

Applied Biological Data Analysis

Nick Green

August 15, 2026

Table of contents

Preface	2
Work in progress warning!	2
Data used in this website	2
License and permissions	3
Course description	3
Course objectives	3
Course requirements	4
Recommended reading	4
Course organization	5
About the author	8
I Statistics in Modern Biology	10
1 Statistics in biology	12
1.1 Overview	12
1.2 Statistics in modern biology	12
1.3 How do we use statistics in biology?	14
1.4 <i>P</i> -values and null hypotheses	14
1.4.1 Null and alternative hypotheses	14
1.4.2 Another way to frame null hypotheses	17
1.4.3 History and status of <i>p</i> -values	18
1.4.4 Where <i>p</i> -values come from	18
1.4.5 What <i>p</i> -values mean and do not mean	23
1.4.6 Do you need a <i>p</i> -value?	24

II	Introduction to R	25
2	Introduction to R	27
2.1	Getting started with R	27
2.1.1	Download and install R (and RStudio)	27
2.1.2	Using R	28
2.2	Commands, objects, and output	32
2.2.1	R commands—basics	32
2.2.2	Using R functions	35
2.2.3	The R workspace	38
2.3	R syntax	39
2.3.1	Object size and structure	39
2.3.2	Operators	41
2.4	R packages	43
2.4.1	Your R library	44
2.4.2	Installing R packages	44
2.4.3	Working with packages in R	45
2.5	Frequently asked questions about R	50
2.5.1	What is R	50
2.5.2	Why do we have to write code?	50
2.5.3	Advantages and disadvantages	51
2.5.4	Base R vs. tidyverse	51
2.5.5	Where can I get R help?	53
3	R data types and objects	62
3.1	R data types	62
3.1.1	Character type	62
3.1.2	Numeric type	64
3.1.3	Integer type	64
3.1.4	Logical type	66
3.1.5	Special values	68
3.2	Basic R data structures	70
3.2.1	Vectors	70
3.2.2	Data frames	77
3.2.3	Matrices and arrays	80
3.2.4	Lists	83
3.2.5	S3, S4, and other paradigms	91

4	Data import and export	92
4.1	Working with your file system	92
4.1.1	Your file system	92
4.1.2	Folder addresses	92
4.1.3	The working directory	93
4.2	Formatting data files	95
4.2.1	Naming data files	95
4.2.2	Setting up data files	96
4.3	Data import	100
4.3.1	Importing from text files	100
4.3.2	Saving and importing R objects	104
4.3.3	Importing from saved workspaces	105
4.3.4	Importing special cases	106
4.4	Data export	108
4.4.1	Exporting text files	108
4.4.2	Saving R objects	110
4.4.3	Saving workspaces	110
4.4.4	Write text to a file (special cases)	111
4.4.5	Exporting graphics	111
5	Data manipulation with R	113
5.1	Making values	113
5.1.1	Make arbitrary values with the <code>c()</code> function	113
5.1.2	Generating sequences	114
5.1.3	Random values	119
5.2	Selecting data with square brackets	123
5.2.1	Basics of brackets	123
5.2.2	Selecting and extracting with logical tests	133
5.3	Dates and temporal data	137
5.3.1	Dates without times	138
5.3.2	Dates with times and time zones	140
5.3.3	Working with dates and times	143
5.4	Character data	145
5.4.1	Combining character strings	145
5.4.2	Printing character strings	146
5.4.3	Matching character strings	148
5.4.4	Searching text with the <code>grep()</code> and <code>grepl()</code> functions	149

5.4.5	Replacing parts of character strings	150
5.4.6	Extracting from and splitting strings	151
5.4.7	Matching text strings as indices	152
5.4.8	Standardizing numbers as text	154
5.5	Data frame management	154
5.5.1	Data frame structure	155
5.5.2	Common operations	156
5.5.3	Other operations	168
III	Exploratory data analysis with R	177
6	Basic descriptive statistics	179
6.1	Sample and population statistics	179
6.2	Central tendency	182
6.2.1	Means	183
6.2.2	Median	186
6.2.3	Mode	187
6.3	Variability	190
6.3.1	Variance and standard deviation (SD)	190
6.3.2	Median absolute deviation (MAD)	193
6.3.3	Quantiles and ranges	193
6.4	Covariance and correlation	196
6.4.1	Linear correlation	198
6.4.2	Nonlinear correlation	201
7	Summarizing biological data	204
7.1	Summarizing a data frame	204
7.1.1	Summarizing by groups (AKA: pivot tables)	205
7.1.2	Summarizing by arbitrary functions	210
7.2	The <code>apply()</code> family	212
7.2.1	<code>apply()</code> for arrays and data frames	212
7.2.2	<code>lapply()</code> and <code>sapply()</code> for lists and other objects	216
7.3	Tabulation (frequency tables)	217
7.3.1	Frequency tables with <code>'table()'</code>	217
7.3.2	Contingency tables with <code>ftable()</code>	219

8 Visualizing biological data	224
8.1 Visualizing single variables	224
8.1.1 Histograms	227
8.1.2 Barplots	232
8.1.3 Boxplots	235
8.1.4 ECDF plots	241
8.1.5 Kernel density plots	248
8.2 Visualizing 2 variables	253
8.2.1 Scatterplots for 2 continuous variables	254
8.3 Visualizing many variables	273
8.3.1 Scatterplot matrices for many variables	273
8.3.2 Ordination (brief introduction)	279
9 Probability distributions for biologists	286
9.1 Introduction	286
9.1.1 Residuals: the random variation left over	286
9.1.2 Probability distributions in R	292
9.2 Discrete distributions	295
9.2.1 Bernoulli distribution	295
9.2.2 Binomial distribution	296
9.2.3 Poisson distribution	303
9.2.4 Negative binomial distribution	305
9.2.5 Geometric distribution	308
9.2.6 Multinomial distribution	312
9.3 Continuous distributions	313
9.3.1 Uniform distribution	317
9.3.2 Normal distribution	320
9.3.3 Lognormal distribution	326
9.3.4 Gamma distribution	330
9.3.5 Beta distribution	332
9.3.6 Exponential distribution	337
9.3.7 Triangular distribution	338
9.3.8 Distribution summary	342

10 Comparing data to distributions	344
10.1 Fitting and testing distributions	344
10.2 Estimating distributional parameters	344
10.3 Graphical methods for examining distributions	346
10.3.1 Comparing distributions with PDF or PMF plots	346
10.3.2 Comparing distributions using CDF and ECDF	350
10.3.3 Comparing distributions using QQ plots	352
10.4 Formal tests for distributions	355
10.4.1 Tests for normality	355
10.4.2 Kolmogorov-Smirnov tests	356
11 Data transformation and rescaling	361
11.1 Data transformations	361
11.2 Why transform?	361
11.3 Transforms vs. link functions	362
11.4 Common transformations	364
11.4.1 Log transformation	364
11.4.2 Log transforming with 0 values	368
11.4.3 Rank transformation	372
11.4.4 Logit and probit transformations	374
11.5 Uncommon transformations	383
11.5.1 Square-root transformation	383
11.5.2 Arcsine square root transformation	384
11.5.3 Box-Cox transformation	387
11.6 Transformations closing thoughts	393
12 Common problems in biological data	394
12.1 Outliers and erroneous values	394
12.1.1 Detecting outliers	394
12.1.2 Dealing with outliers	396
12.2 Autocorrelation	399
12.2.1 Graphical methods to identify autocorrelation	400
12.2.2 Mantel tests for autocorrelation	402
12.2.3 Dealing with autocorrelation	403
12.3 Collinearity	403
12.4 Missing data	406
12.4.1 Option 1: Ignore missing data	406
12.4.2 Option 2: Interpolate or impute missing data	407

13 Planning your data analysis	409
13.1 How to use this guide	409
13.2 What question are you trying to answer?	410
13.2.1 Quantifying your question	410
13.2.2 Types of questions	411
13.3 Testing for a difference in mean or location	411
13.3.1 Additional considerations	412
13.4 Testing for a continuous relationship between two or more variables	412
13.5 Relationships between categorical variables	414
13.6 Classifying observations	414
13.7 Conclusions	415
 IV Introductory statistics with R	 416
14 Introductory statistics with R	418
14.1 Datasets and variables	418
14.2 Frequentist statistics	421
14.2.1 p -values and NHST	422
14.3 Alternatives to frequentist statistics	424
14.3.1 Information-theoretic methods	424
14.3.2 Bayesian inference	429
14.4 Linear models	433
14.4.1 What is “linear”?	433
14.5 Analysis of variance (ANOVA)	440
 15 Tests for differences in mean or location	 441
15.1 Tests for 1 group	441
15.1.1 One-sample, one-tailed t -test	442
15.1.2 One-sample, two-tailed t -test	444
15.2 Tests for 2 groups	445
15.2.1 Student’s t -test (Welch) (2-tail, 2-sample)	445
15.2.2 Two-sample, one-tailed t -tests	450
15.2.3 Equivalence testing	453
15.2.4 Wilcoxon rank-sum test	455
15.3 Tests for paired data	457
15.3.1 Paired t -test	457
15.4 t -test summary	458

15.5 Tests for 3 or more groups	461
15.5.1 Analysis of variance	461
15.5.2 Kruskal-Wallis test	472
15.6 Tests with ordered factors	473
16 Tests for continous patterns	477
16.1 Correlation	477
16.1.1 Linear correlation	479
16.1.2 Nonlinear correlation	482
16.2 Linear models	485
16.2.1 Lots of analyses are linear models in disguise (or can be)	485
16.3 Linear regression	488
16.4 Multiple regression	492
16.5 Analysis of covariance (ANCOVA)	497
16.5.1 Main effects and interaction effects	502
17 Tests for contingency	503
17.1 Fisher's exact test	503
17.2 Chi-squared test	505
17.3 G-test	507
17.4 Contingency tests and odds ratios	508
V Generalized linear models (GLM)	509
18 Introduction to generalized linear models (GLM)	511
18.1 Prelude with linear models	512
18.2 GLM basics	512
18.2.1 Overall structure	512
18.2.2 Response distributions	512
18.2.3 Linear predictors	512
18.2.4 Link functions	512
18.3 Common GLMs	512
18.3.1 GLMs for continuous data	512
18.3.2 GLMs for count data	512
18.3.3 GLMs for binary and proportional data	512
18.4 Deviance and other GLM diagnostics	512
18.4.1 Deviance	512
18.4.2 Pseudo- R^2 or not?	512
18.4.3 AIC and likelihood functions	512

19 GLM for continuous response data	513
19.1 Gaussian GLM	513
19.2 Log-normal GLM	513
19.3 Gamma GLM	513
20 GLM for count data	514
20.1 Poisson GLM for counts	514
20.2 Negative binomial GLM for overdispersed counts	514
21 GLM for categorical and proportional responses	515
21.1 Logistic GLM for binary responses	515
21.1.1 Odds ratios	515
21.2 Binomial GLM for proportions	515
21.3 Ordinal logistic regression	515
VI Extensions of GLMs: nonlinear and mixed models	516
22 Nonlinear methods for biology	518
22.1 Nonlinear least squares	518
22.2 Generalized additive models (GAM)	518
22.3 Generalized nonlinear models (GNM)	518
22.4 Regression trees for nonlinear patterns	518
23 Random effects and mixed models	519
23.1 Fixed and random effects	519
23.2 Linear mixed effects models (LMM)	519
23.3 Generalized linear mixed effects models (GLMM)	519
23.4 Nonlinear mixed effects models (NLME)	519
23.5 Additive mixed models (GAMM)	519
VII Multivariate data analysis	520
24 Introduction to multivariate data and data analysis	522
25 Dissimilarity and distance	523
25.1 Mantel tests	523
26 Clustering for biologists	524

27 Ordination	525
27.1 Principal components analysis	525
VIII Alternatives to p-values: information-theoretic and Bayesian methods	526
28 Multivariate hypothesis testing	528
28.1 Comparing groups	528
28.1.1 MANOVA	528
28.1.2 PERMANOVA	528
28.1.3 MRPP	528
28.2 Analyzing dissimilarity	528
28.2.1 Mantel tests	528
29 Information theoretic model-based inference	529
IX Bonus material	530
30 Introduction to Bayesian inference for biologists	532
31 Advanced R graphics	533
32 Classification and regression trees	534
Appendices	535
A Appendix A	535

Preface

This website accompanies a course I have developed at Kennesaw State University: **BIOL 7410: Applied Biological Data Analysis**. The target audience is biologists who need to analyze data for their research projects. If you find the material here helpful, please let me know! Likewise, if you have comments for improvement, I'd love to hear from you: send all comments to ngreen62@kennesaw.edu!

If you are a returning visitor, please note that the website has been rearranged since 2024. This was done to make the modules smaller and more digestible.

The Fall 2025 course syllabus can be [downloaded here](#).

Work in progress warning!

I am actively updating this site throughout the Fall 2026 semester. This means that many cross-references may be broken, pictures missing, etc. Please be patient with me as I get things back into working order.

Data used in this website

This website presents many examples of data analyses or data manipulation. In past versions of the site (and my course), I have tried to use different and interesting datasets for every module, so that examples would feel more relevant and fresh. However, student feedback has convinced me to pursue the opposite strategy: use a single dataset for most examples, so that people could get more familiar with the dataset over the course of the semester. This would prevent having to learn a completely new dataset and biological system every day in class.

However, it turns out that it's hard to do everything with a single dataset. So, I settled on 3 datasets:

1. The famous `iris` dataset included with R. This will be used primarily in early modules covering data manipulation and data frame management, as well as the review of basic statistics in Modules 14 to 17.
2. Datasets simulated *ad hoc* to demonstrate simple concepts with no underlying biological reality (e.g., numbers to demonstrate the properties of a probability distribution).

3. Small mammal trapping and measurement data from the [National Ecological Observatory Network \(NEON\)](#)(National Ecological Observatory Network (NEON) 2025). The details of which data I used and how I packaged them for this website can be found in Appendix A.

License and permissions

This work and its content is released under the [Creative Commons Attribution-ShareAlike 4.0](#) license. This means that you are allowed to share the material and adapt it for your purposes, under the following conditions:

1. Give appropriate credit to the [author](#). This includes citing the original sources of any third-party datasets used in some of the tutorials. I've done my best to provide appropriate citations alongside these datasets.
2. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

This site is for educational purposes only. This work and its content is released under the [Creative Commons Attribution-ShareAlike 4.0](#) license. Inclusion of third-party data for educational purposes falls under guidelines of fair use as defined in [section 107 of the US Copyright Act of 1976](#).

Course description

This course is a survey of data analysis skills and statistical methods essential for careers in biology. This course takes a holistic approach to the data analysis workflow using the open-source software R, including: data management, exploratory analyses, modeling, and reproducible science practices. Statistical topics covered may include generalized linear models, mixed effects models, nonlinear models, and ordination. Students apply techniques learned in class to real biological datasets as a course project.

Course objectives

1. Engage with current biology literature to explain the role of statistics in the biological sciences and the ways in which analytical results are communicated.
2. Engage in research using modern statistical methods to investigate biological questions.
3. Manipulate, summarize, display, and analyze data using the open-source environment and language R.
4. Use probability distributions to model and think about biological phenomena.
5. Translate biological hypotheses and ideas into statistical models and vice versa.
6. Conduct exploratory data analyses in support of scientific investigations, particularly to detect and diagnose common problems with biological datasets.

7. Communicate data and analytical results to audiences who may or may not have statistical backgrounds.

Course requirements

- Access to a computer or laptop capable of running [R](#) version 4.0 or above. This site was rendered using R version 4.4.3 and the most recent version of any package mentioned (as of August 8, 2025).
- A 64-bit system is highly recommended for speed and stability.
- A separate program in which to write and edit code.
 - Most people prefer to use an “integrated development environment” (IDE) or dedicated code editor for coding. I recommend some in Module 2.
 - Most people nowadays prefer [RStudio](#). The desktop version is free and has a lot of nice features that make working with R easier.
 - IDEs and code editors are often subject to irrationally strong personal and organizational preferences, so just pick the one that works for you. Or, use the one your supervisor tells you to use.
 - My usual workflow includes two windows: the base R GUI, and [Notepad++](#).
- It would be helpful to have a dataset of your own to work with. Each course module uses numerous examples with simulated or published datasets. But, at the end of the day, you’re probably here because you need to apply these methods to your own work.

Recommended reading

This course has no required textbook. If you want a textbook, then I recommend one of the following:

- *Ecological models and data in R* by Ben Bolker (Bolker 2008). This is a general statistics textbook for biologists. The applications and examples are focused on ecology, but the statistical exposition is relevant to any area of biology.
- *Introductory statistics with R* by Peter Dalgaard (Dalgaard 2002). This is a general statistics textbook and introduction to R programming.
- *A beginner’s guide to R* by Alain Zuur and others (Zuur *et al.* 2009b). This book introduces R in a different way than Dalgaard (2002).
- *An introduction to statistical learning: with applications in R* by Gareth James and others (James *et al.* 2013). This is a more advanced statistics book that also introduces machine learning. The second edition is available for [free online](#).

Parts of the course also lean heavily on some more specialized textbooks:

- *Analysing ecological data* (Zuur *et al.* 2007) and *Mixed effects models and extensions in ecology with R* (Zuur *et al.* 2009a) by Alain Zuur and others. These are advanced textbooks aimed at ecologists which introduce GLMs, mixed models, and other topics we will explore in this course.
- *Analysis of ecological communities* by Bruce McCune and James Grace (McCune *et al.* 2002) and *Numerical ecology* by Pierre and Louis Legendre (Legendre and Legendre 2012) are introductions to ordination and other multivariate methods. McCune *et al.* (2002) is a gentler introduction but depends on [proprietary software](#). Legendre and Legendre (2012) is a little more technical (but still relatively easy to understand) and uses R throughout.
- *An introduction to WinBUGS for ecologists* by Marc Kéry (Kéry 2010). This book introduces Bayesian inference using the free programs WinBUGS or JAGS, which can be accessed from R.

An online version of this book is available at <https://greenquanteco.github.io/appl-biol-data-analysis/>. The online version and the PDF hosted there are updated from time to time.

Course organization

This work is roughly organized into 9 parts. The links below will take you straight to the start of each module.

Part 1 introduces the use of statistics in modern biology, setting the stage for the rest of the course.

[Module 1](#) is an introduction to the use of statistics in biology, and explores some fundamental issues related to how biologists make inferences based on data.

Part 2 is an introduction to R split across several modules. Beginners should start in **Module 2**.

[Module 2](#) is an introduction to the R program and programming language.

Module 3 describes some important R data and object types—how R stores data.

[Module 4](#) covers methods for importing data in R, and exporting results from R.

[Module 5](#) takes a deep dive into manipulating data in R. This includes selecting data, relating data tables, and data frame management.

Part 3 discusses **exploratory data analysis** and **planning your data analysis**. It begins with basic statistical procedures such as descriptive statistics, data summarization, and describing variables using probability distributions. It also includes a module on planning your analysis.

[Module 6](#) describes **descriptive statistics**, or summaries, of single variables.

[Module 7](#) describes how to summarize and aggregate data by groups.

[Module 8](#) introduces basic data visualizations like histograms, probability density plots, boxplots, and scatterplots.

Module 9 introduces probability distributions and how to use them to describe your data.

Module 10 covers ways to explore how your data fit different probability distributions.

Module 11 covers common data transforms, like the log, logit, and many more.

Module 12 covers common statistical problems in biology, including lack of replication, autocorrelation, collinearity, and missing data.

Module 13 is a guide to selecting statistics for your research. This module contains dichotomous keys to help you decide what statistical methods are most appropriate for the question you are trying to analyze.

Part 4 is a review of introductory statistics—basically everything you learned in your undergraduate statistics course, but could probably use a refresher on.

Module 14 reviews the structure of biological datasets, null hypothesis significance testing, and the 4 classes of statistical tests used by biologists.

Module 15 reviews tests for comparing the means or locations of groups of observations: *t*-tests, analysis of variance (ANOVA), and their nonparametric alternatives.

Module 16 reviews procedures for testing continuous patterns: correlation, linear regression, and analysis of covariance (ANCOVA). This module also takes a deep dive into **linear models** and their fundamental importance for understanding more advanced methods.

Module 17 reviews tests for contingency and independence such as Fisher's exact test, the χ^2 test, and the G-test.

Part 5 introduces the powerful framework of the generalized linear model (GLM) for modeling for modeling data with a single response variable.

Module 18 introduces generalized linear models (GLM), an extremely general and powerful framework for analyzing biological data.

Module 19 demonstrates GLMs for continuous data: Gaussian, log-normal, and Gamma GLMs.

Module 20 covers GLMs for count data: Poisson and negative binomial GLMs.

Module 21 covers GLMs for binomial data and proportional: logistic regression and binomial GLM.

Part 6 builds upon the GLM framework to introduce nonlinear models and mixed models, which can accommodate many situations where GLMs are inadequate.

Module 22 explores nonlinear models for relationships that don't follow a straight line. These models include biological growth curves and dose response curves.

Module 23 introduces mixed models, which extend GLM and other models to include random effects. Mixed models allow biologists to account for variability from unknown sources.

Part 7 covers multivariate statistics, methods which can consider many variables at a time.

Module 24 introduces **multivariate** statistics, which explore patterns in 2 or more variables at once. This module covers the structure of multivariate datasets.

[Module 25](#) introduces **distance measures** and **dissimilarity metrics**, methods for quantifying how much observations differ from each other. Many of the multivariate methods covered in subsequent modules depend on distance measures.

and distance measures.

[Module 26](#) explores **cluster analysis**, a class of techniques for identifying groups within multivariate data.

[Module 27](#) explores **ordination**, a class of techniques for visualizing patterns in multivariate data. Ordination also includes **dimension reduction**, or the consolidation of information from many dimensions into few.

[Module 28](#) describes some multivariate hypothesis tests, many of which are extensions of univariate techniques.

Part 8 introduces two statistical frameworks that can be an alternative to p -values and null hypothesis significance testing.

[Module 29](#) introduces information-theoretic model selection, an alternative to null hypothesis significance testing (NHST) that evaluates entire models rather than individual parameters and their p -values.

[Module 30](#) introduces Bayesian inference using the program JAGS, another alternative to NHST.

Part 9 is “bonus material”; i.e., additional modules that didn’t quite fit in anywhere else. New modules will be added here until the next major update.

[Module 31](#) takes a deep dive into building quality graphics using base R.

[Module 32](#) introduces classification and regression trees (CART), a technique for flexibly capturing and modeling patterns in data without first deciding on a data model.

Finally, Appendix A describes the NEON dataset that I used to build Parts 4 through 8, and the steps taken to process the raw data download to the files provided here.

About the author



I'm a [quantitative ecologist at Kennesaw State University](#) who studies human impacts on animal populations. Since earning my PhD at Baylor University in 2012, I've been fortunate to have worked in a variety of government and industry roles. Those experiences have shaped my perspective on biology and education. In my courses and advising I do my best to impress upon students the variety of biological experiences and perspectives that different domains bring to the table...and the value that they all bring. Most of my [current research](#) focuses on understanding human impacts on animal communities and how animal populations adapt to rapidly changing environments.



Why am I making this guide? I have been using R since about 2008, when I was a graduate student trying to make a multi-panel plot for a manuscript. I had seen R mentioned in publications and thought that using R would be more effective than kludging something together with Excel and Powerpoint. Needless to say, jumping straight to making plots with the `lattice` package was not the easiest way to learn R. The final result was a little clunky (Kirchner *et al.* 2011) but I am still really proud of it. Eventually I took some courses and did a lot of self-teaching, and am finally (March 2025) pretty decent at R. I have used R in my work pretty much continuously since 2008 for everything from Bayesian data analysis (e.g., Green *et al.* 2020) to simulation modeling (e.g., Green *et al.* 2019).

This guide is meant to be a primer for self-starting with R. In many sections the text is a bit lengthy, but that is because I wanted to provide explanations that might save someone some of the trouble I had getting started. R documentation is notoriously terse and compact—this is great for professionals who need a reminder, but not so much for newcomers who are still trying to learn. This guide also evolved out of course notes, and will continue to evolve based on feedback from students. If you have any feedback or suggestions for improvement, please [send me a message!](#)

Part I

Statistics in Modern Biology

Do we have to?

Chapter 1

Statistics in biology

1.1 Overview

Science works by testing ideas to see if they line up with observations. In other words, checking the predictions of hypotheses against *data*. This process is called **data analysis**, because it involves careful examination of numerical patterns in observational or experimental data. The branch of mathematics concerned with analyzing data is called **statistics**. Statistics is vital to modern science in general, including biology.

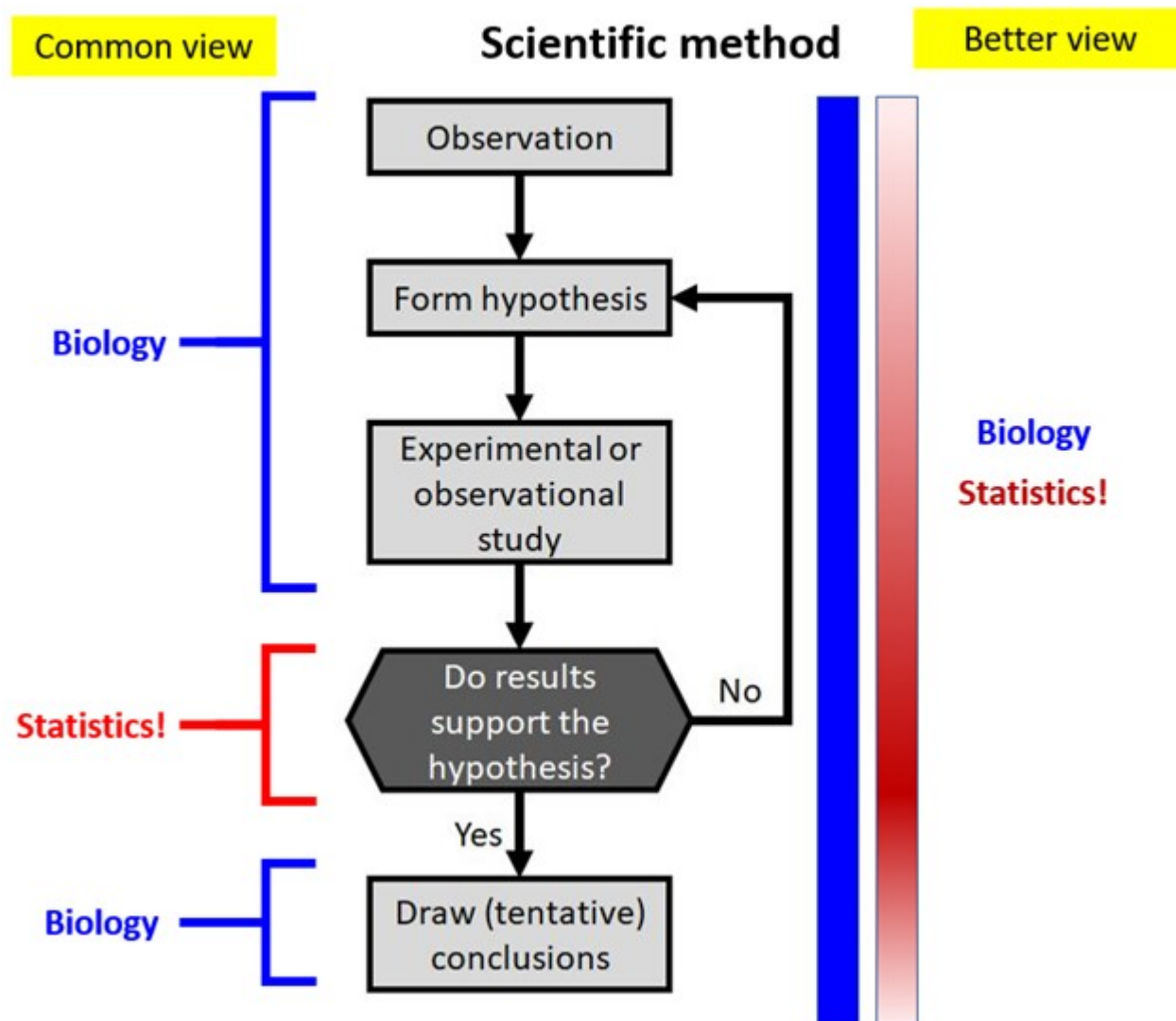
This website and the course it supports is designed to introduce the ways that statistics are used in biology. This includes strategies for data analysis and an introduction to the open source program and language [R](#). Specific tasks in the data analysis workflow (e.g., data manipulation) and specific analysis methods (e.g., generalized linear models) are covered in their own modules. This first module is a more general exploration of the role of statistics in modern biology, common statistical frameworks, and common misuses of statistics.

1.2 Statistics in modern biology

Biology, like all sciences, depends on the **scientific method**. This means testing proposed explanations, or **hypotheses**, against the data to see if the data support the hypothesis or not. That critical step, of deciding whether observations support or refute a hypothesis, is the domain of **statistics**: the branch of mathematics concerned with finding and interpreting patterns in data.

Many biologists traditionally think of “biology” and “data analysis” as two completely separate enterprises, with statistics being like a special wrench that a mechanic pulls out of the toolbox so infrequently that its use is unfamiliar. I think this is a mistake. As the figure below shows, every part of the scientific method can benefit from some knowledge of statistics. This includes experimental design, data collection, and interpreting the findings.

Figure 1.1: Statistics and thinking about data is part of the entire scientific process, not just a single stage.



So why do we, as biologists, need statistics? Some common answers might include:

1. "To tell us whether the patterns we see in our data are real"
2. "To tell whether the patterns in our data are due to chance or not."
3. "To prove our hypotheses."

All 3 of these are completely wrong. **1** is wrong because confirming reality requires observation and experiment, not calculating. **2** is a common misconception. *Any* pattern in data could be caused by chance. Statistics can help you estimate that probability, but cannot tell you for sure. Finally, **3** is wrong because that's not how science works. Hypotheses can be supported or not supported. No amount of statistical analysis will ever completely rule out chance.

So the answer to the question, “Why do we as biologists need statistics?” is that we need a systematic way to determine whether data support hypotheses or not. Note that I said “systematic”, not “absolutely correct”. As we will see shortly, statistical tests are only a guide to decision-making. We should not fool ourselves into thinking that our analyses are completely objective. Many aspects of the analysis process are in fact *subjective* and thus any analysis has to be taken in its proper context, with caveats and biological reality always in mind.

When we “do statistics”, we are arguably not even answering biological questions. Every statistical test answers a particular form of question:

- “Do groups A and B have the same mean?”
- “As variable X changes, does variable Y tend to change?”
- “If variable X increases by this much, how much does variable Y change?”
- “If event G happens, how likely is it that event H also happens?”

Notice that these are “statistical” statements, not necessarily biological statements. Translating biological ideas and questions into statistical statements and questions is one of the hardest parts of “doing statistics”.

1.3 How do we use statistics in biology?

Although there are hundreds, if not thousands, of statistical tests out there, most biologists only ever need a handful. It turns out that statistical hypotheses in biology tend to fall into just a few categories:

1. Testing for a **difference in mean or location** between groups.
2. Testing for a **continuous relationship** between a response variable (or multiple variables) and one or more predictor variables. Also referred to as **predicting** or **modeling** a continuous variable.
3. Testing for (lack of) **independence between categories** of observations.
4. Trying to **classify observations** based on predictor variables.

These categories are not mutually exclusive, and many biological questions can be phrased in more than one way. Understanding what kind of pattern you are looking for—that is, which of the 4 categories above your question falls into—can get you a long way toward understanding how to analyze your data.

1.4 *P*-values and null hypotheses

1.4.1 Null and alternative hypotheses

When scientists design experiments, they are testing one or more hypotheses, or proposed explanations for a phenomenon. Contrary to popular belief, the statistical analysis of experimental results usually does

not test a researcher's hypotheses directly. Instead, statistical analysis of experimental results focuses on comparing experimental results to the results that *might have occurred* if there was no effect of the factor under investigation.

In other words, traditional statistics sets up the following:

- **Null hypothesis or $H[0]$:** default assumption that some effect (or quantity, or relationship, etc.) is zero and does not affect the experimental results.
- **Alternative hypothesis or $H[a]$:** assumption that the null hypothesis is false. Usually this is the researcher's scientific hypothesis, or proposed explanation for a phenomenon.

In (slightly) more concrete terms, if researchers want to test the hypothesis that some factor X has an effect on measurement Y , then they set up an experiment comparing observations of Y across different values of X . They would then have the following set up:

- **Null hypothesis:** Factor X does not affect Y .
- **Alternative hypothesis:** Factor X does affect Y

Notice that in NHST the alternative hypothesis is the **logical negation** of the null hypothesis, *not* a statement about another variable (e.g., "Factor Z affects Y "). This is an easy mistake to make and one of the weaknesses of NHST: the word "hypothesis" in the phrase "null hypothesis" is not being used in the sense that scientists usually use it. This unfortunate choice of vocabulary is responsible for generations of confusion among researchers¹.

In order to test an "alternative hypothesis", what need is to *estimate* how likely the data would be if the null hypothesis were true. If a pattern seen in the data is highly unlikely, this is taken as evidence that the null hypothesis should be rejected in favor of the alternative hypothesis. This is not the same as "accepting" or "proving" the alternative hypothesis.

The null hypothesis is usually translated to a prediction like one of the following:

- The mean of Y is the same for all levels of X .
- The mean of Y does not differ between observations exposed to X and observations not exposed to X .
- Outcome Y is equally likely to occur when X occurs as it is when X does not occur.
- Rank order of Y values does not differ between levels of X .

All of these predictions have something in common: Y is **independent** of X . That's what the null hypothesis means. To test the predictions of the null hypothesis, we need a way to calculate what the data *might have looked like if the null hypothesis was correct*. Once we calculate that, we can compare the data to

¹Not to mention among generations of students taking statistics courses, myself included.

those calculations to see how “unlikely” the actual data were. Data that are sufficiently unlikely under the assumption that the null hypothesis is correct—a “significant difference”—are a kind of evidence that the null hypothesis is false. If this sounds convoluted, that’s because it is.

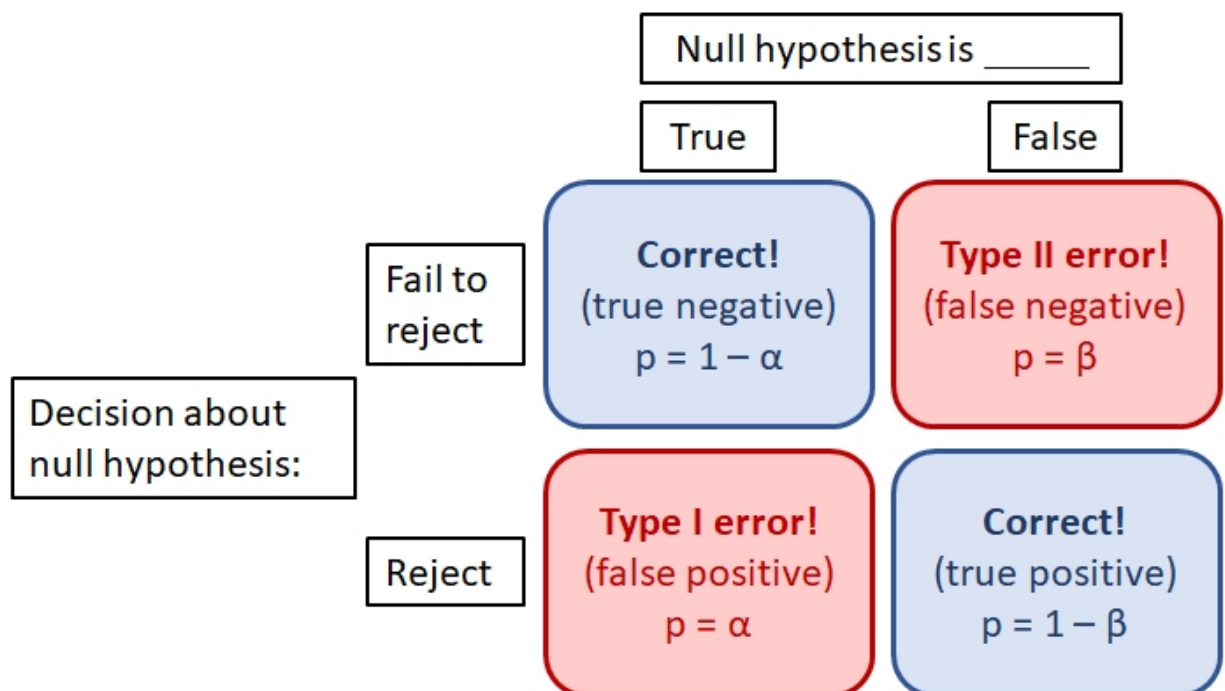
This kind of analysis is referred to as **Null Hypothesis Significance Testing (NHST)** and is one of the most important paradigms in modern science. Whether or not this is a good thing is a subject of fierce debate among scientists, statisticians, and philosophers of science. For better or worse, NHST is all but the *de facto* standard of inference in scientific publishing, so regardless of its merits in your particular case you will need to be able to use it. The purpose of this course module is to encourage you to use it correctly.

The *p*-value allows us to make binary distinctions, in a systematic way, between situations where a null hypothesis is true and no pattern is present, and situations where the null hypothesis is highly unlikely to be true and thus a pattern might be present. Any binary decision process has four possible outcomes:

1. **True positive:** declare a pattern is present when it is present.
2. **True negative:** declare a pattern is absent, when it is absent.
3. **False positive:** declare a pattern is present, when it is actually absent. Also called a **Type I error**.
4. **False negative:** declare a pattern is absent, when it is actually present. Also called a **Type II error**.

These outcomes can be summarized as shown below. Statisticians are particularly concerned with the probabilities of false positives (α) and false negatives (β). The false positive probability α is the standard against which experimental *p*-values are compared. This is largely because we humans are inordinately prone to finding patterns even when no patterns are present—i.e., committing type I errors.

Figure 1.2: Relationship between the null hypothesis and our decision about experimental results. Type I and II errors result when we reject a true null hypothesis and when we fail to reject a false null. The false discovery rate, or Type I error rate, defined by α is a very important value in our approach to science.



Why have we evolved this way? Finding patterns in noisy data has helped us discover virtually everything we know. So, pattern recognition must be a powerful cognitive tool. But there is another interpretation, that in our ancestral lifestyle life false positives were far less costly than false negatives. Imagine you are an Australopithecine or an early *Homo erectus* (or *H. sapiens*, for that matter), wandering through the savanna of what will someday be modern Ethiopia, looking for food. You hear a rustle in the tall grass a few meters ahead. Is that rustle just the wind? Or is it a dangerous animal that will kill and possibly eat you? Here are your choices and their consequences:

Figure 1.3: Illustration of the consequences of Type I and Type II errors, and why human minds may be so prone to Type I errors.

Your choice:	True source of the noise	
	Just the wind	Dangerous animal
Ignore the noise and keep foraging.	True negative No worries! You get to keep foraging a while longer.	False negative! Type II error! You are killed and eaten by a lion. And if you are very, very lucky...they'll do it in that order.
Run away from the possible lion!	False positive! Type I error! You spend some calories running away, and lose some foraging time, but at least you're alive.	True positive You run for your life and live to forage another day. Everyone at the camp totally believes your story about the lion.

Notice that of your choices, only the type II error has immediate evolutionary consequences. In terms of fitness, the difference between a type II error and a true positive (death vs. survival) is much greater than the difference between a true negative and a false positive (a bit more food vs. a mild inconvenience).

What does this have to do with statistics in biology? Our human minds have evolved to find patterns in noisy data, and we are so good at it that we frequently “find” patterns that aren’t really there. So, we use the scientific method and statistical analysis as tools to help mitigate our bias toward pattern-finding.

1.4.2 Another way to frame null hypotheses

When I teach statistics to biology majors, I try to avoid the terms “null hypothesis” and “alternative hypothesis” because most of my students find them confusing. Instead, I ask students to ask two questions about a hypothesis:

1. What should we observe if the hypothesis *is not* true?
2. What should we observe if the hypothesis *is* true?

In order to be *scientific* hypothesis, both of those questions have to be answerable. Otherwise, we have no way of distinguishing between a situation where a hypothesis is supported, or a situation where it is not,

because experimental results could *always* be due to random chance (even if the probability of that is very small). All we can really do is estimate how likely it is that random chance drove the outcome².

1.4.3 History and status of p -values

The p -value is a statistical statement about the likelihood of observing a pattern in data when the null hypothesis is true. They were first developed in the 1700s, and popularized by the statistician R.A. Fisher in the 1920s. In Fisher's formulation, the p -value was only meant as a "rule-of-thumb" or **heuristic** for determining whether the null hypothesis could be safely rejected. His original publications did not imply that rejecting the null hypothesis should automatically imply acceptance of the alternative hypothesis, or offer any justification for particular cut-offs or critical values of p . Those ideas came later.

Fisher suggested a critical p -value of 0.05 for convenience, but did not prescribe its use. In fact, he also suggested using other cut-offs, and even using the p -value itself as a measure of support for a hypothesis. The adoption of 0.05 as a threshold for significance proved popular, and has been taken as a kind of gospel for generations of researchers ever since.

In my opinion, the ubiquity of p -values on scientific research illustrates [Goodhart's Law](#): "When a measure becomes a target, it ceases to be a good measure". Because the cut-off value of $p < 0.05$ is so widely used and accepted as a measure of the "truth" of a hypothesis, many researchers will take steps to ensure that their experiments produce p -values < 0.05 . This can lead to the practices of p -hacking or data dredging. Or worse, setting up experiments to test meaningless null hypotheses or analyzing data selectively.

The attention and effort spent increasing the likelihood of a significant outcome is attention and effort that is not spent on formulating hypotheses, developing theory, or conducting experiments. Thus, intensive focus on p -values can be a real distraction from more important things.

These criticisms of p -values are neither new nor little known. Criticisms of the uncritical devotion to p -values can be found going back decades in the literature. In ecology, for example, debates about the place of NHST flare up occasionally, as new advances in statistical methods have offered opportunities for new inferential paradigms.

Recently, the American Statistical Association (ASA) published a number of papers criticizing the current use of p -values and offering alternatives. Wasserstein and Lazar (2016) provide a good starting point to the series and an overview of the major themes. Some of the proposed alternative or complementary methods have started to make their way into various scientific disciplines. Others have not. By and large, NHST remains the standard method of testing hypotheses with data and will likely remain so for a long time.

1.4.4 Where p -values come from

So where do p -values come from, anyway? Remember, p -values are an expression of how likely the observed data would be if the null hypothesis was true. Then calculation of a p -value involves assumptions about the size of the effect (i.e., difference between the results if the null is true vs. not true), the variability

²You may have noticed that questions 1 and 2 are really the null hypothesis and alternative hypothesis in disguise.

of the outcome (random or otherwise), and the distribution of different summary statistics of the results assuming that the null hypothesis is true. Let's consider the example of a t -test. A t -test is used to compare means of two groups. We'll talk more about them in Section 15.2. We will use R to simulate some data and explore a basic t -test.

```
set.seed(123)
n <- 10
mu1 <- 6
mu2 <- 8
sigma <- 3
x1 <- rnorm(n, mu1, sigma)
x2 <- rnorm(n, mu2, sigma)
t.test(x1, x2)
```

Welch Two Sample t-test

```
data:  x1 and x2
t = -1.796, df = 17.872, p-value = 0.08941
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -5.2131465  0.4091686
sample estimates:
mean of x mean of y
 6.223877  8.625866
```

The results show us that the means of groups x_1 and x_2 are not significantly different from each other ($t = -1.796$, 17.87 d.f., $p = 0.0894$). Because $p \geq 0.05$, we cannot reject the null hypothesis of no difference; i.e., the hypothesis that the means of x_1 and x_2 are the same. This is **not** the same as concluding that the means are the same. All we can say is that we cannot say the means are different.

But why? Where do the numbers in the output come from?

The t value is what's called a **test statistic**. This is a summary of the quantity of interest—difference in means—that depends on the quantity of interest *and* additional terms that describe how that summary might vary randomly. In the case of the t -test, the test statistic t is calculated as:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

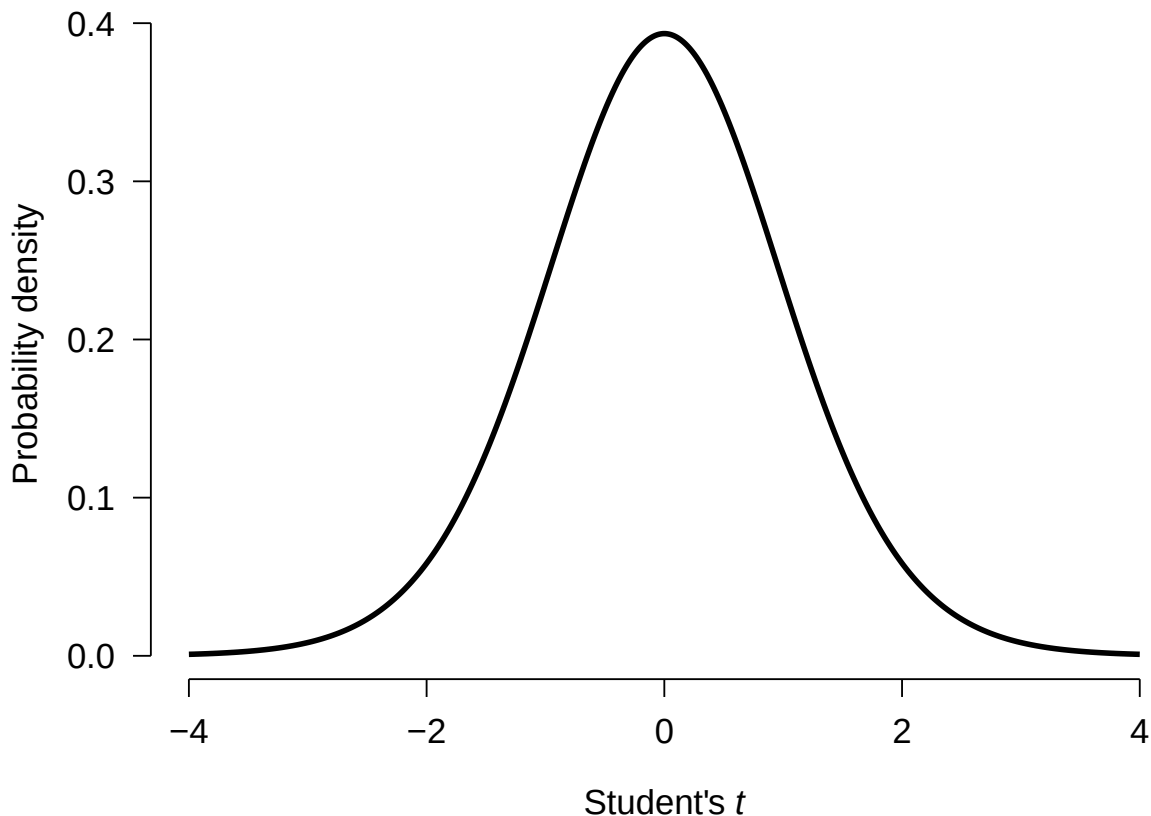
In this equation:

- $\bar{x}_1$ and $\bar{x}_2$ are the means of two sets of values x_1 and x_2 . A bar over a variable usually denotes a sample mean.

- s_1^2 and s_2^2 are the sample variances of x_1 and x_2 . The **variance** is a measure of how spread out values are away from the mean.
- n_1 and n_2 are the number of observations in x_1 and x_2 .

The numerator is the quantity we are interested in: the difference between the means in group 1 and group 2. The denominator scales the difference according to the sample size and the amount of variation in both groups. Thus, t accounts for the quantity of interest (the pattern), the variability in the pattern, and the number of observations of the pattern.

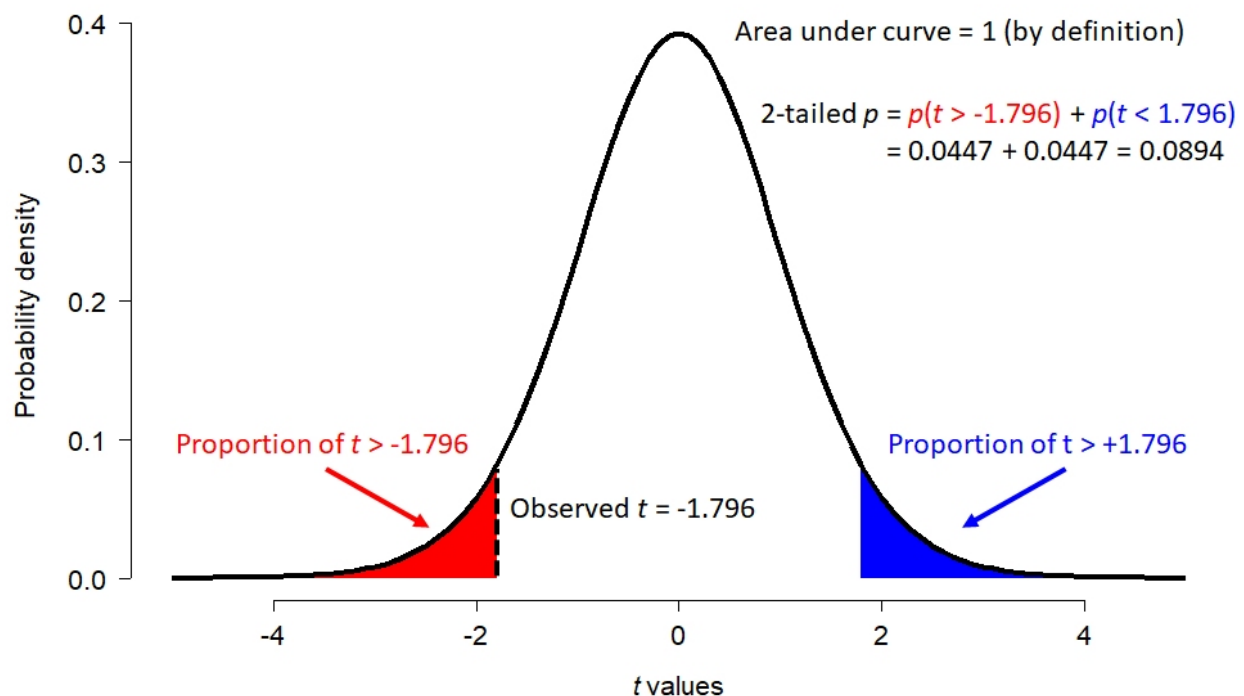
In the example above, the t statistic was -1.796. This was the difference in means (-2.402) scaled by the amount of random variation (s_1^2 and s_2^2) relative to the number of observations (n_1 and n_2). The “significance” of the test comes from the fact that t -statistics themselves follow a pattern called the **Student's t distribution**. The Student's t distribution describes how likely different values of t are given a certain sample size. This sample size used is not the total number of observations, but instead a scaled version of sample size called **degrees of freedom (d.f.)**. For the example above with 17.872 d.f., the t distribution looks like this:



If there was no difference between the means of x_1 and x_2 , then t statistics of randomly generated datasets should cluster around 0 with more extreme values being much less likely. Test statistics ≥ 4 or ≤ -4 almost never happen.

What about our t statistic, -1.796? The figure below shows how the total possible space of t statistics (given 17.872 d.f.) is partitioned. The total area under this curve, called a “probability density plot”, is equals 1. If you think about it, t must take on a value...so the sum of probabilities for all t is 1. The red area shows the probability of a t occurring randomly that is < -1.796 . That is, more extreme than the observed t . But “more extreme” can work the other way too: t could randomly be > 1.796 as well. This probability of this possibility is shown by the blue area. The total of the red and blue areas is the total probability of a t value more extreme than the observed t .

Figure 1.4: The areas under the t distribution probability density curve outside the critical values of t define the two-tailed p -value for the test.



But as we shall see, increasing the sample size can make the same difference appear significant:

```
set.seed(123)
n <- 1000
mu1 <- 6
mu2 <- 8
sigma <- 3
x1 <- rnorm(n, mu1, sigma)
x2 <- rnorm(n, mu2, sigma)
t.test(x1, x2)
```

Welch Two Sample t-test

```

data:  x1 and x2
t = -15.485, df = 1997.4, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.342319 -1.815705
sample estimates:
mean of x mean of y
 6.048384  8.127396

```

We can also find a significant p -value by reducing the amount of variation (σ , or σ). In this example, variance within each group is expressed as the residual standard deviation (SD). **Residual** variation is variation not explained by the model (i.e., not related to the difference in means between groups). The SD is the square root of the variance, and is how R expresses the variation in a normal distribution (`rnorm()`).

```

set.seed(123)
n <- 10
mu1 <- 6
mu2 <- 8
sigma <- 0.001
x1 <- rnorm(n, mu1, sigma)
x2 <- rnorm(n, mu2, sigma)
t.test(x1, x2)

```

Welch Two Sample t-test

```

data:  x1 and x2
t = -4486.7, df = 17.872, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.001071 -1.999197
sample estimates:
mean of x mean of y
 6.000075  8.000209

```

So, what the P -value in our t -test tells us depends on not just the actual difference, but how variable that difference is and how many times we measure it.

The table below summarizes the results of the three simulations we just ran:

True difference ($\bar{x}_1 - \bar{x}_2$)	Residual SD	Sample size	p -value	Significant?
-2	3	20	0.089	No

True difference ($\bar{x}_1 - \bar{x}_2$)	Residual SD	Sample size	p -value	Significant?
-2	3	2000	< 0.001	Yes
-2	0.001	20	< 0.001	Yes

1.4.5 What p -values mean and do not mean

In the simulations above, we know that the true difference was -2 because that is how we generated the numbers. But the conclusion of the experiments depended on experimental parameters other than the one of interest! For each of the three scenarios above, consider these questions:

- Should the researchers reject the null hypothesis, and conclude that there is a difference between x_1 and x_2 ?
- Should the researchers accept the null hypothesis, and conclude that there is no difference between x_1 and x_2 ?
- Should the researchers accept the alternative hypothesis, and conclude that there is a difference between x_1 and x_2 ?

The researchers should reject the null hypothesis only in scenarios 2 and 3. In scenario 1, they should not “accept” the null hypothesis. The correct action is **fail to reject** the null. Accepting the null hypothesis would imply some certainty that the difference was 0. But that is not what the t -test actually says. What the t -test really says is that a difference (measured as t) between -1.796 and 1.796 is likely to occur about 91% (i.e., $1-P$) of the time even if the true difference is 0. In other words, if we repeated the experiment, we would get a $|t| > 1.796$ over 9 times out of 10!

One of the most common misconceptions about p -values is that they represent the probability of the null hypothesis being true, or of the alternative hypothesis being false. **Neither of these interpretations is correct.** Similarly, many people think that 1 minus the p -value represents the probability that the alternative hypothesis is true. This is also incorrect.

Remember:

- The p -value is only a statistical summary, and says nothing about whether or not a hypothesis is true.
- The p -value is only a statement about how unlikely the result would be if there was no real effect.
- A p -value is a purely statistical calculation and says nothing about biological importance or the truth of your hypothesis.
- Interpreting p -values and other statistical outputs is the job of the scientist, not the computer.

If it feels like I’m belaboring these points, it’s because I am. The p -value and NHST in general are merely a set of heuristics to help researchers make binary distinctions like:

- Reject **or** fail to reject the null hypothesis.

- True difference is 0 **or** not 0.
- Difference is ≥ 3 **or** not < 3 .
- And so on.

These distinctions are binary because each condition is the logical negation of the other. But, *rejecting* one side of each dichotomy is *not the same as accepting* the other. By analogy, juries in criminal trials in the USA make one of two determinations: guilty or not guilty. A verdict of not guilty includes innocent as a possibility, but does not necessarily mean that the jury thinks the defendant is innocent. It simply means that the jury does not feel that the defendant has been *proven* guilty.

Likewise, the p -value might help distinguish “difference in means is different from 0 ($p < 0.05$)” vs. “difference in means is not different from 0 ($p \geq 0.05$)”. The latter choice includes a difference in means of 0 as a possibility, but does not actually demonstrate that the value is 0.

1.4.6 Do you need a p -value?

Now that we have a better handle on what a p -value really is, you might be tempted to ask, “Do I need a p -value?” Yes, you probably do. For better or for worse, p -values and NHST are firmly entrenched in most areas of science, including biology. Despite its shortcomings, the NHST paradigm does offer a powerful way to test and reject proposed explanations. This utility means that NHST and p -values are not going away any time soon. If you want an alternative to p -values and NHST, refer to [Chapter 29](#) or [Chapter 30](#).

Part II

Introduction to R

Why. Doesn't. It. Work?

Chapter 2

Introduction to R

This module is an introduction to the R program and language. We will begin with downloading and installing R, then explore the basics of running R commands and storing data, and finally answer some frequently asked questions (FAQ). Some of the topics in this module are covered in greater detail in [Chapter 3](#), [Chapter 4](#), and [Chapter 5](#). The current module is meant to get you up and running before moving on to more advanced material.

By the end of this module you should be able to:

- Explain what R is and what it is used for
- Explain the object-oriented programming paradigm used by R
- Download and install R and RStudio
- Write and execute commands in the R console
- Describe basic R data types and structures
- Manage R code as scripts (.r files)
- Manage and use R packages
- Identify sources of help for R programming

2.1 Getting started with R

2.1.1 Download and install R (and RStudio)

If you are on a university campus, many on-campus computers may already have R installed on them. This is the bare minimum you need to work with R. Many people also prefer to use another program for writing R code. The most popular is probably [RStudio](#), which is a more user-friendly interface for R. If you are on a PC or Mac, you will need to download and install both R and RStudio. If you are using a Linux machine, you may already have R because it is included in some distros. You can still install RStudio to make your life easier.

First, install the latest version of R from the [R-project website](#) (accessed 2021-12-20). Select a mirror (download server) from the list. Once you select a mirror, you will be taken to a page where you will select the version of R that corresponds to your operating system. Click on the appropriate link for your operating system and follow the instructions from there to install R. Like many programs you download from the internet, R will come with an installer that will largely take care of everything automatically. Just let the installer run and accept the default options.

Next, install any code editing or IDE you want.

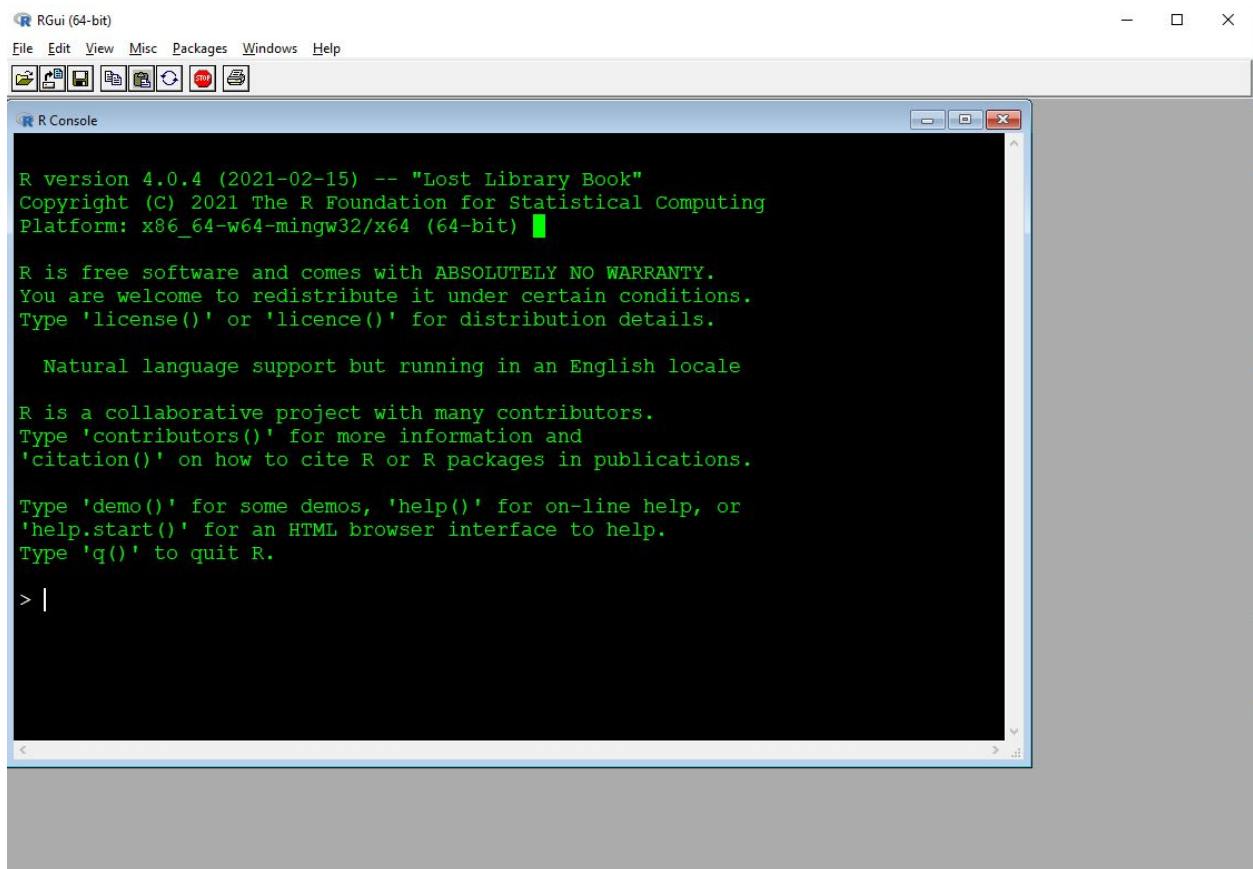
- RStudio can be installed from [here](#) (accessed 2025-08-06). Select the version that corresponds to your operating system. As you did when you installed R, just let the installer run and accept the default options.
- My personal favorite is Notepad++, which isn't an IDE but instead a very feature-rich code editor.
- Other popular choices include [Visual Studio Code](#) and [emacs](#).

2.1.2 Using R

2.1.2.1 Base R GUI

R has a very basic front end, or **graphical user interface (GUI)**. Most users find it easier to write their R code in another program. Some of these programs, such as RStudio, are **integrated development environments (IDE)**. Others are text editors with programming capabilities. We'll explore the default R GUI first and then meet some editor options. The basic R GUI is shown below. Your screen may look different than mine because I set some graphics options.

Figure 2.1: The base R GUI. Some like it (I do) but it's not for everyone.



The main window with R is the **R console**. This is the command line interface to R. Some important features of the console are:

1. Command prompt: The `>` symbol signifies the beginning of a command. You can enter commands here.
2. Continue prompt: `+` signifies the continuation of a previous command. If you have a `+` at the beginning of your line, you can get back to a new line using `ESCAPE`. Note that this will delete the currently incomplete line or command.
3. The `ENTER` key executes the current command. This could mean the current line, or multiple lines going back to the most recent `>`.
4. `[1]` at the beginning of results: because your result is an object (with length ≥ 1). The very first value ("element") in the result is element 1.
5. Up arrow key: cycles through previously entered commands. You can see many previous commands at once by using the command `history()`.

Other important features of the R GUI are found in the menu bar:

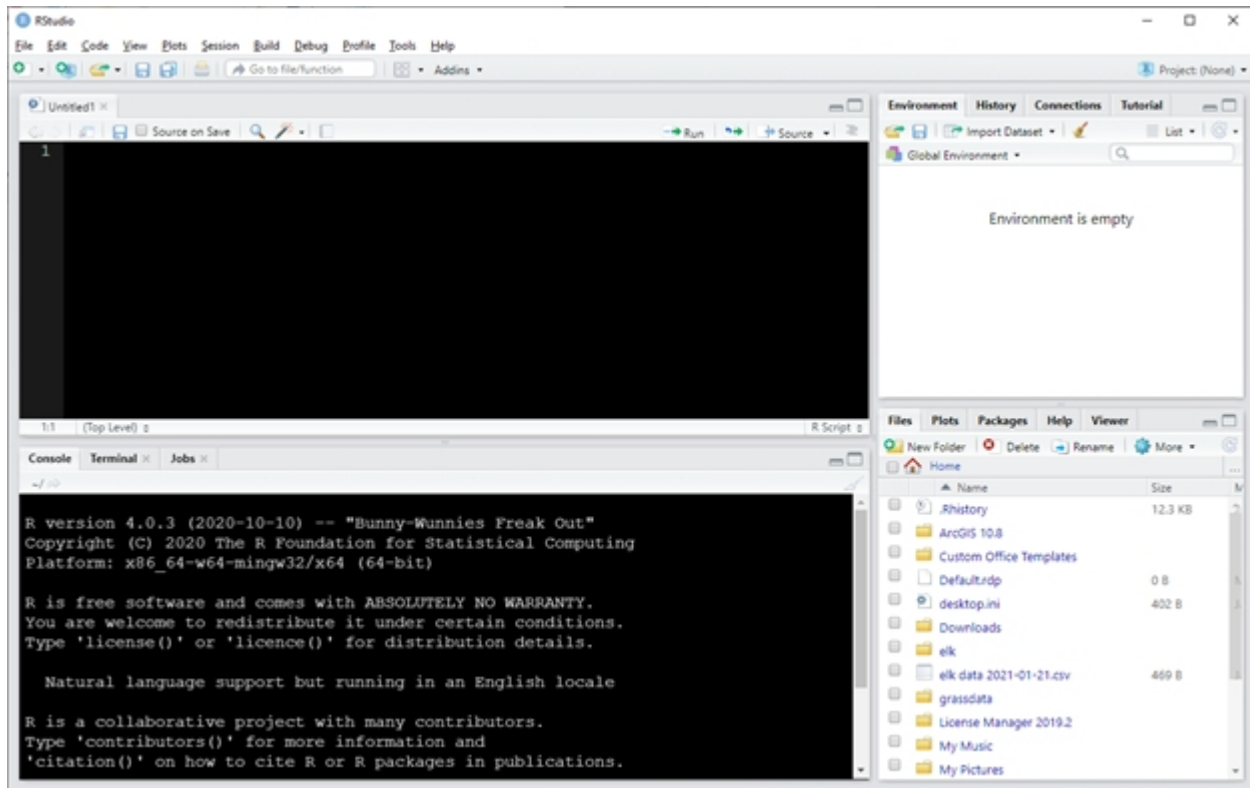
- **File**
 - **New script**: opens a new script window within the GUI.
 - Ability to open saved scripts or load previous workspaces
- **Edit**
 - **Copy, paste, etc.**: very useful!
 - **Data editor**: opens a rudimentary spreadsheet-like viewer for datasets that can be used to make edits. Not recommended for routine use.
 - **GUI preferences**: options to change colors, fonts, font sizes, etc., in the GUI. To make changes permanent you will need to save a new `.RConsole` file in your home directory.
- **View**
- **Misc**
 - **List objects**: lists all objects in workspace. Equivalent to command `ls()`.
 - **Remove all objects**: removes all objects from workspace. Equivalent to command `rm(list=ls(all=TRUE))`.
 - **List search path**: lists currently attached packages. Equivalent to command `search()`.
- **Packages**: functions to download and install packages.
- **Windows**: options to change how windows are displayed within GUI.
- **Help**: options and resources for getting help, including several free R manuals in PDF format.

2.1.2.2 Using R in RStudio

Although you can type R code directly into the console, or type in script windows within the base R GUI, it is usually easiest to use a separate program for writing code. The two best, in my experience, are RStudio and Notepad++ (see above). **RStudio** is an IDE designed to manage R projects, and has many useful features. Unlike some other options, RStudio has R built in so you can write code and execute it within the same program. RStudio can be used on Windows, Mac, and Linux operating systems. RStudio also has built-in support for utilities like RMarkdown and Quarto (which was used to build this website).

Once you have RStudio installed, open it and you will see a screen like this:

Figure 2.2: The default RStudio set up, with some graphics options set for dark mode.



Your RStudio might look different at first. You can change the color scheme by going to Tools - Global Options - Appearance and selecting a theme. Here I'm using the "Classic" theme with size 12 Courier New font and 110% zoom.

RStudio has an entire environment for working with R in one window. Each of the panels performs different functions. Starting at the bottom left and going clockwise, the panels are:

- **Console:** command line interface for R. You can type commands here and execute them by pressing ENTER.
- **Script:** a space where you can type code and later execute it by either clicking "Run" or pressing CTRL-ENTER (CMD-ENTER on a Mac). Usually people write code in the script window rather than typing it directly into the console.
- **Environment:** this window lists all of the data objects that you have loaded.
- **Files:** this window lists all of the files available in your R home directory. You can also navigate to other folders on your machine and select files to open.

The bottom right panel also has a tab called "**Plots**". This is where plots that you make can be seen. Sometimes you will need to resize this panel to see the plots clearly. All of the panels can be resized by clicking and dragging the borders between them.

2.2 Commands, objects, and output

The key to understanding R is to understand [object orientation](#). This means that R code is built around manipulating objects. In programming, an **object** is a construct or thing with well-defined attributes and behaviors. For example, a graphics program might have object classes like “polygon” and “circle”. Rather than redefine “polygon” every time the user needs to draw a polygon, the program creates a polygon object with attributes like “number of sides” and “side length”. The new object is an **instance** of the polygon class. The **class** of an object defines what attributes it has and how it interacts with other objects.

2.2.1 R commands—basics

In R, **everything** is an object. We will learn more about some of the most important R object types in the examples in Chapter 3. Objects are created in R by many functions.

Objects are stored to memory by **assignment**, using the left arrow `<-` (a “less than” followed by a hyphen). If an object already exists, then assignment is used to either modify the object, or replace it entirely. Here is an example:

```
a <- 3
```

This creates an object called `a`, which contains the value 3. You can then recall the value by typing `a` in the console and pressing Enter. You can also use the value stored in `a` in calculations or functions.

```
a
```

```
[1] 3
```

```
a * 2
```

```
[1] 6
```

```
1:a
```

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

Assigning a different value to `a` will overwrite it.

```
a
```

```
[1] 3
```

```
a <- 6
a
```

```
[1] 6
```

```
a^2
```

```
[1] 36
```

Notice that there is no “Undo” button. If you make a mistake, you need to rerun the code before your mistake to get back to where you were. Objects can be deleted by “removing them” with command `rm()`.

```
rm(a)
```

Besides assigning and working with values, R code contains comments: text that is for human readers, and is not interpreted by R. Comments are very important because they tell other people—or future you—what your code is doing and how it works. Some comments are just place markers (e.g., `# Analysis begins here`) to help you find important sections. Other comments might be more lengthy, holding explanations of why you chose to code a task the way that you did (e.g., `# condense with sapply()` (maintains order of subunits)). The latter kind of comment can be very useful to your future self.

In R, comments are made with the `#` symbol. Anything on a line after `#` is not read by R.

```
# here is a comment
b <- 5 # here is another comment
```

You can write your code directly in the console, or in a script window. If you use the script panel in RStudio (as you should), you can send commands to the console by pressing `Ctrl-Enter` (or `Cmd-Enter` on Mac). This will send and execute the entire command that the cursor is on. In the base R GUI script window, the equivalent is `Ctrl-R` (or `Cmd-R` on Mac), but this runs only the line where the cursor rests (not the entire command).

The R console can seem daunting to new users. The `>` symbol at the start of a line is the **command prompt**. Commands must begin here. If there is a `+` at the beginning of a line, then this means that there is an incomplete command. Press `Escape` to exit the incomplete command and get back to `>`.

When you type a command like `2+2`, the output will print to the console with a `[1]` at the start of the line. The `[1]` signifies that that line starts with the first element of the output. This is to help read outputs with lots of elements. Try the command `runif(100)` to see why.

```
2+2
```

```
[1] 4
```

```
rnorm(100)
```

```
[1] -1.600748770 -0.266624052 -0.637836355 0.993755884 2.598512518
[6] 2.058009415 0.582003185 -1.367702261 -1.316874217 -0.135171516
[11] -0.166490940 0.407665140 0.858663531 -0.323472514 0.988322630
[16] -1.277271909 1.686499999 2.215667155 -0.126782716 1.103479887
[21] 0.381607537 -0.544989714 -1.425527683 -0.515057798 0.196673515
[26] 0.329542049 2.169272010 0.145446814 -0.606075762 0.702418529
[31] -0.461643814 -0.429090193 0.116904022 0.436272406 -0.784784033
[36] 1.147497864 0.326705501 0.962396682 -2.386840343 0.520445695
[41] -0.513326419 0.607515614 1.932085554 2.387268675 -0.843865384
[46] 0.643713128 -0.944476207 -0.878426853 -0.355530996 1.002566525
[51] -0.025234993 0.758019117 1.765445578 1.033315126 -0.582001786
[56] 0.230750581 1.252366722 -0.250633391 -0.866463805 0.363897402
[61] -0.069641463 0.001439344 0.297169561 -1.103626198 -1.553981027
[66] -0.158958867 -0.986857363 1.387334985 -1.773075370 -1.455899167
[71] -1.612458825 0.621948653 0.878261109 -0.048466277 0.148077358
[76] -0.986319636 -0.253009168 1.330801369 0.761788954 -0.502555398
[81] 1.598215488 -1.356784839 -0.150874333 0.724587833 0.994102009
[86] -2.306341308 -0.681825906 1.586009695 0.804039268 -0.162220599
[91] -0.008568770 0.013186966 -0.274324348 -0.027369891 0.189356166
[96] -1.291232463 0.403872013 -0.908246452 0.417759657 -0.824080472
```

Commands can be split over multiple lines. When that happens, a + will be seen at the left side of the window instead of a >. A split command will be executed once a complete expression is input.

If you accidentally start a command and aren't sure how to finish it, you can exit and cancel the command with the ESCAPE key. R executes commands, not lines. You can have more than one command on a line separated by a semi-colon ;, but this generally frowned upon.

```
# command split across lines:
2 +
1
```

```
[1] 3
```

```
# 2 commands on a line (don't do this)
2 + 2; 3 + 3
```

```
[1] 4
```

```
[1] 6
```


Below is an example where the a command is broken across lines, with each line getting an explanatory comment. This is possible because R ignores whitespace (spaces, tabs, returns, etc.). Very complex functions or loops might benefit from this kind of commenting. Some programmers do not like this kind of comment—but you comment code the way that works for you.

```
rnorm(  
  10, # sample size  
  10, # mean  
  2   # sd  
      # note no comma after last argument!  
)#rnorm
```

```
[1]  9.310467  9.976210  8.021102  8.521752  9.977428  8.567394  
[7]  7.153071  9.137187 14.897596 10.804013
```

Notice in the above example that the closing parenthesis for `rnorm()` is marked with a comment. This is a good practice if the open and close parentheses are separated by many lines of code. Most IDEs will highlight matching parentheses, but marking them yourself can potentially save some headache.

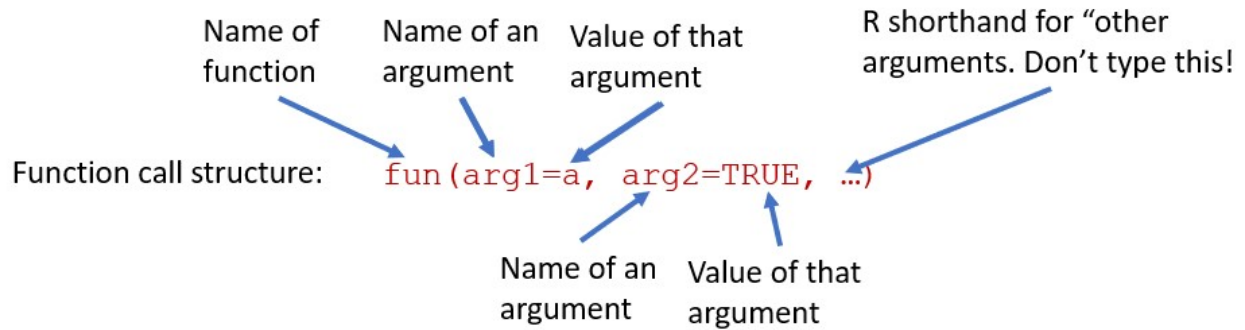
When writing comments or writing multi-line commands you should keep in mind that most code editors don't do line breaks very well. Or, they will wrap text displayed in the editor, but this might not display the same on every device. As a rule I try to keep lines of code to ≤ 80 characters, so I don't have to scroll over to see the end of a command. You can do this too, by breaking up commands and long comments into multiple lines. It also helps to indent additional lines by a set amount—most people prefer either 2 or 4 spaces rather than a tab stop.

2.2.2 Using R functions

R functions are objects that do things to other objects, or create objects, or modify objects... basically, functions **do** stuff while most other objects **store** stuff. Functions are called by name, and they have options or **arguments** contained within parentheses. It's good practice to have the parentheses touching the function name.

Argument values are set using single equals sign `=`. Argument values must be of the correct **type** (numeric, character, logical, etc.), must take on valid values (or be one of the available options), must be spelled and capitalized correctly, and may be named or not at the user's discretion. Most functions will print their results to the console upon completion. If you want to store the results of a function, you need to assign the results to an object using `<-`.

Figure 2.3: Basic structure of an R command and its arguments.



Example function call: `rnorm(n=20, mean=10, sd=2)`

Same function call with unnamed arguments: `rnorm(20, 10, 2)`

See below for some examples of function use using `rnorm()`, which draws random numbers from a normal distribution¹. Your numbers may be different than those below, because the function is *random*.

```
# arguments named
## draw 20 values with mean 10 and sd 2
rnorm(n=20, mean=10, sd=2)
```

```
[1] 10.927458 12.232584  8.908545 10.806007 13.614292 10.260973
[7] 12.176472 10.659363  6.012465  9.888794  8.596251 10.776733
[13] 10.391948  9.115058  9.396709 12.699707 10.028287  9.493144
[19] 13.345148  8.331533
```

```
## equivalent with unnamed arguments
rnorm(20, 10, 2)
```

```
[1] 11.277675 10.488317  9.087422  6.232064  6.709863 10.092402
[7] 13.254079  9.526039  9.224622 10.545279  7.998203 12.479325
[13] 10.905517  7.357862 12.189891 11.475283 11.527040 11.623623
[19]  9.673827  7.925945
```

¹The normal distribution comes up a lot in statistics. It is sometimes called the “bell curve”, but this is inappropriate because many distributions can be bell-shaped. It is also called the “Gaussian” distribution, but Gauss did not discover it. The normal distribution is defined by its mean μ and its standard deviation (SD) σ . In other words, its central tendency and spread, respectively.

```
## very different: 2 values from a normal
## distribution with mean=20 and sd=10
rnorm(2, 20, 10)
```

```
[1] 17.45336 33.53970
```

```
## when in doubt, name your arguments!
rnorm(sd=2, n=20, mean=10)
```

```
[1] 11.760061  7.494362 13.276145  9.814387 15.109935 11.155125
[7] 10.200439  8.436808 10.244158  6.132275 10.079200 10.971345
[13] 11.457399  8.147205  7.040388 13.902765  9.988014  9.660902
[19] 12.001616 10.544832
```

You can use the R help pages to see what the arguments are, the order in which R expects them, and their default values (if any). For `rnorm()`, the mean and sd have defaults 0 and 1, respectively.

```
?rnorm
```

To recap, argument **order** is important when using functions. You can supply arguments to a function without naming them if you put them in the proper order. See the help files to see what the order is. *When in doubt, name your arguments.*

Functions expect arguments in a particular format. If a function is not working, you might have an argument formatted incorrectly. Sometimes it's helpful to look at each argument separately in the R console, outside of the function. To do this, copy/paste each argument to the R console separately and execute by hitting ENTER.

Some function arguments have default values, so you don't have to supply them. For example, `runif()`, which draws random numbers from the uniform distribution, defaults to the interval between 0 and 1. So, the two `runif()` commands below are equivalent because 0 and 1 are the defaults for the limits of the interval. The `set.seed()` command before each `runif()` command is to set the random number seed, ensuring that the results are identical. If you run the `runif()` commands without resetting the random number seed, the commands will return different results because of the random nature of the function.

```
set.seed(123)
runif(10, 0, 1)
## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673 0.0455565
## [7] 0.5281055 0.8924190 0.5514350 0.4566147

set.seed(123)
runif(10)
## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673 0.0455565
## [7] 0.5281055 0.8924190 0.5514350 0.4566147
```

2.2.3 The R workspace

When you use R, all objects and data that you use exist in the R **workspace**. Objects in the workspace can be called from the console. You can think of the workspace like a kind of sandbox or workbench.

Most of the time, when you start R this creates a new workspace with no objects. If you open a previously saved workspace, you will see the message `[Previously saved workspace restored]`. If you saved your R workspace without a unique name, it can be hard to figure out what workspace you are in. You can use the `ls()` command to see what is in the workspace, which might help you figure out what workspace has been loaded. If there are no objects in the workspace, the result of `ls()` will be `character(0)`. This output means that the `ls()` command returned a vector of text values (character) with 0 elements. In other words, the workspace contains no objects.

As you work, your workspace will fill up with objects. This isn't a big deal until you start to either have too many names to keep track of, or start to use too much memory. The first problem can be dealt with by commenting your code. The second problem can be solved by deleting unneeded objects to save space. You can check to see how much memory your R objects are using with the following command:

```
# not run, but try on your machine
sort(sapply(mget(ls()),object.size), decreasing=TRUE)
```

This command will print the sizes of objects in memory in the current workspace, in descending order. The default units are bytes. If you want the results in Mb, divide the output by 1.024e6.

```
# not run, but try on your machine
a <- sort(sapply(mget(ls()),object.size), decreasing=TRUE)
a/1.024e6
```

R objects can be deleted or removed from the workspace with the `rm()` function. You can remove objects one at a time or by supplying a set of names. Note that the set of names is supplied to an argument called `list`, which is confusing because `list` is also the name of a special type of object in R.

```
# not run, but try on your machine
x <- 1
y <- 2
z <- 3

# shows x, y, and z in workspace
ls()

# remove x
rm(x)
ls() # x is gone
```

```
# remove y and z
rm(list=c("y", "z"))
ls() # x, y, and z are all gone
```

2.3 R syntax

2.3.1 Object size and structure

As you've already seen, the most important R function is **assignment**. When you assign a value to a variable, you can then use that variable in place of the value. Assigning a value to variable name will automatically create a new object in the workspace with that name. If an object with that name already exists, assigning a new value to that name will overwrite the old object.

You should always use the left arrow `<-` as the assignment operator. The equals sign `=` can be used for assignment, but you should not do so because `=` is easily confused with the equality test operator `==`. Also, `=` is a mathematical symbol, but `<-` is unambiguously an R operation. You can also use `->` to assign to the right, but doing so usually makes your code harder to read rather than easier. Always use `<-` to assign to the left.

```
aa <- 4
bb <- 1:10
```

Don't do this:

```
cc = 5
```

Don't do this either.

```
5 -> cc
```

And definitely don't do this:

```
a <- 5 -> b
```

You can chain multiple assignments together in a single command. This is useful if you need to quickly create several objects with the same structure:

```
list3 <- list2 <- list1 <- vector("list", 5)
```

Assignment can also be used to **change or overwrite** parts of pre-existing objects. Usually this involves the bracket notation, which we will explore more in Chapter 5. In the example below, the third element of object `a`, `a[3]`, is replaced with the number 42.

```

a <- 1:5
a
## [1] 1 2 3 4 5

a[3] <- 42
a
## [1] 1 2 42 4 5

```

One little-known feature of R assignment is that it can **resize** objects. For example, if you assign a value to the *n*-th element of a vector that has fewer than *n* elements, R will increase the length of the object to the size needed. It should be noted that this can be done accidentally as well as intentionally, so you need to be very careful if you resize objects using assignment. Personally, I don't do this because it is often clearer to change the size of objects in a separate command.

```

# resize by assignment (not recommended)
a <- 1:3
a[5] <- 5
a
## [1] 1 2 3 NA 5

# resize in separate command (preferred)
a <- 1:3
length(a) <- 5
a
## [1] 1 2 3 NA NA

# resize by concatenation (preferred)
a <- 1:3
a <- c(a, 4:5)
a
## [1] 1 2 3 4 5

# alternatively:
a <- 1:3
b <- 4:5
a <- c(a, b)
a
## [1] 1 2 3 4 5

```

2.3.2 Operators

Most math operators in R are similar to those in other languages or programs. For math expressions, the standard order of operations (PEMDAS) applies, but you can use parentheses if you want to be sure. The integer sequence operator `:` comes first, and comparison tests (`==`, `<`, `<=`, `>`, and `>=`) come last. The examples below show how many common R operators function, taking advantage of the fact that you can use an object's name in place of its value in most situations.

```
# make some values to work with
aa <- 5
bb <- 8

# multiply
aa*3
## [1] 15

# add
bb+3
## [1] 11

# square root
sqrt(aa)
## [1] 2.236068

# natural logarithm (base e)
log(aa)
## [1] 1.609438

# logarithm (arbitrary base)
log(aa, base=3)
## [1] 1.464974

log(aa, pi)
## [1] 1.405954

# common logarithm (base 10)
log10(aa)
## [1] 0.69897

# exponentiation with base e (antilog)
exp(aa)
## [1] 148.4132
```

```

# Vectorized (element-wise) multiplication
bb*2:5
## [1] 16 24 32 40

# Exponentiation
bb^2
## [1] 64

# Alternate symbol for exponentiation (rare)
bb**2
## [1] 64

# Remainder (modulus)
## 13 / 5 == 2 remainder 3
13 %% 5
## [1] 3

# Quotient without remainder
13 %/% 5
## [1] 2

# Order of operations
aa+2*3
## [1] 11

# Order of operations
aa+(2*3)
## [1] 11

# Order of operations
(aa+2)*3
## [1] 21

# Sequence of integers
1:10
## [1] 1 2 3 4 5 6 7 8 9 10

# Note that ":" comes first!
1:10*10
## [1] 10 20 30 40 50 60 70 80 90 100

```



```
# Comparisons are last
2*2 > 1:5*2
## [1] TRUE FALSE FALSE FALSE FALSE
```

Some of these examples illustrate a key concept in R programming: **vectorization**. Many R functions can operate on sets of numbers, or **vectors**, just as well as on single values or **scalars**. The function will operate on every element of the vector in order. Most R functions are vectorized to some extent. This is very handy because many operations in statistics deal with vectors of values.

Vectorization can cause headaches if the vectors are not of the same length. If the length of one vector is a multiple of the length of the other, then the shorter vector gets recycled. This is sometimes called the “recycling rule”. The recycling rule can be very convenient if you know what you’re doing and are paying attention. If you don’t pay attention, or aren’t aware that recycling is happening, it can create a headache because values are recycled without telling you.

If the length of the longer vector is not a multiple of the length of the shorter vector, the shorter one will be recycled a non-integer number of times and R will return a warning (longer object length is not a multiple of shorter object length).

```
# lengths compatible:
x <- 1:2
y <- 3:6
x+y
## [1] 4 6 6 8

# lengths not compatible:
a <- 1:3
b <- 4:5
a + b
## [1] 5 7 7
```

2.4 R packages

Packages are modules that can be added to R to extend its capabilities. Most packages are written by R users and researchers who need functions that are not provided with the base R installation. These add-ons are one of the greatest strengths of R compared to other software packages. Commercial and proprietary statistics programs (like SAS) are upgraded only by a single group of developers, if at all. In contrast, new functions and capabilities are added to R by the user community every day.

Some packages are widely used and cited, others less so. Do your homework before depending on a package. Is the package included or recommended with the base R installation? Is the package associated with any peer-reviewed publications? Does the package author appear to have the credentials that would

qualify them to produce a package? Do other packages depend on this package? All of these are good signs that a package is reliable.

Most packages are stored on The Comprehensive R Archive Network, or CRAN. Other repositories include R-Forge, BioConductor, and GitHub. Packages can also be downloaded and installed from local storage.

R-Forge and BioConductor can usually be accessed from the R command line just like CRAN. Installing packages from GitHub may require a different procedure. Most R packages are written in the R language itself, so even if you can't find a way to install a package you can usually just copy and run its source code.

2.4.1 Your R library

When you install R packages, the files are downloaded and stored on your machine for R to access when needed. R will probably create a folder in your R home directory for this purpose. The first time you install a package in R you will be asked if R can create a personal library in which to store the package. Package installations are specific to the version of R that you used for the installation, and specific to your operating system login. This means that if someone else logs in to your machine and wants to use a package, they will have to install the package for themselves. This also means that if you update your R installation to a new version, you may need to install packages again because some packages are version-specific.

2.4.2 Installing R packages

2.4.2.1 Installing R packages using the GUI

In the R GUI, go to Packages – Install package(s).... You will be prompted to select a CRAN mirror (i.e., server from which to download). Pick any mirror you like. Then, scroll to and select your package. If the install fails, try a different mirror by going to Packages – Set CRAN mirror..., and then retrying the installation.

2.4.2.2 Installing packages in RStudio

You can install a package in RStudio by going to Tools–Install Packages... and searching for the name of the package you want.

2.4.2.3 Installing packages using R code

In the R console, you can install packages using function `install.packages()` as shown below. You will still be asked to select your CRAN mirror (most packages are hosted on CRAN).

```
install.packages("vegan")
```

2.4.3 Working with packages in R

You can access the packages you have installed by loading them. In most situations you must load a package before you use it. The usual way is to use the `library()` function. Once a package is loaded, you can use its functions and datasets. The function `require()` does the same thing as `library()`, but it is designed for use inside other functions. Most of the time you should use `library()`. The command below will load the package `vegan` so its functions and datasets become available:

```
library(vegan)
```

Notice that the package name does not have to be in quotation marks within the `library()` function.

You can see what packages are currently attached with `search()` or `sessionInfo()`. Packages can be unloaded detaching them from the workspace: `detach(package:x)`, where `x` is the name of the package you want to detach. In the example below, the first command loads the `MASS` package. Then, we can see that is loaded in the workspace with `search()`. The next command unloads `MASS`, then verifies that `MASS` was unloaded from the workspace with another `search()` command.

```
search()
```

```
[1] ".GlobalEnv"          "package:vegan"       "package:permute"
[4] "package:stats"       "package:graphics"   "package:grDevices"
[7] "package:utils"       "package:datasets"   "package:methods"
[10] "Autoloads"          "package:base"
```

```
library(MASS)
search()
```

```
[1] ".GlobalEnv"          "package:MASS"        "package:vegan"
[4] "package:permute"     "package:stats"       "package:graphics"
[7] "package:grDevices"  "package:utils"       "package:datasets"
[10] "package:methods"    "Autoloads"           "package:base"
```

```
detach(package:MASS)
search()
```

```
[1] ".GlobalEnv"          "package:vegan"       "package:permute"
[4] "package:stats"       "package:graphics"   "package:grDevices"
[7] "package:utils"       "package:datasets"   "package:methods"
[10] "Autoloads"          "package:base"
```

There is rarely any need to unload a package from the workspace unless there are naming conflicts. The most common situation is when two attached packages have functions with the same name. In this situation,

R will use the *most recently loaded* package's function. This phenomenon is called masking. When there is a name conflict, you can use the double colon operator `::` to specify which package's function you want to use.

Using `::` can also access a function without loading the entire package into the workspace. If both packages are already loaded into your workspace, then `::` will simply select which package's function to use. Note that a package must be *installed* on your machine before you can use its functions.

The example below shows how `::` can be used to access the `fitdistr()` function from package MASS without loading the package

```
# simple example
base::log(10)
```

```
[1] 2.302585
```

```
log(10)
```

```
[1] 2.302585
```

```
# use a function without loading its package
MASS::fitdistr(rnorm(100, 50, 10), "normal")
```

```
      mean      sd
50.6073364  9.0431206
( 0.9043121) ( 0.6394452)
```

```
# verify that MASS is not attached:
search()
```

```
[1] ".GlobalEnv"      "package:vegan"    "package:permute"
[4] "package:stats"    "package:graphics" "package:grDevices"
[7] "package:utils"    "package:datasets" "package:methods"
[10] "Autoloads"        "package:base"
```

```
sessionInfo()
```

```
R version 4.4.3 (2025-02-28 ucrt)
Platform: x86_64-w64-mingw32/x64
Running under: Windows 11 x64 (build 26200)
```

```
Matrix products: default
```

```

locale:
[1] LC_COLLATE=English_United States.utf8
[2] LC_CTYPE=English_United States.utf8
[3] LC_MONETARY=English_United States.utf8
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.utf8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods
[7] base

other attached packages:
[1] vegan_2.7-1    permute_0.9-8

loaded via a namespace (and not attached):
 [1] digest_0.6.37      fastmap_1.2.0      Matrix_1.7-2
 [4] mgcv_1.9-1         xfun_0.53          lattice_0.22-6
 [7] splines_4.4.3      parallel_4.4.3     knitr_1.50
[10] htmltools_0.5.8.1  rmarkdown_2.29     cli_3.6.3
[13] grid_4.4.3         compiler_4.4.3     rstudioapi_0.17.1
[16] tools_4.4.3        cluster_2.1.8      nlme_3.1-167
[19] evaluate_1.0.5     yaml_2.3.10        rlang_1.1.7
[22] jsonlite_2.0.0     MASS_7.3-64

```

2.4.3.1 Package dependencies

Many R packages are built taking advantage of other R packages. The packages that a package depends on are called its dependencies or depends. Some packages have no dependencies (other than R itself), while others have many. Something to think about when deciding whether to use a package are the number and nature of its dependencies. If you use a package that depends on many other packages, then you are relying on the authors of those packages to maintain and update their code to work with new versions of R and new versions of packages. And, relying upon the authors of all of those packages' dependencies. And so on.

You can see the dependencies of a package by checking out its page on CRAN. For example, the `drc` package (Ritz *et al.* 2015) is [here](#). You can see the current version, what packages it depends on ("Depends"), and what packages depend on it ("Reverse depends").

The point here is that you don't want your code to break because one of the packages you use hasn't been updated. This can get very frustrating if, for example, several packages you use all depend on each other, or even different versions of each other! Most R package authors and/or maintainers are pretty good about keeping their packages up to date, but some are not. If you really need a function from a package that is no longer maintained, or incompatible with your current version of R, you have three options:

1. Download and use the older version of R that is compatible with the package. This is not recommended because you will lose the benefit of recent bug fixes and modifications to R. This option is also not recommended because by rolling back to an older version of R, you will likely need to roll back to older versions of other packages.
2. Find another, current package that does what you need to do. This can be a quite a headache if you have a large codebase or a very specialized function.
3. Access the source code to the package you need, and "borrow" the functions or code you need.
4. Sometimes, if you only need 1 or 2 functions from a package, or you need to modify a function from a package to suit your needs. In these cases, you can "borrow" directly from the package source code. For most packages, the CRAN page will have a link to an online code repository (usually Github) and/or the package source as a "tarball" (".tar.gz" file). A tarball is a compressed version of the package source code that may or may not need to be compiled using special functions in R.

Because most R packages are written in R itself, you can usually find and adapt the code you need. This is almost always permitted by the license of the package because most packages are released under some version of the GNU Public License (GPL). If you do borrow code from a package, it is considered good form to attribute the package authors in some way (e.g., "we adapted code from the package "foo" version 1.0 (Nobody 2001)").

2.4.3.2 Citing packages

Just like any software, you need to cite a package if you use it. This is both to give appropriate credit to the package authors, and to allow others to understand what you did. The appropriate citation for a package can be accessed from the console with the `citation()` command.

Citation for R itself:

```
citation()
```

To cite R in publications use:

```
R Core Team (2025). _R: A Language and Environment for  
Statistical Computing_. R Foundation for Statistical  
Computing, Vienna, Austria. <https://www.R-project.org/>.
```

A BibTeX entry for LaTeX users is

```
@Manual{,  
  title = {R: A Language and Environment for Statistical Computing},  
  author = {{R Core Team}},  
  organization = {R Foundation for Statistical Computing},  
  address = {Vienna, Austria},  
  year = {2025},  
  url = {https://www.R-project.org/},  
}
```

We have invested a lot of time and effort in creating R, please cite it when using it for data analysis. See also 'citation("pkgname")' for citing R packages.

Citation for a package (note the quotation marks):

```
citation("MASS")
```

To cite the MASS package in publications use:

Venables, W. N. & Ripley, B. D. (2002) Modern Applied Statistics with S. Fourth Edition. Springer, New York. ISBN 0-387-95457-0

A BibTeX entry for LaTeX users is

```
@Book{,  
  title = {Modern Applied Statistics with S},  
  author = {W. N. Venables and B. D. Ripley},  
  publisher = {Springer},  
  edition = {Fourth},  
  address = {New York},  
  year = {2002},  
  note = {ISBN 0-387-95457-0},  
  url = {https://www.stats.ox.ac.uk/pub/MASS4/},  
}
```

2.5 Frequently asked questions about R

2.5.1 What is R

R is both a computer program that does statistics, and the programming language for using that program. It was developed in the late 1990s and became popular among biologists by the mid-2000s. R offers several key advantages over other statistical programs:

1. It is completely free to use. All you need is a computer with the minimum hardware requirements, and a way to download it.
2. It is open source. Unlike commercial software like SAS or JMP, anyone can inspect the [source code of R](#) and see exactly how it does what it does. One can also modify R to suit their purposes, although there are better ways to add functionality.
3. R has an enormous user base, including many academic and professional statisticians and data scientists. This means that (a) new statistical methods are often implemented in R before in any other program; and (b) there are many community-driven sources of help online, such as [Stack Overflow](#).
4. R is designed explicitly for statistical analysis and data manipulation, so the learning curve is usually gentle for common use cases. In terms of difficulty, R is similar to Microsoft Excel formulas and to [SciPy](#) or [pandas](#) (which are extensions of Python, an open source language often recommended for beginners).
5. New methods and functions are constantly being added as add-on packages for R that can be downloaded for free and integrate (mostly) seamlessly with other packages. See the [CRAN task views](#) for a list of package collections by application area.

2.5.2 Why do we have to write code?

Why don't we use a point and click program? Why do we have to write code? There are 3 main reasons:

1. Code is just a way to express instructions *clearly and unambiguously* to the computer. Human language is often imprecise. Computers do exactly what you tell them to do, so you must tell them exactly what you want them to do. A programming language is a structured and orderly way of doing that.
2. Code, once written, can be *easily repeated* by running it again. Repeating a series of "point and click" instructions can be tedious, time-consuming, and highly error prone.
3. Writing the code to execute an analysis has the side effect of *documenting exactly what was done*. This can be extremely useful if there is ever a question about whether an analysis was done correctly, or how the data were handled. In R, we save our code in files called scripts that have a .R file extension.

As you learn, it's better to think of R as a language, rather than as a code. Don't think, "How do I *do* X in R?" Instead, think, "How do I *say* X in R?" With this perspective, it becomes clear that "writing code" is really an exercise in "expressing instructions clearly and efficiently".

2.5.3 Advantages and disadvantages

For all of its advantages, R has some drawbacks. Chief among them:

- R is free. Unlike some paid and proprietary software, there is no official help line to call if you have issues.
- Along the same lines, R is distributed with absolutely no warranty (it says so every time you open R). The accuracy of your analysis depends on the competence of the (unpaid) contributors and the user.
- R often requires more coding than SAS or other tools to get a similar amount of output or perform a similar amount of work. The `tidyverse` ecosystem of packages addresses some of these issues, but not without some costs (see `#sec-r-faq-tidy`).
- R is slower than SAS and other tools at handling large datasets because it holds all data and objects in memory (RAM) at once. This limits the performance of your machine if you are doing other tasks.
- R is single-threaded by default. This means that R does not handle parallel computation very easily, which could potentially speed up many tasks.
- R has a difficult learning curve because it is so programming-focused.
- Error messages that R gives are often vague and infuriating.

2.5.4 Base R vs. tidyverse

A colleague of mine who moved from using mostly SAS to using mostly R says that the `tidyverse` makes R feel more like SAS. The difference he was referring to is the way that SAS functions (“PROC”s, or procedures) tend to automate or abstract away a lot of the low-level functionality and options associated with their methods. Those functions and options are there, but they can be inconvenient to access. The equivalent base R functions, on the other hand, require the user to specify options and manipulate output to a far greater extent. Or, depending on your perspective, they *allow* the user to specify options and manipulate output. Some people like that, and some people don’t. The amount of interaction and coding in R required to get outputs that other programs like SAS produce automatically is, in my experience, one of the biggest stumbling blocks for new R users.

Because of the amount of work required to get base R to do anything, the degree of automation and abstraction available in the `tidyverse` packages (Wickham *et al.* 2019) is very attractive to many R users: they can focus on high-level decisions without having to get into every picayune detail of routine workflows and standard analyses. `Tidyverse` packages allow for very streamlined and (once you learn it) easy to understand code. These advantages come at a cost: because the functions abstract the finer details away, those details are harder to see. This can be problematic if you need to work more directly with function inputs and outputs, or manipulate options or data in non-standard ways. For example, if a journal editor is adamant about some minute detail of formatting on a figure, it might be much easier to make the changes in base graphics rather than `Tidyverse` graphics. The abstraction of the low-level workings of an analysis or program can save a lot

of time and headache, but can also make it harder to track down or even to notice errors (although to be fair, systematic approaches to data manipulation such as in package `dplyr` can also prevent many errors as well).

Recently Norm Matloff of the University of California, Davis posted an [essay about the advantages of base R compared to the tidyverse](#). It argues very well some ideas that I've thought for years, but could never quite articulate. Matloff is the author of several books on R, former editor of the R journal, and a longtime teacher of statistics and computer science. He argues that new R users should focus on learning base R rather than tidyverse. His main points can be summarized as follows:

- Tidyverse packages and code tend to utilize functional programming, rather than object-oriented programming. Functional programming is more abstract than object-oriented programming, which makes it harder to new users with little to no programming background.
- Tidyverse functions eschew use of the `$` operator, square brackets `[]`, and base R data structures in general². This means that people who learn only tidyverse are not equipped to take advantage of some of the most fundamental and powerful features of R—including many statistical functions!
- Tidyverse code avoids loops, a fundamental programming construct. Loops are, for beginning programmers, easy to “step through” and debug. In tidyverse, operations that would be in loops in base R are “hidden” inside functions. This makes it very hard to track down errors. There are base R functions that can be used to avoid loops as well (e.g., `lapply()` and its cousins), and they are equally hard for beginners to use.
- Tidyverse is intentionally designed to equip users with a small set of R tools. This means that tidyverse-only users are less equipped to deal with new situations.

In my own workflow, and in this course, I tend to use base R methods instead of tidyverse for reasons above, as well as a few other reasons that are entirely personal and subjective:

- I learned R before tidyverse became a thing. This isn't an advantage or disadvantage to either paradigm; it just is. For me, the time and effort costs of switching to tidyverse would likely outweigh the benefits.
- I prefer the syntax of base R to the “grammar” of tidyverse. In particular, tidyverse makes extensive use of the pipe operator `%>%`, which tends to make for obfuscated (to me) code. In base R, multi-step operations are laid out on multiple lines, with intermediate objects being available in the workspace. This is not the case in tidyverse.
- I prefer to write code with as few dependencies as possible. This makes it less likely that some random update will break my code.

The other two reasons are more functional and less subjective:

²In my humble opinion, this aspect of tidyverse is just insane.

- Debugging and error correction is easier in base R because of the lack of abstraction. If every step is coded manually and out in the open, errors are easier to find, isolate, and fix.
- Debugging and fixing tidyverse code often requires knowing base R, but the reverse is never true.

The takehome message is that while both base R and tidyverse offer powerful tools for working with and analyzing data, they represent two overlapping but different programming paradigms. Which framework you use depends on what level of abstraction vs. explicitness you are comfortable with in your own code. Base R makes it easier to work flexibly with the different kinds of inputs and outputs, but usually at the cost of a steeper learning curve. tidyverse makes routine tasks more streamlined with a unified syntax and grammar for data manipulation, but at the cost of flexibility. My advice is to learn base R first, then move on to tidyverse if and only if it has a specific tool you need that is not available in base R.

