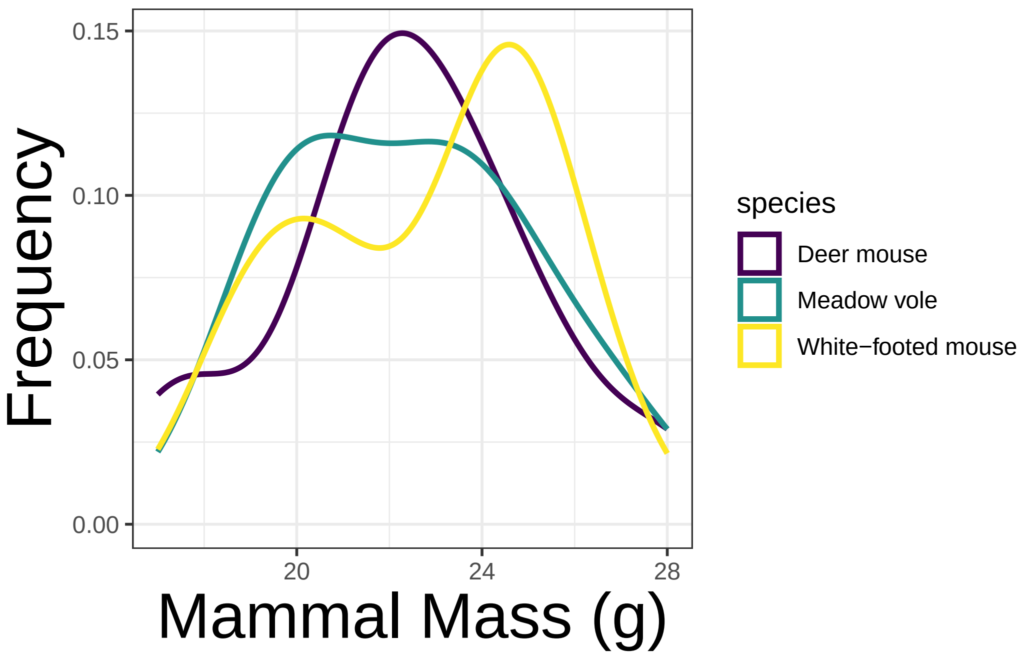
However, you can also use the "ggsave" function for figures made with ggplot, which can be useful for creating higher quality images for poster printing. Here is an example:

```
## first, you need to save your figure as an object in your R environment
## let's do this with the last plot we made
mammal_plot <-
  ggplot(data = fake_mammals, aes(x = mass_g, color = species)) +
  geom_density(linewidth = 1) + ## create density plot, wider lines
  scale_color_viridis_d() + ## set viridis discrete colors
  labs(x = "Mammal Mass (g)", y = "Frequency") + ## nicer labels
  theme_bw() ## my favorite simple theme
```

```r
## then you can save it to your project workspace!

## first let's generate a timestamp like we did for summary data
timestamp <- format(Sys.time(), format = "%Y%m%d_%H%M%S")

## and now use ggsave
## filename is the first argument, followed by the plot to save
## here I manually type ".tiff" in my paste function
## to specify a high quality image filetype
ggsave(paste("mammal_figure_", timestamp, ".tiff", sep = ""),
       plot = mammal_plot)
```

The ggsave function also features arguments to specify the dimensions of the image in inches, centimeters, pixels, etc. This allows you to make larger figures for posters (12x8 inches often works). However, larger images will often feature tiny text if you don't specify the axis labels to be larger fonts. You can specify text font size with the "theme" function when drawing a ggplot. For example:

```r
ggplot(data = fake_mammals, aes(x = mass_g, color = species)) +
  geom_density(linewidth = 1) + ## create density plot, wider lines
  scale_color_viridis_d() + ## set viridis discrete colors
  labs(x = "Mammal Mass (g)", y = "Frequency") + ## nicer labels
  theme_bw() + ## my favorite simple theme
  theme(axis.title = element_text(size = 24)) ## enlarge axis titles
```



There are tons of theme elements that you can modify with the theme function, and you generally have to specify them with “element” wrappers, such as “element_text” above. Do note that if you use a built-in theme like “theme_bw” as I do above, you need to add the “theme” function *after* the built-in theme to customize further.

At first it will take some trial and error to figure out the right combinations of font sizes in the ggplot theme and image sizes in ggsave, but eventually you will likely get a feel for it.

19.5 Further Reading

We have only scratched the surface of what ggplot2 can do! We barely discussed how to edit theme elements, nor did we spend much time on customizing scales.

ggplot2 has an excellent reference website which you can find here:

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

With it you can learn all the ins and outs!

Part V

Appendices

20 Appendix A: More R Resources

Here is a collection of links to other useful resources for learning R!

Ecology-themed tutorial:

https://datacarpentry.org/R-ecology-lesson/index.html

Basic / Base R Materials

Official R manuals:

https://cran.r-project.org/manuals.html

Cookbook for R (features lots of "recipes" for common tasks)

http://www.cookbook-r.com/

Primers and Cheatsheets ("tidyverse"-based)

RStudio primers

https://rstudio.cloud/learn/primers

Tidyverse cheatsheets

https://www.rstudio.com/resources/cheatsheets/

Full Books and Courses

Hadley Wickham's "R for Data Science"

https://r4ds.had.co.nz/index.html

Hadley Wickham's "Advanced R"

https://adv-r.hadley.nz/

Book for "ggplot2" package

https://ggplot2-book.org/

Jenny Bryan's STAT 545 course

http://stat545.com/

Package Function Reference Sites

ggplot2 (data visualization)

https://ggplot2.tidyverse.org/index.html

sf (spatial analysis)

https://r-spatial.github.io/sf/index.html

Miscellaneous Resources

On Style:

http://adv-r.had.co.nz/Style.html

On Reproducibility:

https://reproducible-analysis-workshop.readthedocs.io/en/latest/

https://swcarpentry.github.io/r-novice-gapminder/