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 analysis and using the R programming language.

Much of the content featured is adapted from the work of past CCESR Fellows, including Mariana Cardenas and Bea Baselga.

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 goes over data analysis in general, the second part describes R-related software and workflows, and the third part is intended to give a primer in R coding.

Part I

Data Analysis

1 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 more niche measures like dispersion.

You could also use **inferential statistics**, which is more about using math or simulation techniques to *infer* some conclusion form 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.

2 Data Types

First, let's go over different types of data:

2.1 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 exmaple 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 valuable 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 th biomass of ants on a log. Note: measures that consist of very large integer values 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.

2.2 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.

3 Descriptive Statistics

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

3.1 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

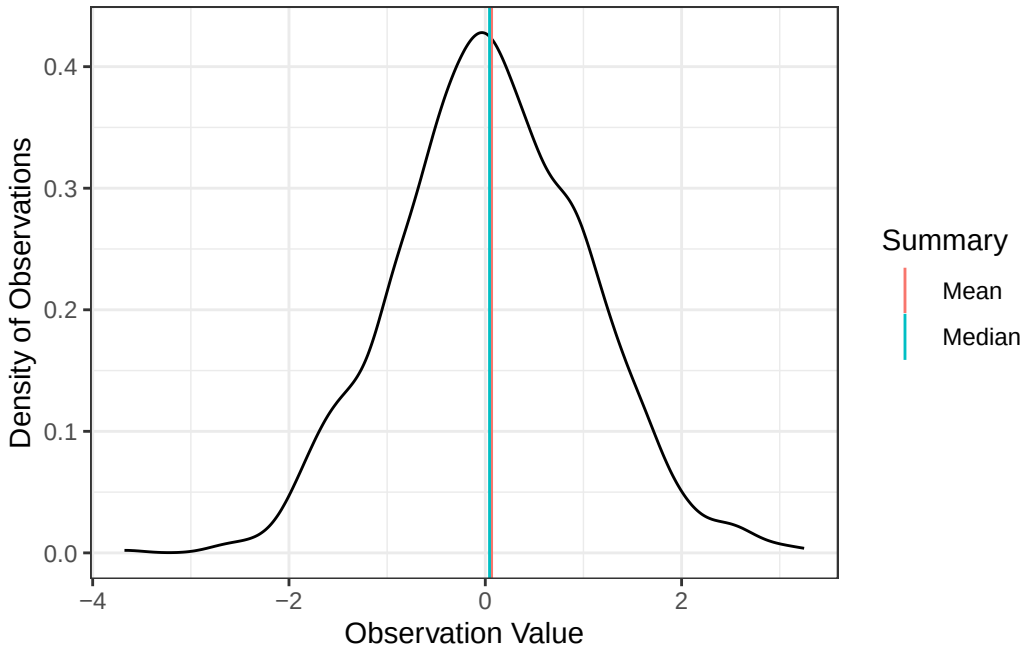
Mode - the most commonly observe value

Usually, the mean is a 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.

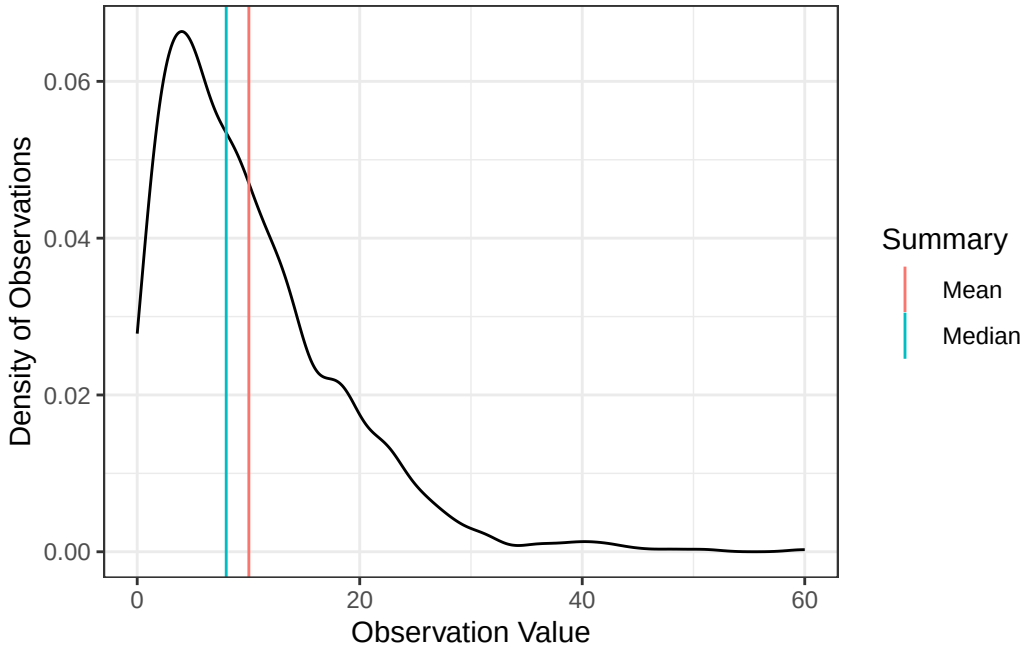
When might you prefer to use the median over the mean?

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:



3.2 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:

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

($\sum$ 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!

3.3 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.

3.4 Ecological Community Descriptors

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

Species Richness - this is just the number of different species present.