2.5.5 Where can I get R help?

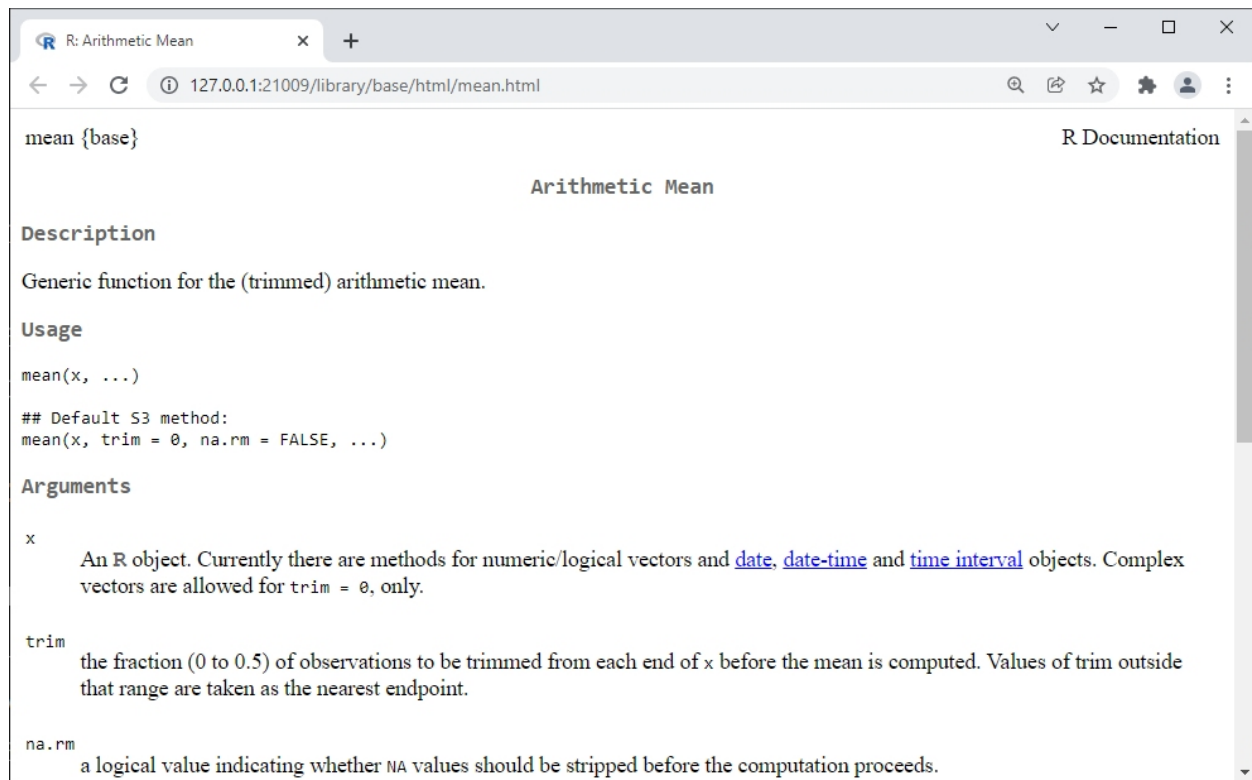
The most important message of this course is simple: ***Don't panic!*** Lots of people have trouble with R. The learning curve is steep, and the error messages are inscrutable. But, there are a lot of resources available to help you.

2.5.5.1 R documentation (help) files

Documentation for each function, or help files, come included with the base R installation. You can access them from within R by using the `?` command. `?functionname` will open the documentation file for function `functionname`. `?` will search all currently attached packages. The command below opens the page for the `mean` function. Notice that this file is stored on your local machine, so no internet access is required.

```
?mean
```

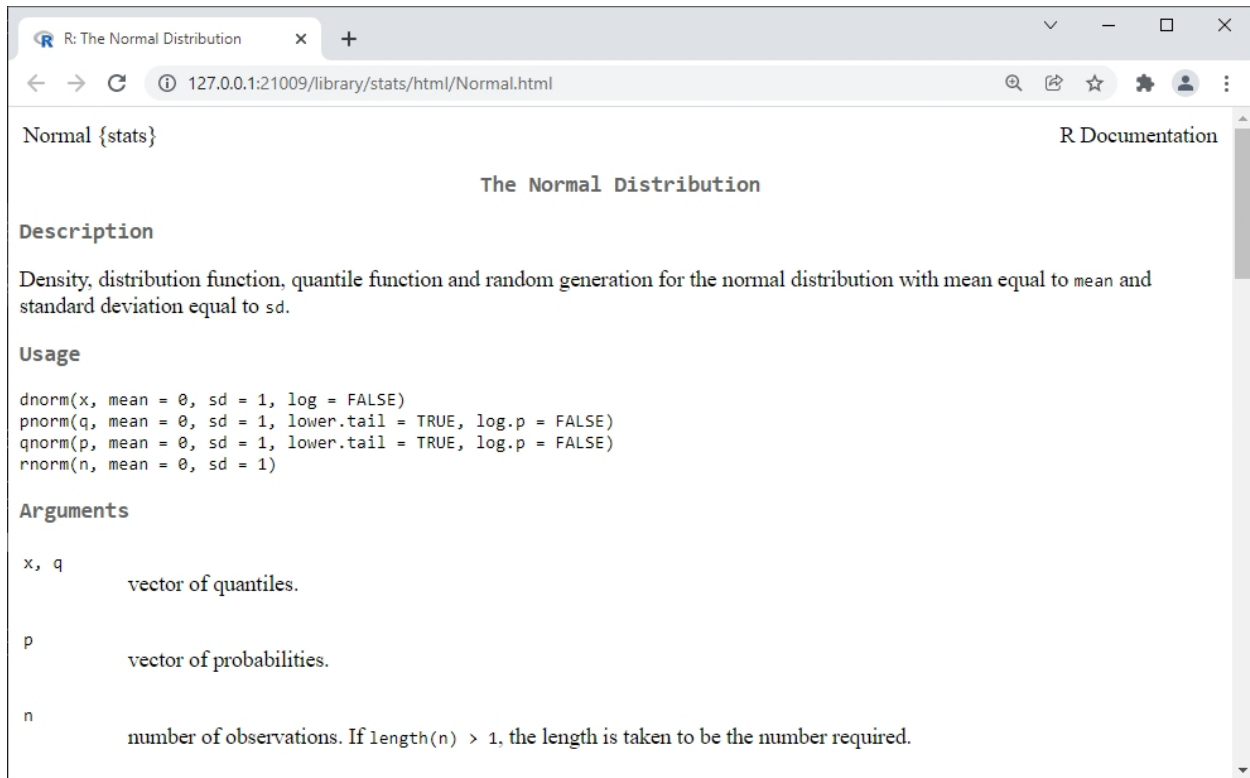
Figure 2.4: Example R documentation page for a common function.



Some functions belong to a family of related functions that share a common documentation page. For example, all of the functions related to the normal distribution share a common page:

```
?dnorm  
?pnorm  
?qnorm  
?rnorm
```

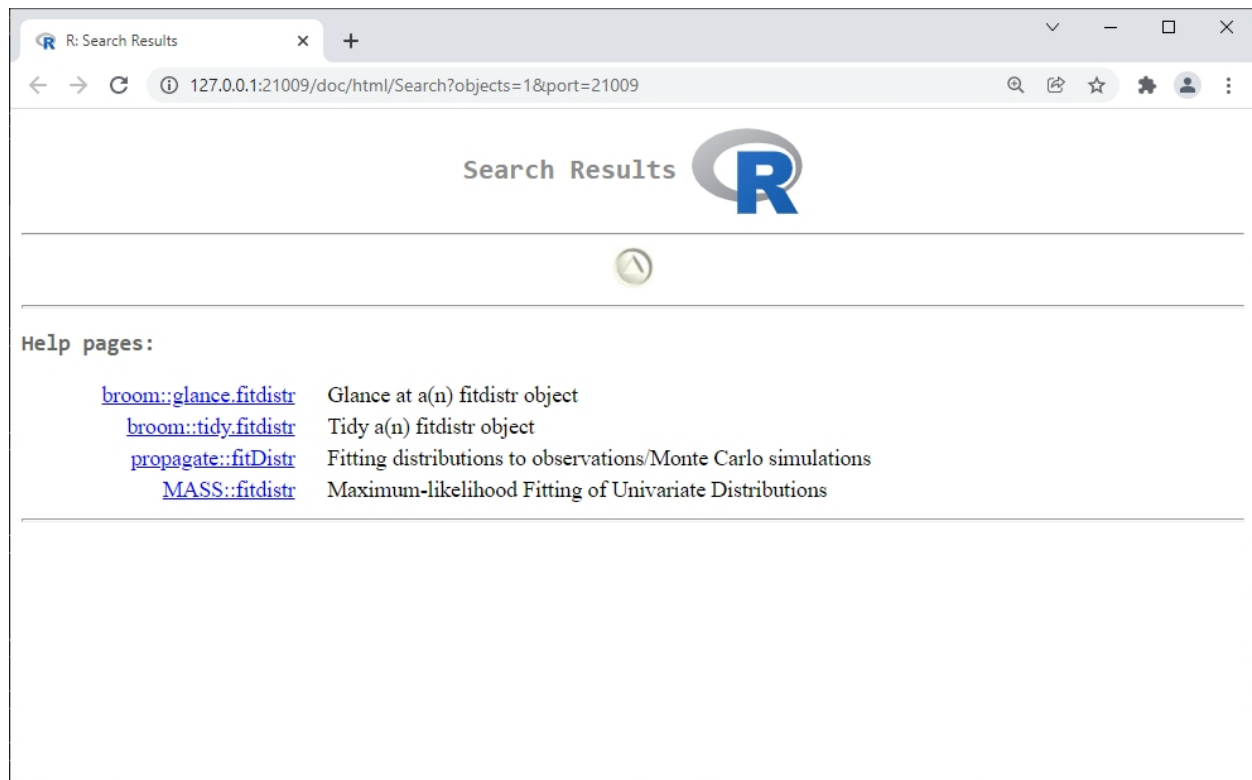
Figure 2.5: Example R documentation page for a family of related functions.



You can also search for help on functions in packages that are not attached. Or, you might want to know about a function but don't know what package it belongs to. Either way, `??functionname` is a more general search option that will search any package in your library. Compare what happens if search for function `fitdistr()` using `?` and `??` (this function is in package `MASS`, and so not loaded by default). With `??`, R searches all installed packages and returns a browser window with every result that might match your search term.

```
?fitdistr
??fitdistr
```

Figure 2.6: Example result of using `??` to search for an R function across many packages.



The R documentation pages all tend to follow the same format:

Description: brief description of what the function does.

Usage: presentation of the function with its arguments in their typical format and expected order.

Arguments: list of the arguments to the function in order, their required type/format, and allowed values. This section is very important. When my code isn't working, the answer to why is usually in the arguments section.

Details: additional information necessary to use the function.

Value: description of the function's outputs.

Source: description of where the original algorithm or method came from.

References: citations for the methods used.

See also: list of similar or related R functions.

Examples: illustration of how to use the function. This section is **very** important. If your code is not working, compare how you are using the function to the examples. Are you inputs in the right format? Did you name your arguments or put unnamed arguments in the wrong order? Careful examination and running of the examples will usually reveal why your code is not working.

R documentation pages tend to be very terse and include very little in the way of background or explanation. They are generally meant to refresh the memory of people already familiar with R and with statistics, not to

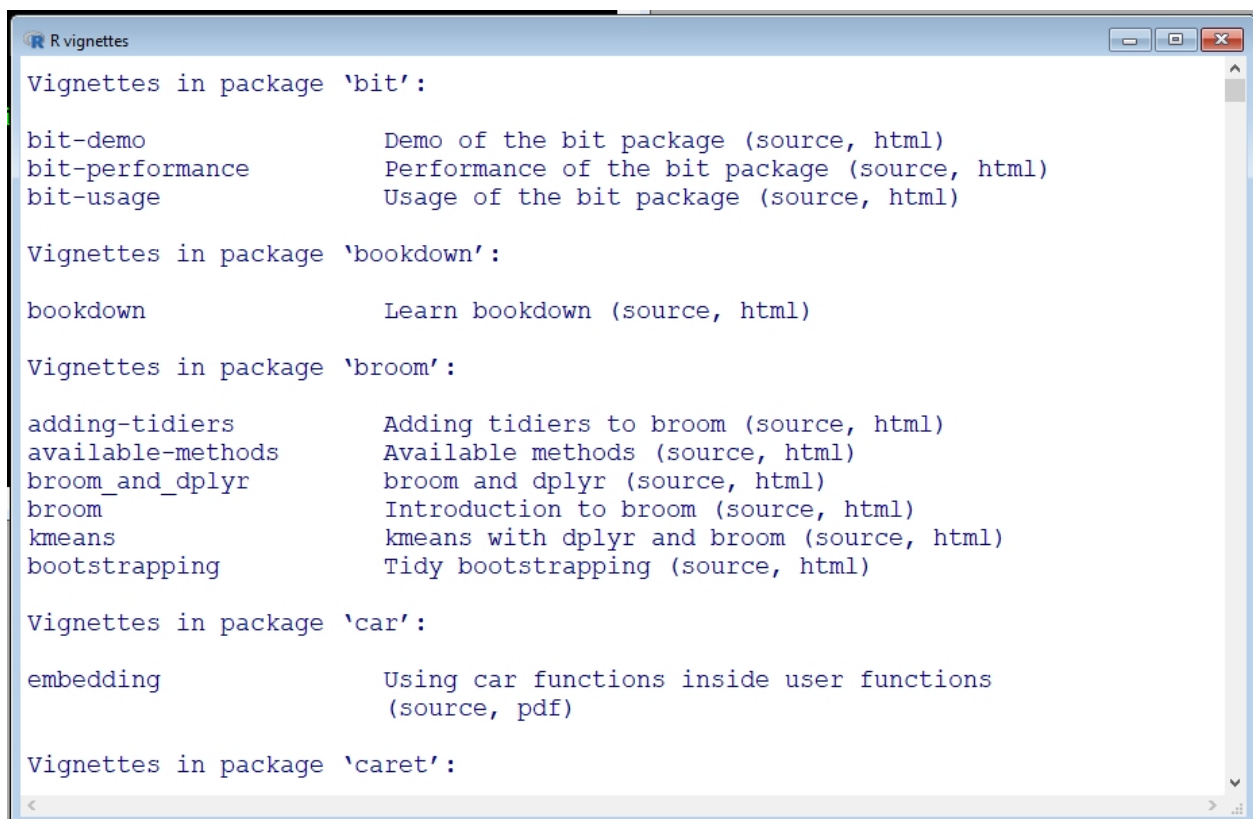
educate new R users or inexperienced statisticians. If you fall into one of the latter categories, or if you are trying to learn a new function, read on.

2.5.5.2 R vignettes

A **vignette** is a document demonstrating the use of an R package or its functions, with more details and explanation than in the help files. Not all packages offer vignettes. You can see what vignettes are available with the command `vignette()`. Specific vignettes are accessed by name and package. The examples below show how to get a list of available vignettes, available vignettes by package, and open specific vignettes.

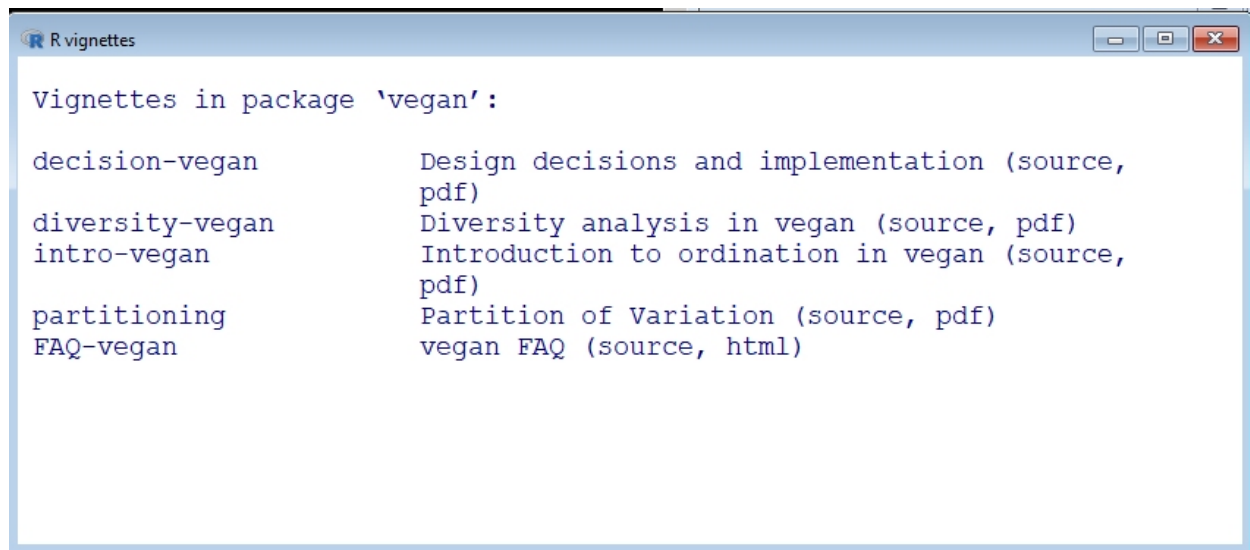
```
# list vignettes in all installed packages
vignette()
```

Figure 2.7: List of vignettes returned by function `vignette()`.



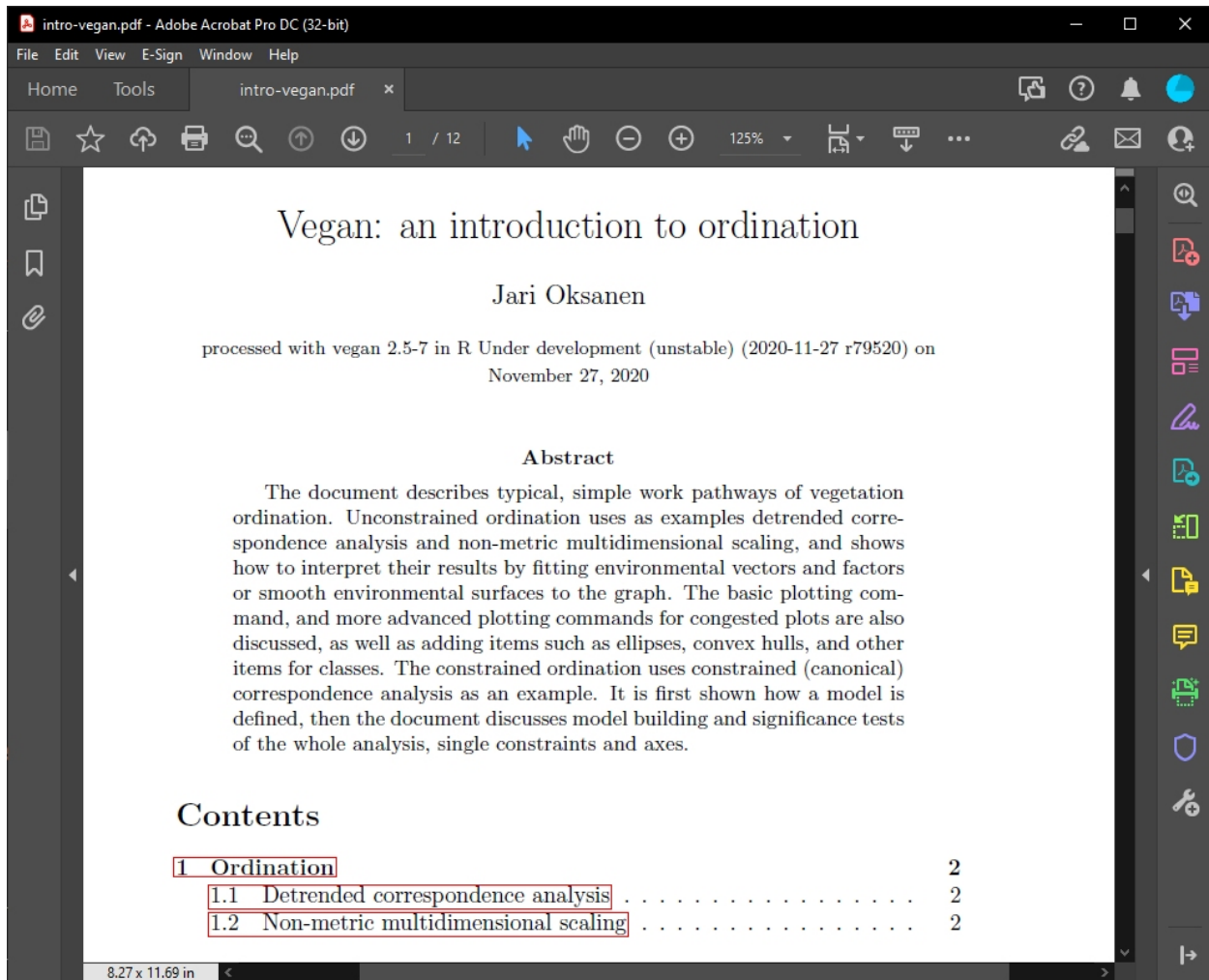
```
vignette(package="vegan")
```

Figure 2.8: List of vignettes in a single package, vegan.



```
# open vignette "intro-vegan" in local PDF reader  
vignette("intro-vegan", package="vegan")
```


Figure 2.9: A vignette from package `vegan` in PDF format.



2.5.5.3 Official resources

The R project maintains an [official help page](#) with lists of places to get help. They also maintain a [list of manuals and documentation](#). If you want a handy desk reference, you could do worse than the [R reference card](#) (accessed 2021-12-21).

The [Journal of Statistical Software](#) publishes papers about new statistical algorithms and software packages. Many of its papers are about new R packages and applications. The papers about new R packages and methods often include extensive code examples.

2.5.5.4 R task views

The [Task Views](#) on CRAN contain frequently updated lists of packages that are useful for specific domains. For example, ecologists might check the [Environmetrics](#), [Hydrology](#) or [Spatial](#) task views. Paleontologists

might be more interested in [Paleontology](#) or [Phylogenetics](#). Physiologists and biochemists might be interested in [Pharmacokinetics](#), [Clinical Trials](#), or [Survival](#). I've spent a lot of time in the [Bayesian](#), [Numerical Mathematics](#), and [Machine Learning](#) task views. There's something for everyone...even a [Teaching Statistics](#) task view with resources to help you learn R and stats at the same time.

(All links accessed 2025-08-06).

2.5.5.5 R mailing lists

If your question has been asked before—and this is very likely—there may be an answer waiting for you on the [R Help mailing list](#). Results from the R help mailing list will usually show up in Google results. You can submit a question to the R mailing list, but be very cautious about doing so. You will need to provide a short, clear, reproducible example in your question so that other people can see the issue you are having³. You also need to be sure that your question hasn't been asked before. Some members of the mailing list will respond with great hostility and condescension if you fail to follow those rules.

There are several [special interest groups \(SIG\)](#), e-mail mailing lists for R users who work in specific fields. For example, [R-sig-ecology](#) is an e-mail list for ecologists who use R. Other SIG of interest to biologists include [R-sig-genetics](#) for genetics, [R-sig-mixed-models](#) for mixed models, [R-sig-phylo](#) for phylogenetics and related methods, and [R-sig-Geo](#) for GIS and spatial analysis.

2.5.5.6 Unofficial online resources

Unofficial resources are becoming more and more abundant and useful. One of the most important sources for R help is [StackOverflow](#). StackOverflow is a hugely popular forum site where programmers ask and answer questions, and has an active [R section](#). When (not if) you find a solution on StackOverflow, include a link to the answer in your code as a comment. This can save you some headache down the line when you need to remember how you solved a problem. Other useful resources include [RSeek.org](#), a search engine for R resources; and [R-Bloggers](#), which aggregates many blogs about R and statistics.

Many R users maintain blogs or websites that may or may not be aggregated on R-Bloggers. You can usually find these via a Google search. I have used the pages on [Quick-R](#) many times. The UCLA Statistical Consulting Group has an [excellent collection of tutorials](#).

2.5.5.7 R books

The last few years have seen an explosion of books published specifically about the R language and its uses. Some of these books are general introductions to R and statistics; others are very domain specific. Inclusion of a book here does not constitute specific endorsement. I own some of these, but others are included because they are common R-related references that you might find helpful.

³This means a very minimal code example that people can use to get the same problem that you are having. Don't post your entire dataset or include many lines of pre-processing steps or other code that led up to your problem. Use base R functions or built-in datasets to create a small dataset or set of values that will immediately provoke the same error code. Often the process of doing this will help you solve your own problem!

- The [Use R! series](#) contains many high-quality general and domain-specific books, although some of the older titles are a bit dated. For example:
 - A Beginner's Guide to R by Alain Zuur et al.
 - Data Manipulation with R by Phil Spector
 - Introductory time series with R by P.S.P. Cowperthwaite and A.V. Metcalfe.
 - Bayesian computation with R by Jim Albert
 - Introductory Statistics with R by Peter Dalgaard
 - Morphometrics with R by Julien Claude
- The R Book by Michael Crawley
- Advanced R by Hadley Wickham
- R for Dummies by Andrie DeVries

When I was first starting out with R I found several books invaluable. Having code examples printed where I could highlight, annotate, etc., was really helpful. Over time I've found myself relying less on textbooks and more on online sources (especially StackOverflow). Still, there are a few books that I use frequently, for their statistical exposition more than their code examples. At least one of Bolker (2008), Kéry (2010), Legendre and Legendre (2012), Zuur *et al.* (2007), and Zuur *et al.* (2009a) is usually on my desk at any given time.

2.5.5.8 Two reminders

I said it at the beginning of this section, and it bears repeating: ***Don't panic!*** R has a steep learning curve, but the pay-off is worth it. There are plenty of resources and people out there who can help you.

Finally, be sure to ***read the manual*** when you are stuck. The R help pages and code examples usually have a solution. Make sure your inputs are in the right format. Check for syntax and spelling error—including capitalization! As a funny meme on I saw on Reddit said, *6 hours of debugging can save you 5 minutes of reading documentation.*

Chapter 3

R data types and objects

3.1 R data types

In R, data are stored in **objects** with a well-defined structure. The structures of these objects can facilitate manipulating and retrieving your data. The structure of an R object is determined by its **class**. The format of data (i.e., the kinds of values contained in an object) is referred to as the data **type**. Most R users only encounter types when trying to understand error messages. This is because many R functions will return errors if their inputs are not of the correct type.

3.1.1 Character type

Character data are text strings. They are input and printed using quotation marks (" "). Note that straight quotes are used. No other kind of quotation mark is valid in R code. If you paste in text from another program that has “curly quotes” (e.g., from Word), you will get an error.

```
a <- c("aardvark", "bunny", "caterpillar")  
a
```

```
[1] "aardvark"      "bunny"         "caterpillar"
```

Notice that in the command above the values in `a` are defined in quotation marks. Without quotation marks, R would interpret those words as the names of objects. For example:

```
caterpillar <- "lion"  
c("aardvark", "bunny", caterpillar)
```

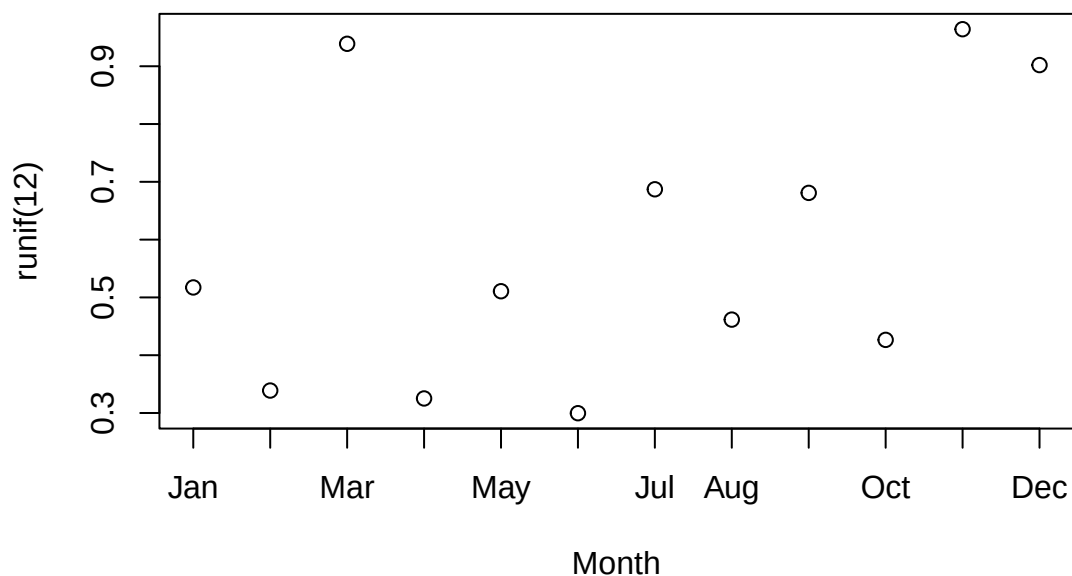
```
[1] "aardvark" "bunny"    "lion"
```

There are some special character vectors pre-built into R that can be useful in programming: `letters` returns the lower case letters a-z, while `LETTERS` returns the upper-case letters A-Z.

```
letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
LETTERS
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
letters[11:20]
## [1] "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
```

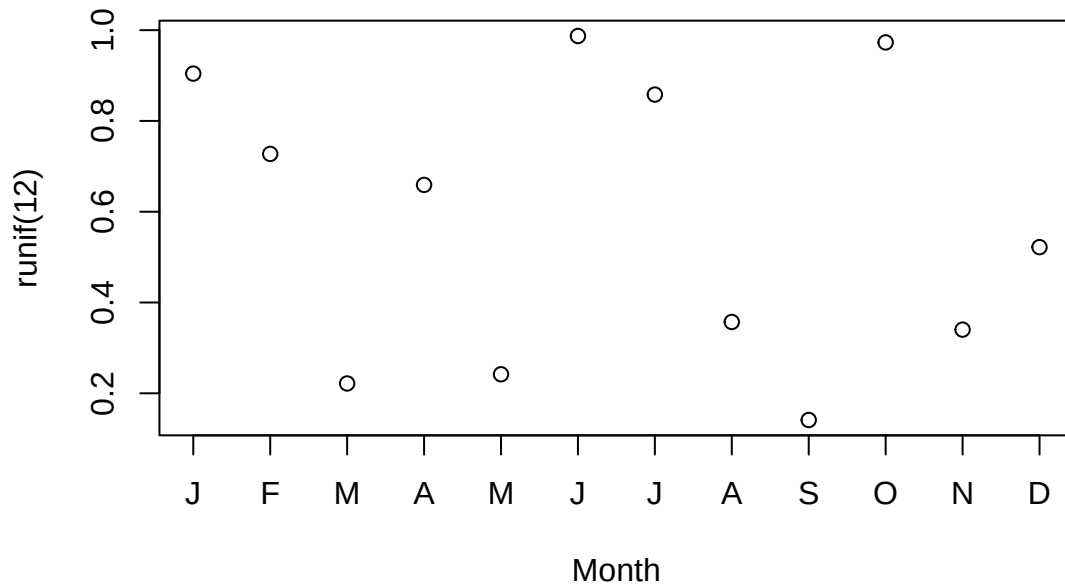
Similarly, `month.name` and `month.abb` contain the names of the 12 months and their 3 letter abbreviations, respectively. The latter is very useful for making figure axes.

```
plot(1:12, runif(12), xaxt="n", xlab= "Month")
axis(side=1, at=1:12, labels=month.abb)
```



As an aside, the function `substr()` extracts parts of character strings (“substring”). One application is getting one letter abbreviations of the months for a figure axis. The example below selects from the month names starting and ending at character 1. The argument `xaxt="n"` in `plot()` suppresses the x-axis, so we can add our own with `axis()`.

```
use.months <- substr(month.name, 1, 1)
plot(1:12, runif(12), xaxt="n", xlab= "Month")
axis(side=1, at=1:12, labels=use.months)
```



3.1.2 Numeric type

Most data are stored as the **numeric** type. This includes **floating point numbers** (e.g., 3.14159 or 3.14e0). Floating point numbers are referred to as type “double” in R (short for “[double-precision floating point](#)”).

```
typeof(2)
```

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

```
typeof(pi)
```

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

3.1.3 Integer type

Integers are numbers with no fractional components. Some numbers are stored as integers, but integers are not always interchangeable with floating point numbers. For example, some probability distribution

functions return integer values rather than floating point values (e.g., binomial or Poisson). This can be problematic if very large numbers are expected because the maximum integer value in R is much smaller than the maximum floating-point value. Note that having the type `integer` in R and *being an integer* from a mathematical standpoint are not synonymous.

```
# numbers are double by default:
typeof(7)
## [1] "double"

# convert to integer with as.integer():
typeof(as.integer(7))
## [1] "integer"

# test whether input is integer type:
is.integer(7)
## [1] FALSE

# test whether input is numeric type:
is.numeric(7)
## [1] TRUE

# some functions return integer type as output:
typeof(rbinom(1, 10, 0.7))
## [1] "integer"

# 7 is mathematically an integer (7 mod 1 == 0),
# but is not of integer type in R
7 %% 1
## [1] 0
```

Integers are used as indices for subsetting and accessing other objects. A sequence of integers can be made using the colon operator as follows:

```
a <- 1:10
a
## [1] 1 2 3 4 5 6 7 8 9 10
typeof(a)
## [1] "integer"
```

An important consequence of using integers as index values is that object dimensions are limited to the **R integer limit**. This is probably not a major concern for most users. Your machine will likely run out of memory before you could make such an object anyway. You can check the integer limit on your machine with this line:

```
.Machine$integer.max
```

```
[1] 2147483647
```

On my machine, this is about 2.14×10^9 . If you need data objects with dimensions larger than this, you may need to either use a specialized R package that can handle huge objects, or you may need rethink what you are doing.

3.1.4 Logical type

The **logical** data type stores `TRUE` and `FALSE` values. Note that `TRUE` and `FALSE` are always written in capital letters, without quotation marks. Logical values are produced by logical tests, such as the test for equality using `==`:

```
1+1 == 2
## [1] TRUE
typeof(1+1 == 2)
## [1] "logical"
typeof(1+1 == 3)
## [1] "logical"
```

Another useful test you can do is to test whether elements of one set are in another:

```
5 %in% 1:10
## [1] TRUE
5 %in% 6:10
## [1] FALSE
1:10 %in% 5:15
## [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
```

You can test with inequalities:

```
2 >= 1
## [1] TRUE
3 < 2
## [1] FALSE
```

Sometimes it can be useful to store logical values as 1 or 0. You can do this by multiplying a logical by 1 or using function `as.numeric`.


```

1*(1+1 == 2)
## [1] 1
1*(1+1 == 3)
## [1] 0
typeof(1+1 == 2)
## [1] "logical"
typeof(1*(1+1 == 2))
## [1] "double"
typeof(as.numeric(1+1 == 1:5))
## [1] "double"

```

The **!** symbol **negates** or **reverses** logical values. It does not negate logical values stored as numbers.

```

2+2 == 4
## [1] TRUE
! 2+2 == 4
## [1] FALSE
1:10 %in% 5:15
## [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
!1:10 %in% 5:15
## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

```

The `any()` and `all()` functions test to see if any values in a set of logical values is true (`any()`) or if all of them are true (`all()`).

```

1:5 < 3
## [1] TRUE TRUE FALSE FALSE FALSE
any(1:5 < 3)
## [1] TRUE
all(1:5 < 3)
## [1] FALSE

```

TRUE and FALSE can also be abbreviated as uppercase T or F, but not lowercase t or f. This means that you should **not** create objects with names T, F, TRUE, etc. (don't name an object t either, because that is the name of the transpose function). The code block below illustrates the mischief that can be created by naming an object T and masking the built-in value for true.

```

# (1+1==2) is true.
T == (1+1==2)
## [1] TRUE

# make a new object T with value 3

```

```

T <- 3

# (1+1==2) is now not true, unless it is
T == (1+1==2)
## [1] FALSE
TRUE == (1+1==2)
## [1] TRUE

# remove the troublesome T
rm(T)

# things are back to normal
T == (1+1==2)
## [1] TRUE

```

3.1.5 Special values

3.1.5.1 Missing values NA

Missing data in R are labeled as NA, or “Not Available”. This is R-speak for a blank or missing value. Some functions can ignore NA, some cannot. Some functions produce NA when you supply them with improper inputs. Note that this is a feature, not a bug. R is set up to produce NA by default in many cases to help you track down where the missing values are coming from. Function `is.na()` tests to see if a value is missing.

```

aa <- c(1:8, NA)

aa + 2
## [1] 3 4 5 6 7 8 9 10 NA

# returns NA
mean(aa)
## [1] NA

# option to ignore NA values
mean(aa, na.rm=TRUE)
## [1] 4.5

# test for NA
is.na(aa)
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

```

```

# which elements are NA?
which(is.na(aa))
## [1] 9

# remove NA elements
aa[-which(is.na(aa))]
## [1] 1 2 3 4 5 6 7 8

# another way to remove NA elements
aa[which(!is.na(aa))]
## [1] 1 2 3 4 5 6 7 8

# returns NA: Poisson distribution supported only on [0,+Inf]
rpois(10,-2)
## [1] NA NA NA NA NA NA NA NA NA NA

```

3.1.5.2 Infinities

Positive and negative infinity ($+\infty$ and $-\infty$) are labeled in R as `Inf` and `-Inf`. Dividing a non-zero number by 0 produces `Inf`. Taking the log of 0 produces `-Inf`. Trying to generate a value greater than R's maximal value can produce `Inf`.

```

5/0
## [1] Inf
log(0)
## [1] -Inf
10^10^10
## [1] Inf

```

3.1.5.3 Undefined values

Some values that are undefined are labeled `NaN` (not a number). This means that a value is undefined, not that it is missing (which would be `NA`) or infinite (which would be `Inf` or `-Inf`). Dividing 0 by 0, or taking the log of a negative number, produces `NaN`. The function `is.nan()` tests to see if a value is `NaN`.

```

0/0
## [1] NaN
log(-1)
## [1] NaN

```

3.1.5.4 Special values are useful

Why are there so many special values? Often they are meant to help track down where erroneous values are coming from. If every undefined value returned the same symbol (or word, etc.), then it would be harder to know why a program was returning undefined values. Instead, the different results help point the way:

- `Inf`: Positive non-zero number divided by 0
- `-Inf`: Negative non-zero number divided by 0 or `log(0)`.
- `NaN`: `0/0` or logarithm of a negative number

3.2 Basic R data structures

In R, data are stored in **objects** with a well-defined structure. The structures of these objects can facilitate manipulating and retrieving your data. The structure of an R object is determined by its class. The format of data (i.e., the kinds of values contained in an object) is referred to as the data type.

For a solid introduction to R classes and data types, refer to [An introduction to R](#), one of the official R help manuals. The book *Data manipulation with R* (Spector 2008) is also good. If you want to explore the odder and less benign aspects of R data structures, I recommend the truly excellent [The R Inferno](#) (accessed 2025-08-06).

3.2.1 Vectors

In R, as in mathematics, a **vector** is an ordered set of values. Vectors are a key concept in statistics and linear algebra. For example, the set `[1, 2, 3, 4, 5]` is a vector. This vector is different than the set (and vector) `[2, 3, 1, 5, 4]`. These sets are different vectors because even though they contain the same values, those values are in different orders. In R we could call either of those sets a “vector”. Note that we don’t call them “lists” because a list is a special kind of object in R that is not synonymous with vector.

As a language designed primarily for statistics, R has a strong emphasis on vectors and their manipulation. Vectors are probably the simplest type of data object in R. Understanding them is the key to understanding other data objects in R.

Another way to think of vectors is as an **array** with one dimension, length. In computing, an “array” is a systematic arrangement of similar elements. An array with 1 dimension is a vector. A two-dimensional array is a **matrix**, and an array with ≥ 3 dimensions is just called an array. Vectors, matrices, and arrays in R can be a convenient way to store structured data that are all of the same type (numeric, logical, character, etc.).

Vectors can be created by the function `vector()` as well as by functions specific to the type of value to be stored. The most common types are **numeric**, **logical**, and **character**. These hold numbers, logical values, and text, respectively. Each type has a default value (`0`, `FALSE`, and `""` (blank), respectively). Notice that in each command below R prints the results to the console because the resulting vectors are not assigned to an object.

```

vector("numeric", 5)
## [1] 0 0 0 0 0
vector("logical", 5)
## [1] FALSE FALSE FALSE FALSE FALSE
vector("character", 5)
## [1] "" "" "" "" ""

# equivalent to above commands:
numeric(5)
## [1] 0 0 0 0 0
logical(5)
## [1] FALSE FALSE FALSE FALSE FALSE
character(5)
## [1] "" "" "" "" ""

```

Creating vectors this way is cumbersome, but useful when programming complex tasks. It is usually more efficient, from a memory usage standpoint, to create an object and fill it later than to create it and repeatedly change its size.

You can also create vectors by defining their contents. When you do this, R will guess at the type of vector based on the values you give it. These guesses are usually okay.

```

# numeric:
class(runif(10))
## [1] "numeric"

# integer, which is stored in numeric vectors
# despite being a different type:
class(1:5)
## [1] "integer"
is.numeric(1:5)
## [1] TRUE

```

Vectors can also be made by putting values together using the function `c()` (short for “combine” or “concatenate”). Note that if you try to combine values of different types, R will convert all values to character strings.

```

# create a vector of numbers
my.vec <- c(1, 2, 4:8, 10)
my.vec
## [1] 1 2 4 5 6 7 8 10

```

```

# create a vector of character strings
my.vec2 <- c("zebra", "hippo", "lion", "rhino", "rhino")
my.vec2
## [1] "zebra" "hippo" "lion"  "rhino" "rhino"

# create a vector with 1 numeric, 1 character,
# and 1 logical value...all are converted to
# character strings.
my.vec3 <- c(1, "hippo", 1+1==3)
my.vec3
## [1] "1"      "hippo" "FALSE"

```

Each value within a vector is called an **element**. Elements of a vector are accessed by the bracket notation.

```

# get some random numbers
a <- runif(5)

# print to console
a
## [1] 0.2508072 0.3258648 0.4004049 0.4817464 0.1671091

# 3rd element
a[3]
## [1] 0.4004049

# elements 2-4
a[2:4]
## [1] 0.3258648 0.4004049 0.4817464

# elements other than 1
a[-1]
## [1] 0.3258648 0.4004049 0.4817464 0.1671091

# elements other than 1 and 2
a[-c(1:2)]
## [1] 0.4004049 0.4817464 0.1671091

```

You can inspect a vector (or any object) by printing it to the console. Do this by typing the object's name into the console at the command prompt `>` and pressing ENTER.

```
my.vec
```

```
[1] 1 2 4 5 6 7 8 10
```

The [1] at the start of the line with your results indicates that that line starts with the first element of the object. If an object runs over to multiple lines, each line will begin with the index of the first value printed on that line. Make your console window narrower and try this command:

```
runif(20)
```

```
[1] 0.96936023 0.32949942 0.84928612 0.89281459 0.22683672 0.24315660
[7] 0.02669942 0.51838261 0.98298806 0.98969328 0.58012573 0.09212624
[13] 0.84867899 0.08735776 0.75321744 0.85389591 0.29513531 0.53199513
[19] 0.89104634 0.78380699
```

Each element of a vector can have a name, set or accessed using the function `names()`. Element names can even be used to access elements of a vector.

```
a <- runif(3)
names(a) <- c("value1", "value2", "value3")
names(a)
## [1] "value1" "value2" "value3"
a
##      value1      value2      value3
## 0.3567507 0.7502037 0.5409710
a["value3"]
##      value3
## 0.540971
```

Vectors can be extended using `c()` or by changing their length. Changing the length of a vector can also shorten it. If a vector is made longer than the number of values it has, the extra elements will be filled in with NA, which is the label R uses for blanks.

```
# make a numeric vector of length 8
b <- 1:8

# combine it with another value to
# increase length to 9
b <- c(b, 10)
b
## [1] 1 2 3 4 5 6 7 8 10
```

```

# remake our vector with length 8
b <- 1:8

# set its length to 10
length(b) <- 10
# gaps filled with NA
b
## [1] 1 2 3 4 5 6 7 8 NA NA

# remake our vector with length 8
b <- 1:8
# change length to 6 (shorter)
length(b) <- 6
# vector truncated
b
## [1] 1 2 3 4 5 6

```

As mentioned elsewhere, many R functions take vectors as input and use them one element at a time. This is very handy when working with data, because you will often need to perform the same operation on every observation. Like any R data object, you can use a vector in a function simply by using the name in place of the values:

```

my.vec <- 1:8
my.vec*2
## [1] 2 4 6 8 10 12 14 16
mean(my.vec)
## [1] 4.5
dpois(my.vec, lambda=5)
## [1] 0.03368973 0.08422434 0.14037390 0.17546737 0.17546737 0.14622281 0.10137527 0.06527804
## [8] 0.06527804

```

3.2.1.1 Atomic vectors

A final note about vectors: some R functions, help pages, and error messages refer to “atomic vectors”. These are any object type whose elements cannot be split (i.e., “atomic” in the way Democritus meant the word). For example, the scalar 2 is considered atomic, being a numeric vector of length 1 whose sole element cannot be split up into something smaller. Likewise, the vector 2 : 3 is also atomic, because neither of its elements (2 and 3) can be split up into small elements. This definition extends to arrays with ≥ 3 dimensions.


```
a <- 4
a
## [1] 4
is.atomic(a)
## [1] TRUE
```

```
a <- runif(10)
a
## [1] 0.5650119 0.2415507 0.3732698 0.9659046 0.7565942 0.6913067 0.6444240
## [8] 0.6210210 0.5673819 0.1042397
is.atomic(a)
## [1] TRUE
```

```
a <- matrix(1:6, ncol=3)
a
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
is.atomic(a)
## [1] TRUE
```

```
a <- array(1:12, dim=c(3,2,2))
a
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
is.atomic(a)
## [1] TRUE
```

On the other hand, a **list** is not atomic but rather **recursive**. This means that any of its elements can be another list. Consider the following:

```

a <- list(x=1:12, y="a", z=list(m=t.test(rnorm(20)), n="a"))
a
## $x
## [1] 1 2 3 4 5 6 7 8 9 10 11 12
##
## $y
## [1] "a"
##
## $z
## $z$m
##
## One Sample t-test
##
## data: rnorm(20)
## t = -0.76611, df = 19, p-value = 0.453
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
## -0.6246549 0.2899000
## sample estimates:
## mean of x
## -0.1673775
##
##
## $z$n
## [1] "a"

```

```

is.atomic(a)
## [1] FALSE
is.recursive(a)
## [1] TRUE

```

The list `a` is recursive because any of its elements could be another list. However, consider the list `b`:

```

b <- list(x=1, y=2, z=3)
b
## $x
## [1] 1
##
## $y
## [1] 2
##

```

```
## $z
## [1] 3
```

```
is.atomic(b)
## [1] FALSE
is.recursive(b)
## [1] TRUE
```

Note that while every element of `b` is atomic in and of itself (satisfying the definition of an atomic object above), `b` is in fact recursive. This is because any element of `b`, **could** be a list like `b`. This is a bit paradoxical, but not enough to cause any trouble.

Why do I bring this up? Because it is very easy to provoke the dreaded, `operator invalid for atomic vectors` error. This usually means you are trying to treat a non-list as a list.

```
# make a simple matrix
a <- matrix(1:4, nrow=2)
colnames(a) <- c("a", "b")
a
```

```
      a b
[1,] 1 3
[2,] 2 4
```

```
# not run, but try on your machine
```

```
# incorrect: using $ to extract from an atomic vector
# (in this case a matrix)
a$a
```

```
# correct: using column names and []
a[, "a"]
```

```
[1] 1 2
```

Atomic vectors can contain 1 and only one type of data: logical, integer, double, character, complex, and raw.

3.2.2 Data frames

Most of the time when you work with data in R, you will work with objects of the **data frame** class. A data frame looks and acts a lot like a spreadsheet. One key aspect of data frames is that data frames can store

data of more than one type, while vectors, matrices, and arrays cannot. The examples below use the built-in data frame `iris`.

Each **row** of a data frame usually corresponds to one **observation**. Each **column** contains the values of one **variable**. You can access data frame columns by name with the `$` operator:

```
iris$Sepal.Length
```

```
[1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
[19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
[37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
[55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
[73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
[91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
[109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
[127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
[145] 6.7 6.7 6.3 6.5 6.2 5.9
```

You can also access data frame columns by number or name with bracket notation:

```
iris[,1]
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
## [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
## [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
## [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
## [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
## [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

```
iris[, "Petal.Width"]
```

```
## [1] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 0.2 0.2 0.1 0.1 0.2 0.4 0.4 0.3
## [19] 0.3 0.3 0.2 0.4 0.2 0.5 0.2 0.2 0.4 0.2 0.2 0.2 0.2 0.4 0.1 0.2 0.2 0.2
## [37] 0.2 0.1 0.2 0.2 0.3 0.3 0.2 0.6 0.4 0.3 0.2 0.2 0.2 0.2 1.4 1.5 1.5 1.3
## [55] 1.5 1.3 1.6 1.0 1.3 1.4 1.0 1.5 1.0 1.4 1.3 1.4 1.5 1.0 1.5 1.1 1.8 1.3
## [73] 1.5 1.2 1.3 1.4 1.4 1.7 1.5 1.0 1.1 1.0 1.2 1.6 1.5 1.6 1.5 1.3 1.3 1.3
## [91] 1.2 1.4 1.2 1.0 1.3 1.2 1.3 1.3 1.1 1.3 2.5 1.9 2.1 1.8 2.2 2.1 1.7 1.8
## [109] 1.8 2.5 2.0 1.9 2.1 2.0 2.4 2.3 1.8 2.2 2.3 1.5 2.3 2.0 2.0 1.8 2.1 1.8
## [127] 1.8 1.8 2.1 1.6 1.9 2.0 2.2 1.5 1.4 2.3 2.4 1.8 1.8 2.1 2.4 2.3 1.9 2.3
## [145] 2.5 2.3 1.9 2.0 2.3 1.8
```

Rows of a data frame are accessed by number with bracket notation. As seen above, you can also select using logical tests.

```
iris[1,]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5         1.4         0.2   setosa
```

```
# not run because output very long; try it on your machine!
iris[which(iris$Species == "setosa"),]
```

Unlike matrices, rows and columns of data frames are something different. This is because under the surface, a data frame is really a list, and what looks like a column of a data frame is really an element of the list. So, a row of a data frame is really a new data frame with one row. When you work with data frames in R, the data frame class and its associated methods offer a convenient way to interact with the underlying list structure.

```
class(iris[1,])
## [1] "data.frame"
class(iris[,1])
## [1] "numeric"
```

Practically, this means that if you select a column from a data frame you get a vector, and if you select a row (or rows) you get another data frame. So, if you want to operate on values within a row, you might need to use an `apply()` function.

```
mean(iris[1,1:4])           # apply function needed
## [1] NA
apply(iris[1,1:4], 1, mean) # works!
##      1
## 2.55
sum(iris[1,1:4])            # works!
## [1] 10.2
```

Columns can be added to data frames by assigning them a value:

```
iris2 <- iris # Make a spare copy
iris2$petal.area <- iris2$Petal.Length*iris2$Petal.Width
```

You can view the first few rows of a data frame with the command `head()`, and the last few rows with the command `tail()`. These are incredibly useful!

```
head(iris2)
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species petal.area
## 1           5.1         3.5         1.4         0.2   setosa         0.28
```

```
## 2      4.9      3.0      1.4      0.2 setosa      0.28
## 3      4.7      3.2      1.3      0.2 setosa      0.26
## 4      4.6      3.1      1.5      0.2 setosa      0.30
## 5      5.0      3.6      1.4      0.2 setosa      0.28
## 6      5.4      3.9      1.7      0.4 setosa      0.68
tail(iris2)
##      Sepal.Length Sepal.Width Petal.Length Petal.Width  Species petal.area
## 145          6.7          3.3          5.7          2.5 virginica      14.25
## 146          6.7          3.0          5.2          2.3 virginica      11.96
## 147          6.3          2.5          5.0          1.9 virginica       9.50
## 148          6.5          3.0          5.2          2.0 virginica      10.40
## 149          6.2          3.4          5.4          2.3 virginica      12.42
## 150          5.9          3.0          5.1          1.8 virginica       9.18
```

Columns can be removed from a data frame by assigning them a value of `NULL`.

```
iris2$petal.area <- NULL
head(iris2)
```

```
      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1          3.5          1.4          0.2 setosa
2           4.9          3.0          1.4          0.2 setosa
3           4.7          3.2          1.3          0.2 setosa
4           4.6          3.1          1.5          0.2 setosa
5           5.0          3.6          1.4          0.2 setosa
6           5.4          3.9          1.7          0.4 setosa
```

We'll explore more aspects of data frames in [Chapter 5](#).

3.2.3 Matrices and arrays

Sometimes you have a structured set of values that does not need to be stored as a data frame, or should not be stored as a data frame for ease of access. These situations call for matrices and arrays.

An **array** with two dimensions is a **matrix**. The dimensions of a matrix are its **rows** and **columns**. Matrices look a little like data frames, but internally they are very different. Unlike a data frame and like all arrays, a matrix can contain values of *only one type*. Matrices are necessary for any matrix operations (e.g., Leslie matrices) and for some function inputs. For most routine data analyses, data frames are easier to work with.

Matrices are created with the `matrix()` function. The matrix function must be provided with the values, the dimensions of the matrix, and the method for filling (by rows or by columns). If you don't know the values yet, you can just use `NA` or `0`.

```

my.mat1 <- matrix(1:12,nrow=3,ncol=4)
my.mat1
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

my.mat2 <- matrix(1:12,nrow=3,ncol=4,byrow=TRUE)
my.mat2
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12

my.mat3 <- matrix(NA,nrow=3,ncol=4)
my.mat3
##      [,1] [,2] [,3] [,4]
## [1,]   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA

```

Just as with other objects, the values in a matrix can be extracted using bracket notation. Because matrices have two dimensions, you must specify indices for both dimensions. This is done using a comma , to separate the row indices and column indices within the brackets. Just as with data frames, row indices are before the comma; column indices are after the comma. Leaving the row index or the column index blank will select all rows or columns, respectively.

```

my.mat1[2,1]# row 2, column 1
## [1] 2
my.mat1[3,3]# row 3, column 3
## [1] 9

```

Entire rows or columns of a matrix can be extracted by leaving the other dimension blank. You must still include the comma within the brackets. Notice that both the rows and columns of a matrix are vectors, not a matrix. If you want to get a row or column as a 1 row or 1 column matrix, you must specifically request this.

```

# first row
my.mat1[1,]
## [1] 1 4 7 10
is.matrix(my.mat1[1,])
## [1] FALSE

```

```
# third column
my.mat1[,3]
## [1] 7 8 9

# convert extracted part to matrix
my.mat3 <- matrix(my.mat1[,3], nrow=1)
is.matrix(my.mat3)
## [1] TRUE
```

An array in R is usually only called an array if it has ≥ 3 dimensions. Data stored this way can make programming easier, if there is a clear relationship between the dimensions. Arrays are not often encountered by the average user. The example below has 3 “layers”, each of which has 4 rows and 2 columns.

```
my.array <- array(1:24, dim=c(4,2,3))
my.array
```

```
, , 1
```

	[,1]	[,2]
[1,]	1	5
[2,]	2	6
[3,]	3	7
[4,]	4	8

```
, , 2
```

	[,1]	[,2]
[1,]	9	13
[2,]	10	14
[3,]	11	15
[4,]	12	16

```
, , 3
```

	[,1]	[,2]
[1,]	17	21
[2,]	18	22
[3,]	19	23
[4,]	20	24

One very nifty (and potentially confusing) ability of arrays is the ability to be sliced into differently shaped

pieces¹. Consider the examples below and see if you can work out why the results are what they are.

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

3.2.4 Lists

One of the most important kinds of R object is the **list**. A list can be thought of like a *series of containers or buckets*. Each bucket can contain completely unrelated things, or even other series of buckets. Because of this flexibility, lists are extremely versatile and useful in R programming. Many function outputs are really lists (e.g., the outputs of `lm()` or `t.test()`). Data frames are also lists.

Lists can be created using the function `vector()` and then filled later. The following example creates an empty list with 4 elements (i.e., buckets), then fills each element with something different.

¹My personal record is a 7-dimensional array, which I used in a community simulation model. The model included many species-, age-, and sex-specific demographic parameters that had to be extracted and summarized, and because of the size of the model an array-based solution was more computationally-efficient than a list- or data frame-based solution. I don't recommend building arrays that complex...it was a nightmare to work with.

```

my.list <- vector("list",length=4)
my.list[[1]] <- 1:10
my.list[[2]] <- c("a","b","c")
my.list[[3]] <- t.test(iris$Sepal.Length)
my.list[[4]] <- iris
my.list
## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10
##
## [[2]]
## [1] "a" "b" "c"
##
## [[3]]
##
## One Sample t-test
##
## data: iris$Sepal.Length
## t = 86.425, df = 149, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
## 5.709732 5.976934
## sample estimates:
## mean of x
## 5.843333
##
##
## [[4]]
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1 5.1 3.5 1.4 0.2 setosa
## 2 4.9 3.0 1.4 0.2 setosa
## 3 4.7 3.2 1.3 0.2 setosa
## 4 4.6 3.1 1.5 0.2 setosa
## 5 5.0 3.6 1.4 0.2 setosa
## 6 5.4 3.9 1.7 0.4 setosa
## 7 4.6 3.4 1.4 0.3 setosa
## 8 5.0 3.4 1.5 0.2 setosa
## 9 4.4 2.9 1.4 0.2 setosa
## 10 4.9 3.1 1.5 0.1 setosa
## 11 5.4 3.7 1.5 0.2 setosa
## 12 4.8 3.4 1.6 0.2 setosa
## 13 4.8 3.0 1.4 0.1 setosa

```

## 14	4.3	3.0	1.1	0.1	setosa
## 15	5.8	4.0	1.2	0.2	setosa
## 16	5.7	4.4	1.5	0.4	setosa
## 17	5.4	3.9	1.3	0.4	setosa
## 18	5.1	3.5	1.4	0.3	setosa
## 19	5.7	3.8	1.7	0.3	setosa
## 20	5.1	3.8	1.5	0.3	setosa
## 21	5.4	3.4	1.7	0.2	setosa
## 22	5.1	3.7	1.5	0.4	setosa
## 23	4.6	3.6	1.0	0.2	setosa
## 24	5.1	3.3	1.7	0.5	setosa
## 25	4.8	3.4	1.9	0.2	setosa
## 26	5.0	3.0	1.6	0.2	setosa
## 27	5.0	3.4	1.6	0.4	setosa
## 28	5.2	3.5	1.5	0.2	setosa
## 29	5.2	3.4	1.4	0.2	setosa
## 30	4.7	3.2	1.6	0.2	setosa
## 31	4.8	3.1	1.6	0.2	setosa
## 32	5.4	3.4	1.5	0.4	setosa
## 33	5.2	4.1	1.5	0.1	setosa
## 34	5.5	4.2	1.4	0.2	setosa
## 35	4.9	3.1	1.5	0.2	setosa
## 36	5.0	3.2	1.2	0.2	setosa
## 37	5.5	3.5	1.3	0.2	setosa
## 38	4.9	3.6	1.4	0.1	setosa
## 39	4.4	3.0	1.3	0.2	setosa
## 40	5.1	3.4	1.5	0.2	setosa
## 41	5.0	3.5	1.3	0.3	setosa
## 42	4.5	2.3	1.3	0.3	setosa
## 43	4.4	3.2	1.3	0.2	setosa
## 44	5.0	3.5	1.6	0.6	setosa
## 45	5.1	3.8	1.9	0.4	setosa
## 46	4.8	3.0	1.4	0.3	setosa
## 47	5.1	3.8	1.6	0.2	setosa
## 48	4.6	3.2	1.4	0.2	setosa
## 49	5.3	3.7	1.5	0.2	setosa
## 50	5.0	3.3	1.4	0.2	setosa
## 51	7.0	3.2	4.7	1.4	versicolor
## 52	6.4	3.2	4.5	1.5	versicolor
## 53	6.9	3.1	4.9	1.5	versicolor
## 54	5.5	2.3	4.0	1.3	versicolor

## 55	6.5	2.8	4.6	1.5 versicolor
## 56	5.7	2.8	4.5	1.3 versicolor
## 57	6.3	3.3	4.7	1.6 versicolor
## 58	4.9	2.4	3.3	1.0 versicolor
## 59	6.6	2.9	4.6	1.3 versicolor
## 60	5.2	2.7	3.9	1.4 versicolor
## 61	5.0	2.0	3.5	1.0 versicolor
## 62	5.9	3.0	4.2	1.5 versicolor
## 63	6.0	2.2	4.0	1.0 versicolor
## 64	6.1	2.9	4.7	1.4 versicolor
## 65	5.6	2.9	3.6	1.3 versicolor
## 66	6.7	3.1	4.4	1.4 versicolor
## 67	5.6	3.0	4.5	1.5 versicolor
## 68	5.8	2.7	4.1	1.0 versicolor
## 69	6.2	2.2	4.5	1.5 versicolor
## 70	5.6	2.5	3.9	1.1 versicolor
## 71	5.9	3.2	4.8	1.8 versicolor
## 72	6.1	2.8	4.0	1.3 versicolor
## 73	6.3	2.5	4.9	1.5 versicolor
## 74	6.1	2.8	4.7	1.2 versicolor
## 75	6.4	2.9	4.3	1.3 versicolor
## 76	6.6	3.0	4.4	1.4 versicolor
## 77	6.8	2.8	4.8	1.4 versicolor
## 78	6.7	3.0	5.0	1.7 versicolor
## 79	6.0	2.9	4.5	1.5 versicolor
## 80	5.7	2.6	3.5	1.0 versicolor
## 81	5.5	2.4	3.8	1.1 versicolor
## 82	5.5	2.4	3.7	1.0 versicolor
## 83	5.8	2.7	3.9	1.2 versicolor
## 84	6.0	2.7	5.1	1.6 versicolor
## 85	5.4	3.0	4.5	1.5 versicolor
## 86	6.0	3.4	4.5	1.6 versicolor
## 87	6.7	3.1	4.7	1.5 versicolor
## 88	6.3	2.3	4.4	1.3 versicolor
## 89	5.6	3.0	4.1	1.3 versicolor
## 90	5.5	2.5	4.0	1.3 versicolor
## 91	5.5	2.6	4.4	1.2 versicolor
## 92	6.1	3.0	4.6	1.4 versicolor
## 93	5.8	2.6	4.0	1.2 versicolor
## 94	5.0	2.3	3.3	1.0 versicolor
## 95	5.6	2.7	4.2	1.3 versicolor

## 96	5.7	3.0	4.2	1.2	versicolor
## 97	5.7	2.9	4.2	1.3	versicolor
## 98	6.2	2.9	4.3	1.3	versicolor
## 99	5.1	2.5	3.0	1.1	versicolor
## 100	5.7	2.8	4.1	1.3	versicolor
## 101	6.3	3.3	6.0	2.5	virginica
## 102	5.8	2.7	5.1	1.9	virginica
## 103	7.1	3.0	5.9	2.1	virginica
## 104	6.3	2.9	5.6	1.8	virginica
## 105	6.5	3.0	5.8	2.2	virginica
## 106	7.6	3.0	6.6	2.1	virginica
## 107	4.9	2.5	4.5	1.7	virginica
## 108	7.3	2.9	6.3	1.8	virginica
## 109	6.7	2.5	5.8	1.8	virginica
## 110	7.2	3.6	6.1	2.5	virginica
## 111	6.5	3.2	5.1	2.0	virginica
## 112	6.4	2.7	5.3	1.9	virginica
## 113	6.8	3.0	5.5	2.1	virginica
## 114	5.7	2.5	5.0	2.0	virginica
## 115	5.8	2.8	5.1	2.4	virginica
## 116	6.4	3.2	5.3	2.3	virginica
## 117	6.5	3.0	5.5	1.8	virginica
## 118	7.7	3.8	6.7	2.2	virginica
## 119	7.7	2.6	6.9	2.3	virginica
## 120	6.0	2.2	5.0	1.5	virginica
## 121	6.9	3.2	5.7	2.3	virginica
## 122	5.6	2.8	4.9	2.0	virginica
## 123	7.7	2.8	6.7	2.0	virginica
## 124	6.3	2.7	4.9	1.8	virginica
## 125	6.7	3.3	5.7	2.1	virginica
## 126	7.2	3.2	6.0	1.8	virginica
## 127	6.2	2.8	4.8	1.8	virginica
## 128	6.1	3.0	4.9	1.8	virginica
## 129	6.4	2.8	5.6	2.1	virginica
## 130	7.2	3.0	5.8	1.6	virginica
## 131	7.4	2.8	6.1	1.9	virginica
## 132	7.9	3.8	6.4	2.0	virginica
## 133	6.4	2.8	5.6	2.2	virginica
## 134	6.3	2.8	5.1	1.5	virginica
## 135	6.1	2.6	5.6	1.4	virginica
## 136	7.7	3.0	6.1	2.3	virginica

## 137	6.3	3.4	5.6	2.4	virginica
## 138	6.4	3.1	5.5	1.8	virginica
## 139	6.0	3.0	4.8	1.8	virginica
## 140	6.9	3.1	5.4	2.1	virginica
## 141	6.7	3.1	5.6	2.4	virginica
## 142	6.9	3.1	5.1	2.3	virginica
## 143	5.8	2.7	5.1	1.9	virginica
## 144	6.8	3.2	5.9	2.3	virginica
## 145	6.7	3.3	5.7	2.5	virginica
## 146	6.7	3.0	5.2	2.3	virginica
## 147	6.3	2.5	5.0	1.9	virginica
## 148	6.5	3.0	5.2	2.0	virginica
## 149	6.2	3.4	5.4	2.3	virginica
## 150	5.9	3.0	5.1	1.8	virginica

Another way to make a list is to make its elements first and combine them into a list, with function `list()`.

```
x <- runif(10)
y <- "zebra"
z <- matrix(1:12, nrow=3)
a <- list(x, y, z)
a
```

```
[[1]]
[1] 0.6910715 0.7080356 0.1003079 0.3473342 0.7076533 0.1869121 0.6725983
[8] 0.3593569 0.7071916 0.7920872
```

```
[[2]]
[1] "zebra"
```

```
[[3]]
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8    11
[3,]     3     6     9    12
```

Optionally, you can name each element as you make the list.

```
b <- list(x=x, y=y, z=z)
b
```

```
$x
```

```
[1] 0.6910715 0.7080356 0.1003079 0.3473342 0.7076533 0.1869121 0.6725983
[8] 0.3593569 0.7071916 0.7920872
```

```
$y
```

```
[1] "zebra"
```

```
$z
```

```
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8    11
[3,]     3     6     9    12
```

Much like vectors, lists can be extended by using concatenation:

```
a2 <- c(a,b)
a2
```

```
[[1]]
[1] 0.6910715 0.7080356 0.1003079 0.3473342 0.7076533 0.1869121 0.6725983
[8] 0.3593569 0.7071916 0.7920872
```

```
[[2]]
[1] "zebra"
```

```
[[3]]
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8    11
[3,]     3     6     9    12
```

```
$x
```

```
[1] 0.6910715 0.7080356 0.1003079 0.3473342 0.7076533 0.1869121 0.6725983
[8] 0.3593569 0.7071916 0.7920872
```

```
$y
```

```
[1] "zebra"
```

```
$z
```

```
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8    11
[3,]     3     6     9    12
```

Single elements of a list can be accessed with **double brackets** `[[ ]]`. As with other objects, elements can be selected by name or by index. If selecting by name, the `$` symbol can be used instead of double brackets.

```
a[[1]]
## [1] 0.6910715 0.7080356 0.1003079 0.3473342 0.7076533 0.1869121 0.6725983
## [8] 0.3593569 0.7071916 0.7920872
b[[2]]
## [1] "zebra"
b[["z"]]
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
b$z
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Using **single brackets** to access a list will return a *list* of the requested elements, not the requested elements themselves. This can be a little confusing. Compare the following two commands and notice that the first returns a vector containing the numbers 1 through 10 (the first element of `my.list`), while the second returns a list with 1 element, and that element is a vector with the numbers 1 through 10.

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

The reason for this confusion is that the single brackets notation allows you to extract multiple elements of the list at once to make a new list.

```
a2 <- a[1:2]
a2
```

```
[[1]]
[1] 0.6910715 0.7080356 0.1003079 0.3473342 0.7076533 0.1869121 0.6725983
[8] 0.3593569 0.7071916 0.7920872

[[2]]
[1] "zebra"
```


3.2.5 S3, S4, and other paradigms

We've seen so far that R is object-oriented. What is little appreciated is that R has at least three object-oriented paradigms: **S3**, **S4**, and **R5**. Most objects in R are S3 objects: data frames, matrices, vectors, etc. The R5 objects are more formally called "Reference Classes" and are still being developed. The S4 objects are only occasionally encountered by the average user, although some R developers believe that S4 objects should replace S3 in general use. Their reasons are esoteric and we won't get into them here. In a nutshell, S4 classes are structured more formally than S3 classes and are better for validation and code testing.

One key difference in practice between S4 and S3 objects is that S4 objects contain **slots** or **attributes**, which are accessed using `@` instead of `$`. Within each slot, an S4 object can contain all sorts of data, often in list form.

Chapter 4

Data import and export

For a beginning R user, getting data into and out of R can seem like the hardest part. This is because R interacts with your computer in a way that you're probably not used to. Once you understand a few things like how your computer stores files and how to save in the correct format, things will get easier!

4.1 Working with your file system

4.1.1 Your file system

Your file system begins with the **root directory**. In Windows, these are the top folder of a “letter drive” like **C:**, **E:**, and so on. On Mac and Linux, the root directory is simply **/**. Users should not store files in the root directory without a very good reason.

Every folder in the file system is a **branch** or **subdirectory** of the root. This creates a **hierarchy**, where folders and files are nested within other folders (except the root, which is not contained by anything).

Each folder and file has a **name**. Names of files are also called **filenames** or **file names**.

4.1.2 Folder addresses

The directory path or just **path** is the “address” of a folder or file within the hierarchy. Paths are constructed by combining the names of the root, the containing folders, and finally the terminal name (for folders) or filename (for files). The elements of the path are separated by slashes:

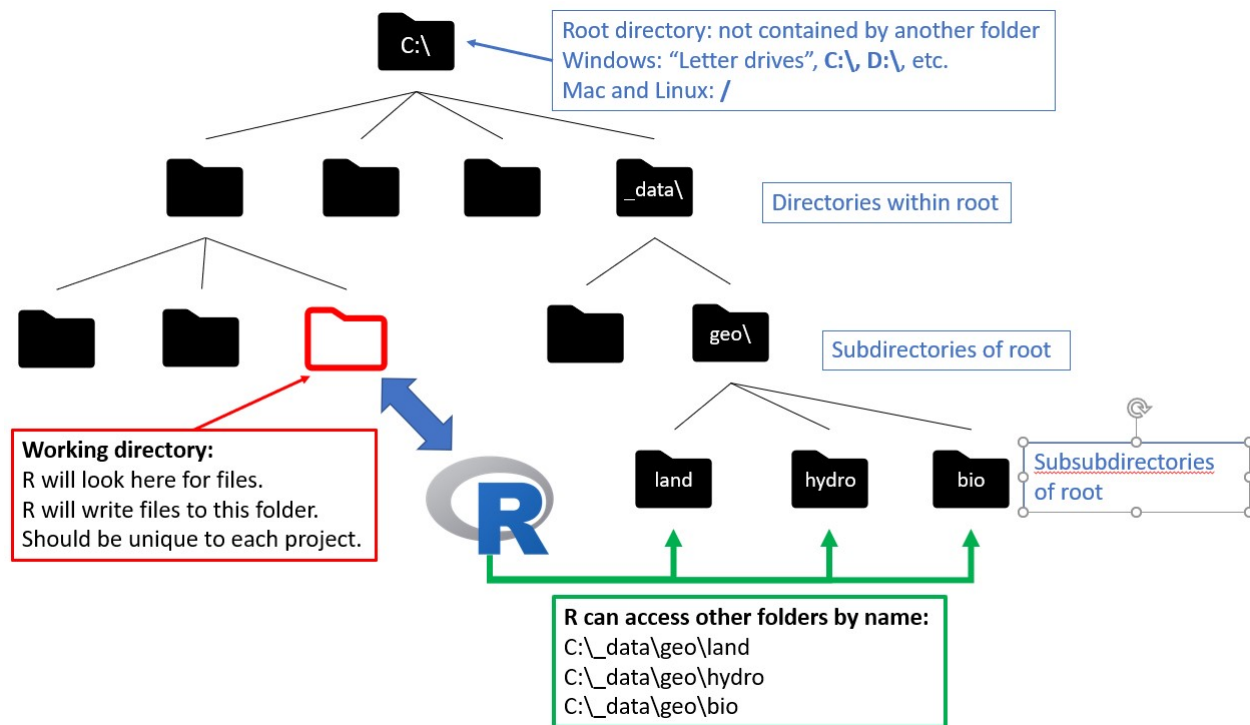
- In Windows, elements of paths are separated by backslashes ****.
- In Mac and Linux, elements of paths are separated by front-slashes **/**.

R uses front-slashes. So, Windows addresses need to have their **** changed to **/**, or escaped by doubling each to **** (see figure below).

4.1.3 The working directory

Many programs, including R, will designate a folder as the **working directory**: a folder which contains the data the program can access, and which will contain any files created by the program. The default R working directory is the **Documents** folder on Windows, or the **Home** folder on a Mac. You should choose a different folder to be your working directory, however. One common practice is to create a single folder for each project which contains all the data needed for that project, and which will receive all the output from R for the project.

Figure 4.1: Example of hierarchical structure of folders relative to the file system's root. Depending on context, the root can be the system root or the root for a project.



You can, and should, have a separate working directory for every project. This way files stay together, and the acts of importing and exporting files are easier.

In R, you can find your working directory in the console with `getwd()`:

```
getwd()

[1] "C:/_courses/appl-biol-data-analysis/chapters"
```

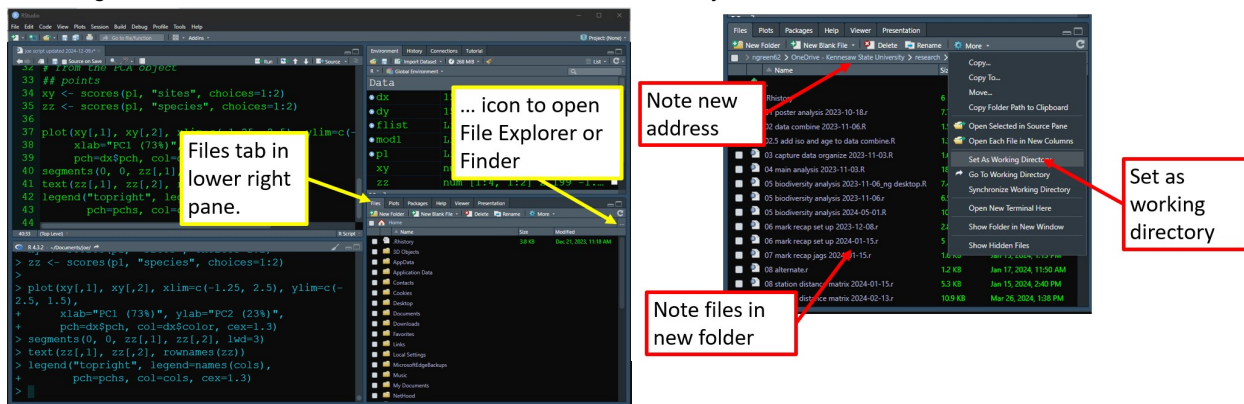
You can change it with `setwd()`:

```
# example; not run
setwd("C:/Users/ngreen62/Documents/biol 7410")
```

These commands are usually run first in a script, to facilitate data import.

In RStudio, you can set your working directory in the Files tab by first navigating to a folder, then clicking **More - Set as working directory**.

(a) To set your working directory, first click on the ... button (a) Then, select your folder and click More... Set as working directory.



What is the advantage of setting your working directory? R will look in the working directory **first** for any files, and put outputs there by default.

For example, the command `dir()` lists the contents of a folder. Without any arguments, it will automatically list the contents of the working directory.

```
# not run, but try on your machine:

# equivalent commands:
dir()
dir(getwd())

# for other folders, you need to specify the path:
dir("C:/Users/ngreen62/Documents/biol 7410/data")
```

When importing data, R will import from the working directory by file name, but requires a path for a file in any other folder.

```
# not run

# import from working directory:
my.data <- read.csv("example.csv")
```

```
# equivalent:
my.data <- read.csv(paste(getwd(), "example.csv", sep="/"))
# import from somewhere else:
my.data <- read.csv("C:/Users/ngreen62/Documents/example.csv")
```

If you open an R script with R or RStudio that is contained in a folder, that folder will be automatically set as the working directory. Clever use of the working directory can save you a lot of time. Sloppy use of the working directory (e.g., not specifying the working directory in each R or RStudio session) can lead to disaster and frustration.

4.2 Formatting data files

Before you try to import your data into R, take a few minutes to set yourself up for success.

4.2.1 Naming data files

Good file names are essential for a working biologist. Chances are you will work on many projects over the course of your career, each with its own collection of data files. Even if you store each project's data in a separate folder, it is a very good idea to give data files within those folders names that are descriptive. I have worked with many people who insist on using file names like "data.xlsx" and relying on their own memory or paper notes to tell them which file is which. Consider this list of particularly egregious file names:

- data.txt
- Data.txt
- data.set.csv
- analysis data.xlsx
- thesis data.xlsx
- DATA FINAL - copy.txt
- data current.revised.2019.txt.xlsx
- final thesis data revised revised final revised analysis.csv

We are all guilty of this kind of bad naming from time to time. What do all of the above have in common? They are vague. The names contain no meaningful information about what the files actually contain. Some of them are ambiguous: different operating systems or programs might read "data.txt" and "Data.txt" as the same file, or as different files!

Data file names should be short, simple, and descriptive. There are no hard and fast rules for naming data files so I'm not going to prescribe any. Instead, I'll share some examples and explain why these names are helpful. As you go along and get better at working with data you will probably develop your own habits, and that's okay. Just try to follow the guiding principles of short, simple, and descriptive.

Here are some example file names with explanations of their desirable properties:

sturgeon_growth_data_2017-02-21.txt

This name declares what the file contains (growth data for sturgeon), and the date it was created. Notice that the date is given as "YYYY-MM-DD". This format is useful because it automatically sorts chronologically. This file name separates its words with underscores rather than spaces or periods. This can prevent some occasional issues with programs misreading file names (white space and periods can have specific meanings in some contexts).

dat_combine_2020-07-13.csv

This name signifies that the file contains a "combined" dataset, and includes the date of its creation. This would be kept in a folder with the files that were combined to make it. Within a project I usually use prefixes on file names to help identify them: "dat" for datasets with response and explanatory variables; "met" for "metadata"; "geo" or "loc" for geospatial datasets and GIS layers; and "key" for lookup tables that relate dat and met files. You might develop your own conventions for types of data that occur in your work.

dat_vegcover_nrda_releaseversion_2021-05-03.csv

This name is similar to the last name, but includes some additional signifiers for the project (NRDA) and the fact that this is the final released version.

4.2.2 Setting up data files

Recently Broman and Woo (2018) reviewed good practices for storing data in spreadsheets. Their article is short and easy to read, and gave a few recommendations for data formatting:

- Be consistent
- Choose good names for things
- Write dates as YYYY-MM-DD
- No empty cells
- Put just one thing in a cell
- Make it a rectangle
- Create a data dictionary
- No calculations in the raw data files
- Do not use font color or highlighting as data

- Make backups
- Use data validation to avoid errors
- Save the data in plain text files

Most people who work with data, myself included, probably agree with their recommendations. Better, there is plenty of room for individual preferences in the exact implementation of each principle. You will save yourself a lot of headache if you follow their advice.

4.2.2.1 Variable names

Variable names, like data file names, should be brief but informative. There are few restrictions on what variable names can be in R. For simplicity, I recommend you use only lowercase letters, numbers, underscores (`_`), and periods (`.`) in variable names. Other symbols are either inconvenient to type, or can cause issues being parsed by R. Some common pitfalls include:

Leading numerals will cause an error if not enclosed in quotation marks:

```
# not run, but try on your machine:

x <- iris[1:5,]
names(x)[1] <- "3a"
x$3a
x$'3a'
x[, "3a"]
```

Numeral-only names will cause an error. They can also be misinterpreted as column numbers. Don't give your variables numeral-only names.

```
# not run, but try on your machine:

x <- iris[1:5,]
names(x)[1] <- "3"
x$3      # error
x$'3'    # works
x[,3]    # column number 3
x[, "3"] # column named "3"
```

Spaces in variable names will cause errors because R interprets a space as whitespace, not part of a name. If you insist on using spaces, enclose the variable name in quotation marks every time you use it. This is one of the few places where R cares about whitespace.

```
# not run, but try on your machine:
```

```
x <- iris[1:5,]  
names(x)[1] <- "a b"
```

```
# causes error:
```

```
x$a b
```

```
# works:
```

```
x$'a b'
```

Special characters in variable names can cause issues with parsing. Many symbols that you might think to use, such as hyphens (aka: dashes, -) or slashes / are also R operators. Again, try to avoid these issues by using only letters, numbers, underscores, and periods.

```
# not run, but try on your machine:
```

```
x <- iris[1:5,]  
names(x)[1] <- "a/b"  
b <- 10
```

```
# error:
```

```
x$a/b
```

```
# works:
```

```
x$'a/b'
```

```
names(x)[1] <- "surv%"
```

```
x
```

```
# error:
```

```
x$surv%
```

```
# works:
```

```
x$'surv%'
```

Uppercase letters in variable names work just fine, but they can be inconvenient. Because R is case sensitive, you need to type the variable exactly the same way each time. This can be tedious if you have several variables with similar names. In my code I prefer to use only lower case letters, so I don't ever have to remember how names are capitalized.

```
x <- iris[1:5,]  
names(x)[1] <- "sep.len" # convenient  
names(x)[2] <- "SEP.LEN" # slightly inconvenient  
names(x)[3] <- "Sep.Len" # more inconvenient
```



```
names(x)[4] <- "SeP.lEn" # just don't
x
##      sep.len SEP.LEN Sep.Len SeP.lEn Species
## 1      5.1      3.5      1.4      0.2   setosa
## 2      4.9      3.0      1.4      0.2   setosa
## 3      4.7      3.2      1.3      0.2   setosa
## 4      4.6      3.1      1.5      0.2   setosa
## 5      5.0      3.6      1.4      0.2   setosa
```

4.2.2.2 Data values

All data in R have a type: numeric, logical, character, etc., as we learned in Section 3.1. Within a vector, matrix, or array, all values must have the same type or they will be converted to character strings. Each variable in a data frame (the most common R data structure) is essentially a vector. This means that in your input file you should try to have one and only one type of data in each column. Otherwise, R will coerce the values into character strings, which may or may not be a problem. Putting only one type of value into each column will also help get your data into “wide” or “tidy” format (Wickham 2014).

Entering data should also include some measure of **quality assurance and quality control (QA/QC)**. This basically means making sure that your data are entered correctly, and are formatted correctly. Correct entry can be accomplished by direct comparison to original datasheets or lab notebooks. Cross-checking is a valuable tool here: have two people enter the data and compare their files.

Data formatting requires thinking about what kinds of data are being entered. For numbers, this means entering the correct values and an appropriate number of significant digits. For text values, this may mean spell-checking or settling on a standard set of possible values. The figure below shows some problems that a QA/QC step might fix:

Figure 4.4: Examples of bad data management that some simple QA/QC might fix.

	A	B	C
1	species	tsn	common
2	Peromyscus leucopus	180278	white-footed mouse
3	Peromyscus leucopus	180278	White foot mouse
4	Peromyscus leucodus	180278	White_footed_mouse
5	Microtus ochrogaster	180312	Prairie_vole
6	Microtus ochrogaster	180312	Prairie vole
7	...	...	...

Misspelled (points to 'leucopus' in row 3)

Similar but not identical (points to 'white-footed mouse', 'White foot mouse', and 'White_footed_mouse')

Similar but not identical (points to 'Prairie_vole' and 'Prairie vole')

4.3 Data import

4.3.1 Importing from text files

4.3.1.1 Importing from the working directory

Data for R should be stored in plain-text files. Two of the most popular options are tab-delimited files (**.txt**) and comma-delimited files (**.csv**). The “delimited” part means that some special character (like tab or comma) is used to separate values on a row of the file. Text files can be produced from Excel spreadsheets by going to **Save as**, selecting a folder, and then selecting “Text (Tab delimited)(*.txt)” or “CSV (Comma delimited)(*.csv)” from the dropdown menu for file type. Do **not** select “CSV UTF-8 (Comma delimited)(*.csv)” as this will cause issues in R.

Download the files `neon_taxa_2025-08-03.txt` and `neon_taxa_2025-08-03.csv`. Create a new folder for this exercise, save both files there, and set that folder as your R working directory before continuing.

Use `read.table()` to import a tab-delimited file, and save it to an object called `dx`. Notice that the filename must be spelled and capitalized correctly. Notice also that because the file is in your working directory, no folder path is needed. The argument `header=TRUE` tells R that the first row of the file contains column names, not data.

```
dx <- read.table("neon_taxa_2025-08-03.txt", header=TRUE)
```

Use `read.csv()` to import a comma-delimited file. Note how similar the syntax is to the first command, except that for `read.csv()` you don't need `header=TRUE` because this is the default (unlike `read.table()`).

```
dx <- read.csv("neon_taxa_2025-08-03.csv", header=TRUE)
```

While `read.table()` imports tab-delimited files by default, it can also import other types of files by setting the `sep` argument. Below is an example of using `read.table()` to import a CSV file.

```
dat <- read.table("neon_taxa_2025-08-03.csv",  
  header=TRUE, sep=",")
```

4.3.1.2 Importing from another folder

Working from the working directory is often quick and convenient, but if your data files are in different folders it can be awkward and time-consuming to continually change the working directory. Instead, you can skip the working directory and specify folders directly.

Imagine you have data files spread across 2 folders, “capture” and “geo”, which are both contained by another folder “_data”. You can combine folder names together in R with the `paste()` command. This

command concatenates, or sticks together, character strings. Separating folder names from file names also makes your code more portable. For example, if you move your data to another server or drive, you could only have to change one line of code (provided that you maintain the directory structure that it contains).

Consider this example. In most of my projects I call datasets from multiple locations (because I use many common data files that are too large to make copies of in project-specific directories), so this is how I structure data import:

```
# Not run; for illustration purposes only

# directory that contains everything
top.dir <- "K:/_data"

# folder that contains the capture data
cap.dir <- "capture"

# folder that contains geospatial data
geo.dir <- "geo"

# import something from capture
in.fold <- paste(top.dir, cap.dir, sep="/")
in.name <- "capture_data.csv"
in.path <- paste(in.fold, in.name, sep="/")
dx <- read.csv(in.path)

# import something from geospatial folder
in.fold <- paste(top.dir, cap.dir, sep="/")
in.name <- "site_coords.csv"
in.path <- paste(in.fold, in.name, sep="/")
cx <- read.csv(in.path)
```

4.3.1.3 Arguments to `read.table()` and `read.csv()`

The text file reading functions have a lot of options, but there are a few that I find very useful (in addition to header above).

4.3.1.3.1 `sep`

As seen in one of the examples above, the `sep` argument specifies what character separates values on each line of the data file. This argument has some useful default values. If you are using `read.table()` with a tab-delimited file, or `read.csv()` with a comma-delimited file, you don't need to set `sep`.

The default in `read.table()` reads any white space as a delimiter. This is fine if you have a tab-delimited file, but if the file contains a mix of spaces and tabs you might have some problems. For example, if entries are separated by tabs, but some values contain spaces. In those cases, specify the appropriate delimiter:

```
# not run:
# file is in working directory
in.name <- "neon_taxa_2025-08-03.txt"

# tab-delimited (sep="\t"):
dat <- read.table(in.name, header=TRUE, sep="\t")

# space-delimited (sep=" "):
dat <- read.table(in.name, header=TRUE, sep=" ")
```

The `sep` argument is usually necessary when using the “clipboard trick” (see Section 4.3.4)

4.3.1.3.2 row.names

When R imports a table of data, it will assign names to the rows. The default names are just the row numbers. Some data files will have their own row names. If you want to not import these names, you can set `row.names` argument to `FALSE` to get the default numbers. You can also specify names to assign upon import as a character vector.

```
# not run:

# make rownames "row_1", "row_2", and so on
use.rownames <- paste("row", 1:20, sep="_")
dat <- read.table(in.name, header=TRUE, row.names=use.rownames)
dat
```

4.3.1.3.3 skip

Some data files will contain rows of text or other information before the tabular data starts. The argument `skip` allows you to skip these lines. Without `skip`, R will try to read those rows of text as if they were data and fail to import. For example, 5 rows of preliminary text could be skipped with the argument `skip=5`.

If your data file has non-tabular header rows, the error message that `skip` will prevent can be frustrating if you don’t understand how R parses text files. What is happening is that R uses the number of elements in the first row to calculate the dimensions of the data frame that it makes upon import. For example, if a text file contains the line “These are data from Nobody (2020)” on the first line, followed by the tabular data, R would assume that the table contained 6 columns headed by “These”, “are”, “data”, and so on. But, if the table contained another number of columns, R would return an error.

Why does this issue vex so many new R users (and experienced R users)? Often we will have incomplete records in a data set, where a row is missing values in one or more columns. Sometimes blank values are not read correctly.

There are a few ways to deal with the errors caused by blanks. The first is to go back to the original datafile and put R's blank value, `NA`, in the blanks. You could also put in a value that is not likely to occur, such as `-9999` in variables that must be positive. This would make your file more readable, but without locking you in to using R. Either solution will ensure that missing values are not interpreted as extra white space. This method can be very time consuming for very large datasets, even in Excel (or especially in Excel). The best time to fix this issue is at data entry. Recall that having no empty cells is one of the recommendations of Broman and Woo (2018).

Another way is to save the file as a comma-delimited file, because commas are less ambiguous than white space for defining missing values. This method usually works, but might still create problems if there are multiple blank values per record or blanks at the ends of rows. The third option is to use the `fill` argument in the R import functions (see below).

4.3.1.3.4 `fill`

As described above, missing values can cause problems when importing data into R. One way to deal with missing values that occur at the end of a line in the data file is to use the `fill` argument. Setting `fill=TRUE` will automatically pad the data frame with blanks so that every row has the same number of values.

As convenient as `fill=TRUE` can be, you need to be careful when using it. R may add the wrong number of values, or values of the wrong type. One common mistake is for a blank character string (`" "`) to be put in place of a missing value `NA`. This is an important difference because `" "` and `NA` are treated very differently by many R functions.

4.3.1.3.5 `stringsAsFactors`

A **factor** is a variable in a dataset that defines group membership: control vs. treated, red vs. blue, gene 1 vs. gene 2 vs. gene 3, and so on. The different groups to which an observation can belong are called the **levels** of the factor. This is very useful for statistical modeling, but can be a pain in the neck when trying to work with data in R.

Functions `read.table()` and `read.csv()` have an argument to automatically convert character strings to factors for convenience: `stringsAsFactors`. This is a logical switch that will interpret character strings as levels of a factor (`stringsAsFactors=TRUE`) or as character strings (`stringsAsFactors=FALSE`). The default was `TRUE` until around version 4.0 or so, but now the default is the much more convenient `FALSE`.

The reason that factors can cause headaches in R is because although they look like character strings, they are really numeric codes and the text you see is just a label. If you try to use a factor as if it was a character string, you can get strange errors. My advice is to always import data with `stringsAsFactors=FALSE` and only convert to factor later if needed. Most statistical functions

will convert to factor for you automatically anyway. The only time when you may want to store data as a factor is when you have a very large dataset with many levels of a factor; in those situations, factors will use memory more efficiently.

If you get tired of typing `stringsAsFactors=TRUE` in your import commands, you can set an option at the beginning of your script that will change the default behavior of `read.table()` and `read.csv()`:

```
# not run
options(stringsAsFactors=FALSE)
```

Back when this option was `TRUE` by default I put this command at the start of every new R script to avoid problems with text strings being misinterpreted as factors.

4.3.2 Saving and importing R objects

Some R objects are not easily represented by tables. For example, the outputs of statistical methods are usually **lists**. Such objects can be saved using function `saveRDS()` and read into R using `readRDS()`. The example below will save an R data object to file `rds_test.rds` in your R working directory.

```
# fit a linear model and save its output
# as an R object (list)
mod1 <- lm(Petal.Width~Sepal.Width, data=iris)
mod1
##
## Call:
## lm(formula = Petal.Width ~ Sepal.Width, data = iris)
##
## Coefficients:
## (Intercept)  Sepal.Width
##      3.1569      -0.6403
saveRDS(mod1, "rds_test.rds")
```

```
# now read it back in:
b <- readRDS("rds_test.rds")
b
```

```
Call:
lm(formula = Petal.Width ~ Sepal.Width, data = iris)
```

```
Coefficients:
(Intercept)  Sepal.Width
      3.1569      -0.6403
```

There is a lazier way to accomplish this task using function `dput()`. This function saves a plain text representation of an object that can later be used to recreate the object (i.e., R code that can reproduce the object). The help file for `dput()` cautions that this is not a good method for transferring objects between R sessions and recommends using `saveRDS()` instead. That being said, I've used this method for years and have never had a problem. The procedure is to copy the output of `dput()` into your R script, and later assign that text to another object. Use `dput()` at your own risk.

```
a <- iris[1:6,]
dput(a)
```

```
structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4),
  Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9), Petal.Length = c(1.4,
  1.4, 1.3, 1.5, 1.4, 1.7), Petal.Width = c(0.2, 0.2, 0.2,
  0.2, 0.2, 0.4), Species = structure(c(1L, 1L, 1L, 1L, 1L,
  1L), levels = c("setosa", "versicolor", "virginica"), class = "factor")),
  6L), class = "data.frame")
```

4.3.3 Importing from saved workspaces

Everything you do in R takes place within the **workspace**. This can be thought of as a sandbox where all of your data and functions reside. If you want to save this entire environment and continue working later you can do this by saving the workspace with `save.image()` (see Section 4.4.3).

Workspaces can be loaded using function `load()`. The example below loads workspace `example.RData` from your home directory. Notice that this command will load every object from the saved workspace into your current workspace. This might cause problems if the saved workspace and current workspace have objects with the same names.

```
# not run:
load("example.RData")
```

When loading a workspace, it is sometimes more useful to import it into a new **environment**. An environment is like a container within your workspace that contain objects that aren't visible to the outer workspace. In other words, you can import objects from that workspace without putting all of its objects into your current workspace. I use this method a lot for working with large sets of model outputs. The typical workflow is:

1. Run an analysis or set of simulations in multiple R instances and save the results of each in its own R workspace.
2. After simulations are complete, import the results of each simulation from its individual workspace into a new environment.
3. Copy the objects from the environment containing the loaded workspace to the main workspace.
4. Delete the environment containing the loaded workspace.

5. Repeat until all results are imported.

Here is an example of importing individual objects from a workspace into a new environment, then copying that object into the current workspace:

```
# not run:
rm(a) # will return error if a does not already exist

# set up a new environment
temp <- new.env()

# load the previously saved workspace
# into the new environment
load("example.RData", env=temp)

# access an object called "a" from the
# imported workspace. Note that a will
# not be available from outside temp.
a <- temp$a

# remove the container; create a new one
# for the next import or step of the loop
rm(temp)

# inspect your imported object
a
```

4.3.4 Importing special cases

4.3.4.1 The clipboard trick

Sometimes it can be convenient to copy and paste data directly from a spreadsheet into R. This is fine for quick or one-off data imports, but not recommended for routine use because it is tedious to reproduce and error-prone. The procedure is as follows:

1. In the R console, type this command but do NOT run it: `a <- read.table("clipboard", header=TRUE, sep="\t")`
2. In Excel, highlight your data and copy it to the clipboard using Ctrl-C (or Cmd-C on a Mac).
3. Click in the R console to move your focus there. In R, hit the ENTER key.
4. The command you typed—but did not execute—in step 1 will read the table from the clipboard.

Note that if you try to copy/paste the `read.table("clipboard", ...)` command into the R console, you will then have that command in your clipboard and not your data.

4.3.4.2 Entering data using `scan()`

The `scan()` function lets you enter values one-by-one, separated by pressing ENTER. You can use this just like typing values into Excel. Or, you can copy and paste a single column of values from Excel. The latter case works because R interprets the values within a column as being separated by a line return (i.e., pressing ENTER).

Type the name of an object to which data will be saved, the assignment operator `<-`, `scan()`, then ENTER, then values (each followed by ENTER), then hit ENTER twice after the last value. This is obviously inefficient for large datasets, but it's a nice trick to be aware of.

```
# Not run, but try on your machine.

a <- scan()
1
2
3
4
5

a
```

You can use this function to enter values from a column in Excel, highlight and copy the column. Then go to R, enter the command below, and press ENTER. Then paste your values and press ENTER again.

4.3.4.3 Entering data directly using `c()`

Function `c()` combines its arguments as a vector. As with all functions, arguments must be separated by commas. Arguments can be on the same line or on different lines.

```
c(1:3, 9:5)
## [1] 1 2 3 9 8 7 6 5

a <- c(1,
      3,
      5
    )
a
## [1] 1 3 5
```

Like `scan()`, data can be entered directly into the console using `c()`. Also like `scan()`, this is comically inefficient but can be useful for small numbers of values.

4.4 Data export

4.4.1 Exporting text files

Everything you do in R takes place within the “workspace”. You can save entire workspaces, but in most situations it is more useful to write out data or results as delimited text files. I’ve found that comma-delimited files are slightly less likely to have issues because they use a character (,) rather than whitespace to separate values. So, for the rest of this course we’ll use CSV files.

Tab-delimited files are written out using `write.table()`. In the example below, the first argument `dat` is the object from the R workspace that you want to write out. The argument `sep` works the same way as in `read.table()`. The example below will save a file called `mydata.txt` in working directory. Option `sep="\t"` can be provided to ensure that the file is written as a tab-delimited file and not a space-delimited file.

```
# not run:
dat <- iris[1:6,]
write.table(dat, "mydata.txt", sep="\t")
```

Comma-delimited files are written out by `write.csv()`. The syntax is similar to `write.table()`. Note that the file extension must be “.csv” instead of “.txt”. If you save a file with the wrong extension, then it may be corrupt or unusable (or, R might return an error).

```
# not run:
write.csv(dat, "mydata.csv")
```

Most of the time you will not be saving files to your home directory, but to another folder. The folder and the file name must both be specified to save somewhere else. It is usually a good idea to define the file name and file destination separately. This way if you re-use your code on another machine, or a different folder, you only need to update the folder address once.

```
# not run:

# do this:
out.dir <- "C:/_data"
out.name <- "mydata.csv"
write.csv(dat, paste(out.dir, out.name, sep="/"))

# or this:
out.dir <- "C:/_data"
out.name <- "mydata.csv"
out.path <- paste(out.dir, out.name, sep="/")
```

```
write.csv(dat, out.path)

# not this:
write.csv(dat, "C:/_data/mydata.csv")
```

The last example is problematic because the output file name contains both the destination folder and the filename itself. If you try to run your code on another machine, it will take longer to update the code with a new folder address. This method is also problematic because exporting multiple files will require typing the folder address multiple times. Consider this example:

```
# not run:

dat1 <- iris[1:6,]
dat2 <- iris[7:12,]

write.csv(dat1, "C:/_data/mydata1.csv")
write.csv(dat2, "C:/_data/mydata2.csv")
```

Running that code on a new machine, or modifying it for a new project in a different folder, requires changing the folder address twice. The better way is to define the folder name only once:

```
# not run:

out.dir <- "C:/_data"
write.csv(dat1, paste(out.dir, "mydata1.csv", sep="/"))
write.csv(dat2, paste(out.dir, "mydata2.csv", sep="/"))
```

See the difference? The revised code is much more portable and reusable. This illustrates a good practice for programming in general (not just R) called **“Single Point of Definition”**. Defining variables or objects once and only once makes your code more robust to changes and easier to maintain or reuse. The example below shows a more compact way to export the files `mydata1.csv` and `mydata2.csv`, with even fewer points of definition for file names:

```
# not run:

out.dir <- "C:/_data"
out.names <- paste0("mydata", 1:2, ".csv")
out.files <- paste(out.dir, out.names, sep="/")
write.csv(dat1, out.files[1])
write.csv(dat2, out.files[2])
```

One handy trick is to auto-generate an output filename that includes the current date. This can really help keep multiple versions of analyses or model runs organized. The `paste0()` command pastes its arguments together as they are, with no separator between them (equivalent to `paste(..., sep="")`). Function `Sys.Date()` prints the current date (according to your machine) in YYYY-MM-DD format.

```
# not run:

out.dir <- "C:/_data"
out.name <- paste0("mydata", "_", Sys.Date(), ".csv")
write.csv(dat, paste(out.dir, out.name, sep="/"))
```

4.4.2 Saving R objects

Some data do not fit easily into tables. Individual R objects can be saved to an “.rds” file using the function `saveRDS()`. This function saves a single object that can later be loaded into another R session with function `readRDS()`.

```
# not run:

x <- t.test(rnorm(100))

# save to working directory:
saveRDS(x, "zebra.rds")

# load the file:
z <- readRDS("zebra.rds")
```

4.4.3 Saving workspaces

Entire R workspaces or parts of workspaces can be saved using the function `save.image()`. The command by itself will save the entire current workspace with a generic name to the working directory. It is more useful to save the workspace under a particular name and/or only save specific objects.

```
# not run:

# save entire workspace with generic name (avoid):
save.image()

# save entire workspace with specific name (better):
save.image(file="test1.RData")
```

```
# save specific objects (best):
x <- rnorm(100)
z <- t.test(x)
save(list=c("x", "z"), file="testworkspace.RData")
```

4.4.4 Write text to a file (special cases)

Sometimes you need to write out data that do not fit easily into tables. An example might be the results of a single *t*-test or ANOVA. This is also sometimes necessary when working with programs outside of R. For example, the program JAGS requires a text file that defines the model to be fit (see Chapter 30). This text file can be generated by R, which makes your workflow simpler because you don't need to use a separate program to store the outputs.

The function to write text files with arbitrary content (i.e., not tables) is `sink()`. Functions `print()` and `cat()` are also useful in conjunction with `sink()`.

- `sink()` works in pairs. The first command defines a file to put text into. The second command finishes the file and writes it.
- `print()` prints its arguments as-is. It should produce the same text as running its argument in the R console.
- `cat()` stands for “concatenate and print”. Use this to combine expressions and then print them.

```
# not run, but try it on your machine

my.result <- t.test(rnorm(50))
sink("C:/_data/myresult.txt")
cat("t-test results:", "\n") #\n = new line
print(my.result)
sink()
```

4.4.5 Exporting graphics

Finally, at some point you will need to export R graphics. In the base R GUI, you can right-click your image and select “Copy as bitmap”, and then paste it somewhere as a .jpeg. In RStudio, you can right-click and “Save image as...”. However, when preparing presentation or publication quality graphs, it helps to exert more control over the final figure with graphics functions.

R can write images in a variety of formats, including .jpeg, .png, .tiff, and so on. The basic process is:

1. Start a new graphics device using `jpeg()` or `.png` or whichever function is appropriate. This function will include the target path as an argument.

2. Run the code to make your plot.
3. Use `dev.off()` to tell R that your plot is done, and to go ahead and render and save it.

Here is an example:

```
# not run, but try on your machine
# will put a figure in your working directory

jpeg("example_fig.jpg",           # file name
      width=8, height=5, units="in", # dimensions
      res=300)                     # resolution
hist(rnorm(100))                   # make plot
dev.off()                          # finish figure
```

Chapter 5

Data manipulation with R

In `#sec-r` we introduced the basics of the R language and program environment. In this module, we'll go a few steps further and get into the finer details of **data manipulation**: the practice of arranging, sorting, cleaning, and modifying data—and the objects that contain data. Data manipulation is an important R skill because real datasets usually need some fixing before they can be properly analyzed. Data manipulation is also an essential part of R programming for tasks other than data analysis. For example, [simulation modeling](#) or document production (e.g., this website!).

5.1 Making values

The typical analysis workflow in R involves importing data from text files or other sources. Sometimes, however, it is necessary to make values within R. This can be for a variety of reasons:

1. Producing test data sets with desired distributions
2. Making sets of indices or look-up values to use in other functions
3. Modifying existing datasets
4. Running simulations
5. And many more...

This section demonstrates some methods for producing values—i.e., data—within R.

5.1.1 Make arbitrary values with the `c()` function

The function `c()` combines its arguments as a **vector**. As with all functions, arguments must be separated by commas and can be on different lines. This method can be used to make vectors with any set of values.

```
c(1:3, 9:5)
```

```
[1] 1 2 3 9 8 7 6 5
```

```
a <- c(1,  
      3,  
      5  
      )  
a
```

```
[1] 1 3 5
```

Like `scan()`, this is an inefficient way to make vectors but can be useful for small numbers of values. Or, for sets of values that can't be defined algorithmically.

5.1.2 Generating sequences

5.1.2.1 Consecutive values with the colon `:` operator

Consecutive values can be generated using the colon operator `:`. A command like `X:Y` will generate consecutive values from X to Y in increments of 1 or -1. If $X < Y$, the increment is 1; if $X > Y$, the increment is -1. The usual use case is to make sets of consecutive integers.

```
# Simple examples:  
1:10
```

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

```
5:2
```

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

```
-1:10
```

```
[1] -1 0 1 2 3 4 5 6 7 8 9 10
```

```
10:-1
```

```
[1] 10 9 8 7 6 5 4 3 2 1 0 -1
```

Sometimes you may need to use parentheses to get the intended values because of the order in which R runs operators (aka: “operator precedence”).


```
# effect of parentheses (operator precedence):  
a <- 2:6  
length(a)
```

```
[1] 5
```

```
1:2*length(a)
```

```
[1] 5 10
```

```
1:(2*length(a))
```

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

X and Y don't have to literally be numbers; they can be variables or R expressions that evaluate to a number. Below are some examples:

```
a <- 2:6  
# length(a) evaluates to a number (5)  
length(a):10
```

```
[1] 5 6 7 8 9 10
```

Most people use `:` to produce sets of integers, but it can also make non-integers. The values will be separated by 1 or -1. The first value will be the value before the `:`. The last value might not be the value after the `:`. It will be the value up to or before the second number along the sequence from the first number. If the value before the `:` is an integer, then the sequence will consist of integers. Note that if you want a sequence that includes non-integers, it might be safer to use `seq()` (see below).

```
2.3:5
```

```
[1] 2.3 3.3 4.3
```

```
5:2.3
```

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

```
2.7:6.1
```

```
[1] 2.7 3.7 4.7 5.7
```

5.1.2.2 Regular sequences with the seq() function

Function `seq()` generates regular sequences of a given length (`length=`), or by an interval (`by=`). The command must be supplied with a length or an interval, but not both. If you specify a length, R will calculate the intervals for you. If you specify an interval, R will calculate the length for you. If you specify a length and an interval, R will return an error.

```
seq(0, 20, by=0.5)
```

```
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0
[16] 7.5 8.0 8.5 9.0 9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5
[31] 15.0 15.5 16.0 16.5 17.0 17.5 18.0 18.5 19.0 19.5 20.0
```

```
seq(-pi, pi, by=0.1)
```

```
[1] -3.14159265 -3.04159265 -2.94159265 -2.84159265 -2.74159265 -2.64159265
[7] -2.54159265 -2.44159265 -2.34159265 -2.24159265 -2.14159265 -2.04159265
[13] -1.94159265 -1.84159265 -1.74159265 -1.64159265 -1.54159265 -1.44159265
[19] -1.34159265 -1.24159265 -1.14159265 -1.04159265 -0.94159265 -0.84159265
[25] -0.74159265 -0.64159265 -0.54159265 -0.44159265 -0.34159265 -0.24159265
[31] -0.14159265 -0.04159265 0.05840735 0.15840735 0.25840735 0.35840735
[37] 0.45840735 0.55840735 0.65840735 0.75840735 0.85840735 0.95840735
[43] 1.05840735 1.15840735 1.25840735 1.35840735 1.45840735 1.55840735
[49] 1.65840735 1.75840735 1.85840735 1.95840735 2.05840735 2.15840735
[55] 2.25840735 2.35840735 2.45840735 2.55840735 2.65840735 2.75840735
[61] 2.85840735 2.95840735 3.05840735
```

```
seq(-1, 1, length=20)
```

```
[1] -1.00000000 -0.89473684 -0.78947368 -0.68421053 -0.57894737 -0.47368421
[7] -0.36842105 -0.26315789 -0.15789474 -0.05263158 0.05263158 0.15789474
[13] 0.26315789 0.36842105 0.47368421 0.57894737 0.68421053 0.78947368
[19] 0.89473684 1.00000000
```

```
# returns an error:
## not run:
seq(1, 20, by=3, length=3)
```

If you want the values to go from greater to smaller, the `by` argument must be negative.

```
# not run, but try on your machine:  
# returns an error  
seq(-1, -10, by=1)
```

```
# works:  
seq(-1, -10, by=-1)
```

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

```
# extra steps, but could also work:  
rev(seq(-10, -1, by=1))
```

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

```
-1*(seq(1, 10, by=1))
```

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

5.1.2.3 Repeated values with the rep() function

The function `rep()` creates a sequence of repeated elements. It can be used in several ways.

One way is to specify the number of times its first argument gets repeated. Notice that in the second command `rep(1:3, 5)`, the entire first argument `1 2 3` is repeated 5 times.

```
rep(2, times=5)
```

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

```
rep(1:3, times=5)
```

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

Another way is to repeat each element of its input a certain number of times each:

```
rep(1:3, each=5)
```

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

The arguments `times` and `each` can be combined. Notice that `each` takes precedence over `times`.

```
rep(1:3, each=4, times=2)
```

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

```
rep(1:3, each=2, times=4)
```

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

5.1.2.4 The recycling rule

One quirk of R sequences and vectors is the **recycling rule**. Values in one vector can be “recycled” to match up with values in another vector. This rule is **only** invoked if the length of the longer vector is a multiple of the length of the shorter vector. Otherwise you will get an error. Consider the following example:

```
my.df <- data.frame(x=1:12)

# add a value with 3 unique values
my.df$x3 <- 1:3

# add a value with 4 unique values
my.df$x4 <- 1:4

# add a value with 2 random values:
my.df$x2 <- runif(2)

my.df
```

	x	x3	x4	x2
1	1	1	1	0.7366545
2	2	2	2	0.5808679
3	3	3	3	0.7366545
4	4	1	4	0.5808679
5	5	2	1	0.7366545
6	6	3	2	0.5808679
7	7	1	3	0.7366545
8	8	2	4	0.5808679
9	9	3	1	0.7366545
10	10	1	2	0.5808679
11	11	2	3	0.7366545
12	12	3	4	0.5808679

Trying to add a variable to `my.df` with a number of values that is not a factor of 12—like 5, 7, 8, 9, 10, or 11—will usually return an error.

```
# not run  
my.df$x7 <- 1:7
```

5.1.3 Random values

R can generate many types of **random values**. This makes sense because R was designed for statistical analysis, and randomness is a key feature of statistics. What most people don't realize is that random values in R (or in most programs) are not really random—they are **pseudorandom**. This means that while the values appear random, and can pass many statistical tests for randomness, they are calculated deterministically from the state of your computer. So, if you want your analysis or script to be fully reproducible, you need to set the state so that the same pseudorandom values are always returned. Otherwise, you will get different results every time.

5.1.3.1 The random number seed

The random number seed can be set with function `set.seed()`. Setting the seed defines the initial state of R's pseudo-random number generator.

```
# different results each time:  
runif(3)  
## [1] 0.6297984 0.6509338 0.8286519  
runif(3)  
## [1] 0.9541006 0.3668470 0.1027333  
runif(3)  
## [1] 0.8495104 0.3247613 0.5685642  
  
# same results each time:  
set.seed(42)  
runif(3)  
## [1] 0.9148060 0.9370754 0.2861395  
  
set.seed(42)  
runif(3)  
## [1] 0.9148060 0.9370754 0.2861395  
  
set.seed(42)  
runif(3)  
## [1] 0.9148060 0.9370754 0.2861395
```

5.1.3.2 Random values from a set with the sample() function

The function `sample()` pulls random values from a set of values. This can be done with or without replacement. If you try to pull too many values from a set without replacement R will return an error. The default argument for `replace=` is `FALSE`.

```
set.seed(456)
sample(1:10, 5, replace=FALSE)
## [1] 5 3 6 10 4
sample(1:10, 5, replace=TRUE)
## [1] 9 10 9 10 3
```

```
# returns an error
# not run:
sample(1:10, 20)
```

The command above will return an error because it basically asks R to pull 20 cards from a deck of 10, without putting any cards back. The command below, with `replace=TRUE`, puts the cards back in the deck after each draw so that 20 cards can be drawn.

```
# this works:
sample(1:10, 20, replace=TRUE)
## [1] 7 8 5 4 9 1 6 8 5 9 2 4 10 4 10 1 10 8 9 4
```

You can use `sample()` with values other than numbers. The function will pull values from the vector in its first argument.

```
x1 <- c("heads", "tails")
sample(x1, 10, replace=TRUE)
## [1] "tails" "heads" "tails" "tails" "tails" "heads" "tails" "heads" "head"
## [10] "tails"

x2 <- c(2.3, pi, -3)
sample(x2, 10, replace=TRUE)
## [1] 2.300000 2.300000 2.300000 3.141593 3.141593 -3.000000 -3.000000
## [8] 3.141593 3.141593 2.300000
```

By default, every element in the set has an equal probability of being chosen. You can alter this with the argument `prob`. This argument must have a probability for each element of the set from which value are chosen, and those probabilities must add up to 1.

```

# a is an unfair coin where heads
# comes up 75% of the time.
# b is a fair coin.
x <- c("heads", "tails")
probs <- c(0.75, 0.25)
a <- sample(x, 100, replace=TRUE)
b <- sample(x, 100, replace=TRUE, prob=probs)

# compare:
table(a)
## a
## heads tails
##      53     47
table(b)
## b
## heads tails
##      74     26

```

5.1.3.3 Random values from a distribution

One of the strengths of R is its ease of working with probability distributions. We will explore probability distributions more in Chapter 9, but for now just understand that a probability distribution is a function that describes how likely different values of a random variable are. In R, random values from a probability distribution can be drawn using a function with a name like `r_()`, where `_` is the (abbreviated) name of the distribution.

The syntax of the functions for working with distributions in R is very consistent. For random values, the first argument is always the sample size (`n`), and the next few arguments are the distributional parameters of the distribution. These arguments can be provided with or without their names. The default order of the arguments is shown on the help page for each function (e.g., `?rnorm`). Below are some examples of random draws from different distributions, displayed as histograms.

```

# sample size
N <- 1000

# draw random distributions
## normal with mean 0 and SD = 1
a1 <- rnorm(N)

## normal with mean 25 and SD = 100
a2 <- rnorm(N, 25, 100)

```

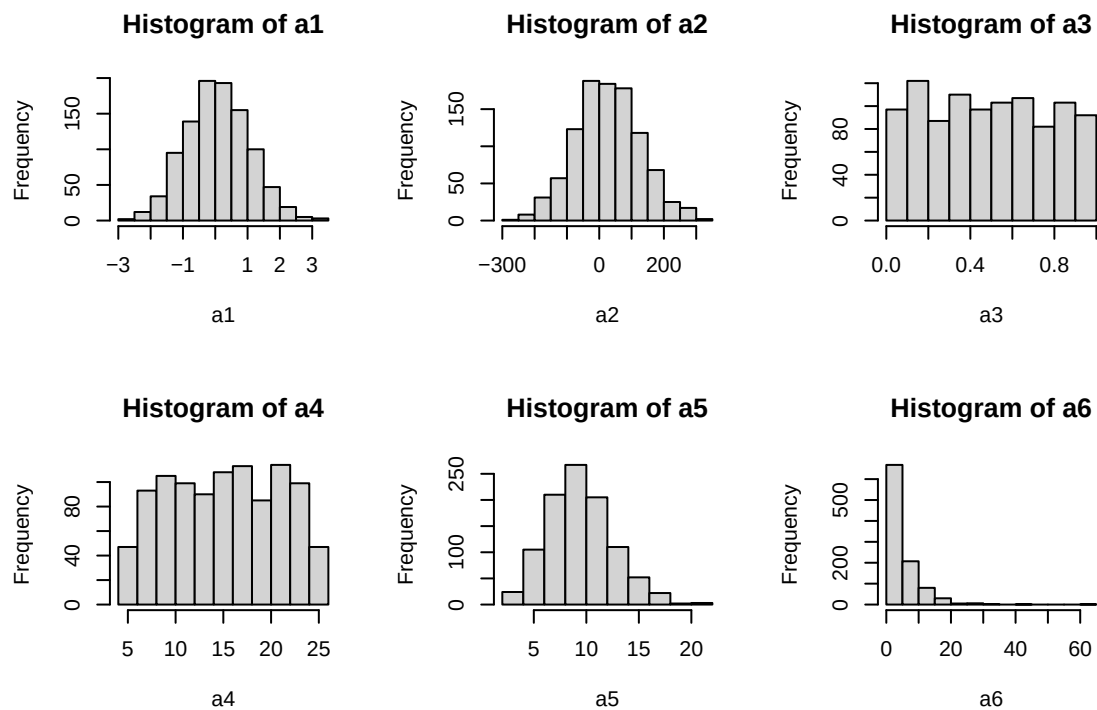
```
## uniform in [0, 1]
a3 <- runif(N)

## uniform in [5, 25]
a4 <- runif(N, 5, 25)

## Poisson with lambda = 10
a5 <- rpois(N, 10)

## lognormal with logmu = 1.2 and logsd = 2
a6 <- rlnorm(N, 1.2, 0.9)

# produce histogram of each distribution above
par(mfrow=c(2,3))
hist(a1)
hist(a2)
hist(a3)
hist(a4)
hist(a5)
hist(a6)
```




```
# reset graphical parameters
# so the next figure is 1 x 1 panel
par(mfrow=c(1,1))
```

5.2 Selecting data with square brackets

One of the most common data manipulation tasks is selecting, or **extracting**, from your data. The most common way of selecting subsets of an object (aka: extracting) is the R **bracket notation**. This refers to the symbols used, `[ ]`. These are also sometimes called **square brackets**, to distinguish them from **curly brackets**, `{ }`. The latter are more properly called **braces** and have a different purpose in R.

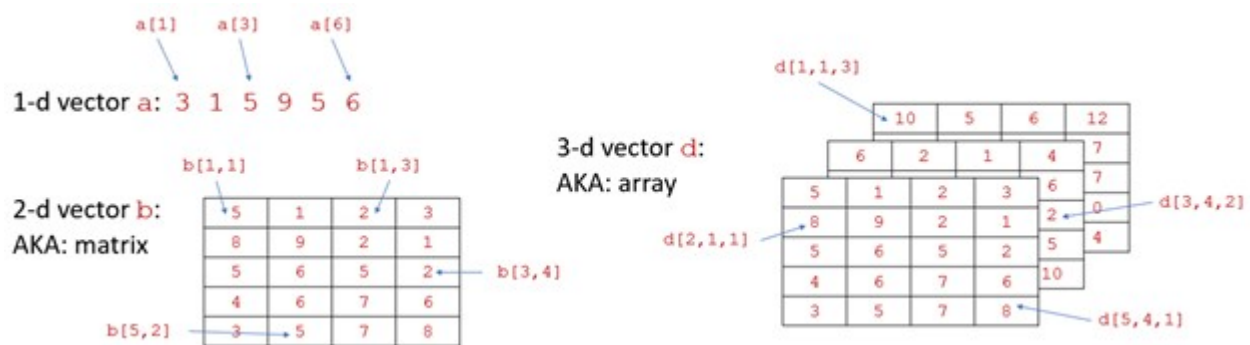
Brackets tell R that you want a part of an object. Which part is determined either by a name, or an **index** inside the brackets. Indices should usually be supplied as numbers, but other types can be used. For objects with more than 1 dimension, you need to supply as many indices, or vectors of indices, as the object has dimensions. Within brackets, vectors of indices are separated by commas.

Brackets can also be used to exclude certain elements by using the negative sign, but this can be tricky. We'll explore how negative indices work below.

5.2.1 Basics of brackets

The most direct way to work with most R objects is the **bracket notation**. Square brackets `[ ]` allow you to access parts of objects with a simple syntax to specify position within the object.

Figure 5.1: Illustration of how bracket notation accesses elements of vector-like objects in R.



Positions within an object are accessed using bracket notation. For example, `a[3]` denotes the third element of vector `a`. For vectors with > 1 dimension, commas are used to separate dimensions. The first dimension is rows, the second dimension is columns, and the third dimension is layers (maybe?). The general term for the dimensions of an object rows, columns, etc., is **margins**. So, the rows are the first margin, the columns are the second margin, and so on.

The values inside the brackets that specify rows or columns or whatever are called **indices**. In R, *indices start at 1* (this is unlike some other languages, like Python). The trickiest part of bracket notation is that you must specify indices for each margin. If you don't, you will either get an error (bad) or something unexpected (worse). Indices can be a single value, a vector of values, or left blank to request all values.

Figure 5.2: In R bracket notation, the comma separates margins and allows you to select an entire margin worth of values.

b[1,] selects row 1, all columns	b[, 2] selects column 2, all rows	b[3, 2] selects row 3, column 2																																																												
<table><tr><td>[1, 1]</td><td>[1, 2]</td><td>[1, 3]</td><td>[1, 4]</td></tr><tr><td>[2, 1]</td><td>[2, 2]</td><td>[2, 3]</td><td>[2, 4]</td></tr><tr><td>[3, 1]</td><td>[3, 2]</td><td>[3, 3]</td><td>[3, 4]</td></tr><tr><td>[4, 1]</td><td>[4, 2]</td><td>[4, 3]</td><td>[4, 4]</td></tr><tr><td>[5, 1]</td><td>[5, 2]</td><td>[5, 3]</td><td>[5, 4]</td></tr></table>	[1, 1]	[1, 2]	[1, 3]	[1, 4]	[2, 1]	[2, 2]	[2, 3]	[2, 4]	[3, 1]	[3, 2]	[3, 3]	[3, 4]	[4, 1]	[4, 2]	[4, 3]	[4, 4]	[5, 1]	[5, 2]	[5, 3]	[5, 4]	<table><tr><td>[1, 1]</td><td>[1, 2]</td><td>[1, 3]</td><td>[1, 4]</td></tr><tr><td>[2, 1]</td><td>[2, 2]</td><td>[2, 3]</td><td>[2, 4]</td></tr><tr><td>[3, 1]</td><td>[3, 2]</td><td>[3, 3]</td><td>[3, 4]</td></tr><tr><td>[4, 1]</td><td>[4, 2]</td><td>[4, 3]</td><td>[4, 4]</td></tr><tr><td>[5, 1]</td><td>[5, 2]</td><td>[5, 3]</td><td>[5, 4]</td></tr></table>	[1, 1]	[1, 2]	[1, 3]	[1, 4]	[2, 1]	[2, 2]	[2, 3]	[2, 4]	[3, 1]	[3, 2]	[3, 3]	[3, 4]	[4, 1]	[4, 2]	[4, 3]	[4, 4]	[5, 1]	[5, 2]	[5, 3]	[5, 4]	<table><tr><td>[1, 1]</td><td>[1, 2]</td><td>[1, 3]</td><td>[1, 4]</td></tr><tr><td>[2, 1]</td><td>[2, 2]</td><td>[2, 3]</td><td>[2, 4]</td></tr><tr><td>[3, 1]</td><td>[3, 2]</td><td>[3, 3]</td><td>[3, 4]</td></tr><tr><td>[4, 1]</td><td>[4, 2]</td><td>[4, 3]</td><td>[4, 4]</td></tr><tr><td>[5, 1]</td><td>[5, 2]</td><td>[5, 3]</td><td>[5, 4]</td></tr></table>	[1, 1]	[1, 2]	[1, 3]	[1, 4]	[2, 1]	[2, 2]	[2, 3]	[2, 4]	[3, 1]	[3, 2]	[3, 3]	[3, 4]	[4, 1]	[4, 2]	[4, 3]	[4, 4]	[5, 1]	[5, 2]	[5, 3]	[5, 4]
[1, 1]	[1, 2]	[1, 3]	[1, 4]																																																											
[2, 1]	[2, 2]	[2, 3]	[2, 4]																																																											
[3, 1]	[3, 2]	[3, 3]	[3, 4]																																																											
[4, 1]	[4, 2]	[4, 3]	[4, 4]																																																											
[5, 1]	[5, 2]	[5, 3]	[5, 4]																																																											
[1, 1]	[1, 2]	[1, 3]	[1, 4]																																																											
[2, 1]	[2, 2]	[2, 3]	[2, 4]																																																											
[3, 1]	[3, 2]	[3, 3]	[3, 4]																																																											
[4, 1]	[4, 2]	[4, 3]	[4, 4]																																																											
[5, 1]	[5, 2]	[5, 3]	[5, 4]																																																											
[1, 1]	[1, 2]	[1, 3]	[1, 4]																																																											
[2, 1]	[2, 2]	[2, 3]	[2, 4]																																																											
[3, 1]	[3, 2]	[3, 3]	[3, 4]																																																											
[4, 1]	[4, 2]	[4, 3]	[4, 4]																																																											
[5, 1]	[5, 2]	[5, 3]	[5, 4]																																																											

5.2.1.1 Selecting from a vector

Vectors have 1 dimension, so you must specify a single vector of indices. Notice that the brackets can be empty, in which case all elements can be returned. If the vector supplied is of length 0, then no elements will be returned. When a vector of indices with > 0 elements is specified, R will return the elements of the vector in the order requested. For example, `a[c(1, 3, 5)]` and `a[c(3, 5, 1)]` will return the same values but in different order.

```
a <- 11:20
# all elements returned
a[]
## [1] 11 12 13 14 15 16 17 18 19 20

# second element
a[2]
## [1] 12

# elements 3, 4, 5, 6, and 7
a[3:7]
## [1] 13 14 15 16 17

# error (not run)
# a[3:7,2]

# correct version of previous command
a[c(3:7,2)]
## [1] 13 14 15 16 17 12
```

```
# notice different ordering from previous command
a[c(2:7)]
## [1] 12 13 14 15 16 17
```

5.2.1.2 Selecting from a matrix

Matrices have 2 dimensions, so you must specify two dimension's worth of indices. If you want all of one dimension (e.g., rows 1 to 3 and all columns), you can leave the spot for the index blank. Separate the dimensions using a comma. Positive indices select elements; negative elements remove elements.

I'll reiterate this because it is a common mistake: you must specify all dimensions using a comma. Failing to do so will result in an error (bad) or unexpected results (worse).

Remember: in R, indices start at 1. In some languages such as Python indices start at 0. R is not Python.

```
a <- matrix(1:12, nrow=3, byrow=TRUE)
a
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12

# value in row 1, column 2
a[1,2]
## [1] 2

# values in rows 1-3 and columns 1-2
a[1:3, 1:2]
##      [,1] [,2]
## [1,]    1    2
## [2,]    5    6
## [3,]    9   10

# rows 1 and 2, all columns
a[1:2,]
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8

# columns 1 and 2, all rows
```

```
a[,1:2]
##           [,1] [,2]
## [1,]         1    2
## [2,]         5    6
## [3,]         9   10
```

Negative indices remove elements, or return all positive indices.

```
# rows except for row 1
a[-1,]
```

```
           [,1] [,2] [,3] [,4]
[1,]         5    6    7    8
[2,]         9   10   11   12
```

Removing multiple elements with `-` can be a bit tricky. All elements of the vector of things to remove must be negative. In the first example below, R returns an error because you request elements -1, 0, 1, and 2. Element -1 does not exist, and you don't want elements 1 and 2!

```
# try to request rows except for
# rows 1 and 2
# not run (error):
a[-1:2,]
```

There are several correct ways to get rows other than 1 and 2. Notice that all of them result in indices with a 1-, a -2, and no positive values. The first option is probably the best for most situations.

```
a[-c(1,2),]
## [1]  9 10 11 12
a[-1*1:2,]
## [1]  9 10 11 12
a[-2:0,] # includes -2, -1, and 0,
## [1]  9 10 11 12
# but 0 doesn't do anything
```

Notice that all three of the commands above return a vector, not a matrix. This is because a single row or column of a matrix simplifies to a vector. If you want to keep your result in matrix format, you can specify this using the argument `drop=FALSE` after the last dimension and a comma:

```
a[-c(1,2),,drop=FALSE]
```

```
           [,1] [,2] [,3] [,4]
[1,]         9   10   11   12
```

5.2.1.3 Selecting from an array

Selecting from an array is a lot like selecting from a matrix, only with one more dimension. Consider an array with 3 dimensions: rows, columns, and “layers”¹.

```
Ar <- array(1:24, dim=c(4,2,3))
Ar
```

```
, , 1
```

	[,1]	[,2]
[1,]	1	5
[2,]	2	6
[3,]	3	7
[4,]	4	8

```
, , 2
```

	[,1]	[,2]
[1,]	9	13
[2,]	10	14
[3,]	11	15
[4,]	12	16

```
, , 3
```

	[,1]	[,2]
[1,]	17	21
[2,]	18	22
[3,]	19	23
[4,]	20	24

Each element in any dimension of the array is a 2-dimensional slice of the array—i.e., a matrix:

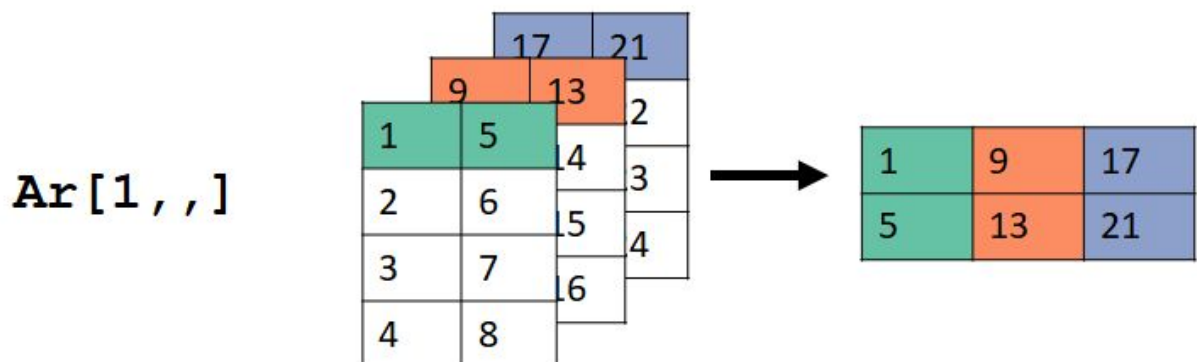
```
Ar[1,,]
##      [,1] [,2] [,3]
## [1,]    1    9   17
## [2,]    5   13   21
Ar[,2,]
##      [,1] [,2] [,3]
## [1,]    5   13   21
```

¹ I don't know if there is an actual name for this, but I like to call them layers.

```
## [2,]      6     14     22
## [3,]      7     15     23
## [4,]      8     16     24
Ar[, ,3]
##      [,1] [,2]
## [1,]     17     21
## [2,]     18     22
## [3,]     19     23
## [4,]     20     24
```

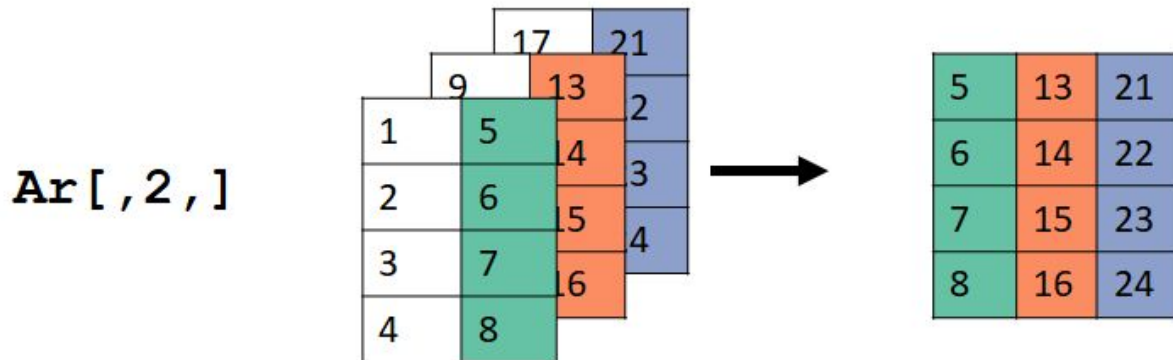
The images below show what happened. For `Ar[1, , ]`, the values in the first row (dimension 1), all columns (dimension 2), and all layers (dimension 3) were selected. Dimension 2 in the input `Ar` became dimension 1 in the output, and dimension 3 in `Ar` became dimension 2 in the output.

Figure 5.3: R bracket notation carries across dimensions within an array, but the result may not be what you expect. Notice how rows are converted to columns.



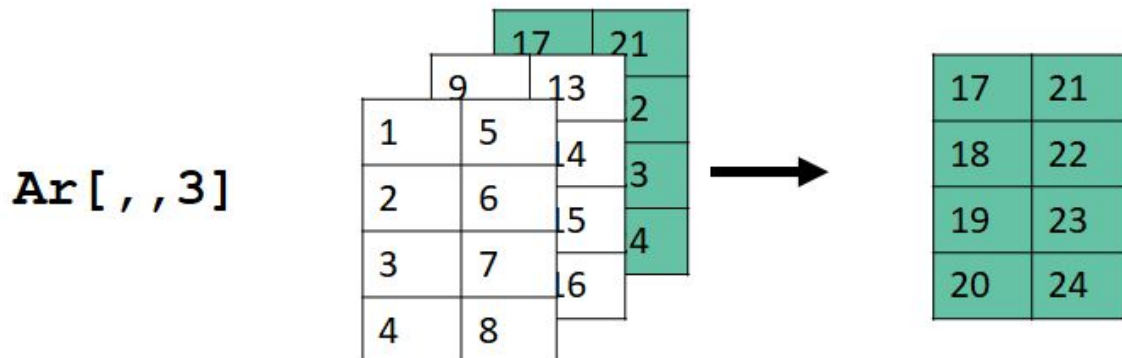
For `Ar[, 2, ]`, the values in the second column (dimension 2), all rows (dimension 1), and all layers (dimension 3) were selected. Dimension 2 in the input `Ar` became dimension 1 in the output, and dimension 3 in `Ar` became dimension 2 in the output.

Figure 5.4: Another example of R bracket notation behavior across layers of an array.



For `Ar[,2,]`, the values in the all rows (dimension 1), all columns (dimension 2), and layer 3 (dimension 3) were selected. Dimension 2 in the input `Ar` became dimension 1 in the output, and dimension 3 in `Ar` became dimension 2 in the output.

Figure 5.5: Selecting an entire “layer” of a 3-dimensional array.

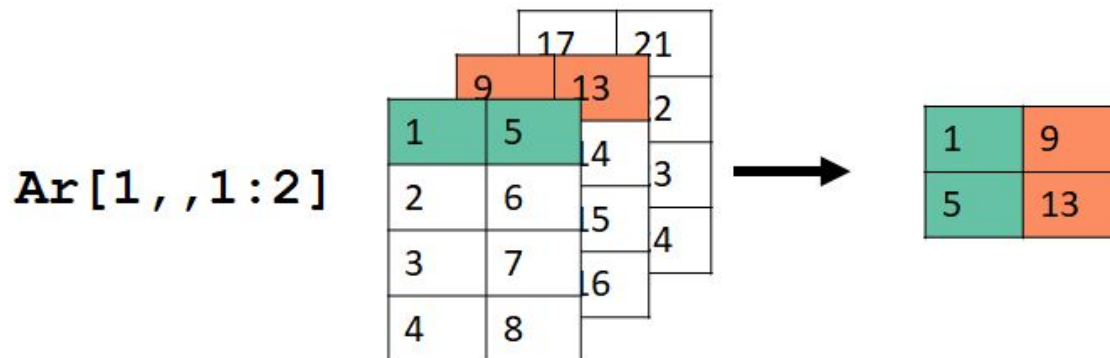


Selecting by ≥ 1 dimension can be tricky. Consider this example:

```
Ar[1,,1:2]
##      [,1] [,2]
## [1,]    1    9
## [2,]    5   13
```

Here we selected the first row (dimension 1), all columns (dimension 2), and layers 1 and 2 (dimension 3).

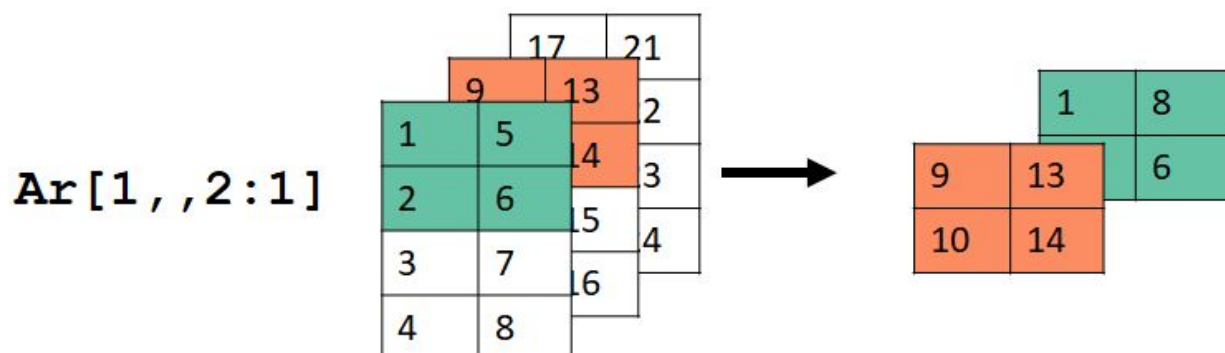
Figure 5.6: Illustration of selecting the first 2 rows of the first 2 elements of a 4 by 2 by 3 array.



And if you want a headache, consider this example:

```
Ar[1:2,,2:1]
## , , 1
##
##      [,1] [,2]
## [1,]    9   13
## [2,]   10   14
##
##      , , 2
##
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
```

Figure 5.7: Sometimes, R bracket notation defies common sense.



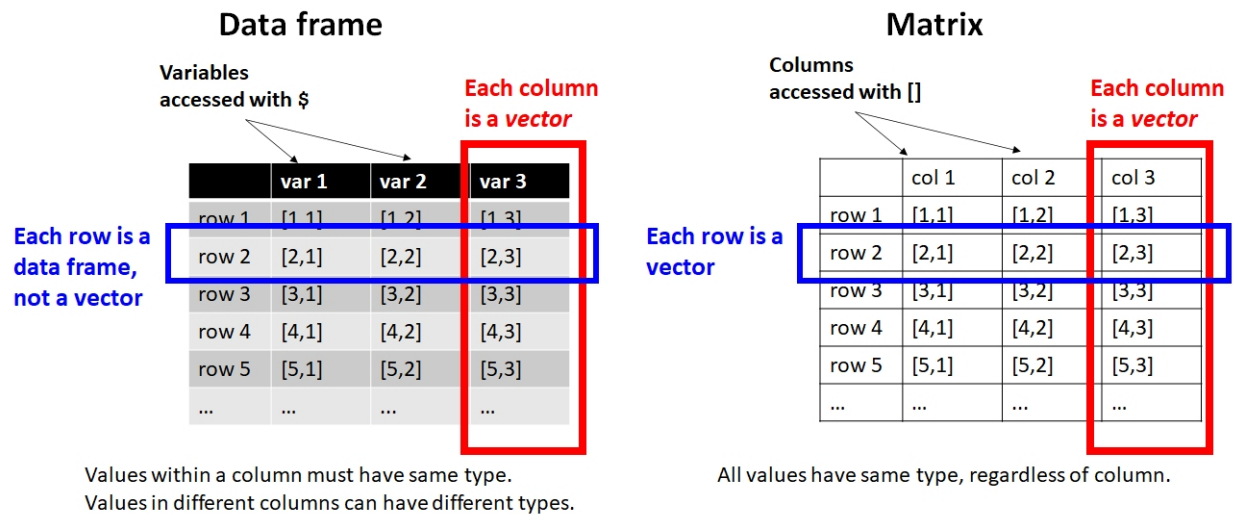
I can't think of a reason why you would need to do something like the last example, but it's nice to know that

you could.

5.2.1.4 Selecting from a data frame

Selecting from a data frame using brackets is largely the same as selecting from a matrix. Just keep in mind that the result might come back as a vector (if you select rows in a single column) or a data frame (any other situation).

Figure 5.8: Illustration of how bracket notation works on data frames, which are not matrices despite their appearance.



```
# make a small data frame from
# the first 5 rows of iris
a <- iris[1:5,]

# returns a vector
a[,1]
## [1] 5.1 4.9 4.7 4.6 5.0

# returns a data frame
a[1,]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5         1.4         0.2   setosa

# returns a data frame
a[1:3,]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5         1.4         0.2   setosa
```

```
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa

# returns a data frame
a[1:3, 1:3]
##      Sepal.Length Sepal.Width Petal.Length
## 1      5.1      3.5      1.4
## 2      4.9      3.0      1.4
## 3      4.7      3.2      1.3
```

5.2.1.5 Selecting from lists

One of the most important kinds of R object is the **list**. A list can be thought of like a series of containers or buckets. Each bucket can contain completely unrelated things, or even other series of buckets. Each bucket is called an **element** of the list. Because of this flexibility, lists are extremely versatile and useful in R programming. Many function outputs are really lists (e.g., the outputs of `lm()` or `t.test()`).

Single elements of a list can be accessed with **double brackets** `[[]]`. Elements can be selected by name or by index. Elements that have a name can be selected by name using the `$` symbol instead of by double brackets.

```
# make a list "a"
x <- runif(10)
y <- "zebra"
z <- matrix(1:12, nrow=3)
a <- list(e1=x, e2=y, e3=z)

a[[1]]
## [1] 0.81537826 0.85151339 0.94974433 0.24860408 0.20883937 0.23072284
## [7] 0.80287954 0.06142657 0.65110133 0.21388695
a[[2]]
## [1] "zebra"
a[["e1"]]
## [1] 0.81537826 0.85151339 0.94974433 0.24860408 0.20883937 0.23072284
## [7] 0.80287954 0.06142657 0.65110133 0.21388695
a$e1
## [1] 0.81537826 0.85151339 0.94974433 0.24860408 0.20883937 0.23072284
## [7] 0.80287954 0.06142657 0.65110133 0.21388695
```

Multiple elements of a list can be selected using **single brackets**. Doing so will return a new list containing the requested elements, **not** the requested elements themselves. This can be a little confusing. Compare

the following two commands and notice that the first returns the 10 random numbers in the first element of `a`, while the second returns a list with 1 element, and that element is the numbers 1 through 10.

```
a[[1]]
## [1] 0.81537826 0.85151339 0.94974433 0.24860408 0.20883937 0.23072284
## [7] 0.80287954 0.06142657 0.65110133 0.21388695
a[1]
## $e1
## [1] 0.81537826 0.85151339 0.94974433 0.24860408 0.20883937 0.23072284
## [7] 0.80287954 0.06142657 0.65110133 0.21388695
```

Next, observe what happens when we select more than one element of the list `a`. The output is a list.

```
a[1:2]
## $e1
## [1] 0.81537826 0.85151339 0.94974433 0.24860408 0.20883937 0.23072284
## [7] 0.80287954 0.06142657 0.65110133 0.21388695
##
## $e2
## [1] "zebra"
class(a[1:2])
## [1] "list"
```

The reason for this syntax is that the single brackets notation allows you to extract multiple elements of the list at once to quickly make a new list.

```
a2 <- a[1:2]
a2
```

```
$e1
[1] 0.81537826 0.85151339 0.94974433 0.24860408 0.20883937 0.23072284
[7] 0.80287954 0.06142657 0.65110133 0.21388695

$e2
[1] "zebra"
```

5.2.2 Selecting and extracting with logical tests

Selecting data from a data frame is one of the most common data manipulations. R offers several ways to accomplish this. One of the most fundamental is the function `which()`, returns the indices in a vector where a logical condition is `TRUE`. This means that you can select any piece of a dataset that can be

identified logically. Consider the example below, which extracts the rows of the `iris` dataset where the variable `Species` matches the character string `"setosa"`.

```
# not run (long output); try it on your machine!
iris[which(iris$Species == "setosa"),]

# better: define which indices you want,
#         and make selection in separate
#         commands.
flag <- which(iris$Species == "setosa")
iris[flag,]
```

In most situations, `which()` can be omitted because it is implicit in the command. However, not using it can sometimes produce strange results. Because of this, I always use `which()` in my code. Including `which()` can also make your code a little clearer to someone who is not familiar with R. For example, the two commands below are exactly equivalent:

```
# not run (long output); try it on your machine!
iris[which(iris$Species == "setosa"),]
iris[iris$Species == "setosa",]
```

What `which()` is really doing in the commands above is returning a vector of numbers. That vector is then used by the containing brackets `[]` to determine which pieces of the data frame to pull. The numbers returned by `which()` are the indices at which a logical vector is `TRUE`. Compare the results of the two commands below, which were used in the commands above to select from `iris`:

```
# logical vector:
iris$Species == "setosa"
## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [13] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [25] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [37] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [49] TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [97] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [121] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [145] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# which identifies which elements of vector are TRUE:
which(iris$Species == "setosa")
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Multiple conditions can be used to select data using the & (and) and | (or) operators.

```
# not run (long output):
iris[which(iris$Species == "versicolor" & iris$Petal.Length <= 4),]

iris[which(iris$Species == "versicolor" | iris$Petal.Length <= 1.6),]
```

The examples above could also be accomplished in several lines; in fact, splitting the selection criteria up can make for cleaner code. The commands below also show how splitting the selection commands up can help you abide by the “single point of definition” principle.

```
# not run (long output):
flag1 <- iris$Species == "versicolor"
flag2 <- iris$Petal.Length <= 4

## meet both conditions:
iris[flag1 & flag2,]
iris[intersect(flag1, flag2),]

## meet either condition
iris[flag1 | flag2,]
iris[union(flag1, flag2),]
```

More than 2 conditions can be combined just as easily:

```
flag1 <- iris$Species == "versicolor"
flag2 <- iris$Petal.Length <= 4
flag3 <- iris$Petal.Length > 0.5
iris[flag1 & flag2 & flag3,]
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
54	5.5	2.3	4.0	1.3	versicolor
58	4.9	2.4	3.3	1.0	versicolor
60	5.2	2.7	3.9	1.4	versicolor
61	5.0	2.0	3.5	1.0	versicolor
63	6.0	2.2	4.0	1.0	versicolor

65	5.6	2.9	3.6	1.3	versicolor
70	5.6	2.5	3.9	1.1	versicolor
72	6.1	2.8	4.0	1.3	versicolor
80	5.7	2.6	3.5	1.0	versicolor
81	5.5	2.4	3.8	1.1	versicolor
82	5.5	2.4	3.7	1.0	versicolor
83	5.8	2.7	3.9	1.2	versicolor
90	5.5	2.5	4.0	1.3	versicolor
93	5.8	2.6	4.0	1.2	versicolor
94	5.0	2.3	3.3	1.0	versicolor
99	5.1	2.5	3.0	1.1	versicolor

Selections can return 0 matches. This usually manifests as a data frame or matrix with 0 rows.

```
flag1 <- iris$Species == "versicolor"
flag2 <- iris$Petal.Length <= 1
iris[flag1 & flag2,]
```

```
[1] Sepal.Length Sepal.Width Petal.Length Petal.Width Species
<0 rows> (or 0-length row.names)
```

A related function is the binary operator² `%in%`. For a command like `A %in% B`, R returns a vector of logical values for each element in A, indicating whether those values occur in B.

```
A <- c("a", "b", "c", "d", "e")
B <- c("c", "d", "e", "f")
A %in% B
## [1] FALSE FALSE TRUE TRUE TRUE
B %in% A
## [1] TRUE TRUE TRUE FALSE
```

While we're here, the set operations `union()`, `intersect()`, and `setdiff()` can be useful for selecting data.

```
A <- c("a", "b", "c", "d", "e")
B <- c("c", "d", "e", "f")

# elements in both sets: intersect
# i.e., AND
```

²A binary operator uses two arguments, one before the symbol and one after. For example, `+` is a binary operator that adds the the argument before the `+` to the argument after the `+`

```
intersect(A, B)
## [1] "c" "d" "e"

# elements in either set:
# i.e., OR
union(A, B)
## [1] "a" "b" "c" "d" "e" "f"

# elements in first set only:
# (asymmetric)
# i.e., XOR (one-sided)
setdiff(A, B)
## [1] "a" "b"
setdiff(B, A)
## [1] "f"

# elements in only one set
# (symmetric)
# i.e., XOR (two-sided)
union(setdiff(A, B), setdiff(B, A))
## [1] "a" "b" "f"
```

One example of how this might be useful is if you want to select observations from a dataset containing one of several values. The example below illustrates this with `iris`:

```
# not run (long output):
iris[iris$Species %in% c("setosa", "versicolor"),]
```

We'll go over some more complicated selection methods in a future section, but what you've seen so far can be very powerful.

5.3 Dates and temporal data

Most of the data we deal with in R are numbers, but textual and temporal data are important two. Working with dates, times, and character strings can be significantly more complicated than working with numbers...so, I've put together an entire page on it. I am the first to admit that the approaches demonstrated here are a little “old-school” compared to some of the newer packages. In particular, many people prefer the tidyverse package `lubridate` (Grolemund and Wickham 2011) for temporal data. Much of this section is based on Spector (2008).

Temporal data can be read and interpreted by R in many formats. The general rule is to use the simplest way that you can, because the date and time classes can be very complicated. For example, don't store

date and time if you only need date. This doesn't mean to discard potentially useful data; just to be judicious in what you import to and try to manipulate in R.

It should be noted that the easiest way to deal with dates and times is "don't". In many situations it's actually not necessary to use temporal data classes. For example, if a field study was conducted in two years, and there is no temporal component to the question, it might be sufficient to just call the years `year1` and `year2` in the dataset. Or, if an elapsed time is used as a variable, it might make more sense to store the elapsed time (e.g., 10 minutes, 11.2 minutes, etc.) rather than use full date/time codes for each observation time.

5.3.1 Dates without times

The easiest way to handle dates is to use the base function `as.Date()`. This function converts character strings to the R date format. This implies that you will have a much easier time if you send dates to R in YYYY-MM-DD format. You can do this in Excel by setting a custom number format. Once in R, use `as.Date()` to convert your properly-formatted text strings into date objects. Once converted, R can do simple arithmetic such as finding intervals between dates.

```
# enter dates as text strings
a <- c("2021-07-23", "2021-08-01", "2021-09-03")

# convert to date
b <- as.Date(a)

# check classes
class(a)
## [1] "character"
class(b)
## [1] "Date"

# can do math with dates!
b[2] - b[1]
## Time difference of 9 days
```

The examples above use the [ISO 8601 date format](#), Year-Month-Day (or YYYY-MM-DD). This format has the advantage that sorting it alphanumerically is the same as sorting it chronologically. Each element has a fixed number of digits to avoid issues like January and October (months 1 and 10) being sorted next to each other.

I suggest that you get in the habit of storing dates this way. This format is the default date format in R (and many other programs), such that R can read dates in this format without additional arguments. If you insist on storing dates in other formats, you can still read those dates into R. You just need to tell

`as.Date()` the format that the date is in. The syntax for formatting dates and times is handled by the function `strptime()` (see `?strptime` for more examples). Something common to all of the formats below is that each element has a *fixed number of digits*, 4 for year and 2 for month or day. This is to avoid ambiguity: does “7/2/2019” “July second, 2019”, or “February 7, 2019”? This also avoids issues arising from treating numbers as character strings. For example, January and October (months 1 and 10) might be sorted next to each other unless represented by “01” and “10” instead of “1” and “10”.

```
# YYYYMMDD
a <- "20210721"
as.Date(a, "%Y%m%d")
## [1] "2021-07-21"

# MMDDYYYY
a <- "07212021"
as.Date(a, "%m%d%Y")
## [1] "2021-07-21"

# MM/DD/YYYY
a <- "07/21/2021"
as.Date(a, "%m/%d/%Y")
## [1] "2021-07-21"

# DD-MM-YYYY
a <- "21-07-2021"
as.Date(a, "%d-%m-%Y")
## [1] "2021-07-21"
```

An even simpler option for dates, if all dates are within the same year, is to use Julian dates (i.e., day-of-year). If you want to read Julian dates into R as dates, you can also use `as.Date()`. Notice in the example below that the Julian dates must be supplied as *character strings with 3 digits*, not as numerals. Notice also that R assigns the current year to the dates unless you specifically tell it to do something else.

```
a <- as.character(c(204, 213, 246))
as.Date(a, "%j")
## [1] "2026-07-23" "2026-08-01" "2026-09-03"

## not run:
## causes error because nchar("15") < 3:
#a <- c(15, 204, 213, 246)
#as.Date(a, "%j")
```

```
# convert all numbers to 3 digit
# character strings first with formatC():
a <- c(15, 204, 213, 246)
x <- formatC(a, width=3, flag="0")
as.Date(x, "%j")
## [1] "2026-01-15" "2026-07-23" "2026-08-01" "2026-09-03"
```

Dates can be reformatted using the function `format()`. Notice that the output is a character string, not a date. The syntax for the conversion comes from `strptime()`, just as with `as.Date()` above. If you need to use a formatted date (character string) as a date (R class) you need to convert it with `as.Date()`.

```
# make some dates as character strings and convert to date
a <- c("2021-07-23", "2021-08-01", "2021-09-03")
x <- as.Date(a)

# Julian day (day of year), as character
format(x, "%j")
## [1] "204" "213" "246"
# as number
as.numeric(format(x, "%j"))
## [1] 204 213 246

# ISO format
format(x, "%Y-%m-%d")
## [1] "2021-07-23" "2021-08-01" "2021-09-03"

# use month abbreviations, separate by slashes
format(x, "%b/%d/%y")
## [1] "Jul/23/21" "Aug/01/21" "Sep/03/21"
```

5.3.2 Dates with times and time zones

[Here be dragons.](#)

Things get more complicated when time is included with a date, but the basic procedures are the same as date-only data. By default, R assumes that times are in the time zone of the current user. So, if you are collaborating with someone in a different time zone, you need to account for time zone in your code. Another default behavior is that R will assume that times without dates are on the current day.

The main function for working with dates *and* times is `strptime()`. This function takes in a character vector and a format argument, which tells R how the character vector is formatted. It returns objects of class `POSIXlt` and `POSIXt`, which contain date and time information in a standardized way.

```
# create some times as character strings
a <- c("01:15:03", "14:28:21")

# time with seconds
strptime(a, format="%H:%M:%S")
## [1] "2026-08-10 01:15:03 EDT" "2026-08-10 14:28:21 EDT"

# time with seconds dropped
strptime(a, format="%H:%M")      # seconds dropped
## [1] "2026-08-10 01:15:00 EDT" "2026-08-10 14:28:00 EDT"
```

Notice that the current date was included in the result of `strptime()` (at least, the date when I rendered the document).

Time zones should be specified when working with data that contain dates and times. Remember that R will assume the time is in the current user's time zone. This almost always means the time zone used by the operating system in which R is running. If you did something silly like record data in one time zone and analyze it in another, you have to be careful. For example, I once ran a field study in Indiana where data were recorded in Eastern time. But, the office where I analyzed the data was in Missouri (Central time). R kept converting the time-stamped data to Central time, which interfered with the analysis until I learned how to work with time zones in R. To make things worse, data were recorded in the summer (Eastern daylight time) but analyzed in both early fall (Central daylight time) and late fall (Central standard time).

```
# make some date-times and convert to date-time class
a <- c("2021-07-23 13:21", "2021-08-01 09:15")
x <- strptime(a, "%Y-%m-%d %H:%M")
x
## [1] "2021-07-23 13:21:00 EDT" "2021-08-01 09:15:00 EDT"

# times without dates will be assigned current date:
a2 <- c("13:21", "09:15")
x2 <- strptime(a2, "%H:%M")
x2
## [1] "2026-08-10 13:21:00 EDT" "2026-08-10 09:15:00 EDT"
```

Time zone names are not always portable across systems. R has a list of built-in time zones that it can handle, and these should work, but be careful and **always** validate any code that depends on time zones. The list of time zones R can use can be printed to the console with the command `OlsonNames()`.

```
# US Central Daylight Time (5 hours behind UTC)
a <- c("2021-07-23 13:21", "2021-08-01 09:15")
```

```
# strptime() will assign whatever time zone you enter:
b <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Eastern")
b
## [1] "2021-07-23 13:21:00 EDT" "2021-08-01 09:15:00 EDT"

b <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Central")
b
## [1] "2021-07-23 13:21:00 CDT" "2021-08-01 09:15:00 CDT"
```

Internally, dates and times are stored as the number of seconds elapsed since midnight (00:00 hours) on 1 January 1970, in UTC. You can see these values with `as.numeric()`. The time zone of a value affects what is printed to the console. That time zone is stored by R as an **attribute** of the value. Attributes are a kind of metadata associated with some R data types that can be accessed with `attributes()`.

```
attributes(b)$tzone
```

```
[1] "US/Central" "CST"          "CDT"
```

To change the time zone associated with value, simply set the attribute to the desired time zone. This is basically telling R the time zone that a date and time refer to. Note that will change the actual value of a date/time. It does NOT translate times in one time zone to another.

```
as.numeric(b)
## [1] 1627064460 1627827300
attr(b, "tzone") <- "America/Los_Angeles"

# Central and Pacific time different
# by 7200 seconds (2 hours)
as.numeric(b)
## [1] 1627071660 1627834500
```

To convert one time zone to another, the values must first be converted to the `POSIXct` class.

```
b <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Central")
b2 <- as.POSIXct(b)

# check time zone
attributes(b2)$tzone
## [1] "US/Central"
```

```
# check actual time (seconds since epoch)
as.numeric(b2)
## [1] 1627064460 1627827300

# change time zone of POSIXct object
attr(b2, "tzzone") <- "Europe/London"

# actual time same as before:
as.numeric(b2)
## [1] 1627064460 1627827300
```

5.3.3 Working with dates and times

Why should we go through all of this trouble to format dates and times as dates and times? The answer is that R can perform many useful calculations with dates and times. One of these is time intervals by using subtraction or the function `difftime()`. Subtracting two date/times will return an answer but perhaps not in the units you want; use `difftime()` if you want to specify the units.

```
# how many days, hours, and minutes until Dr. Green can retire?

## current time:
a <- Sys.time()

## retirement date per USG:
b <- "2048-07-01 00:00"

## convert to POSIXlt
x <- strptime(a, "%Y-%m-%d %H:%M")
y <- strptime(b, "%Y-%m-%d %H:%M")

## calculate time in different units:
difftime(y,x, units="days")
## Time difference of 7995.517 days
difftime(y,x, units="hours")
## Time difference of 191892.4 hours
difftime(y,x, units="mins")
## Time difference of 11513545 mins
y-x
## Time difference of 7995.517 days
```

Another application: what is the time difference between Chicago and Honolulu?

```
a <- c("2025-08-08 14:17")
x <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Central")
y <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Hawaii")
difftime(x,y)
```

Time difference of -5 hours

Oddly, `difftime()` cannot return years. This is probably because there are [different types of years](#) (sidereal, tropical, anomalistic, lunar, etc.). You can convert a time difference in days to one in years by dividing by the number of days in the kind of year you want. The example below converts to does this and changes the “units” attribute of the `difftime` output to match.

```
d <- difftime(y,x, units="days")

# julian year
d <- d/365.25
attr(d, "units") <- "years"
d
## Time difference of 0.0005703856 years
```

The Julian year is useful because it is defined in terms of SI units as 31557600 seconds, or 365.25 days³. Most of the alternative years at the link above vary by less than a few minutes.

R can also generate regular sequences of dates and times just like it can numeric values. See `?seq.date` for details and syntax.

```
a <- as.Date("2025-08-08")
seq(a, by="day", length=10)
## [1] "2025-08-08" "2025-08-09" "2025-08-10" "2025-08-11" "2025-08-12"
## [6] "2025-08-13" "2025-08-14" "2025-08-15" "2025-08-16" "2025-08-17"
seq(a, by="days", length=6)
## [1] "2025-08-08" "2025-08-09" "2025-08-10" "2025-08-11" "2025-08-12"
## [6] "2025-08-13"
seq(a, by="week", length=10)
## [1] "2025-08-08" "2025-08-15" "2025-08-22" "2025-08-29" "2025-09-05"
## [6] "2025-09-12" "2025-09-19" "2025-09-26" "2025-10-03" "2025-10-10"
seq(a, by="6 weeks", length=10)
## [1] "2025-08-08" "2025-09-19" "2025-10-31" "2025-12-12" "2026-01-23"
## [6] "2026-03-06" "2026-04-17" "2026-05-29" "2026-07-10" "2026-08-21"
```

³This is slightly longer than a *Rent* year, which is 525600 minutes (31536000 seconds).

```
seq(a, by="-4 weeks", length=5)
## [1] "2025-08-08" "2025-07-11" "2025-06-13" "2025-05-16" "2025-04-18"
```

5.4 Character data

Recall from earlier that “character” is just another **type** of data in R—the type that contains text. This means that text values can be stored and manipulated in many of the same ways as numeric data. The vectorized functions in R can make managing your text-based data very fast and efficient.

A value containing text, or stored as text, is called a **character string**. Character strings can be stored in many R objects. A set of character strings is called a **character vector**.

5.4.1 Combining character strings

Character strings can be combined together into larger character strings using the function `paste()`. Notice that this is not the same as making a vector containing several character strings. That operation is done with `c()`.

```
x <- paste("a", "b", "c")
y <- c("a", "b", "c")
x
## [1] "a b c"
y
## [1] "a" "b" "c"
length(x)
## [1] 1
length(y)
## [1] 3
```

The function `paste()`, and its cousin `paste0()`, combine their arguments together and output a character string. The inputs do not have to be character strings, but the outputs will be. The difference between `paste()` and `paste0()` is that `paste0()` does not put any characters between its inputs, while `paste()` can place any valid character between its inputs. The default separator is a space.

```
paste0("a", "zebra", 3)
## [1] "azebra3"
paste("a", "zebra", 3)
## [1] "a zebra 3"
paste("a", "zebra", 3, sep="_")
```

```
## [1] "a_zebra_3"
paste("a", "zebra", 3, sep=".")
## [1] "a.zebra.3"
paste("a", "zebra", 3, sep="/")
## [1] "a/zebra/3"
paste("a", "zebra", 3, sep="giraffe")
## [1] "agiraffezebragiraffe3"
```

One of my favorite uses for the paste functions is to automatically generate folder names and output file names. The example below adds the current date (`Sys.Date()`) into a file name.

```
# not run
top.dir <- "C:/_data"
out.dir <- paste(top.dir, "test", sep="/")
out.name <- paste0("output_", Sys.Date(), ".csv")
write.csv(iris, paste(geo.dir, out.name, sep="/"))

# even better:
top.dir <- "C:/_data"
out.dir <- paste(top.dir, "test", sep="/")
out.name <- paste0("output_", Sys.Date(), ".csv")
out.path <- paste(geo.dir, out.name, sep="/")
write.csv(iris, out.path)
```

When the inputs to a `paste()` function are contained in a single vector, then the desired separator must be supplied using the argument `collapse` instead of `sep`.

```
a <- c("a", "b", "c")
paste(a, collapse="-")
## [1] "a-b-c"
paste(a)
## [1] "a" "b" "c"
```

5.4.2 Printing character strings

The `cat()` function can not only put strings together, but also print them to the R console. This is useful if you want to generate messages from inside functions or loops. When `cat()` receives inputs that are not character strings, it evaluates them and converts them to character strings. The examples below show how inputs can be evaluated and printed. It also illustrates the use of `\n` and `\t` to insert new lines and tabs, respectively. Finally, the command `flush.console()` ensures that the console is updated with the most recent outputs.


```

# using cat() in a function:
my.fun <- function(x){
  cat("You entered", x, "!", "\n")
  cat("\t", "1 more is", x+1, "\n")
}
my.fun(7)
## You entered 7 !
##   1 more is 8
my.fun(42)
## You entered 42 !
##   1 more is 43

# using cat() in a loop:
for(i in 1:5){
  cat("starting iteration", i, "\n")
  for(j in 1:3){
    cat("\t", "iteration", i, "part", j, "complete!", "\n")
    flush.console()
  }
  cat("iteration", i, "complete!", "\n")
}
## starting iteration 1
##   iteration 1 part 1 complete!
##   iteration 1 part 2 complete!
##   iteration 1 part 3 complete!
## iteration 1 complete!
## starting iteration 2
##   iteration 2 part 1 complete!
##   iteration 2 part 2 complete!
##   iteration 2 part 3 complete!
## iteration 2 complete!
## starting iteration 3
##   iteration 3 part 1 complete!
##   iteration 3 part 2 complete!
##   iteration 3 part 3 complete!
## iteration 3 complete!
## starting iteration 4
##   iteration 4 part 1 complete!
##   iteration 4 part 2 complete!
##   iteration 4 part 3 complete!
## iteration 4 complete!

```

```
## starting iteration 5
## iteration 5 part 1 complete!
## iteration 5 part 2 complete!
## iteration 5 part 3 complete!
## iteration 5 complete!
```

The other printing function, `sink()`, is specifically for printing outputs into text files. Refer back to the [Section 4.4.4](#) for an example.

5.4.3 Matching character strings

Like numeric data, character strings can be matched exactly with the comparison operator `==`.

```
# make a character vector
x <- c("zebra", "hippo")

# element-wise test
x == "zebra"
## [1] TRUE FALSE

# get indices where TRUE or FALSE
which(x == "zebra")
## [1] 1
which(x != "zebra")
## [1] 2

# test all at once
any(x == "zebra")
## [1] TRUE
all(x == "zebra")
## [1] FALSE
```

Notice that this kind of matching is case-sensitive. If you need to match regardless of case, you can either force all of the comparands to be the same case, or use regular expressions (see [Section 5.4.4](#)).

```
x <- c("zebra", "hippo")
any(x == "Zebra")
## [1] FALSE
any(x == tolower("zebra"))
## [1] TRUE
```

R is smart enough to sort text alphabetically, and to make comparisons based on alphabetical sorting.

```
x <- c("zebra", "hippo", "lion")
sort(x)
## [1] "hippo" "lion"  "zebra"
"zebra" > "hippo"
## [1] TRUE
"lion" > "zebra"
## [1] FALSE
which(x > "jaguar")
## [1] 1 3
```

5.4.4 Searching text with the `grep()` and `grepl()` functions

Regular expressions, or **regex**, are a [syntax for searching text strings](#). Learning how to use regular expressions is easily a semester-long course on its own. Here we'll just explore a few common use cases. The R functions that deal with regular expressions are `grep()` and its cousins. The most commonly used versions are `grep()` and `grepl()`.

- `grep()` returns the indices in the input vector that match the pattern. The length of the output will be the number of matches.
- `grepl()` returns a logical value (TRUE or FALSE) telling whether each element in the input vector matches the pattern. The length of the output will be same as the length of the input.

This means that `grepl()` is analogous to using `==` to test for equality, while `grep()` is analogous to using `which()`. Compare the results of the two commands on the simple vector `x` below. It's not hard to imagine how powerful these functions can be.

```
x <- c("zebra", "hippo", "lion", "tiger")
grep("e", x)
## [1] 1 4
grepl("e", x)
## [1] TRUE FALSE FALSE TRUE
```

Some of the arguments to `grep()` can save you some time. Setting `value=TRUE` will return the values where the input matches the pattern.

```
grep("e", x, value=TRUE)
## [1] "zebra" "tiger"
```

```
# equivalent to:
x[grep("e", x)]
## [1] "zebra" "tiger"
```

Setting `invert=TRUE` will find elements of the input that do NOT match the pattern.

```
grep("e", x)
## [1] 1 4
grep("e", x, invert=TRUE)
## [1] 2 3
```

By default, `grep()` is case-sensitive. This can be overridden with the option `ignore.case`.

```
x <- c("zebra", "hippo", "lion", "tiger", "Tiger")
grep("tiger", x)
## [1] 4
grep("Tiger", x)
## [1] 5
grep("tiger", x, ignore.case=TRUE)
## [1] 4 5
grep("TIGER", x, ignore.case=TRUE)
## [1] 4 5
```

We will see some uses of `grep()` and its cousins in the examples later.

5.4.5 Replacing parts of character strings

Text within character strings can be replaced or modified in several ways. If you are comfortable with regular expressions (see previous section) then you can use function `gsub()`. The syntax for `gsub()` is similar to that of `grep()`, with an additional input for the replacement string.

Note that `gsub()` will replace every occurrence of the pattern, whether or not an occurrence is a whole word or not. The related function `sub()` works like `gsub()`, but only replaces the first occurrence of the pattern.

```
x <- c("zebra", "hippo", "lion", "tiger", "Tiger")
gsub("tiger", "giraffe", x)
## [1] "zebra" "hippo" "lion" "giraffe" "Tiger"
gsub("tiger", "giraffe", x, ignore.case=TRUE)
## [1] "zebra" "hippo" "lion" "giraffe" "giraffe"
gsub("z", "Z", x)
## [1] "Zebra" "hippo" "lion" "tiger" "Tiger"
```

```
sub("tiger", "giraffe", x, ignore.case=TRUE)
## [1] "zebra"      "hippo"      "lion"      "giraffe" "giraffe"
```

5.4.6 Extracting from and splitting strings

Character strings can be split up using the function `strsplit()` (short for “string split”). This function can use regular expressions like `grep()` or `gsub()` can. It works by breaking a character string apart at a given text string. Unfortunately, its output can be a little confusing if you’re not used to working with R objects:

```
x <- c("Peromyscus_leucopus",
      "Sigmodon_hispidus",
      "Microtus_pennsylvanicus")
strsplit(x, "_")
```

```
[[1]]
[1] "Peromyscus" "leucopus"

[[2]]
[1] "Sigmodon" "hispidus"

[[3]]
[1] "Microtus"      "pennsylvanicus"
```

Function `strsplit()` returns a list with one element for each element in the input vector. Each element of that list is a character vector, consisting of the pieces of that element of the input vector split by the splitting string. So, if you want only one part of each element of the input vector, you need to extract them from the output list. In this example, we can get the generic epithets by pulling the first element of each list element. The function `sapply()` applies a function to each element of a list and returns a result with the simplest possible structure (`sapply` is short for “apply and simplify”). The two commands below are equivalent, but the first is much simpler and should be the method you use.

```
sapply(strsplit(x, "_"), "[", 1)
## [1] "Peromyscus" "Sigmodon"   "Microtus"

# equivalent, not as easy:
do.call(c, lapply(strsplit(x, "_"), "[", 1))
## [1] "Peromyscus" "Sigmodon"   "Microtus"
```

The “`sapply-strsplit`” method is useful for breaking apart strings whose parts carry information. One common use case is breaking up dates into year, month, and day. Note that there are more elegant

ways to do this using the R date classes, but treating dates as text strings can be convenient because the date and time classes are such a pain in the neck. When using this method, you will need to keep track of when date components are stored as text and when they are stored as numbers.

The code block below shows how to break apart dates using `strsplit()` and `sapply()`.

```
x <- c("2019-08-03", "2020-08-15", "2021-08-07")
ssx <- strsplit(x, "-")
a <- data.frame(year=sapply(ssx, "[", 1),
                mon=sapply(ssx, "[", 2),
                day=sapply(ssx, "[", 3))

class(a$year) # character, not a number!
## [1] "character"
class(a$mon) # character, not a number!
## [1] "character"
class(a$day) # character, not a number!
## [1] "character"

a$year <- as.numeric(a$year)
a$mon <- as.numeric(a$mon)
a$day <- as.numeric(a$day)

class(a$year) # numbers!
## [1] "numeric"
class(a$mon) # numbers!
## [1] "numeric"
class(a$day) # numbers!
## [1] "numeric"
```

5.4.7 Matching text strings as indices

As biologists we often keep our data in several separate spreadsheets or tables. These tables are interrelated by a set of “key” variables. If that sounds like a sketchy, haphazard way to build a database, that’s because it is. Databases are outside the scope of this course, but we will learn how to relate tables using R. If you’ve ever used the VLOOKUP function in Excel, then you know exactly what problem `match()` is designed to solve.

The function `match()` finds the indices of the matches of its first argument in its second. In other words, it searches for the values in one vector in a second vector, then returns the positions of the matches from the second vector. A simple example is below:

```

vec1 <- c("gene1", "gene2", "gene3", "gene4")
vec2 <- c("gene1", "gene2", "gene1", "gene2",
          "gene5", "gene3", "gene3", "gene4")
vec3 <- c("gene1", "gene2", "gene1", "gene2",
          "gene4", "gene3", "gene3", "gene4")

match(vec1, vec2)
## [1] 1 2 6 8
match(vec2, vec1)
## [1] 1 2 1 2 NA 3 3 4
match(vec3, vec1)
## [1] 1 2 1 2 4 3 3 4

```

The first example shows what `match()` is doing. For element in the first vector, `match()` returns the first position where that value occurs in the second vector. The second example shows what happens when trying to match a value that is not in the second vector: R returns `NA`. The third example shows how `match()` can be used to relate tables. The longer vector `vec3` is matched to the shorter vector `vec1`, which contains each unique value only once. If `vec1` was a variable in a data frame that contained other variables associated with genes, then the result of `match(vec3, vec1)` would return the rows in `vec1`'s data frame associated with each row of `vec3`'s data frame.

The code below demonstrates how `match()` might be used in practice. This example uses `iris` and an imaginary second table. The commands use `match()` to assign a second variable, petal color, to `iris` using species as the key variable. We'll revisit this example dataset again in [Section 5.5.3.1](#) when we learn how to merge data frames.

```

# not run (see next section)

options(stringsAsFactors=FALSE)
name1 <- "dat_samples_2021-08-04.csv"
name2 <- "dat_stream_indices_2021-08-04.csv"
name3 <- "loc_meta_vaa_2020-11-18.csv"

x1 <- read.csv(name1, header=TRUE)
x2 <- read.csv(name2, header=TRUE)
x3 <- read.csv(name3, header=TRUE)

# x2 contains the response variable
dat <- x2
# add site name, year, and date from x1
matchx <- match(dat$uocc, x1$uocc)
dat$site <- x1$loc[matchx]

```

```

dat$year <- x1$year[matchx]
dat$doy  <- x1$doy[matchx]

# equivalent to:
#dat$site <- x1$loc[match(dat$uocc, x1$uocc)]
#dat$year <- x1$year[match(dat$uocc, x1$uocc)]
#dat$doy  <- x1$doy[match(dat$uocc, x1$uocc)]

# grab latitude and longitude from x3
matchx <- match(dat$site, x3$loc)
dat$lat <- x3$lat[matchx]
dat$long <- x3$long[matchx]

```

5.4.8 Standardizing numbers as text

The function `formatC()` formats numbers in a manner similar to the language C⁴. This is useful for naming objects or parts of objects with numeric names that have a fixed width or format. The most common use case is converting numbers like “1, 12, 112” to strings like “001, 012, 112”. The example below uses `formatC()` in combination with `paste()` to generate some file names.

```

file.numbers <- formatC(1:4, width=3, flag="0")
file.names <- paste("file", file.numbers, sep="_")
file.names

```

```
[1] "file_001" "file_002" "file_003" "file_004"
```

This is useful for generating text strings that need to be sorted. In the example above, the files “file_001.txt”, “file_002.txt”, “file_003.txt”, and “file_004.txt” would be in the order you expect, and a file like “file_010.txt” would come *after* “file_009.txt”. If the leading zeros had not been added, then the order would be “file_1.txt”, “file_10.txt”, “file_2.txt”, “file_3.txt”, etc. (notice that 10 was between 1 and 2).

5.5 Data frame management

Most of the time when you work with data in R, those data will be stored in **data frames**. A data frame looks like a matrix, but functions in many ways spreadsheet. Internally, a data frame is really a list, but for most day-to-day use you can think of a data frame as a table with *rows* and *columns*.

Being able to manage data frames is an important part of working with R.

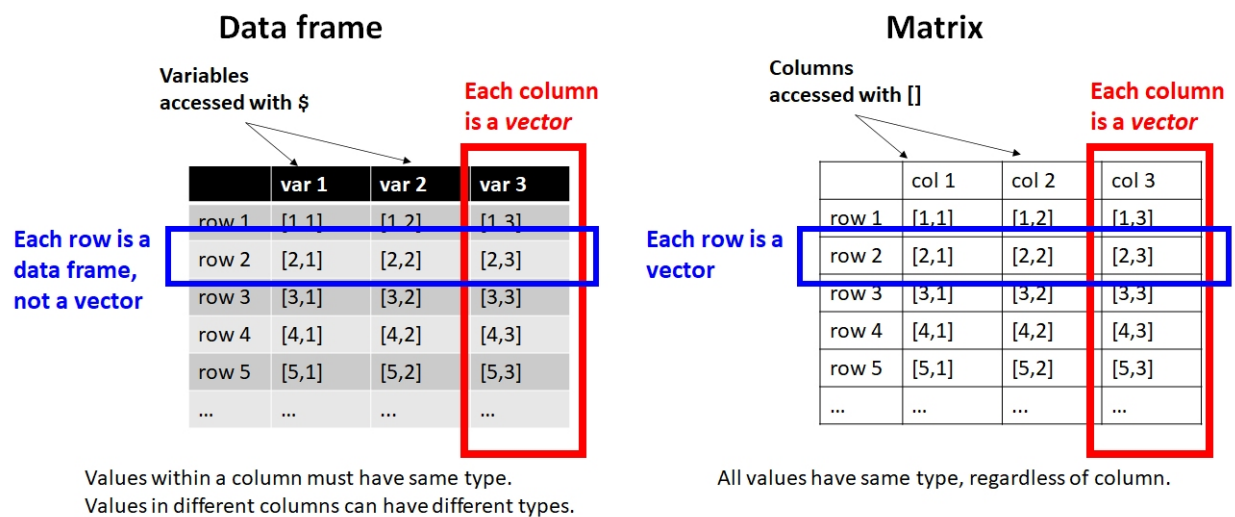
This section demonstrates some of the most common data frame operations.

⁴Or so the documentation claims. I’ve never learned C.

5.5.1 Data frame structure

Data frames and matrices look very similar, but their internal structures are very different. This has huge implications for how you work with and manipulate data frames and matrices. The figure below shows some of the most important differences.

Figure 5.9: Illustration of the differences between an R data frame (left) and an R matrix (right).



When using **bracket notation**, accessing parts of a data frame is the same as for a matrix. There are a couple of differences though:

- Matrices do not need column names, but data frames do. They are accessed with `colnames()` for matrices, but `names()` for data frames.
- Columns of a data frame can be called with `$`; e.g., `iris$Petal.Length`. The equivalent for a matrix would be `iris[, "Petal.Length"]`. The `$` works for data frames only, while the brackets method works for both data frames and matrices.

Here are some common methods of accessing values in a data frame:

```
# single variables (columns): $
iris$Petal.Length
## [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4
## [19] 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2
## [37] 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7 4.5 4.9 4.0
## [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0
## [73] 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0
## [91] 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1 5.9 5.6 5.8 6.6 4.5 6.3
## [109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0
```

```
## [127] 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9
## [145] 5.7 5.2 5.0 5.2 5.4 5.1

# multiple variables: brackets []
## head() shows only first few rows (output very long)
a <- iris[,c("Petal.Length", "Sepal.Length")]
head(a)
##      Petal.Length Sepal.Length
## 1           1.4           5.1
## 2           1.4           4.9
## 3           1.3           4.7
## 4           1.5           4.6
## 5           1.4           5.0
## 6           1.7           5.4

# rows: brackets []
# remember the comma!
iris[1:4,]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1           3.5           1.4           0.2   setosa
## 2           4.9           3.0           1.4           0.2   setosa
## 3           4.7           3.2           1.3           0.2   setosa
## 4           4.6           3.1           1.5           0.2   setosa
iris[1,]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1           3.5           1.4           0.2   setosa
```

When a data frame is printed to the console, this is only a representation of the data frame's true form: a **list**. This is why single columns are accessible by a `$`: this is the symbol for accessing an element of a list. The list nature of a data frame is also why **rows** of a dataset are actually data frames themselves and not vectors.

5.5.2 Common operations

5.5.2.1 Adding and deleting variables

New variables can be added to a data frame by **assignment** (`<-`). Assigning values to a variable in a data frame will create that variable if it doesn't already exist—and overwrite values if the variable is already in the data frame.

```

# spare copy with just a
# few rows for demonstration
x <- iris[1:5,]

# make a new column with <-
x$Petal.Area <- apply(x[,3:4], 1, prod)

# shows data frame with new variable:
x
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Area
## 1          5.1          3.5          1.4          0.2  setosa      0.28
## 2          4.9          3.0          1.4          0.2  setosa      0.28
## 3          4.7          3.2          1.3          0.2  setosa      0.26
## 4          4.6          3.1          1.5          0.2  setosa      0.30
## 5          5.0          3.6          1.4          0.2  setosa      0.28

# overwrite the new variable
x$Petal.Area <- 3
x
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Area
## 1          5.1          3.5          1.4          0.2  setosa          3
## 2          4.9          3.0          1.4          0.2  setosa          3
## 3          4.7          3.2          1.3          0.2  setosa          3
## 4          4.6          3.1          1.5          0.2  setosa          3
## 5          5.0          3.6          1.4          0.2  setosa          3

```

Variables can also be added using function `cbind()`. This name is short for “column bind”~[`cbind()` works on matrices too.]. To use `cbind()`, the input data frames must have the same number of rows. Note that `cbind()` sticks the data frames together as they are. If the records in the inputs need to be matched up and are not already in the same order, then either the data frames need to be aligned, merged with function `merge()`, or the variables need to be added using `match()` and a key variable.

```

# make 2 data frames where rows line up
x <- iris[1:5, 1:2]
y <- iris[1:5, 3:5]
cbind(x,y)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1          3.5          1.4          0.2  setosa
## 2          4.9          3.0          1.4          0.2  setosa
## 3          4.7          3.2          1.3          0.2  setosa
## 4          4.6          3.1          1.5          0.2  setosa
## 5          5.0          3.6          1.4          0.2  setosa

```

```
# shows that cbind() uses inputs as-is
# and makes no attempt at matching.
cbind(x, y[5:1,])
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1           3.5           1.4           0.2   setosa
## 2           4.9           3.0           1.5           0.2   setosa
## 3           4.7           3.2           1.3           0.2   setosa
## 4           4.6           3.1           1.4           0.2   setosa
## 5           5.0           3.6           1.4           0.2   setosa
```

Variables can be deleted by setting them to NULL (notice all caps) or by making a new version of the data frame without the variable. If I am only removing a few variables, I prefer setting them individually to NULL. This makes it less likely that I will accidentally remove the wrong columns.

```
# delete this variable
z <- x
z$Petal.Area <- NULL
z
##      Sepal.Length Sepal.Width
## 1           5.1           3.5
## 2           4.9           3.0
## 3           4.7           3.2
## 4           4.6           3.1
## 5           5.0           3.6

# delete 1 variable with brackets
z <- x
z <- z[,which(names(z) != "Petal.Area")]
z
##      Sepal.Length Sepal.Width
## 1           5.1           3.5
## 2           4.9           3.0
## 3           4.7           3.2
## 4           4.6           3.1
## 5           5.0           3.6

# delete >1 variable
z <- x
del.cols <- c("Petal.Area", "Petal.Length")
z <- z[,which(!names(z) %in% del.cols)]
z
```

```
##      Sepal.Length Sepal.Width
## 1           5.1           3.5
## 2           4.9           3.0
## 3           4.7           3.2
## 4           4.6           3.1
## 5           5.0           3.6
```

5.5.2.2 Modifying variables

Variables in a data frame (or anywhere) can be modified by assigning them new values. Note that this overwrites the previous values. If you think you might want the old values for some reason, save the modified values in a new variable or even new object instead.

```
# not run

# spare copy with just a
# few rows for demonstration
x <- iris[1:5,]
x$Petal.Length <- log(x$Petal.Length)

# better: keep modified values in new variable

# spare copy with just a
# few rows for demonstration
x <- iris[1:5,]
x$log.petlen <- log(x$Petal.Length)
```

5.5.2.3 Changing values using a rule

By combining the logical tests, bracket notation, and assignment, values in a variable can be replaced according to their values.

```
# not run (long output), but try it on your machine!
x <- iris

# select and replace by one condition:
flag <- which(x$Sepal.Length >= 6)
x$Sepal.Length[flag] <- 5.99
x
```

```

# select and replace by 2 conditions:
flag1 <- which(x$Petal.Length >= 3)
flag2 <- which(x$Petal.Length <= 5)
x$Petal.Length[intersect(flag1, flag2)] <- 4
x

# equivalent to above, but with &
flag1 <- which(x$Petal.Length >= 3)
flag2 <- which(x$Petal.Length <= 5)
x$Petal.Length[flag1 & flag2] <- 4
x

```

5.5.2.4 Renaming variables

Renaming variables in Excel is easy: just enter the new name into the cell at the top of a column. Things are a little more complicated in R but not much.

There are a few ways to rename columns, and they all involve accessing the variable names using function `names()`.

```

# spare copy with just a
# few rows for demonstration
x <- iris[1:5,]

# rname by column number:
names(x)[1] <- "sep.len"
x
##      sep.len Sepal.Width Petal.Length Petal.Width Species
## 1      5.1         3.5         1.4         0.2    setosa
## 2      4.9         3.0         1.4         0.2    setosa
## 3      4.7         3.2         1.3         0.2    setosa
## 4      4.6         3.1         1.5         0.2    setosa
## 5      5.0         3.6         1.4         0.2    setosa

# match name exactly:
names(x)[which(names(x) == "Petal.Length")] <- "pet.len"
x
##      sep.len Sepal.Width pet.len Petal.Width Species
## 1      5.1         3.5         1.4         0.2    setosa
## 2      4.9         3.0         1.4         0.2    setosa
## 3      4.7         3.2         1.3         0.2    setosa

```

```
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa

# equivalent, but separate search and action
flag <- which(names(x) == "Petal.Width")
names(x)[flag] <- "pet.wid"
x
##      sep.len Sepal.Width pet.len pet.wid Species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa

# by position:
names(x)[ncol(x)] <- "species"
x
##      sep.len Sepal.Width pet.len pet.wid species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa

names(x)[1] <- "sepal.len"
x
##      sepal.len Sepal.Width pet.len pet.wid species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa

# change several names at once:
names(x)[1:4] <- paste0("x", 1:4)
x
##      x1  x2  x3  x4 species
## 1  5.1  3.5  1.4  0.2  setosa
## 2  4.9  3.0  1.4  0.2  setosa
## 3  4.7  3.2  1.3  0.2  setosa
## 4  4.6  3.1  1.5  0.2  setosa
```

```
## 5 5.0 3.6 1.4 0.2 setosa
```

Finally, if you have a lot of time on your hands, you can make a new copy of the variable with the name you want, then delete the original version. This has the disadvantage of changing the order of columns.

```
x <- iris[1:5,] # spare copy for demonstration
x$pet.len <- x$Petal.Length
x$Petal.Length <- NULL
x
```

	Sepal.Length	Sepal.Width	Petal.Width	Species	pet.len
1	5.1	3.5	0.2	setosa	1.4
2	4.9	3.0	0.2	setosa	1.4
3	4.7	3.2	0.2	setosa	1.3
4	4.6	3.1	0.2	setosa	1.5
5	5.0	3.6	0.2	setosa	1.4

5.5.2.5 Rearranging columns

Reordering records or columns within a data frame is accomplished by creating a new version of the dataset with the desired arrangement. This rearrangement is accomplished using methods we have already seen.

```
# spare copy with just a
# few rows for demonstration
x <- iris[1:5,]

# rearrange by number
x <- x[,c(5, 3, 4, 1, 2)]
x
##   Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1 setosa          1.4          0.2          5.1          3.5
## 2 setosa          1.4          0.2          4.9          3.0
## 3 setosa          1.3          0.2          4.7          3.2
## 4 setosa          1.5          0.2          4.6          3.1
## 5 setosa          1.4          0.2          5.0          3.6

# rearrange by name
col.order <- c("Species", "Sepal.Length", "Sepal.Width",
               "Petal.Length", "Petal.Width")
x <- x[,col.order]
x
```


##	Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
## 1	setosa	5.1	3.5	1.4	0.2
## 2	setosa	4.9	3.0	1.4	0.2
## 3	setosa	4.7	3.2	1.3	0.2
## 4	setosa	4.6	3.1	1.5	0.2
## 5	setosa	5.0	3.6	1.4	0.2

5.5.2.6 Sorting and ordering data

Sorting data frames is mostly done using the function `order()` (function `sort()` works on vectors, not data frames). Function `order()` is usually put inside brackets in the rows slot. The arguments to `order()` define the variables by which the data frame will be sorted. Rows are sorted by the first variable, then the second, and so on. By default, rows are placed in *ascending* order.

Use `sort()` on vectors:

```
# values 1:12 in random order
set.seed(123)
w <- sample(1:12, 12, replace=FALSE)
w
## [1] 3 12 10 2 6 11 5 4 9 8 1 7

# sort ascending
w.inc <- sort(w)
w.inc
## [1] 1 2 3 4 5 6 7 8 9 10 11 12

# sort descending:
w.dec1 <- sort(w, decreasing=TRUE)
## or:
w.dec2 <- rev(sort(w))
w.dec1
## [1] 12 11 10 9 8 7 6 5 4 3 2 1
w.dec2
## [1] 12 11 10 9 8 7 6 5 4 3 2 1
```

One of my favorite uses of `sort()` is to get the unique values of a vector in ascending (especially alphabetical) order.

```
# get 100 random letters from a:e
a <- sample(letters[1:5], 30, replace=TRUE)
a
```

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

# get the unique values, in alpha order
a.uniq <- sort(unique(a))
a.uniq
## [1] "a" "b" "c" "d" "e"
```

Using `order()` on a data frame is similar to running 1 or more `sort()` commands, but where `sort()` returns the sorted vector, `order()` actually returns the indices of the sorted object.

```
# 6 random rows of iris:
set.seed(124)
z <- iris[sample(1:nrow(iris), 6),]
z
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
65	5.6	2.9	3.6	1.3	versicolor
5	5.0	3.6	1.4	0.2	setosa
134	6.3	2.8	5.1	1.5	virginica
74	6.1	2.8	4.7	1.2	versicolor
143	5.8	2.7	5.1	1.9	virginica
91	5.5	2.6	4.4	1.2	versicolor

```
# order z by increasing Petal.Length
```

```
## returns indices
order(z$Petal.Length)
```

```
[1] 2 1 6 4 3 5
```

```
## uses those indices in []
## to produce sorted data frame
z[order(z$Petal.Length),]
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5	5.0	3.6	1.4	0.2	setosa
65	5.6	2.9	3.6	1.3	versicolor
91	5.5	2.6	4.4	1.2	versicolor
74	6.1	2.8	4.7	1.2	versicolor
134	6.3	2.8	5.1	1.5	virginica
143	5.8	2.7	5.1	1.9	virginica

```
## order by species (alphabetical ascending),
## then by petal length (ascending)
z[order(z$Species, z$Petal.Length),]
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5	5.0	3.6	1.4	0.2	setosa
65	5.6	2.9	3.6	1.3	versicolor
91	5.5	2.6	4.4	1.2	versicolor
74	6.1	2.8	4.7	1.2	versicolor
134	6.3	2.8	5.1	1.5	virginica
143	5.8	2.7	5.1	1.9	virginica

You can multiple variables to `order()`, by name, and R will order the rows in each variable in the order supplied. By default, records are sorted in ascending order. If you want a mix of ascending and descending order, you have to specify the `radix` method and specify whether you want the order to be decreasing (TRUE) or not (FALSE) for each variable:

```
# sort alpha by species, then
# by decreasing petal length
z[order(z$Species, z$Petal.Length,
       decreasing=c(TRUE, FALSE),
       method="radix"),]
```

5.5.2.7 Adding and removing rows

Removing rows works a lot like removing columns. You can use the bracket notation to either remove the rows with negative indices, or make a new version of the data frame with only some of the rows.

```
# spare copy with just a
# few rows for demonstration
x <- iris[1:10,]

# remove rows by their number
x[-1,]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2           4.9           3.0           1.4           0.2   setosa
## 3           4.7           3.2           1.3           0.2   setosa
## 4           4.6           3.1           1.5           0.2   setosa
## 5           5.0           3.6           1.4           0.2   setosa
## 6           5.4           3.9           1.7           0.4   setosa
## 7           4.6           3.4           1.4           0.3   setosa
```

```
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
x[-c(1:3),]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
x[-c(1, 3, 5, 7, 9),]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2      4.9      3.0      1.4      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa

# or, make a new copy with the
# rows to keep
x[2:10,]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
x[c(2, 4, 6, 8, 10),]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2      4.9      3.0      1.4      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
```

Adding rows can be done with `rbind()` (short for “row bind”). This function is analogous to `cbind()`, which joins objects by their columns.

Adding rows to a data frame is tantamount to combining it with another data frame. The data frames being combined *must have the same columns*, with the *same names*, but the *order of columns can be different*.

```
x <- iris[1:5,]
y <- iris[51:55,]

# columns match exactly:
rbind(x, y)
##      Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 1           5.1           3.5           1.4           0.2    setosa
## 2           4.9           3.0           1.4           0.2    setosa
## 3           4.7           3.2           1.3           0.2    setosa
## 4           4.6           3.1           1.5           0.2    setosa
## 5           5.0           3.6           1.4           0.2    setosa
## 51          7.0           3.2           4.7           1.4 versicolor
## 52          6.4           3.2           4.5           1.5 versicolor
## 53          6.9           3.1           4.9           1.5 versicolor
## 54          5.5           2.3           4.0           1.3 versicolor
## 55          6.5           2.8           4.6           1.5 versicolor

# columns in different order
# is still ok, but rbinding
# without making column orders
# match makes me nervous.
y2 <- y[,c(3,5,2,1,4)]
rbind(x,y2)
##      Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 1           5.1           3.5           1.4           0.2    setosa
## 2           4.9           3.0           1.4           0.2    setosa
## 3           4.7           3.2           1.3           0.2    setosa
## 4           4.6           3.1           1.5           0.2    setosa
## 5           5.0           3.6           1.4           0.2    setosa
## 51          7.0           3.2           4.7           1.4 versicolor
## 52          6.4           3.2           4.5           1.5 versicolor
## 53          6.9           3.1           4.9           1.5 versicolor
## 54          5.5           2.3           4.0           1.3 versicolor
## 55          6.5           2.8           4.6           1.5 versicolor

# result inherits order of columns from first argument:
```

```

rbind(y2,x)
##      Petal.Length      Species Sepal.Width Sepal.Length Petal.Width
## 51          4.7 versicolor          3.2          7.0          1.4
## 52          4.5 versicolor          3.2          6.4          1.5
## 53          4.9 versicolor          3.1          6.9          1.5
## 54          4.0 versicolor          2.3          5.5          1.3
## 55          4.6 versicolor          2.8          6.5          1.5
## 1           1.4      setosa          3.5          5.1          0.2
## 2           1.4      setosa          3.0          4.9          0.2
## 3           1.3      setosa          3.2          4.7          0.2
## 4           1.5      setosa          3.1          4.6          0.2
## 5           1.4      setosa          3.6          5.0          0.2

```

5.5.3 Other operations

5.5.3.1 Merging data frames

Data sets for a complex project might reside in several tables, related by **key variables**. These are variables that are shared by more than one table, and define which records in each table are related to records in another table. This is exactly what **relational databases** do, but most biologists have small enough datasets that proper databases aren't really necessary.

Data frames can be combined by manually matching records between tables, using the function `match()`. The example below uses three interrelated tables from a stream dataset I work with. You need the datasets [neon_grids_2025-08-03.csv](#), [neon_measures_2025-08-03.csv](#), and [neon_taxa_2025-08-03.csv](#). Download them, put them in a folder, and make that folder your R working directory.

```

# data file names
name1 <- "neon_grids_2025-08-03.csv"
name2 <- "neon_measures_2025-08-03.csv"
name3 <- "neon_taxa_2025-08-03.csv"

# import
grd <- read.csv(name1, header=TRUE)
mes <- read.csv(name2, header=TRUE)
tax <- read.csv(name3, header=TRUE)

# spare copies for later, so we can
# revert without having to reimport
grd.orig <- grd
mes.orig <- mes
tax.orig <- tax

```

Notice that the tables have only “key” variables in common. If you have > 2 tables, each table doesn’t necessarily have to share a variables with every other table. Each table only needs to share a variable with at least one other table.

```
intersect(names(grd), names(mes))
## [1] "grid"
intersect(names(grd), names(tax))
## character(0)
intersect(names(mes), names(tax))
## [1] "abb"
```

Data frames `grd` (trapping grid characteristics) and `mes` (animal measurements) share variable “grid”. Each animal in `mes` was captured on one trapping grid (`mes$grid`), and the characteristics (location, region, etc.) are stored in `grd`.

Data frames `mes` and `tax` are related by variable `abb`. Each animal measured in `mes` is identified by a four-letter abbreviation of its scientific name (`mes$abb`), usually the first 2 letters of its generic eptithet and the first 2 letters of its specific epithet⁵. Characteristics of that species are held in data frame `tax`.

So, if you wanted to tally up the mean body mass (`mes$mass`) for each species in each habitat in which it occurs, you would have to somehow get the habitat of each grid (`grd$nlcd`) from `grd` into `mes`. You could build the necessary data frame by merging `grd` and `mes`, using their common variable `grid` to make sure that data from one data frame are associated with the correct rows of the other.

A trickier problem would be tallying up how many animals in each taxonomic family (`tax$family`) were measured in each habitat (`grd$nlcd`)—because `tax` and `grd` have no variables in common! You would need to daisy chain the data frames together: first, join `grd` to `mes` by “grid”, then join the result to `tax` by `abb`.

There are two methods to accomplish these joins in base R: `match()` and `merge()`. I use a mix of both in my workflows:

- `match()` can be more efficient when only a single or a few columns are needed from each data frame.
- `merge()` can be more useful when all variables are needed.

Here is an example of using `merge()`:

```
# add habitat from grid table grd
# to measurement table mes
# by matching grid names
matchx <- match(mes$grid, grd$grid)
```

⁵(except where abbreviating this way would have created duplicate abbreviations).

```

mes$nlcd <- grd$nlcd[matchx]
head(mes)
##           grid           tag sex  abb mass totlen tail foot pinna
## 1 BART_015 NEON.MAM.D01.006039  F NAIN  24   231  140   30   14
## 2 HARV_021  NEON.MAM.D01.0270  F PEMA  24   158   87   21   13
## 3 BART_015  NEON.MAM.D01.3245  F PEMA  17   166   88   21   18
## 4 HARV_010 NEON.MAM.D01.977981  F MIPE  34    92   46   22   11
## 5 HARV_016 NEON.MAM.D01.L0003  M PEMA  26    74   88   19   14
## 6 HARV_016 NEON.MAM.D01.L0004  M PEMA  24   159   85   21   18
##           nlcd
## 1      mixedForest
## 2 evergreenForest
## 3      mixedForest
## 4      mixedForest
## 5    woodyWetlands
## 6    woodyWetlands

# mes now has habitat (nlcd), which
# will allow us to relate habitat to
# taxonomic family from table tax
matchz <- match(mes$abb, tax$abb)
mes$family <- tax$family[matchz]
head(mes)
##           grid           tag sex  abb mass totlen tail foot pinna
## 1 BART_015 NEON.MAM.D01.006039  F NAIN  24   231  140   30   14
## 2 HARV_021  NEON.MAM.D01.0270  F PEMA  24   158   87   21   13
## 3 BART_015  NEON.MAM.D01.3245  F PEMA  17   166   88   21   18
## 4 HARV_010 NEON.MAM.D01.977981  F MIPE  34    92   46   22   11
## 5 HARV_016 NEON.MAM.D01.L0003  M PEMA  26    74   88   19   14
## 6 HARV_016 NEON.MAM.D01.L0004  M PEMA  24   159   85   21   18
##           nlcd      family
## 1      mixedForest  Dipodidae
## 2 evergreenForest  Cricetidae
## 3      mixedForest  Cricetidae
## 4      mixedForest  Cricetidae
## 5    woodyWetlands  Cricetidae
## 6    woodyWetlands  Cricetidae

```

Matching many variables by hand can become very tedious and error-prone. An alternative, and usually faster way is the function `merge()`. This is a shortcut for the matching that we did manually above. The downside of using `merge()` is that it can be a little obtuse.

- `merge()` will attempt to match records exactly, which may or may not return the results you expect.
- Using `merge()` can also sometimes result in many duplicate columns or extra rows in the result.

Always inspect the results of `merge()` to make sure it is doing what you expect (i.e., that you wrote the code correctly to get what you wanted!).

The commands below duplicate the merging done piecemeal above.

```
# get the original versions
# of each data frame that we
# set aside earlier
grd <- grd.orig
mes <- mes.orig
tax <- tax.orig

# merge by the variables in common:
res <- merge(mes, grd)
res <- merge(res, tax)
head(res)
```

##	abb	grid	tag	sex	mass	totlen	tail	foot	pinna
## 1	BATA	CLBJ_041	NEON.MAM.D11.R1700	M	7	97	37	14	10
## 2	BATA	CLBJ_034	NEON.MAM.D11.R1599	F	11	95	18	12	10
## 3	BATA	CLBJ_041	NEON.MAM.D11.L2697	F	9	97	37	12	9
## 4	BATA	CLBJ_034	NEON.MAM.D11.R1597	F	7	91	42	13	10
## 5	BLBR	SCBI_039	NEON.MAM.D02.0039G220170915	U	13	87	26	14	13
## 6	BLBR	SCBI_035	NEON.MAM.D02.0035H7170617	U	13	84	25	13	13

##	domain	site	lat	lon	elev	nlcd	tsn
## 1	D11_Southern Plains	CLBJ	33.41225	-97.66223	302.3	grassland	Herbaceous 1
## 2	D11_Southern Plains	CLBJ	33.32251	-97.57807	325.0	grassland	Herbaceous 1
## 3	D11_Southern Plains	CLBJ	33.41225	-97.66223	302.3	grassland	Herbaceous 1
## 4	D11_Southern Plains	CLBJ	33.32251	-97.57807	325.0	grassland	Herbaceous 1
## 5	D02_Midatlantic	SCBI	38.89493	-78.16656	324.4	pasture	Hay 122928
## 6	D02_Midatlantic	SCBI	38.88219	-78.16998	324.0	pasture	Hay 122928

##	species	order	suborder	superfamily	family	subfamily
## 1	Baiomys taylori	Rodentia	Myomorpha	Muroidea	Cricetidae	Neotomina
## 2	Baiomys taylori	Rodentia	Myomorpha	Muroidea	Cricetidae	Neotomina
## 3	Baiomys taylori	Rodentia	Myomorpha	Muroidea	Cricetidae	Neotomina
## 4	Baiomys taylori	Rodentia	Myomorpha	Muroidea	Cricetidae	Neotomina
## 5	Blarina brevicaudus	Eulipotyphla	<NA>	<NA>	Soricidae	Soricinae
## 6	Blarina brevicaudus	Eulipotyphla	<NA>	<NA>	Soricidae	Soricinae


```
##      genus
## 1 Baiomys
```

```
## 2 Baiomys
## 3 Baiomys
## 4 Baiomys
## 5 Blarina
## 6 Blarina

# what if you only want some variables to be merged?
## specify which columns you want!
use.cols.grd <- c("grid", "nlcd")
use.cols.mes <- c("grid", "abb", "mass")
use.cols.tax <- c("abb", "genus")

res2 <- merge(mes[, use.cols.mes],
              grd[, use.cols.grd])
res2 <- merge(res2, tax[, use.cols.tax])
head(res2)
##      abb      grid mass      nlcd      genus
## 1 BATA CLBJ_041      7 grasslandHerbaceous Baiomys
## 2 BATA CLBJ_034     11 grasslandHerbaceous Baiomys
## 3 BATA CLBJ_041      9 grasslandHerbaceous Baiomys
## 4 BATA CLBJ_034      7 grasslandHerbaceous Baiomys
## 5 BLBR SCBI_039     13      pastureHay Blarina
## 6 BLBR SCBI_035     13      pastureHay Blarina
```

Some of the tidyverse packages offer alternative solutions for merging datasets that can be more or less elegant than `merge()`. However, the methods shown above use only base R functions and will get the job done.

5.5.3.2 Reshaping data frames

In your career you are likely to encounter datasets organized in a variety of formats. Most of these formats fall into one of two categories: **wide** and **long**.

Figure 5.10: Illustration of long format (left) and wide format (right) storage of the same values.

Long format			Wide format		
site	attribute	value	site	lat	long
site001	lat	33.7579	site001	33.7579	-82.3836
site002	lat	35.0121	site002	35.0121	-83.2067
site003	lat	34.952	site003	34.952	-83.5166
site004	lat	34.6649	site004	34.6649	-83.3698
site001	long	-82.3836			
site002	long	-83.2067			
site003	long	-83.5166			
site004	long	-83.3698			

In a **long-format dataset**, every *value* is stored in a separate *row*. The variable (aka: feature or attribute) that each value represents is kept in a separate *column*. Many public or online data sources serve datasets in long format. This is usually because the data are pulled from a database, and it left to the user to reshape the data to fit their needs. Storing data in this manner can sometimes be more efficient from a computational standpoint, but it is usually not convenient for analysis. The reason that long format is inconvenient for analysis is because all of the information about individual samples (i.e, observations) are on different rows of the dataset.

In a **wide-format dataset**, each row contains **all values** associated with a particular **sample**. This means that each variable gets its own column, and each cell contains a single value. This format is sometimes called **tidy** format by the authors of the `tidyverse` set of R packages (Wickham 2014), but researchers were formatting data this way long before the tidyverse packages or R were developed (Broman and Woo 2018). As a rule, you should store and manage your data in wide/tidy format. This format makes it easy to take advantage of R's vectorized nature. This is also the format expected by most functions that work with data frames.

Base R has some functions for reshaping data, but most sources recommend using the `reshape2` package or the `tidyverse`. The examples below demonstrate reshaping a dataset from a typical long format to a more useful wide format. You will need the file [reshape_example_2025-08-12.csv](#) in your R home directory and package `reshape2`.

```
library(reshape2)
in.name <- "reshape_example_2025-08-12.csv"
x <- read.csv(in.name)
head(x)
```

```
##           grid attribute      value
## 1 BART_062      lat    44.05735
## 2 BART_084      lat    44.048501
## 3 HARV_001      lat    42.422943
## 4 HARV_006      lat    42.401991
## 5 HARV_008      lat    42.443394
## 6 BART_062      lon   -71.308646
class(x$value)
## [1] "character"
```

Notice that the `value` column contains a mix of numeric and text values—so the numbers are stored as text. If we wanted to do something with those values, they need to be pulled out of the vector `value` and converted to numeric values. Because a vector can have only one type of value, the attributes will need to end up in their own columns. The fastest way to do this is to use the function `dcast()` from package `reshape2`.

```
dx <- dcast(x, grid~attribute)
dx$lat <- as.numeric(dx$lat)
dx$lon <- as.numeric(dx$lon)
dx
```

	grid	lat	lon	nlcd
1	BART_062	44.05735	-71.30865	deciduousForest
2	BART_084	44.04850	-71.29797	evergreenForest
3	HARV_001	42.42294	-72.25872	deciduousForest
4	HARV_006	42.40199	-72.25478	deciduousForest
5	HARV_008	42.44339	-72.22512	deciduousForest

If we needed to go the other way, from wide to long format, we use function `melt()`. This is essentially the reverse of `dcast()`.

```
mx <- melt(dx,
           id.vars="grid",
           measure.vars=c("lat", "lon", "nlcd"))

# default:
head(mx)
##           grid variable      value
## 1 BART_062      lat    44.05735
## 2 BART_084      lat    44.048501
## 3 HARV_001      lat    42.422943
```

```
## 4 HARV_006      lat  42.401991
## 5 HARV_008      lat  42.443394
## 6 BART_062      lon -71.308646

# sort by grid, then variable:
head(mx[order(mx$grid, mx$variable),])
##           grid variable      value
## 1  BART_062      lat        44.05735
## 6  BART_062      lon       -71.308646
## 11 BART_062      nlcd deciduousForest
## 2  BART_084      lat        44.048501
## 7  BART_084      lon       -71.297965
## 12 BART_084      nlcd evergreenForest
```

The tidyverse equivalents for `melt()` and `dcast()` are `pivot_longer()` and `pivot_wider()`, respectively. The syntax of these tidyverse functions are similar to the `reshape2` functions. Note that the tidyverse functions will output a **tibble**, which is a form of data frame peculiar to the tidyverse. Some non-tidyverse functions can use tibbles, and some cannot. Tibbles can be converted to base R data frames with function `data.frame()`.

```
library(tidyr)

# long to wide:
# eqv. to reshape2::dcast
tdx <- pivot_wider(x,
                   names_from="attribute",
                   values_from="value")

head(tdx)
## # A tibble: 5 x 4
##   grid      lat      lon      nlcd
##   <chr>   <chr>   <chr>   <chr>
## 1 BART_062 44.05735 -71.308646 deciduousForest
## 2 BART_084 44.048501 -71.297965 evergreenForest
## 3 HARV_001 42.422943 -72.258721 deciduousForest
## 4 HARV_006 42.401991 -72.254785 deciduousForest
## 5 HARV_008 42.443394 -72.225117 deciduousForest

# wide to long:
# eqv. to reshape2::melt
tmx <- pivot_longer(tdx,
                    cols=c("lat", "lon", "nlcd"),
```

```

names_to="variable")
head(tmx)
## # A tibble: 6 x 3
##   grid      variable value
##   <chr>      <chr>    <chr>
## 1 BART_062 lat      44.05735
## 2 BART_062 lon     -71.308646
## 3 BART_062 nlcd    deciduousForest
## 4 BART_084 lat      44.048501
## 5 BART_084 lon     -71.297965
## 6 BART_084 nlcd    evergreenForest

# if needed:
tdx.df <- data.frame(tdx)

```

Part III

Exploratory data analysis with R

If it looks like a duck, and quacks like a duck, it's not a normal distribution.