Species Diversity - this is an index that takes into account the richness as well as the relative abundances of each species. E.g. Shannon's Diversity Index, where higher numbers mean more species more evenly distributed.

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.

4 Inferential Statistics

Now, let's think about how to use your data to answer your questions. There are a couple approaches statisticians use, and we will talk about frequentist statistics, where probabilities are though 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 may have taken or will take!), but we are thinking in just the basics for now.

4.1 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).

4.1.1 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 eachother 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 height are normally distributed about 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 divcey 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.

4.1.2 Categorical Predictor/s, Numeric Response

4.1.2.1 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 changes, you can do a **t-test with unequal variance**.

The effect size here is the difference between means.

4.1.2.2 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' 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.

4.1.2.3 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

4.1.3 Numeric Predictor/s, Numeric Response

4.1.3.1 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.

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 likelihod estimation (don't worry about it) 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.

4.1.4 Numeric Predictor/s, Categorical Response

4.1.4.1 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 one category value over the other increases with a variable.

4.1.4.2 Multiple Response Categories

Multinomial regression *(under construction)*

4.1.5 Categorical Predictor/s, Categorical Response

Chi-square test *(under construction)*

4.2 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 generat 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 II

R on your Computer

5 R Itself

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

5.1 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.

5.2 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.

5.3 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.

6 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.

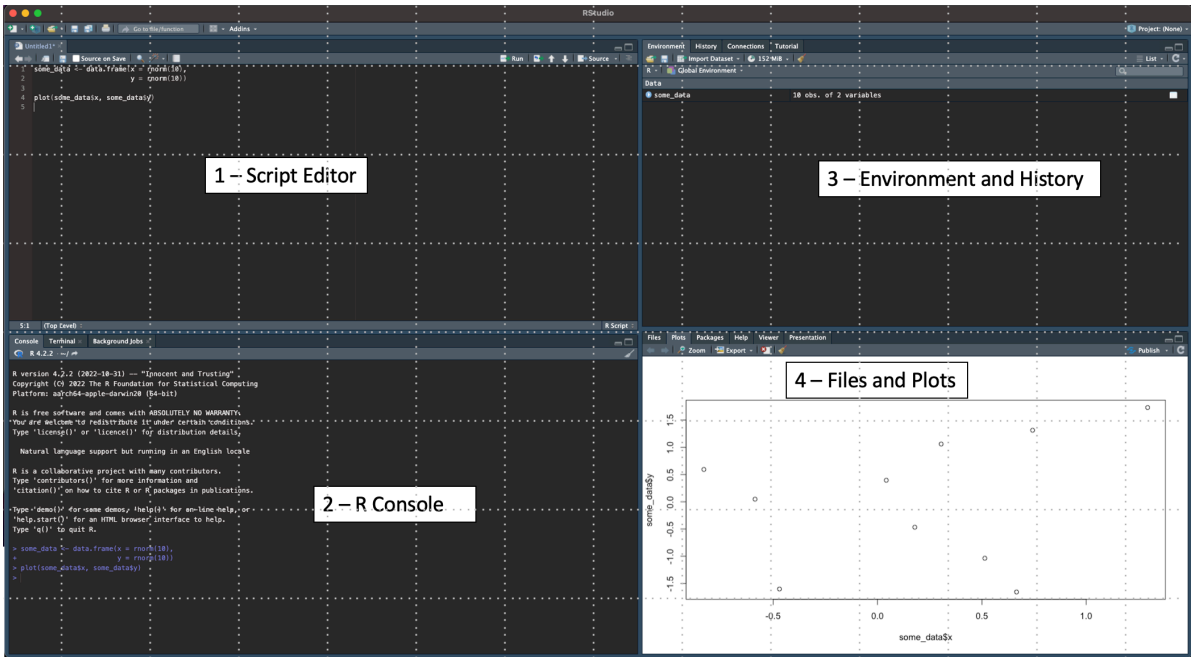
You can download and install RStudio from here:

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

Which should be more straightforward than downloading and installing R.

6.1 RStudio at a Glance

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



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.

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.

6.2 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.

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)

7 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 III

R Programming

8 The Basics

8.1 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 you today to you 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**.

8.2 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:

```
## 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.

8.3 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] "2023-07-22"
```

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

```
[1] "2023-07-22 16:35:50 CDT"
```

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

8.4 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 argumen
## n specifies how many numbers to return, and mean and sd specify shape of the distributi
rnorm(n = 10, mean = 5, sd = 1)
```

```
 [1] 4.285041 6.574632 4.293243 3.770870 5.612214 6.296137 6.308386 7.089698
 [9] 6.104815 3.120595
```

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
```

8.5 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).

8.5.1 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 the

## 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).

8.5.2 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.

8.5.2.1 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.

8.5.3 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 a 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 10
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
```

```r
## can also use the combine function, but it will default to a vector when data types are
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.

```r
## 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!

9 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.

9.1 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 silliness. 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:

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

9.2 Subsetting

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

9.2.1 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:

```
## 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"
```

9.2.2 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:

```
## 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"
```

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

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

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

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

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

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

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

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

```
## 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
```

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 is a 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
```

9.2.3 Data Frames

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.

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

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

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

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

```r
## 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
```

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

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

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

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

```
[1] TRUE
```

```r
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
```

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     1.1.1     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- 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:

```r
## 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
```

```r
## 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
```

9.3 Optional: Flow Control

9.4 Optional: Writing Functions

10 Importing Data

11 Wrangling Data

12 Summarizing Data

13 Analyzing Data

14 Visualizing Data

References