Chapter 6

Basic descriptive statistics

One of the first steps in exploratory data analysis is understanding how values in a variable are spread out (aka: distributed), and what a “typical” value is.

Biologists use a few common **summary statistics**, or **descriptive statistics**, to describe variables. What makes descriptive statistics is that they are calculated directly from the data, and do not estimate any unknown parameters. Estimating unknown parameters is the realm of **inferential statistics**. Any method that tests a hypothesis—such as a *t*-test, ANOVA, linear regression, etc.—is an inferential technique.

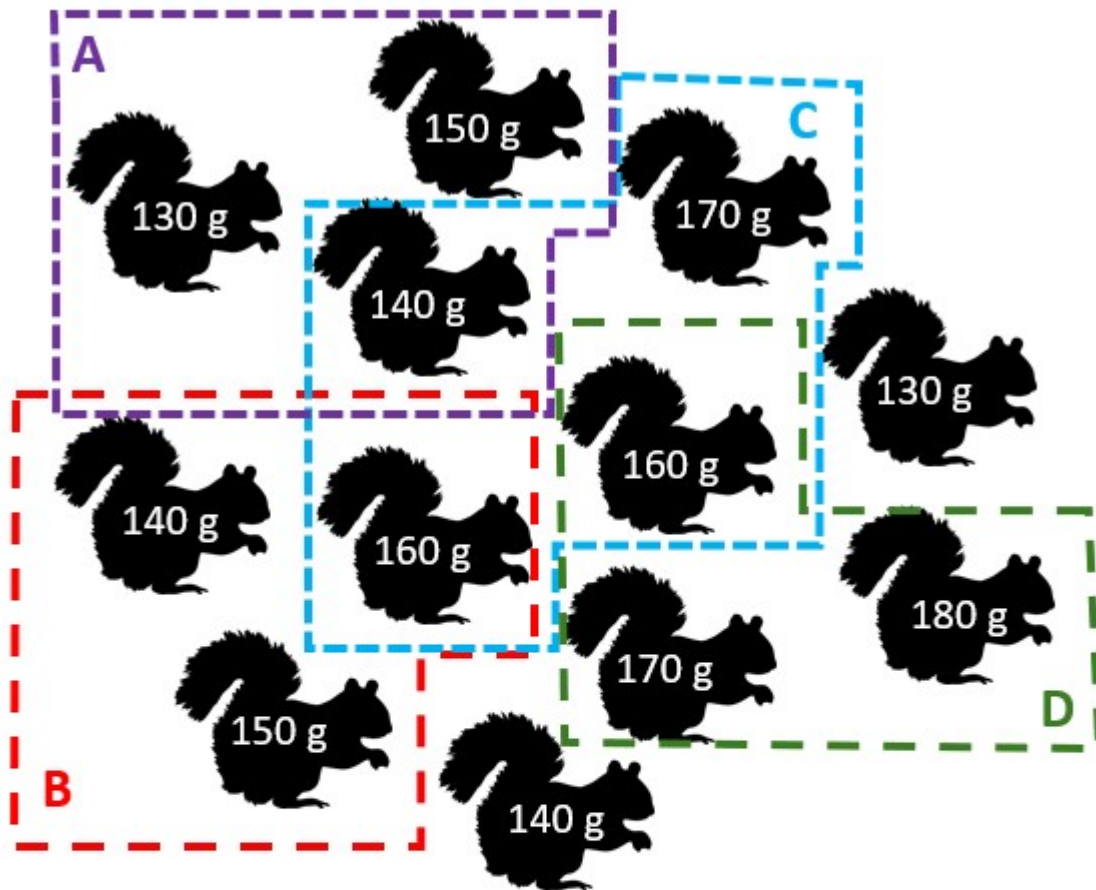
6.1 Sample and population statistics

In biology, we never have all of the possible values which could be observed. For example, if you wanted to study the body masses of squirrels, you could go and collect, say, 30 squirrels and weigh them. However, those 30 are only a small fraction of the millions of squirrels alive in the world. You would be assuming that your 30 squirrels somehow represent all squirrels.

- The values that are observed are **samples**, and they are summarized by **sample statistics**.
- The set of all possible values that could be observed, even if only in principle, are the **population**. **Population parameters** or **population statistics** describe this set of all possible values.

In science, we usually want to *estimate* population parameters, but these are unobservable. Instead, we use sample statistics to estimate the parameters of the underlying population.

Figure 6.1: Illustration of how sampling bias can affect estimates of population parameters like the sample mean $\bar{x}$ and sample standard deviation s .



Sample A: 140 ± 10 g

Sample B: 150 ± 10 g

Sample C: 157.5 ± 12.6 g

Sample D: 170 ± 10 g

Population: 151.7 ± 16.4 g

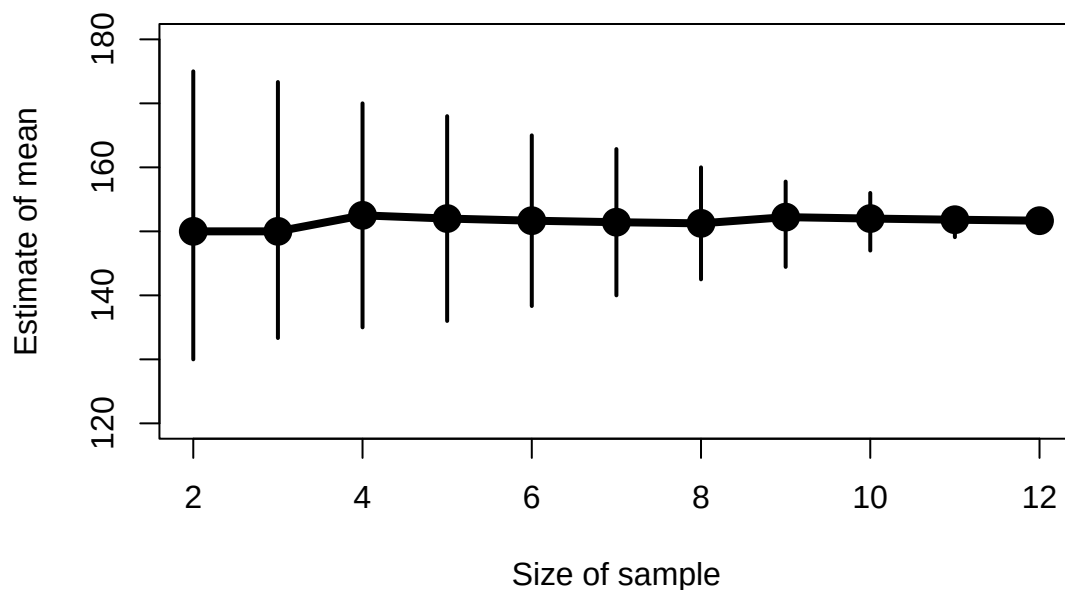
In the figure above, the *population mean* of all squirrels is 151.7 g. However, finding all squirrels in an area (much less the world) is not practical. So, researchers took 4 samples of a few squirrels each, and obtained *sample means* of 140, 150, 157.5, and 170 g. Notice that the mean of these sample means, 154.38 g, is pretty close to the population mean. In fact, the more squirrels you include in your sample, the less uncertain your estimate of the mean will be. In the figure below, notice that as the size of the sample increases, the range of estimates of the mean shrinks.

```

all.wts <- c(130, 150, 170,
            140, 140, 160,
            160, 130, 150,
            170, 180, 140)
npop <- length(all.wts)
ns <- 2:npop
ests <- vector("list", length(ns))
for(i in 1:length(ns)){
  ests[[i]] <- combn(all.wts, m=ns[i], FUN=mean)
}

mins <- sapply(ests, min)
maxs <- sapply(ests, max)
meds <- sapply(ests, median)
plot(ns, meds, type="o", pch=16,
     ylim=c(120, 180), cex=2, lwd=4,
     ylab="Estimate of mean", xlab="Size of sample")
segments(ns, mins, ns, maxs, lwd=2)

```



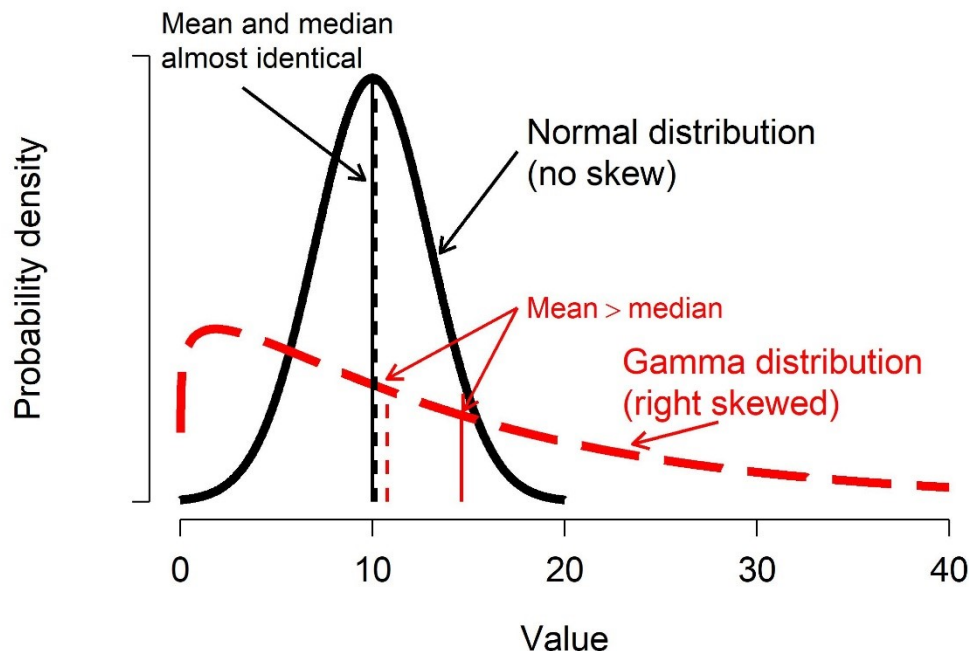
6.2 Central tendency

The **central tendency** of a variable is a value that represents a “typical” value from that set of values. Central tendency is usually expressed in one of three ways:

- **Mean:** the sum of values divided by the number of values. This value is also called the “**arithmetic mean**”. Answers the question, “*What value would all values be if they were the same?*”. When people say “mean” or “average”, this usually what they are referring to. The mean is useful because it is one of the defining features of the **normal distribution**, a foundational concept in modern statistics.
- **Median:** the middle value, which is greater than half of the values and less than the other half of the values. Also known as the 50th percentile or 0.5 quantile. Answer the question, “*If you lined up the values from smallest to largest, what value would be in the middle?*”.
- **Mode:** the most common value.

For many datasets, the mean and median will be approximately the same. Data that are **skewed** will have a median different from their mean.

Figure 6.2: Illustration of skew. The solid black line shows an unskewed distribution that is roughly symmetrical about its mean, with its mean and median almost identical. The dashed red line shows a right-skewed distribution that has mostly small values and very few large values. Its median is very different from its mean.



Biologists typically summarize data using the mean, but median can be appropriate in some situations—especially when data are not normally distributed.

6.2.1 Means

6.2.1.1 Arithmetic mean

The **arithmetic mean** or just **mean** is the value that all values would be if they were all the same. Biologists consider both the **sample mean** $\bar{x}$ and the **population mean** μ .

- The **population mean** μ is the mean of all possible observations, and defines the underlying distribution of values. This can be a parameter of a distribution, but is usually unknown in practice and has to be estimated from data.
- The **sample mean** is $\bar{x}$ the mean of observed values.

The sample mean is calculated as the sum of all values in x divided by the number of values n :

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

The mean is useful because it is part of the definition of the normal distribution, a very important concept in modern statistics.

You can get the mean of a variable in R using the function `mean()`:

```
# random normal dist. with mean = 5 and sd = 1
x <- rnorm(100, 5, 1)
mean(x)
```

```
[1] 5.056531
```

Like many summarization functions in R, `mean()` does not automatically handle missing values (NA). If your dataset contains missing values, you need to set the argument `na.rm=TRUE`. This can seem like an annoying bug, but it's actually a feature. By returning NA when the data contain missing values, R is helping you track down where missing values are coming from (and thus, why your code may not be working at some later point).

```
x <- c(2, 3, 6, 2, 6, 1, NA)
mean(x)
## [1] NA
mean(x, na.rm=TRUE)
## [1] 3.333333
```

While most people only ever use the arithmetic mean, there are other means.

6.2.1.2 Geometric mean

The **geometric mean** is defined as the n -th root of the product of all values:

$$GM(x) = \sqrt[n]{\prod_{i=1}^n x}$$

Or equivalently, as Euler's constant e raised to the power of the mean of the values on the log scale¹:

$$GM(x) = \exp\left(\frac{1}{n} \sum_{i=1}^n \log(x)\right)$$

The geometric mean is sometimes used for values that are meant to be multiplied together, such as growth rates. One application is in comparative morphology, where the geometric mean of a set of anatomical measurements is used as a sort of overall measure of body size against which individual measurements can be relativized. For example, a morphologist might calculate the geometric mean of 15 skull measurements to arrive at the size of a “hypervolume” which represents skull size.

Because of the way that it is defined, the geometric mean is only defined for positive real values.

There is no base function for geometric mean, but it's easy to get by combining base functions.

```
# lognormal with logmean 2 and logsd=0.5
x <- rlnorm(10, 2, 0.5)

# custom function for geometric mean:
geomean <- function(x) {exp(mean(log(x)))}

# test it out:
geomean(x)
## [1] 5.958454

# another way:
geomean2 <- function(x) {prod(x)^(1/length(x))}
geomean2(x)
## [1] 5.958454
```

6.2.1.3 Harmonic mean

The **harmonic mean** is less common than the arithmetic or geometric means. It is used when an average *rate* is needed. The harmonic mean is the reciprocal of the arithmetic mean of reciprocals of the values:

¹Natural log. Unless otherwise indicated, always assume a mathematician or statistician means natural log.

$$HM(x) = \left(\frac{1}{n} \sum_{i=1}^n x^{-1} \right)^{-1}$$

An example use case for the harmonic mean is as follows: a migrating bird travels from Atlanta to Knoxville, Tennessee (250 km) at an average speed of 60 km/h, and then from Knoxville to Lexington, Kentucky (230 km) at an average speed of 52 km/h. Its average speed on the trip is not the arithmetic mean of the speeds, but the harmonic mean:

$$HM(v) = \frac{\text{Total distance}}{\text{Total time}} = \frac{2}{\frac{1}{60} + \frac{1}{52}} \approx 55.7 \text{ km/h}$$

I'm not aware of a base function for harmonic means, but it is easy to get:

```
# only defined for postive real values
a <- rlnorm(20, 3, 1)

# custom function to get harmonic mean
har.mean <- function(x) {1/(mean(1/x))}

# compare mean, geometric mean, and harmonic mean:
mean(a)
## [1] 30.75888
geomean(a)
## [1] 20.89664
har.mean(a)
## [1] 13.71774
```

Interestingly, for any set of positive values with at least 2 unique values, the following inequality will always be true:

$$AM(x) > GM(x) > HM(x)$$

6.2.1.4 Weighted mean

Sometimes we need to some observations to count more than others. The **weight** of an observation is the amount that it contributes to a statistic or analysis. A **weighted mean** is useful when some observations represent more data or experimental units.

For example, when trying to calculate the mean mass of elephants across multiple populations, it might make sense to calculate the mean of the population-level means, but weight the population means by the number of elephants measured in each population. This would be done because the populations with more

measurements represent more data, and thus give us greater confidence about what the true mean might be.

The R function for the weighted mean takes a set of values `x` and weights `w`. The weights in `w` must add up to 1.

```
# make some values
x <- c(14, 3, 5, 4)

# weights
# defining as fractions over total
# sample size will guarantee sum(wts)==1
wts <- c(10, 1, 1, 2)/14

# compare mean to weighted mean
mean(x)
## [1] 6.5
weighted.mean(x, wts)
## [1] 11.14286
```

6.2.2 Median

The **median** is the central value of a set of values. If you were to rank the values from smallest to largest, the median would be the value in the middle (or, if there were an even number of values, the median would be the mean of the two central values). The median is also known as the 50th percentile, or 0.5 quantile.

The median is sometimes a more useful measure of central tendency than the mean when data are strongly skewed. A classic use case is personal income. In 2020, the US median household income was about \$67000. It's hard to find exact numbers, but at one point in 2020 Jeff Bezos had made about \$13 billion since 2019. This implies that in a room with 99 “average” Americans and 1 Jeff Bezos, the *mean* annual income was about \$130 million. Is that a good representation of what most households made?

Of course not. However, the median would be about \$67000, which would be more representative.

The easiest way to get the median in R is with the `median()` function. You can also use `quantile()` with argument `probs` (the second argument) set to 0.5.

```
x <- rnorm(100)
median(x)
## [1] -0.111393
quantile(x, 0.5)
##      50%
## -0.111393
```


6.2.3 Mode

The **mode** is the *most common value* in a set of values. For many distributions, the mode will be the same as, or close to, the mean and median. Most distributions have one mode, or are **unimodal**. Another common case is distributions with two modes, or are **bimodal**.

There is no base function to get the mode, but it can be obtained in several ways by combining base functions.

```
# generate some random values in 1:8
set.seed(123)
a <- sample(1:8, 20, replace=TRUE)

# simple function for mode
# Will return >1 result if >1 mode.
# Capitalized because already a function called "mode" that
# does something else and don't want to overwrite it
Mode <- function(x) {
  un <- sort(unique(x))
  ta <- table(x)
  mx <- max(ta)
  return(un[which(ta==mx)])
}
Mode(a)
## [1] 3

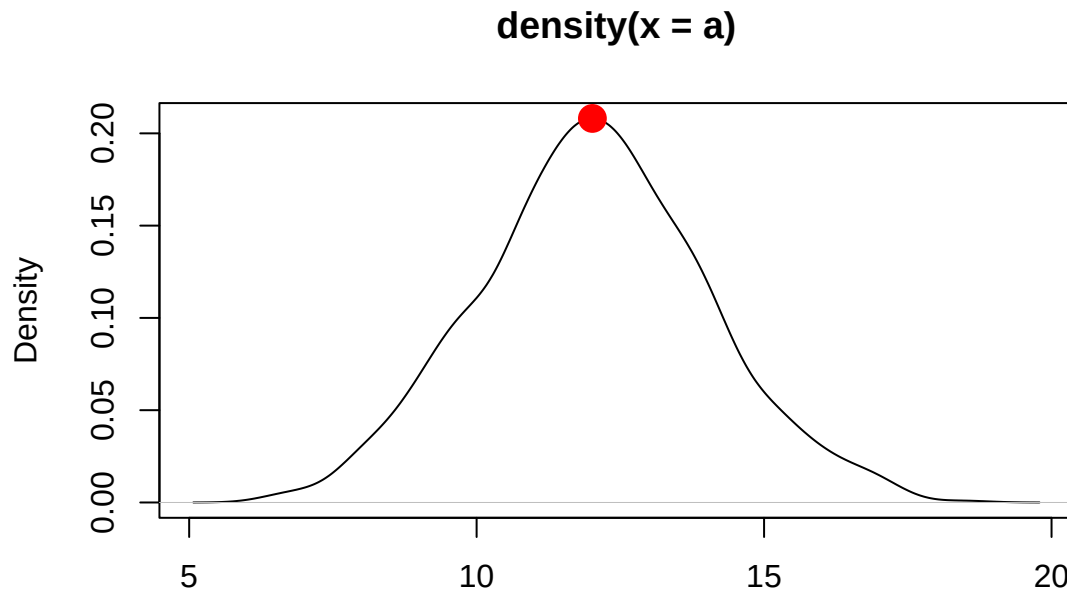
# illustration with 2 modes:
b <- c(4,4,4,3,3,3,2,1)
Mode(b)
## [1] 3 4
```

Another way to get the mode is to find peaks on a kernel density plot. We'll discuss what these plots really mean later in Section 8.1.5, but for now, a kernel density plot shows how likely different values are a distribution. This method might be more useful when there are many values, or the values have a great deal of precision, making it hard to find a mode by counting up occurrences.

In the example below, the distribution has a peak around $x = 12.01$. This means that the mode is about 12.01, which is very close to the mean and median as well as the true mean used to generate the distribution.

```
set.seed(123)
a <- rnorm(1e3, 12, 2)
den <- density(a)
plot(den)
```

```
points(den$x[which.max(den$y)], max(den$y),
       pch=16, col="red", cex=2)
```



N = 1000 Bandwidth = 0.4363

```
# function to estimate mode
Mode2 <- function(x) {
  de <- density(x)
  return(de$x[which.max(de$y)])
}
Mode2(a)
```

```
[1] 12.0136
```

The function we made, `Mode2()`, works well enough if there is only one mode.

But, many real life distributions have > 1 mode. Such distributions are called **bimodal** if they have 2 modes, or **multimodal** if they have 3 or more. A classic example of bimodality is grades in college courses: it is very common to have many Bs, and many Ds, but not many Fs, Cs, or As.

If your data appear to have > 1 mode, you can estimate them by finding points where the slope of the density plot changes sign. The code below is inelegant but it works.

```

set.seed(123)
a1 <- rnorm(1e2, 12, 2)
a2 <- rnorm(1e2, 20, 2)
a <- c(a1, a2)
den <- density(a)

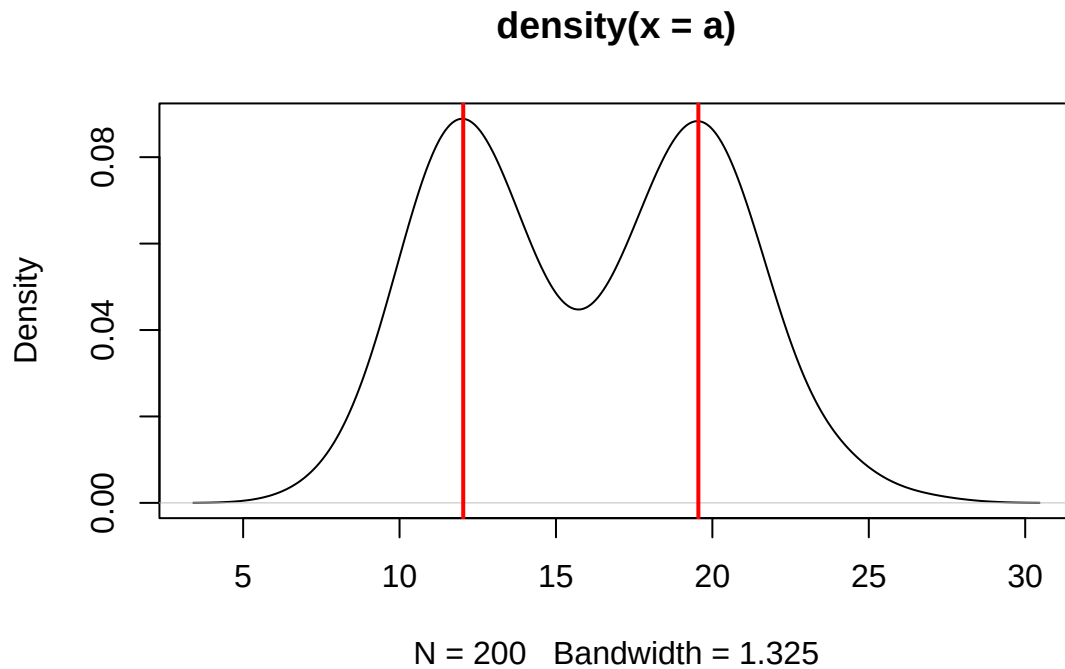
# shows bimodal distribution:
plot(den)

# function to find peaks on density plot
# i.e., where slope changes from positive to negative
Mode3 <- function(x){
  de <- density(x)
  dif <- diff(de$y)
  flag <- numeric(length(dif))
  for(i in 2:length(dif)){
    if(dif[i-1] >= 0 & dif[i] < 0){flag[i] <- 1}
  }#i
  return(de$x[which(flag==1)])
}
Mode3(a)

[1] 12.03519 19.55226

abline(v=Mode3(a), lwd=2, col="red")

```



6.3 Variability

6.3.1 Variance and standard deviation (SD)

Variability, both random and otherwise, is the *sine qua non* of statistics. If it weren't for all the uncertainty and variability inherent in experimental and real world data, data analysis would be a lot easier and we would barely [need a whole field for it](#). Understanding variability starts with summarizing it.

The classical way to express the variability in a dataset is the **variance**. Just like with the mean, we use different symbols for different variances:

- The **population variance** σ^2 ("sigma squared") defines the normal distribution, and describes the spread of all possible observations.
- The **sample variance** s^2 ("s squared") is the variance of a set of observations (i.e., a sample).

The variance is defined as the expected (mean) squared deviation from the mean:

$$\sigma^2(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

The sample variance of a set of observations is calculated the same way, except the sample mean $\bar{x}$ replaces the population mean μ in the expression.

Like the mean μ , the variance σ^2 is part of the definition of the normal distribution and so it shows up a lot in statistics.

The differences between each value x_i and the mean are squared for two reasons: first, so that positive and negative deviations do not cancel out (because the square of any real number is positive); and second, so that greater deviations count more than small deviations.

The main drawback of the variance is that it is in *squared* units of the original values. For example, the variance of a set of length measurements taken in cm will be in units of cm^2 . Trying to think about lengths in units of area is kind of silly.

For this reason, the variance is often expressed as its square root, the **standard deviation (SD)**. As with the variance and mean, we use different symbols to distinguish between the **population SD** σ (“sigma”) and the **sample SD** s .

The population SD is defined as:

$$\sigma(x) = \sqrt{\sigma^2(x)}$$

The expression for the sample SD is the same, except that the sample variance s^2 replaces the population variance σ^2 . Because the SD is the square root of the variance, it is in the same units as the original measurements (and thus, the mean). This makes SD easier to interpret than the variance.

For even more convenience, the **coefficient of variation (CV)** is the ratio of the SD to the mean. Some people prefer CV to SD because it allows comparisons between variables with very different means or SD. CV can be expressed as ratios or as percentages. Whichever you use, just be clear about what you did.

```
set.seed(123)
# different SD, same CV:
a <- rnorm(1000, 4, 2)
b <- rnorm(1000, 20, 10)

# function for CV:
cv <- function(x, pct=TRUE) {
  res <- sd(x)/mean(x)
  if(pct){
    return(100*res)
  } else {
    return(res)
  }
}

sd(a)
## [1] 1.98339
```

```

cv(a)          # as percent
## [1] 49.1881
cv(a, FALSE) # as ratio
## [1] 0.491881

sd(b)
## [1] 10.09674
cv(b)          # as percent
## [1] 49.43409
cv(b, FALSE) # as ratio
## [1] 0.4943409

```

6.3.1.1 Do not calculate the standard error (SE)

Another statistic related to the variance is the **standard error (SE)**, sometimes also called the “standard error of the mean (SEM)”. It is calculated as:

$$SE(x) = \frac{s(x)}{\sqrt{n}}$$

In other words, the SE is the ratio of the sample SD to the square root of the sample size.

There is no built-in function for **standard error (SE)** so it must be calculated using the sample size (function `length()`). Note that you should probably never compute SE because SD is more appropriate in almost every situation where you would think to calculate it. This is because the **SE is a statement of uncertainty about a mean or point estimate**, whereas **SD is a statement about variability among values**.

If you insist on calculating SE, here is how:

```

a <- rnorm(50)
var(a)
## [1] 0.8029621
sd(a)
## [1] 0.8960815
sqrt(var(a)) # equivalent to sd(a)
## [1] 0.8960815

# standard error of the mean
sd(a)/sqrt(length(a))
## [1] 0.1267251

```

Again, you almost certainly should **not** calculate a SE.

6.3.2 Median absolute deviation (MAD)

An statistic called **median absolute deviation (MAD)** expresses variation as *absolute* deviations from the median rather than *squared* deviations from the mean. The MAD is useful when data do not follow a normal distribution, or any of the many bell-shaped distributions. The MAD is also considered more robust to outliers than the SD. “Robust” in this context means “not as influenced by outliers”.

The MAD is similar to the interquartile range (see below).

The MAD can be obtained with the R function `mad()`.

```
# generate some skewed data
x <- rlnorm(100, 5, 1)
mad(x)
## [1] 92.2208

# compare to:
sd(x)
## [1] 143.226
```

6.3.3 Quantiles and ranges

A **quantile** is a value in a distribution or sample such that some specified proportion of observations are less than or equal to that value. For example, the 0.5 quantile is the value such that half of values are less than or equal to that value. Quantiles are usually expressed either as proportions in $[0, 1]$ or as percentiles in $[1, 100]$. R uses proportions to express quantiles, but you can always multiply by 100 to get percentiles.

Some ways of dividing up a distribution have special names:

- **Quartiles** divide a distribution into quarters.
- **Terciles** divide a distribution into thirds.
- **Quintiles** divide a distribution into fifths.
- **Deciles** divide a distribution into tenths.
- **Percentiles** divide a distribution into one-hundredths.

Some quantiles themselves have special names:

- The **minimum** is the 0th percentile or 0.0 quantile.
- The **maximum** is the 100th percentile or 1.0 quantile.
- The **median** is the 50th percentile or 0.5 quantile.

- The **first quartile** and **third quartile** are the 25th and 75th percentiles, respectively.

Function `range()` reports the minimum and maximum values in a vector. Like many of the functions for descriptive statistics, this function does not handle missing (NA) values by default. The argument `na.rm=TRUE` must be set if there are missing values.

```
range(iris$Petal.Length)
## [1] 1.0 6.9

# illustration of working with missing values
x <- c(rnorm(20), NA)
range(x)
## [1] NA NA
range(x, na.rm=TRUE)
## [1] -2.256535 1.714763
```

Quantiles of data are calculated using the function `quantile()`.

Special quantiles are also available as their own functions: minimum `min()`, median `median()`, and maximum `max()`. Remember that argument `na.rm` is needed for many of these functions if there are missing values!

```
a <- rnorm(50)

# 20th percentile (aka: 1st quintile)
quantile(a, probs=0.2)
##          20%
## -0.6203262

# 75th percentile (aka: 3rd quartile)
quantile(a, probs=0.75)
##          75%
## 0.6651086

# 2.5 and 97.5 percentiles
# (i.e. 95% CI of a normal distribution)
quantile(a, probs=c(0.025, 0.975))
##          2.5%          97.5%
## -1.663476  1.706863

# minimum, or 0th percentile
min(a)
```



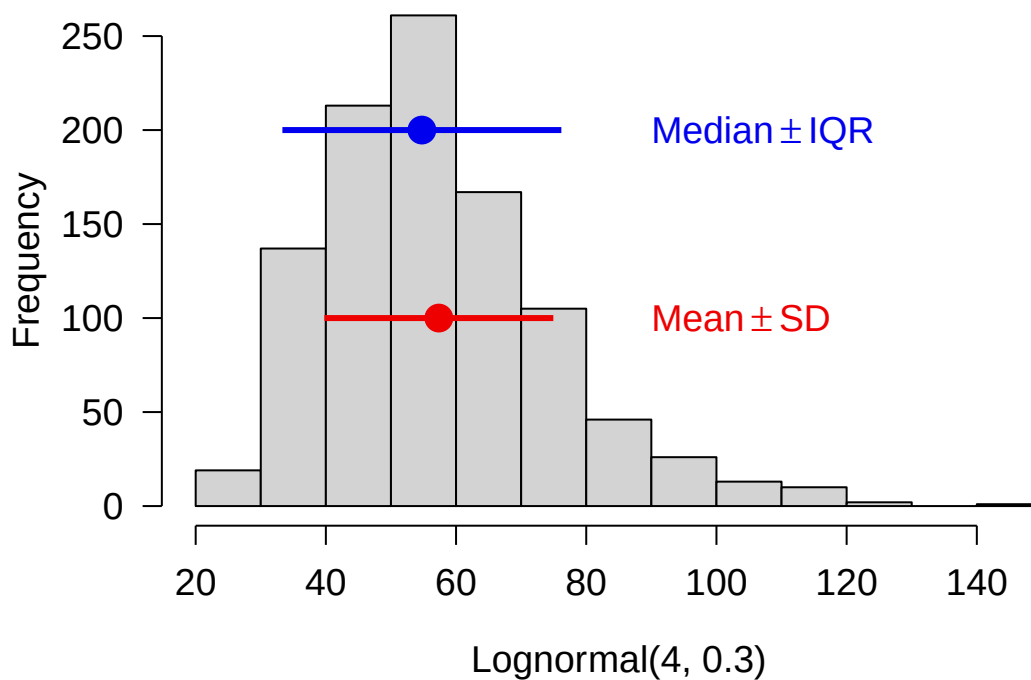
```
## [1] -1.948787

# maximum, or 100th percentile
max(a)
## [1] 2.281967

# demonstration of na.rm
b <- c(a, NA)
min(b)
## [1] NA
min(b, na.rm=TRUE)
## [1] -1.948787
```

If you are using the median to summarize a set of values, it is probably more appropriate to report their variability using the interquartile range (IQR) rather than the SD. This will likely come up with data that are right-skewed. Such data are often normally distributed on the log scale, but log-scale summary statistics aren't very useful for a reader.

Notice in the figure below how the mean and median differ.



The **range** of a variable is the interval defined by its minimum and maximum. You can get it with function `range()`. As always, you may need to handle missing values with `na.rm=TRUE`.

```
# range of a variable in a data frame
range(iris$Petal.Length)
```

```
[1] 1.0 6.9
```

```
# make a vector with some numbers and a missing value
a <- c(rnorm(10), NA)

# returns NA
range(a)
```

```
[1] NA NA
```

```
# returns range of non-NA values
range(a, na.rm=TRUE)
```

```
[1] -2.549343 2.416207
```

6.4 Covariance and correlation

If anyone knows anything about statistics, or thinks they do, they know the phrase “Correlation does not equal causation”. And that statement is true, both literally and philosophically. **Correlation** means that two sets of values tend to change together: larger values of one variable are associated with larger values of the other variable. Or, large values of one variable are associated with smaller values of the other variable. All this means is that the values seem to reliably change together. Actually demonstrating that one variable is causing another takes careful observation, experiments, and a different kind of analysis than correlation.

Despite the fact that correlation does not necessarily imply causation, a correlation can indicate that something interesting is going on. Maybe one variable really is causing the other. Maybe both variables are secretly correlated with a third variable, which explains them both. For example, in animals the length of the tibia may be correlated with the mass of the hippocampus. What do these have to do with each other? Probably nothing. So why are they correlated?

Because bigger animals have bigger parts...just because they are bigger! Both tibia length and hippocampus mass probably correlate well with body length, or body mass, measures of *overall* size. Here noticing a correlation didn't get us anywhere. We had to use biological reasoning to interpret the numerical result.

Correlation comes in two flavors. The first, **linear correlation**, assumes that a relationship between two variables is linear—that is, a plot of one variable against another approximates a straight line. The second flavor, **nonlinear correlation**, assumes only that two variables change together and does not assume any particular shape to that relationship.

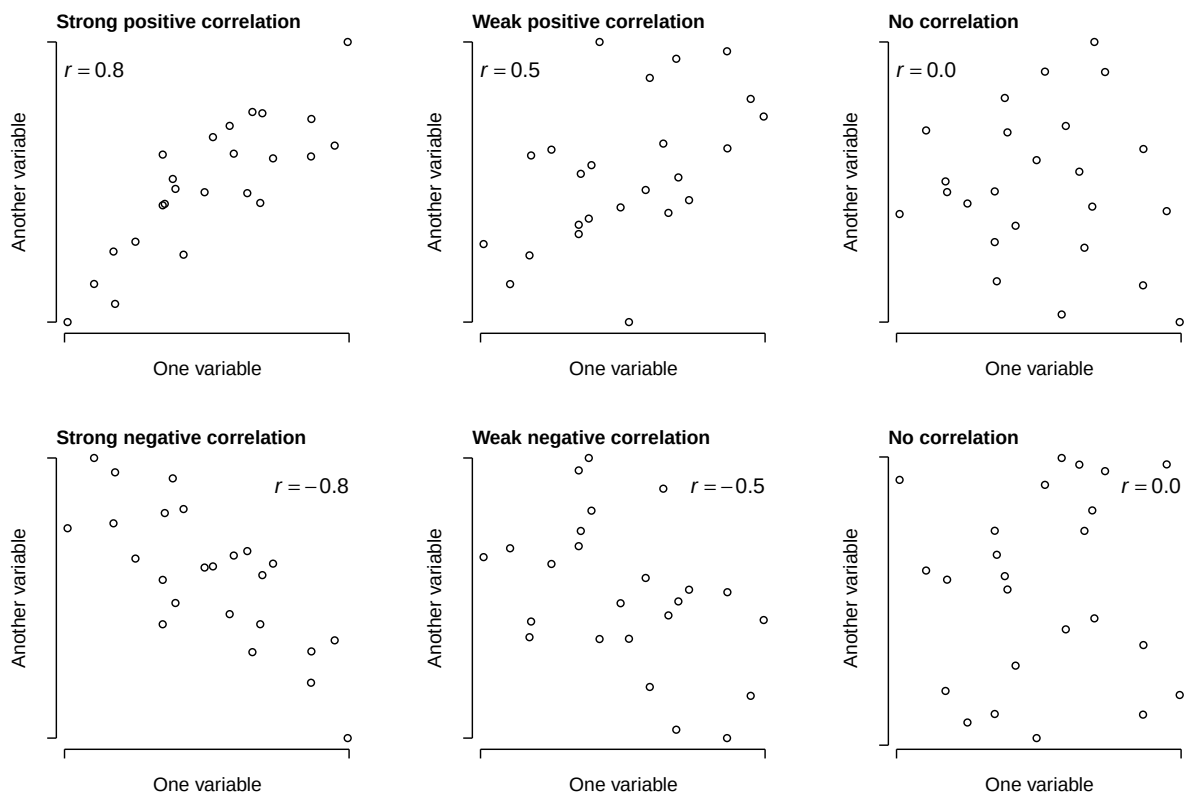
Correlation is a statistical relationship where changes in one variable are reliably associated with changes in another variable. This relationship need not be causal, hence the famous mantra, “correlation does not equal causation”. As such, correlation is usually a **descriptive technique**, which lets researchers characterize patterns, rather than an **inferential technique**, which lets researchers make estimates about unknown parameters (i.e., infer unobserved quantities).

Correlations are typically expressed as a **correlation coefficient**, a value that indicates the strength of the correlation by its magnitude and the direction of the correlation by its sign.

- **Positive correlation:** as one variable gets larger, the other variable does too. Correlation coefficient approaches $+1$.
- **Negative correlation:** as one variable gets larger, the other variable gets smaller. Correlation coefficient approaches -1 .

When a correlation coefficient is near 0, this indicates no correlation.

Figure 6.3: Illustration of possible positive and negative relationships between pairs of variables, with different levels of consistency (r).



6.4.1 Linear correlation

6.4.1.1 Correlations between 2 variables

When we suspect that two variables are correlated, and that the shape of that relationship is a straight line, we can use **Pearson's product moment correlation**, also called **Pearson's r** .

```
# calculate linear correlation coefficient
cor(iris$Petal.Length, iris$Petal.Width)
```

```
[1] 0.9628654
```

You can also request a significance test on the coefficient, although some might consider this unnecessary because correlation is inherently a descriptive technique. If you are using correlation as a hypothesis test, however, you do need a p -value.

```
cor.test(iris$Petal.Length, iris$Petal.Width)
```

```
Pearson's product-moment correlation
```

```
data: iris$Petal.Length and iris$Petal.Width
t = 43.387, df = 148, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9490525 0.9729853
sample estimates:
      cor
0.9628654
```

6.4.1.2 Correlations between many pairs of variables

One of the neat features of `cor()` is that you can get a correlation matrix showing the correlation coefficients between many pairs of variable at once.

```
# correlation matrix between the numerical variables in iris
cor(iris[,1:4])
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Sepal.Length	1.0000000	-0.1175698	0.8717538	0.8179411
Sepal.Width	-0.1175698	1.0000000	-0.4284401	-0.3661259
Petal.Length	0.8717538	-0.4284401	1.0000000	0.9628654
Petal.Width	0.8179411	-0.3661259	0.9628654	1.0000000

Here we see that each variable is perfectly correlated with itself (the 1s on the diagonal). Sepal length and sepal width are negatively correlated (-0.1175698) while petal length and petal width are strongly positively correlated (0.8717538). You only need to report correlation coefficients to 2 or 3 decimal places.

There isn't a built-in way to check for significant correlations between many pairs of variables at once. This is because `cor.test()` only works with one pair at a time. Fortunately, it's not hard to do in a loop:

```
# data frame with just the numeric variables
dx <- iris[,1:4]

# set up data frame to hold results
# combn(..., 2) returns matrix of all unique pairs of names
# t(ranspose) this and turn into a data frame.
cor.df <- as.data.frame(t(combn(names(dx), 2)))
cor.df$r <- NA
cor.df$p <- NA

# find the r and p values in a loop
for(i in 1:nrow(cor.df)){
  # test once per iteration
  curr <- cor.test(dx[,cor.df$V1[i]], dx[,cor.df$V2[i]])
  # pull out values we need from correlation test result
  # and round to 3 decimal places
  cor.df$r[i] <- round(curr$estimate, 3)
  cor.df$p[i] <- round(curr$p.value, 3)
}#close i loop

# print result
cor.df
```

	V1	V2	r	p
1	Sepal.Length	Sepal.Width	-0.118	0.152
2	Sepal.Length	Petal.Length	0.872	0.000
3	Sepal.Length	Petal.Width	0.818	0.000
4	Sepal.Width	Petal.Length	-0.428	0.000
5	Sepal.Width	Petal.Width	-0.366	0.000
6	Petal.Length	Petal.Width	0.963	0.000

6.4.1.3 Scatterplot matrices

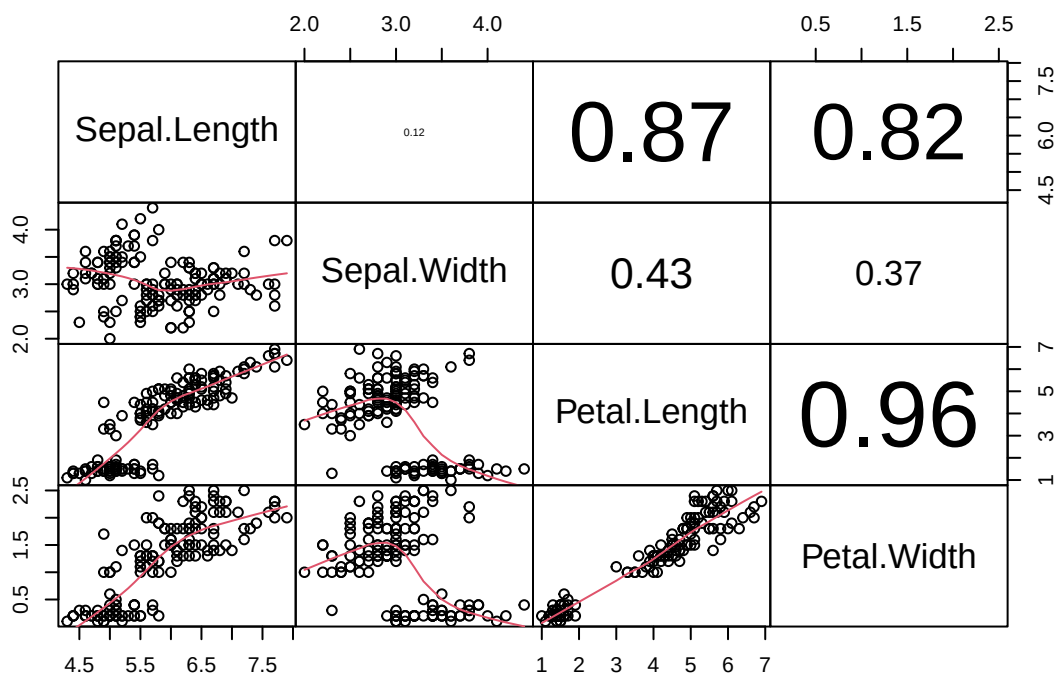
If you want to visualize many potential correlations at once, you can construct a **scatterplot matrix** using function `pairs()`. This will plot every pair of variables in a data frame. We are going borrow a function

called `panel.cor()` from the `pairs()` help page so that we get both the scatterplots *and* the linear correlation coefficients.

```
# from ?pairs
# just paste this in once to define the function
# try changing some arguments to change the behavior!
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  par(usr = c(0, 1, 0, 1))
  r <- abs(cor(x, y))
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex.cor * r)
}

# make the scatterplot matrix
pairs(iris[,1:4], lower.panel=panel.smooth, upper.panel=panel.cor, gap=0)
```

Figure 6.4: Scatterplot matrix showing possible correlations between the numeric variables in the `iris` data frame.



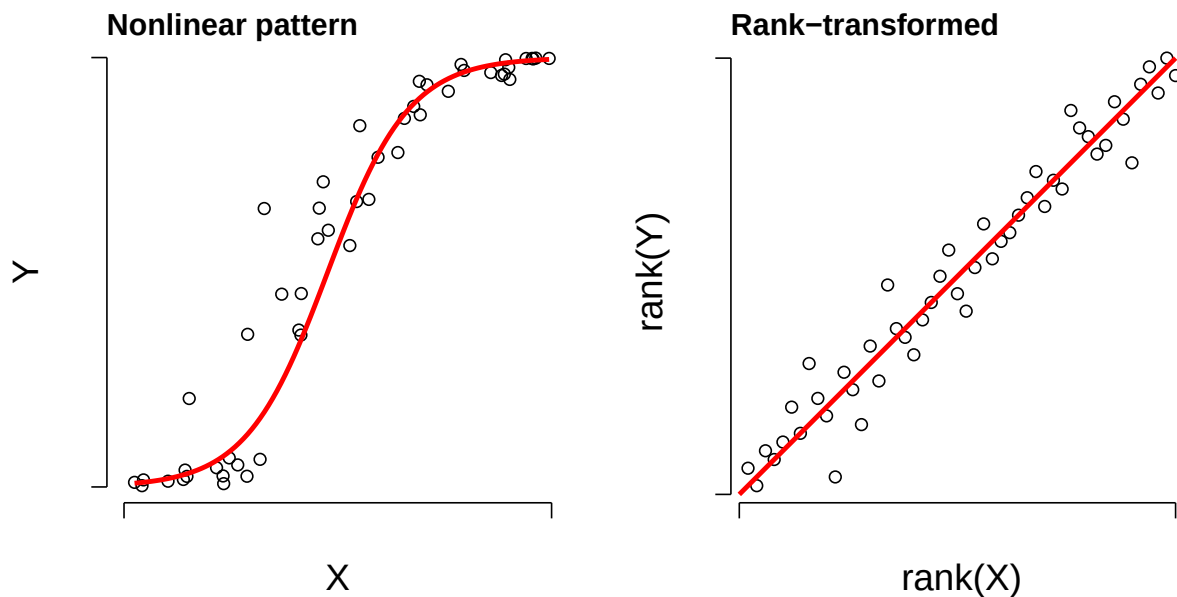
In this figure, scatterplots in the lower triangle correspond to the variables in the diagonal whom they intersect.

For example, the far left and far bottom scatterplot has Sepal.Length for its horizontal axis, and Petal.Width for its vertical axis. Go one panel to the right, and the horizontal axis is now Sepal.Width. The correlation coefficients in the upper triangle are identified the same way. For example, the large 0.96 on the far right, 3rd panel down, is the correlation coefficient between Petal.Length and Petal.Width.

6.4.2 Nonlinear correlation

Many processes and relationships in biology are not linear, and sometimes we want to calculate correlation coefficients for those relationships. Consider the figure below. The left panel clearly shows some relationship between X and Y , but it is not linear and using a linear correlation coefficient might miss something.

Figure 6.5: Left: illustration of a nonlinear correlation. Right: When values are rank-transformed, the non-linear trend becomes linear.



However, if you **rank-transform** the data, the relationship often turns into something more linear. A “rank-transform” assigns the smallest value value 0, the second smallest value to 1, and so on. Rank-transforming these data lets us calculate a nonparametric, nonlinear correlation. The nonparametric equivalent of Pearson’s r is **Spearman’s ρ** (“rho”). It can be obtained using code similar to that for linear correlation.

```
# just the correlation
cor(iris$Petal.Length, iris$Sepal.Width, method="spearman")
```

```
[1] -0.3096351
```

```
# significance test
cor.test(iris$Petal.Length, iris$Sepal.Width, method="spearman")
```

Spearman's rank correlation rho

```
data: iris$Petal.Length and iris$Sepal.Width
S = 736637, p-value = 0.0001154
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
-0.3096351
```

When you run a nonlinear correlation test, you will often get the warning `Warning in cor.test.default ... : Cannot compute exact p-value with ties`. All this means is that some of the rankings are tied. This warning can be safely ignored.

6.4.2.1 Nonlinear correlations between many pairs of variables

Every function that works with linear correlation will also work with nonlinear correlation. For example, we can get a matrix of Spearman correlations by using the `method` argument to `cor()`.

```
# Nonlinear correlation matrix
cor(iris[,1:4], method="spearman")
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Sepal.Length	1.0000000	-0.1667777	0.8818981	0.8342888
Sepal.Width	-0.1667777	1.0000000	-0.3096351	-0.2890317
Petal.Length	0.8818981	-0.3096351	1.0000000	0.9376668
Petal.Width	0.8342888	-0.2890317	0.9376668	1.0000000

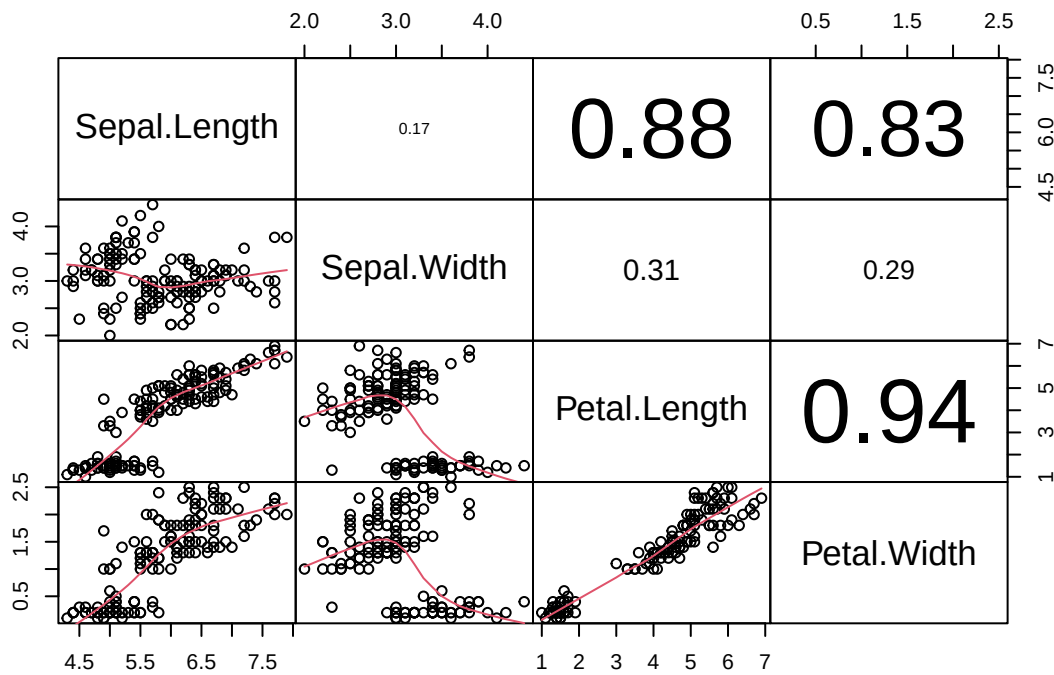
And we can modify the accessory functions of `pairs()` to use the Spearman correlation.

```
# from ?pairs
# just paste this in once to define the function
# try changing some arguments to change the behavior!
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  par(usr = c(0, 1, 0, 1))
  r <- abs(cor(x, y, method="spearman")) # <-- changed here
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex.cor * r)
}
```



```
# make the scatterplot matrix
pairs(iris[,1:4], lower.panel=panel.smooth, upper.panel=panel.cor, gap=0)
```

Figure 6.6: Scatterplot matrix showing possible nonlinear correlations between the numeric variables in the `iris` data frame.



Chapter 7

Summarizing biological data

In the last module we saw how single variables can be described or summarized using descriptive statistics (see Chapter 6). In this chapter we will extend that to summarizing values by groups. Indeed, comparing the mean values in different groups is one of the most important types of tests we do in biology (see Chapter 15).

7.1 Summarizing a data frame

You can get a quick numerical summary of a vector or a dataset with `summary()`. The examples below use the built-in dataset `iris`.

```
# summarize a data frame
summary(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100
1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300
Median :5.800	Median :3.000	Median :4.350	Median :1.300
Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199
3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800
Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500

Species

setosa :50
versicolor:50
virginica :50

```
# summarize a single variable
summary(iris$Petal.Length)
```

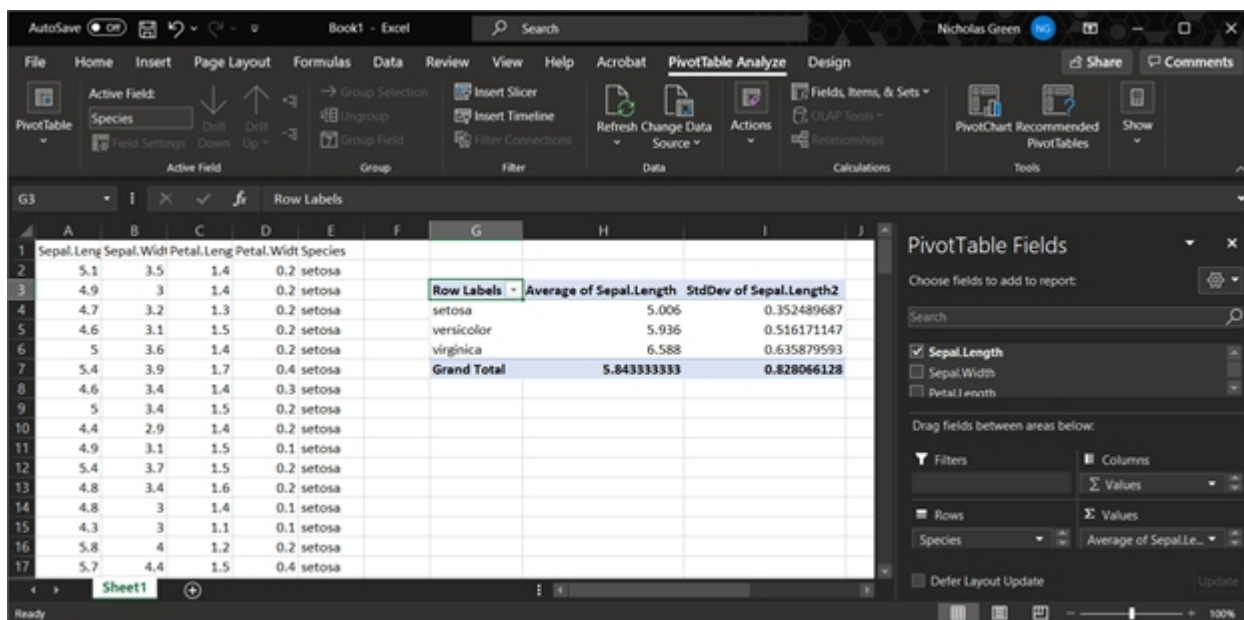
```
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.000   1.600   4.350   3.758   5.100   6.900
```

The function `summary()` has methods for many other types of R objects such as `glm`, `lm`, `aov`, and so on.

7.1.1 Summarizing by groups (AKA: pivot tables)

One of the most common data operations is **aggregation**, or **summarization by groups**. In Excel, this is done using **Pivot Tables**. In R, the functions `aggregate()` and `tapply()` are used for pivot table-like functionality. The tidyverse equivalent for `aggregate()` and `tapply()` is `dplyr::summarise()`. All of these can be much more powerful than an Excel pivot table because while Excel pivot tables are limited to a small set of built-in functions, R can aggregate by any function.

Figure 7.1: Example of a pivot table in Excel



7.1.1.1 aggregate()

The base R workhorse for summarizing data by groups is `aggregate()`. It is straightforward and powerful enough to cover almost all use cases. The most important inputs to `aggregate()` are the data to be summarized, the variable or variables that define the groups, and the function by which to summarize. The examples below shows how to calculate the mean petal length for each species in the `iris` dataset.

If the variable to be summarized and the grouping variables are in the same data frame (which is usually the case), you can use the formula interface to `aggregate()`; otherwise, use the `by=` interface.

```
# example using by=
aggregate(iris$Petal.Length,
          by=list(iris$Species),
          mean)
```

	Group.1	x
1	setosa	1.462
2	versicolor	4.260
3	virginica	5.552

```
# better method using formula
aggregate(Petal.Length~Species,
          data=iris,
          mean)
```

	Species	Petal.Length
1	setosa	1.462
2	versicolor	4.260
3	virginica	5.552

The formula interface is cleaner and easier than the `by=` interface (first example), and has the key advantage that it automatically passes variable names to the result. If you use `by=` you may want to rename the columns in the result.

In my own code I prefer to name my aggregation tables `agg`, with additional parts like `agg1`, `agg.fish`, and so on if needed. Function `aggregate()` produces a data frame with a column for each grouping variable (`Group.1`, `Group.2` and so on) and a single column `x` containing the summarized values. We can use this property to construct tables that summarize by more than one variable.

```
# spare copy
x <- iris

# add another grouping variable
x$color <- c("red", "white")

# by= example: notice how columns in result need to be renamed
agg <- aggregate(x$Petal.Length,
                 by=list(x$Species, x$color),
                 mean)

agg
```

	Group.1	Group.2	x
1	setosa	red	1.456
2	versicolor	red	4.308
3	virginica	red	5.564
4	setosa	white	1.468
5	versicolor	white	4.212
6	virginica	white	5.540

```
# change column names
names(agg) <- c("spp", "color", "mean")
agg
```

	spp	color	mean
1	setosa	red	1.456
2	versicolor	red	4.308
3	virginica	red	5.564
4	setosa	white	1.468
5	versicolor	white	4.212
6	virginica	white	5.540

```
# formula example: much easier
agg <- aggregate(Petal.Length~Species+color, data=x, mean)
agg
```

	Species	color	Petal.Length
1	setosa	red	1.456
2	versicolor	red	4.308
3	virginica	red	5.564
4	setosa	white	1.468
5	versicolor	white	4.212
6	virginica	white	5.540

Notice that in `agg` the values are sorted by the grouping variables from right to left: in this example, by `color`, then `Species`. This is the reverse of the order in the original command. The result of `aggregate()` is a data frame, so data within it can be rearranged using `order()`.

Often we want to summarize a variable by multiple functions—for example, to get a table with the mean and SD in each group. My preferred way to do this is to use several `aggregate()` commands, and then combine the results. The examples below shows two ways to construct a table with the mean and SD of a variable.

```
# define a formula and save some typing:
f1 <- formula(Petal.Length~Species)

# method 1: single table using formula
agg <- aggregate(f1, data=x, mean)
agg$sd <- aggregate(f1, data=x, sd)$Petal.Length
names(agg)[which(names(agg) == "Petal.Length")] <- "mn"

agg
```

	Species	mn	sd
1	setosa	1.462	0.1736640
2	versicolor	4.260	0.4699110
3	virginica	5.552	0.5518947

The `$Petal.Length` on the end of `aggregate()` is important because we want only column `Petal.Length` of the output. Without `$Petal.Length`, the result will be messy when we try to add it to `agg`. If you ever want to do this with the `by=` method, the column of the output containing the summarized values will be named `x`.

```
# method 1a: single table, using by=
agg <- aggregate(iris$Petal.Length, by=list(iris$Species), mean)
## note use of $x:
agg$sd <- aggregate(iris$Petal.Length, by=list(iris$Species), sd)$x

# change names in result:
names(agg)[which(names(agg) == "x")] <- "mn"
names(agg)[which(names(agg) == "Group.1")] <- "species"
agg
```

	species	mn	sd
1	setosa	1.462	0.1736640
2	versicolor	4.260	0.4699110
3	virginica	5.552	0.5518947

The second approach is to make several `aggregate()` tables, and combine them into a single result.

```
# method 2: multiple tables
# (note: there are lots of ways to do this)
agg.mn <- aggregate(f1, data=x, mean)
agg.sd <- aggregate(f1, data=x, sd)
```

```
agg <- data.frame(agg.mn, sd=agg.sd$Petal.Length)
names(agg)[2] <- "mn"
agg
```

	Species	mn	sd
1	setosa	1.462	0.1736640
2	versicolor	4.260	0.4699110
3	virginica	5.552	0.5518947

7.1.1.2 tapply()

The base function `tapply()`, like `aggregate()`, summarizes a numeric variable by a grouping variable. In fact, `aggregate()` uses `tapply()` internally. The difference is that while `aggregate()` returns a data frame, `tapply()` returns an array with number of dimensions equal to the number of grouping variables. The inputs are a vector to be summarized, a grouping variable or set of variables, and the function by which to summarize.

```
# spare copy
x <- iris
# add some extra grouping variables
x$color <- c("red", "white")
x$treat <- c("a", "b", "c")

# tapply by 1 variable
tapply(x$Petal.Length, x$Species, mean)
##      setosa versicolor  virginica
##      1.462      4.260      5.552
```

```
# tapply by 2 variables
tapply(x$Petal.Length, x[,c("Species", "color")], mean)
```

		color	
Species		red	white
setosa		1.456	1.468
versicolor		4.308	4.212
virginica		5.564	5.540

```
# tapply by 3 variables
tapply(x$Petal.Length, x[,c("Species", "color", "treat")], mean)
```

```
# vary the arrangement of groups:
tapply(x$Petal.Length, x[,c("color", "Species")], mean)
```

	Species		
color	setosa	versicolor	virginica
red	1.456	4.308	5.564
white	1.468	4.212	5.540

Personally, I prefer `aggregate()` to `tapply()` because it is usually more convenient to have the results in a data frame. You also need to be more careful with `tapply()` because it will interpret factors as factors and thus return values for missing levels. `aggregate()`, on the other hand, will drop the missing levels.

```
# aggregate drops the missing levels.
aggregate(Petal.Length~Species, data=iris[1:100,], mean)
```

	Species	Petal.Length
1	setosa	1.462
2	versicolor	4.260

```
# tapply includes the missing levels
tapply(iris$Petal.Length[1:100], iris$Species[1:100], mean)
```

	setosa	versicolor	virginica
	1.462	4.260	NA

Which behavior you want depends on what you are trying to do. Just be aware that `aggregate()` and `tapply()` might return different numbers of values depending on what you have already done to your data frame.

7.1.2 Summarizing by arbitrary functions

If you've ever used Excel Pivot Tables, you will have noticed that data can only be summarized by a limited set of functions. On the other hand, `aggregate()` (and `tapply()`, but we'll focus on `aggregate()`) can be used with pretty much any function that takes in a vector and returns a scalar. You can even write your own functions! Below are some examples. Notice in the first example that arguments to the summarizing function follow its name.

```
# first quartile (25th percentile)
aggregate(Petal.Length~Species, data=iris, quantile, 0.25)
```


	Species	Petal.Length
1	setosa	1.4
2	versicolor	4.0
3	virginica	5.1

```
# number of values
aggregate(Petal.Length~Species, data=iris, length)
```

	Species	Petal.Length
1	setosa	50
2	versicolor	50
3	virginica	50

The following examples illustrate **anonymous functions**: functions defined within another function and never defined in the R workspace. The `x` in `function(x)` is taken to be the values within each group defined by the variable(s) on the right-hand side of `~` or in `by=`. Notice that the end of such a command will need a lot of `)`, `}`, and sometimes `]`. Using an editor with syntax highlighting is very helpful!

```
# number of missing values
aggregate(Petal.Length~Species,
          data=iris,
          function(x){length(which(is.na(x))))})
```

	Species	Petal.Length
1	setosa	0
2	versicolor	0
3	virginica	0

```
# number of non-missing values
aggregate(Petal.Length~Species,
          data=iris,
          function(x){length(which(!is.na(x))))})
```

	Species	Petal.Length
1	setosa	50
2	versicolor	50
3	virginica	50

```
# number of values >= 2
aggregate(Petal.Length~Species,
          data=iris,
          function(x){length(which(x >= 2))})
```

	Species	Petal.Length
1	setosa	0
2	versicolor	50
3	virginica	50

```
# number of UNIQUE values >= 2
aggregate(Petal.Length~Species,
          data=iris,
          function(x){length(unique(x[which(x >= 2)]))})
```

	Species	Petal.Length
1	setosa	0
2	versicolor	19
3	virginica	20

Being able to use almost any function, and to define your own functions, makes `aggregate()` an immensely powerful tool to have in your R toolbox.

7.2 The `apply()` family

The `apply()` family is a group of functions that operate over the **dimensions** or **margins** an object. You can use the `apply()` family to summarize across rows or columns, or across elements of a list, and so on. The function used most often is `apply()`, which works on data frames, matrices and arrays. There are also `lapply()`, which works on lists and returns a list; `sapply()`, which operates on lists and returns the simplest object possible; and a few others.

We already saw `tapply()`, which works on subsets of a data frame or array.

7.2.1 `apply()` for arrays and data frames

Function `apply()` works on data frames or arrays with ≥ 2 dimensions. Its arguments are the object to be operated on, the margin (or dimension) on which to operate, and the operation (or function). The first margin is rows, the second is columns, and so on. The result of `apply()` will *always* have one fewer dimension than the input.

Some typical `apply()` commands are demonstrated below. The first calculates the sum of values in each row (margin 1) of its input `x`.

```
# test dataset with only numeric values
x <- iris[,1:4]
```

```
# margin 1 = rows --> row sums
apply(x, 1, sum)
```

```
[1] 10.2  9.5  9.4  9.4 10.2 11.4  9.7 10.1  8.9  9.6 10.8 10.0  9.3  8.5 11.2
[16] 12.0 11.0 10.3 11.5 10.7 10.7 10.7  9.4 10.6 10.3  9.8 10.4 10.4 10.2  9.7
[31]  9.7 10.7 10.9 11.3  9.7  9.6 10.5 10.0  8.9 10.2 10.1  8.4  9.1 10.7 11.2
[46]  9.5 10.7  9.4 10.7  9.9 16.3 15.6 16.4 13.1 15.4 14.3 15.9 11.6 15.4 13.2
[61] 11.5 14.6 13.2 15.1 13.4 15.6 14.6 13.6 14.4 13.1 15.7 14.2 15.2 14.8 14.
[76] 15.4 15.8 16.4 14.9 12.8 12.8 12.6 13.6 15.4 14.4 15.5 16.0 14.3 14.0 13.
[91] 13.7 15.1 13.6 11.6 13.8 14.1 14.1 14.7 11.7 13.9 18.1 15.5 18.1 16.6 17.
[106] 19.3 13.6 18.3 16.8 19.4 16.8 16.3 17.4 15.2 16.1 17.2 16.8 20.4 19.5 14.
[121] 18.1 15.3 19.2 15.7 17.8 18.2 15.6 15.8 16.9 17.6 18.2 20.1 17.0 15.7 15.
[136] 19.1 17.7 16.8 15.6 17.5 17.8 17.4 15.5 18.2 18.2 17.2 15.7 16.7 17.3 15.
```

This command calculates the sum of values in each column (margin 2).

```
# margin 2 = columns --> column sums
apply(x, 2, sum) # 2 = columns
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           876.5           458.6           563.7           179.9
```

The results are vectors whose lengths are the same as the corresponding dimension of the input (number of rows or columns).

The first command above is equivalent to:

```
y <- numeric(nrow(x))
for(i in 1:nrow(x)){y[i] <- sum(x[i,])}

# verify that results are the same:
all(apply(x, 1, sum) == y)
```

```
[1] TRUE
```

Below are some examples of calculations you can do with the `apply()` function. Notice that when the function to be applied takes its own arguments, those arguments are supplied following the function name, in the same order as they would be supplied to the function if used outside of `apply()`.

```
# column minima
apply(x, 2, min)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
4.3	2.0	1.0	0.1

```
# 75th percentile of each column
apply(x, 2, quantile, 0.75)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
6.4	3.3	5.1	1.8

```
# interquartile range of each row
apply(x, 1, IQR)
```

```
[1] 2.800 2.375 2.550 2.300 2.850 2.900 2.575 2.625 2.175 2.400 2.950 2.500
[13] 2.375 2.475 3.500 3.500 3.200 2.775 2.925 2.925 2.575 2.825 3.050 2.350
[25] 2.275 2.250 2.500 2.750 2.750 2.325 2.275 2.675 3.225 3.425 2.375 2.700
[37] 2.975 2.850 2.325 2.650 2.825 1.800 2.475 2.525 2.600 2.325 2.875 2.450
[49] 2.925 2.625 2.525 2.200 2.700 2.325 2.600 2.375 2.225 1.650 2.600 1.850
[61] 2.125 2.000 2.600 2.525 1.600 2.300 2.150 2.250 2.900 2.175 2.225 2.100
[73] 3.000 2.650 2.325 2.350 2.850 2.750 2.325 1.850 2.150 2.100 2.050 2.900
[85] 2.100 1.925 2.500 2.825 1.900 2.175 2.425 2.375 2.200 1.750 2.200 2.025
[97] 2.075 2.275 1.375 2.075 2.975 2.775 3.425 3.150 3.175 4.075 2.300 3.925
[109] 3.700 3.050 2.550 3.075 3.050 2.800 2.575 2.600 3.050 3.550 4.575 3.225
[121] 3.025 2.475 4.350 2.775 2.950 3.450 2.600 2.500 3.175 3.500 3.850 3.425
[133] 3.150 2.925 3.425 3.675 2.625 2.950 2.400 2.925 2.950 2.650 2.775 3.150
[145] 2.850 2.750 2.975 2.775 2.475 2.600
```

Arrays with > 2 dimensions can have functions applied across multiple dimensions:

```
# object with 3 rows, 2 columns, and 3 "layers":
x <- array(1:18, dim=c(3,2,3))
x
```

```
, , 1
```

	[,1]	[,2]
[1,]	1	4
[2,]	2	5
[3,]	3	6

```
, , 2
```

```

      [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12

```

```

, , 3

```

```

      [,1] [,2]
[1,]   13   16
[2,]   14   17
[3,]   15   18

```

```

# how many dimensions does the result have?
apply(x, c(1,2), sum)

```

```

      [,1] [,2]
[1,]   21   30
[2,]   24   33
[3,]   27   36

```

You can also write custom functions and apply them to rows or columns. Below are some useful functions to apply. These functions can be defined inside `apply()` or outside `apply()`. Each of the functions below takes an input called `x`, but this is not as the `x` that is the input to `apply()`. The `x` in `function(x)` is used inside the braces `{ }` as the name for the input passed from `apply()` to the inner function.

```

x <- iris[,1:4]

# count values > 2
apply(x, 2, function(x){length(which(x > 2))})

```

```

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           150           149           100           23

```

```

# count missing (NA) values
apply(x, 2, function(x){length(which(is.na(x)))})

```

```

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           0           0           0           0

```

```
# count non-missing values
count.nonmissing <- function(x){length(which(!is.na(x)))}
apply(x, 2, count.nonmissing)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           150           150           150           150
```

7.2.2 lapply() and sapply() for lists and other objects

The other members of the `apply()` family that are most often used are `lapply()` and `sapply()`.

- `lapply()`, operates on lists and returns a *list*. Short for “list apply”, and usually pronounced “el apply” (or maybe “lap-lie”).
- `sapply()` operates on many object types (including lists) and returns the simplest object possible. Short for “simplify apply”, and usually pronounced “ess apply” or “sap-lie”. Usually the goal with `sapply()` is to get a vector or matrix out of a list.

The code block below makes a list called `mylist` and illustrates the use of `lapply()` and `sapply()`.

```
# make mylist, a list with 5 elements
mylist <- vector("list", 5)

# make each element of mylist a vector
# of 20 random numbers from standard normal dist.
for(i in 1:length(mylist)){
  x <- rnorm(20)
  mylist[[i]] <- t.test(x)
}#i

# get estimated mean of each element of mylist, in a list
lapply(mylist, function(x){x$estimate})
```

```
[[1]]
mean of x
0.04580683
```

```
[[2]]
mean of x
0.000698894
```

```
[[3]]
mean of x
-0.1796158
```

```
[[4]]
mean of x
-0.3097512
```

```
[[5]]
mean of x
-0.1202498
```

```
# get estimated mean of each element, simplified to vector
sapply(mylist, function(x){x$estimate})
```

```
mean of x    mean of x    mean of x    mean of x    mean of x
0.045806829  0.000698894 -0.179615831 -0.309751173 -0.120249809
```

The function `lapply()` can sometimes be combined with `do.call()` to produce similar outputs as `sapply()`:

```
do.call(c, lapply(mylist, function(x){x$estimate}))
```

```
mean of x    mean of x    mean of x    mean of x    mean of x
0.045806829  0.000698894 -0.179615831 -0.309751173 -0.120249809
```

Clever use of `lapply()` and `sapply()` can save you a lot of time and typing. Interestingly, using these functions can make base R code feel a lot like `tidyverse` code, in the sense that commands are issued as functions with much less focus on manipulating objects.

7.3 Tabulation (frequency tables)

Aggregation is summarizing numeric variables by group. The equivalent for counting up occurrences of values in non-numeric variables is **tabulation**. A lot of tabulation is just counting. R can do a lot of counting for you.

7.3.1 Frequency tables with ‘table()’

The `table()` function tallies the occurrences of each unique value in a vector. This is called a **frequency table**. Notice that `table()` only returns a result for the values that actually occur in the input.

```
aa <- rpois(50,10) # Random Poisson distribution
table(aa)
```

```
aa
 4  6  7  8  9 10 11 12 13 14 15 16 17 21
 3  5  3  5  6  4  3  7  7  1  1  2  2  1
```

If you need a count that includes 0s for missing values, there are a few ways to do this. My preferred solution is to use the custom function below. This function counts how many times each value in `values` occurs in `input`.

```
table.all <- function(input, values){
  res <- sapply(values,
                function(x){length(which(input == x))})
  names(res) <- as.character(values)
  return(res)
}
```

```
# use with random poisson values:
x1 <- rpois(20, 10)
table.all(x1, 0:30)
```

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 0  0  0  1  1  0  1  1  4  2  4  1  1  1  1  1  1  0  0  0  0  0  0  0  0
26 27 28 29 30
 0  0  0  0  0
```

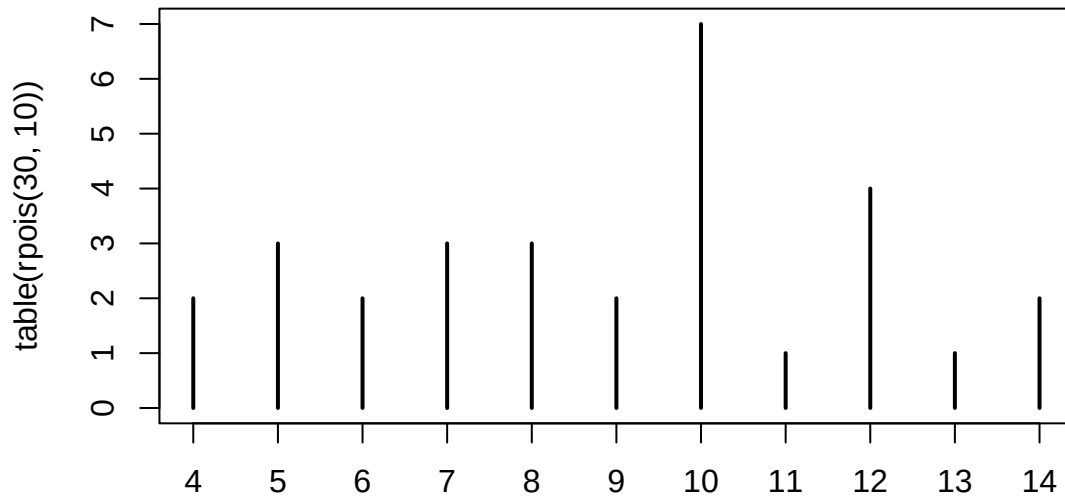
```
# compare to:
table(x1)
```

```
x1
 3  4  6  7  8  9 10 11 12 13 14 15 16
 1  1  1  1  4  2  4  1  1  1  1  1  1
```

If you plot a `table()` result, you get a sort of bar graph. This is similar to what you get with a command like `plot(..., type = "h")`.¹

¹The ellipsis `...` is a stand in for other arguments to the function that are irrelevant to the point being made; there are some circumstances where an ellipsis is necessary to control how variables are passed from one function to another.


```
plot(table(rpois(30, 10)))
```



7.3.2 Contingency tables with `fTable()`

Contingency tables summarize how many observations occur in combinations of categories. They are calculated with `fTable()`. This function counts the number of values in each combination of different factors. The example below uses the built-in dataset `iris` modified to have some additional factors.

```
set.seed(123)
x <- iris
x$color <- c("purple", "white")
x$bloom <- sample(c("early", "late"), nrow(x), replace=TRUE)
color.ftab <- ftable(x$color~x$bloom)
color.ftab
```

```
      x$color purple white
x$bloom
early      39      37
late       36      38
```

Contingency tables produced by `fTable()` can be used directly for chi-squared (χ^2) tests:

```
chisq.test(color.ftab)
```

Pearson's Chi-squared test with Yates' continuity correction

```
data: color.ftab
```

```
X-squared = 0.026671, df = 1, p-value = 0.8703
```

The function `ftable()` also has a formula interface, much like many statistical functions (e.g., `lm()`) or `aggregate()`. This is useful if you want more complicated tables or more combinations of categories. Notice that the order of variables affects the structure of the resulting table. The 6 commands below produce equivalent tables, but the results are organized differently.

```
ftable(Species~color+bloom, data=x)
```

		Species setosa	versicolor	virginica
color	bloom			
purple	early	17	13	9
	late	8	12	16
white	early	13	14	10
	late	12	11	15

```
ftable(Species~color+bloom, data=x)
```

		Species setosa	versicolor	virginica
color	bloom			
purple	early	17	13	9
	late	8	12	16
white	early	13	14	10
	late	12	11	15

```
ftable(Species~bloom+color, data=x)
```

		Species setosa	versicolor	virginica
bloom	color			
early	purple	17	13	9
	white	13	14	10
late	purple	8	12	16
	white	12	11	15

```
ftable(color~Species+bloom, data=x)
```

		color	purple	white
Species	bloom			
setosa	early		17	13
	late		8	12
versicolor	early		13	14
	late		12	11
virginica	early		9	10
	late		16	15

```
ftable(color~bloom+Species, data=x)
```

		color	purple	white
bloom	Species			
early	setosa		17	13
	versicolor		13	14
	virginica		9	10
late	setosa		8	12
	versicolor		12	11
	virginica		16	15

```
ftable(bloom~color+Species, data=x)
```

		bloom	early	late
color	Species			
purple	setosa		17	8
	versicolor		13	12
	virginica		9	16
white	setosa		13	12
	versicolor		14	11
	virginica		10	15

```
ftable(bloom~Species+color, data=x)
```

		bloom	early	late
Species	color			
setosa	purple		17	8
	white		13	12
versicolor	purple		13	12
	white		14	11

virginica	purple	9	16
	white	10	15

Variables on the left side of the ~ will be on the top of the table, and variables on the right side of the ~ will be on the left side of the table. On each side, variables are ordered in reverse order from the table matrix outwards. I can never remember how this works, so usually just have to tinker with the code until I get the table I want.

Compare the results of these commands:

```
ftable(Species~color+bloom, data=x)
```

		Species setosa	versicolor	virginica
color	bloom			
purple	early	17	13	9
	late	8	12	16
white	early	13	14	10
	late	12	11	15

```
ftable(Species~bloom+color, data=x)
```

		Species setosa	versicolor	virginica
bloom	color			
early	purple	17	13	9
	white	13	14	10
late	purple	8	12	16
	white	12	11	15

The output of `ftable()` is of class `ftable`, which *looks* like a matrix or data frame but does not function like one. Sometimes it is necessary to convert an `ftable()` output to different class in order to pull values from it. This can be done with base R conversion functions. Notice that the data frame form of `handed.table` resembles the “long” data format we discussed in Section 5.5.3.2.

```
spp.color.ftab <- ftable(x$Species~x$color)
as.matrix(spp.color.ftab)
```

		x\$Species		
x\$color		setosa	versicolor	virginica
purple		25	25	25
white		25	25	25

```
spp.color.df <- as.data.frame(spp.color.ftab)
names(spp.color.df) <- c("color", "species", "freq")
spp.color.df
```

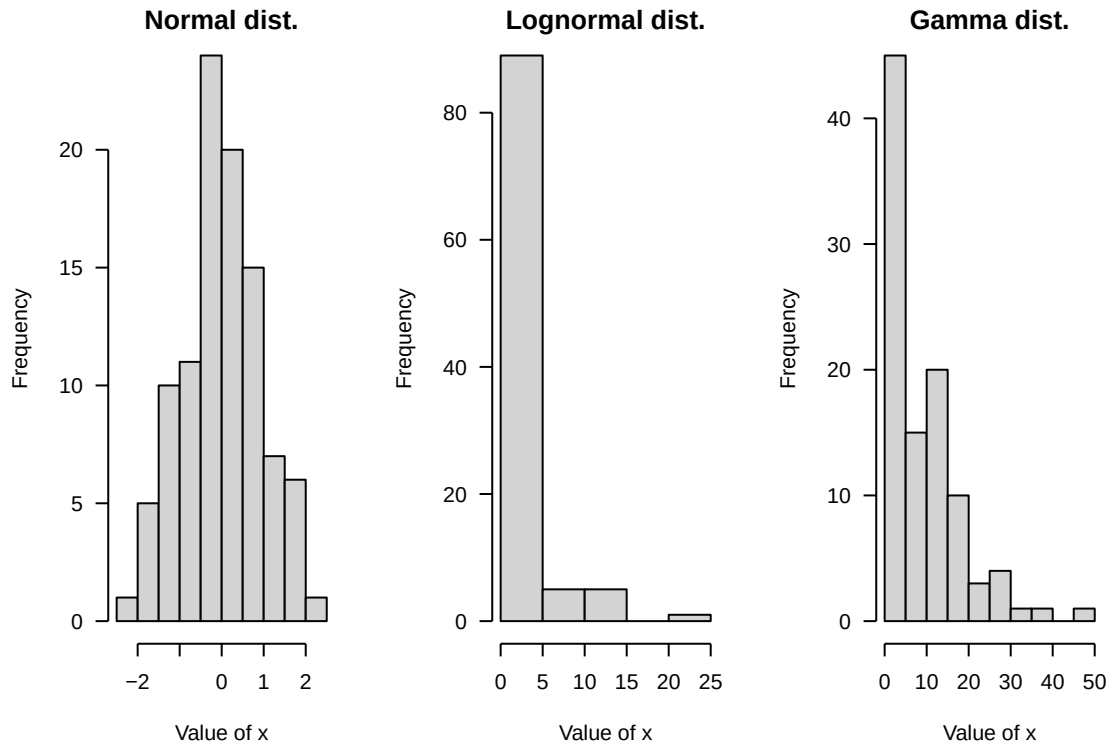
	color	species	freq
1	purple	setosa	25
2	white	setosa	25
3	purple	versicolor	25
4	white	versicolor	25
5	purple	virginica	25
6	white	virginica	25

Chapter 8

Visualizing biological data

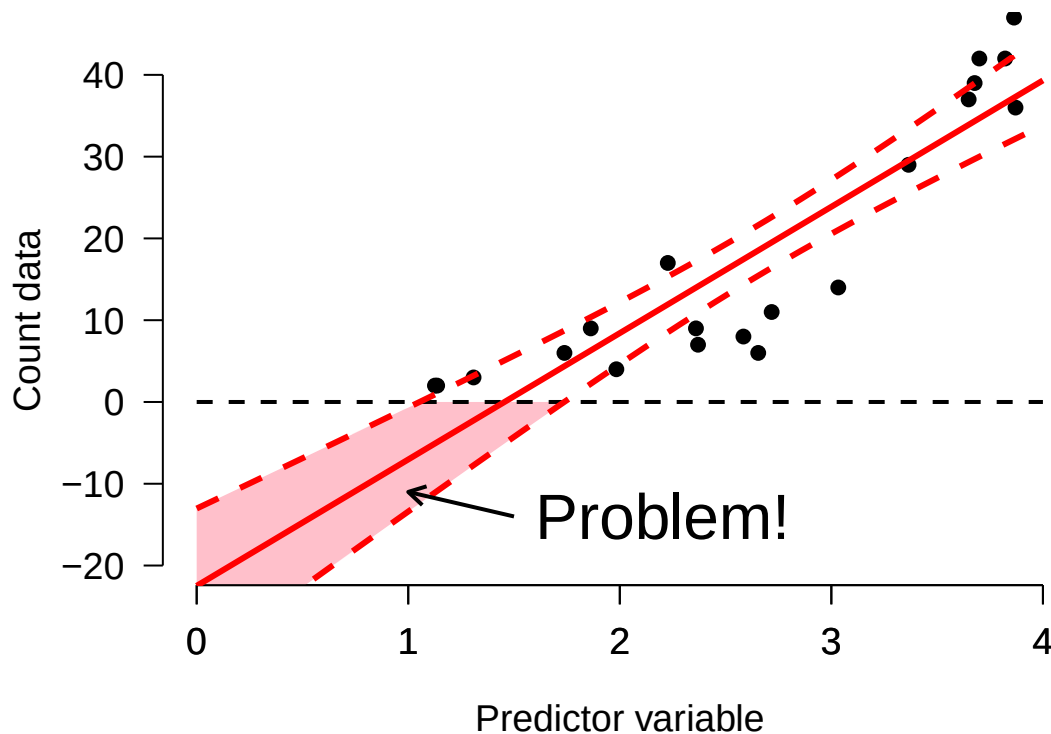
8.1 Visualizing single variables

In statistics, we think of values within a variable coming from a probability distribution. **Probability distributions**, or **distributions** for short, are mathematical functions that describe the way that values vary randomly. Distributions are a key idea in statistics and data analysis and we'll cover them in detail in the **next module**. We usually consider data to be essentially random, but with the values forming predictable patterns over many observations. The nature of those patterns is what probability distributions attempt to model. A probability distribution does not tell us the value of any particular observation, but it does let us estimate the likelihood of observing any particular value. The figure below demonstrates some common probability distributions: the heights of the bars reflect the likelihood of the values on the x-axes occurring under those distributions.



Successful data analysis requires paying attention to and thinking carefully about the way that your data are distributed. Different kinds of biological processes, like counts, waiting times, measurements, etc., have randomness that is described by different probability distributions. The assumption that values follow certain distributions is baked into most statistical methods. Trying to use a method that assumes an inappropriate distribution will probably lead to invalid results.

For example, if you try to analyze count data as if they came from a continuous, unbounded distribution, your statistical model could predict nonsensical outcomes such as negative or non-integer counts. The figure below shows this: the red shaded area are predictions of negative counts!



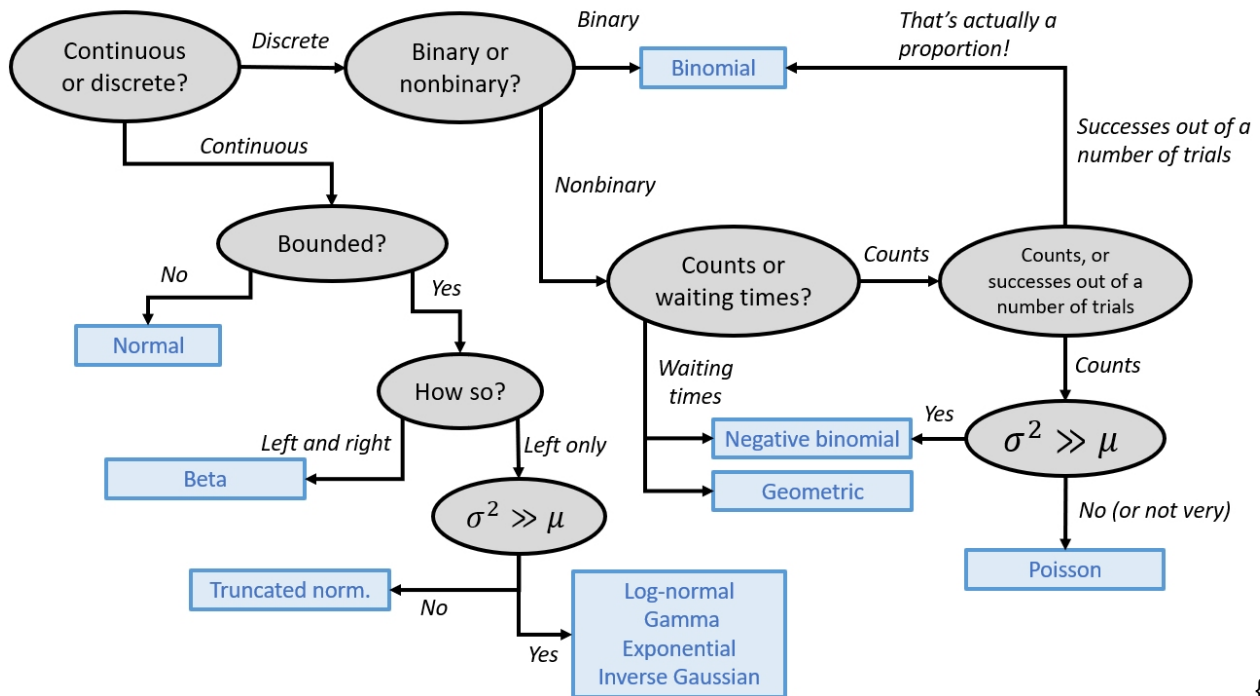
Sometimes it is not always obvious what distribution your data should follow. Some cases are relatively straight forward: counts should usually follow a Poisson or negative binomial distribution. Other cases are less straightforward: should gene expression data be considered normally, log-normally, or some other-ly distributed? Usually there is a clear *a priori* answer to this question that can be obtained by thinking about what kind of process gave rise to the numbers.

A related question to “what distribution does my data follow?” is “do my data follow this distribution?” Even if you have some expectation of how your data should be distributed, that is no guarantee that your study system cooperated. You should always check to see if your data follow the distribution you assumed, and the distribution assumed by your statistical test.

Deciding what distribution your data follow is always somewhat subjective, and real datasets often contain some departure from expectations. In this section we will explore some graphical techniques and heuristics for determining how data are distributed.

Knowing about **data distributions** is a one thing, but is quite another. Like much of the data analysis workflow, picking a distribution can be as much an art as it is a science. This is because distributions of real data are often messy, or do not conform exactly to any one distribution, or may conform partly to several! Sometimes the answer to “what distribution should I use to analyze this data?” is not clear-cut.

The figure below gives a (very) rough guide to starting to identify a response distribution. Note that this is only a guide for where to look first, and not a definitive guide to selecting a response distribution. Even if your choices lead you to, say, the normal distribution, you must still verify that your data conform (at least roughly) to that distribution. Note also that this diagram does not include every distribution out there, but rather a select set of commonly encountered distributions in biology.



{#fig-

alt="Decision tree for selecting a probability distribution based on the response variable. First determine whether the response is continuous or discrete. Continuous variables that are unbounded are modeled with a normal distribution. Bounded continuous variables are modeled with a beta distribution if bounded on both sides. If bounded only on the left, use a truncated normal distribution when the variance is not much larger than the mean, or a log-normal, gamma, exponential, or inverse Gaussian distribution when the variance greatly exceeds the mean. For discrete variables, first determine whether the response is binary. Binary responses use a binomial distribution. Nonbinary discrete responses are divided into counts and waiting times. Waiting times use a geometric distribution or, when overdispersed, a negative binomial distribution. Count data are next classified as simple counts or counts out of a fixed denominator (proportions). Counts out of a denominator are treated as binomial data. For simple counts, use a Poisson distribution when the variance is approximately equal to the mean and a negative binomial distribution when the variance greatly exceeds the mean."}

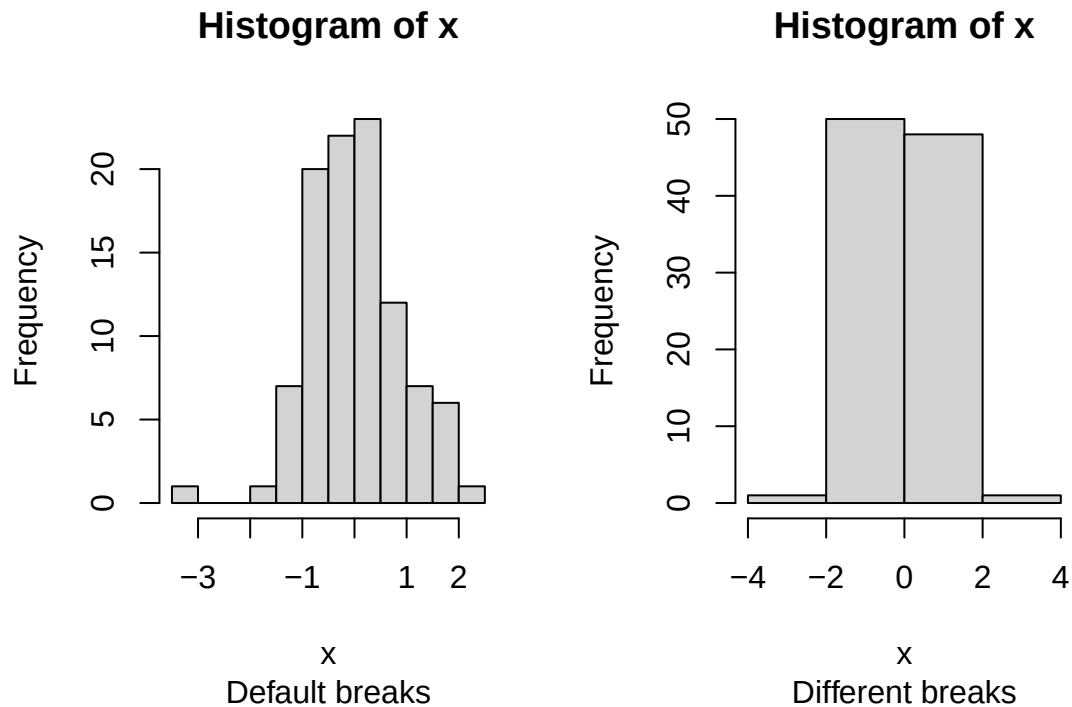
8.1.1 Histograms

Histograms show how values of a distribution are spread across different intervals. These intervals are sometimes called **cells** or **bins**. A good histogram will show, approximately, the shape of a **probability density function (PDF)** of a **continuous distribution** or the **probability mass function (PMF)** of a **discrete distribution**. A histogram with bin widths of 0 should look like a **kernel density plot**.

The base function `hist()` will automatically select intervals that look nice, but you can specify the intervals with argument `breaks`.

```
x <- rnorm(100)

# 1 row, 2 column plot layout
par(mfrow=c(1,2))
hist(x, sub="Default breaks")
hist(x, breaks=seq(-4, 4, by=2), sub="Different breaks")
```



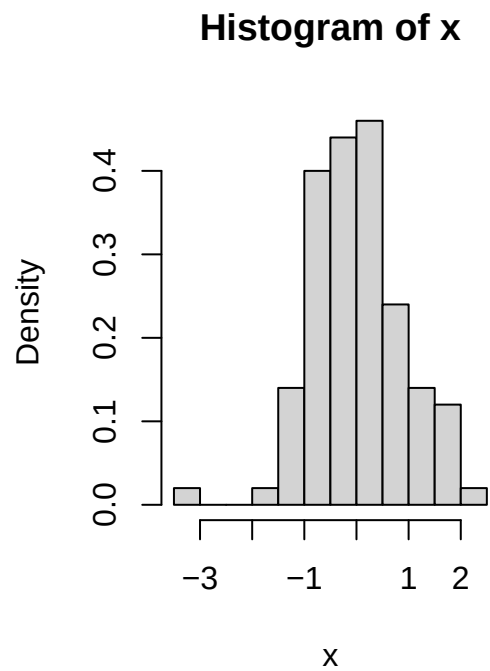
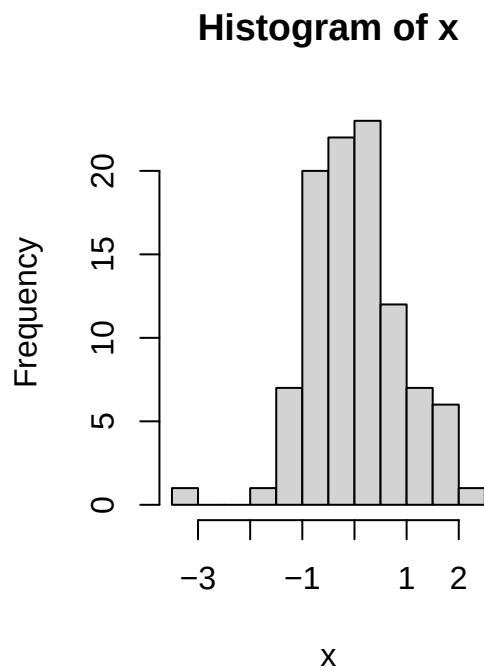
```
# reset plot layout to 1 x 1
par(mfrow=c(1,1))
```

The *area* of each bar is also proportional to the number of values in each interval.

R histograms can be presented as counts (`freq=TRUE`, the default), or as probability density (`freq=FALSE`). We'll talk more about probability density later in the [next section](#).

Compare these results:

```
par(mfrow=c(1,2))
hist(x)
hist(x, freq=FALSE)
```



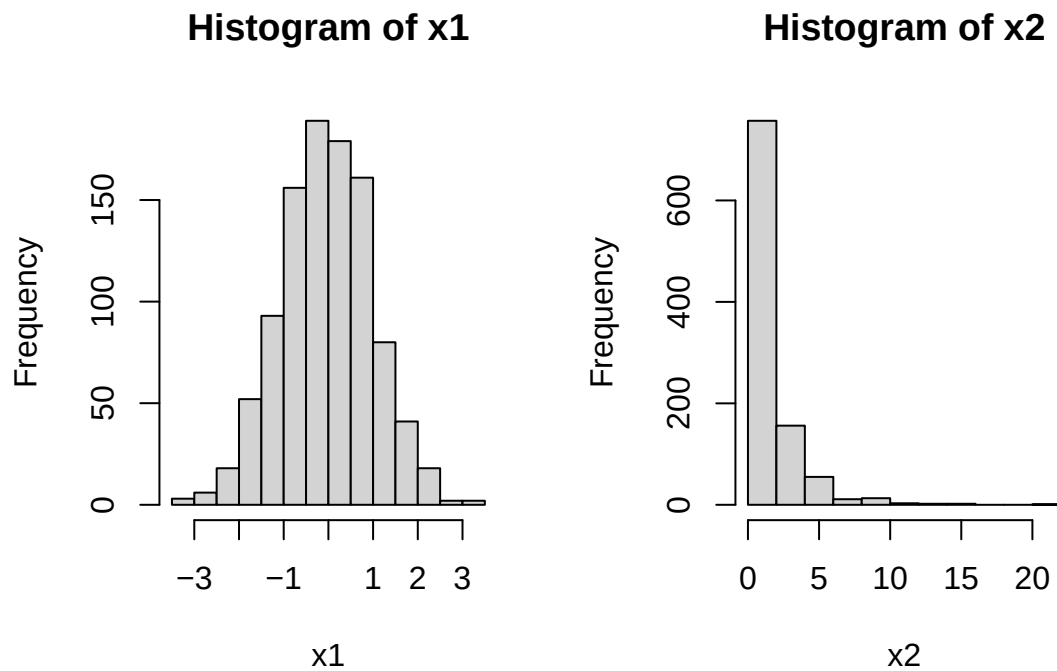
```
# reset
par(mfrow=c(1,1))
```

Because the second histogram shows probability densities, the areas under the bars sum to 1 (this is part of the definition of a PDF) (see Section 9.3).

A histogram can provide a visual first clue as to the shape of a distribution's PDF or PMF. For example, the histograms of `x1` and `x2` below suggest very different distributions:

```
x1 <- rnorm(1000)
x2 <- rlnorm(1000)

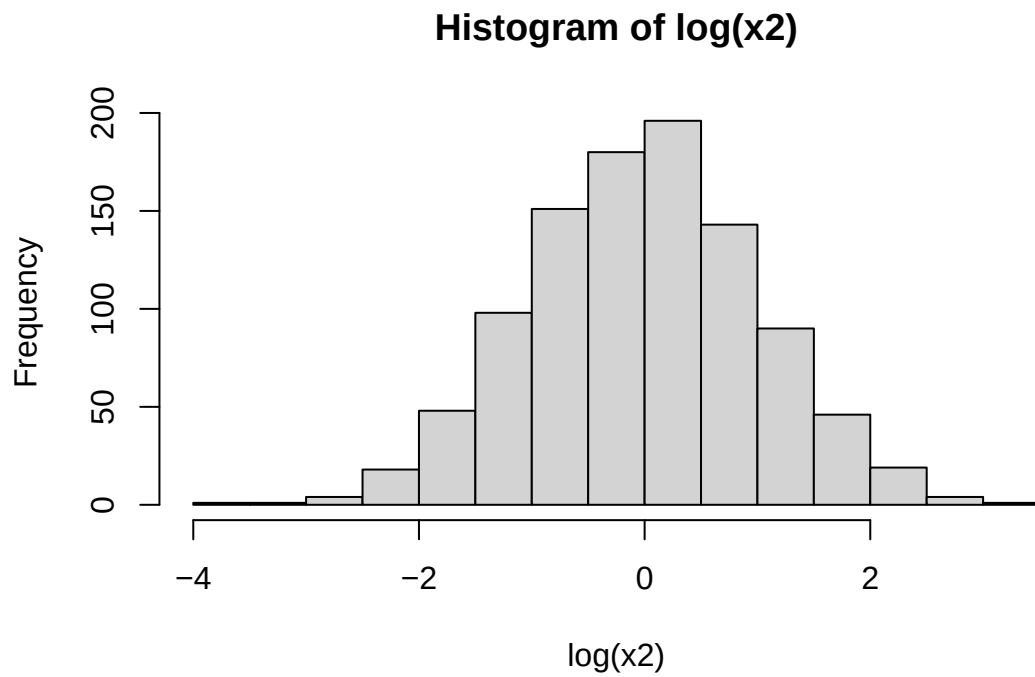
par(mfrow=c(1,2))
hist(x1)
hist(x2)
```



```
# reset
par(mfrow=c(1,1))
```

Notice that `x1` appears roughly normal. It is symmetric, has a bell-shape, and is concentrated near its center. Contrast this with `x2`. Distribution `x2` is strongly right-skewed (i.e., lots of small values with a long positive tail), with most values near 0. It also has no negative values (`range(x2)`). Because of these properties we might suspect that `x2` is really a log-normal distribution. We can check this by making a histogram of the log of `x2`:

```
hist(log(x2))
```

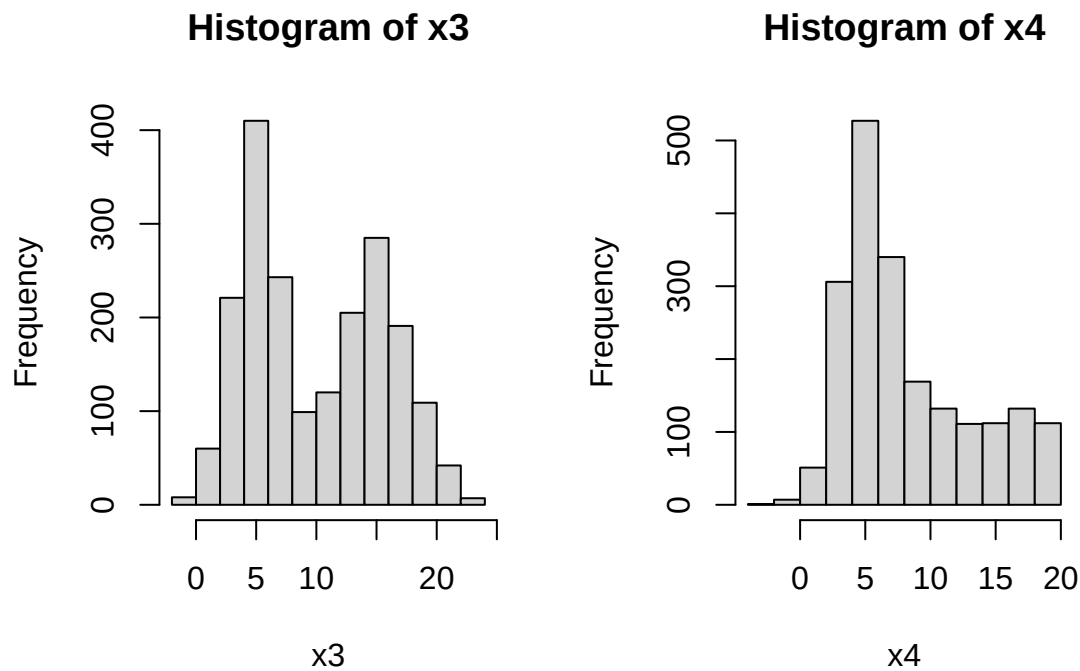


Other distributions might be tricky. Consider the histograms below:

```
x1 <- rnorm(1000, 5, 2)
x2 <- rnorm(1000, 15, 3)
x3 <- c(x1, x2)

x1 <- rnorm(1000, 5, 2)
x2 <- runif(1000, 3, 20)
x4 <- c(x1, x2)

par(mfrow=c(1,2))
hist(x3)
hist(x4)
```



```
# reset
par(mfrow=c(1,1))
```

What a mess!

- Histogram x_3 shows what is commonly referred to as a **bimodal** distribution. This is a distribution with two **modes**, or most common values, with a gap between them. Bimodal distributions often arise from a **mixture** of two distributions (in this case, two normals). Or, they can indicate that something in the data generating process leads to diverging outcomes. For example, in many college courses the grade distribution can be bimodal, with most students making either a B or D.
- Histogram x_4 shows a distribution with some right skew, but not the long, tapering tail characteristic of the lognormal distribution (see x_2 , above). The fact that the mean is near 5 and not 0 is another point against a lognormal. But the clincher is that there are negative values, which a lognormal distribution cannot take (think about why this is).

Think for a few minutes about what distribution x_4 might be, then click the footnote to find out¹.

8.1.2 Barplots

Barplots are usually used to show the *mean* within each group. But, that's all they show. Barplots might be more appropriate than barplots if you have so many groups that a boxplot is too cluttered to read, or if the

¹Distribution x_4 is a mixture of a normal with mean = 5 and SD = 2 with a uniform in [3, 20].

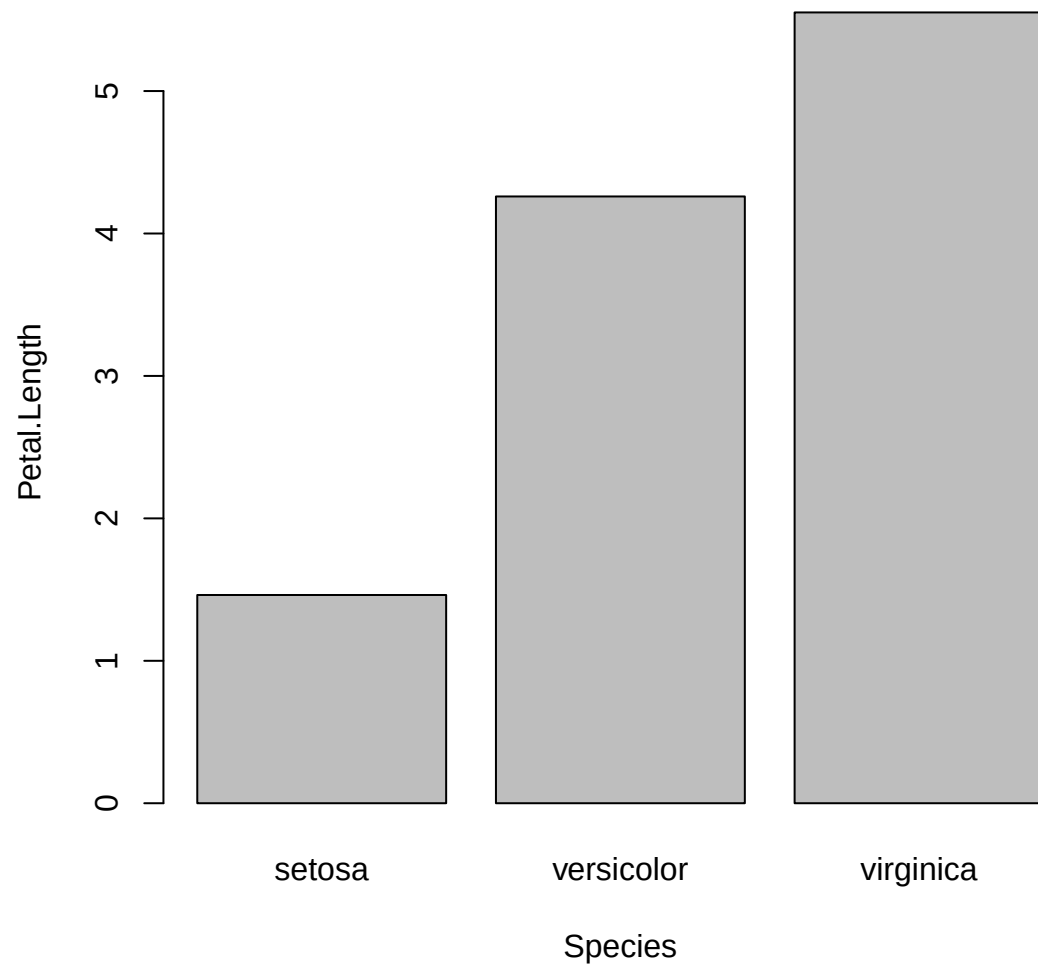
distributions within each group are known to be symmetrical and about the same width (and thus, showing them is not necessary).

Unlike `boxplot()`, `barplot()` does not calculate its own values. The values supplied to `barplot()` will be used as the bar heights. This means you need to calculate the means (or medians or whatever) ahead of time using a function like `aggregate()`. In the example, below, SD is also calculated so the plot will include some measure of variability. This is one of the disadvantages of barplots in R compared to boxplots: `boxplot()` automatically computes and displays variability about the mean, but `barplot()` does not.

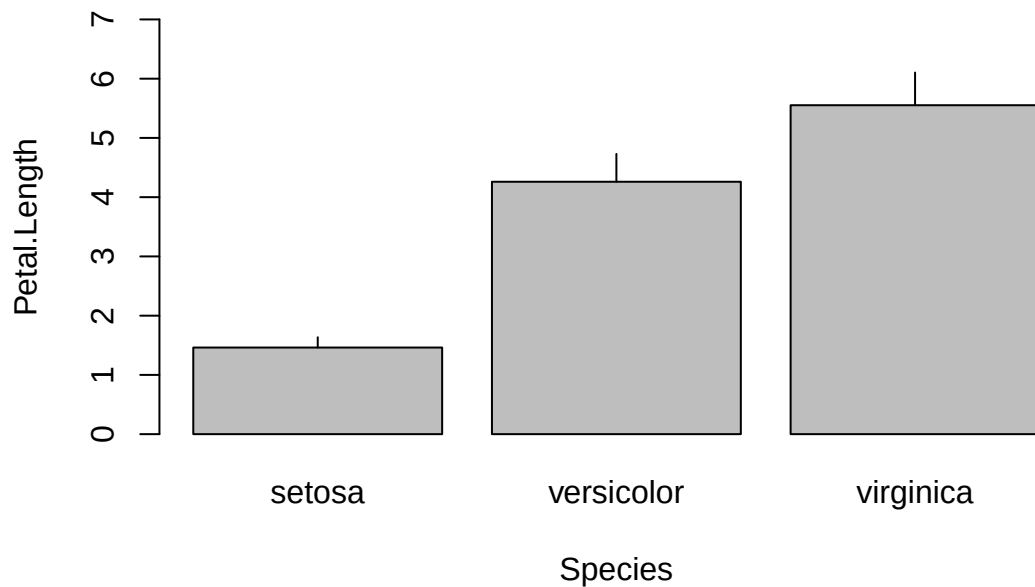
Another difference is that unlike `boxplot()`, the x coordinates in a barplot are calculated by the `barplot()` function. What this means is that you often need to run it twice: once to get the x coordinates, and again to make the plot. Those x coordinates are needed to place the line segments that show the SD.

```
# get means and SD of a y variable by species in iris
agg <- aggregate(Petal.Length~Species, data=iris, mean)
agg$sd <- aggregate(Petal.Length~Species,
                    data=iris, sd)$Petal.Length

# get x axis values
bar <- barplot(Petal.Length~Species, data=agg)
```



```
# make plot with second call
barplot(Petal.Length~Species, data=agg, ylim=c(0, 7))
segments(bar[,1], agg$Petal.Length, bar[,1], agg$Petal.Length+agg$sd)
```

The above plot is called a “dynamite plot” because the bar and error bar look like an old stick of dynamite. Because they show less than a boxplot, and are harder to make than a boxplot, I almost never make dynamite plots. Instead, I suggest you use boxplots or something like the “point-and-whisker” plots below (Section [8.1.3](#)).

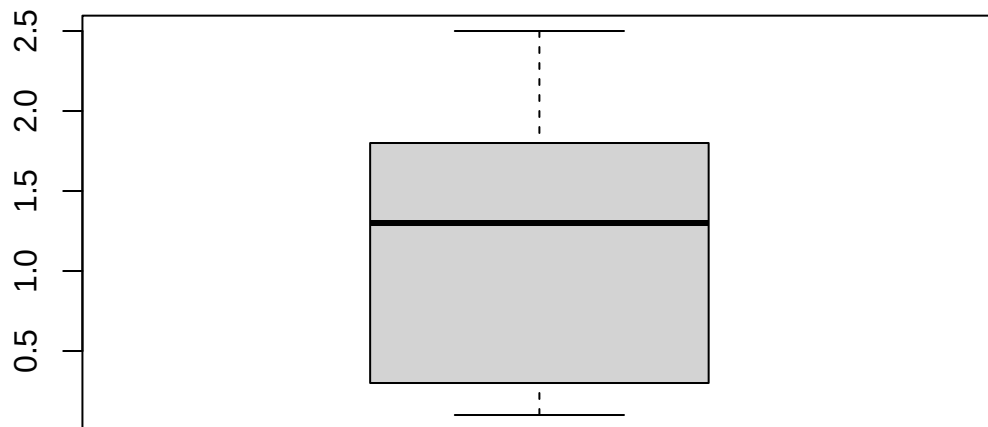
8.1.3 Boxplots

Boxplots, or **box-and-whisker plots**, summarize the distribution of continuous variables. They are often used to summarize data by a factor, making it easier to see how the distribution of a variable differs between groups. The boxplot shows the median, the first and third quartiles, and extreme values (whiskers and points)².

Boxplot of a single variable:

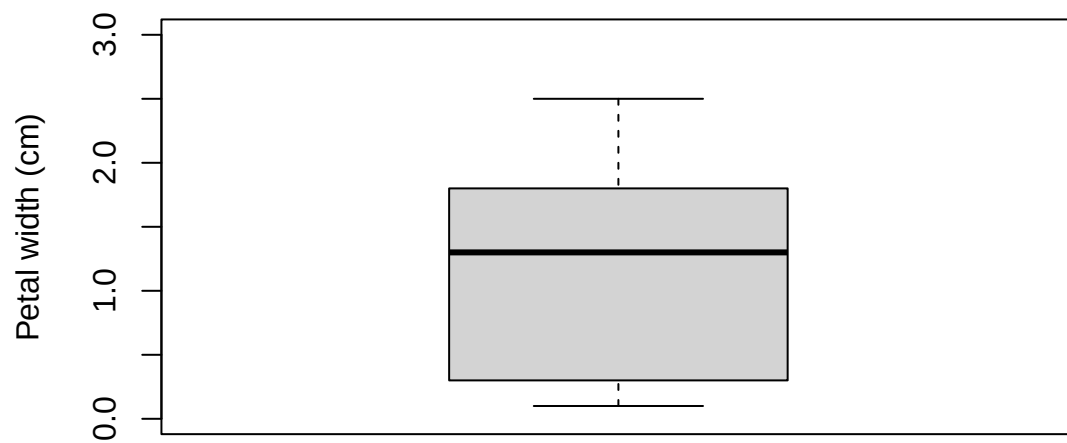
²Or close enough. The actual, slightly different values, are in the documentation. See `?boxplot.stats`.

```
boxplot(iris$Petal.Width)
```



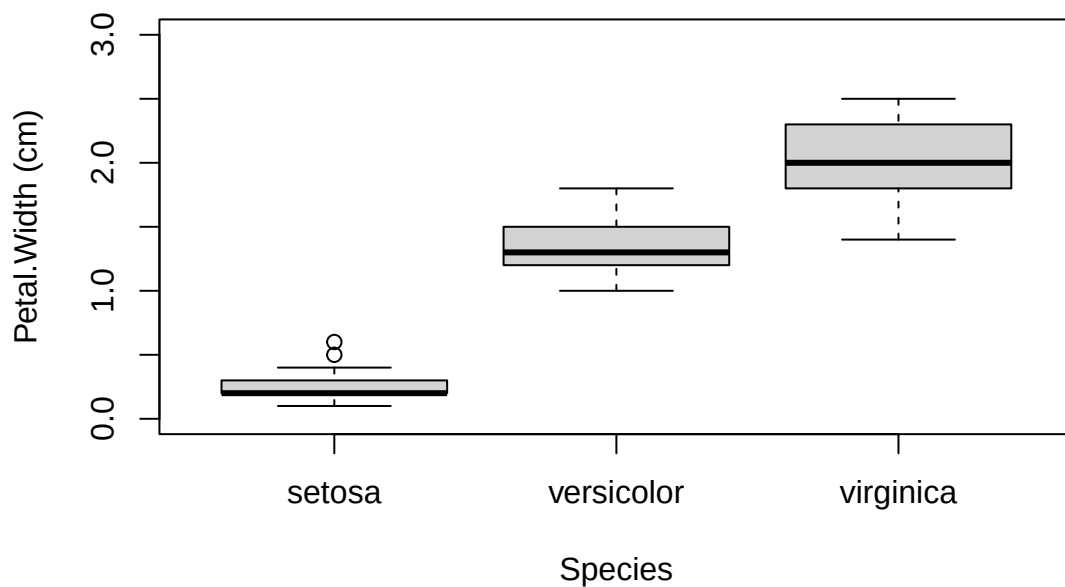
Boxplot with some options for a nicer plot:

```
boxplot(iris$Petal.Width,  
        ylab="Petal width (cm)",  
        ylim=c(0, 3))
```



Boxplot across levels of a factor (note formula interface):

```
boxplot(iris$Petal.Width~iris$Species,  
        xlab="Species",  
        ylab="Petal.Width (cm)",  
        ylim=c(0,3))
```



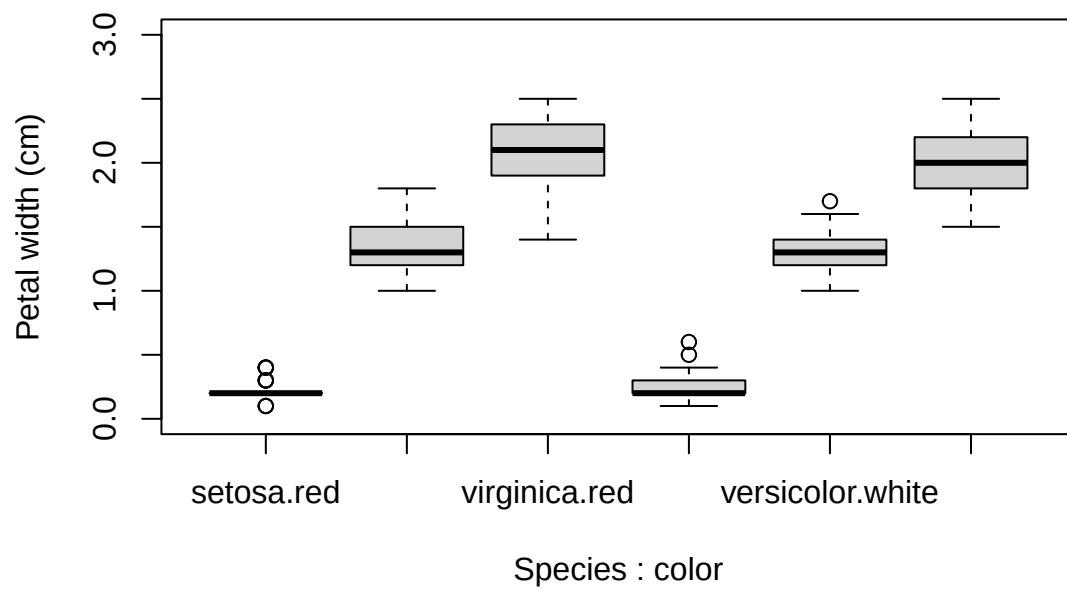
A highly asymmetric box-and-whisker plot can indicate that data are skewed or non-normal.

The base `boxplot()` function can produce summaries across multiple variables. Essentially it treats each combination of grouping variables as a separate set of values. There are better ways to plot group differences (e.g., for publication), but the base `boxplot()` does just fine for data exploration.

Boxplots can be constructed using several grouping variables. The order of grouping variables in the formula will determine the order of boxes in the plot.

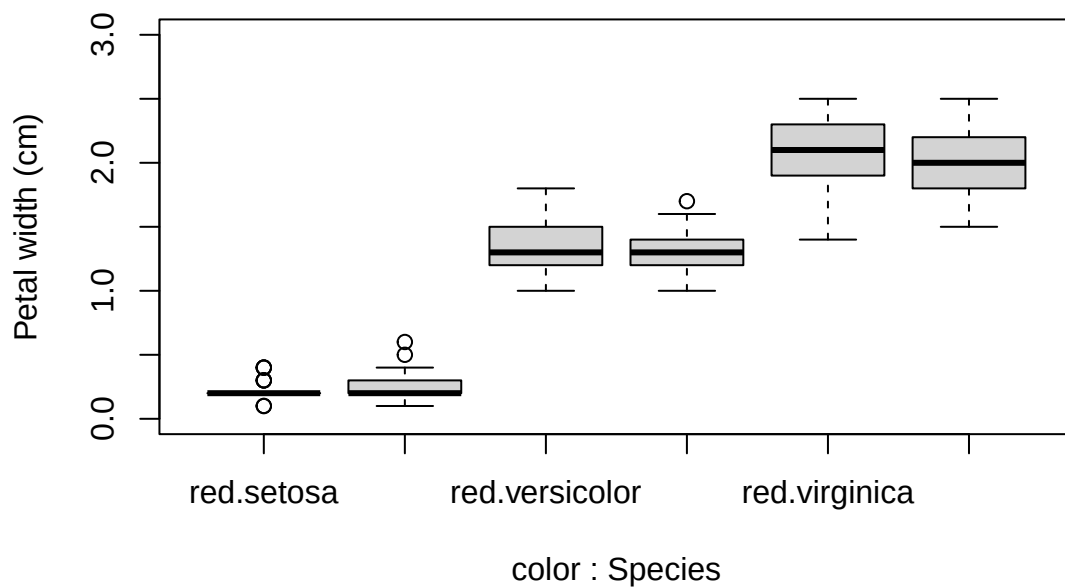
```
x <- iris
x$color <- c("red", "white") #made up variable

# group by species and color:
boxplot(Petal.Width~Species+color, data=x,
        ylab="Petal width (cm)", ylim=c(0,3))
```



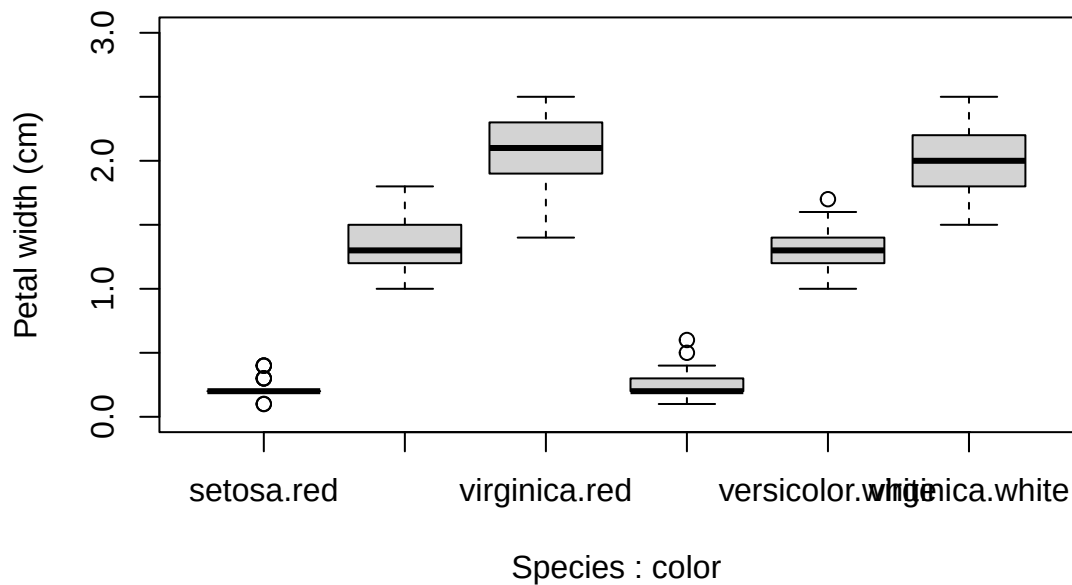
Notice the difference in order of grouping:

```
# difference in order of grouping:  
boxplot(Petal.Width~color+Species, data=x,  
        ylab="Petal width (cm)", ylim=c(0,3))
```



When constructing complicated boxplots, you may need to manually fix the x-axis labels, because they are not all rendered. Fortunately, the x-axis tick marks are simply numbered from left to right. So if, say, the 6th label was missing, you can draw it manually like this:

```
boxplot(Petal.Width~Species+color, data=x,
        ylab="Petal width (cm)", ylim=c(0,3))
# add missing tick mark with axis()
axis(side=1, at=6, labels="virginica.white")
```



If you want to construct a totally custom box and whisker plot, you can do so using base graphics functions like `rect()`, `arrows()`, `segments()`, and so on. All you need is to get the values for the different parts of the boxplot by saving a boxplot to an object.

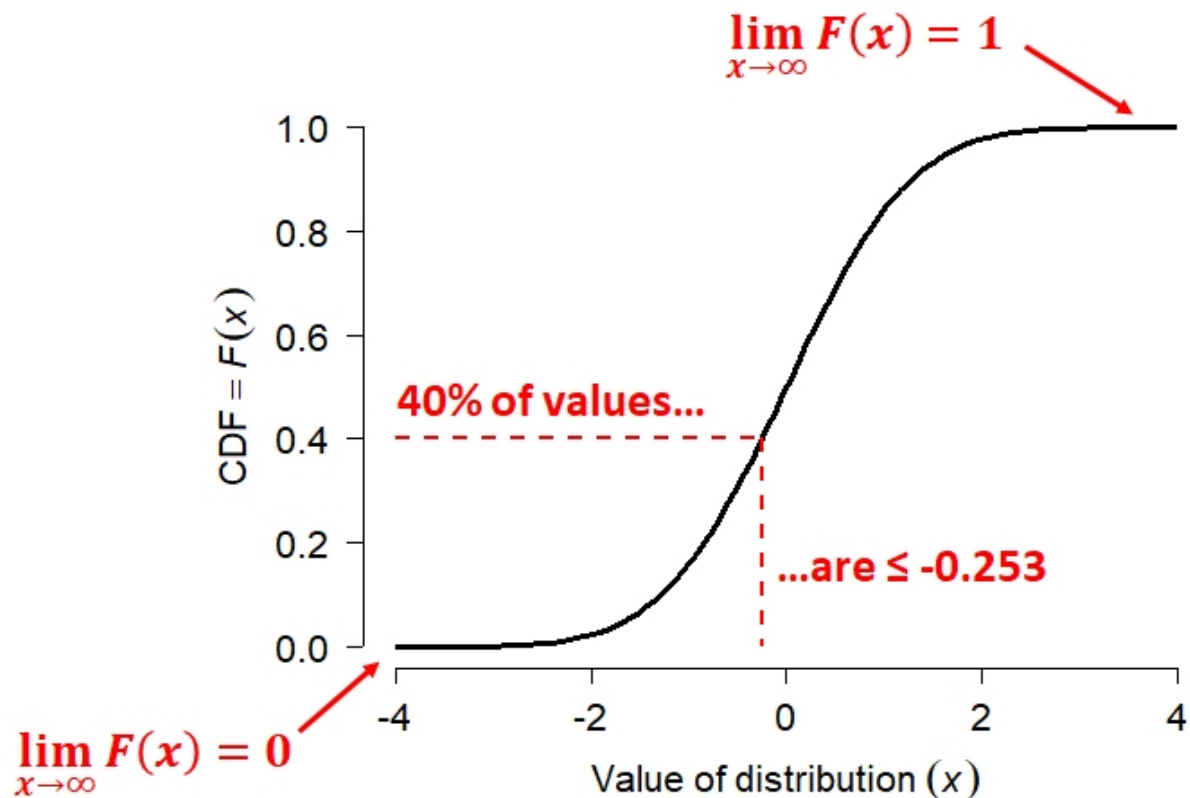
```
# not run
myboxplot <- boxplot(Petal.Width~Species+color, data=x)
```

The `stats` part of the resulting list will contain most of what you need. See `?boxplot.stats` for a guide.

8.1.4 ECDF plots

The **cumulative distribution function (CDF)** is the probability that a random variable will take on a value less than or equal to some value. Formally, we say that a continuous distribution x will take on value $\leq x$ with probability $F(x)$, where $F(x)$ is the CDF.

The figure below shows what this means.



{alt-

text: "Plot of the cumulative distribution function (CDF) of a normal distribution. The CDF is an S-shaped curve that increases smoothly from 0 to 1 as the value of the variable increases. Dashed red lines highlight the point where the CDF equals 0.4, showing that 40% of observations are less than or equal to $x = -0.253$. Additional annotations indicate that the CDF approaches 0 for very small values of x and approaches 1 for very large values of x . The figure illustrates that the CDF gives the cumulative probability up to a specified value of the variable."

The figure shows the CDF of a normal distribution with mean = 0 and SD = 1 (i.e., the standard normal distribution). Like all CDFs, $F(x)$ increases monotonically (slope never changing sign) from 0 to 1 as x increases from the lower bound to the upper bound of the distribution.

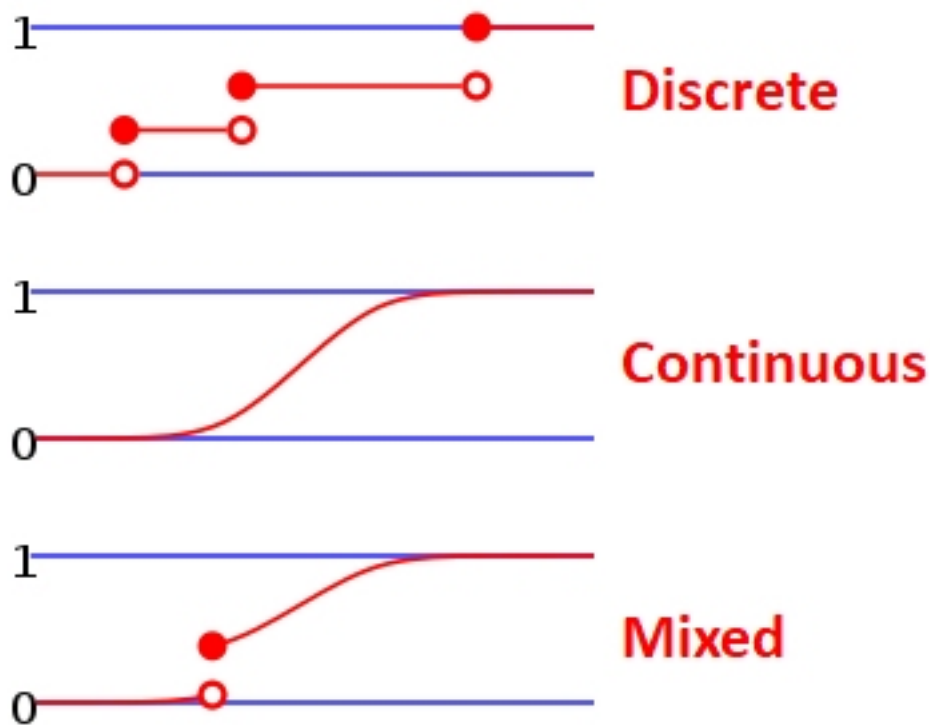
In the case of the normal distribution, the bounds of x are $[-\infty, +\infty]$. The CDF approaches 0 as x decreases, and approaches 1 as x increases.

The red dashed lines show how to interpret the relationship between the axes. For any value on the x -axis, the y -axis shows what proportion of values are $\leq x$. For any value on the y -axis, the x -axis shows the value at the y -th quantile of the distribution.

The CDF is in a very real sense the *definition* of a probability distribution.

Every probability distribution can be identified by a unique CDF. It doesn't matter whether a distribution is continuous, discrete, or a mixture of both. The figure below shows what the CDFs of a discrete or mixed

distribution look like compared to that of a continuous distribution³.

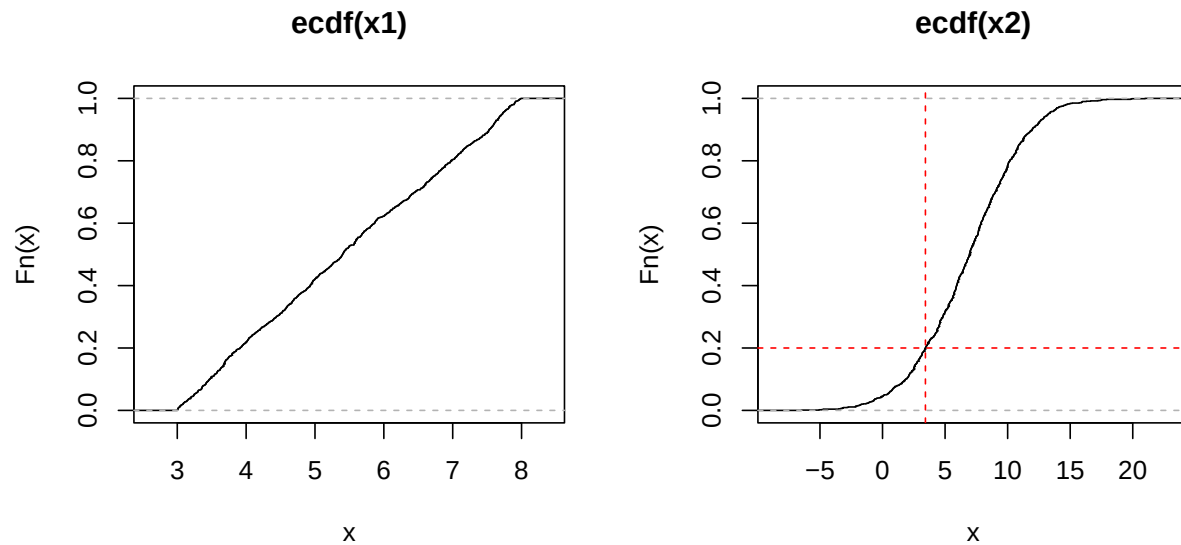


{Three schematic cumulative distribution functions (CDFs) illustrating different types of distributions. The top panel, labeled “Discrete,” shows a step function with horizontal segments and jumps; open circles indicate the CDF immediately before each jump and filled circles indicate the value after the jump. The middle panel, labeled “Continuous,” shows a smooth S-shaped curve with no jumps, indicating that the CDF changes continuously. The bottom panel, labeled “Mixed,” shows a smooth curve containing a single jump, illustrating a distribution with both continuous probability and discrete point masses. The figure emphasizes that CDFs are step functions for discrete variables, smooth for continuous variables, and a combination of both for mixed distributions.}

The examples below calculate and plot ECDF plots for a uniform (left) and a normal distribution (right).

```
set.seed(42)
n <- 1e3
x1 <- runif(n, 3, 8)
x2 <- rnorm(n, 7, 4)
par(mfrow=c(1,2))
plot(ecdf(x1))
plot(ecdf(x2))
# add lines at 20th percentile:
abline(h=0.2, lty=2, col="red")
abline(v=quantile(x2, 0.2), lty=2, col="red")
```

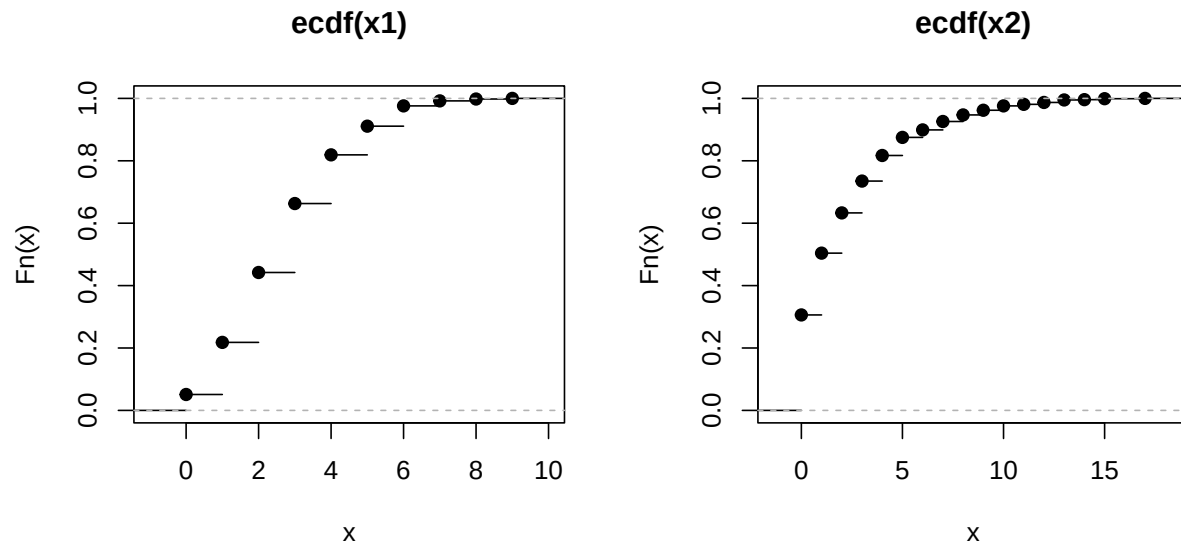
³adapted from https://commons.wikimedia.org/wiki/File:Discrete_probability_distribution_illustration.svg (public domain).



The y-axis of an ECDF plot shows the calculated quantile of the distribution (e.g., 0.4 quantile = 40th percentile—40% of x are $\leq F(x)$). The x-axis shows the values of the distribution. In the **left** plot, we see that the slope of the ECDF plot is roughly constant. This suggests a uniform distribution. The plot on the **right** shows the S-shaped ECDF typical of a bell-shaped curve, so we might suspect a normal distribution. The red lines on the right plot illustrate that about 20% of values are ≤ 3.6 .

ECDF plots can be calculated for discrete distributions as well—they just look like step functions. Step-like ECDF plots can also appear when a continuous distribution has many more observations than unique values.

```
set.seed(42)
n <- 1e3
x1 <- rpois(n, 3) # Poisson
x2 <- rgeom(n, 0.3) # geometric
par(mfrow=c(1,2))
plot(ecdf(x1))
plot(ecdf(x2))
```



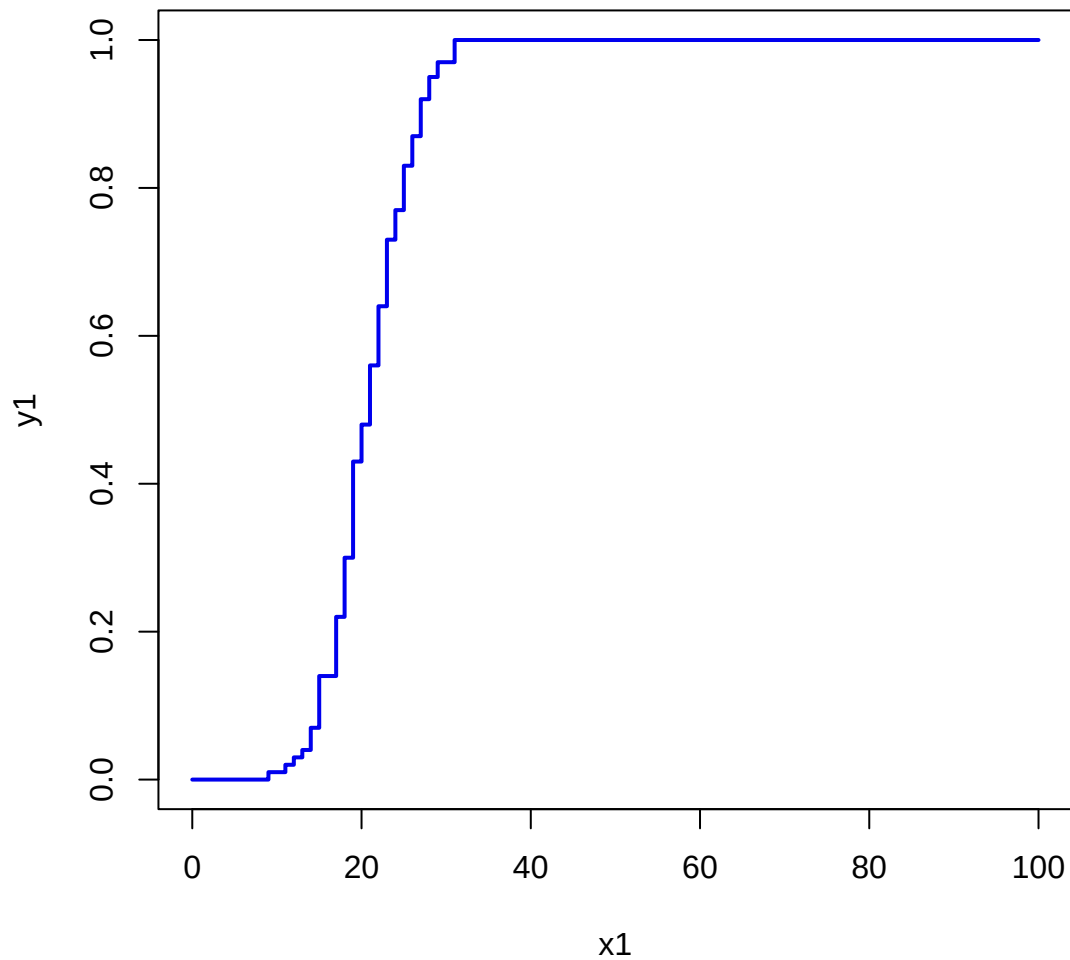
Recall that every result in R is an object. The output of the function `ecdf()` is actually another function that calculates the estimated CDF for a new value. This can be very handy if you want to interpolate the ECDF to make a smoother curve.

```
class(ecdf(rnorm(100)))
```

```
[1] "ecdf"      "stepfun"   "function"
```

Here is how to use it. The example below can be useful if you need to calculate the ECDF for values that are not in the original dataset, but within its domain. In the example below, the ECDF is estimated for $x = 30$, which is not in the original data x . Notice what happens for values in $x1$ that are greater than $\max(x)$ and less than $\min(x)$.

```
set.seed(123)
x <- rnorm(1e2, 20, 5)
e1 <- ecdf(x) # define function for CDF(x)
x1 <- 0:100 # define new x values at which to calculate CDF
y1 <- e1(x1) # calculate CDF at each new x
par(mfrow=c(1,1))
plot(x1, y1, type="s", lwd=2, col="blue2")
```



Obviously, the more data points you have, the better and smoother the estimate of the ECDF will be. Consider the simulated example below that estimates the ECDF using different numbers of points.

```
set.seed(123)
n <- 1e4
mu <- 20
sig <- 5
x4 <- rnorm(n, mu, sig)
x1 <- x4[sample(1:n, 10, replace=TRUE)]
x2 <- x4[sample(1:n, 100, replace=TRUE)]
x3 <- x4[sample(1:n, 1000, replace=TRUE)]

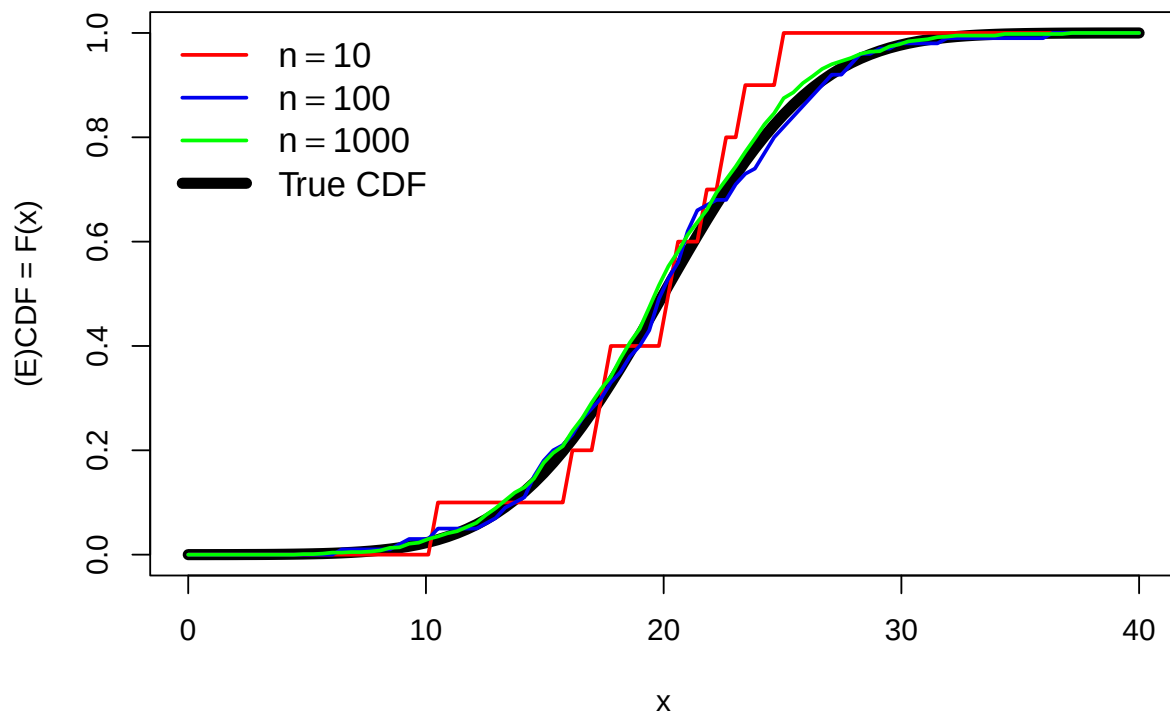
xseq <- seq(0, 40, length=100)
```

```

y1 <- ecdf(x1)(xseq)
y2 <- ecdf(x2)(xseq)
y3 <- ecdf(x3)(xseq)
yt <- pnorm(xseq, mu, sig)

plot(xseq, yt, type="l", lwd=6, xlab="x",
      ylab="(E)CDF = F(x)")
points(xseq, y1, type="l", lwd=2, col="red")
points(xseq, y2, type="l", lwd=2, col="blue2")
points(xseq, y3, type="l", lwd=2, col="green")
legend("topleft",
      legend=c(expression(n==10), expression(n==100),
                  expression(n==1000), "True CDF"),
      lwd=c(2,2,2,6),
      col=c("red", "blue2", "green", "black"),
      bty="n", cex=1.2)

```



The estimate with 10 points (red) is a very rough fit, but serviceable. The estimates using 100 or 1000 points

are much closer to the truth (black line).

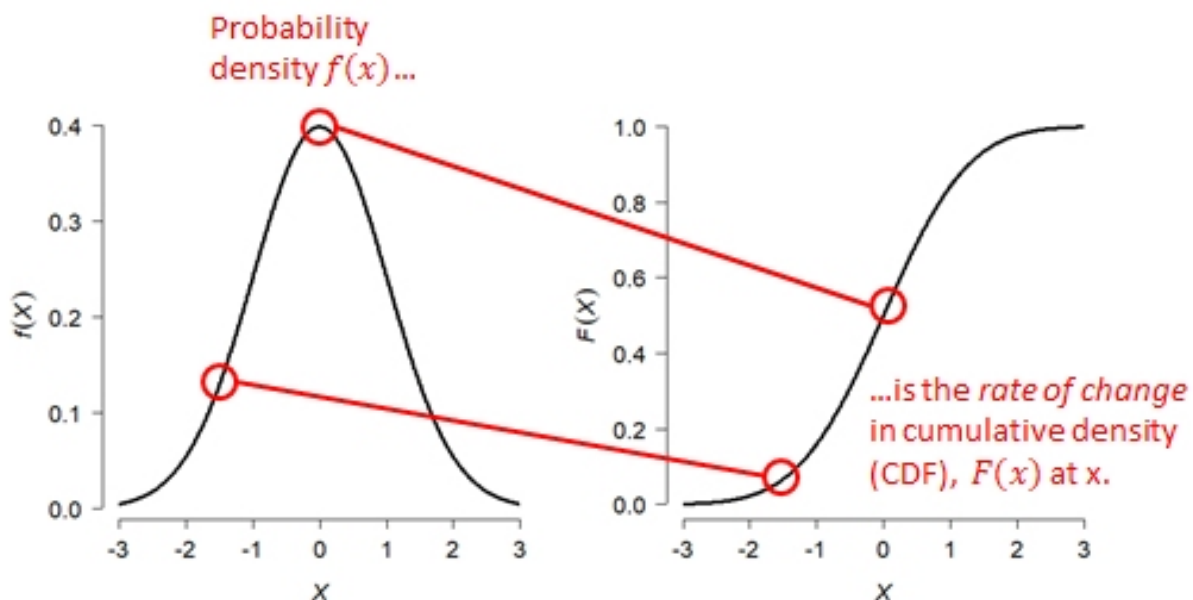
8.1.5 Kernel density plots

Kernel density plots are a way to empirically estimate and visualize the **probability distribution function (PDF)** of a distribution (Section 9.3). A plot of the estimated PDF is essentially a histogram where the bin width approaches 0. The PDF tells us two things. Practically speaking, the PDF of a distribution at a given value is related to how likely that value is to occur relative to other values. However, that is NOT the same as the probability of that value occurring.

What the PDF really expresses is the *rate of change in cumulative density at a value*.

- The **cumulative density** of a distribution at some value x is the probability of a value $\leq x$.
- The **cumulative density function (CDF)** gives the cumulative density at x .

The PDF is defined as is the *slope or first derivative of the CDF*. Conversely, the CDF is the integral of the PDF. The PDF is the rate at which the probability of observing a value $\leq x$ increases at each x . Again, this is related to but not the same as the probability of that value occurring.



{fig-

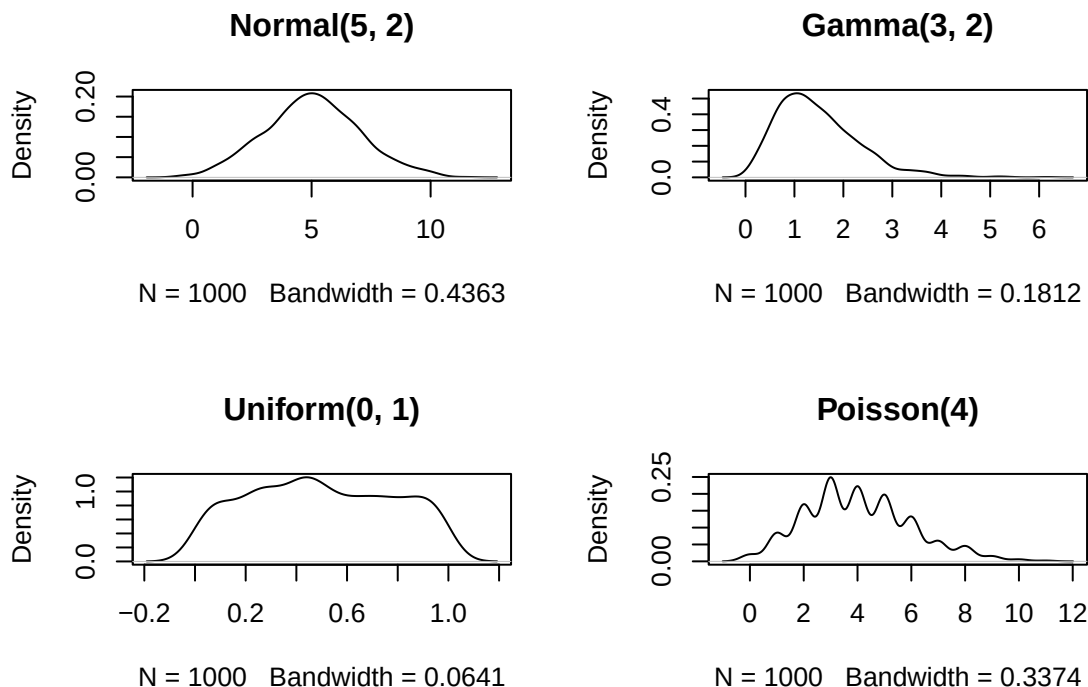
alt: "Relationship between the probability density function (PDF) and cumulative distribution function (CDF) of a normal distribution. The left panel shows the PDF, $f(x)$, as a bell-shaped curve. The right panel shows the corresponding CDF, $F(x)$, as an S-shaped curve increasing from 0 to 1. Red annotations connect corresponding values of x on the two graphs, illustrating that the probability density $f(x)$ at a given value equals the rate of change (slope) of the cumulative distribution function $F(x)$ at that same value. Larger PDF values correspond to steeper slopes of the CDF, while smaller PDF values correspond to flatter portions of the CDF."}

The kernel density plot estimated by R is to the PDF what the empirical cumulative distribution function (ECDF; see Section 8.1.4) is to the CDF. Just like an ECDF plot presents an estimate of the true CDF, the kernel density plot presents an estimate of the true PDF. This means that presenting a kernel density plot or an ECDF plot is largely a matter of preference because they convey the same information in different ways.

Note that a kernel density plot makes more sense for a **continuous distribution** than for a **discrete distribution**. The equivalent plot for a discrete distribution is the **probability mass function (PMF)** plot, which looks like a barplot because the PMF is only non-zero for integer values.

The example below shows kernel density plots for four distributions: *Normal* ($\mu = 5, \sigma = 2$), *Gamma* ($k = 3, \theta = 2$), *Uniform* ($\min = 0, \max = 1$), and *Poisson* ($\lambda = 4$). The first 3 are continuous distributions and the last is a discrete distribution.

```
set.seed(123)
n <- 1e3
par(mfrow=c(2,2))
plot(density(rnorm(n,5,2)), main="Normal(5, 2)")
plot(density(rgamma(n, 3, 2)), main="Gamma(3, 2)")
plot(density(runif(n)), main="Uniform(0, 1)")
plot(density(rpois(n,4)), main="Poisson(4)")
```

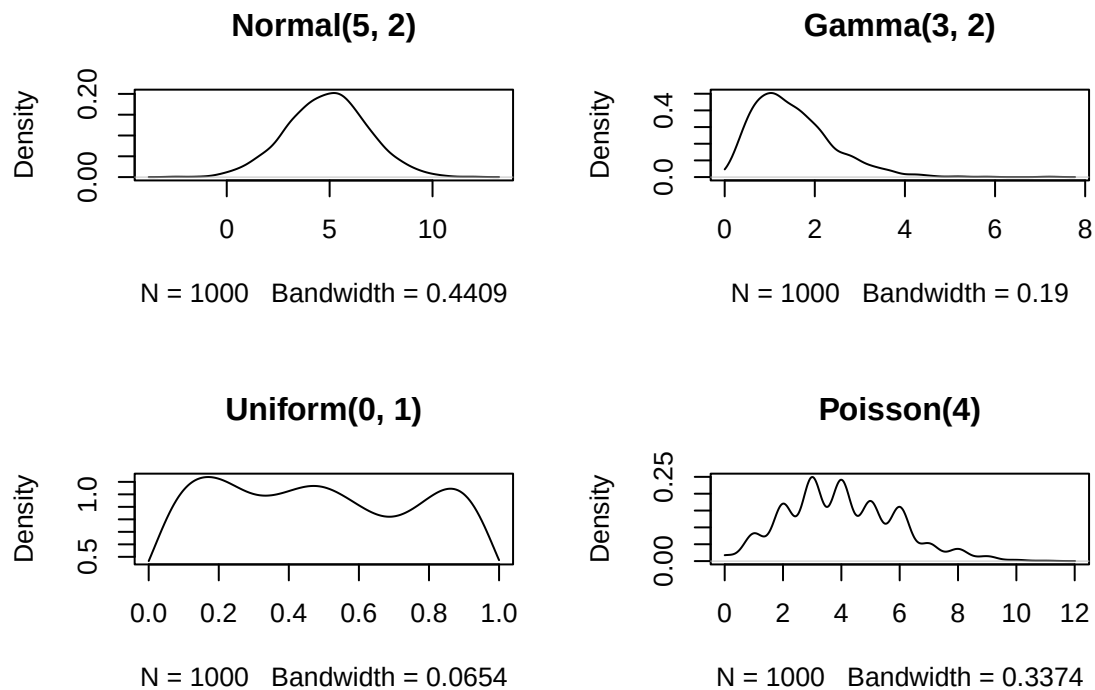


The estimated density functions aren't too far off from the real functions. But, notice that the plots for the gamma, uniform, and Poisson variables extend outside of the domains of the underlying distributions. For example, the density for the Poisson variable extends below 0—because the Poisson distribution models

counts, it is only supported for non-negative integers (think about why...you can't count -3 or 2.5 fish). You can truncate the plot using arguments `from` or `to` for density. This is a good idea if you are exploring data that you suspect have a natural domain.

For example, lengths and times must be non-negative, so you should include `from=0` in the `density()` command.

```
par(mfrow=c(2,2))
plot(density(rnorm(n,5,2)), main="Normal(5, 2)")
plot(density(rgamma(n, 3, 2), from=0), main="Gamma(3, 2)")
plot(density(runif(n), from=0, to=1), main="Uniform(0, 1)")
plot(density(rpois(n,4), from=0), main="Poisson(4)")
```



The estimated density functions are now in their supported intervals, which is nice. There's one more issue with the estimates for the Poisson distribution. Notice that the bottom right plot shows non-zero probability density for non-integer values. This doesn't make sense for a Poisson distribution, which can only take non-negative integer values. What's going on here is that the `density()` function does not know whether a distribution is supposed to be discrete or continuous. As far as the function is concerned, the distribution is continuous and just happens to have no non-integer values.

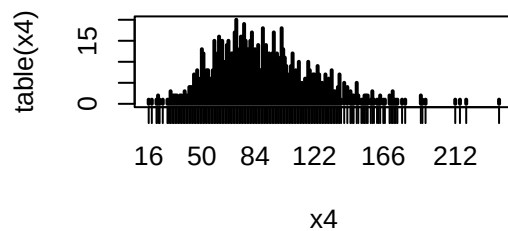
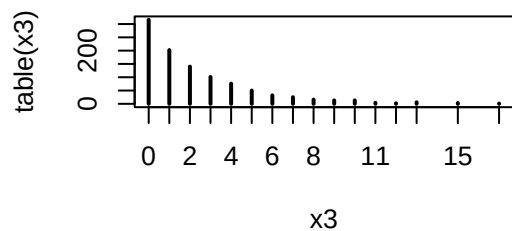
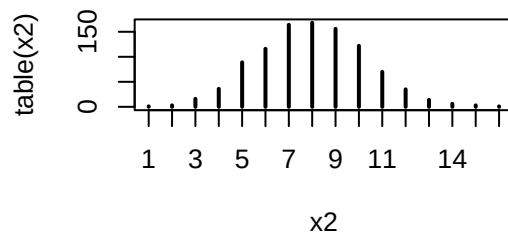
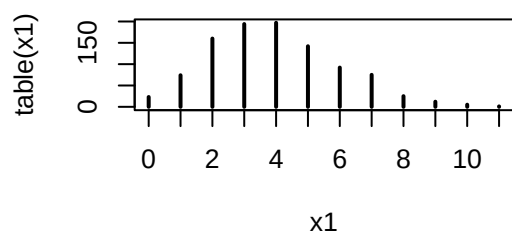
If you have data that you have good reason to suspect are discrete, there is a better way than `density()` to visualize the relative likelihood of different values: the **probability mass function (PMF, not PDF)**. The PMF of a discrete distribution can be calculated rather than estimated.

In the example below we use the `table()` function, which tallies up the number of times each value occurs

in a vector. We could convert the cell counts to PMF estimates by dividing by the number of observations (e.g., `plot(table(x1)/n)`).

```
# make some data
set.seed(42)
n <- 1e3
x1 <- rpois(n, 4)
x2 <- rbinom(n, 20, 0.4)
x3 <- rgeom(n, 0.3)
x4 <- rnbinom(n, 10, 0.1)

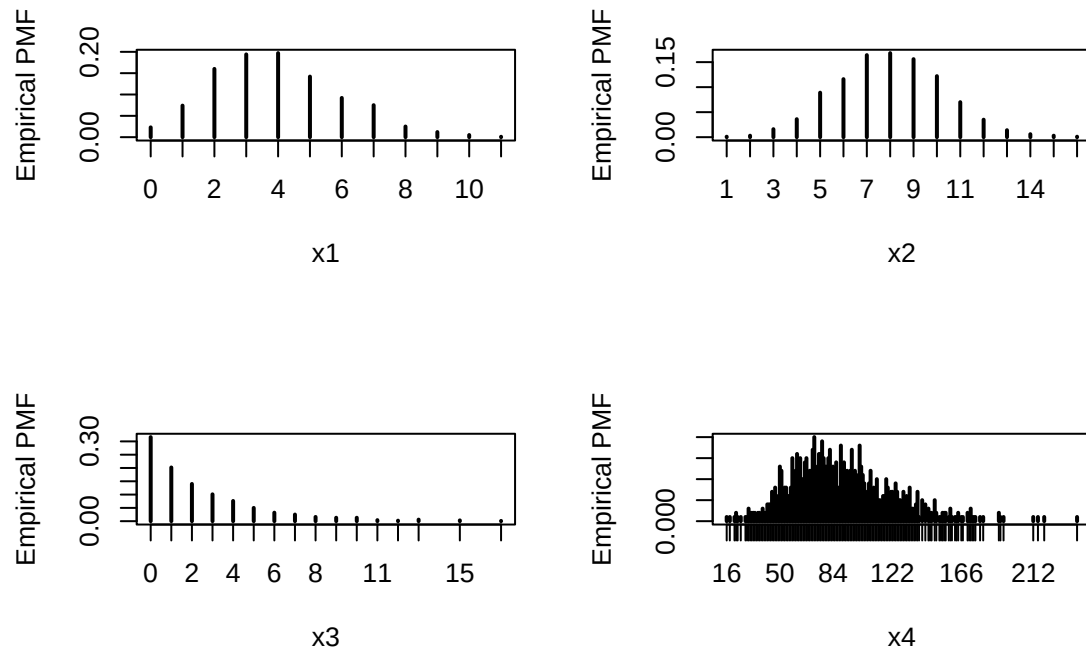
# plot cell counts:
par(mfrow=c(2,2))
plot(table(x1))
plot(table(x2))
plot(table(x3))
plot(table(x4))
```



Dividing the cell counts by the number of total observations (n) estimates the empirical PMF:

```
par(mfrow=c(2,2))
plot(table(x1)/n, ylab="Empirical PMF")
plot(table(x2)/n, ylab="Empirical PMF")
```

```
plot(table(x3)/n, ylab="Empirical PMF")
plot(table(x4)/n, ylab="Empirical PMF")
```

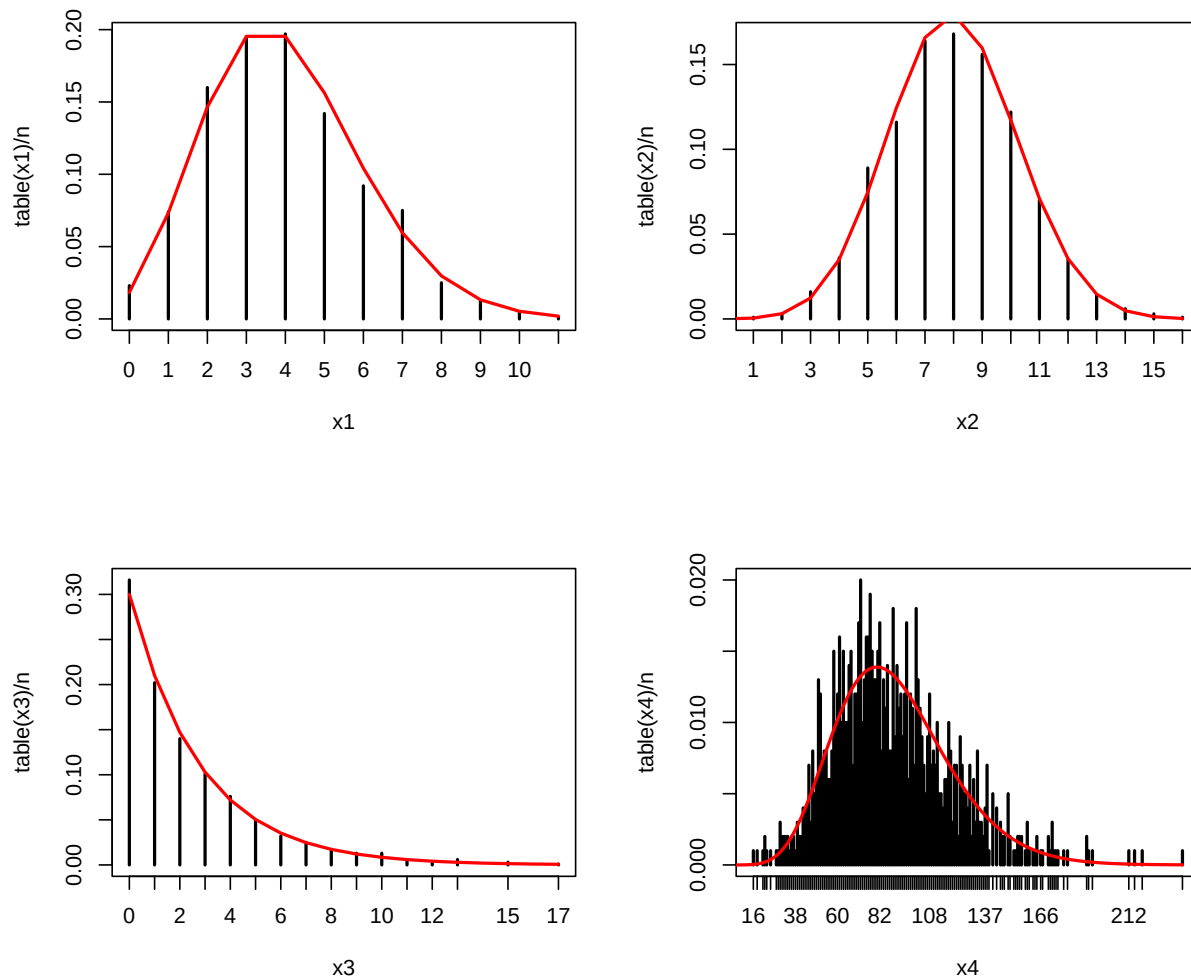


We can verify that the empirical PMFs that we calculated match the actual PMFs:

```
# compare to true PMF:
v1 <- 0:max(x1)
v2 <- 0:max(x2)
v3 <- 0:max(x3)
v4 <- 0:max(x4)
y1 <- dpois(v1, 4)
y2 <- dbinom(v2, 20, 0.4)
y3 <- dgeom(v3, 0.3)
y4 <- dnbinom(v4, 10, 0.1)

par(mfrow=c(2,2))
plot(table(x1)/n)
points(v1, y1, type="l", col="red", lwd=2)
plot(table(x2)/n)
points(v2, y2, type="l", col="red", lwd=2)
plot(table(x3)/n)
points(v3, y3, type="l", col="red", lwd=2)
```

```
plot(table(x4)/n)
points(v4, y4, type="l", col="red", lwd=2)
```

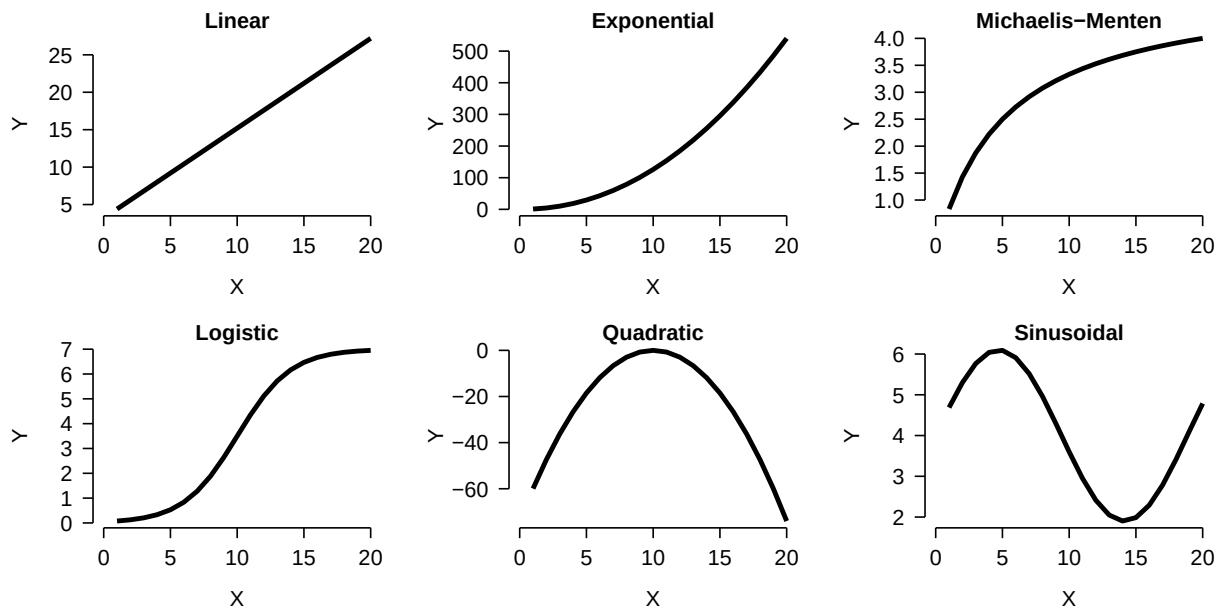


8.2 Visualizing 2 variables

So far in this course we have explored ways to explore distributions of single variables: histograms, ECDF plots, probability density plots, and so on. In this section we will explore ways to explore relationships *between* variables. This is a vital preliminary step in studying how two or more variables might be correlated with each other, or how one might cause the other. The focus on this page is on exploratory, correlative methods. Actual inference about relationships between two variables is better handled by **linear models (LM)**, **generalized linear models (GLM)**, or **other kinds of models**.

8.2.1 Scatterplots for 2 continuous variables

The **scatterplot** is one of the most important tools in the biologist's data toolbox. It is probably the simplest type of figure in routine use: simply plot the values in one variable on one axis, and the values of another variable on another (perpendicular) axis. The role of the scatterplot in exploratory data analysis is to help visualize the form of the relationship between two variables (Bolker 2008). The figure below shows just 6 of the possible curve types that you might discover.



8.2.1.1 R plotting basics

We have already made some scatterplots, but it's worth taking a closer look at the R methods for scatterplots. The basic function for scatterplots is `plot()`. The first argument is taken to be the x coordinates, and the second argument to be the y coordinates. You can also use the formula interface `plot(y ~ x)`, but the coordinates method `plot(x, y)` usually makes for cleaner code.

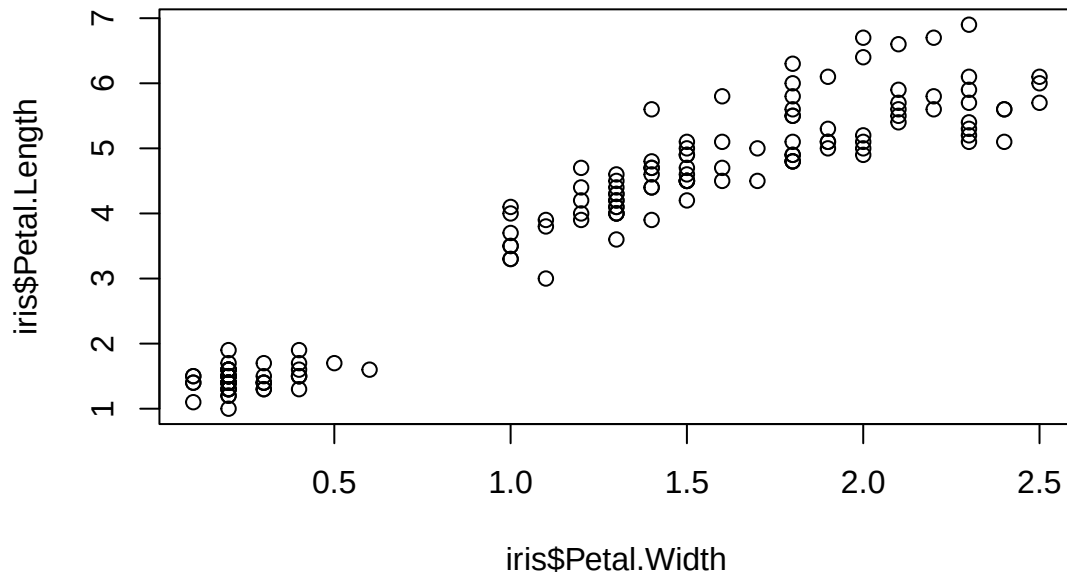
Calling `plot()` does two things: first, it *creates a new plot*; second, it *plots the data* on the new plot area. Other components can be added to the plot with subsequent commands:

- `points()` for points
- `text()` for text
- `lines()` for lines
- `polygon()` for polygons
- `segments()` for line segments

and so on. These functions will **add** to an existing plot, but will not create a new plot area. So, call `plot()` first, then `points()` or one of the others if needed. Calling `points()` or one of the “adding” functions if there is not already an active plot area will return an error.

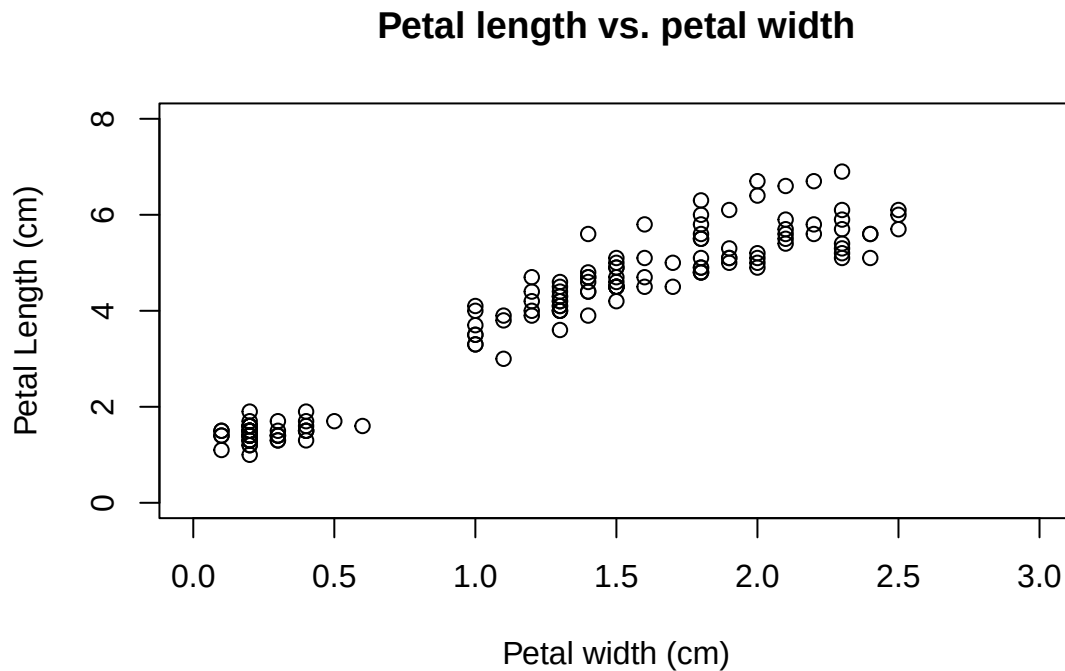
Here is a basic scatterplot:

```
plot(iris$Petal.Width,iris$Petal.Length)
```



The default scatterplot is rather unattractive, but works just fine for exploring data. Fortunately, almost every aspect of a plot can be customized using different plot arguments and graphical parameters. Here are some commonly used plot options:

```
plot(iris$Petal.Width,iris$Petal.Length,  
      xlab="Petal width (cm)",           # x label  
      ylab="Petal Length (cm)",         # y label  
      xlim=c(0,3),                      # x axis limits  
      ylim=c(0,8),                      # y axis limits  
      main="Petal length vs. petal width" # plot title  
)#plot
```



Sometimes it helps to change the color or style of points to show categories in the data. This can be done using some arguments to `plot()`:

- `pch` changes the symbol used for points. Think “**p**oint **c**haracter”. See `?points` for a list of available symbols.
- `cex` changes the size of symbols. Think “**c**haracter **e**xpansion”. The default `cex` is 1; other values scale the points relative to this. E.g., `cex=2` makes points twice as large.
- `col` changes the color. Think “**c**olor”. R can produce many colors; run the command `colors()` to see a named list⁴.

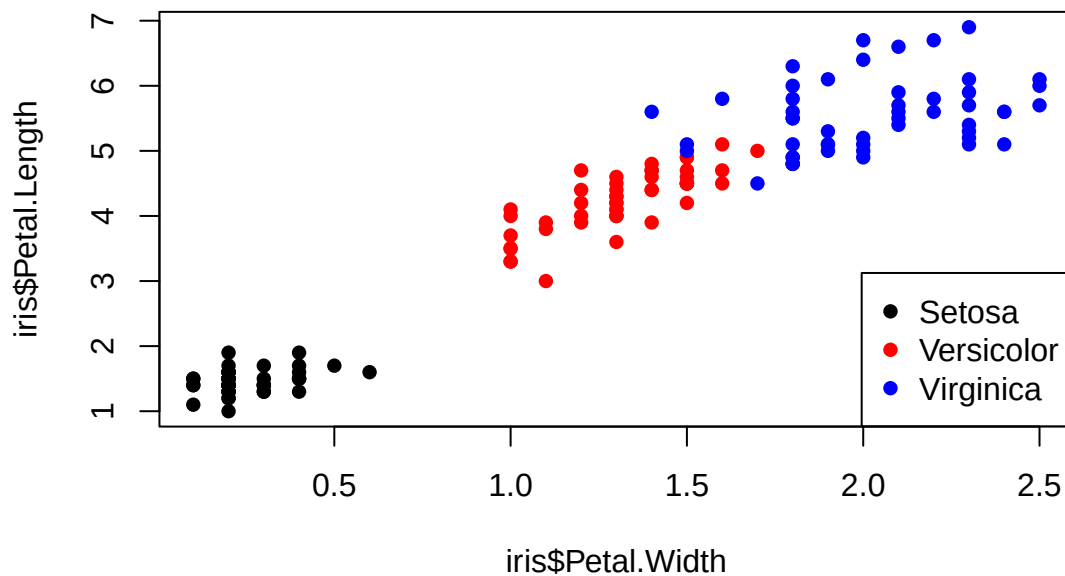
Each these arguments can take vector of values, so you can assign colors or shape to each point. The values will be used in the same order as the observations used to draw the points (e.g., rows of a data frame). I find it convenient to put colors, shapes, and other point-specific graphical parameters into the dataframe that contains the coordinates. That way, if I ever subset the data frame, or sort the data frame, the symbology information stays with each point.

The examples below illustrate several ways of assigning symbology to observations within a data frame. In the first, `ifelse()` is used to make vectors of colors (`use.col`) defined by the variable `species`. Notice that a legend is provided to tell the reader which symbols mean what. This method works fine if you only have a few groups to define symbols for.

⁴You can also specify colors by their RGB or hex codes. Just google “R color chart” to get some examples. The [ColorBrewer](#) website (accessed 2026-08-05) and R package can help picking appropriate color schemes.

```
# method 1: ifelse()
iris$color <- ifelse(iris$Species == "setosa", "black",
  ifelse(iris$Species == "versicolor", "red", "blue"))

plot(iris$Petal.Width, iris$Petal.Length,
  col=iris$color, # color by species
  pch=16        # solid dots
)
legend("bottomright",
  legend=c("Setosa", "Versicolor", "Virginica"),
  pch=16, col=c("black", "red", "blue"))
```



A better way is to first set up a vector of colors, then name the elements of the vector, and finally use those names to assign colors to rows of the data frame. ***This is the method I use in my code, and recommend you do too.*** When getting a vector of colors, I often use [colorbrewer2](#) (date accessed 2026-08-05). Notice in the `legend()` code below that we don't have to retype the species names or colors because they are already in the object `spps` and `cols`.

```
# method 2: match()
#### recommended method ####

## get the species names
```

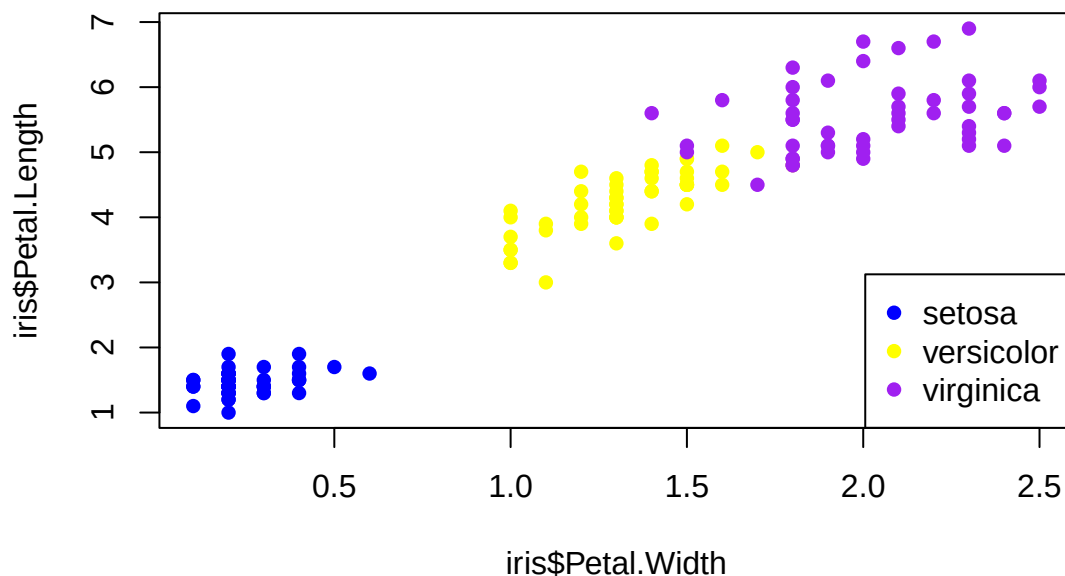
```
spps <- sort(unique(iris$Species))

## vector of colors
## can also use rainbow() or colorbrewer2.
cols <- c("blue", "yellow", "purple")

## assign names to colors
names(cols) <- spps

## assign to data frame
iris$color <- cols[iris$Species]

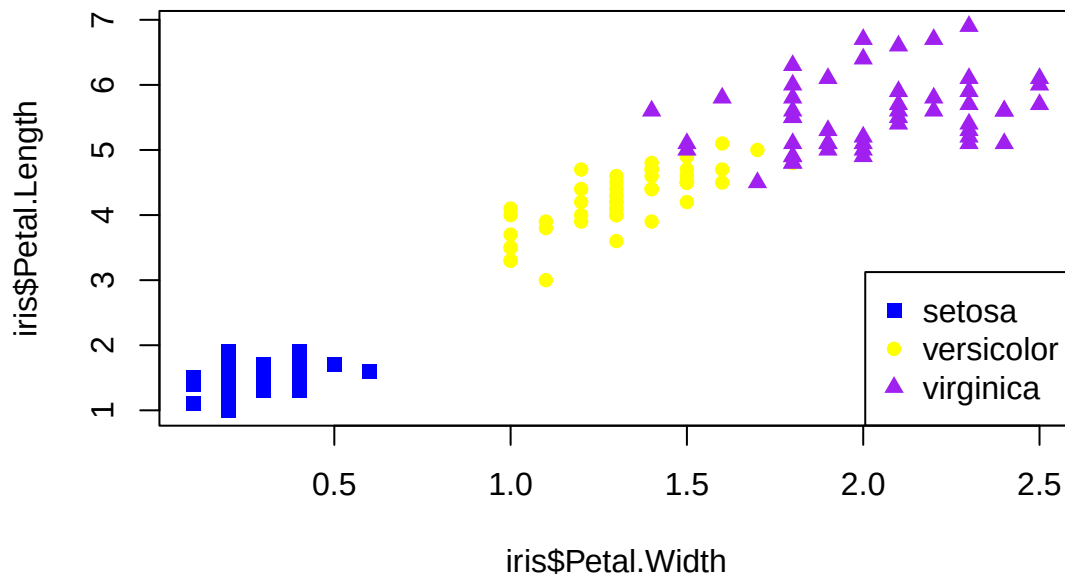
## use the new colors in a plot:
plot(iris$Petal.Width, iris$Petal.Length,
     col=iris$color, # color by species
     pch=16         # solid dots
)
legend("bottomright", legend=spps, pch=16, col=cols)
```



Below is an example of using shapes (use `.pch`) to define groups. See the help page for `points()` (`?points`) to see the available plot symbols. In my plots I usually use both shape and color to define groups.

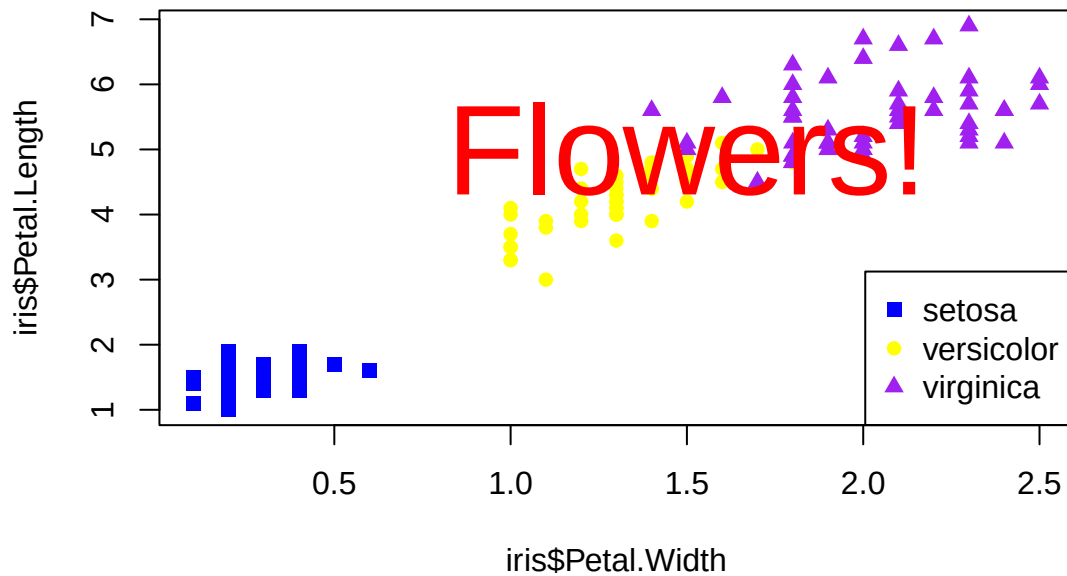

```
# method 3: names()
spps <- sort(unique(iris$Species))
shps <- c(15, 16, 17)
names(shps) <- spps
iris$pch <- shps[iris$Species]

plot(iris$Petal.Width, iris$Petal.Length,
     pch=iris$pch, col=iris$color)
legend("bottomright", legend=spps, pch=shps, col=cols)
```



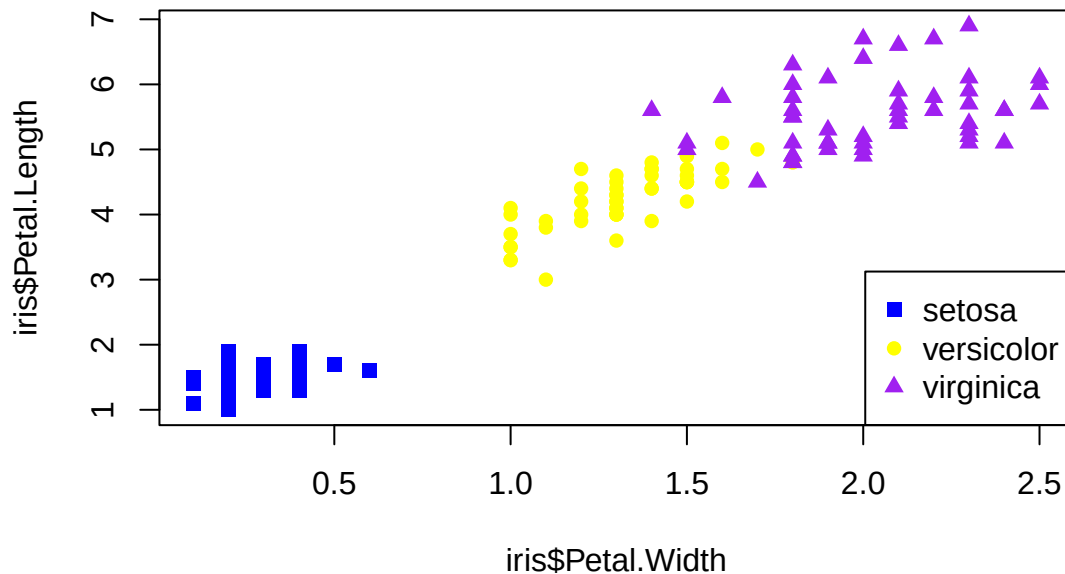
You can think of the plot generated by `plot()` like a canvas. Once the canvas is in place, things can be “painted” on, but like paint cannot be removed. If you want to remove something, you’ll need to remake the plot without it. You can also just comment out the command you don’t want.

```
plot(iris$Petal.Width, iris$Petal.Length,
     pch=iris$pch, col=iris$color)
legend("bottomright", legend=spps, pch=shps, col=cols)
text(1.5, 5, "Flowers!", cex=4, col="red")
```



There's no way to remove the offending text: R doesn't have an "Undo" button! Our only option is just to remake the plot without the text.

```
# remake without the offending text:
plot(iris$Petal.Width, iris$Petal.Length,
     pch=iris$pch, col=iris$color)
legend("bottomright", legend=spps, pch=shps, col=cols)
```



Sometimes I'll just comment out the plot component that needs deleting; this way it is easier to add back in later. I do this a lot when I'm trying to iteratively build a figure with lots of components.

```
# not run:

# remake without the offending text (commented out):
plot(iris$Petal.Width,iris$Petal.Length,
     col=use.col, pch=16
)
#text(1.5, 5, "Flowers!", cex=4, col="red")
legend("bottomright", legend=spps, col=cols, pch=16)
```

8.2.1.2 Advanced R plotting with `par()`

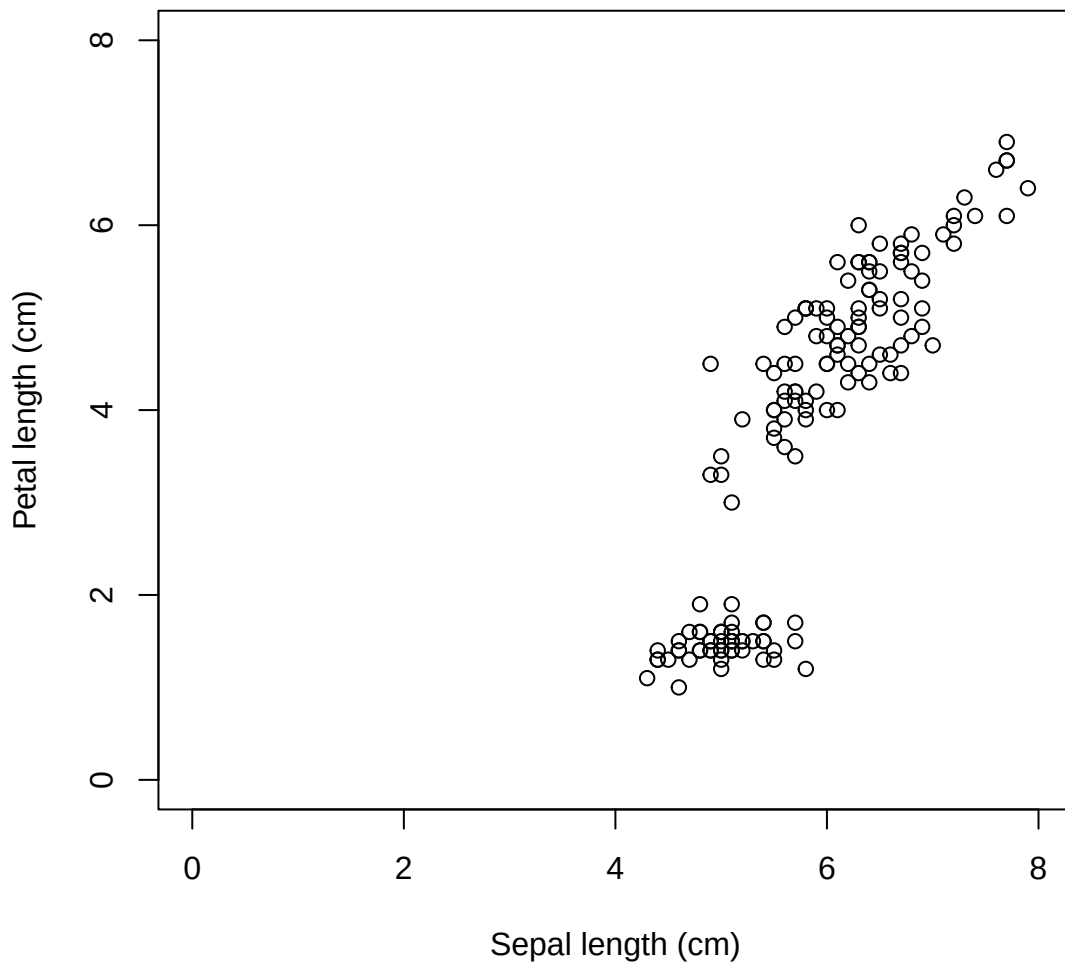
8.2.1.2.1 Plot formatting

Many options that affect plots can only be set using the `par()` function. Or, are best set using `par()`. Take a look at the `par()` help page to get a sense of the variety of options available (`?par`). It is important to keep in mind is that once `par()` options are set in an R session, they will stay that way until you change them using `par()` again. So, if you are making multiple figures within the same R workspace or work session you will need to pay attention to what you have done. You can always see all current settings by running the command `par()`.

Below is an illustration of using `par()` to change graphics options:

Without `par()`:

```
plot(iris$Petal.Length~iris$Sepal.Length,  
     xlab="Sepal length (cm)",  
     ylab="Petal length (cm)",  
     xlim=c(0,8),ylim=c(0,8))
```

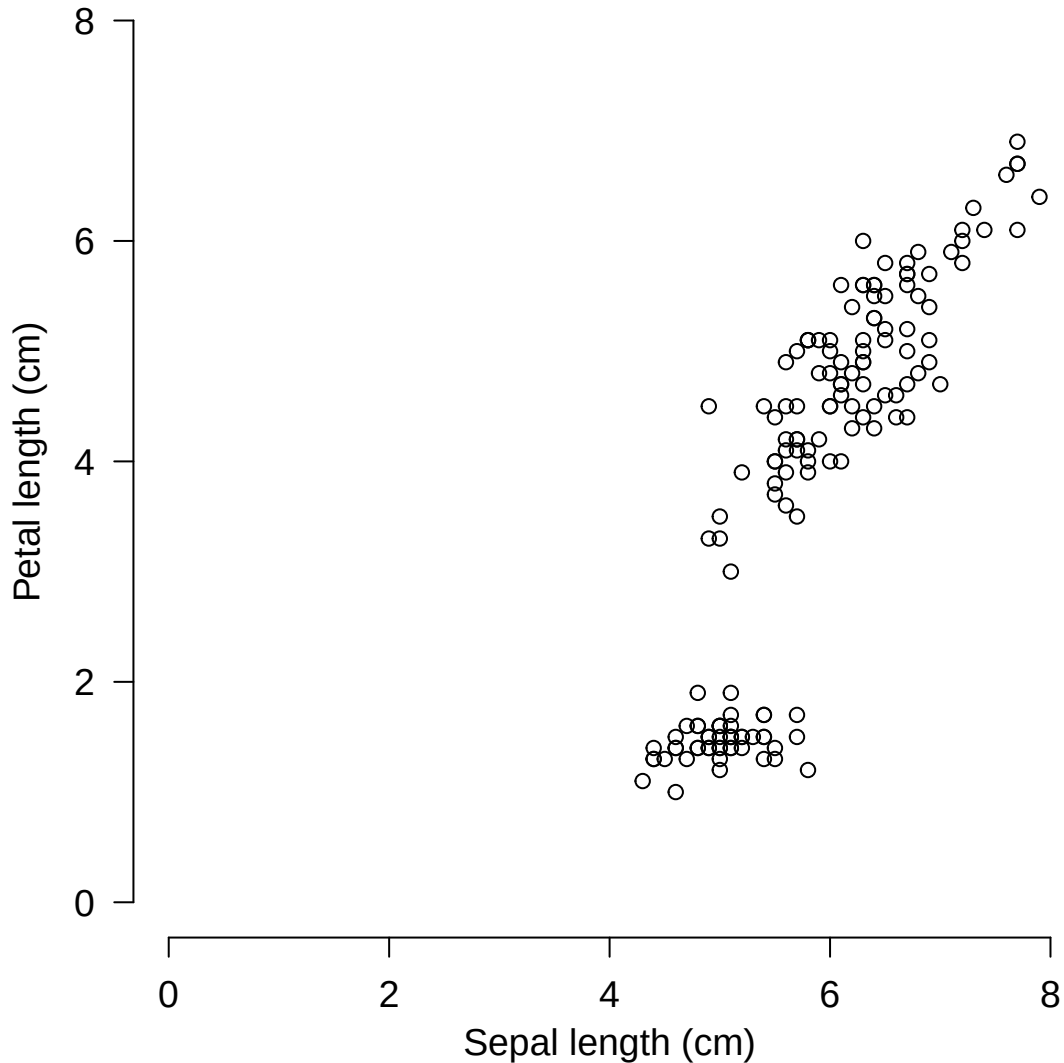


With `par()`, we can do the following:

- Narrow the margins (`mar`) so the plot takes up more of the figure space
- Remove the box (`bty`) because it is unnecessary
- Turn y-axis numbers to horizontal (`las`) so they are easy to read
- Bring axis labels in closer (`mgp`) to save space

- Set size of axis text (`cex.axis`) and labels (`cex.lab`) for legibility

```
par(mar=c(4.1, 4.1, 1.1, 1.1), # margin sizes
    bty="n",                     # no box around plot
    las=1,                      # axis labels in reading
                                # direction
    mgp=c(2.25, 1, 0),          # position of axis components
    cex.axis=1.2,               # size of axis numbers
    cex.lab=1.2)                # size of axis titles
plot(iris$Petal.Length~iris$Sepal.Length,
     xlab="Sepal length (cm)",
     ylab="Petal length (cm)",
     xlim=c(0, 8), ylim=c(0, 8))
```



We will use `par()` often in this course to clean up figures and make them more attractive. Most of the time when you are exploring data you won't need to mess with `par()` (except for making multi-panel figures). But, you should definitely use `par()` when preparing figures for presentations or publications. Proper use of `par()` is the key to making clean, informative, and professional-looking figures.

8.2.1.2.2 Multi-panel figures with `par()`

One of the most common ways to use `par()` is to make multi-panel figures. The panels are specified in terms of the number of rows and columns in the figure. The arguments `mfrow` and `mfcol` define the panel layout.

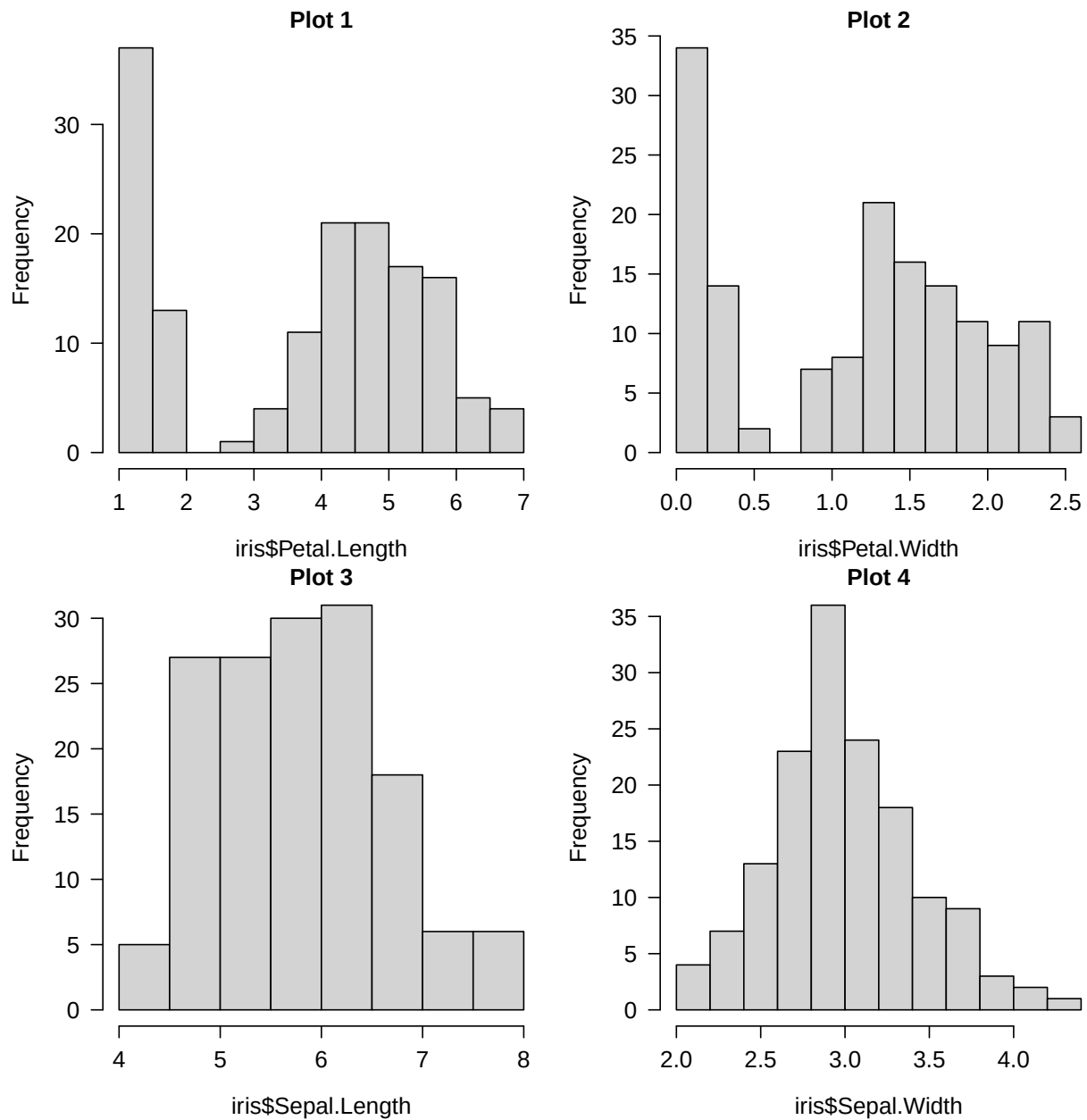
- `mfrow`: you supply the number of rows and number of columns, in that order.
- `mfcol`: works the same way, but in reverse: supply the number of columns and number of rows, in that order.

Once you set `mfrow` or `mfcol` (but never both), a new plot area within the graphics window will be produced each time you call `plot()`. The graphics window will be divided evenly according to the number of panels you requested (e.g., `mfrow=c(2,3)` will yield 6 panels). Panels are drawn by row and then by column (with `mfrow`) or by column and then by row (with `mfcol`).

If you produce more plots than you have “slots” specified by `mfrow` or `mfcol`, the graphics device will be cleared and the plot panels will be filled again, in the same order as before. So, if you set `mfrow=c(2,2)`, and then make 5 plots, you will end up with a graphics window that has 1 plot (the fifth) in the upper left corner.

The example below shows the use of `par()` `$mfrow` to make a 2×2 figure.

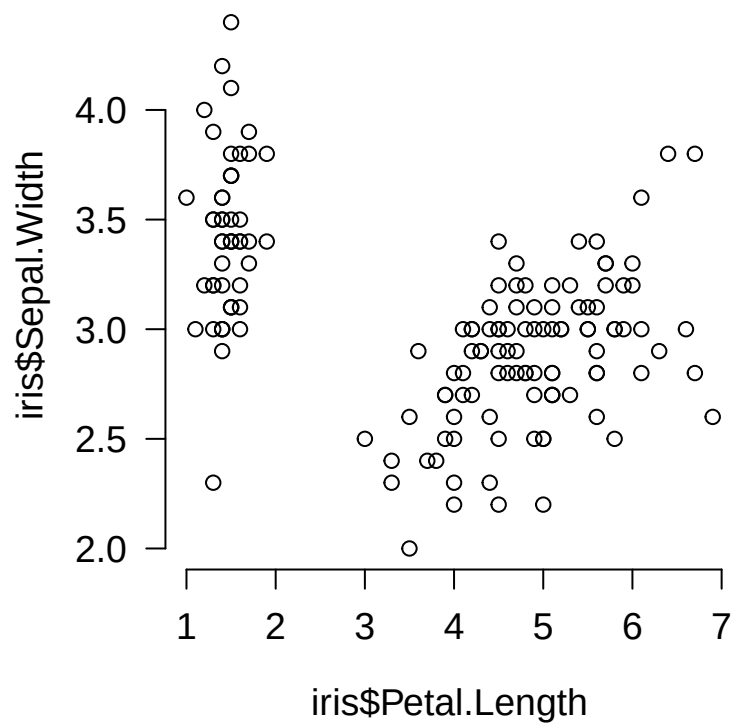
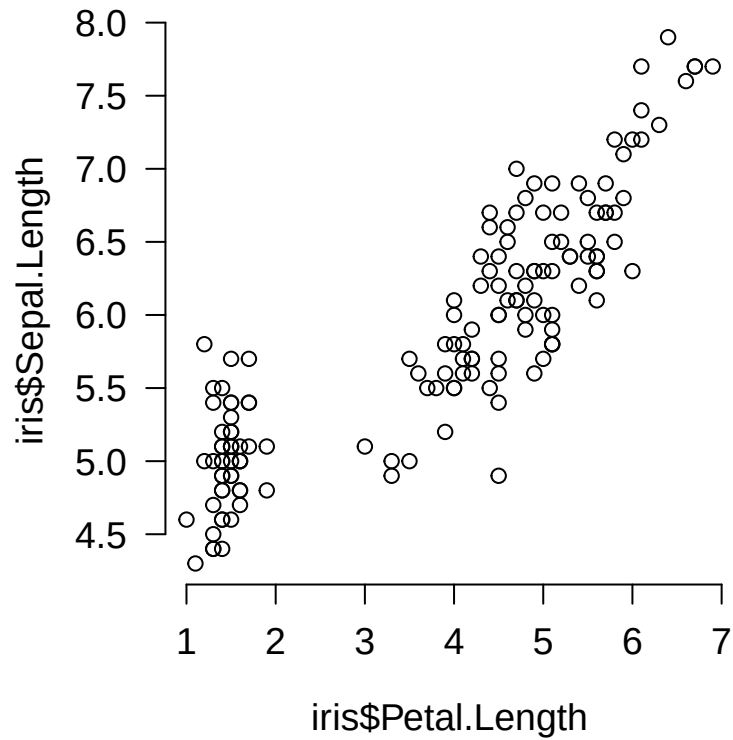
```
par(mfrow=c(2,2),           # layout
    mar=c(4.1,4.1,1.1,1.1), # margin sizes
    bty="n",                 # no box around plot
    las=1,                   # rotate axis text
    cex.axis=1.2,            # axis text size
    cex.lab=1.2)             # label text size
hist(iris$Petal.Length, main="Plot 1")
hist(iris$Petal.Width,  main="Plot 2")
hist(iris$Sepal.Length, main="Plot 3")
hist(iris$Sepal.Width,  main="Plot 4")
```



When making multi-panel plots, it helps to line up axes that correspond to each other. The commands below show two separate figures with aligned axes. This can be helpful when showing different responses to the same explanatory variable (the 2 row by 1 column figure):

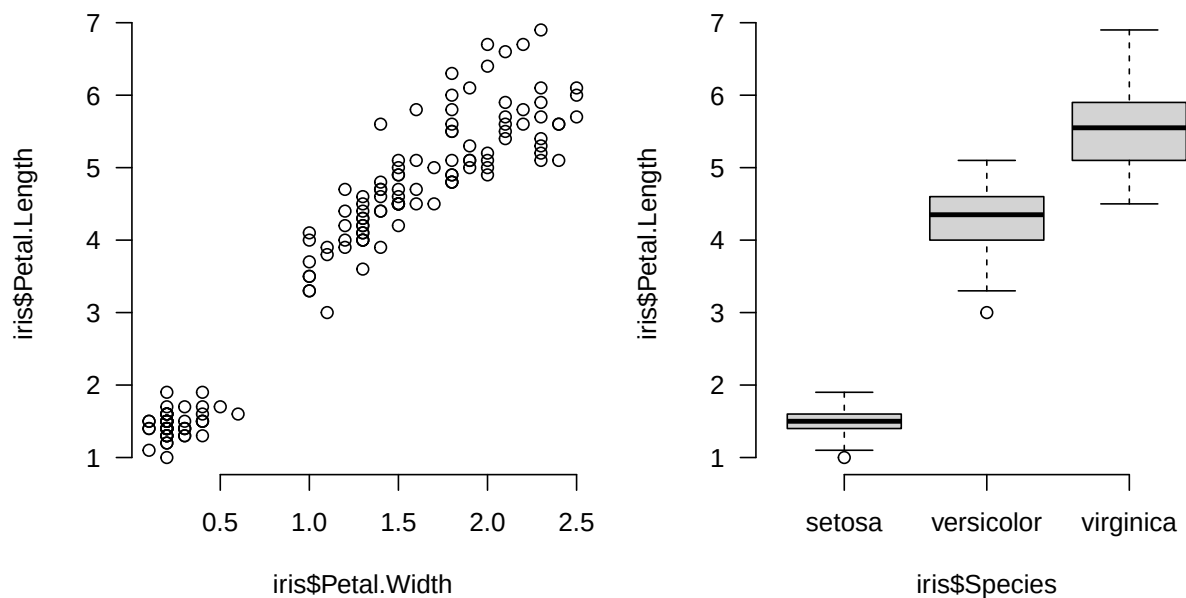
```
par(mfrow=c(2,1),           # layout
    mar=c(4.1,4.1,1.1,1.1), # margin sizes
    bty="n",                 # no box around plot
    las=1,                   # rotate axis text
    cex.axis=1.2,            # axis text size
    cex.lab=1.2)             # label text size
```

```
plot(iris$Petal.Length, iris$Sepal.Length)
plot(iris$Petal.Length, iris$Sepal.Width)
```



The figure below is a 1 row by 2 column figure that shows how a single response variable relates to two different predictors.

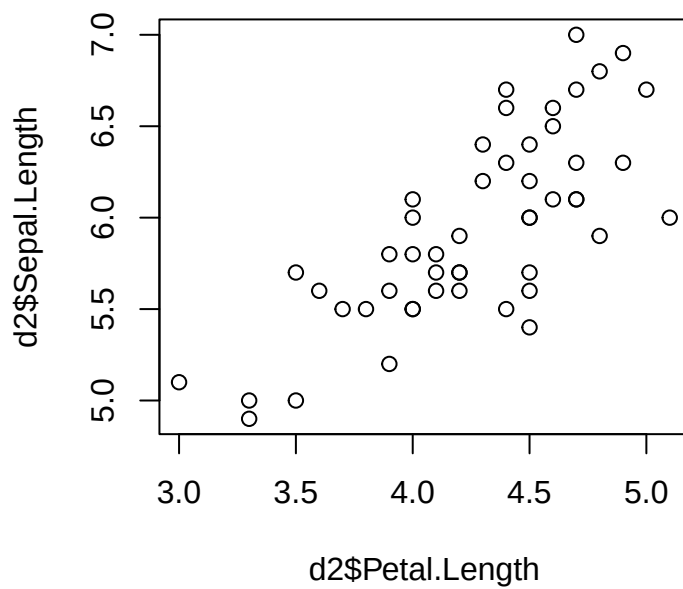
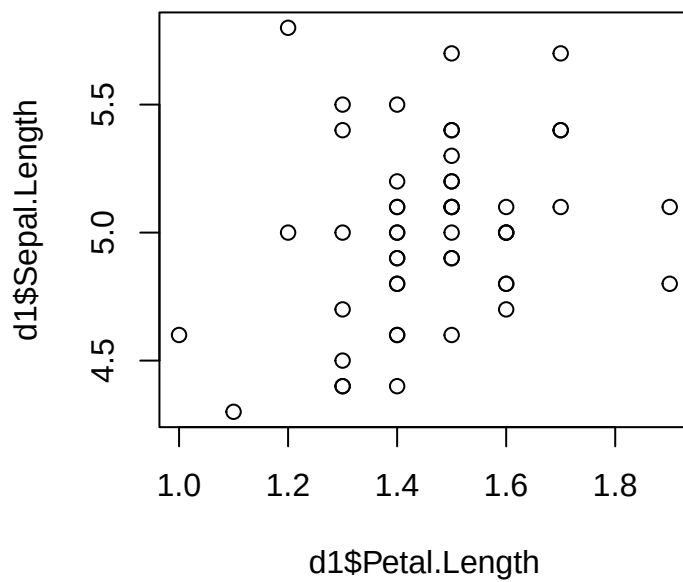
```
par(mfrow=c(1,2),           # layout
    mar=c(4.1,4.1,1.1,1.1), # margin sizes
    bty="n",                 # no box around plot
    las=1,                   # rotate axis text
    cex.axis=1,              # axis text size
    cex.lab=1)               # label text size
plot(iris$Petal.Width, iris$Petal.Length)
boxplot(iris$Petal.Length~iris$Species)
```



Sometimes you may need to manually set axis limits to make sure that the panels line up exactly.

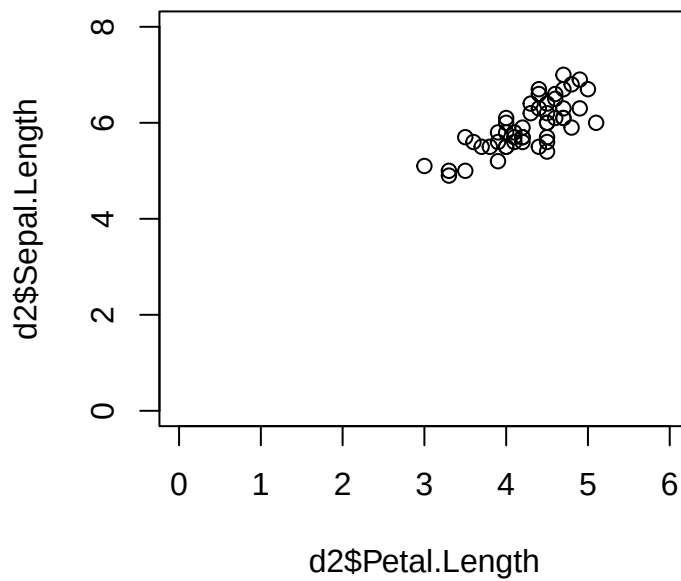
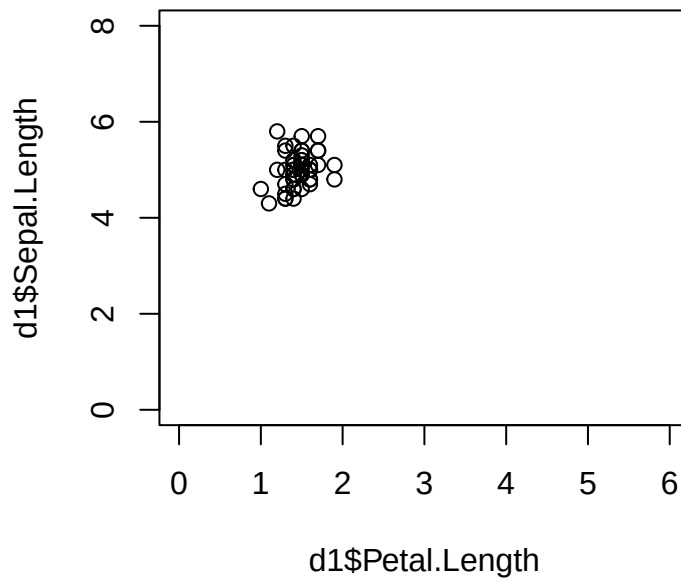
```
# example with X and Y axes not aligned:
d1 <- iris[which(iris$Species == "setosa"),]
d2 <- iris[which(iris$Species == "versicolor"),]

par(mfrow=c(2,1))
plot(d1$Petal.Length, d1$Sepal.Length)
plot(d2$Petal.Length, d2$Sepal.Length)
```



Here's the same plot, but with the axes lined up by setting the axis limits with `xlim` and `ylim`.

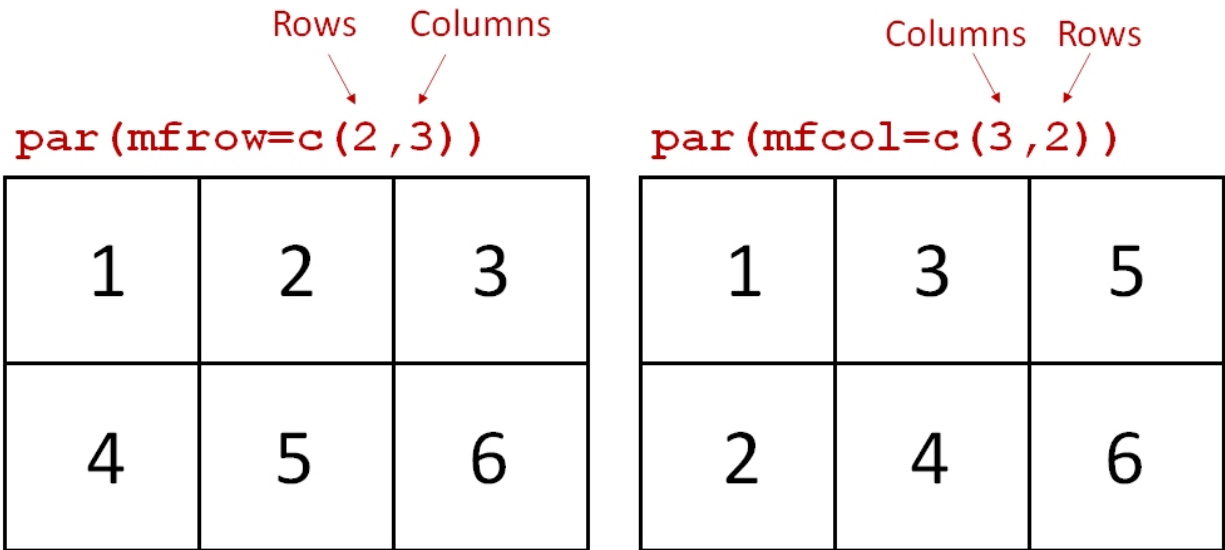
```
# same plot but with axes lined up:
par(mfrow=c(2,1))
plot(d1$Petal.Length, d1$Sepal.Length, xlim=c(0,6), ylim=c(0,8))
plot(d2$Petal.Length, d2$Sepal.Length, xlim=c(0,6), ylim=c(0,8))
```



The argument `mfc` to `par()` works similarly to `mrow`, but on columns instead of rows. For `mfc` you supply the number of columns, then the number of rows. Likewise, panels are filled by column first instead of row first. The panel layout `mrow=c(2, 3)` is the same as `mfc=c(3, 2)`, but the panels

will be filled in a different order:

Figure 8.1: Illustration of how `par()` `$mfrow` and `par()` `$mfc` draw and fill figure panels. `mfrow` fills by rows, while `mfc` fills by columns.

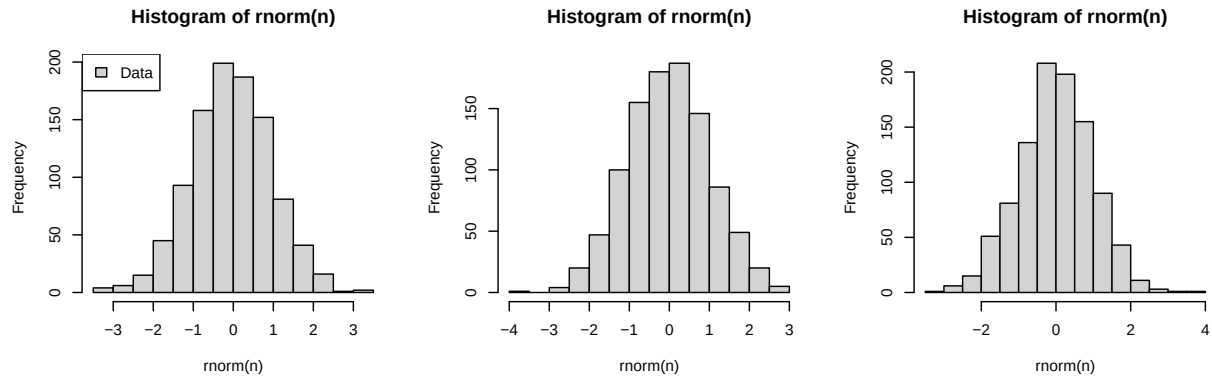


Whether to use `mfrow` or `mfc` is usually a matter of preference. Clever use of one or the other can allow you to automatically make multi-panel plots in a preferred layout in a `for()` loop from data stored in a list or indexed by a vector.

One last point about multi-panel plots: the `plot()` function both creates new panels and creates plots within each panel. Part of this process is defining the coordinate system for a panel. When you make a plot, subsequent commands that add elements to plots such as `points()`, `abline()`, `legend()`, etc., will use the coordinate system of the most recent panel. Think about this when designing complicated figures. The example below demonstrates how a legend is placed according to the coordinate system of the most recently-created plot.

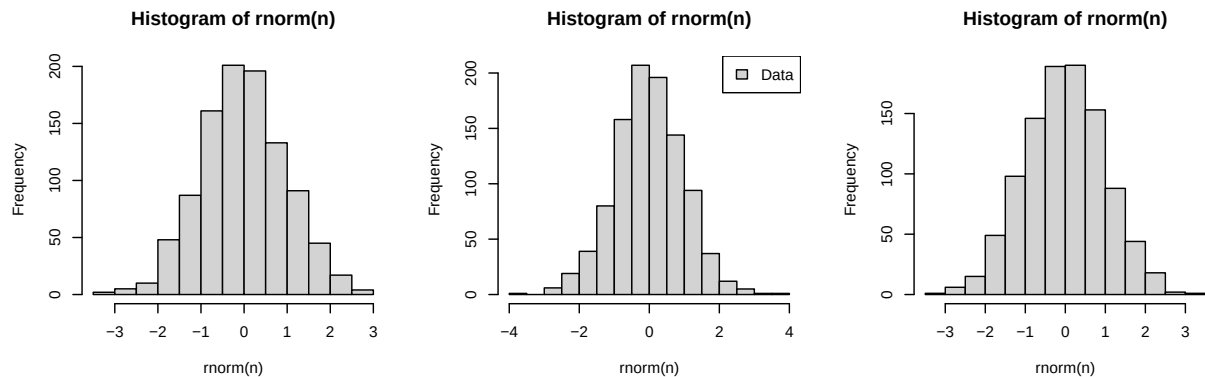
```
n <- 1e3

# legend in first panel:
par(mfrow=c(1,3))
hist(rnorm(n))
legend("topleft", legend="Data", fill="lightgrey")
hist(rnorm(n))
hist(rnorm(n))
```



Here is a different figure, with the legend in a different panel:

```
# legend in second panel (note where legend() is):
par(mfrow=c(1,3))
hist(rnorm(n))
hist(rnorm(n))
legend("topright", legend="Data", fill="lightgrey")
hist(rnorm(n))
```



For more advanced plotting, refer to Chapter 31. There we'll explore special symbols and equations with `plotmath()`, fun with axes, adding images to figures, and `lattice` graphics.

8.3 Visualizing many variables

8.3.1 Scatterplot matrices for many variables

8.3.1.1 Basic `pairs()` plots

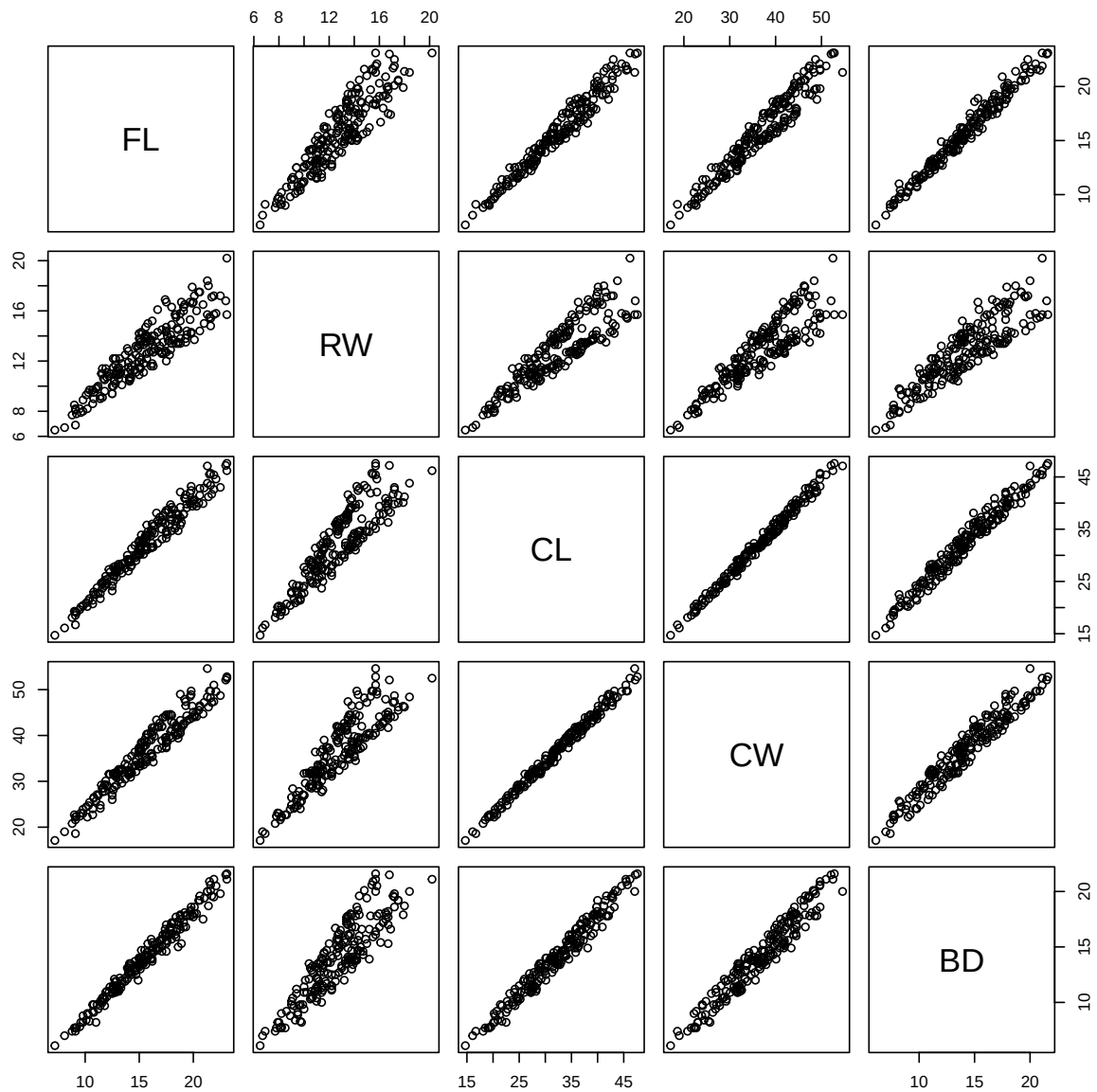
A scatterplot plots one variable against another, and is probably the best way to see the relationship (if any) between two variables. But what if you have many variables? Making dozens of scatterplots can be tedious and time consuming. A scatterplot matrix makes many scatterplots at once, allowing relationships between many variables to be visualized at once. The base function to do this is `pairs()`.

```
# load some data
data(crabs, package="MASS")

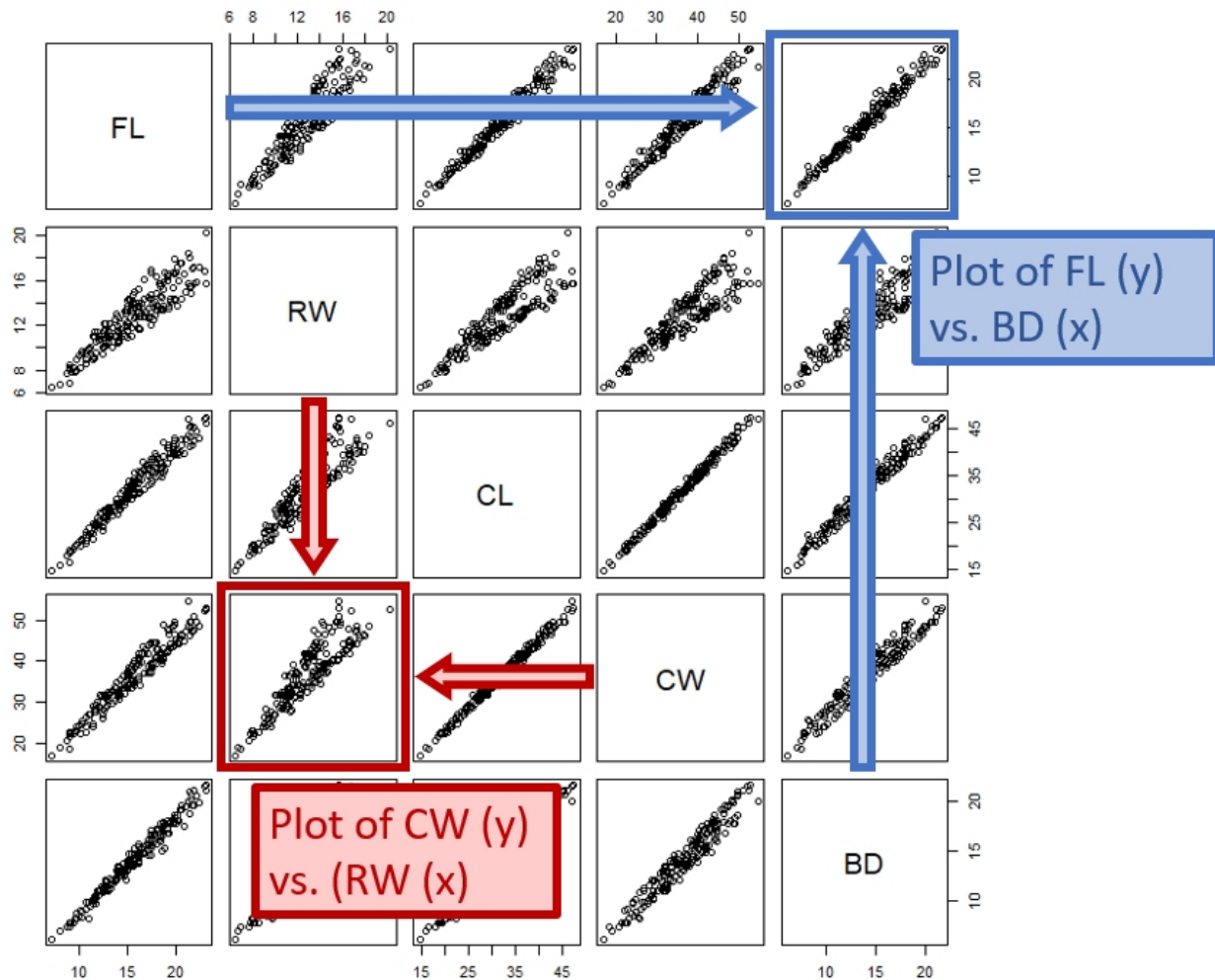
# make a spare copy
x <- crabs

# define columns for scatterplot matrix
dat.cols <- 4:8

# make scatterplot matrix
pairs(x[,dat.cols])
```



In the scatterplot matrix, every variable is plotted against every other variable. Variables are labeled on the diagonal. E.g., the plots in the first row have “FL” as their y-axis, and “RW”, “CL”, “CW”, and “BD” as their x-axes. Likewise, plots in the first column have “FL” as their x-axis, and “RW”, “CL”, “CW”, and “BD” as their y-axes.



{Illustration

of how to read a scatterplot matrix produced by `pairs()`. Each panel plots the variable in its row against the variable in its column.}

8.3.1.2 Improved `pairs()` plots

One common modification to the default `pairs()` plot is to replace scatterplots above the diagonal with correlation coefficients. Another modification is to add linear regression or LOESS⁵ lines to the lower plots to help highlight the relationships. The function below is adapted from the help page for `pairs()`. Notice that the size of the text for each coefficient is scaled to the magnitude of the coefficient. Can you figure out what piece of the code is doing that?

```
# define a function to add correlation coefficients
# borrowed from ?pairs
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  #usr <- par("usr"); on.exit(par(usr))
```

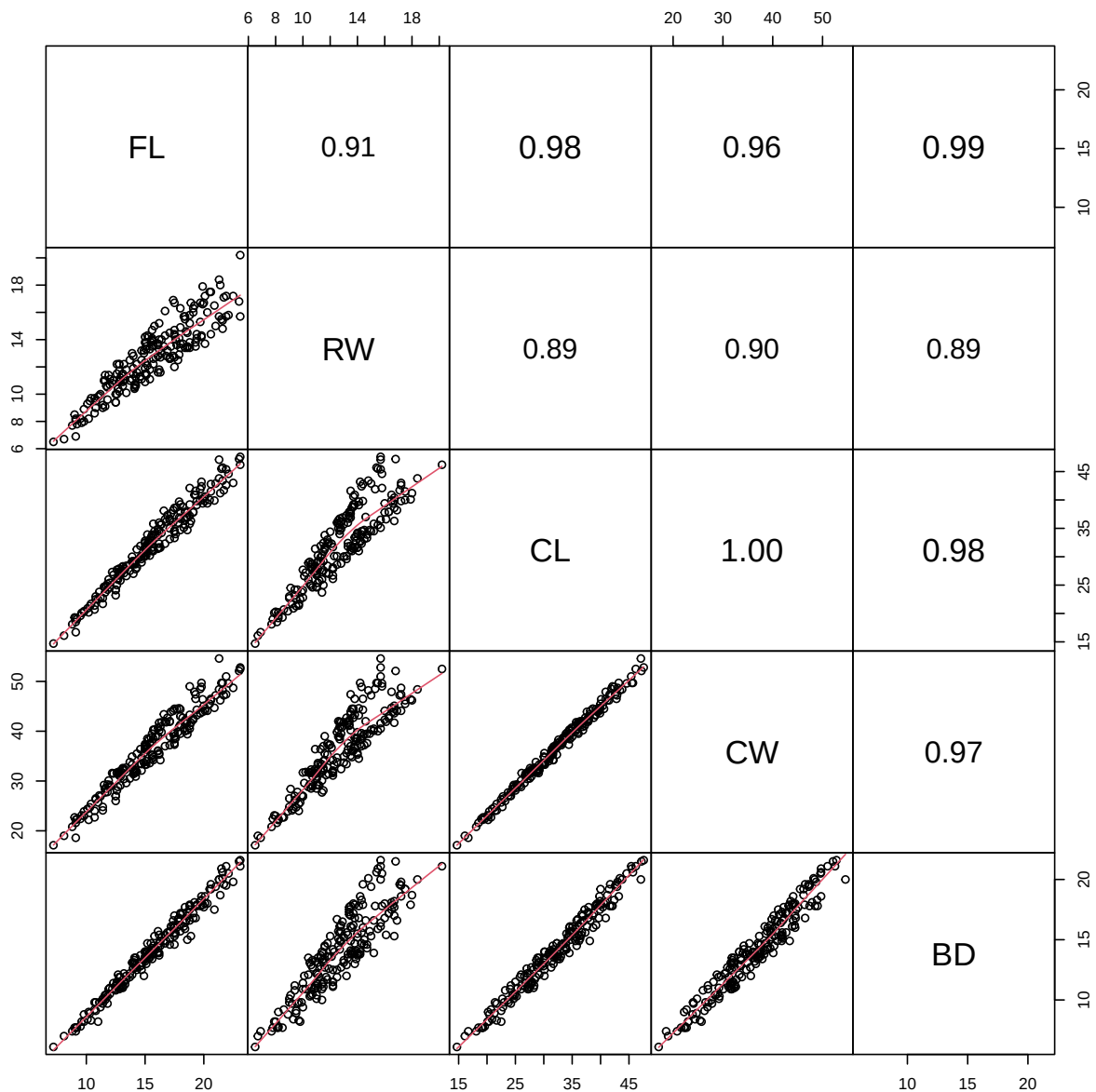
⁵Locally estimated sums of squares, a common smoothing curve algorithm.

```

par(usr = c(0, 1, 0, 1))
r <- cor(x, y)
txt <- format(c(r, 0.123456789), digits = digits)[1]
txt <- paste0(prefix, txt)
if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
text(0.5, 0.5, txt, cex = 2*abs(r))
}

# example of use:
pairs(x[,dat.cols],
      lower.panel=panel.smooth,
      upper.panel=panel.cor, gap=0)

```



Sometimes the Spearman rank correlation coefficient ρ is more informative than the Pearson linear correlation coefficient r (Section 6.4). The function `panel.cor()` can be modified to use ρ instead of r :

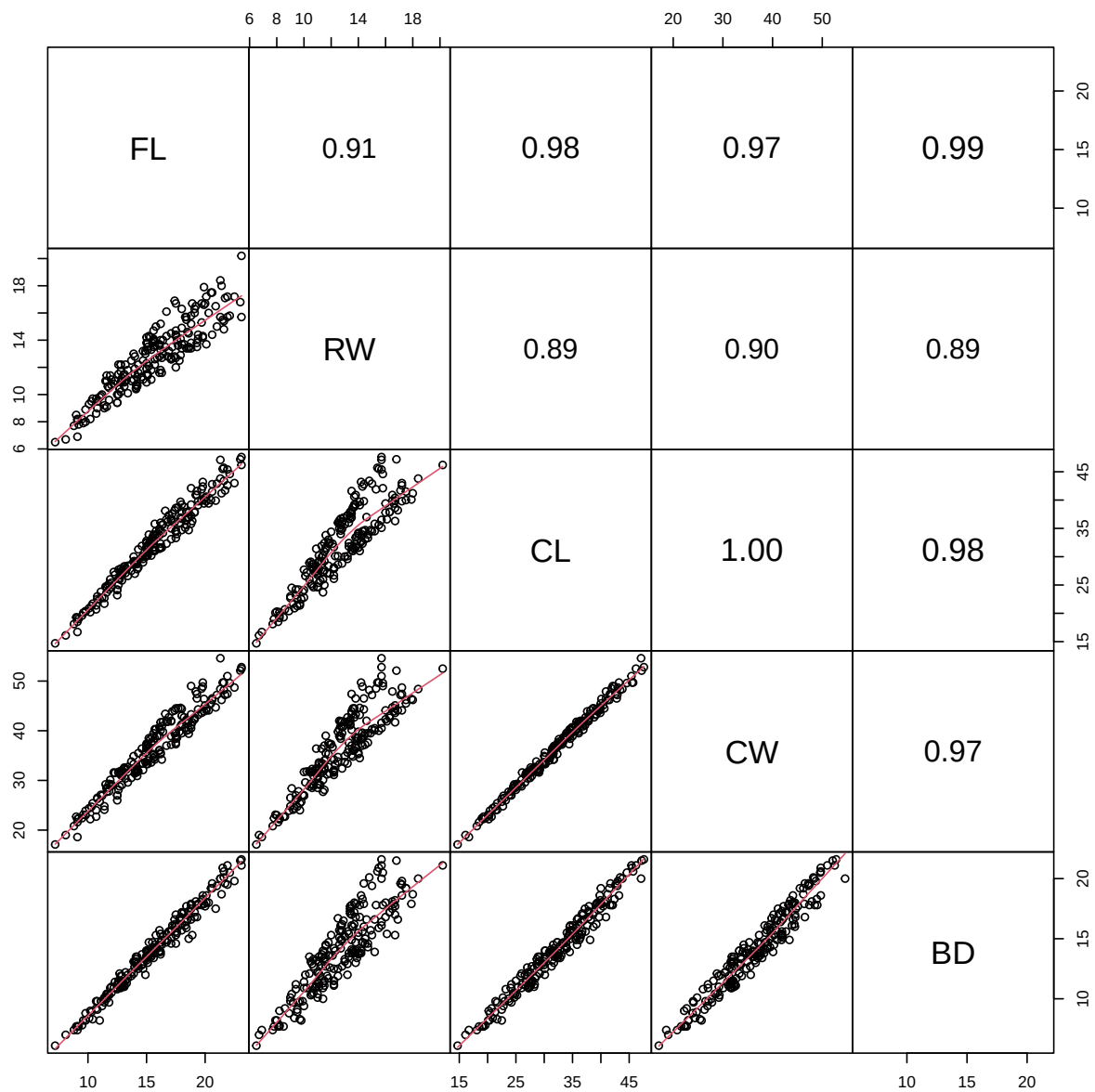
```
panel.cor2 <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  par(usr = c(0, 1, 0, 1))
  r <- cor(x, y, method="spearman") ## changed here!
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = 2*abs(r))
}
```

```

}

# example of use:
# (might return warnings but these are ok)
pairs(x[,dat.cols],
      lower.panel=panel.smooth,
      upper.panel=panel.cor2, gap=0)

```



8.3.2 Ordination (brief introduction)

The methods above are a way to examine single variables, or single pairs of variables. But what about many variables at once? In other words, how can we examine variation between samples by taking into account multiple variables at once? This is the realm of **multivariate statistics**.

Ordination is literally plotting or ordering observations along two or more axes. How that ordering is done varies wildly among techniques. We will get deeper into ordination later in the course (Chapter 27), but for now let's look at one of the most common techniques, **principal components analysis (PCA)**, with a focus on interpretation rather than calculation. Some of the goals of all ordination techniques are:

- **Cluster identification:** observations that are closer to each other in the ordination space are more similar to each other
- **Dimension reduction:** the axes of an ordination are estimates of synthetic variables that combine information about many variables at once

Biologists use many different ordination techniques to analyze patterns in many variables at once. Regardless of the specific technique, the basic interpretation is usually the same:

1. Points that are closer together are more similar to each other.
2. Variables get bigger in the direction of their arrow.

These two rules will work for almost any ordination you are likely to encounter in biology: PCA, redundancy analysis, canonical correspondence analysis, nonmetric multidimensional scaling (my favorite), linear discriminant analysis, and the list goes on.

8.3.2.1 Principal components analysis (PCA)

Principal components analysis (PCA) is a method for extracting synthetic gradients from a multivariate dataset that capture most of the variation in that dataset. These gradients are calculated by finding linear combinations of the variables that minimize sums of squared deviations from the gradient. In other words, PCA translates the variation between observations in terms of the original, many variables, into variation along a few artificial variables called **principal components**.

The basic problem is this: you can see a pattern in one variable with a histogram (Section 8.1.1). You can see a pattern in 2 or 3 variables (i.e., dimensions) with a scatterplot (Section 8.2.1). But what if you have 4 variables? 10 variables? If you've never heard of PCA or ordination, it might be worth watching a video that explains and shows the basic ideas. Here is [one that only takes 5 minutes and has a nice theme song](#) (accessed 2021-08-10). For a more in-depth introduction to PCA, see [Module 24](#). For now, let's just go with "PCA helps us visualize patterns involving many variables at once".

PCA is available in base R using the functions `prcomp()` and `princomp()`. Both of these functions produce similar outputs, but there are some subtle differences between them. For this course we are going to use the PCA function in package `vegan`. Although designed for community ecology, `vegan` is widely

used for ordination and multivariate analysis in many fields. Importantly, `vegan` is actively maintained and updated with new techniques as they are developed. Also importantly, `vegan` offers a common interface for many kinds of ordination.

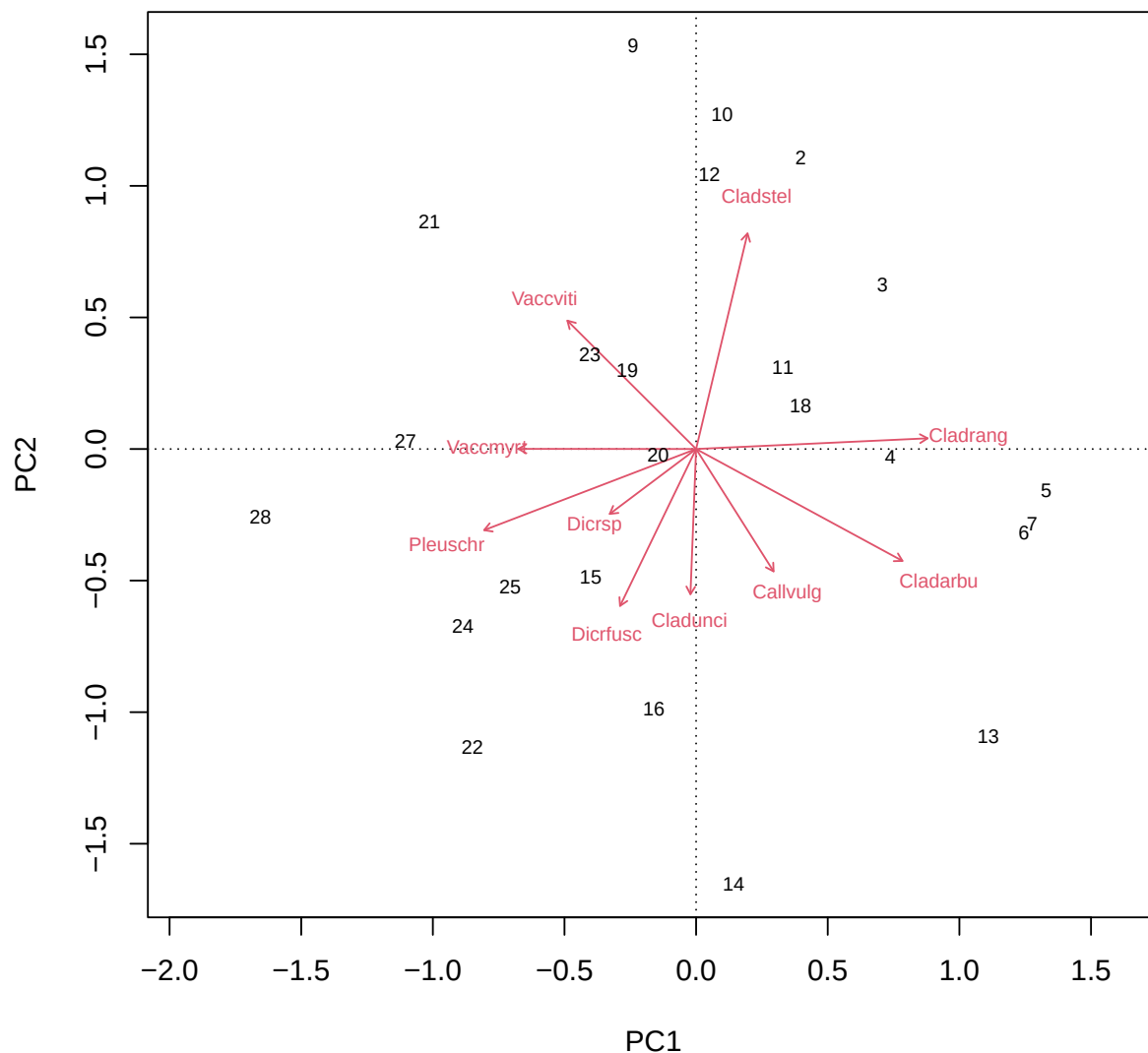
The example below uses the `varespec` dataset from `vegan`. Each column contains % cover by different species of plants in lichen pastures in northern Finland and Scandinavia. For the sake of making a simple example, we will ordinate only the 15 species with the most variance (many species in the dataset are rare or sparse, so including them makes the plot harder to interpret...in “real life”, don’t remove variables from your analysis without good reason!).

```
library(vegan)

# get data and make a spare copy
data(varespec)
x <- varespec

# identify the variables with most variance
vx <- rev(sort(apply(x, 2, var)))
x <- x[,names(vx)[1:10]]

# calculate PCA and plot it
p1 <- pca(x, scale=TRUE)
biplot(p1)
```



The figure above is called a **PCA biplot**. Each point is a sample (site) where vegetation was sampled. By default, the samples are identified by their row number. The red arrows show sites where different species were more abundant (or, more generally, where the variables are increasing). For example, sites towards the top right had greater cover by species “Cladstel”. Sites towards the lower left had less cover by Cladstel. Sites towards the top left had more cover by “Pleuschr”.

The `summary()` command will print a lot of information about the ordination. It prints the eigenvalues of each PC, then scales them as “Proportion explained”. I.e., the proportion of variation explained by each PC is the same as its eigenvalue divided by the sum of all eigenvalues.

```
summary(p1)
```

Call:

```
pca(X = x, scale = TRUE)
```

Partitioning of correlations:

	Inertia	Proportion
Total	10	1
Unconstrained	10	1

Eigenvalues, and their contribution to the correlations

Importance of components:

	PC1	PC2	PC3	PC4	PC5	PC6	PC7
Eigenvalue	2.7808	1.9382	1.3148	1.0987	0.83622	0.7150	0.69134
Proportion Explained	0.2781	0.1938	0.1315	0.1099	0.08362	0.0715	0.06913
Cumulative Proportion	0.2781	0.4719	0.6034	0.7132	0.79687	0.8684	0.93750

	PC8	PC9	PC10
Eigenvalue	0.33428	0.17151	0.11919
Proportion Explained	0.03343	0.01715	0.01192
Cumulative Proportion	0.97093	0.98808	1.00000

The “Importance of components” part tells us that the first principal component (PC1) explains about 27.8% of the variation in the dataset. PC2 explains an additional 19.4%. One rule of thumb people use is to present enough PCs to account for $\geq 80\%$ of the variation. If it takes many PCs to reach 80% of variation, that may be a sign that the PCs are not capturing much meaningful structure in your dataset (or, there simply isn’t much structure to capture).

Finally, we can see the variable **loadings**, which describe the contribution of each original variable to each principal component⁶. Loadings with larger absolute values indicate variables that contribute more strongly to an axis, while the sign indicates the direction of that relationship. In the example below, unscaled loadings for PCs 1-4 are requested using `choices=1:4` and `scaling=0`.

```
scores(p1, choices = 1:4, display = "species", scaling = 0)
```

	PC1	PC2	PC3	PC4
Cladstel	0.1118986	0.5621722123	-0.14606363	0.4467186
Pleuschr	-0.4612677	-0.2115389675	0.17670529	-0.2214946
Cladrang	0.5037497	0.0279351716	0.18537465	-0.3339431

⁶Different software packages define loadings slightly differently. Some report the coefficients (eigenvectors) used to calculate the principal components, while others report the correlations between the original variables and the principal components. The two are closely related and usually lead to the same qualitative interpretation. We’ll discuss this more in Chapter 27.


```

Cladarbu  0.4490762 -0.2918262462  0.08645467 -0.3032787
Dicrfusc -0.1655069 -0.4091013432  0.12191016  0.4270946
Vaccviti -0.2801468  0.3347151730 -0.09280300 -0.1347303
Dicrsp   -0.1879928 -0.1693471498 -0.57673161 -0.2878989
Callvulg  0.1685866 -0.3191415425  0.21724150  0.4741263
Cladunci -0.0123003 -0.3786062372 -0.50705444  0.1445745
Vaccmyrt -0.3866379  0.0003420561  0.49518742 -0.1368990
attr(,"const")
[1] 3.894323

```

The loadings indicate that PC1 is strongly associated with species *Cladrang* and, to a lesser extent, *Pleuschr*. The signs of the loadings indicate whether the species vary in the positive or negative direction along PC1. Compare this to the orientation and length of the corresponding arrow in the biplot. Variables with larger absolute loadings generally have longer arrows and exert a stronger influence on the principal component.

8.3.2.2 Plotting ordinations

Extracting the scores from an ordination object can let you plot points with all of the base graphics formatting methods we learned above (Section 8.2), such as using color or shape to indicate group membership. In my experience this is more flexible than using the built-in graphics capabilities in *vegan*⁷, but your mileage may vary.

```

# load some data
data(crabs, package="MASS")

# make a spare copy
x <- crabs

# fit the PCA on the numerical variables
p1 <- pca(x[,4:8], scale=TRUE)

# extract the "scores": coordinates on PC axes
site.scores <- scores(p1, choices=1:2,
                      display="sites", scaling="sites")

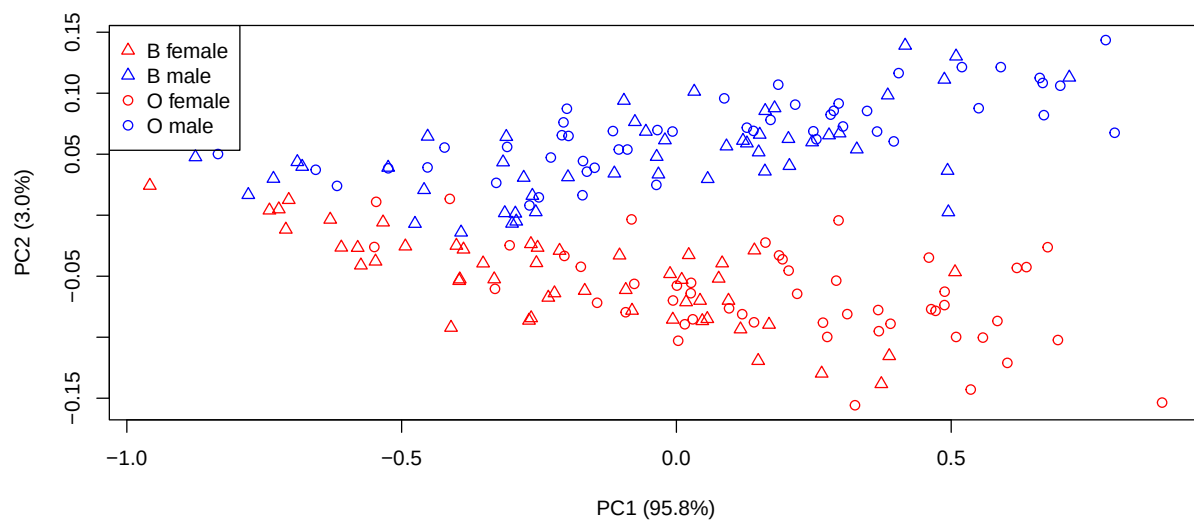
# define some colors for sex
use.colors <- c("red", "blue")
names(use.colors) <- sort(unique(x$sex))
x$color <- use.colors[x$sex]

```

⁷An introduction by the package's lead author can be found by running this command: `vignette(topic="intro-vegan", package="vegan")`.

```
# define some shapes for species
use.pch <- c(2,1)
names(use.pch) <- sort(unique(x$sp))
x$pch <- use.pch[x$sp]

# build the plot
par(mfrow=c(1,1))
plot(site.scores, col=x$color, pch=x$pch,
     xlab="PC1 (95.8%)", ylab="PC2 (3.0%)")
legend("topleft",
     legend=c("B female", "B male", "O female", "O male"),
     pch=c(2, 2, 1, 1),
     col=c("red", "blue", "red", "blue"))
```



Some interpretations of this plot:

1. Almost all of the variance is on PC1. Because the underlying data are measurements, and measurements are often correlated with overall size, PC1 is probably a gradient of overall body size. If we wanted to use PCA to capture shape variation, we would need to factor out overall size first somehow. Or, not interpret PC1.
2. The species B and O (triangles and circles) do not appear to be well distinguished by either PC1 or PC2.
3. The sexes (red and blue) are well separated by PC2, with females having smaller values and males having greater values.

8.3.2.3 Ordination wrap-up (for now)

Ordination is a collection of methods for reducing the dimensionality of data. Biological datasets often have many variables, but most variation in those datasets can often be captured using only a few of them (or few combinations of the variables). The positions of the samples in the ordination space can thus reveal patterns that are too complicated to see in the full data space. Interpreting ordination axes in biological terms can be very fruitful. When an axis captures important information about the samples, it can be used as a variable in another method (e.g., as a predictor variable in a linear model). Analysis of clustering or positional differences between groups in ordination space (especially NMDS space) can allow for researchers to test for differences in many variables at once.

Chapter 9

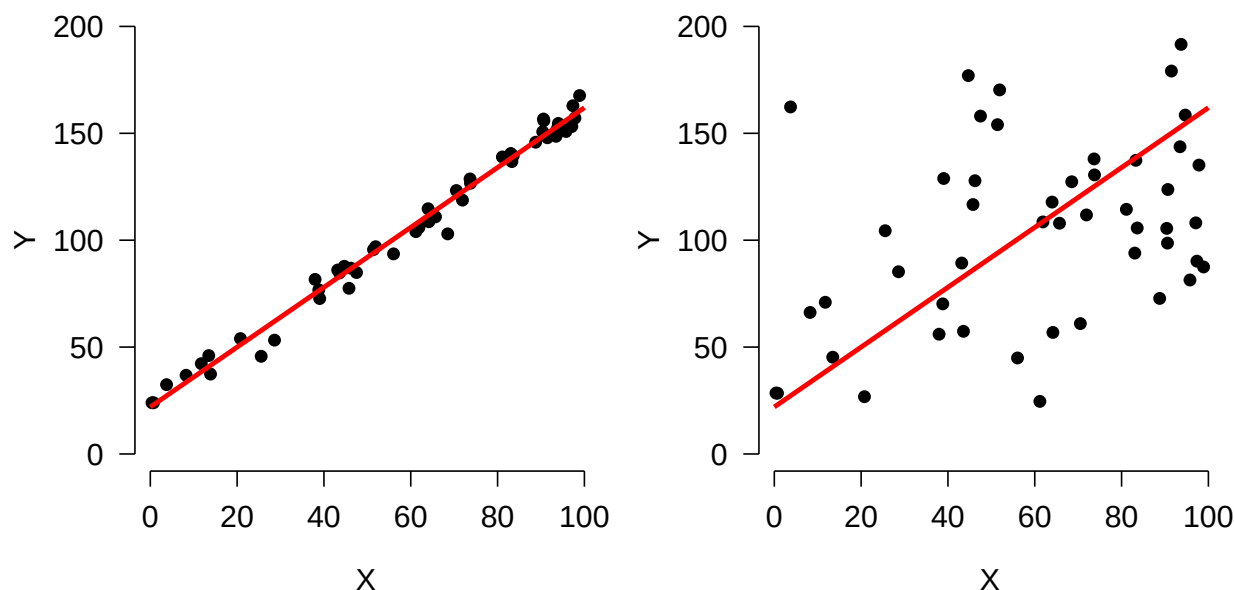
Probability distributions for biologists

9.1 Introduction

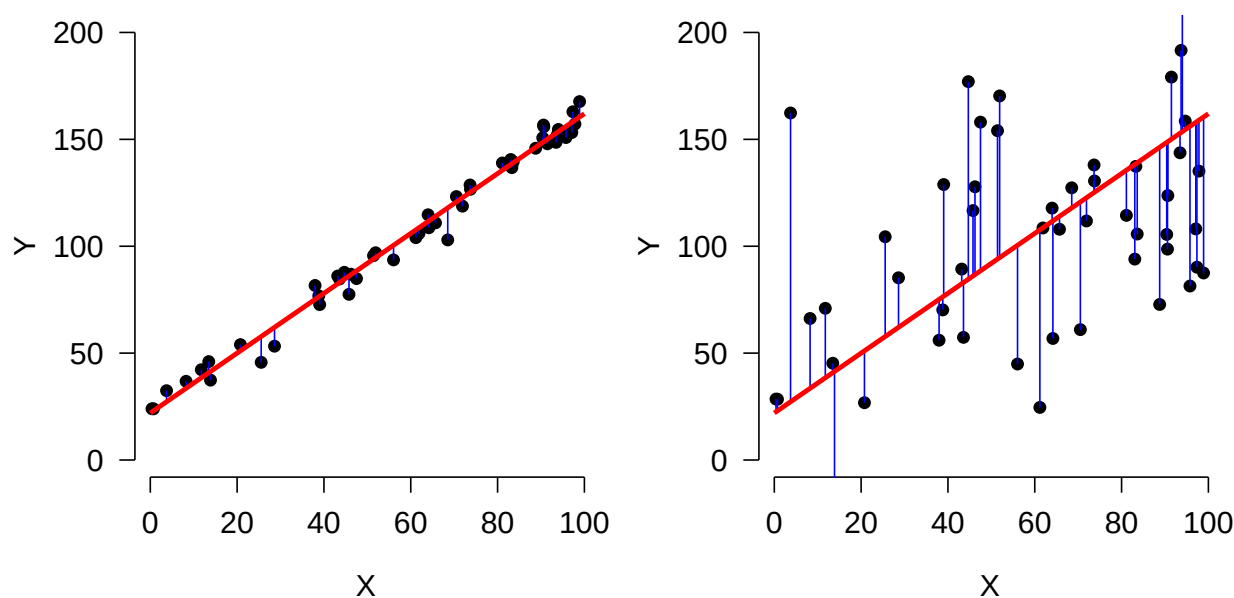
All data contain some element of randomness. Understanding the nature and consequences of that randomness is what motivates much of modern statistics. **Probability distributions** are the mathematical constructs that describe randomness in data. This module describes some probability distributions commonly encountered in biology (and a few that aren't common). The emphasis here is on practical understanding of what the distributions imply about data—not on the theoretical underpinnings or mathematical details. If you want or need a rigorous introduction to probability theory, this is probably not the right place.

9.1.1 Residuals: the random variation left over

Consider the figures below:



Both scatterplots show a linear relationship between X and Y . But what is different about the plot on the right? Variation. Both plots show the same relationship ($Y = 22 + 1.4X$), but they differ in that the variation about the expected value (the red line) is much greater in the right plot than the left plot. Consequently, X appears to explain much more variation in Y in the left plot than in the right plot. The next figure shows that the **residual variation**, or difference between the expected and observed values, is much greater in the right panel. Each of these differences between the observed value of Y (Y_i) and the expected value of Y ($E(Y_i)$), or $Y_i - E(Y_i)$, is called a **residual**.



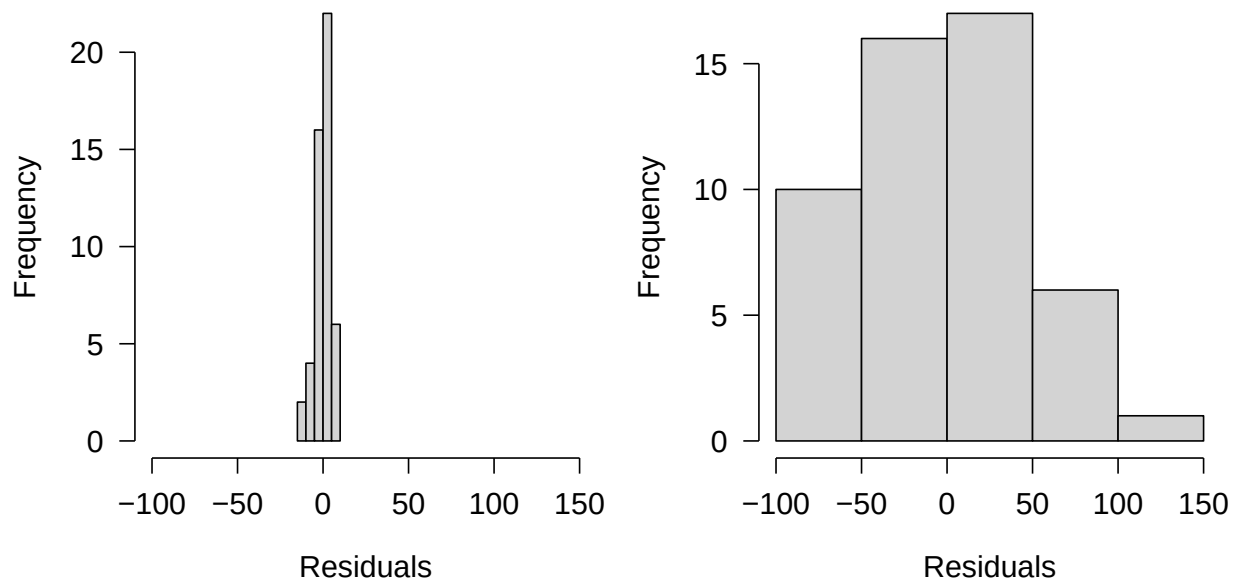
Residuals are an important part of statistical analysis for two reasons. First, most statistical models make assumptions about the distribution of residuals. Second, the total magnitude of the residuals (usually ex-

pressed as sum of squared residuals) is useful for calculating many measures of how well a model fits the data. The residuals of a statistical model are the differences between the observed values and the values predicted by the model. The residual for observation i in variable Y , R_i , is calculated as:

$$R_i = Y_i - E[Y_i]$$

where $E(Y_i)$ is the **expected value** of Y_i . This means that when an observation has a greater value than predicted, the residual is positive; similarly, when an observation is smaller than expected, the residual is negative. In many statistical methods, residuals are squared so that (1) positive and negative residuals do not cancel out; and (2) larger residuals carry more weight.

The figures below show the distributions of residuals for the example linear regressions above. Notice that the left dataset has a much smaller distribution of residuals than the right dataset. This is because of the much tighter fit of the left dataset's Y values to the predicted curve.



Notice also that both distributions of residuals have a similar shape, despite the difference in width. This shape is actually very important: the **normal distribution**. If the residuals were not distributed this way (i.e., did not follow a normal distribution), then we would be in trouble for two reasons. First, the data were generated using a normal distribution for residuals, so something would have had to have gone wrong with our statistical test; and second, the linear regression model used to analyze the data assumes that residuals are normally distributed. If they are not, then the test is not going to produce valid estimates of statistical significance or model parameters.

The example above is an example of the importance of thinking about statistical distributions when analyzing data. So just what is a statistical distribution? Distributions are mathematical functions that define the probabilities of random outcomes. Here **random** means that there is some element of chance in what we observe. This randomness is not completely unpredictable. While the outcome of specific observations might

be unknowable, we can make predictions about long run frequencies or averages of lots of observations. In other words, single observations are not predictable, but the properties of sets of observations are. Such properties might include the number of times a specific outcome occurs (e.g., number of times a flipped coin comes up heads) or some summary of observations (e.g., the mean tail length of chipmunks). Another term for this kind of randomness that follows a pattern is **stochastic**.

Another example: coin flips

Consider flipping a coin. A fair coin will come up heads 50% of the time, and tails the other 50%. If you flip a coin once, the probability of getting heads is 50%. But what about if you flip the coin 10 times? How many heads should you get? 5 is a reasonable guess. The table below shows the possible outcomes to 10 coin flips:

Heads	Tails
0	10
1	9
2	8
3	7
4	6
5	5
6	4
7	3
8	2
9	1
10	0

This table shows that 5 heads is only one of 11 possibilities! So, is the probability of 5 heads 1 in 11 ($1/11 \approx 0.091$)? Of course not, because not all outcomes are equally likely.

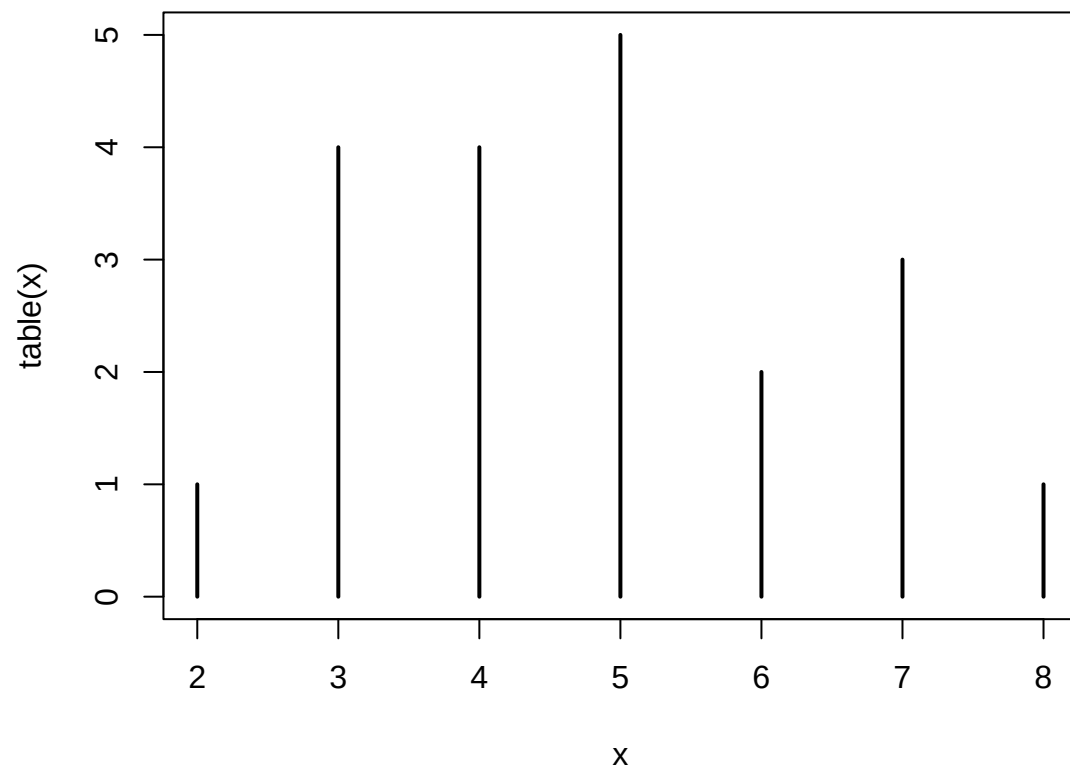
One of the reasons statistics is so fun is that we can often use simulation to discover patterns. The R code below will simulate the effects of flipping 1 coin 10 times. Run this code a few times and see what happens.

```
rbinom(1,10,0.5)
```

```
[1] 4
```

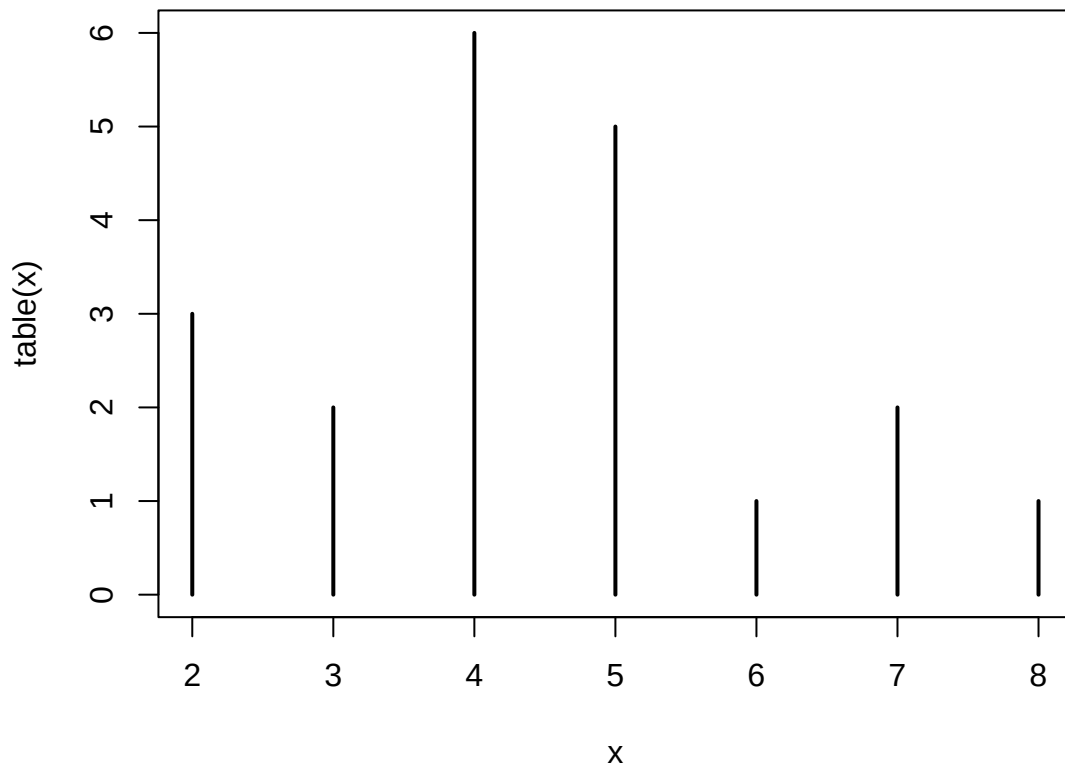
A typical run of results might be something like 4 5 5 4 3 5 6 6 5 5 4 6. We can repeat this line many times and graph the results:

```
x <- numeric(20)
for(i in 1:20){x[i] <- rbinom(1,10,0.5)}
plot(table(x))
```



There is actually a faster way, by requesting 20 draws of 10 flips each all at once:

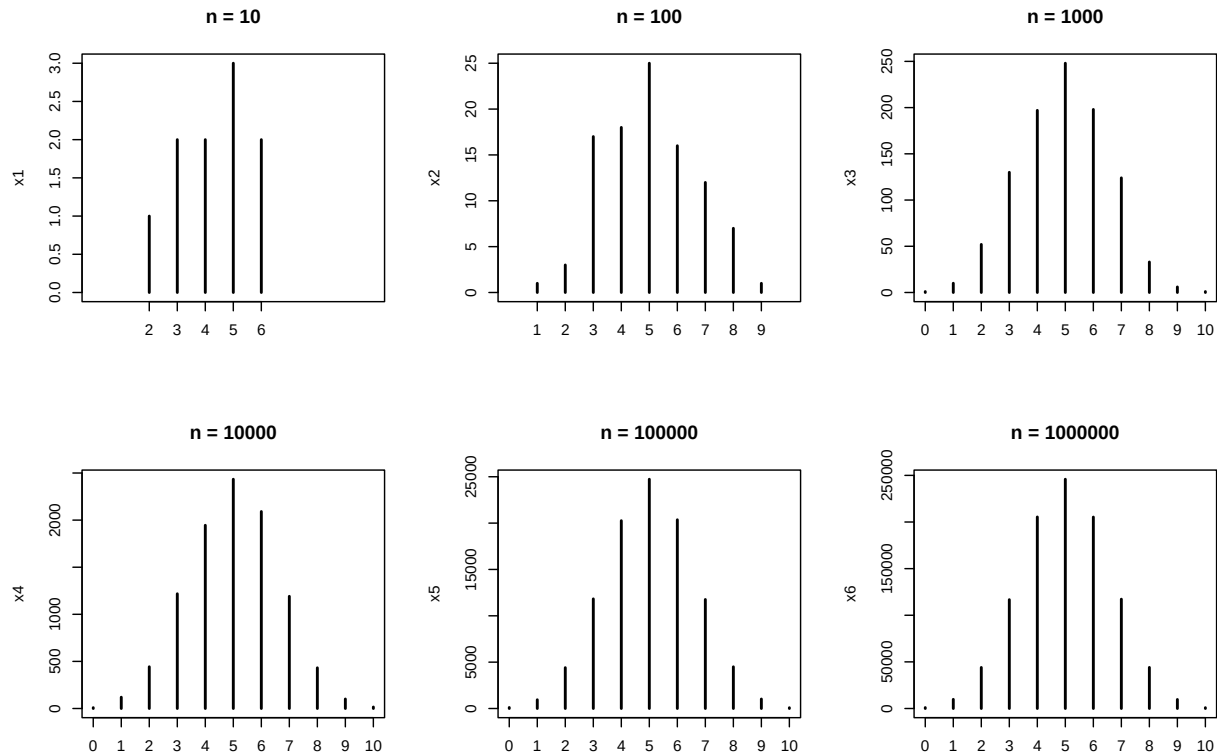
```
x <- rbinom(20,10,0.5)
plot(table(x))
```

Is 5 the most likely value? What about if we get a larger sample?

```
x1 <- table(rbinom(10,10,0.5))
x2 <- table(rbinom(100,10,0.5))
x3 <- table(rbinom(1000,10,0.5))
x4 <- table(rbinom(10000,10,0.5))
x5 <- table(rbinom(100000,10,0.5))
x6 <- table(rbinom(1000000,10,0.5))

par(mfrow=c(2,3))
plot(x1, xlim=c(0, 10), main="n = 10")
plot(x2, xlim=c(0, 10), main="n = 100")
plot(x3, xlim=c(0, 10), main="n = 1000")
plot(x4, xlim=c(0, 10), main="n = 10000")
plot(x5, xlim=c(0, 10), main="n = 100000")
plot(x6, xlim=c(0, 10), main="n = 1000000")
```



```
# reset graphical parameters
par(mfrow=c(1,1))
```

You may have guessed by now that the properties of a set of coin flips—such as what number of heads to expect—are described by some kind of mathematical function. This function is called a **probability distribution**. This particular distribution is the **binomial distribution**, which describes the outcome of any random process with a binary outcome. The name “binomial” is descriptive: “bi” for two, and “nomial” for names or states. Can you think of a biological situation where the binomial distribution might apply?

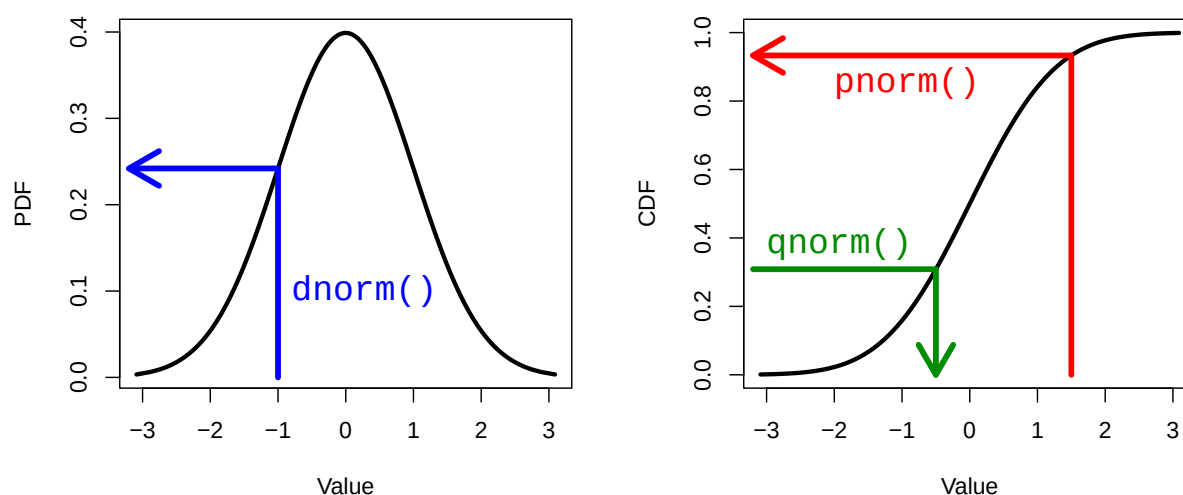
The sections below describe some distributions commonly encountered in biology, and what kinds of processes give rise to them. It is important to be able to relate biological phenomena to statistical distributions, because the underlying nature of the randomness in some process can affect how we analyze that process statistically. Note that in this class we will focus on the practical applications of these distributions, rather than their mathematical derivations.

9.1.2 Probability distributions in R

Every distribution supported in R has four functions associated with it: $r\_()$, $q\_()$, $d\_()$, and $p\_()$, where $\_$ is the name of the distribution (or an abbreviation). These 4 functions calculate different values defined by the distribution:

- `r__()` draws **random numbers** from the distribution.
- `p__()` calculates the **cumulative distribution function (CDF)** at a given value. The reverse of `q__()`.
- `d__()` calculates the **probability density function (PDF)**; i.e., the height of the density curve, or first derivative of the CDF.
- `q__()` calculates the **value at a given quantile**. The reverse of `p__()`.

The figure below shows these functions in relation to the PDF and CDF for a normal distribution.



- If you want to know the value at the Xth percentile of a distribution, use the `q__` function.
- If you want to know the percentile of a value of a distribution, use the `p__` function.
- If you need to know the PDF of a distribution at some value, use the `d__` function.

Here we see the functions used on a normal distribution with mean 20 and SD 4 (i.e., $Normal(\mu = 20, \sigma = 4)$ or $N(20, 4)$).

```
# 0.3 quantile (30th percentile)
qnorm(0.3, 20, 4)
```

```
[1] 17.9024
```

```
# What value is at the 64th percentile? (0.64 quantile)?
pnorm(0.65, 20, 4)
```

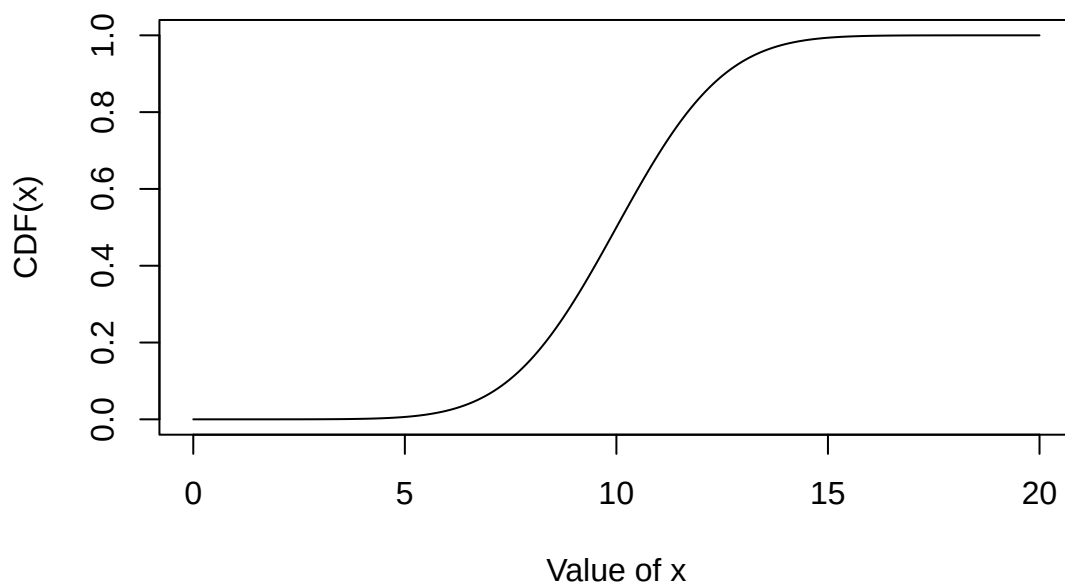
```
[1] 6.574119e-07
```

```
# What is the PDF at x == 23?  
dnorm(23, 20, 4)
```

```
[1] 0.07528436
```

One use case might be drawing a CDF or PDF of a distribution manually to compare to your data or some other distribution. For example, here is how to draw the CDF of a normal with mean 10 and SD 2:

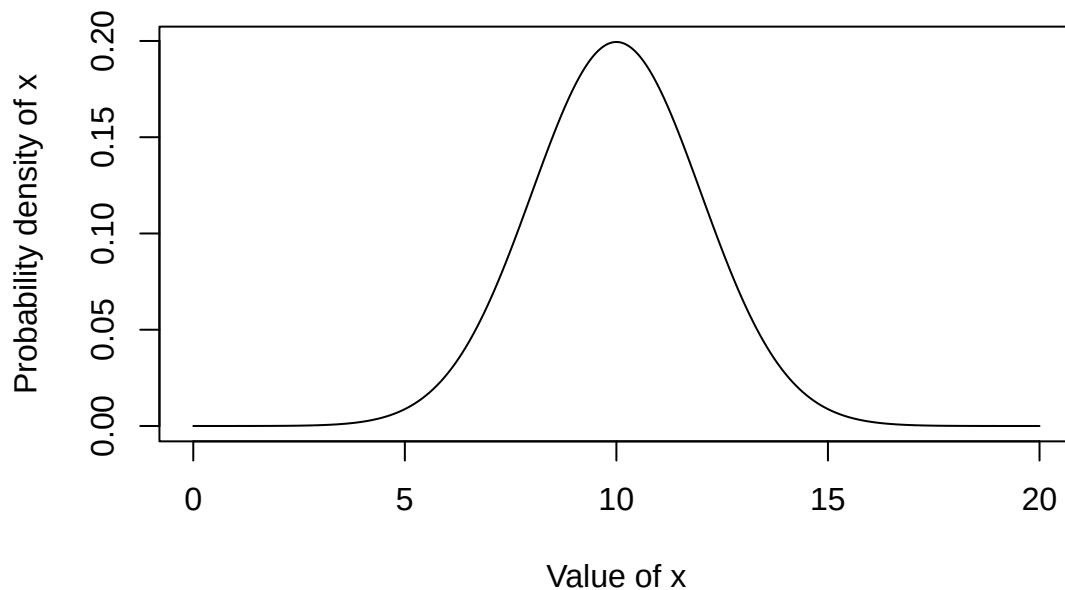
```
# values at which to draw CDF  
xvals <- seq(0, 20, length=1000)  
  
# distribution parameters  
mu <- 10  
sigma <- 2  
  
# calculate CDF  
xcdf <- pnorm(xvals, mu, sigma)  
  
# make the plot  
plot(xvals, xcdf, type="l", xlab="Value of x", ylab="CDF(x)")
```



Here is how to draw the PDF of the same distribution:

```
# calculate CDF
xpdf <- dnorm(xvals, mu, sigma)

# make the plot
plot(xvals, xpdf, type="l", xlab="Value of x",
     ylab="Probability density of x")
```



9.2 Discrete distributions

Discrete distributions can take on integer values only. Because of this, many discrete distributions are related in some way to **count data**. Count data result from, well, counting things. Count data can also describe the number of times something happened.

9.2.1 Bernoulli distribution

The simplest discrete distribution is the **Bernoulli distribution**. It describes the outcome of a single random event with probability p . Thus, the Bernoulli distribution takes the value 1 with probability p or the value 0 with probability $q = (1 - p)$. Any opportunity for the event to happen is also called a **Bernoulli trial**, and the process that it describes a **Bernoulli process**. The probability p can take on any value in the closed interval $[0, 1]$.

For convenience we usually consider a Bernoulli distribution to take the value of 0 or 1, but really it could represent any binary outcome. For example, *yes* vs. *no*, *dead* vs. *alive*, < 3 vs. ≥ 3 , etc., are all outcomes that could be modeled as Bernoulli variables. By convention, the event that occurs with probability p is considered a **success** and has the numerical value 1; the event that occurs with probability $1 - p$ is considered a **failure** and takes the numerical value 0. This is how Bernoulli variables are represented in most statistical packages, including R.

The Bernoulli distribution is rarely used on its own in an analysis. Instead, it's useful to think of the Bernoulli distribution as a special case of, or a building block of, more complicated distributions such as the **binomial distribution** (Section 9.2.2). For example, a single observation of a binomially-distributed variable could be thought of as a Bernoulli distribution.

The Bernoulli distribution is a special case of the binomial distribution, and so it is accessed in R using the functions associated with the binomial. With the `size` argument set to 1, the binomial distribution is the Bernoulli distribution.

```
# flip one fair coin  
rbinom(1, 1, 0.5)
```

```
[1] 0
```

```
# flip 10 fair coins  
rbinom(10, 1, 0.5)
```

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

```
# flip 10 coins with a 70% chance of heads  
rbinom(10, 1, 0.7)
```

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

9.2.2 Binomial distribution

The binomial distribution describes the **number of successes** in a set of independent Bernoulli trials. Each trial has probability of success p , and there are n trials. The values n and p are the **parameters** of the binomial distribution—the values that describe its behavior. The coin flipping example above is an example of a binomial distribution. Biological examples of binomial processes might be the number of fish that die in an experiment, or the number of plants that flower in a season. In such a case, the number of fish that start the experiment is the sample size n and the individual mortality rate is p .

Because it is so simple, thinking about the binomial distribution is a good warm up for learning about the characteristics of probability distributions. One of the most important characteristics is the **expected value** of the distribution. This is the value that most likely to occur, or the **central tendency** of values. There are

several kinds of expected value, but they are all related to the most common or central value. The expected value, or mean, of a binomial distribution X is

$$E(x) = \mu = np$$

This is the answer to the question earlier about how many heads to expect if a fair coin is flipped 10 times:

$$E(heads) = n(flips)p(heads)$$

If a variable comes from a binomial process, we can make other inferences about it. For example, we can estimate its variance as:

$$Var(x) = \sigma^2 = np(1 - p) = npq$$

We can also estimate the probability of any number of successes k as:

$$P(k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

The first term (n over k) is known as the **binomial coefficient** and is calculated as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This term represents the number of ways of seeing k successes in n trials. For example, a set of 4 trials could have 2 successes in 6 ways: HHTT, HTHT, THHT, HTTH, THTH, and TTTH.

When n is small, the binomial distribution can be quite skewed (i.e., asymmetric) because the distribution has a hard lower bound of 0. Note that the expression for $P(k)$ is what is calculated by function `dbinom()` below, and related to what is calculated by function `pbinom()`. For large n , the binomial distribution can be approximated by a normal distribution¹ with mean = np and variance = npq .

9.2.2.1 Binomial distribution in R

R uses a family of 4 functions to work with each probability distribution. The binomial and Bernoulli distributions are accessed using the `_binom` group of functions: `dbinom()`, `pbinom()`, `qbinom()`, and `rbinom()`. Each function calculates or returns something different about the binomial distribution:

- `dbinom()`: Calculates probability mass function at x for binomial distribution given n and p . Answers the question “What is the probability of x successes in n trials with probability p ?”

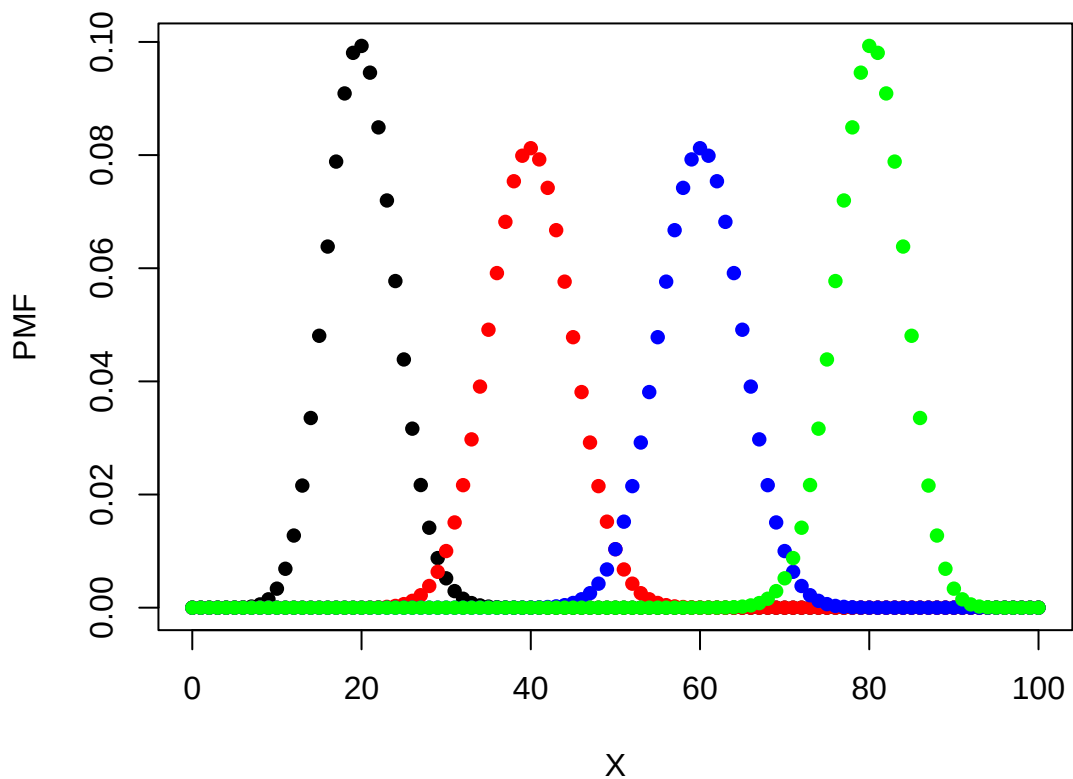
¹Although the values must be rounded to non-negative integers!

- `pbinom()`: Calculates integral of the probability mass function for a binomial distribution given n and p , from 0 up to x . In other words, given some binomial distribution, at what quantile of that distribution should some value fall? The reverse of `qbinom()`. Answers the question “What is the probably of at least x successes in n trials with probability p ?”
- `qbinom()`: Calculates the value at specified quantile of a binomial distribution. Essentially the reverse of `pbinom()`.
- `rbinom()`: Draws random numbers from the binomial distribution defined by n and p (or from the Bernoulli distribution if $n = 1$).

Let's explore the binomial distribution using these functions. In the plots produced in the two examples, notice how the variance of each distribution (x_1 , x_2 , etc.) depends on both n and p . The variance in these plots is shown by the width of the distribution.

```
# N = 100, P various
N <- 100
X <- 0:100
x1 <- dbinom(X, N, 0.2)
x2 <- dbinom(X, N, 0.4)
x3 <- dbinom(X, N, 0.6)
x4 <- dbinom(X, N, 0.8)

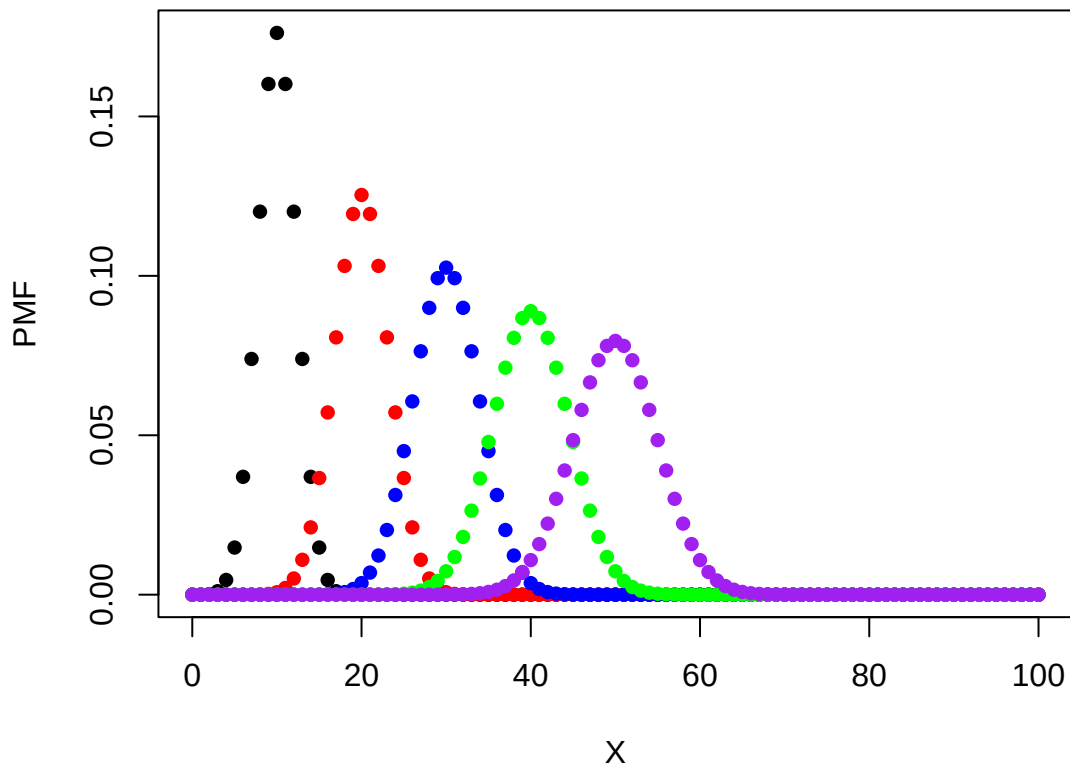
par(mfrow=c(1,1))
plot(X, x1, pch=16, xlab="X", ylab="PMF")
points(X, x2, pch=16, col="red")
points(X, x3, pch=16, col="blue")
points(X, x4, pch=16, col="green")
```

Here is a plot showing the effect of varying n .

```
# P = 0.5, N various
P <- 0.5
x1 <- dbinom(0:100, 20, P)
x2 <- dbinom(0:100, 40, P)
x3 <- dbinom(0:100, 60, P)
x4 <- dbinom(0:100, 80, P)
x5 <- dbinom(0:100, 100, P)

plot(0:100, x1, pch=16, xlab="X", ylab="PMF")
points(0:100, x2, pch=16, col="red")
points(0:100, x3, pch=16, col="blue")
points(0:100, x4, pch=16, col="green")
points(0:100, x5, pch=16, col="purple")
```

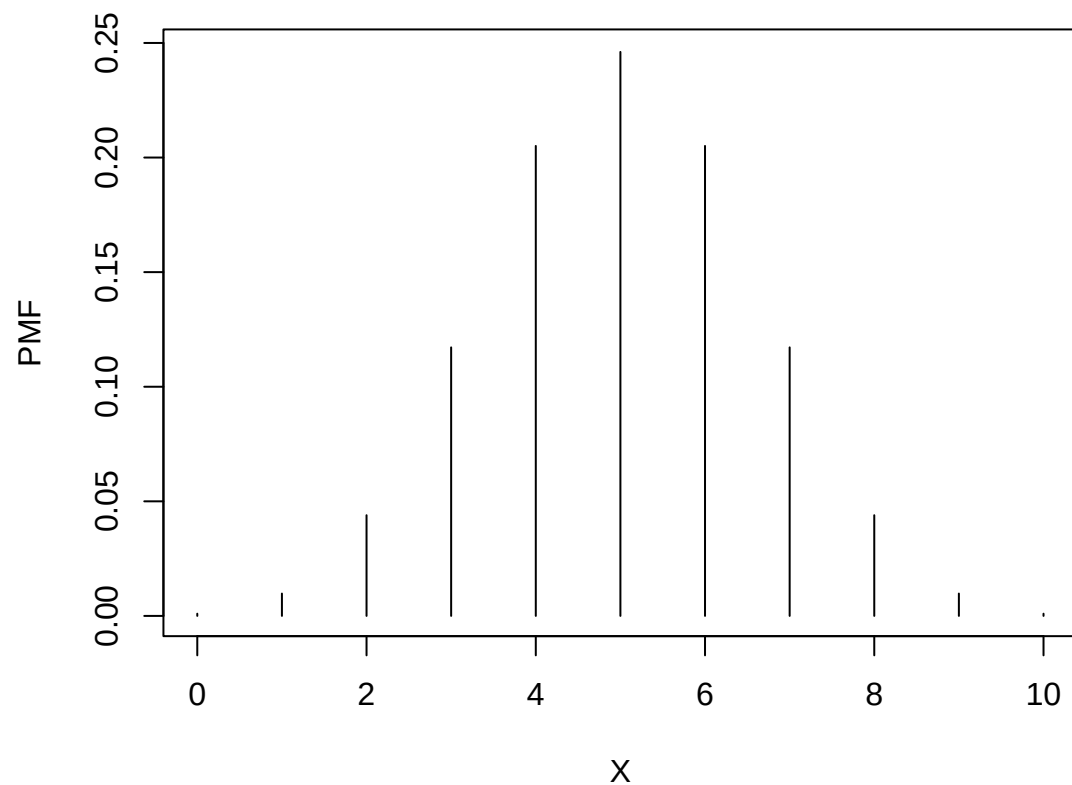


In these plots, the height of the points is called the **probability mass function (PMF)**. For discrete distributions like the binomial, the PMF of any value is the probability that the distribution takes on that value. The sum of the PMF for all integers from 0 to n (inclusive) must be equal to 1. You can verify this by calculating the sum of any of the vectors of densities above.

The function `pbinom()` sums the PMF from 0 up to and including some value. In other words, it calculates the **cumulative distribution function (CDF)**. This answers the question “where in the distribution is value x ?”; put another way, “at what quantile of the distribution does value x lie?”. Yet another way to ask this is, “What is the probability of at least X successes?”.

Consider a binomial distribution with $n = 10$ and $p = 0.5$. What is the probability that the distribution takes a value ≤ 7 ? We can calculate this as the sum of the PMF for 0 through 7.

```
# plot the distribution to see it
N <- 10
P <- 0.5
xd <- dbinom(0:10, N, P)
plot(0:10, xd, type="h", xlab="X", ylab="PMF")
```



```
# calculate  $p(x \leq 7)$ 
sum(xd[1:8]) # indices 1:8 correspond to values 0:7
```

```
[1] 0.9453125
```

```
# same value:
pbinom(7, N, P)
```

```
[1] 0.9453125
```

Or, we could want to know the probability that the distribution takes on a value > 6 . This would be the complement of the sum up to and including 6.

```
1 - sum(xd[1:7])
```

```
[1] 0.171875
```

```
# same value:  
1-pbinom(6,N,P)
```

```
[1] 0.171875
```

If the last two calculations seem familiar, that's because this is exactly how p -values for statistical tests are calculated (e.g., using `pf()` to get the probability from an F distribution for an ANOVA).

The function `qbinom()` is basically the reverse of function `pbinom()`. Rather than calculate the quantile at which a value falls in the distribution, `qbinom()` calculates the value at which a quantile falls. For example, what value do we expect to find at the 60th percentile of a distribution? Put another way, how many successes should 60% of experiments have, on average? The example below shows how `qbinom()` and `pbinom()` are reversible.

```
N <- 10  
P <- 0.5  
pbinom(6, N, P)
```

```
[1] 0.828125
```

```
# same value  
qbinom(0.828125, N, P)
```

```
[1] 6
```

Finally, `rbinom()` draws random values from the binomial or Bernoulli distributions. The syntax of this function can be a little confusing. The first argument, `n`, is the *number of random draws* that you want. The second argument, `size`, is the *number of values in each draw*; that is, the parameter n of the binomial distribution. Compare these results, all with $p = 0.5$:

```
# 1 draw of 1 trial  
rbinom(1, 1, 0.5)
```

```
[1] 0
```

```
# 1 draw of 10 trials  
rbinom(1, 10, 0.5)
```

```
[1] 8
```

```
# 10 draws of 1 trial per draw  
rbinom(10, 1, 0.5)
```

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

```
# 10 draws of 10 trials per draw  
rbinom(10, 10, 0.5)
```

```
[1] 3 5 5 7 3 5 5 5 4 3
```

- Result 1 shows a Bernoulli distribution with $n = 1$.
- Result 2 shows a single value from a binomial distribution with $n = 10$.
- Result 3 shows 10 results from Bernoulli distributions. Notice that if you add up the values in Result 3, you get a result like Result 2.
- Finally, Result 4 shows 10 draws from a binomial distribution, which itself has $n = 10$.

The take home message is that the first argument to `rbinom()` is not a parameter of the binomial distribution. It is instead the number of draws from the distribution that you want. The `size` parameter is the argument that helps define the distribution.

9.2.3 Poisson distribution

The **Poisson distribution** is widely used in biology to model **count data**. If your data result from some sort of count or abundance per time interval, spatial extent, or unit of effort, then the Poisson distribution should be one of the first things to try in the analysis. The Poisson distribution has one parameter, λ ("lambda"), which represents the **expected number** of objects counted in a sample (objects being trees, fish, cells, mutations, kangaroos, etc.). This parameter is also the **variance** of the distribution. This is a very important property, as we will see **later**: the expected value and the variance of the Poisson distribution are both λ :

$$E(X) = Var(X) = \lambda$$

Like the binomial distribution, the Poisson distribution is discrete, meaning that it can only take on integer values. Unlike the binomial distribution, which is bounded by 0 and n , the Poisson distribution is bounded by 0 and $+\infty$. However, values $\gg \lambda$ are highly improbable. If there is a well-defined upper bound for your count, then you might consider them to come from a **binomial model** instead of a **Poisson model**. For example, if you are counting the number of fish in a toxicity trial that survive, the greatest possible count is the number of fish in the trial. This means that your "count" is really another way of representing a **proportion**. On the other hand, if your counts have no *a priori* upper bound, then use the Poisson.

9.2.3.1 Poisson distribution in R

The Poisson distribution is accessed using the `_pois` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dpois()`: Calculates **probability mass function (PMF)** at x for Poisson distribution given λ . Answers the question, “what is the probability of observing a count of x given λ ?”
- `ppois()`: Calculates CDF, or integral of PMF, from 0 up to x given λ . In other words, given some Poisson distribution, at what quantile of that distribution should some value fall? The reverse of `qpois()`.
- `qpois()`: Calculates the value at specified quantile of a Poisson distribution. The reverse of `ppois()`.
- `rpois()`: Draws random numbers from the Poisson distribution defined by λ .

Something important to keep in mind about the Poisson distribution is that it only makes sense for discrete counts, not for continuous measurements that are rounded. For example, if you are measuring leaf lengths and round every length to the nearest mm, it might be tempting to use a Poisson distribution to analyze the data because all of the values are integers. But that would be incorrect, because leaf lengths could theoretically take on any positive value. Furthermore, treating rounded continuous values as discrete leads to the awkward issue that changing measurement units can change dimensionless statistics.

For example, the **coefficient of variation (CV)** is the ratio of a distribution's SD to its mean. Thus, it is unitless and should be independent of units. The CV of a Poisson distribution X is:

$$CV(X) = \frac{\sqrt{\lambda}}{\lambda}$$

So, if you measured 20 leaves and found a mean length of 17 cm, the CV would thus be ≈ 0.242 or 24%. But if you convert the measurements to mm, the CV would be about 0.077, or 7.7%!

The figure below shows the effect of varying λ on a Poisson distribution.

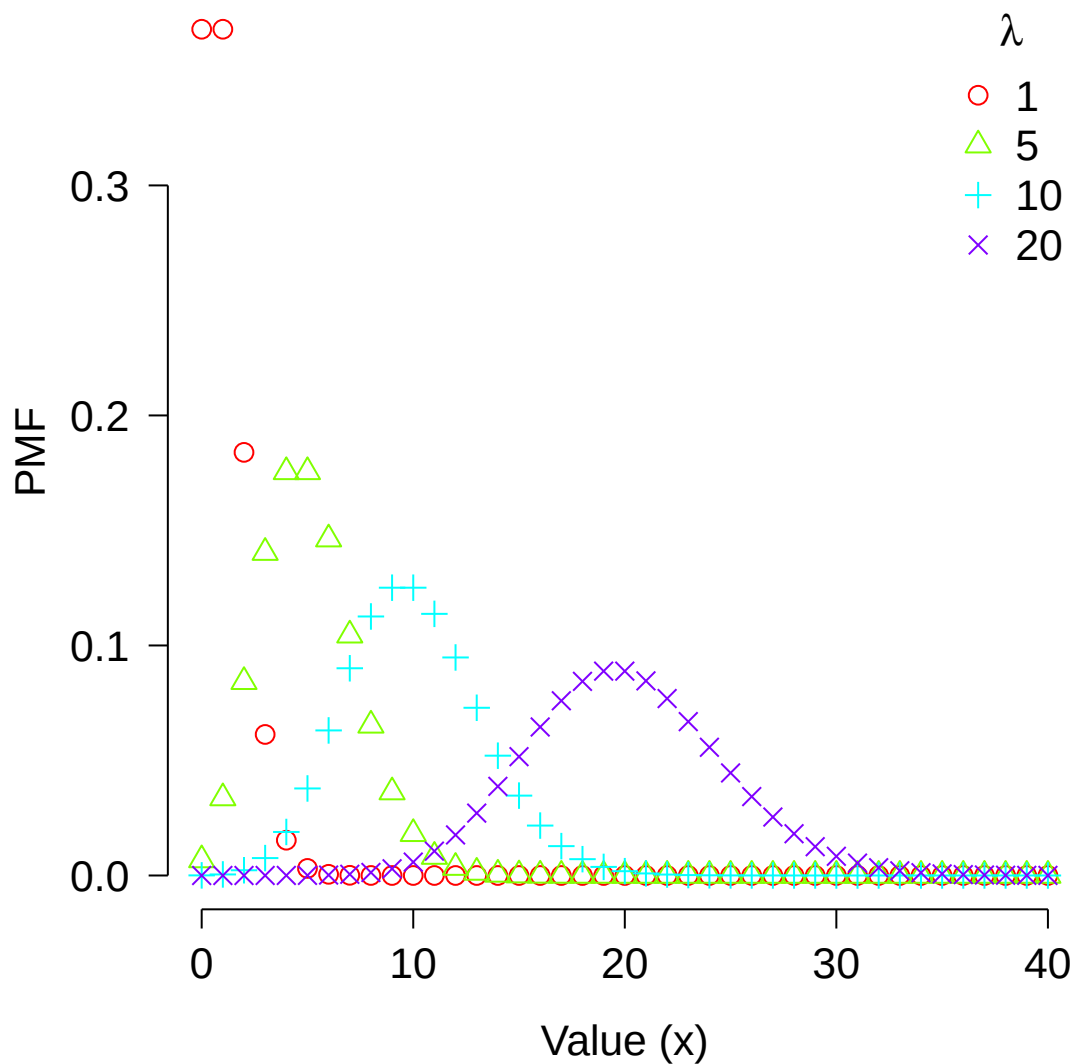
```
x <- 0:40
lams <- c(1, 5, 10, 20)
nlams <- length(lams)
dlist <- vector("list", nlams)
for(i in 1:nlams){dlist[[i]] <- dpois(x, lams[i])}
cols <- rainbow(nlams)

par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, bty="n", lend=1,
    cex.lab=1.3, cex.axis=1.3)
plot(x, dlist[[1]], ylim=c(0, max(sapply(dlist, max))),
     type="n", ylab="PMF", xlab="Value (x)")
```

```

for(i in 1:nlams){
  points(x, dlist[[i]], pch=i, cex=1.3,
        col=cols[i])
}
legend("topright", legend=lams,
      pch=1:nlams, pt.cex=1.3, col=cols, bty="n", cex=1.3,
      title=expression(lambda))

```



9.2.4 Negative binomial distribution

The **negative binomial distribution** as usually used in biology is used for counts that are **overdispersed**—that is, they have a variance greater than expected given other parameters. In this case, that means $\sigma^2 >>$

λ . This is *very common* in biological count data. For example, a bird species might have low abundance (0 to 4) at most sites in a study, but very high abundance (30 to 40) at a handful of sites. In that case using the Poisson distribution would not be appropriate because doing so would mean assuming that the variance was approximately equal to the mean when it really wasn't.

The definition that is not used as commonly in biology is actually the original definition: the negative binomial distribution also describes the number of failures that occur in a series of Bernoulli trials until some predetermined number of successes is observed. For example, when flipping a fair coin, how many times should you expect to see tails before you observe 8 heads? This definition is, therefore, a discrete version of the **gamma distribution**. This also makes the original definition of the negative binomial similar to the geometric distribution (Section 9.2.5).

The version of the negative binomial that biologists use is parameterized by its mean μ and its overdispersion k . This definition views the negative binomial as a Poisson distribution with parameter λ , where λ itself is a random variable that follows a Gamma distribution (Section 9.3.4). For this reason, some authors refer to the negative binomial as a **Gamma-Poisson mixture distribution**. A mixture distribution is exactly what it sounds like: a distribution that is formed by “mixing” or combining two distributions. Often a mixture is formed by having parameters of one distribution vary according to another distribution.

The overdispersion parameter k is called `size` in the R functions that work with the negative binomial. Counterintuitively, the overdispersion in a negative binomial distribution gets larger as k becomes smaller. This is seen in the expression for the variance of a negative binomial distribution:

$$Var(X) = \mu + \frac{\mu^2}{k}$$

As k becomes large, the ratio μ^2/k becomes small and approaches 0, and thus $Var(x)$ approaches μ . This means that a negative binomial distribution with large k approximates a Poisson distribution. As k approaches 0, the ratio μ^2/k becomes larger, and thus $Var(x)$ increases to be much larger than μ .

9.2.4.1 Negative binomial distribution in R

The negative binomial distribution is accessed using the `_nbinom` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

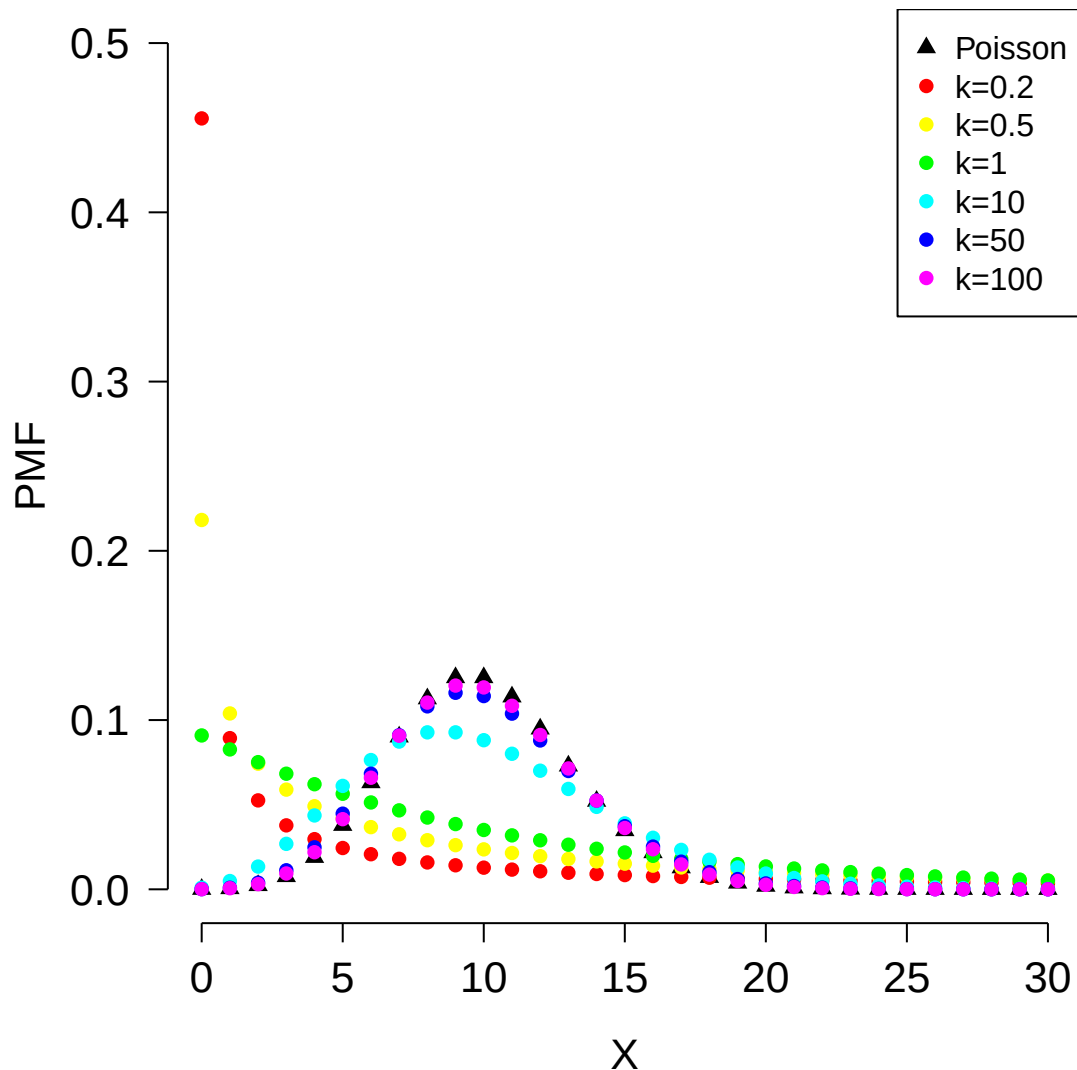
- `dnbinom()`: Calculates PMF at x . Answers the question, “what is the probability of observing a count of x ?”
- `pnbinom()`: Calculates CDF, or integral of PMF, from 0 up to x . In other words, given some negative binomial distribution, at what quantile of that distribution should some value fall? The reverse of `qnbinom()`.
- `qnbinom()`: Calculates the value at specified quantile of a Poisson distribution. The reverse of `pnbinom()`.
- `rnbinom()`: Draws random numbers from the negative binomial distribution.

The R functions for the negative binomial distribution can work with either parameterization (waiting time or mean with overdispersion). Some of the argument names are used for both methods. If you are working with the negative binomial distribution in R you *need to name your arguments to make sure you get the version of the negative binomial that you want*.

The figure below shows the effect of different overdispersion parameters. Notice that as k increases, the distribution looks more and more like a Poisson distribution with $\lambda = 10$. As k gets smaller, the distribution gets more and more concentrated near 0, and more and more right-skewed.

```
xp <- 0:30
y1 <- dpois(xp, 10)
y2 <- dnbinom(xp, size=0.2, mu=10)
y3 <- dnbinom(xp, size=0.5, mu=10)
y4 <- dnbinom(xp, size=1, mu=10)
y5 <- dnbinom(xp, size=10, mu=10)
y6 <- dnbinom(xp, size=50, mu=10)
y7 <- dnbinom(xp, size=100, mu=10)

cols <- rainbow(6)
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, bty="n", lend=1,
    cex.lab=1.3, cex.axis=1.3)
plot(xp, y1, pch=17, ylim=c(0, 0.5), xlab="X", ylab="PMF")
points(xp, y2, pch=16, col=cols[1])
points(xp, y3, pch=16, col=cols[2])
points(xp, y4, pch=16, col=cols[3])
points(xp, y5, pch=16, col=cols[4])
points(xp, y6, pch=16, col=cols[5])
points(xp, y7, pch=16, col=cols[6])
legend("topright",
legend=c("Poisson", "k=0.2", "k=0.5", "k=1", "k=10",
"k=50", "k=100"),
    pch=c(17, rep(16,6)), col=c("black", cols))
```



9.2.5 Geometric distribution

The **geometric distribution** describes the number of failures that are seen before observing the first success, given that the probability remains constant. It is thus related to the **binomial** and **negative binomial**. In fact, the geometric distribution is a special case of the negative binomial, with the number of successes = 1. The geometric distribution is the discrete version of the **exponential distribution**.

The geometric distribution arises in biology when modeling waiting times or life spans in discrete time steps. For example, if an animal has a constant annual mortality rate, the geometric distribution models the number of years that an individual organism completes before dying. Age- or stage-based population matrix models such as Leslie matrices are common in wildlife and fisheries management, and are based on annual monitoring data. So, it makes sense to use a discrete distribution to model survival in yearly time steps rather than continuous time. Another biological example of when to use a geometric distribution would be counting

the number of generations until a mutation in a cell line occurs, if mutations occur at a constant rate per generation.

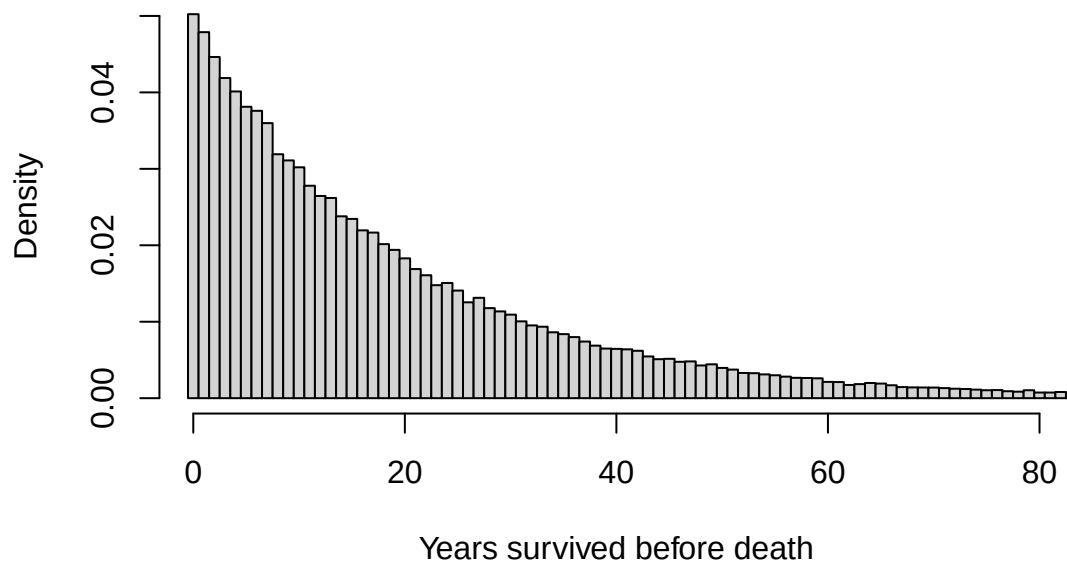
Imagine a fish that has a 5% probability of dying each year (i.e., a 95% annual survival rate). If X is the number of complete years the fish survives before dying, then X follows a geometric distribution with $p = 0.05$. Let's simulate a cohort of 100,000 fish to see what this looks like.

```
# set annual mortality rate
P <- 0.05

# simulate 100000 fish
fish <- rgeom(1e5, P)

# plot the resulting lifespans
use.breaks <- seq(-0.5, max(fish)+0.5, by=1)
hist(fish, breaks=use.breaks, freq=FALSE, xlim=c(0, 80),
     xlab="Years survived before death")
```

Histogram of fish



Notice that the distribution of simulated fish lifespans has a long right tail: there are many small values, and a few very large values. We also call this **right skewed**. Practically, this means that while the median lifespan is 13 years, the average fish lives to 19. The long tail pulls the arithmetic mean upwards.

```
# by what age do 50% of fish die?  
median(fish)
```

```
[1] 13
```

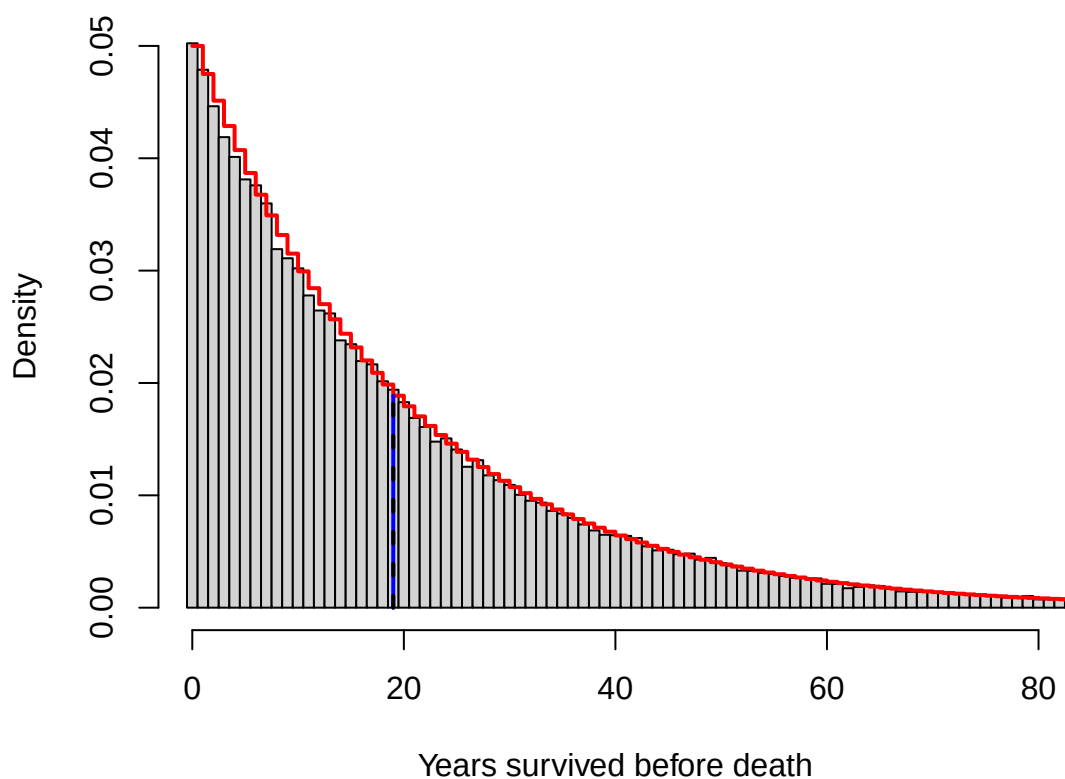
```
# how long does the "average" fish live?  
mean(fish)
```

```
[1] 19.07688
```

Compare our simulation to the theoretical distribution:

```
X <- 0:max(fish)  
  
pmf <- dgeom(X, P)  
hist(fish, breaks=use.breaks, freq=FALSE, xlim=c(0, 80),  
     xlab="Years survived before death",  
     ylim=c(0, 0.05),  
     main="Simulated vs. theoretical distribution")  
points(X, pmf, type="s", lwd=2, col="red")  
meanfish <- round(mean(fish), 0)  
segments(meanfish, 0, meanfish, dgeom(meanfish, P),  
         lwd=2, col="blue")  
# calculate theoretical mean - matches simulation  
theomean <- (1-P)/P  
segments(theomean, 0, theomean, dgeom(theomean, P),  
         lwd=2, lty=2, col="black")
```

Simulated vs. theoretical distribution



The figure shows that the probability of living to any given age falls off quickly at first, then more gradually as time goes on. We can also answer questions like “What is the probability that a fish lives to at least 30 years old?”

```
# answer based on simulation
# off a bit because of sampling variation
length(which(fish >= 30)) / length(fish)
```

```
[1] 0.21554
```

```
# answer based on distribution:
pgeom(29, P, lower.tail=FALSE)
```

```
[1] 0.2146388
```

Or equivalently:

```
(1-P) ^30
```

```
[1] 0.2146388
```

What age will 95% of fish reach?

```
qgeom(0.95, P)
```

```
[1] 58
```

9.2.6 Multinomial distribution

The **multinomial distribution** is a generalization of the binomial distribution to cases with more than 2 possible outcomes. For example, rolling a 6-sided die many times would yield a distribution of outcomes (1, 2, 3, 4, 5, or 6) described by the multinomial distribution. The multinomial distribution is parameterized by the number of possible outcomes k , the number of trials n , and the probabilities of each outcome $p_1, \dots, p_k$. The probabilities must sum to 1.

There are some special cases of the multinomial distribution:

- When $k = 2$ and $n = 1$, the multinomial distribution reduces to the Bernoulli distribution
- When $k = 2$ and $n > 1$, the multinomial distribution reduces to the binomial distribution
- When $k > 2$ and $n = 1$, the multinomial distribution is the categorical distribution.

The example below shows how to work with the multinomial distribution in R.

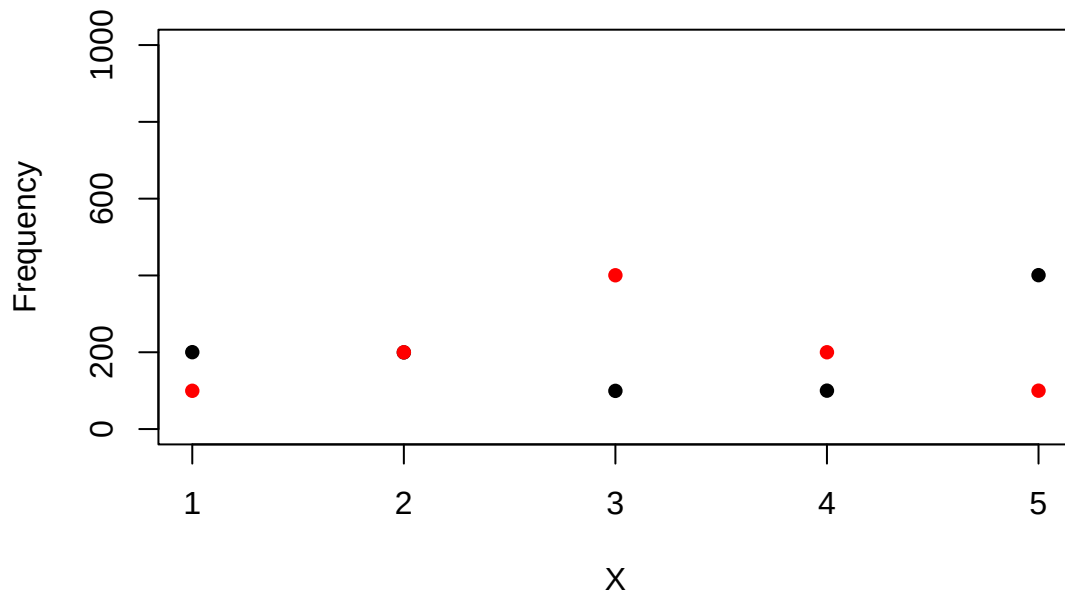
```
# define 2 different sets of probabilities for
# 5 different outcomes
p1 <- c(0.2, 0.2, 0.1, 0.1, 0.4)
p2 <- c(0.1, 0.2, 0.4, 0.2, 0.1)

# sample from the multinomial distribution
N <- 1e3
r1 <- rmultinom(N, N, prob=p1)
r2 <- rmultinom(N, N, prob=p2)

# summarize by outcome
y1 <- apply(r1, 1, mean)
y2 <- apply(r2, 1, mean)

# plot the results
```

```
plot(1:5, y1, pch=16, xlab="X",
     ylab="Frequency", ylim=c(0, 1000))
points(1:5, y2, pch=16, col="red")
```



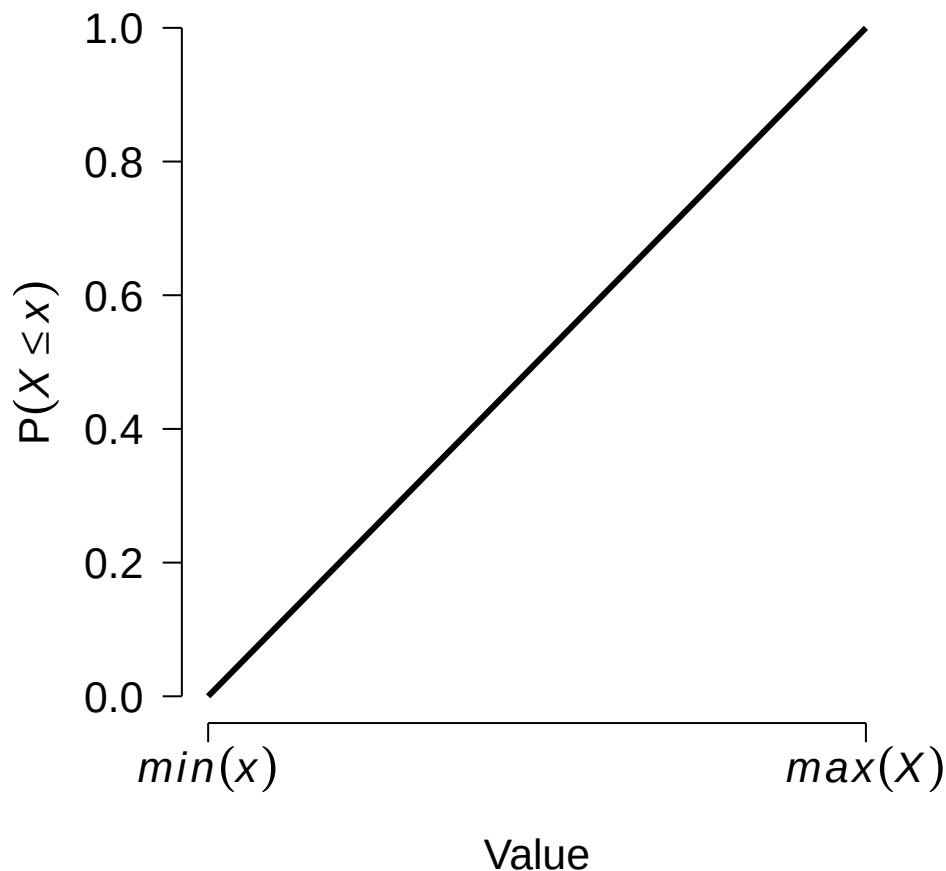
9.3 Continuous distributions

A **continuous distribution** can take on any real value within its supported interval. This means that unlike discrete distributions, continuous distributions are not restricted to integers. Measurements like length, mass, temperature, etc., are best modeled using continuous distributions because, although we often record them to the nearest mm, gram, or degree, the underlying quantity varies continuously. The apparent discreteness comes from the limited precision of our measuring instruments.

Computing the probability of any possible outcome of a discrete distribution is relative straightforward because there are a limited number of outcomes. For example, a binomial distribution with $n = 10$ and $p = 0.5$ has only 11 possible outcomes $(0, 1, 2, \dots, 9, 10)$. The probabilities $p(x = 0)$, $p(x = 1)$, and so on up to $p(x = 10)$ must all sum to 1. The probability mass function (PMF) of the binomial distribution at any value is the probability of that value occurring.

This leads to an interesting question. How could we calculate the probability of any given outcome of a distribution with an infinite number of outcomes? If the probabilities of each individual value must all sum to 1, and there are infinitely many possible values, then each individual value must have probability 0. If the probability of every possible outcome is 0, how can the distribution take on any value at all?

To resolve this apparent contradiction, we have to take a step back and think about what distributions really mean. The first step is to realize that even if any single value has probability 0, an *interval* of values can have a non-zero probability. After all, the entire supported interval of a distribution has probability 1 by definition. We can define some interval extending from the lower limit of the distribution up to some value x . This means that for any value x in a distribution X (note lower case vs. upper case) we can calculate the probability of observing a value $\leq x$, or $P(X \leq x)$. The figure below shows this:



The y-axis in the figure, $P(X \leq x)$, is called the **cumulative distribution function (CDF)**. This name derives from the fact that the CDF of some value x gives the cumulative or total probability of all values up to and including x . The CDF is sometimes labeled $F(x)$ (note the capitalized F). Another way to interpret this is that a value x lies at the $CDF(x)$ quantile of a distribution. For example, if $CDF(x) = 0.4$, then x lies at the 0.4 quantile or 40th percentile of the distribution. Critically, this means that 40% of the total probability of the distribution lies at or to the left of x , and the remaining 60% of the probability lies to the right of x .

Just as the PMF describes how probability is distributed among the possible values of a discrete random variable, the **probability density function PDF** describes how probability is distributed over the continuum of possible values. Although the PDF is not itself a probability, the area under the PDF between any two

values equals the probability that the random variable falls within that interval. Mathematically, the PDF is the rate at which the CDF changes with respect to x . That is, the PDF, $f(x)$, is the derivative of the CDF $F(x)$ evaluated at x :

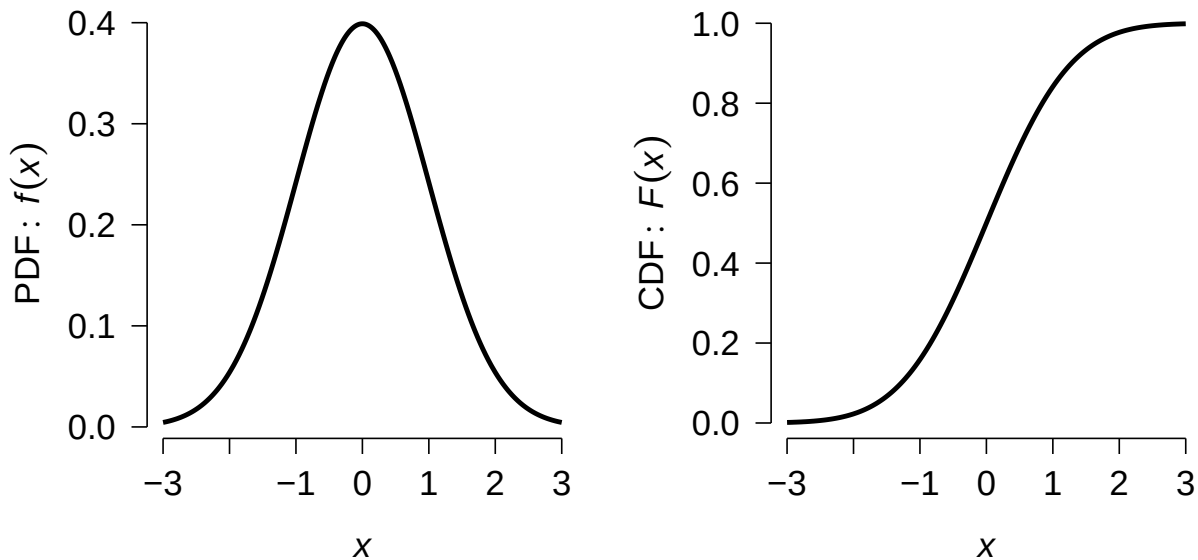
$$f(x) = \frac{dF(x)}{dx}$$

This also means that the CDF is the integral of the PDF up to x , according to the fundamental theorem of calculus:

$$F(x) = \int_{-\infty}^x f(x) dx$$

In other words, probabilities correspond to areas under the PDF, while the PDF itself represents the local density of probability. These areas under the PDF correspond to values of the CDF.

This might be easier to see with an example. The plots below show the PDF and CDF of a standard normal distribution with mean 0 and SD 1 (Section 9.3.2). Because the normal is unbounded, its PDF and CDF have the domain $[-\infty, +\infty]$, but we usually truncate the plot to a range that covers most of the CDF. For the normal, the mean plus or minus 3 standard deviations, or $\mu \pm 3\sigma$, covers over 99.7% of values.



On the left, notice that the PDF peaks near $x = 0$ and tapers off quickly to either side. On the right, the slope of the CDF is greatest at $x = 0$. The value of the PDF at $x = 0$, about 0.4, is neither the probability of observing $x = 0$ (which is 0) nor the probability of observing $x \leq 0$ (which is 0.5). The value $f(0) = 0.4$ is the rate of change or slope in $F(X)$ at $x = 0$.

If we wanted to calculate something like the probability of $x = 0$, we would actually have to calculate the probability of x being in some interval that included 0. For example, the probability that $-0.1 \leq X \leq +0.1$,

or that $X \in [-1, +1]$, is

$$P(-0.1 \leq X \leq +0.1) = F(0.1) - F(-0.1)$$

In R we can calculate:

```
pnorm(0.1)-pnorm(-0.1)
```

```
[1] 0.07965567
```

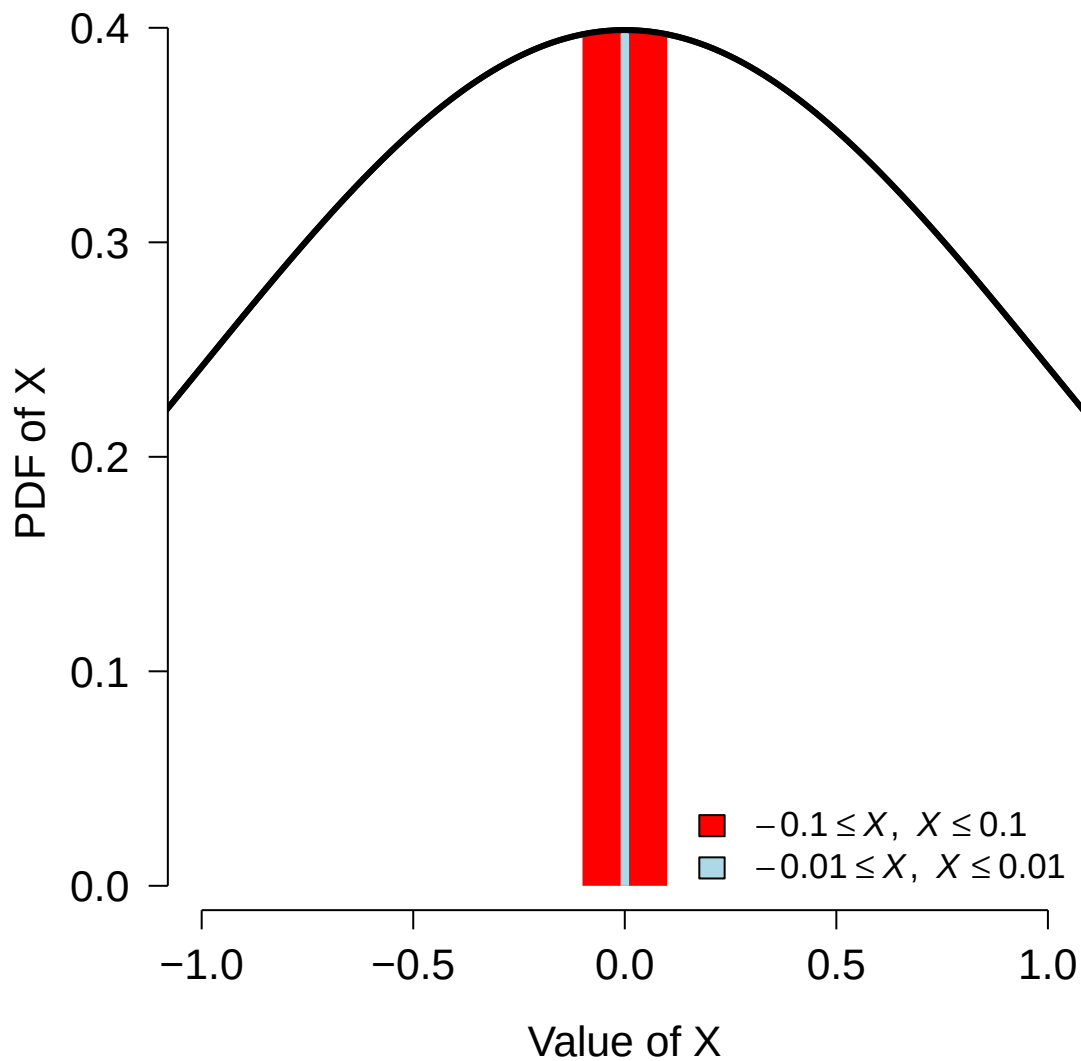
So the probability that x is within 0.1 of 0 is about 0.08 or 8%. If we use a narrow interval, say $-0.01 \leq X \leq +0.01$, the calculation becomes:

```
pnorm(0.01)-pnorm(-0.01)
```

```
[1] 0.007978713
```

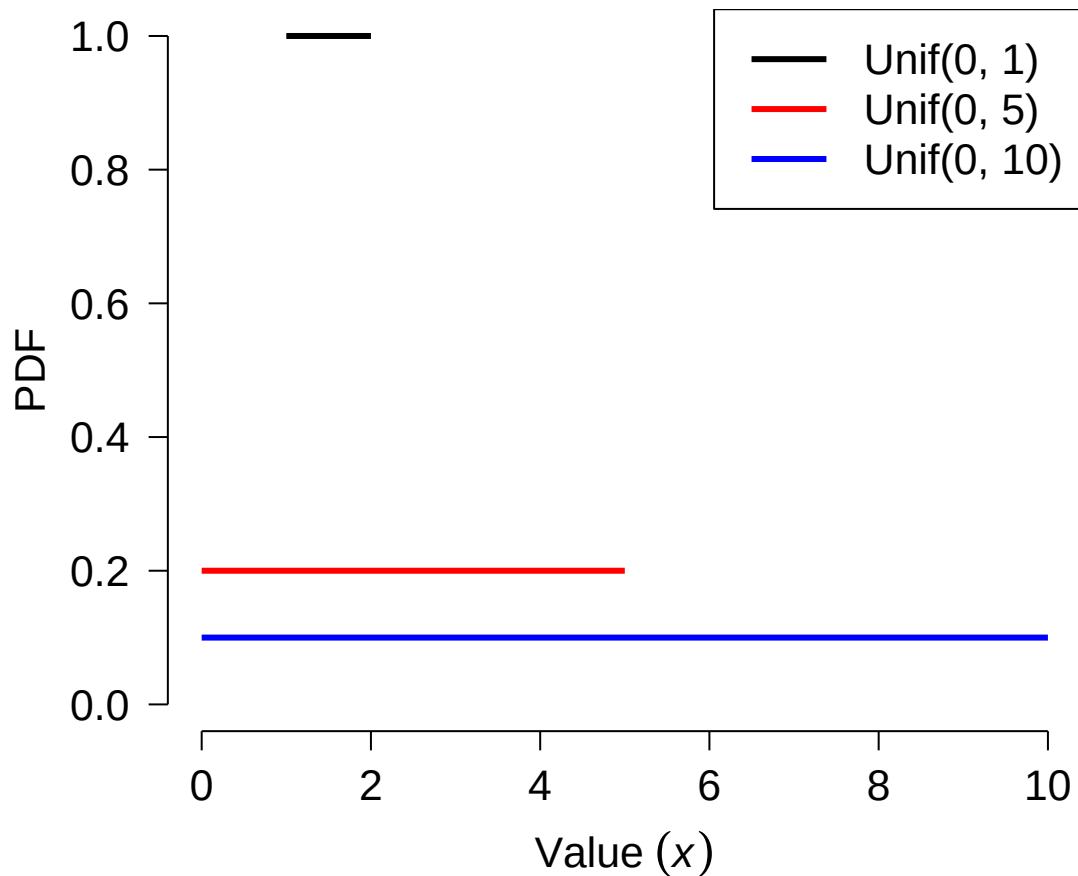
Notice that as the interval becomes narrower, its probability becomes smaller. In the limit as the interval shrinks to a single point, the probability becomes 0. This is why the probability of observing any exact value in a continuous distribution is zero, even though the probability of observing values within any finite interval is positive.

Visually, we can see this in the figure below. Notice how the red area is much larger than the light blue area—these are the intervals defined by ± 0.1 and ± 0.01 .



9.3.1 Uniform distribution

The simplest continuous distribution is the **uniform distribution**. The uniform distribution is parameterized by its upper and lower bounds, usually called a and b , respectively. All values of a uniform distribution in the interval $[a, b]$ are equally likely, and all values outside $[a, b]$ have probability 0. The figure below shows the PDFs of 3 different uniform distributions.



Notice how each distribution has a flat PDF, but that the value of the PDF decreases as the width of the uniform distribution increases. Given what you learned above about the relationship between the CDF and the PDF, can you work out why this is?²

Like the normal distribution, there is a standard uniform distribution that is frequently used. The **standard uniform** is defined as $Uniform(0, 1)$: a uniform distribution in the interval $[0, 1]$. The uniform distribution can sometimes be abbreviated as $U(a, b)$ or $Unif(a, b)$.

The mean of a uniform distribution is just the mean of its limits:

$$\mu(Uniform(a, b)) = \frac{a + b}{2}$$

The variance of a uniform distribution is:

$$\sigma^2(Uniform(a, b)) = \frac{(b - a)^2}{12}$$

²The area under the PDF must be 1, so the height of the PDF of a uniform distribution must be 1 divided by its width.

9.3.1.1 Uniform distribution in R

The uniform distribution is accessed using the `_unif` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dunif()`: Calculates probability density function (PDF) at `x`.
- `punif()`: Calculates CDF, from `a` up to `x`. Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qunif()`.
- `qunif()`: Calculates the value at a specified quantile or quantiles. The reverse of `punif()`.
- `runif()`: Draws random numbers from the uniform distribution.

The help files for many R functions include a call to `runif()` to generate example values from the uniform distribution. This can be confusing to new users, who might interpret “runif” as “run if” (i.e., some sort of conditional statement).

One extremely useful application is the generation of values from the standard uniform distribution:

```
x <- runif(20)
```

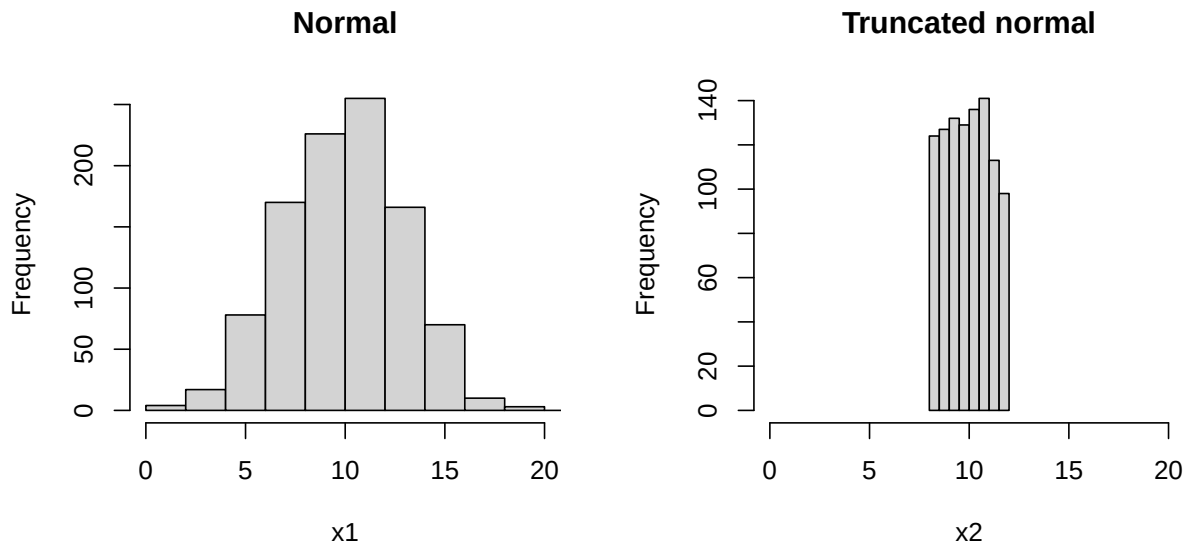
The line above generates 20 numbers from the interval $[0, 1]$. These values can be used as probabilities or values of a CDF.

If you want to implement a new probability distribution in R, you can use `runif()` followed by the `q__()` function that you define for your new distribution to implement a random number generator for it. The example below shows how to implement a “truncated normal” distribution:

```
rtnorm <- function(N, lower, upper, mu, sigma){
  a <- pnorm(lower, mu, sigma)
  b <- pnorm(upper, mu, sigma)
  x <- runif(N, a, b)
  y <- qnorm(x, mu, sigma)
  return(y)
}

use.n <- 1e3
use.mu <- 10
use.sd <- 3
x1 <- rnorm(use.n, use.mu, use.sd)
x2 <- rtnorm(use.n, 8, 12, use.mu, use.sd)

par(mfrow=c(1,2))
hist(x1, main="Normal", xlim=c(0, 20))
hist(x2, main="Truncated normal", xlim=c(0, 20))
```



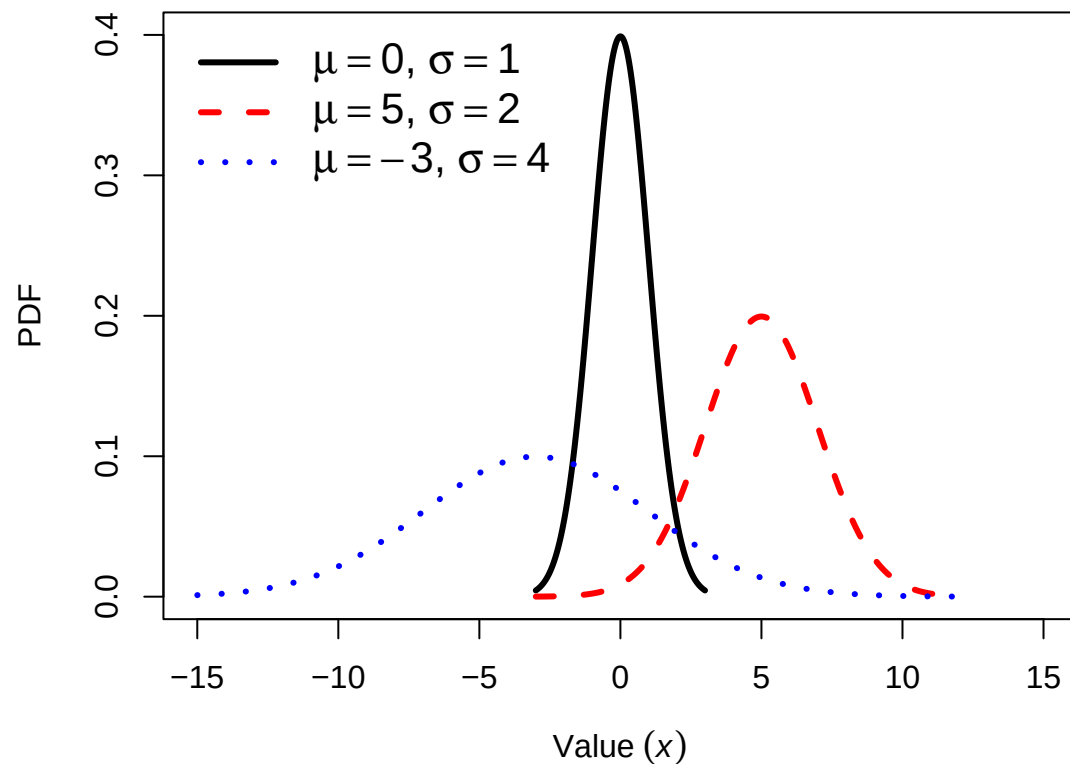
9.3.2 Normal distribution

The **normal distribution** is probably the most important distribution in statistics. Many natural processes result in normal distributions. Consequently, most classical statistical methods (e.g., *t*-tests, ANOVA, and linear models) assume that data come from a normal distribution. The normal distribution is also sometimes called the **Gaussian** distribution because of the work of mathematician Carl Friedrich Gauss, although he did not discover it.

The normal distribution is also sometimes called the **bell curve** because of the shape of its PDF. This term should be avoided because other probability distributions also have a bell shape, and because normal distributions are not always bell-shaped.

The normal distribution is parameterized by its mean, μ , and standard deviation, σ . Some sources prefer the variance σ^2 instead of the standard deviation, and some even use the precision $1/\sigma^2$ or σ^{-2} . This is fine because, as the symbols imply, the SD, variance, and precision are all related. Just pay attention to the notation being used. Using σ instead of σ^2 or $1/\sigma^2$ can be convenient because σ is in the same units as μ .

The figure below shows several normal distributions with different means and standard deviations.



Many phenomena result in normal distributions because of the **Central Limit Theorem (CLT)**. The CLT states that under certain conditions the sum of many random variables will approximate a normal distribution. The main condition is that the random variables will be independent and identically distributed (often abbreviated “*i.i.d.*”). In other words, the values do not depend on each other, but they all come from the same distribution. The exact distribution does not matter. The example below uses the uniform distribution. The CLT also works with the mean of the random variables instead of the sum, because mean is just the sum scaled by sample size.

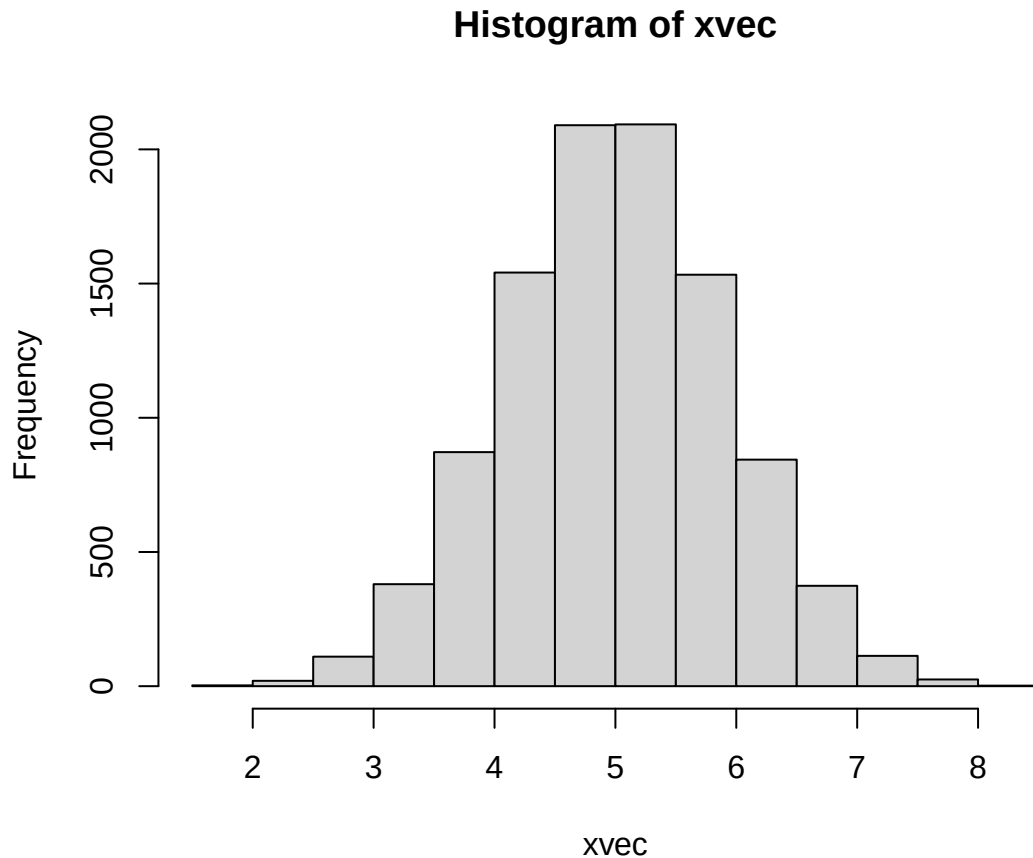
```
# large sample size
N <- 1e4

# set up a vector to hold the sums of
# the random values
xvec <- numeric(N)

# in a loop, draw 10 random values and
# calculate their sum...10000 times
```

```
for(i in 1:N){xvec[i] <- sum(runif(10))}

# histogram of the 10000 sums
hist(xvec)
```



The **standard normal distribution** has $\mu = 0$ and $\sigma = 1$. This is often written as $N(0, 1)$. These parameters are the defaults in the R functions that work with the normal distribution.

9.3.2.1 Normal distribution in R

The normal distribution is accessed using the `_norm` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

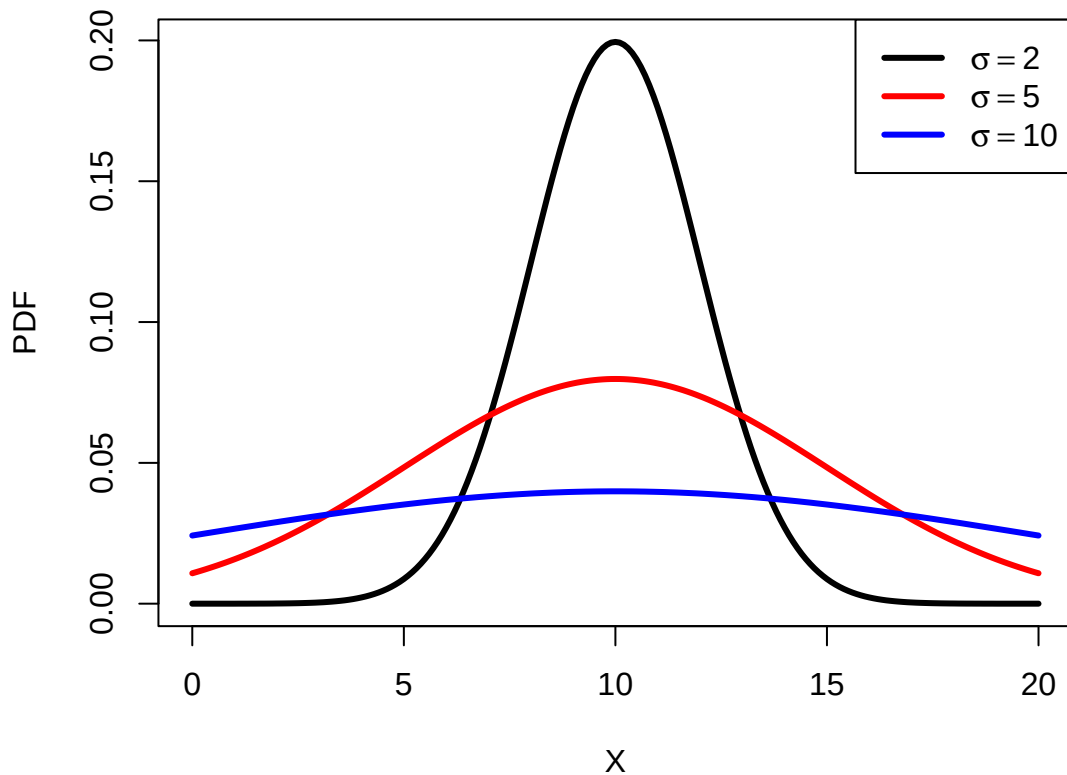
- `dnorm()` : Calculates probability density function (PDF) at x .
- `pnorm()` : Calculates CDF, from $-\infty$ up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qnorm()`.
- `qnorm()` : Calculates the value at a specified quantile or quantiles. The reverse of `pnorm()`.

- `rnorm()`: Draws random numbers from the normal distribution.

The normal distribution in R is parameterized by the mean (argument `mean`) and the SD, not the variance (argument `sd`). The figure generated below shows the effect of increasing the SD on the shape of a distribution. Greater variance (i.e., greater SD) means that the distribution is more spread out.

```
mu <- 10
x <- seq(0, 20, by=0.1)
y1 <- dnorm(x, mu, 2)
y2 <- dnorm(x, mu, 5)
y3 <- dnorm(x, mu, 10)

par(mfrow=c(1,1))
plot(x, y1, type="l", lwd=3, xlab="X", ylab="PDF")
points(x, y2, type="l", lwd=3, col="red")
points(x, y3, type="l", lwd=3, col="blue")
legend("topright", legend=c(expression(sigma==2),
                             expression(sigma==5), expression(sigma==10)),
      lwd=3, col=c("black", "red", "blue"))
```



The normal distribution has several useful properties. Generally, the mean and median are the same (if the distribution is not skewed). Approximately 68% of values fall within 1 SD of the mean, 95% of values fall within about 2 SD of the mean (technically 1.9599 SD, or `qnorm(0.975)`), and 99.7% of values fall within about 3 SD of the mean. So, deviations of > 3 SD are very rare and may signify outliers. The number of SD a value falls away from the mean is sometimes referred to as a **z-score** or a **sigma**. Z-scores are calculated as:

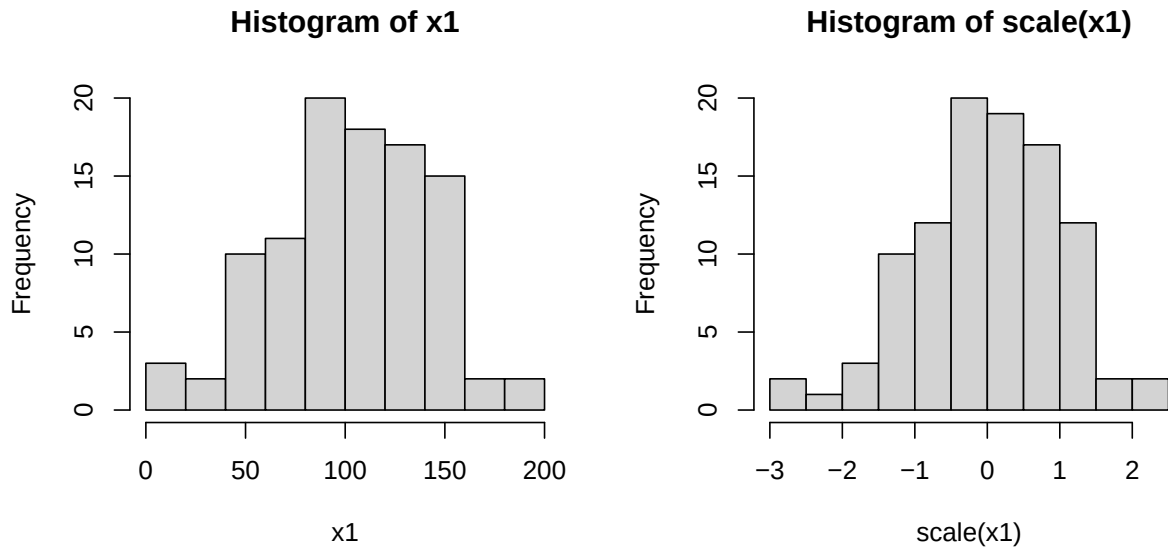
$$z(x) = \frac{x - \mu}{\sigma}$$

Where x is the value, μ is the mean, and σ is the SD.

Z-scores are sometimes used when comparing variables that are normally distributed, but on very different scales. Converting the raw values to their z-scores is referred to as **standardizing** them. Any variable, once standardized, has a mean of 0 and SD of 1. The units of a z-scaled variable are “standard deviations away from the mean”.

Standardized values are sometimes used in regression analyses in place of raw values. Many ordination methods such as principal components analysis (PCA) standardize variables automatically. Z-scores (or standardized values) are calculated in R using the `scale()` function:

```
x1 <- rnorm(100, 100, 40)
par(mfrow=c(1,2))
hist(x1)
hist(scale(x1))
```



```
mean(x1)
```

```
[1] 102.6749
```

```
sd(x1)
```

```
[1] 39.16439
```

```
mean(scale(x1))
```

```
[1] 1.765851e-16
```

```
sd(scale(x1))
```

```
[1] 1
```

Because standardized variables all have the same mean and SD, standardizing a set of variables has the effect of putting all of the variables on equal footing. Without standardizing, variables with larger values or more variance may exert more influence on the analysis than they should simply because of their magnitude.

9.3.3 Lognormal distribution

The **lognormal distribution (a.k.a.: log-normal)** is, as the name implies, a distribution that is *normal on a logarithmic scale*. If a variable X is normally distributed and we exponentiate it, the result $Z = e^X$ will be lognormally distributed. Conversely, if variable Z is lognormally distributed, then $\log(Z)$ will be normally distributed. By convention the **natural log** $\ln$, or $\log_e$, is used to define the lognormal distribution. R's lognormal distribution functions use the natural log scale.

The lognormal distribution arises naturally from **multiplicative** processes. Remember that logarithms relate the operations of multiplication and addition:

$$\log(ab) = \log(a) + \log(b)$$

Suppose some quantity z is determined by multiplying together many positive numbers:

$$z = X_1 X_2 X_3 \cdots X_n$$

Taking its logarithm gives:

$$\log(Z) = \log(X_1) + \log(X_2) + \log(X_3) + \cdots + \log(X_n)$$

This connects the lognormal distribution to the Central Limit Theorem (Section 9.3.2). The ordinary CLT tells us that the sum of many independent random contributions tends toward a normal distribution under fairly general conditions. If a biological quantity instead results from the product of many independent positive contributions, then taking its logarithm converts that product into a sum. Consequently, the logarithm of the resulting quantity can approach a normal distribution, meaning that the original quantity approaches a lognormal distribution.

This is relevant to many biological processes in which changes are proportional rather than additive. For example, imagine an organism whose size changes during each time interval by a somewhat random growth multiplier. Its final size can be written as

$$S = S_0 G_1 G_2 \cdots G_n$$

where each G_i is a growth multiplier applied to the starting size S_0 . Small random differences in growth therefore accumulate multiplicatively, potentially producing a lognormal distribution of final sizes³.

We can demonstrate the same idea with a simulation. Suppose we repeatedly draw 10 random positive values and multiply them together:

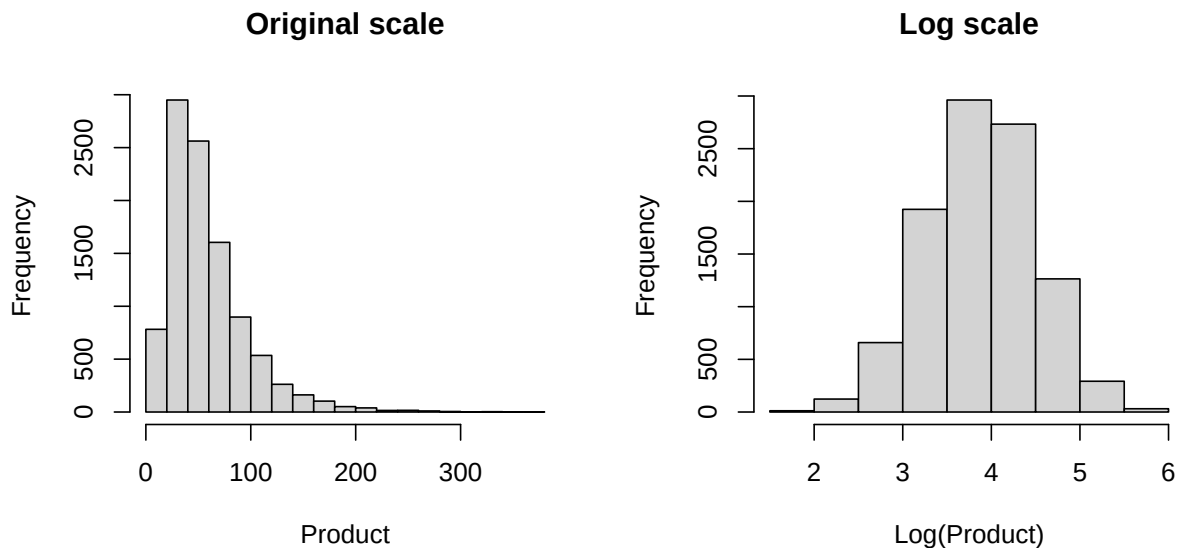
³If the growth factors G_n are constant, then $S_t = S_0 G^t$, or more generally an exponential function $f(t) = ae^{bt}$. Attempting to calculate the derivative of this function leads to preferring Euler's constant e as the base because the function e^t is its own derivative. Here is a [youtube video](#) that explains this visually.

```

# big sample size (10000)
N <- 1e4
# set up vector for 10000 values
xvec <- numeric(N)
# in a loop, draw 10 random values in [1,2]
# and multiply them all together...10000 times
for(i in 1:N){xvec[i] <- prod(runif(10, 1, 2))}

# plot a histogram of the products,
# and histogram of the log of the products
par(mfrow=c(1,2))
hist(xvec, xlab="Product", main="Original scale")
hist(log(xvec), xlab="Log(Product)", main="Log scale")

```



The products in `xvec` are strongly right skewed on their original scale, but their logarithms are approximately normally distributed. This happens because

$$\log \left(\prod_{i=1}^{10} X_i \right) = \sum_{i=1}^{10} \log(X_i)$$

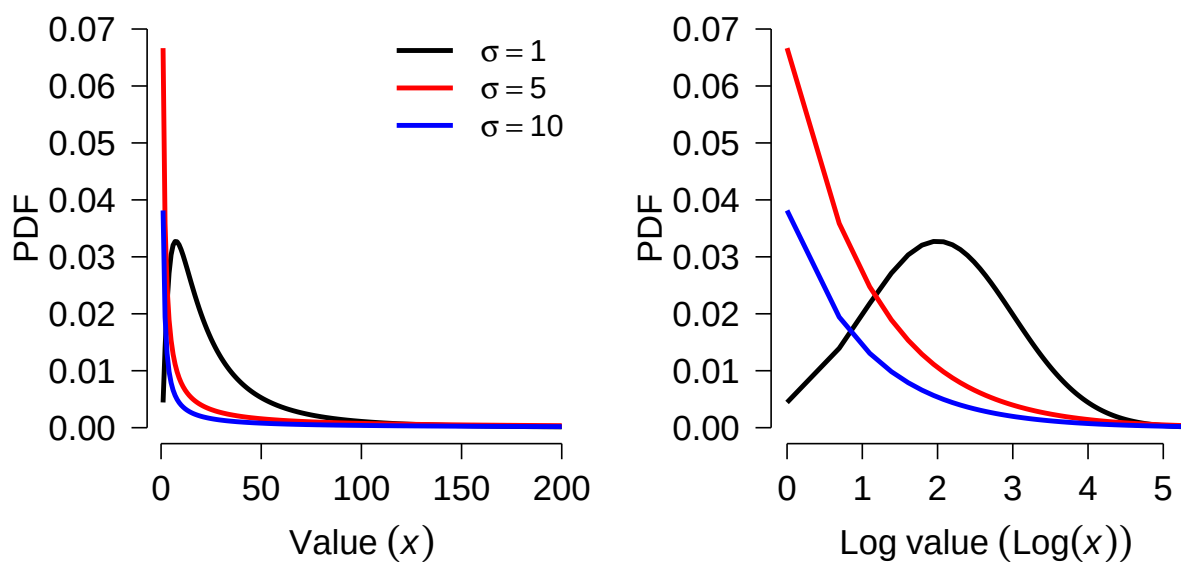
turning the multiplicative process into an additive one.

More generally, lognormal distributions are useful for describing positive, right-skewed variables in which most observations are relatively small but a few observations are much larger. Such patterns are common in biological measurements including organism sizes, abundances, concentrations, and other quantities influenced by proportional or multiplicative processes. Other positive, right-skewed distributions, particularly

the gamma distribution (Section 9.3.4), can produce superficially similar shapes, so the shape of a histogram alone does not establish that a process is lognormal.

The lognormal distribution is parameterized by the mean and standard deviation of the variable after taking its logarithm. In R these parameters are named `meanlog` and `sdlog`. Thus, they are not the mean and standard deviation of the lognormally distributed variable on its original scale. Forgetting which quantities are on the log scale is an easy way to produce confusing results, especially when calculating expected values or graphing distributions.

The figure below shows various lognormal distributions with $\mu = 3$ and different SD on a logarithmic scale (left) and a linear scale.



The figures show the effect that variance can have on the lognormal distribution. When the variance on the log scale is $< \mu$, the distribution is approximately normal on the log scale. As the variance gets larger, the distribution gets more and more right-skewed. Right-skewed means that the values are more and more concentrated near 0, and there are fewer very large values.

Interestingly, the mean of a log-normal distribution is not simply e^μ as one might expect. The mean of a lognormal distribution X on the linear scale is

$$\text{mean}(X) = e^{\left(\frac{\mu + \sigma^2}{2}\right)}$$

In this equation, μ and σ are the mean and SD on the logarithmic scale, or `logmean` and `logsigma` in R terms. The variance on the linear scale is:

$$\text{Var}(X) = e^{2\mu + \sigma^2} (e^{\sigma^2} - 1)$$

What this means is that you cannot transform the parameters of a lognormal distribution from the logarithmic to the linear scale by simply exponentiating or taking logarithms of parameters.

9.3.3.1 Lognormal distribution in R

The lognormal distribution is accessed using the `_lnorm` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dlnorm()`: Calculates probability density function (PDF) at x .
- `plnorm()`: Calculates CDF, from 0 up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qlnorm()`.
- `qlnorm()`: Calculates the value at a specified quantile or quantiles. The reverse of `plnorm()`.
- `rlnorm()`: Draws random numbers from the lognormal distribution.

When working with the lognormal distribution in R it is important to keep in mind that the parameters (`meanlog` and `sdlog`) of the `_lnorm` functions are on the *natural log scale*. The `meanlog` parameter is the *mean of the logarithm* of the values, not the mean of the values or the logarithm of the mean of the values. Similarly, `sdlog` is the SD of the logarithm of the values, not the SD of the values or the logarithm of the SD of the values. The example below shows this:

```
# draw some values
x1 <- rlnorm(1e4, 2, 1)

# mean of the values
mean(x1)
```

```
[1] 12.16433
```

```
# mean of the log(values)
mean(log(x1))
```

```
[1] 1.997859
```

```
# not the log of the mean of the values
log(mean(x1))
```

```
[1] 2.498508
```

```
# sd of the values
sd(x1)
```

```
[1] 16.08477
```

```
# sd of the log(values)
sd(log(x1))
```

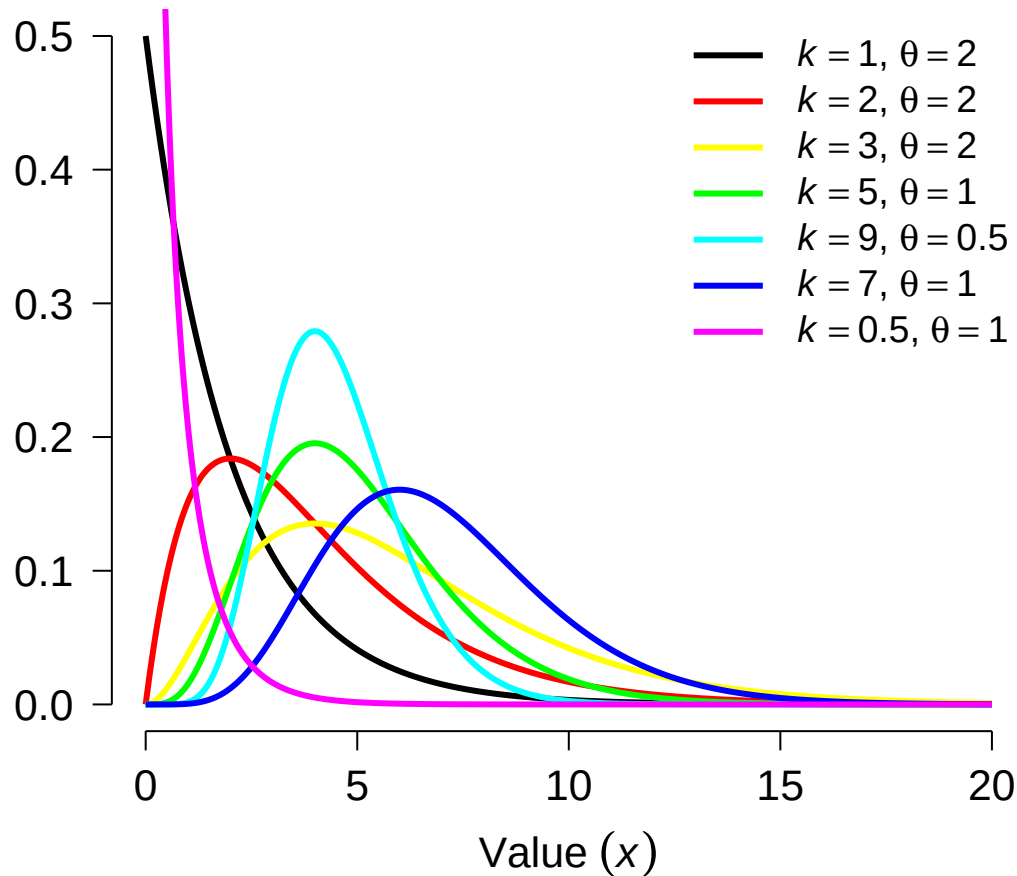
```
[1] 0.9971798
```

```
# exp(meanlog) != mean(x1)
c(exp(2),
  mean(x1))
```

```
[1] 7.389056 12.164334
```

9.3.4 Gamma distribution

The **gamma distribution** describes waiting times until a certain number of events takes place. This means it is the continuous analogue of the **negative binomial distribution**, which describes the number of trials until some certain number of successes. The gamma distribution can also be used empirically to describe positive data whose SD is much larger than the mean. This makes it an alternative to the lognormal distribution for right-skewed data, much like the negative binomial is an alternative to the Poisson. Example gamma distributions are shown below.



The Gamma distribution is parameterized by its **shape** and **scale**.

- The shape parameter k describes the number of events
- The scale parameter θ ("theta") describes the mean time until each event.
- The scale is sometimes expressed as its reciprocal, the **rate**.

In R, the gamma can be parameterized by either the `shape` and `scale`, or `shape` and `rate`. This means you need to name your arguments when working with the gamma distribution.

For example, $\Gamma(k = 4, \theta = 2)$ is the distribution of length of time in days it would take to observe 4 events if events occur on average once every two days. Equivalently, it is the time to observe 4 events if events occur at a rate of 1/2 event per day. Consequently, the mean of a gamma distribution given shape k and scale θ (or rate r) is:

$$\mu = k\theta = \frac{k}{r}$$

The variance is calculated as:

$$\sigma^2 = k\theta^2 = \frac{k}{r^2}$$

Like the negative binomial, the gamma distribution is often used phenomenologically; that is, used to describe a variable even if its mechanistic description (waiting times) doesn't make sense biologically. Just as the negative binomial can be used to model count data with variance greater than the mean (i.e., an overdispersed Poisson), a gamma distribution can be used for positive data with $\sigma \gg \mu$. I.e., the gamma can be used in many of the same cases as the lognormal, although the lognormal may have a clearer biological interpretation.

9.3.4.1 Gamma distribution in R

The gamma distribution is accessed using the `_gamma` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dgamma()` : Calculates probability density function (PDF) at x .
- `pgamma()` : Calculates CDF, from 0 up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qgamma()`.
- `qgamma()` : Calculates the value at a specified quantile or quantiles. The reverse of `pgamma()`.
- `rgamma()` : Draws random numbers from the gamma distribution.

Note that when working with the gamma distribution in R, you must supply either the `shape` and the `rate`, or the `shape` and the `scale`. Supplying `scale` and `rate` will return an error.

```
# will work:
rgamma(10, shape=3, scale=2)
rgamma(10, shape=3, rate=0.5)

# will return error:
rgamma(10, scale=2, rate=0.5)
```

9.3.5 Beta distribution

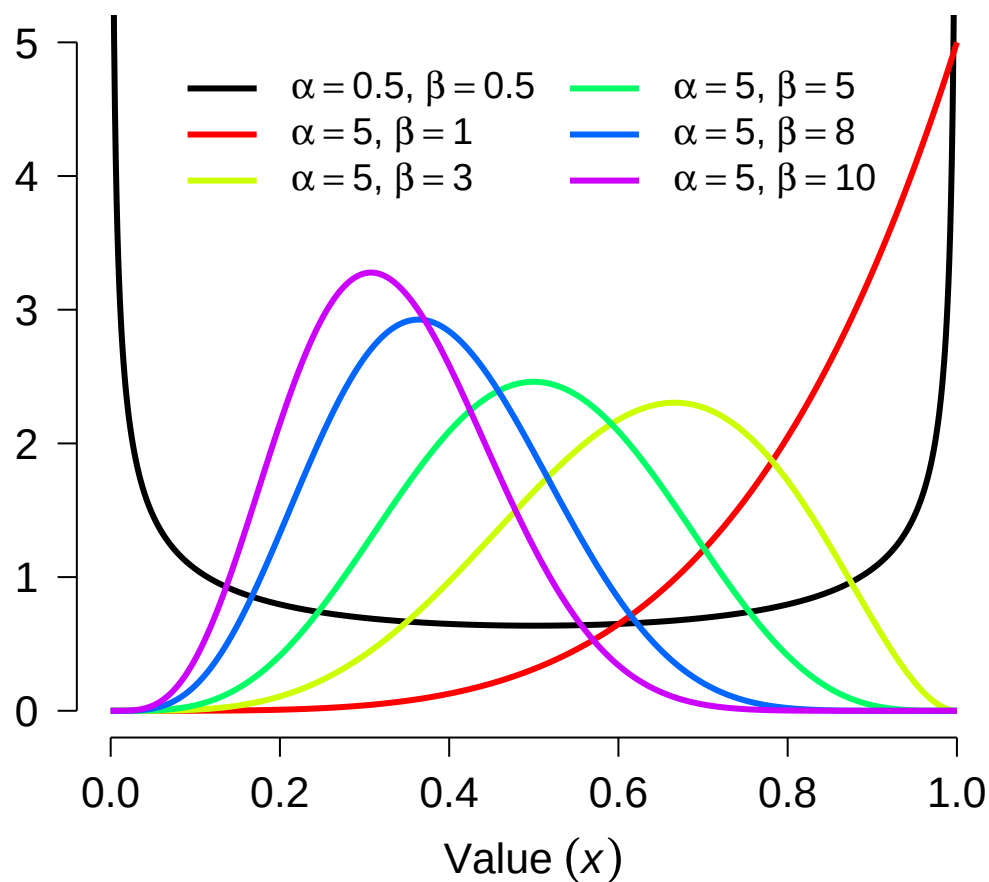
The **beta distribution** is defined on the interval $[0, 1]$ and is parameterized by two positive shape parameters, α and β . For this reason, the beta distribution is often used to model the distribution of probabilities. The beta distribution is probably the most commonly used distribution for variables constrained to lie between 0 and 1, such as probabilities, proportions, and fractions.

The beta distribution is closely related to the binomial distribution. One way to think about the two shape parameters is as summaries of a binomial trial. Parameter α behaves like the number of successes plus one, while β behaves like the number of failures plus one. This interpretation provides useful intuition for

the shape of the distribution: increasing α shifts probability toward 1, while increasing β shifts probability toward 0. However, α and β are simply shape parameters and do not have to be integers; but, they do have to be positive.

An even simpler explanation is that the balance of α and β determine where most of the probability lies. Increasing α places more probability near 1, whereas increasing β places more probability near 0. Increasing both will crowd the distribution near 0.5. Decreasing both will push the probability out towards 0 and 1 and away from 0.5.

The figure below shows various beta distributions. Notice how probability is shifted toward the center when α and β both increase, and how the distribution becomes U-shaped when they both decrease.



The beta distribution has mean

$$\mu = \frac{\alpha}{\alpha + \beta}$$

and variance

$$\sigma^2 = \frac{\alpha\beta}{(\alpha + \beta)^2 (\alpha + \beta + 1)}$$

These expressions can be re-arranged to solve for α and β given a mean and variance:

$$\alpha = \left(\frac{1 - \mu}{\sigma^2} - \frac{1}{\mu} \right) \mu^2$$

$$\beta = \alpha \left(\frac{1}{\mu} - 1 \right)$$

This is useful if you want to use proportions or percentages that were reported as a mean and variance (or SD, or SE). For example, if a paper reports a survival rate estimate as 0.25 ± 0.18 SE, it's tempting to misinterpret that as meaning that the survival rate is a normally-distributed variable. However, treating the survival rate as normally distributed would imply that about 8% of the time, the survival rate was negative!

```
pnorm(0, 0.25, 0.18)
```

```
[1] 0.08243327
```

So, you should convert these estimates to a beta distribution if you want to understand how they might vary:

$$\alpha = \left(\frac{1 - 0.25}{(0.18)^2} - \frac{1}{0.25} \right) 0.25^2 \approx 1.1968$$

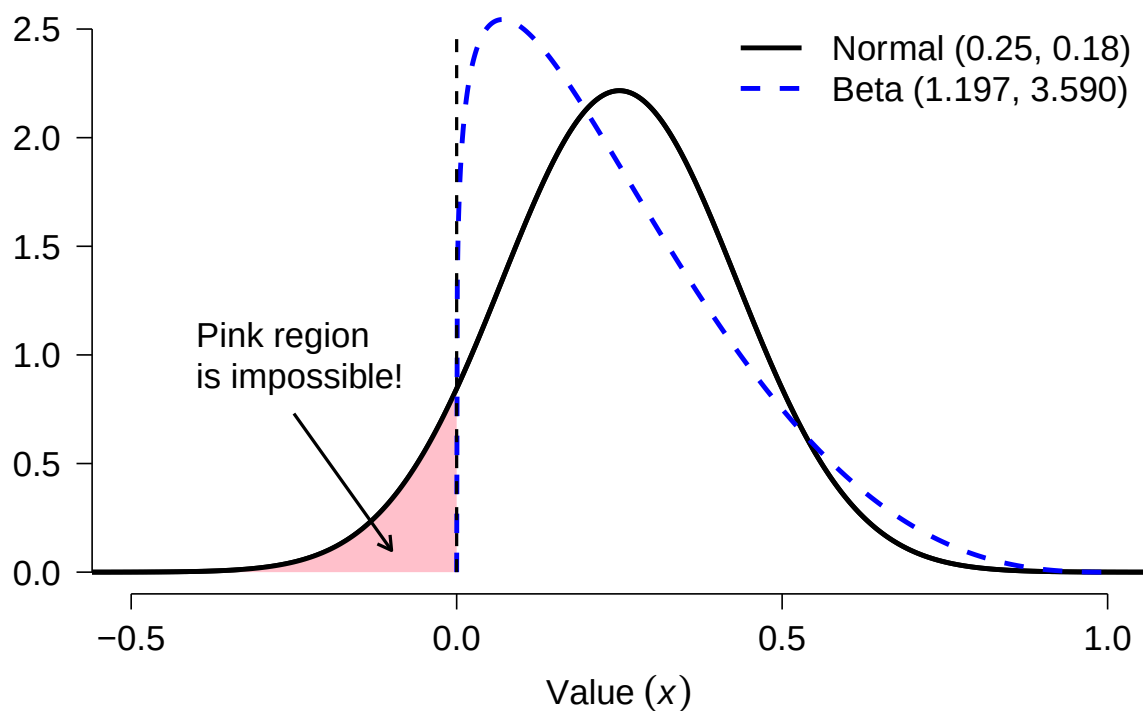
$$\beta = 1.1968 \left(\frac{1}{0.25} - 1 \right) \approx 3.5903$$

The figure below shows the two distributions, the normal in black and the beta in blue. The dashed vertical line at 0 shows that part of the normal distribution implied by the mean and SE of a survival “probability” is not possible. Converting to a beta distribution preserves the mean and variance while ensuring that all values fall between 0 and 1. This more accurately reflects the true nature of the uncertainty about the probability.

Note that this conversion cannot be done for some combinations of μ and σ : both α and β must both be positive, so if you calculate a non-positive value for either parameter then the conversion won't work. These formulas also require that

$$\sigma^2 < \mu(1 - \mu)$$

This can sometimes happen for probabilities very close to 0 or 1 with large variances.



9.3.5.1 Beta distribution in R

The beta distribution is accessed using the `_beta` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dbeta()` : Calculates probability density function (PDF) at x .
- `pbeta()` : Calculates CDF, from 0 up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qbeta()`.
- `qbeta()` : Calculates the value at a specified quantile or quantiles. The reverse of `pbeta()`.
- `rbeta()` : Draws random numbers from the beta distribution.

9.3.5.2 The beta and the binomial

The relationship between the beta and binomial distributions is especially useful for making inferences about probabilities. The binomial distribution describes the number of successes we observe when the probability of success p is fixed. The beta distribution can instead be used to describe our uncertainty about the value of p itself.

We can take advantage of the relationship between the binomial and beta distributions to make inferences from count data. Imagine a study where biologists track the probability that fruit flies die before they are 3 weeks old. They report that 30 out of 130 flies die on or before day 21 of life. What can we estimate about the underlying distribution of 3-week survival rates?

One common starting point is to treat all survival probabilities between 0 and 1 as equally plausible before seeing the data. This corresponds to $\alpha = \beta = 1$. We then add the observed successes to α and failures to β .

Then we can model uncertainty either in the survival probability or mortality probability, depending on how we frame successes or failures. If survival is a success, then

$$p_{\text{survival}} \sim \text{Beta}(\alpha = 101, \beta = 31)$$

Alternatively, we can model mortality rate using mortality as a success:

$$p_{\text{mortality}} \sim \text{Beta}(\alpha = 31, \beta = 101)$$

The mean survival rate is:

```
Alpha <- 101
Beta <- 31

# mean estimated survival probability
Alpha / (Alpha + Beta)
```

```
[1] 0.7651515
```

```
# middle 95% of distribution
qbeta(c(0.025, 0.975), Alpha, Beta)
```

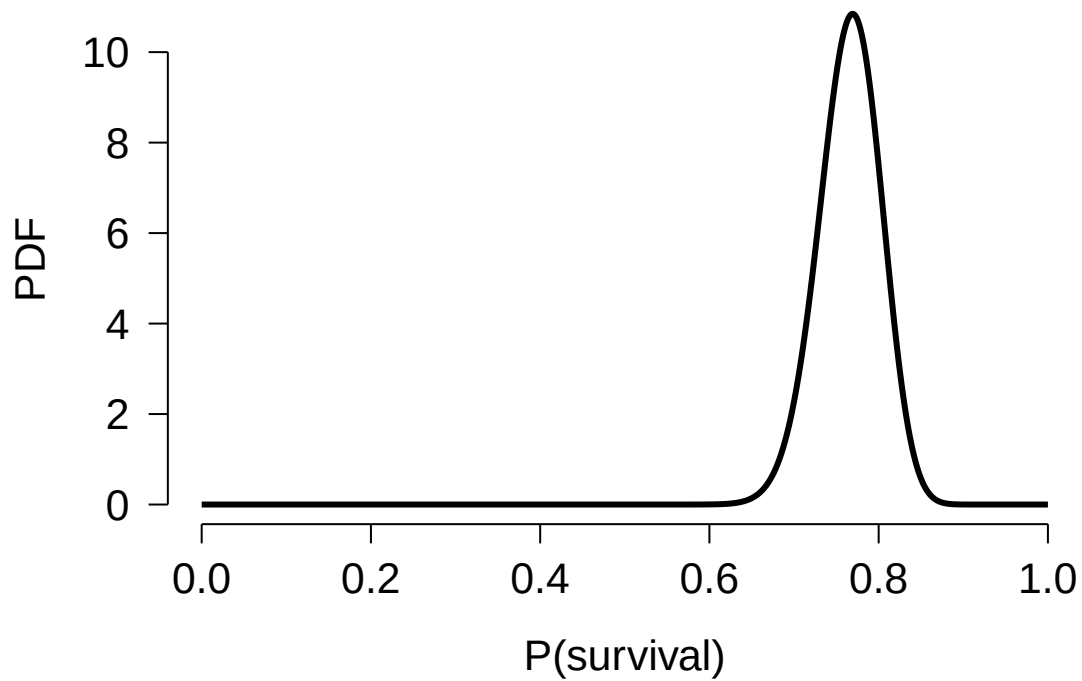
```
[1] 0.6894960 0.8332005
```

This tells us that, given our experimental observation of 100 survivors out of 130 flies, that the true underlying survival rate is about 0.765 and is probably in the interval $[0.689, 0.833]$. We can see that visually below:

```
# domain of probability
x <- seq(0, 1, length=1e3)

# survival rate distribution
y.surv <- dbeta(x, shape1=101, shape2=31)

par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1, las=1,
    cex.axis=1.3, cex.lab=1.3)
plot(x, y.surv, type="l", lwd=3,
     xlab="P(survival)", ylab="PDF")
```



9.3.6 Exponential distribution

The **exponential distribution** describes the distribution of waiting times until a single event occurs, given a constant rate per unit time of that event occurring. This makes it the continuous analog of the **geometric distribution**. The exponential distribution is also a special case of the **gamma distribution** where shape parameter $k = 1$.

The exponential distribution should not be confused with the **exponential family of distributions**, although the exponential distribution is a member of that family. The exponential family of distributions is a broad class that includes the normal, exponential, gamma, beta, Poisson, and many others.

The exponential distribution is parameterized by a single rate parameter, λ , which describes the expected event rate per unit time. For example, $\lambda = 2$ per day means an average waiting time of $1/0.2 = 5$ days. It does not mean exactly 0.2 events occur every day—this is a long run average. This property means that λ must be strictly positive. This is exactly the same as the rate parameter r sometimes used to describe the gamma distribution. The mean of an exponential distribution X is:

$$\mu(X) = \frac{1}{\lambda}$$

and the variance is:

$$\sigma^2(X) = \frac{1}{\lambda^2}$$

Interestingly, this implies that the standard deviation σ is the same as the mean μ . Contrast this with the Poisson distribution, where the *variance* σ^2 is the same as the mean. Thus, the CV of the exponential distribution is always 1.

The exponential distribution is supported for all non-negative real numbers; i.e., the half-open interval $[0, +\infty)$. Like the gamma distribution, the exponential distribution can be used when its mechanistic assumptions make sense, such as waiting times or lifespans with a constant event rate. It also produces strongly right-skewed values, with many small and few large values, but right skew alone is not sufficient reason to assume an exponential distribution.

One other important property of the exponential distribution is that it is *memoryless*. If the event rate is constant, then the probability of waiting an additional amount of time does not depend on how long you have already waited. For example, if mortality follows an exponential distribution with a constant hazard, an individual that has already survived 5 years has the same expected remaining lifespan as a newly observed individual. The memoryless property is related to why the **gambler's fallacy** is a fallacy. If independent events occur with constant probability or rate, a long period without an event does not make that event "due." The geometric distribution has this property in discrete time, and the exponential distribution has the analogous property in continuous time.

9.3.6.1 Exponential distribution in R

The exponential distribution is accessed using the `_exp` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dexp()` : Calculates probability density function (PDF) at x .
- `pexp()` : Calculates the cumulative probability $P(X \leq x)$. I.e., it tell you what proportion of the distribution lies at or below a specified value. The reverse of `qexp()`.
- `qexp()` : Calculates the value at a specified quantile or quantiles. The reverse of `pexp()`.
- `rexp()` : Draws random numbers from the exponential distribution.

9.3.7 Triangular distribution

The **triangular distribution** is sometimes used as a simple "lack of knowledge" distribution when little information is available beyond a plausible minimum, maximum, and most likely value. It is used more often to represent uncertainty in simulations and risk models than to model observed biological data directly.

Sometimes we need to model a process about which we have very little information. For example, we may want to simulate population dynamics without knowing a key survival rate, or organismal growth without knowing a key growth constant. In these situations we might have only a vague idea of how a parameter or an outcome are distributed. At a minimum, we can usually infer or estimate the range and central tendency of a variable. Those are enough to estimate a triangular distribution.

The triangular distribution has three parameters: the lower limit a , the upper limit b , and the mode (most common value) c . Any triangular distribution must satisfy $a < b$ and $a \leq c \leq b$.

The mean of a triangular distribution X is the mean of its parameters:

$$\mu(X) = \frac{a + b + c}{3}$$

and the variance is

$$\sigma^2(X) = \frac{a^2 + b^2 + c^2 - ab - ac - bc}{18}$$

In addition to serving as a stand-in distribution when data are scarce, the triangular distribution can arise in several natural situations. For example, the mean of two standard uniform variables follows a triangular distribution (see below).

9.3.7.1 Triangular distribution in R

The triangular distribution is available in the add-on package `extraDistr` (Wolodzko 2026). The triangular distribution is accessed using the `_triang` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dtriang()`: Calculates probability density function (PDF) at x .
- `ptriang()`: Calculates $P(X \leq x)$, the CDF from a to x . The reverse of `qtriang()`.
- `qtriang()`: Calculates the value at a specified quantile or quantiles. The reverse of `ptriang()`.
- `rtriang()`: Draws random numbers from the triangular distribution.

The code below shows the PDFs of various triangular distributions. Notice that `c` is not used as a variable name, to avoid potentially masking the very critical R function `c`.

```
library(extraDistr)
ax <- c(1, 1, 3)
bx <- c(5, 10, 7)
cx <- c(3, 3, 6)

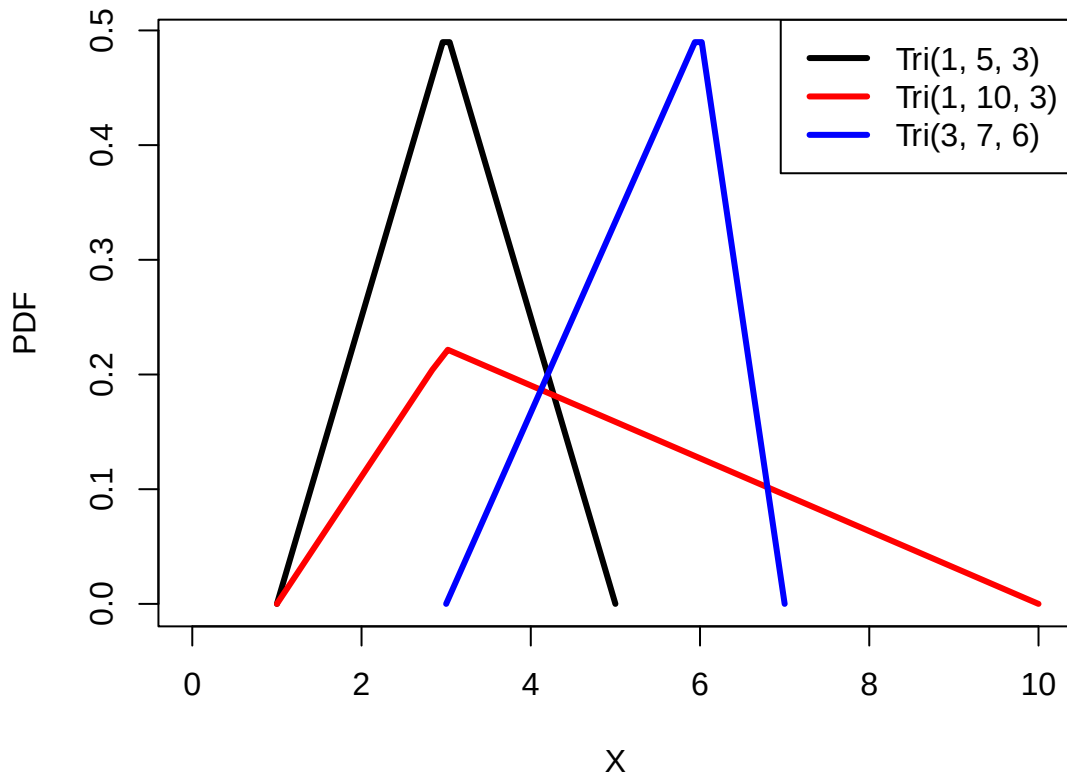
x1 <- seq(ax[1], bx[1], length=50)
x2 <- seq(ax[2], bx[2], length=50)
x3 <- seq(ax[3], bx[3], length=50)
y1 <- dtriang(x1, ax[1], bx[1], cx[1])
y2 <- dtriang(x2, ax[2], bx[2], cx[2])
y3 <- dtriang(x3, ax[3], bx[3], cx[3])

plot(x1, y1, type="l", lwd=3, xlab="X", ylab="PDF",
      xlim=c(0, 10))
```

```

points(x2, y2, type="l", lwd=3, col="red")
points(x3, y3, type="l", lwd=3, col="blue")
legend("topright", legend=c("Tri(1, 5, 3)", "Tri(1, 10, 3)", "Tri(3, 7, 6)"),
      lwd=3, col=c("black", "red", "blue"))

```



The code below demonstrates how a triangular distribution can arise as the distribution of means of two standard uniform variables x_1 and x_2 . The command `density()` calculates the kernel density estimate of a vector. This kernel density estimate is an empirical estimate of the PDF of a variable.

```

# random number seed for reproducibility
set.seed(42)

# sample size
n <- 10^(2:4)

# set up lists to hold simulation results
x1 <- vector("list", length(n))

```

```

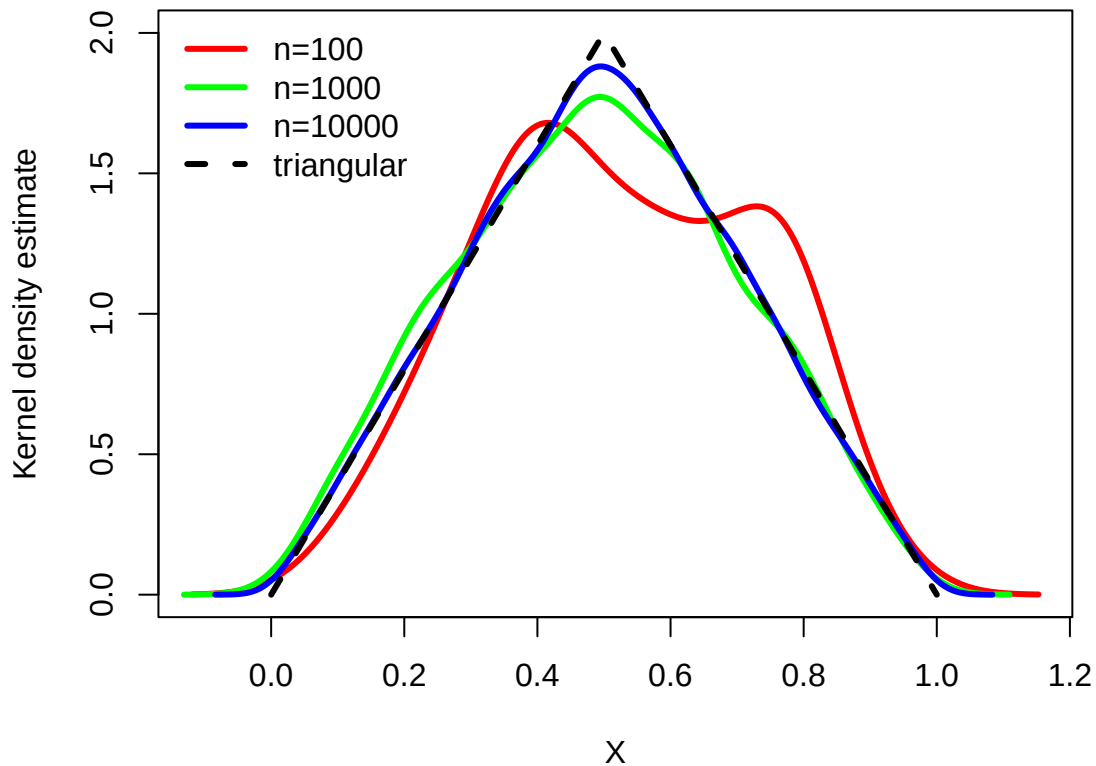
y <- vector("list", length(n))
x <- vector("list", length(n))

# simulate uniforms in a for loop
for(i in 1:length(n)){
  x[[i]] <- (runif(n[i]) + runif(n[i]))/2
}

# calculate the empirical density function
for(i in 1:length(n)){
  z <- density(x[[i]])
  x1[[i]] <- z$x
  y[[i]] <- z$y
}

# make the graph
cols <- rainbow(3)
plot(x1[[1]], y[[1]], type="l", lwd=3, col=cols[1],
      xlab="X", ylab="Kernel density estimate", ylim=c(0, 2))
for(i in 2:3){
  points(x1[[i]], y[[i]], type="l", lwd=3, col=cols[i])
}
legend("topleft", legend=c("n=100", "n=1000", "n=10000", "triangular"),
      bty="n", lwd=3, col=c(cols[1:3], "black"), lty=c(1,1,1,2))
# add the triangular distribution
curve(dtriang(x, 0, 1, 0.5), from=0, to=1, add=TRUE, lwd=3, lty=2)

```



9.3.8 Distribution summary

The table below summarizes some the key features of the **discrete distributions** we explored in this module.

Distribution	Support	What it models
Bernoulli	$x \in \{0, 1\}$	Outcome of a single binary trial
Binomial	$x \in \{0, 1, \dots, n\}$	Number of successes in n independent trials with constant probability p
Poisson	$x \in \{0, 1, 2, \dots\}$	Number of events occurring independently at a constant rate
Negative binomial	$x \in \{0, 1, 2, \dots\}$	Number of failures before a specified number of successes; commonly used for overdispersed count data
Geometric	$x \in \{0, 1, 2, \dots\}$	Number of failures before the first success, with constant probability p

Distribution	Support	What it models
Beta-binomial	$x \in \{0, 1, \dots, n\}$	Number of successes in n trials when the probability p varies
Multinomial	$x_i \in \{0, 1, \dots, n\}$, with $\sum_i x_i = n$	Counts among k categories from n trials

The table below summarizes some the key features of the **continuous distributions** we explored in this module.

Distribution	Support	What it models
Uniform	$x \in [a, b]$	Continuous variables where all values within a bounded interval are equally likely
Normal	$x \in (-\infty, +\infty)$	Continuous variables arising from many additive processes; symmetric data concentrated around a mean
Lognormal	$x \in (0, +\infty)$	Positive variables that are normally distributed on a log scale; often arise from multiplicative processes
Gamma	$x \in (0, +\infty)$	Positive, right-skewed variables; waiting times until a specified number of events occurring at a constant rate
Beta	$x \in [0, 1]$	Continuous proportions and probabilities, including uncertainty about the probability of a binary event
Exponential	$x \in [0, +\infty)$	Waiting times until the first event when events occur at a constant rate
Triangular	$x \in [a, b]$	Bounded continuous variables when only the minimum, maximum, and most likely value are known

Chapter 10

Comparing data to distributions

10.1 Fitting and testing distributions

Deciding what probability distribution to use to analyze your data is an important step in biological data analysis. The next step is to see how well your data actually match a distribution. This section demonstrates ways to compare data to probability distributions. We will also explore some methods to explicitly test whether or not a set of values came from a particular distribution.

10.2 Estimating distributional parameters

The function `fitdistr()` in package `MASS` estimates the **parameters** of a distribution most likely to produce random variables supplied as input. Simply put, if you have a variable `x` and want to know what parameters of some distribution are most likely to produce `x`, then `fitdistr()` will estimate this for you.

```
library(MASS)

set.seed(123)
n <- 50

# estimate parameters of a normal
x1 <- rnorm(n, 12, 2)
fitdistr(x1, "normal")
```

```
      mean      sd
12.0688071  1.8331290
( 0.2592436) ( 0.1833129)
```

```
# estimate parameters of a poisson
x2 <- rpois(n, 8)
fitdistr(x2, "Poisson")
```

```
lambda
8.2200000
(0.4054627)
```

```
# estimate parameters of a geometric
x3 <- rgeom(n, 0.7)
fitdistr(x3, "geometric")
```

```
prob
0.90909091
(0.03876377)
```

```
# estimate parameters of a beta
# unlike most distributions, need to
# supply starting values
x4 <- rbeta(n, 2, 0.5)
fitdistr(x4, "beta", start=list(shape1=1, shape2=1))
```

```
shape1      shape2
2.1520168    0.5070293
(0.4828815) (0.0844015)
```

```
# estimate parameters of a negative binomial
x5 <- rnbinom(n, 4, 0.2)
fitdistr(x5, "negative binomial")
```

```
size      mu
3.674590   15.660000
( 0.907684) ( 1.283731)
```

How did `fitdistr()` do? Most of the parameter estimates were pretty close to the true values. The estimates generally get closer as the sample size increases. Try increasing `n` to 500 and rerunning the code above.

10.3 Graphical methods for examining distributions

10.3.1 Comparing distributions with PDF or PMF plots

We can compare the distributions implied by the estimated parameters to the actual data using the quantile or distribution functions. The basic idea is to plot the data distribution, and superimpose the equivalent plots of the hypothetical distributions.

In the example below, we compare the probability mass function (PMF) of a discrete distribution (e.g., counts) to the PMFs of two possible distributions that might match the data. Note that the PMF of a discrete distribution can be estimated as the proportion of observations that take on each value.

```
set.seed(123)

# the counts:
x <- rlnbinom(100, mu=4, size=2)

# fit parameters for poisson and negative binomial
f1 <- fitdistr(x, "Poisson")
f2 <- fitdistr(x, "negative binomial")

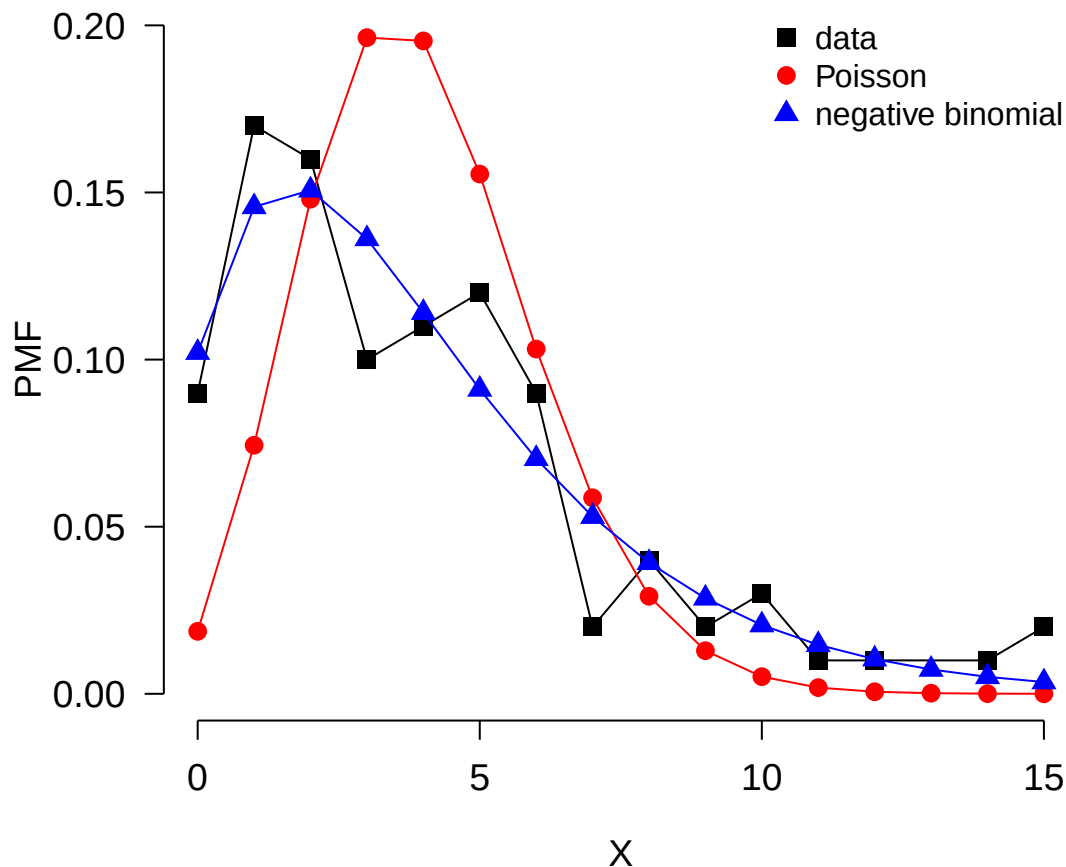
# calculate PMFs of candidate distributions across
# range of data
xx <- 0:max(x)
y1 <- dpois(xx, f1$estimate)
y2 <- dnbinom(xx, size=f2$estimate["size"],
              mu=f2$estimate["mu"])

# empirical PMF of counts
y3 <- table(x)/length(x)
plot.x <- as.numeric(names(y3))
plot.y <- as.numeric(y3)

# make plot
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, lend=1,
    cex.axis=1.2, cex.lab=1.2, bty="n")
plot(plot.x, plot.y, type="o",
     pch=15, cex=1.3, ylim=c(0, 0.2),
     ylab="PMF", xlab="X")
points(xx, y1, type="o", pch=16, cex=1.3, col="red")
points(xx, y2, type="o", pch=17, cex=1.3, col="blue")
```



```
legend("topright", legend=c("data", "Poisson", "negative binomial"),
      pch=15:17, pt.cex=1.3,
      col=c("black", "red", "blue"), bty="n")
```



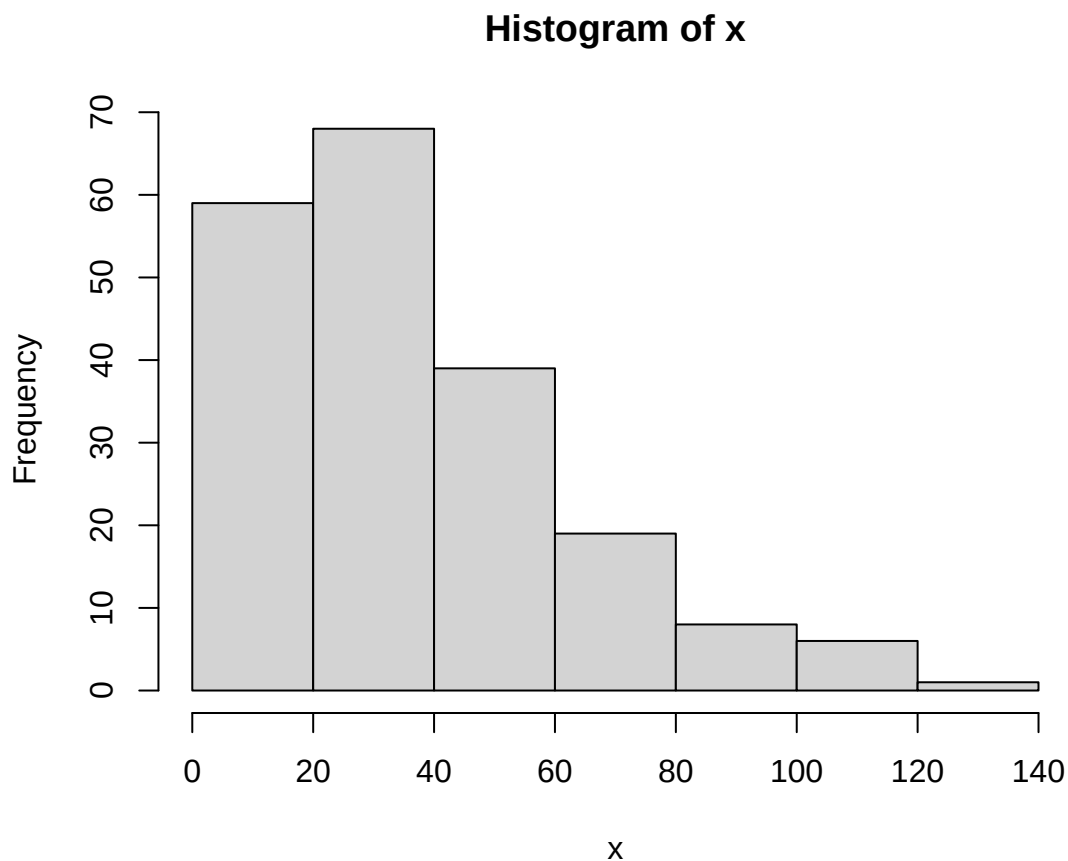
In this example, the data (black) seem to be better described by the negative binomial (blue) than by the Poisson (red).

The same idea can be applied to continuous distributions. The only difference is in how the probability density function (PDF) of the data is estimated (note PDF, not PMF). The PDF of a continuous distribution is estimated with function `density()`. It is a good idea to limit the estimation of density functions to the domain of the data. Keep in mind, however, that different distributions have different limits. If the data contain values outside the supported interval of the suspected distribution, that distribution cannot be correct.

The example below illustrates this: the data (x) are all positive, but one of the distributions tested (normal) will produce negative values using the estimated parameters (`f1`). That alone should lead the researcher in this situation to eliminate the normal as a possibility; the plot only confirms the unsuitability of the normal for modeling x .

```
set.seed(123)

# the data:
x <- rgamma(200, shape=2, scale=20)
hist(x)
```



```
# fit parameters for poisson and negative binomial
f1 <- fitdistr(x, "normal")
f2 <- fitdistr(x, "lognormal")
f3 <- fitdistr(x, "gamma")

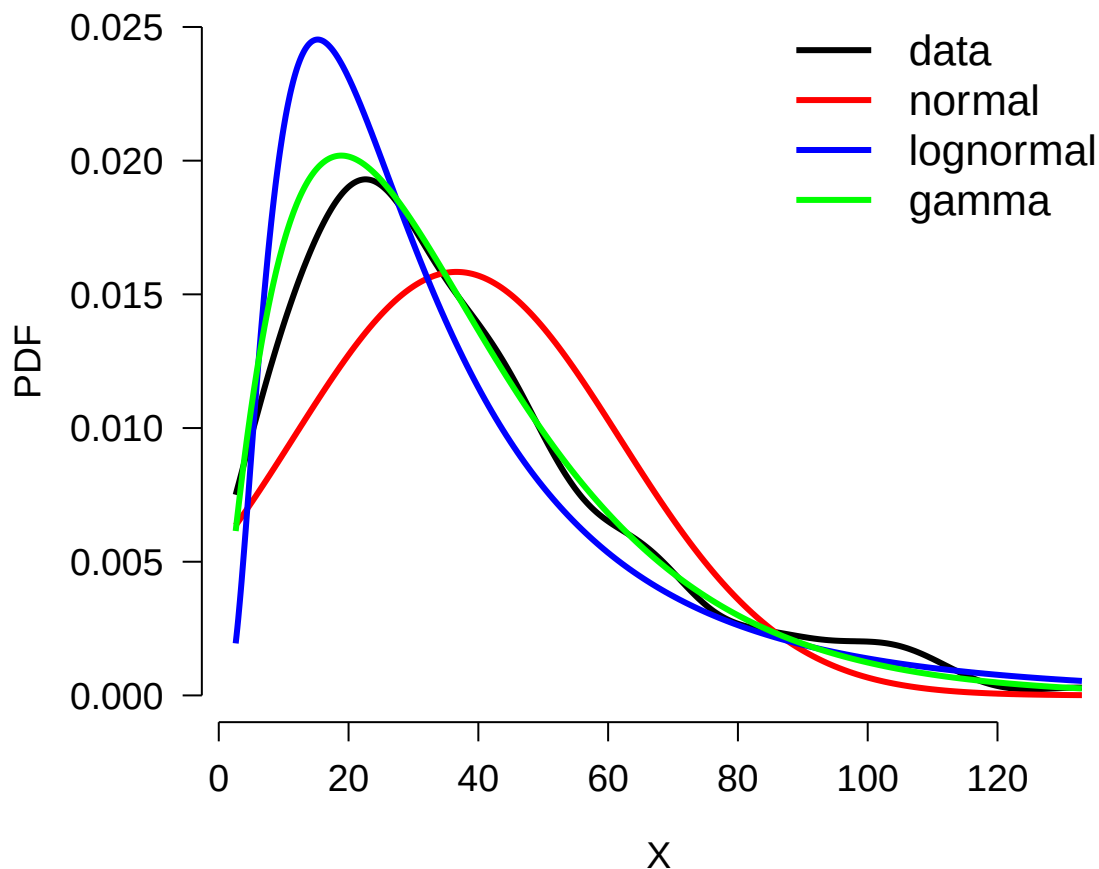
# calculate PMFs of candidate distributions across
# range of data
xx <- seq(min(x), max(x), length=1000)
y1 <- dnorm(xx, f1$estimate[1], f1$estimate[2])
y2 <- dlnorm(xx, f2$estimate[1], f2$estimate[2])
y3 <- dgamma(xx, shape=f3$estimate[1], rate=f3$estimate[2])
```

```

# empirical PDF of data
dx <- density(x, from=min(x), to=max(x))

# make plot
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, lend=1,
    cex.axis=1.2, cex.lab=1.2, bty="n")
plot(dx$x, dx$y,
     type="l", lwd=3, ylim=c(0, 0.025),
     ylab="", xlab="X")
title(ylab="PDF", line=4)
points(xx, y1, type="l", lwd=3, col="red")
points(xx, y2, type="l", lwd=3, col="blue")
points(xx, y3, type="l", lwd=3, col="green")
legend("topright",
     legend=c("data", "normal", "lognormal", "gamma"),
     lwd=3, col=c("black", "red", "blue", "green"),
     cex=1.3, bty="n")

```



The plot suggests that the data are better described by a gamma distribution than by a normal or lognormal. This is because the curve for the data is most closely matched by the curve for the gamma. Try adjusting the sample size in the example above (e.g., `n <- 50`) and see if the correct distribution always matches the data.

10.3.2 Comparing distributions using CDF and ECDF

Another way to compare data to distributions is with the cumulative distribution function (CDF). The basic idea is the same as using density functions: compare the empirical CDF (ECDF) of the data to the CDFs of potential distributions and judge which one best matches the data.

The example below shows how to use ECDF and CDF to compare a continuous distribution to various target distributions.

```
set.seed(123)

# make the data:
```

```

x <- rlnorm(100, meanlog=log(4), sdlog=log(3))

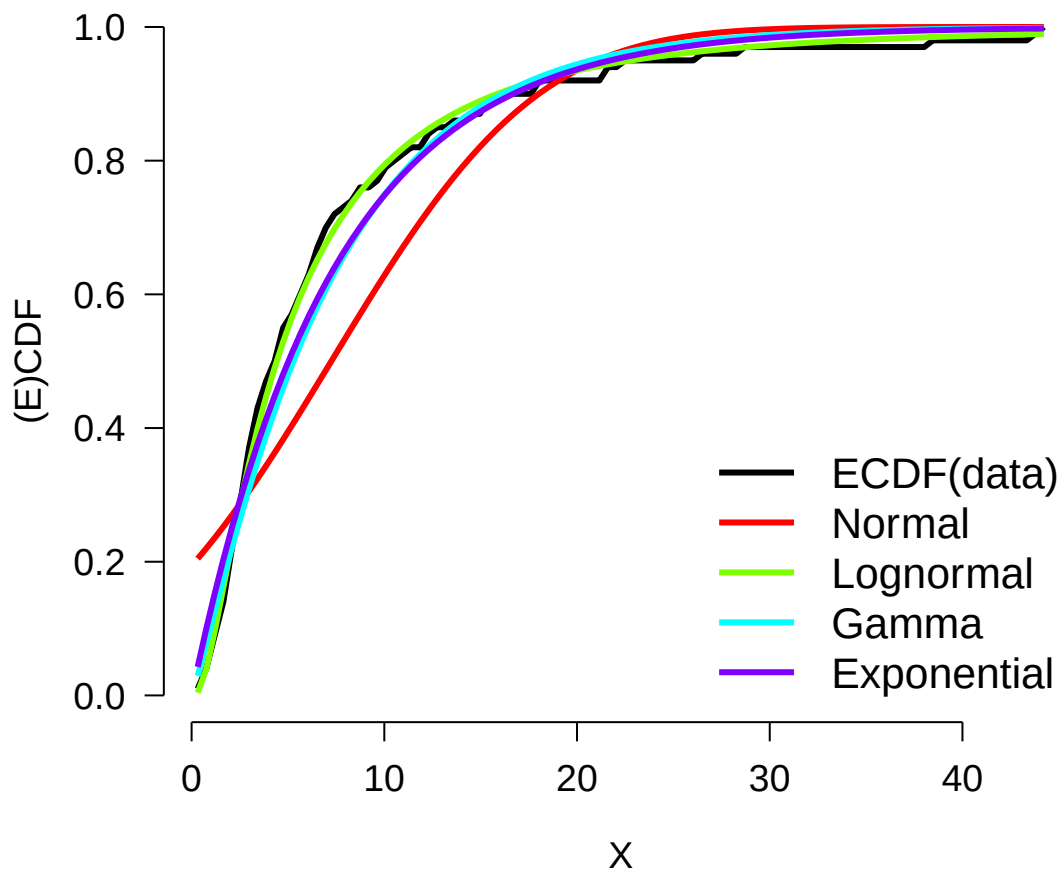
# fit some distributions
f1 <- fitdistr(x, "normal")
f2 <- fitdistr(x, "lognormal")
f3 <- fitdistr(x, "gamma")
f4 <- fitdistr(x, "exponential")

# calculate density functions over domain of x
xx <- seq(min(x), max(x), length=100)
y1 <- pnorm(xx, f1$estimate["mean"], f1$estimate["sd"])
y2 <- plnorm(xx, f2$estimate["meanlog"], f2$estimate["sdlog"])
y3 <- pgamma(xx,
             shape=f3$estimate["shape"],
             rate=f3$estimate["rate"])
y4 <- pexp(xx, rate=f4$estimate["rate"])

# ecdf of data
y5 <- ecdf(x)(xx)

# fancy plot
cols <- c("black", rainbow(4))
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, lend=1,
    cex.axis=1.2, cex.lab=1.2, bty="n")
plot(xx, y5, type="l", lwd=3, xlab="X",
     ylab="(E)CDF", ylim=c(0,1), col=cols[1])
points(xx, y1, type="l", lwd=3, col=cols[2])
points(xx, y2, type="l", lwd=3, col=cols[3])
points(xx, y3, type="l", lwd=3, col=cols[4])
points(xx, y4, type="l", lwd=3, col=cols[5])
legend("bottomright",
     legend=c("ECDF(data)", "Normal", "Lognormal", "Gamma", "Exponential"),
     lwd=3, col=cols, bty="n", cex=1.3)

```



In this example, the ECDF of the original data (black) appears to best match the lognormal (green)—as well it should, because the data came from a lognormal distribution. The normal (red) is clearly inappropriate because it diverges from the actual data over most of the range of the data. The gamma and exponential are each better than the normal, but not as close to the data as the lognormal.

10.3.3 Comparing distributions using QQ plots

Finally, you can use **quantile-quantile (QQ) plots** to compare data to different distributions that you suspect they may follow. The example below uses a QQ plot instead of ECDF as seen above.

```
set.seed(123)

# true distribution: lognormal
n <- 100
x <- rlnorm(n, meanlog=log(4), sdlog=log(3))

# estimate distributional parameters
```

```

f1 <- fitdistr(x, "normal")
f2 <- fitdistr(x, "lognormal")
f3 <- fitdistr(x, "gamma")
f4 <- fitdistr(x, "exponential")

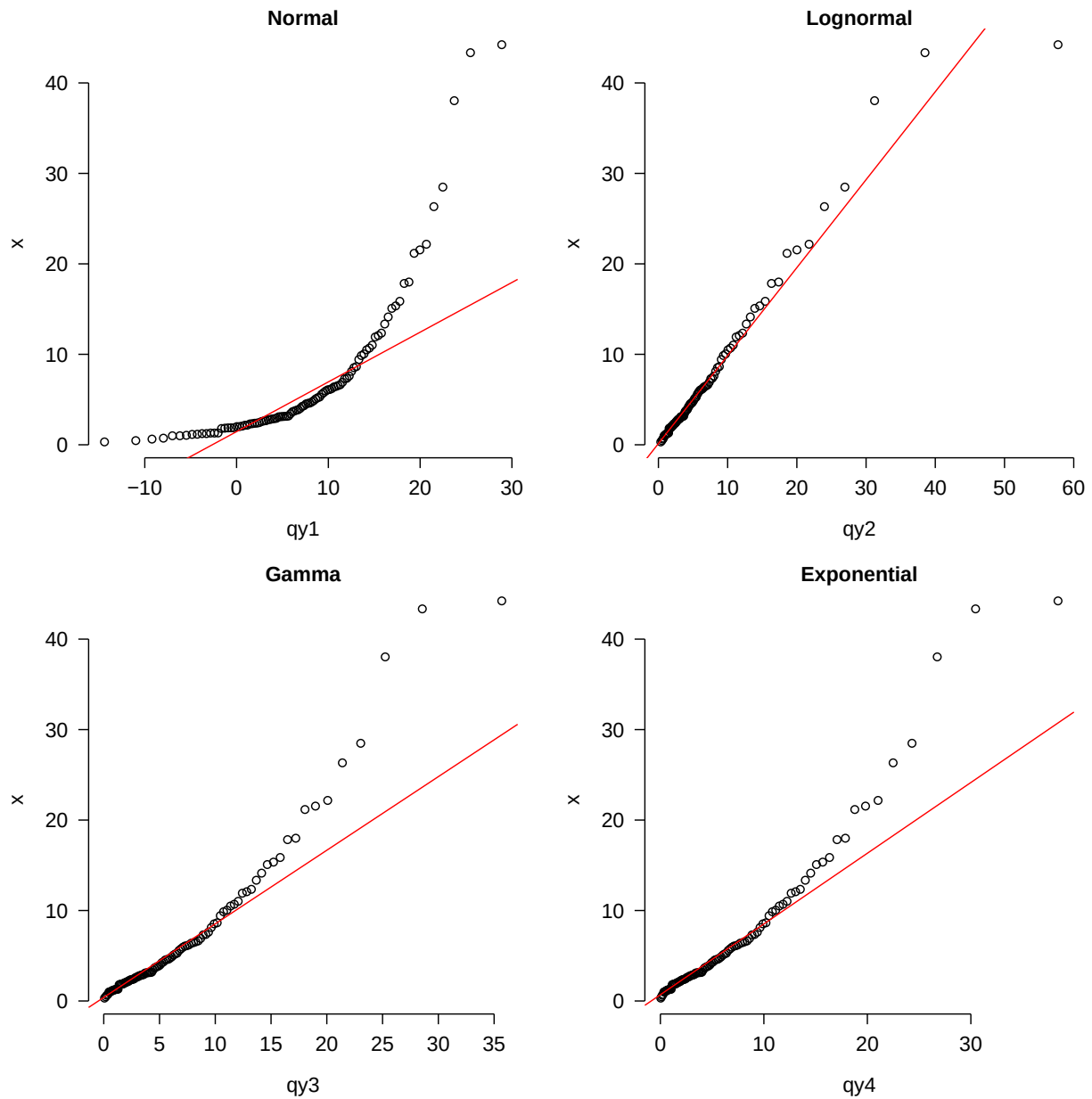
# quantiles of reference distribution
qx <- ppoints(n)

# define functions to draw from each reference distribution
dfun1 <- function(p){qnorm(p, f1$estimate[1], f1$estimate[2])}
dfun2 <- function(p){qlnorm(p, f2$estimate[1], f2$estimate[2])}
dfun3 <- function(p){qgamma(p, shape=f3$estimate[1],
                           rate=f3$estimate[2])}
dfun4 <- function(p){qexp(p, f4$estimate)}

# get values at each reference quantile
qy1 <- dfun1(qx)
qy2 <- dfun2(qx)
qy3 <- dfun3(qx)
qy4 <- dfun4(qx)

# make plot
par(mfrow=c(2,2), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, lend=1,
    cex.axis=1.2, cex.lab=1.2, bty="n")
qqplot(qy1, x, main="Normal")
qqline(x, distribution=dfun1, col="red")
qqplot(qy2, x, main="Lognormal")
qqline(x, distribution=dfun2, col="red")
qqplot(qy3, x, main="Gamma")
qqline(x, distribution=dfun3, col="red")
qqplot(qy4, x, main="Exponential")
qqline(x, distribution=dfun4, col="red")

```



The figure shows that the normal is clearly inappropriate because the points do not fall on the line. The gamma and exponential are okay over most of the domain of x , but depart for large values. The lognormal plot (top right) shows the best match between the data and the reference distribution.

It is important to keep in mind that the comparisons we've made here are of idealized cases. Real data are often much messier and noisier than the distributions used in these examples. Sometimes real data are best characterized by mixture distributions for which there are no straightforward ways to estimate parameters. Still, the steps outlined above for estimating distributional parameters and comparing to idealized distributions are important first steps in data analysis.

10.4 Formal tests for distributions

10.4.1 Tests for normality

The normal distribution is so important for so many statistical methods that people have developed tests to formally determine whether data come from a normal distribution. The most common test for normality is the **Shapiro-Wilk test**. The Shapiro-Wilk test tests the null hypothesis that a sample $x_1, \dots, x_n$ comes from a normally distributed population. The test statistic W is calculated as:

$$W = \frac{\left(\sum_{i=1}^n a_i x_{(i)}\right)^2}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

where $\bar{x}$ is the sample mean, $x_{i\sim}$ is the i^{th} order statistic, and a_i is a coefficient related to expected values of *i.i.d.* normal variables. When the null hypothesis of a Shapiro-Wilk test is rejected, then there is evidence that the data are not normally distributed. The code below shows some examples of the Shapiro-Wilk test in R.

```
set.seed(123)
x1 <- rnorm(20)
shapiro.test(x1)
##
##  Shapiro-Wilk normality test
##
## data:  x1
## W = 0.96858, p-value = 0.7247
```

```
x2 <- rpois(20, 10)
shapiro.test(x2)
```

Shapiro-Wilk normality test

```
data:  x2
W = 0.90741, p-value = 0.0569
```

```
x3 <- runif(20)
shapiro.test(x3)
```

Shapiro-Wilk normality test

```
data:  x3
W = 0.9316, p-value = 0.1658
```

```
x4 <- runif(100)
shapiro.test(x4)
```

Shapiro-Wilk normality test

```
data:  x4
W = 0.96392, p-value = 0.007723
```

Interestingly, the Shapiro-Wilk test turns out to be very powerful. That is, it can detect very small differences from the null hypothesis (i.e., normality) if the sample size is large enough. So, if you have data that you suspect are normal, but the Shapiro-Wilk test says otherwise, you need to visually inspect the data to determine how serious the departure from normality is.

10.4.2 Kolmogorov-Smirnov tests

The **Kolmogorov-Smirnov (KS) test** is a non-parametric test that compares a dataset to a distribution. Unlike the Shapiro-Wilk test, which only tests for normality, the KS test can compare data to any distribution. The KS test is useful because it is sensitive to differences in both location (i.e., mean) and shape. The example below illustrates the use of the KS test and plots the distributions.

```
n <- 1e2
x1 <- rnorm(n, 10, 2)
x2 <- rnorm(n, 10, 2)
x3 <- rnorm(n, 14, 2)
x4 <- rnorm(n, 14, 20)
x5 <- rpois(n, 14)

# run KS tests:
ks.test(x1, x2)
```

Asymptotic two-sample Kolmogorov-Smirnov test

```
data:  x1 and x2
D = 0.1, p-value = 0.6994
alternative hypothesis: two-sided
```

```
ks.test(x1, x3)
```

Asymptotic two-sample Kolmogorov-Smirnov test

```
data:  x1 and x3
D = 0.66, p-value < 2.2e-16
alternative hypothesis: two-sided
```

```
ks.test(x3, x4)
```

Asymptotic two-sample Kolmogorov-Smirnov test

```
data:  x3 and x4
D = 0.41, p-value = 1.001e-07
alternative hypothesis: two-sided
```

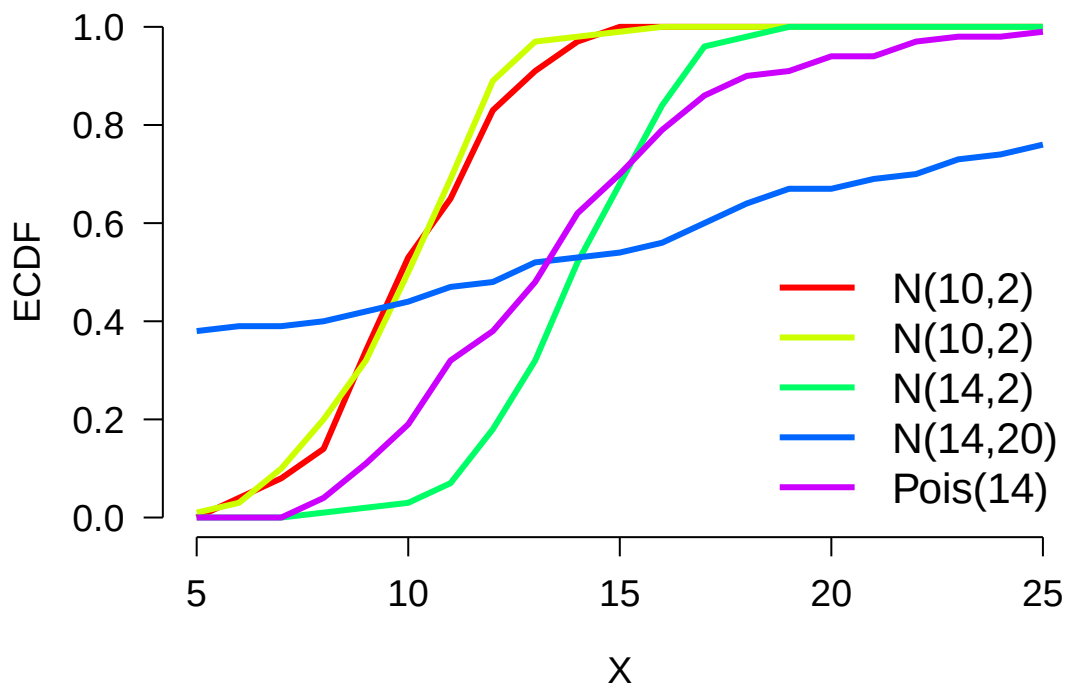
```
ks.test(x3, x5)
```

Asymptotic two-sample Kolmogorov-Smirnov test

```
data:  x3 and x5
D = 0.25, p-value = 0.003861
alternative hypothesis: two-sided
```

```
# make a plot
xx <- 5:25
y1 <- ecdf(x1)(xx)
y2 <- ecdf(x2)(xx)
y3 <- ecdf(x3)(xx)
y4 <- ecdf(x4)(xx)
y5 <- ecdf(x5)(xx)

cols <- rainbow(5)
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", las=1, lend=1,
    cex.axis=1.2, cex.lab=1.2)
plot(xx, y1, type="l", lwd=3,
     xlab="X", ylab="ECDF", col=cols[1])
points(xx, y2, type="l", lwd=3, col=cols[2])
points(xx, y3, type="l", lwd=3, col=cols[3])
points(xx, y4, type="l", lwd=3, col=cols[4])
points(xx, y5, type="l", lwd=3, col=cols[5])
legend("bottomright",
     legend=c("N(10,2)", "N(10,2)",
              "N(14,2)", "N(14,20)", "Pois(14)"),
     lwd=3, col=cols, bty="n", cex=1.3)
```



The KS test results show that distributions `x1` and `x2` are not different ($p > 0.05$). This makes sense because they are both normal distributions with the same mean and SD. Distributions `x1` and `x3` are different because they have different means (10 vs. 14). Distributions `x3` and `x4` were different because they had different SD (2 vs. 20). Finally, distributions `x3` and `x5` were different because they were different distributions with different shapes (normal vs. Poisson).

The KS test can be useful when you want to see how well your data matches up with their theoretical distribution. Consider the example of a lognormally distributed response below. If we suspect that the data come from a lognormal distribution or a gamma distribution, we can use the function `fitdistr()` to estimate the parameters of a lognormal and a gamma distribution that are most likely to produce those data. Then, we use a KS test to compare the data to the idealized candidates.

```
# random number seed
set.seed(123)

# generate data
x1 <- rlnorm(100, 2, 2)

# estimate moments
fit1 <- fitdistr(x1, "lognormal")
fit2 <- fitdistr(x1, "gamma")
```

Notice that the parameters from `fitdistr()` are supplied as arguments to `ks.test()`, following the name of the distribution we want to compare to. If parameters are not supplied, then R will perform the

KS test using the default parameter values (e.g., `meanlog=0` and `sdlog=1` for `plnorm()`; none for `pgamma()`), which is not what we want.

```
ks.test(x1, "plnorm", fit1$estimate[1], fit1$estimate[2])
```

Asymptotic one-sample Kolmogorov-Smirnov test

```
data: x1
D = 0.058719, p-value = 0.8808
alternative hypothesis: two-sided
```

The test shows that we cannot conclude that `x1` is different from a lognormal distribution.

```
ks.test(x1, "pgamma", shape=fit2$estimate[1],
        rate=fit2$estimate[2])
```

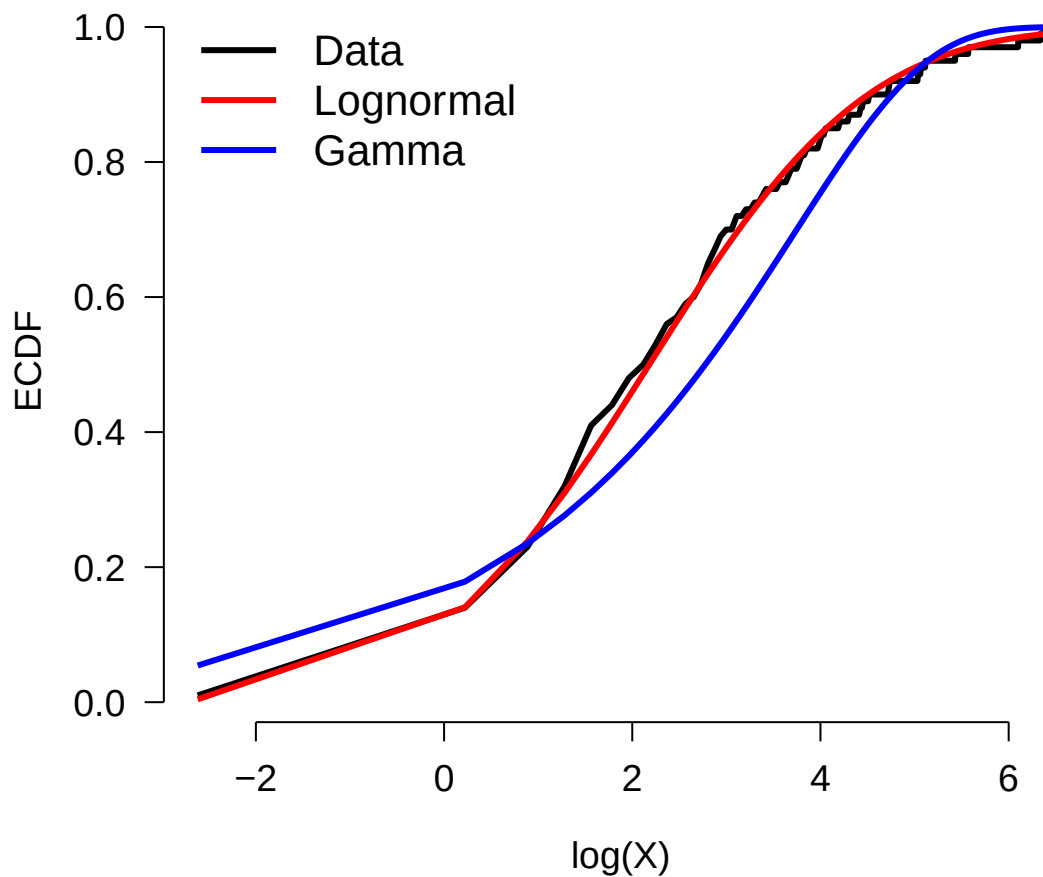
Asymptotic one-sample Kolmogorov-Smirnov test

```
data: x1
D = 0.16211, p-value = 0.01043
alternative hypothesis: two-sided
```

The test shows that `x1` is likely different from a gamma distribution.

```
# plot x1 and its theoretical distribution
xx <- seq(min(x1), max(x1), length=500)
y1 <- ecdf(x1)(xx)
y2 <- plnorm(xx, fit1$estimate[1], fit1$estimate[2])
y3 <- pgamma(xx, shape=fit2$estimate[1], rate=fit2$estimate[2])

par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1), bty="n",
    las=1, lend=1, cex.axis=1.2, cex.lab=1.2)
plot(log(xx), y1, type="l", lwd=3, xlab="log(X)", ylab="ECDF")
points(log(xx), y2, type="l", lwd=3, col="red")
points(log(xx), y3, type="l", lwd=3, col="blue")
legend("topleft",
      legend=c("Data", "Lognormal", "Gamma"),
      lwd=3,
      col=c("black", "red", "blue"),
      bty="n",
      cex=1.3)
```



While neither the lognormal distribution nor the gamma distribution are awful at representing the data, the KS tests suggest that the simulated data $\times 1$ come from a lognormal distribution, not a gamma distribution. This is because the KS test was non-significant when comparing the data to the lognormal, and significant when comparing the data to the gamma distribution. The test comparing the data to the gamma distribution was, however, close to non-significant ($p \approx 0.01$). See what happens if you change the random number seed and re-run the random generation and the test.

Chapter 11

Data transformation and rescaling

11.1 Data transformations

Data transformation is the process of changing the values in your data according to a function. This is usually done to make the data better conform to the assumptions of a statistical test, or to make graphs easier to interpret. Transformation is done to make patterns easier to detect. In many fields transformations may be standard practice, and with good reason. Just make sure that you understand what the transform function is doing, and most importantly, why you are transforming your data.

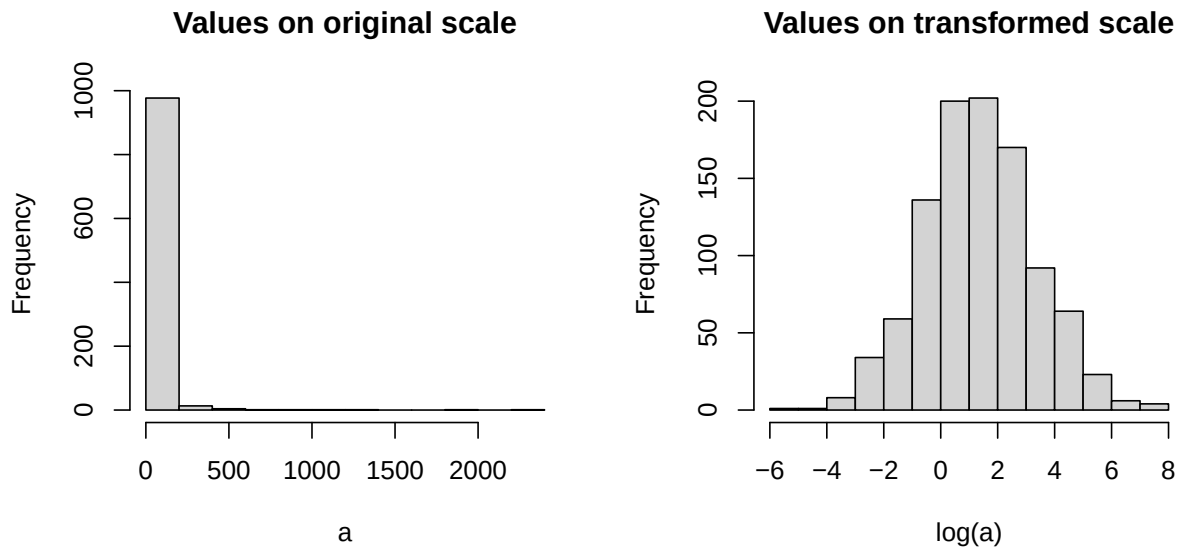
11.2 Why transform?

Data transformation is the process of applying a function to your data to change their values. This is usually done so that the data better conform to the assumptions of a statistical test, or to make graphs of your data more interpretable. The functions that transform data are usually called “transforms”. There are many transforms out there, but only a few are very common in biology. All convert values on the **original scale** to values on the **transformed scale**. Variables often have different properties after a transformation, such as normality, or a different domain, or a different distribution shape. A function for transforming data must be both **deterministic** and **invertible**:

- **Deterministic:** the function always returns the same transformed value for a given input.
- **Invertible:** values on the transformed scale can be converted unambiguously to values on the original scale.

Data transformations are commonly used to make data fit certain distributions. For example, consider the figure below:

```
a <- rlnorm(1000, 1.5, 2)
par(mfrow=c(1,2))
hist(a, main="Values on original scale")
hist(log(a), main="Values on transformed scale")
```



The left panel shows data with mostly small values and a strong positive skew. The right panel shows the same data on the log scale. Notice how the log-transformed values look much more like a normal distribution. This is very important for many statistical methods, which require normality in the residuals. The log-transformed plot also lets us see more details about the distribution that are obscured on the original scale.

11.3 Transforms vs. link functions

In **generalized linear models (GLM)** the response variable is modeled on a scale defined by a **link function**. That link function can look like a transform, and is often confused with a transform, but there is an important difference.

- A **transform** changes the observed response variables themselves: $Y \rightarrow g(Y)$, and the model is fit to the transformed observations.
- A **link function** changes the scale on which the expected value of the response is modeled: $g(E[Y]) = g(\mu)$.

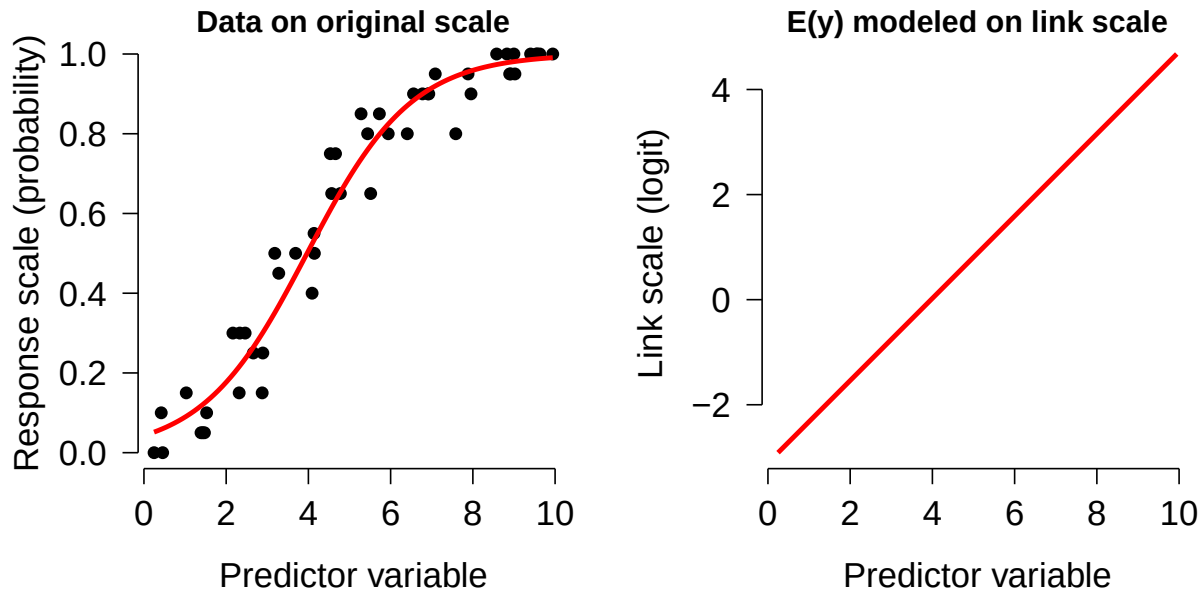
For example, in a binomial GLM with a logit link,

$$\text{logit}(p) = X\beta$$

you are not literally taking the logit of observed proportions and fitting a linear regression to those transformed values. The observations remain binomial, and their variance follows a binomial distribution.

Thus, fitting `lm(log(y) ~ x)` actually transforms the observed values of y , whereas fitting `glm(y ~ x, family=poisson(link="log"))` does not take the logarithm of the observed values of y .

The figure below shows an example of this: the response variable on its original scale (left panel), a proportion, is analyzed on the logit scale (right panel). The deterministic part of the model (the relationship between the predictors and the response) is linear on the logit scale, while the stochastic part (the variance) still follows a binomial distribution on the original scale.



Let's look at this in another context, because this distinction is important. **A GLM link function is not a transformation of the response variable.** When a response is transformed before fitting a linear model, the transformed values themselves are modeled, including their residual variation. In a GLM, the response retains its specified probability distribution on its original scale; the link function instead relates the expected value of that distribution to the linear predictor. The equations below illustrate the difference using two models that both involve a logarithm.

Linear model (LM) with transformation

In a linear model on transformed data, both the deterministic part of the model ($\eta_i = X\beta$) and the variance ($\sigma_{residual}^2$) are on the transformed scale.

$$y_i = e^{\eta_i}$$

$$\eta_i = X\beta + \varepsilon_i$$

$$\varepsilon_i \sim \text{Normal}(0, \sigma_{\text{residual}}^2)$$

Generalized linear model (GLM) with link function

In a GLM with a link function, the deterministic part of the model ($\eta_i = X\beta$), while the variance is on the original scale.

$$y_i \sim \text{Normal}(z_i, \sigma_{\text{residual}}^2)$$

$$\log(z_i) = \eta_i$$

$$\eta_i = X\beta$$

Notice where the stochasticity appears. In the transformed LM, normally distributed residual variation is added to $\log(y_i)$. In the GLM, observations y_i are drawn from a normal distribution with mean μ_i , variance σ^2 , and the predictors are expressed on the log scale. Thus, `lm(log(y) ~ x)` and `glm(y ~ x, family=gaussian(link="log"))` are not equivalent, even though they both involve logarithms.

11.4 Common transformations

11.4.1 Log transformation

The **log-transform** is one of the most common transforms in biology. It is simply the **logarithm of the values**. Recall that the logarithm is the inverse of exponentiation: if $a = b^c$, then $\log_b(a) = c$. For example, $\log_2(8) = 3$ because $2^3 = 8$. Most biologists use either the natural logarithm ($\log_e$, $\ln$ or just $\log$) or the base-10 logarithm ($\log_{10}$). The choice of base is not that important, because any log-transform will accomplish the same effect. The default in math and statistics (and in R) is the natural logarithm. The **common** or base-10 logarithm $\log_{10}$ is also very common in biology and engineering. Whatever base you use, just be consistent and make it clear to your audience what you did¹. It is depressingly common to be unable to use values from old papers because they didn't specify which base logarithm they used.

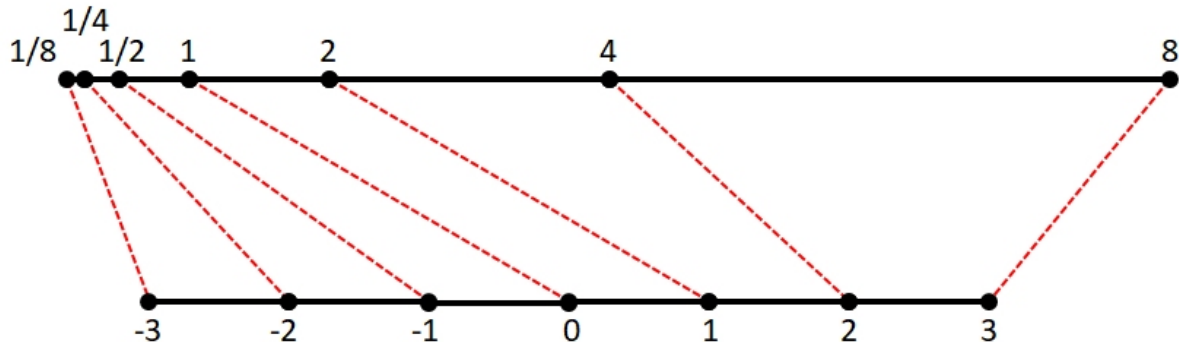
There are usually two reasons to use a log-transform: **normalization** (i.e., variance stabilization) and **positivity**. As **seen in the last module**, many variables that appear non-normal turn out to be normally distributed on a logarithmic scale. This often occurs when something about the values is multiplicative in nature. For

¹And whatever you do, don't call the natural logarithm "lawn".

example, body sizes, population sizes, and bank account balances are often log-normally distributed. This is because these quantities tend to grow at a rate that is proportional to their current size. Transforming to the logarithmic scale changes multiplicative changes to additive changes.

The figure below shows how values separated by a factor of 2 on the linear scale map to values separated by 1 (i.e., 20) on the $\log_2$ scale. Values separated by a common *scaling* factor on the original scale are mapped to values separated by a common *additive* constant on the log scale².

Linear scale



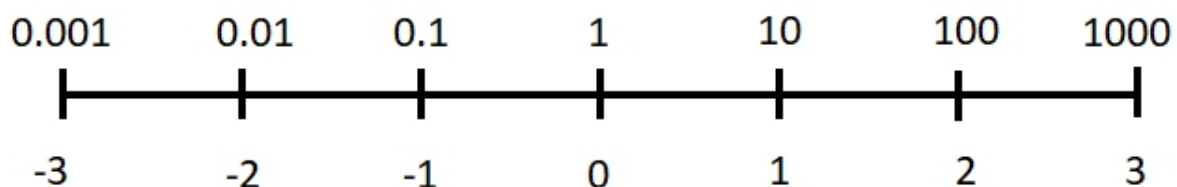
$\log_2$ scale

{alt-

text: "Number line showing how multiplicative changes on a linear scale (top) become additive changes on a logarithmic scale (bottom). In this example, one-eighth, one-quarter, one-half, one, two, four, and eight are mapped to their base 2 logarithms: -3, -2, -1, 0, 1, 2, and 3. Each step on the linear scale multiplies by 2, but each step on the logarithmic scale adds 1."}

As a side effect of this normalization, log-transformation allows researchers to work with variables that vary over several orders of magnitude. This is because scalar multiplication on the untransformed scale is the same as addition on the log-transformed scale³. The figure below shows how values ranging from 0.001 to 1000 (spanning 6 orders of magnitude) can be managed on the log-transformed scale.

Linear scale



$\log_{10}$ scale

{alt-

²This video illustrates the usefulness of logarithms in statistics **and** has a nice theme song.

³Remember that exponential functions relate addition and multiplication. This is how exponentiation is extended to non-integer powers: $X^a X^b = X^{a+b}$. And, in some sense the *definition* of exponentiation.

text: “Number line showing how multiplicative changes on a linear scale (top) become additive changes on a logarithmic scale (bottom). In this example, 0.001, 0.01, 0.1, 1, 10, 100, and 1000 are mapped to their base 10 logarithms: -3, -2, -1, 0, 1, 2, and 3. Each step on the linear scale multiplies by 10, but each step on the logarithmic scale adds 1.”}

The second reason that the log-transform is commonly used is to ensure that values are always positive. One of the properties of exponents function is that for any positive base, raising that base to a real power will always return a positive result. For this reason, it is common practice to analyze variables with a positive domain either after a log transform or with a log link function. Such variables include lengths, masses, and counts, which must be positive or non-negative. Log transformation can ensure that your analysis does not predict impossibilities like negative masses.

Values in a data frame can be log-transformed with the `log()` (natural log) or `log10()` ($\log_{10}$) function. Other bases can be used via arguments to the `log()` function (e.g., `log(8, base=2)`). It is usually a good idea to keep the transformed values in their own variable, rather than overwriting the original values. This way you can quickly access the original values without having to back-transform. It is often useful to have the original values around for making figures. Keeping the original values also means you don't have to remember whether a variable or object contains the original or transformed values.

```
# spare copy:
x <- iris

# natural log transform:
x$log.petal.len <- log(x$Petal.Length)

# log base 10 transform:
x$log.petal.wid <- log10(x$Petal.Width)
```

You can also log-transform many variables at once using the function `apply()`. In a data frame this means apply-ing the `log()` function to multiple columns (margin 2).

```
# load the dataset from package MASS
data(crabs, package="MASS")

# make a spare copy
x <- crabs
head(x)
```

	sp	sex	index	FL	RW	CL	CW	BD
1	B	M	1	8.1	6.7	16.1	19.0	7.0
2	B	M	2	8.8	7.7	18.1	20.8	7.4
3	B	M	3	9.2	7.8	19.0	22.4	7.7
4	B	M	4	9.6	7.9	20.1	23.1	8.2

```
5 B M 5 9.8 8.0 20.3 23.0 8.2
6 B M 6 10.8 9.0 23.0 26.5 9.8
```

```
# overwrite (transform in place)
# (ok but can cause headaches later)
x[,4:8] <- apply(x[,4:8], 2, log)
head(x)
```

```
sp sex index FL RW CL CW BD
1 B M 1 2.091864 1.902108 2.778819 2.944439 1.945910
2 B M 2 2.174752 2.041220 2.895912 3.034953 2.001480
3 B M 3 2.219203 2.054124 2.944439 3.109061 2.041220
4 B M 4 2.261763 2.066863 3.000720 3.139833 2.104134
5 B M 5 2.282382 2.079442 3.010621 3.135494 2.104134
6 B M 6 2.379546 2.197225 3.135494 3.277145 2.282382
```

```
# better way:
# make new variables and
# add to original data frame (better)
x <- crabs
z <- apply(x[,4:8], 2, log)
z <- as.data.frame(z)
names(z) <- paste(names(z), "log", sep=".")
x <- cbind(x, z)
head(x)
```

```
sp sex index FL RW CL CW BD FL.log RW.log CL.log CW.log
1 B M 1 8.1 6.7 16.1 19.0 7.0 2.091864 1.902108 2.778819 2.944439
2 B M 2 8.8 7.7 18.1 20.8 7.4 2.174752 2.041220 2.895912 3.034953
3 B M 3 9.2 7.8 19.0 22.4 7.7 2.219203 2.054124 2.944439 3.109061
4 B M 4 9.6 7.9 20.1 23.1 8.2 2.261763 2.066863 3.000720 3.139833
5 B M 5 9.8 8.0 20.3 23.0 8.2 2.282382 2.079442 3.010621 3.135494
6 B M 6 10.8 9.0 23.0 26.5 9.8 2.379546 2.197225 3.135494 3.277145
BD.log
1 1.945910
2 2.001480
3 2.041220
4 2.104134
5 2.104134
6 2.282382
```

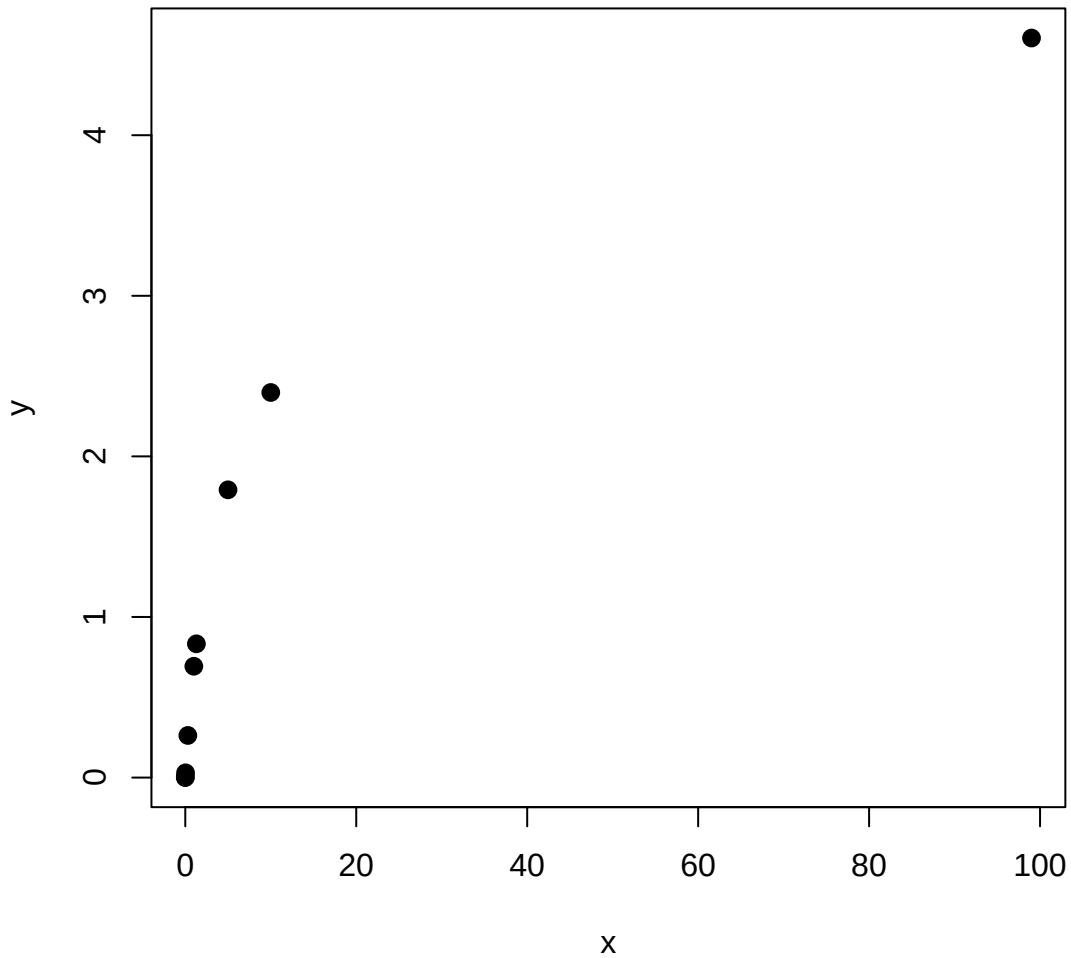
11.4.2 Log transforming with 0 values

One problem that comes up with log-transformation is what to do with values of 0. The logarithm of 0 is undefined, and returns $-\infty$ or `-Inf` in R. This is usually a problem with counts, rather than with measurements. The simplest solution is to add a small constant to the values before transforming. If the smallest nonzero value is 1, then it is easiest to just add 1 to the values:

$$y_i = \log(x_i + 1)$$

This works just fine if the smallest nonzero is approximately 1. Values between 0 and 1 can cause problems. Adding 1 to these values before log-transforming will tend to compress the lower end of the distribution and expand the upper end of the distribution. The R commands below show this:

```
x <- c(0, 0.003, 0.03, 0.3, 1, 1.3, 5, 10, 99)
y <- log(x+1)
plot(x, y, cex=1.3, pch=16)
```



If the smallest nonzero value differs from 1 by more than an order of magnitude (i.e., ≥ 10 or ≤ 0.01), then a slightly different procedure is called for. The transformation below generalizes the $\log(x + 1)$ method so as to result in 0 for 0 values and somewhat preserve the original orders of magnitude (McCune *et al.* 2002).

$$y_i = \log(x_i + d) - c$$

In this equation:

$$\begin{aligned}
 c &= \text{Trunc}(\log(\min(x))) \\
 d &= e^c \\
 \min(x) &= \text{smallest nonzero } x \\
 \text{Trunc}(x) &= \text{function that truncates } x \text{ to an integer}
 \end{aligned}$$

This can be implemented in R using the following custom function:

```
mgtrans <- function(x) {  
  minx <- min(x[x > 0])  
  cons <- trunc(log(minx))  
  y <- log(x + exp(cons)) - cons  
  return(y)  
}
```

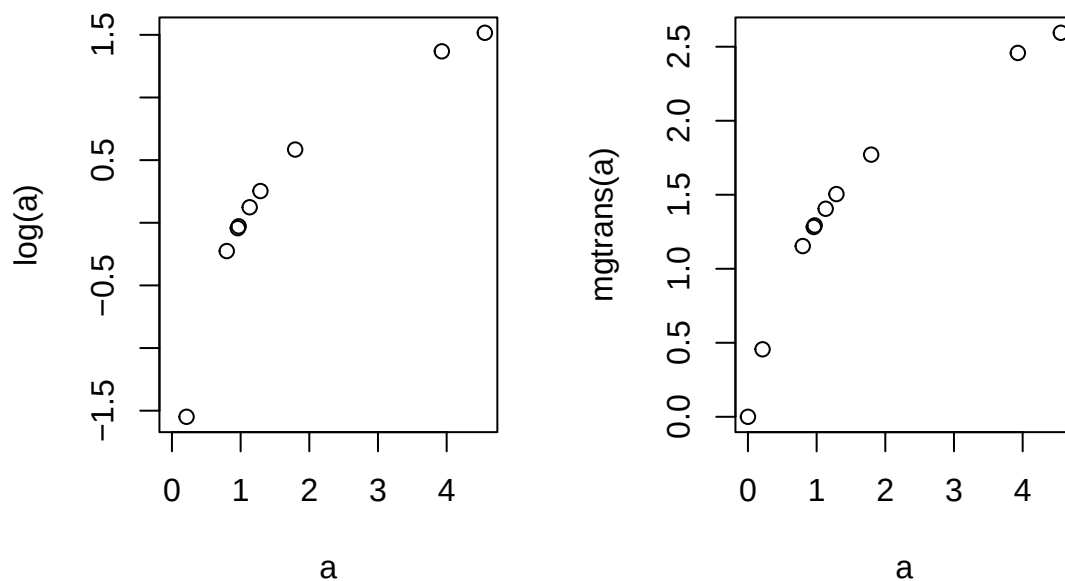
```
# example usage:  
a <- c(0, rlnorm(9))  
log(a)
```

```
[1]      -Inf  0.25331851 -0.02854676 -0.04287046  1.36860228 -0.22577099  
[7]  1.51647060 -1.54875280  0.58461375  0.12385424
```

```
mgtrans(a)
```

```
[1] 0.000000 1.504510 1.292473 1.282103 2.458088 1.153391 2.594120 0.455949  
[9] 1.771116 1.405285
```

```
# count the points:  
par(mfrow=c(1,2))  
plot(a, log(a))      # missing one!  
plot(a, mgtrans(a))
```

Note that if you use the McCune and Grace transform, you should keep the original values around for plotting because back-transforming can be tricky (if not impossible, if you don't know d and c). If you are going to back-transform other values, such as model predictions, you will need to know d and c .

The example below shows how the McCune and Grace transform can be advantageous. First, we create 1000 random values from a lognormal distribution. Then, randomly set 100 of the values to 0. Log-transforming the data to achieve normality comes at the cost of losing 10% of the values because $\log(0)$ is undefined (left panel). The $\log(x+1)$ transform retains all of the values, but distorts the shape of the distribution (middle panel) because many values are between 0 and 1. The McCune and Grace method (right panel) achieves normality (mostly) *and* retains all values.

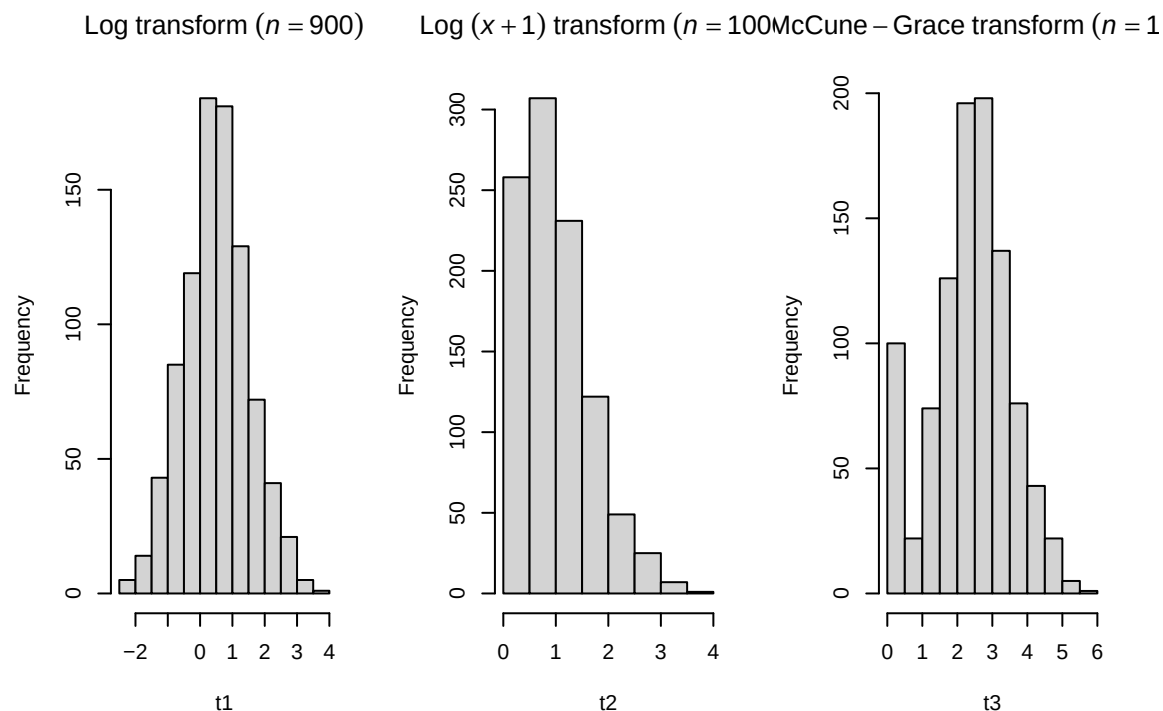
```
set.seed(123)
n <- 1e3
x <- rlnorm(n, 0.5, 1)
x[sample(1:n, 100, replace=FALSE)] <- 0

t1 <- log(x)
t2 <- log(x+1)
t3 <- mgtrans(x)

length(which(is.finite(t1)))
```

```
[1] 900
```

```
par(mfrow=c(1,3), cex.main=1.2)
hist(t1, main=expression(Log~transform~(italic(n)==900)))
hist(t2, main=expression(Log~(italic(x)+1)~transform~(italic(n)==1000)))
hist(t3, main=expression(McCune-Grace~transform~(italic(n)==1000)))
```



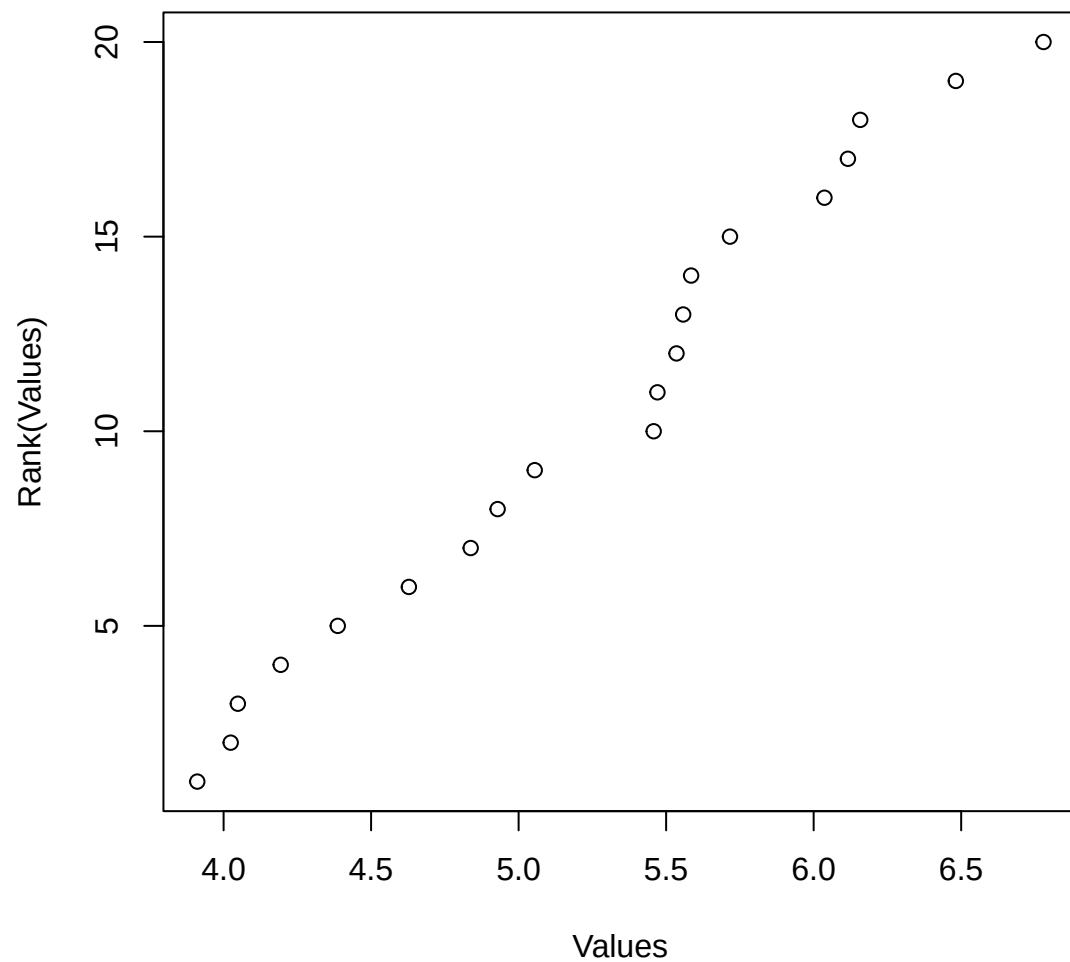
The spike in the right histogram comes from the fact that the McCune and Grace method maps values of 0 to 0.

11.4.3 Rank transformation

The **rank transform** replaces values with their ranks. The R function for this is `rank()`. Smaller values having smaller ranks and greater values having greater ranks. The smallest value has rank = 1, and the largest value has a rank equal to the number of values. That is, for any vector `x` `rank(x)[x==min(x)] = 1` and `rank(x)[x==max(x)] = length(x)`.

```
x <- rnorm(20, 5, 1)
y <- rank(x)

par(mfrow=c(1,1))
plot(x,y, xlab="Values", ylab="Rank(Values)")
```



Notice how the transformed variables y are evenly spaced while the original values are not. This can be useful if values are very “clumped” or “spread out”. Rank transforms are commonly used in nonparametric procedures. Note that the `rank()` function may return fractional or repeated ranks if there are ties:

```
# make some random values
x <- runif(10)

# force a tie
x[1] <- x[2]

# fractional ranks for ties:
rank(x)
```

```
[1] 1.5 1.5 4.0 6.0 7.0 9.0 5.0 10.0 8.0 3.0
```

```
# force a three-way tie:  
x[3] <- x[2]  
  
# repeated ranks:  
rank(x)
```

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

11.4.4 Logit and probit transformations

11.4.4.1 Logit transformation

The **logit transformation** and **probit transformation** are used to map **proportions** to the **real number line**. Often this changes proportions into something resembling a normal distribution. The **logit** is defined as the *logarithm of the odds ratio*. For any value x in the open interval⁴ $(0, 1)$, the logit is calculated as:

$$\text{logit}(x) = \log\left(\frac{x}{1-x}\right)$$

The inverse of the logit function is the **logistic function**.

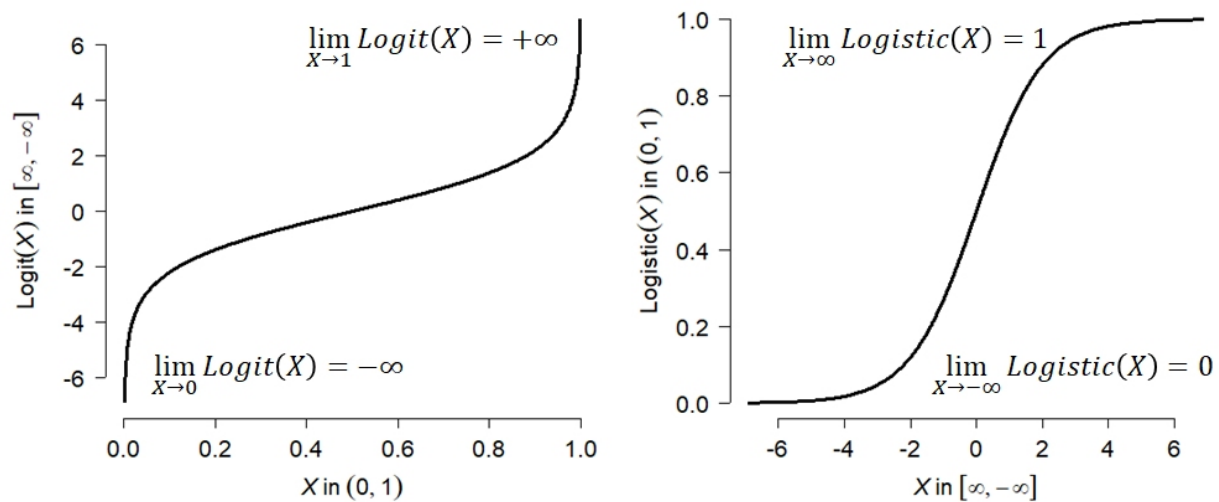
$$\text{logistic}(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}$$

If x is a probability or proportion, then $\text{logit}(x)$ is the logarithm of the odds ratio. For this reason, the logit is also called the “log-odds”. The logit function is also the inverse of the logistic function.

The figure below shows the relationship between the logit and logistic functions:

⁴An **open interval** is an interval (a, b) that contains values $> a$ and $< b$, but never a or b . Open intervals are denoted by parentheses $()$. Intervals that contain their limits are called **closed intervals**, denoted with brackets $[]$: $[a, b]$.

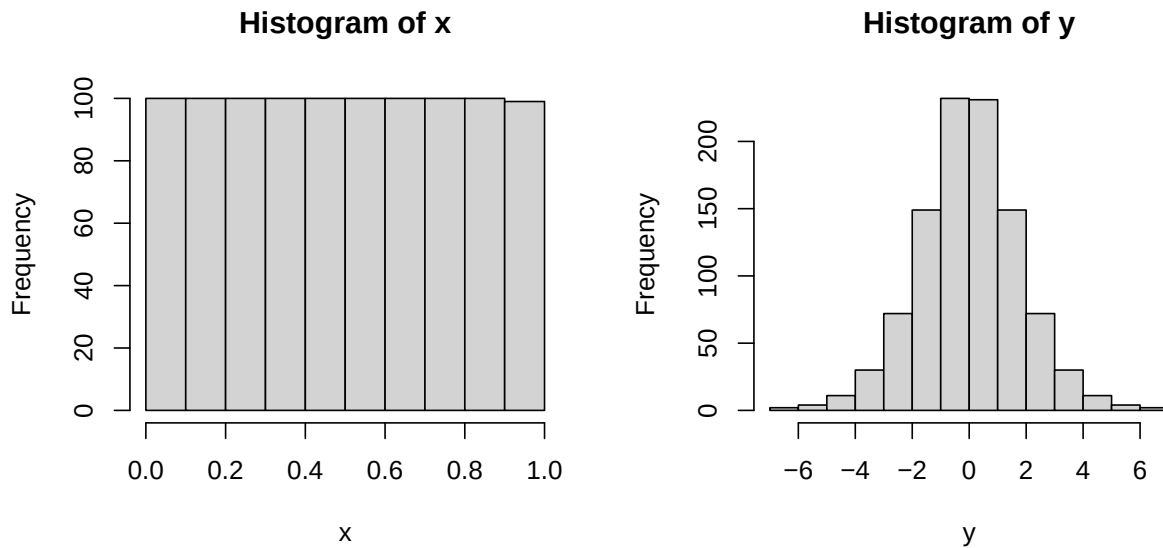
Figure 11.1: The logit and logistic functions. The logit function (left) maps probabilities in the interval $(0, 1)$ onto the unbounded interval $(-\infty, +\infty)$. The logistic function (right) is its inverse, mapping values from $(-\infty, +\infty)$ back onto $(0, 1)$.



The logit transformation has the effect of stretching out the ends of a distribution, while keeping the relationship between the original and transformed values roughly linear near the middle of the distribution. The logit transformation can be accomplished in R using the function `qlogis()`; its inverse, the logistic function, is `plogis()`.

```
x <- 1:999/1000
y <- qlogis(x)

# compare:
par(mfrow=c(1,2))
hist(x)
hist(y)
```



Why is this useful? The logistic function takes any real input and outputs a value between 0 and 1. This means that the logit function takes values between 0 and 1 and outputs a real number. The example below shows how this can be applied to analyze proportions:

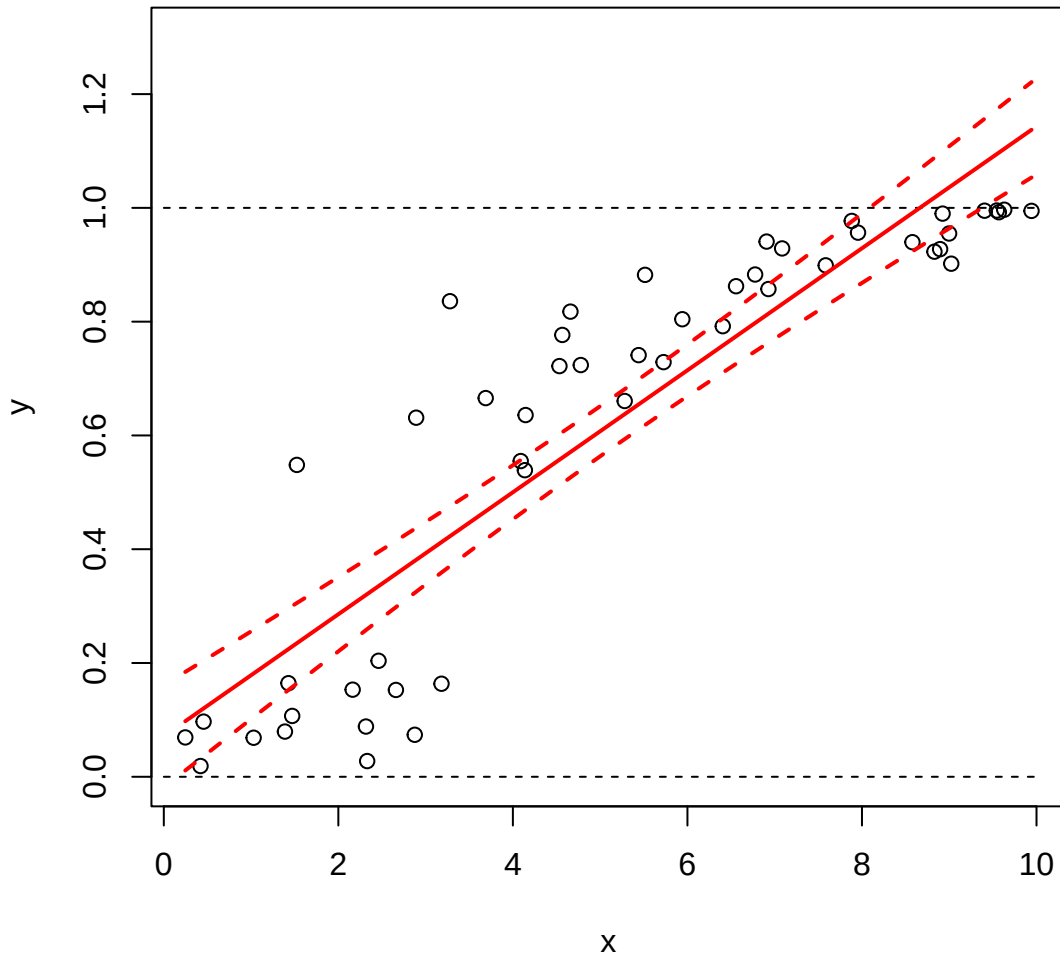
The first figure below shows how analyzing proportions with linear models leads to clearly inappropriate results. The predicted values of the model extend well outside the possible range of a probability.

```
set.seed(123)
n <- 50
x <- runif(n, 0, 10)
z <- -3 + 0.75*x + rnorm(n, 0, 1)
y <- plogis(z)

mod1 <- lm(y~x)
px <- seq(min(x), max(x), length=100)
pred <- predict(mod1, newdata=data.frame(x=px), se.fit=TRUE)
mn <- pred$fit
lo <- qnorm(0.025, mn, pred$se.fit)
up <- qnorm(0.975, mn, pred$se.fit)

par(mfrow=c(1,1))
plot(x, y, ylim=c(0, 1.3))
segments(0, 1, 10, 1, lty=2)
segments(0, 0, 10, 0, lty=2)
points(px, mn, type="l", lwd=2, col="red")
points(px, lo, type="l", lwd=2, lty=2, col="red")
```

```
points(px, up, type="l", lwd=2, lty=2, col="red")
```



The linear model predicts probabilities (y) less than 0 and greater than 1, so clearly it is inappropriate for analyzing probabilities. However, applying a logit transform to the data for analysis (and later back-transforming using the logistic function) can allow you to analyze proportions. Another (and sometimes better way) to accomplish this is to use a **GLM with binomial family and logit link**.

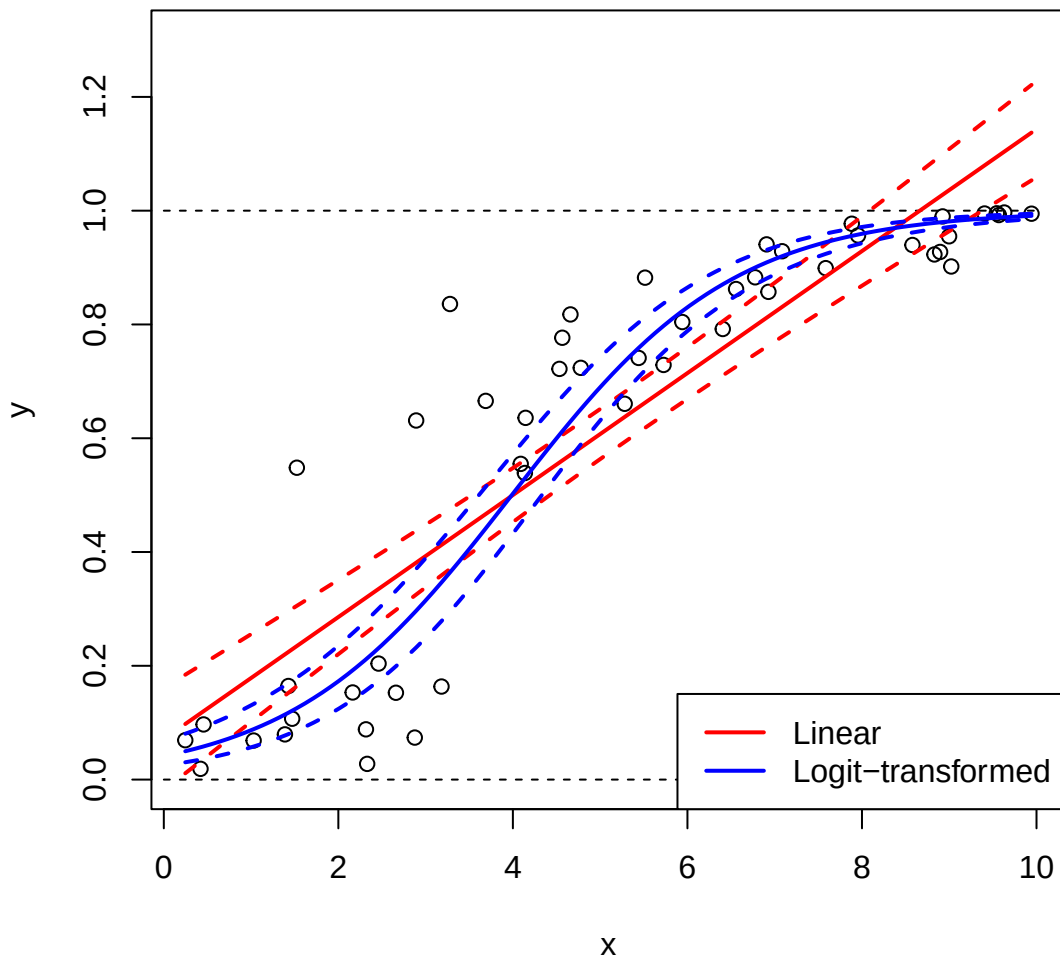
```
y2 <- qlogis(y)
mod2 <- lm(y2~x)
pred2 <- predict(mod2, newdata=data.frame(x=px), se.fit=TRUE)
mn2 <- pred2$fit
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)
```

```

# backtransform
mn2 <- plogis(mn2)
lo2 <- plogis(lo2)
up2 <- plogis(up2)

par(mfrow=c(1,1))
plot(x, y, ylim=c(0, 1.3))
segments(0, 1, 10, 1, lty=2)
segments(0, 0, 10, 0, lty=2)
points(px, mn, type="l", lwd=2, col="red")
points(px, lo, type="l", lwd=2, lty=2, col="red")
points(px, up, type="l", lwd=2, lty=2, col="red")
points(px, mn2, type="l", lwd=2, col="blue")
points(px, lo2, type="l", lwd=2, lty=2, col="blue")
points(px, up2, type="l", lwd=2, lty=2, col="blue")
legend("bottomright",
      legend=c("Linear", "Logit-transformed"),
      lwd=2, col=c("red", "blue"), bg="white")

```

The downside of the logit transform is that it is undefined at 0 and 1 (it approaches $-\infty$ and $+\infty$ as the inputs approach 0 and 1). One workaround is to add a small constant to avoid taking $\log(0)$ or dividing by 0 (McCune *et al.* 2002; Warton and Hui 2011). The latter reference suggests adding the minimum non-zero value. You could also censor the values to be arbitrarily close to 0 or 1: for example, set all values < 0.001 to 0.001 and all values > 0.999 to 0.999. This practice is sometimes called “stabilizing the logit” (Kéry 2010).

```
# change some values in y to 1 and 0
y[which.max(y)] <- 1
y[which.min(y)] <- 0

# logit transformed values include infinities!
qlogis(y)
```

```
[1] -2.5298619  3.7500756  0.2207000  2.4844936  5.3073196 -2.2318620
[7]  0.6657197  4.5882685  2.0138961  1.2461916  4.8648903  0.9539238
[13]  2.0198681  0.9887879 -2.6085359  3.0539803 -1.3622593      -Inf
[19]  1.6283614  5.3667394  2.5484363  1.7931407  1.3371457  5.2369884
[25]  1.8344244  2.5672970  1.0519484  1.4131947  0.5373003 -2.1224186
[31]      Inf  2.2184900  2.7649033  3.0898599 -2.5994558  0.9631093
[37]  2.1861231 -1.7101479 -1.6322178 -2.3345978 -1.6254712  0.5573073
[43]  0.1559367  0.6886083  0.1934203 -2.4499857 -3.5614131  1.5004569
[49] -1.7144060  2.7456992
```

```
# stabilize logit by setting max to 0.999 and min to 0.001
y <- pmin(0.999, y)
y <- pmax(0.001, y)
qlogis(y)
```

```
[1] -2.5298619  3.7500756  0.2207000  2.4844936  5.3073196 -2.2318620
[7]  0.6657197  4.5882685  2.0138961  1.2461916  4.8648903  0.9539238
[13]  2.0198681  0.9887879 -2.6085359  3.0539803 -1.3622593 -6.9067548
[19]  1.6283614  5.3667394  2.5484363  1.7931407  1.3371457  5.2369884
[25]  1.8344244  2.5672970  1.0519484  1.4131947  0.5373003 -2.1224186
[31]  6.9067548  2.2184900  2.7649033  3.0898599 -2.5994558  0.9631093
[37]  2.1861231 -1.7101479 -1.6322178 -2.3345978 -1.6254712  0.5573073
[43]  0.1559367  0.6886083  0.1934203 -2.4499857 -3.5614131  1.5004569
[49] -1.7144060  2.7456992
```

11.4.4.2 Probit transformation

Like the logit function, the probit function maps probabilities to the real number line. However, the probit function uses the CDF of the standard normal distribution instead of the logit function. The logit and the probit can be used in many of the same situations. Whereas the logit maps a probability using the log-odds, the probit maps a probability to the corresponding quantile of the standard normal distribution:

$$\text{probit}(x) = \Phi^{-1}(x)$$

where Φ is the CDF of the standard normal distribution (i.e., $Normal(\mu = 1, \sigma = 1)$). For example, a probability of 0.5 corresponds to a probit of 0, while probabilities of 0.025, and 0.975 correspond to probits of -1.96 and +1.96.

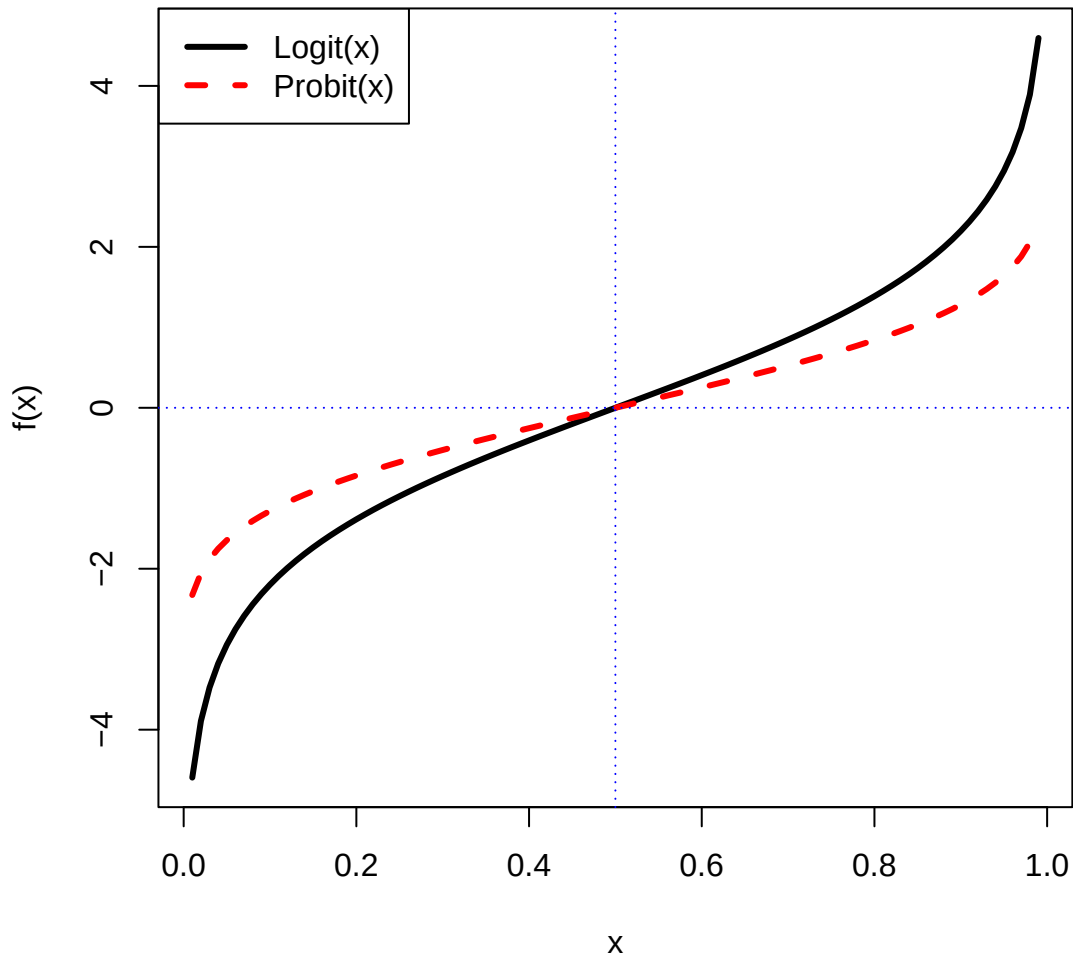
The R command for the probit function is `qnorm()`. As you might suspect from [the previous module](#), the inverse function is `pnorm()`.

```

x <- 1:99/100
x.log <- qlogis(x)
x.prb <- qnorm(x)

plot(x, x.log, type="l",
      lwd=3, xlab="x", ylab="f(x)")
points(x, x.prb, type="l", lwd=3,
        col="red", lty=2)
abline(h=0, lty=3, col="blue")
abline(v=0.5, lty=3, col="blue")
legend("topleft",
       legend=c("Logit(x)", "Probit(x)"),
       lwd=3, col=c("black", "red"), lty=1:2)

```



The figure above shows a logit and probit transform of probabilities from 0.01 to 0.99. The blue dashed lines show that both functions map $p = 0.05$ to 0. The logit and probit functions are similar near the center of the probability range but differ increasingly near 0 and 1. The logit approaches $\pm\infty$ more rapidly because the corresponding logistic distribution has heavier tails than the normal distribution. Despite this difference, logit and probit GLMs usually produce very similar fitted probabilities except when probabilities are extreme.

In practice, logit and probit links or transforms usually produce very similar fitted probabilities, and the choice between them often has little effect on the biological conclusions. The logit is particularly convenient because its coefficients can be interpreted using odds ratios (see Section 21.1.1). The probit can be useful when there is a reason to imagine an underlying normally distributed continuous variable that produces a binary outcome when it crosses a threshold. For example, an organism might die when an unobserved physiological stress exceeds some threshold, or an animal might choose one habitat when an underlying preference exceeds a threshold. This latent-variable interpretation is one reason probit models are used in some fields.

Probit models have historically been especially common in toxicology and dose-response studies. If individuals differ in their tolerance to a toxin and those tolerances are approximately normally distributed, the proportion responding at a given dose can be modeled using the normal CDF. This approach was historically known as probit analysis and has been widely used to estimate quantities such as LD50.

11.5 Uncommon transformations

11.5.1 Square-root transformation

Square-root transforms map values to their square roots. This is sometimes used as a variance-stabilizing transformation, similar to the log transform. Square-root transforms are useful when a response variable is proportional to the square of the explanatory variable, or when the data show heteroscedasticity (non-constant variance). Counts are sometimes analyzed using a square-root transform. The square-root transform is a special case of the **power transform** (where values are raised to a power) and a special case of the Box-Cox transform (see Section 11.5.3). The square-root function in R is `sqrt()`. There are several ways to accomplish a square-root transform in R:

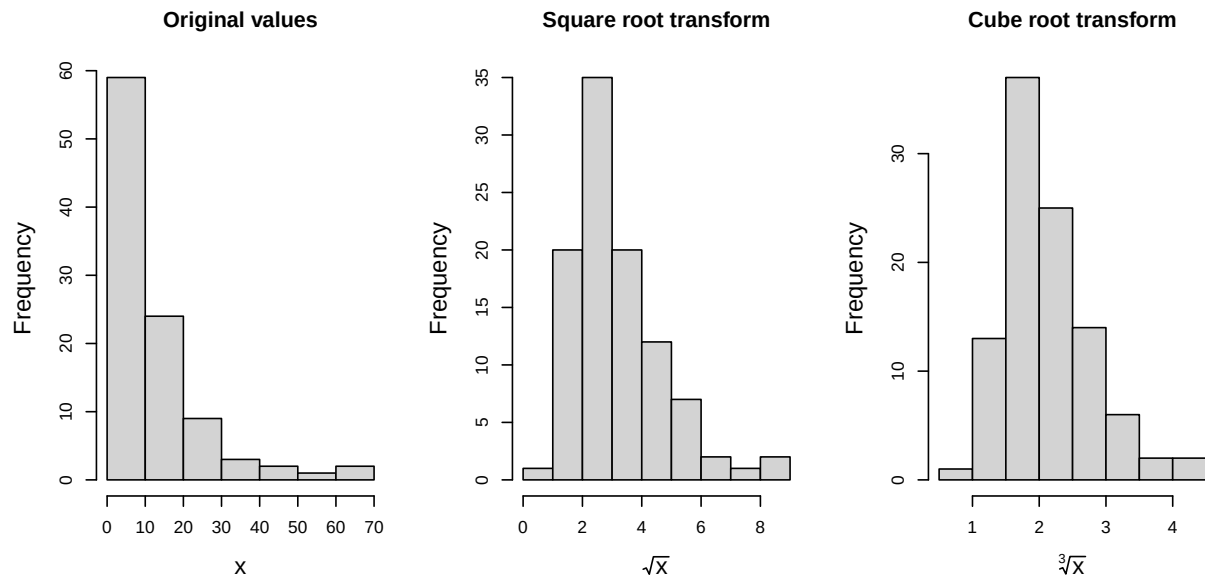
```
x <- rnorm(20, 5, 1)
y <- sqrt(x)
z <- x^0.5
```

Higher roots (cube root, fourth root, etc.) are sometimes used as transforms, but this is not very common.

```
w <- x^(1/3) # cube root
```

Here is an example of using the square root transform on right-skewed data:

```
set.seed(123)
x <- rlnorm(100, 2, 1)
par(mfrow=c(1,3), cex.lab=1.4)
hist(x, main="Original values")
hist(sqrt(x), main="Square root transform", xlab=expression(sqrt(x)))
hist(x^(1/3), main="Cube root transform", xlab=expression(root(x,3)))
```



Notice how the square- and cube-root transformed data appear much more normally-distributed than the original data.

```
# reset plot layout
par(mfrow=c(1,1))
```

11.5.2 Arcsine square root transformation

Another transformation historically used for proportional data is to use the **arcsine square root transform**, or **arcsine transform** for short:

$$y_i = \frac{2}{\pi} \arcsin(\sqrt{x_i})$$

Some sources omit the coefficient $2/\pi$; others do not. Including the $2/\pi$ rescales the function so that the transformed values range from 0 to 1 (McCune *et al.* 2002). Without this factor, the function maps probabilities to $[0, \pi/2]$ instead of $[0, 1]$.

For a binomial proportion, variance depends on the underlying probability p , with proportions near 0 or 1 having different variance than those near 0.5. The arcsine transform was used as a variance-stabilizing transformation to reduce the dependence of the variance on the mean.

Although historically common in biology, the arcsine square-root transform is generally not recommended when proportions arise from an identifiable binomial process. A binomial GLM (Section 21.2) models the mean-variance relationship directly while respecting the bounded nature of the response, and is usually preferable to transforming the proportions. The transformation has also been criticized for ecological proportional data more generally (Warton and Hui 2011). Thus, its main importance today may be in understanding older literature rather than as a default method for analyzing new data.

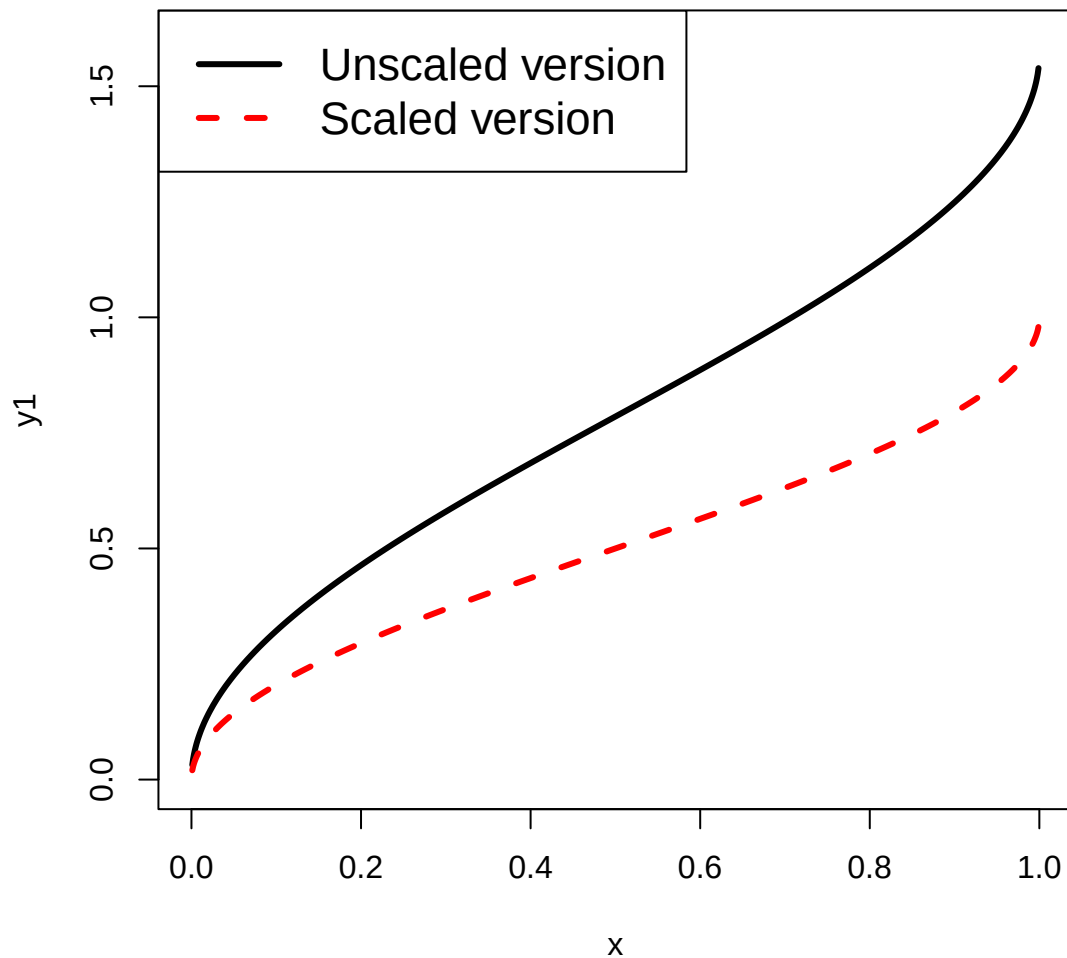
If you insist on using the arcsine transform, you can do it in R using `asin()` function. In the examples below, `asin()` is embedded inside custom functions that perform the other parts of the arcsine transform.

```
# unscaled version:
asin.trans1 <- function(p) {asin(sqrt(p))}

# scaled version:
asin.trans2 <- function(p) {2/pi*asin(sqrt(p))}

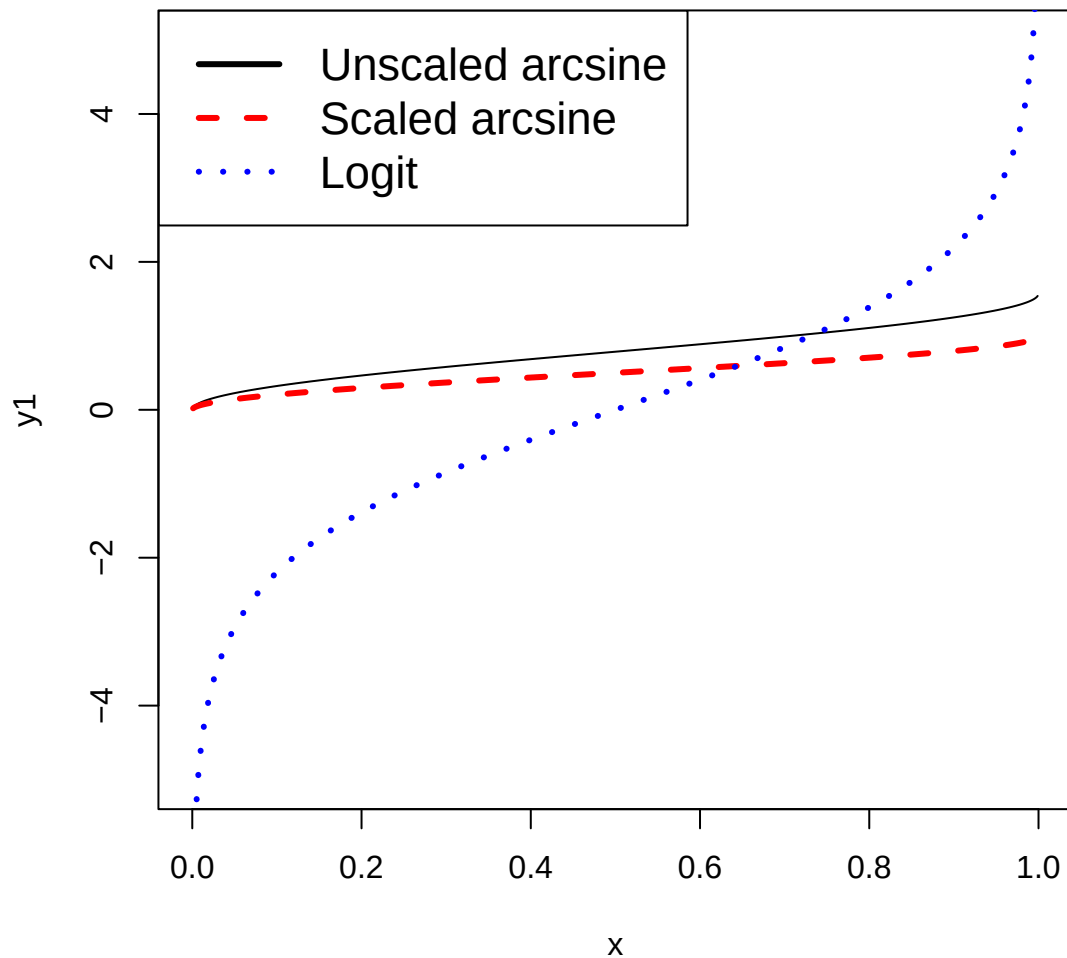
# example of use
x <- seq(0.001, 0.999, by=0.001)
y1 <- asin.trans1(x)
y2 <- asin.trans2(x)

# compare with and without 2/pi coefficient:
plot(x,y1, ylim=c(0, 1.6), type="l", lwd=3)
points(x, y2, col="red", type="l", lwd=3, lty=2)
legend("topleft",
      legend=c("Unscaled version", "Scaled version"),
      lwd=3, lty=1:2, col=c("black", "red"),
      cex=1.4)
```



The plot below compares the arcsine to the logit transformation for proportional data.

```
# compare to logit transform:
y3 <- qlogis(x)
plot(x,y1, ylim=c(-5, 5), xlim=c(0,1), type="l")
points(x, y2, col="red", type="l", lwd=3, lty=2)
points(x, y3, col="blue", type="l", lwd=3, lty=3)
legend("topleft",
      legend=c("Unscaled arcsine",
               "Scaled arcsine",
               "Logit"),
      lty=1:3, lwd=3, col=c("black", "red", "blue"), cex=1.4)
```

Unlike the logit, the arcsine transform remains bounded at $p = 0$ and $p = 1$. Consequently, it can transform observed proportions of exactly 0 or 1 without adjustment, whereas logit is undefined at those values.

11.5.3 Box-Cox transformation

The Box-Cox transformation is a family of power transformations used to identify a useful transformation of a positive response variable in a statistical model (Box and Cox 1964). Rather than choosing a transformation such as square root or logarithm by hand, the Box-Cox procedure estimates a parameter λ that determines the transformation.

$$y_i^{(\lambda)} = \begin{cases} \log(y_i), & \lambda = 0 \\ \frac{y_i^\lambda - 1}{\lambda}, & \lambda \neq 0 \end{cases}$$

Thus, Box-Cox includes many familiar transformations as special cases. A value of $\lambda = 1$ corresponds essentially to no transformation, $\lambda = 0.5$ to a square-root transformation, $\lambda = 0$ to a logarithmic transformation, and $\lambda = -1$ to a reciprocal transformation.

Table 11.1: Common values of λ and their approximate corresponding transformations in the Box-Cox family.

λ	Approximate transformation
2	y^2
1	y (no transformation)
0.5	$\sqrt{y}$
0	$\log(y)$
-0.5	$1/\sqrt{y}$
-1	$1/y$

Here the parameter λ is some constant that must be optimized for a given variable. When $\lambda = 0$, the transformation is defined as $y_i = \log(x_i)$.

Unlike the transformations considered above, Box-Cox is usually applied in the context of a statistical model. A model is fitted repeatedly using different values of λ , and the likelihood is calculated for each. The value of λ that maximizes the likelihood is the estimated optimal transformation.

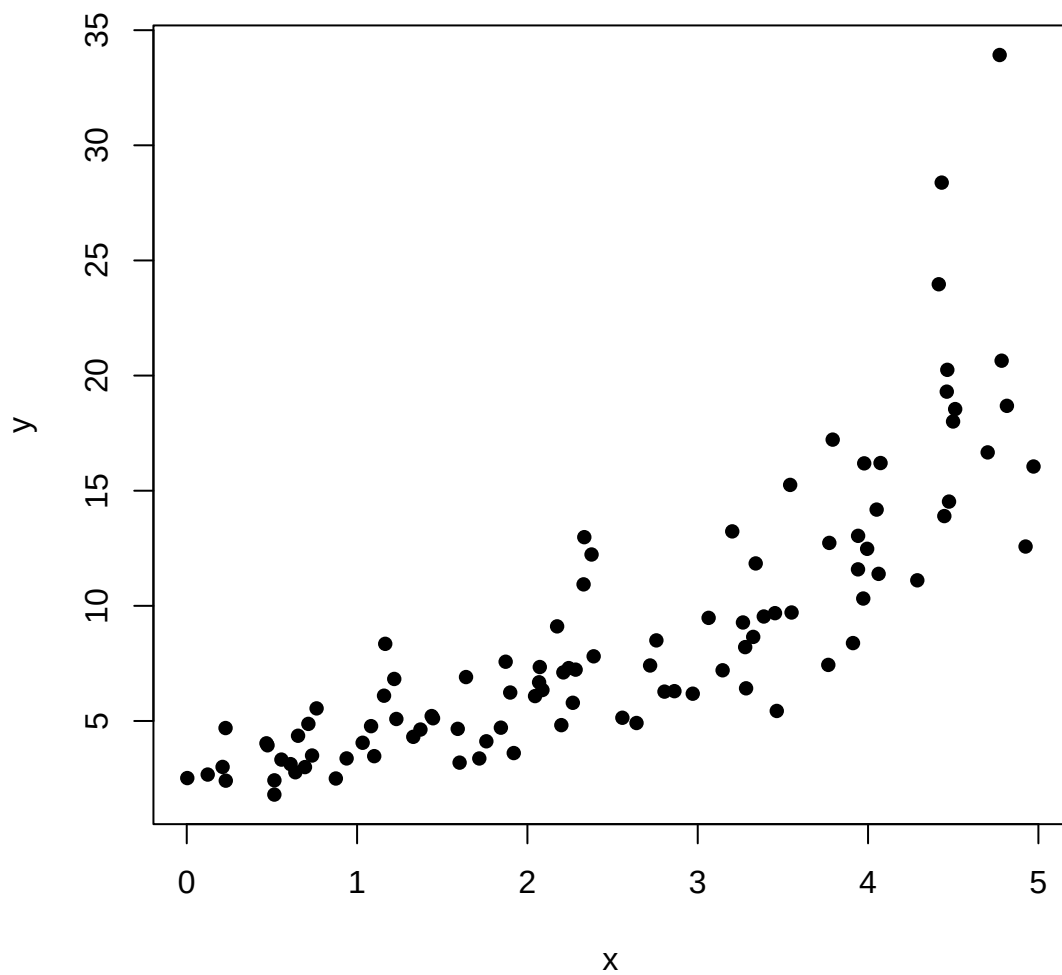
The Box-Cox transformation is available in several packages, but not base R. Here is an example using the functions in package MASS:

```
set.seed(123)

n <- 100
x <- runif(n, 0, 5)

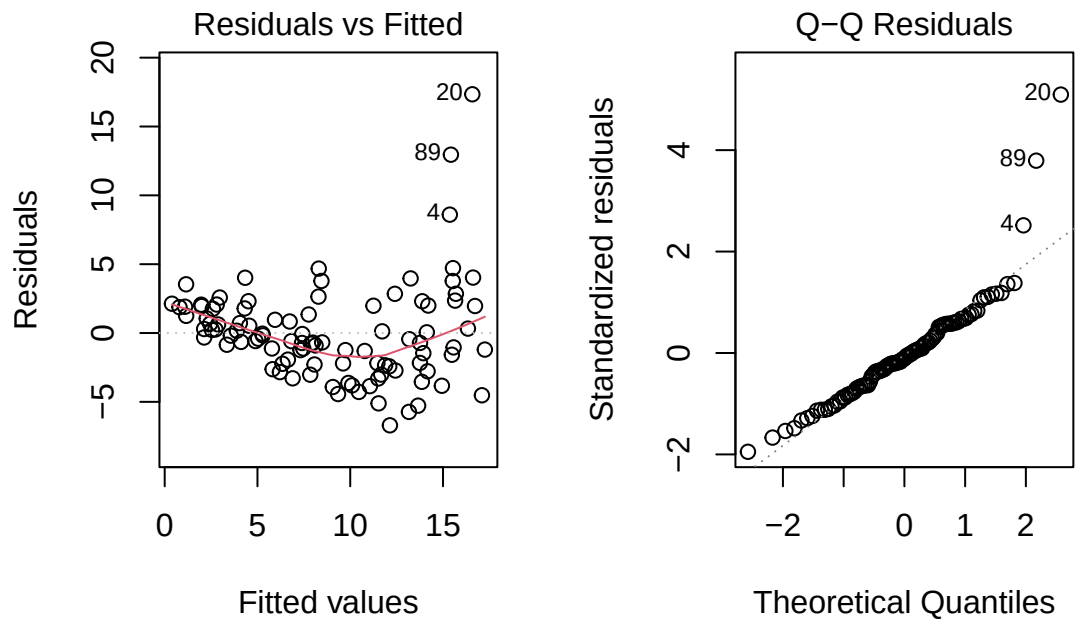
# relationship is linear with normal errors on log scale
logy <- 1 + 0.4*x + rnorm(n, 0, 0.3)
y <- exp(logy)

# but on linear scale, relationship appears nonlinear
plot(x, y, pch=16)
```



```
# fit a linear model
m1 <- lm(y ~ x)

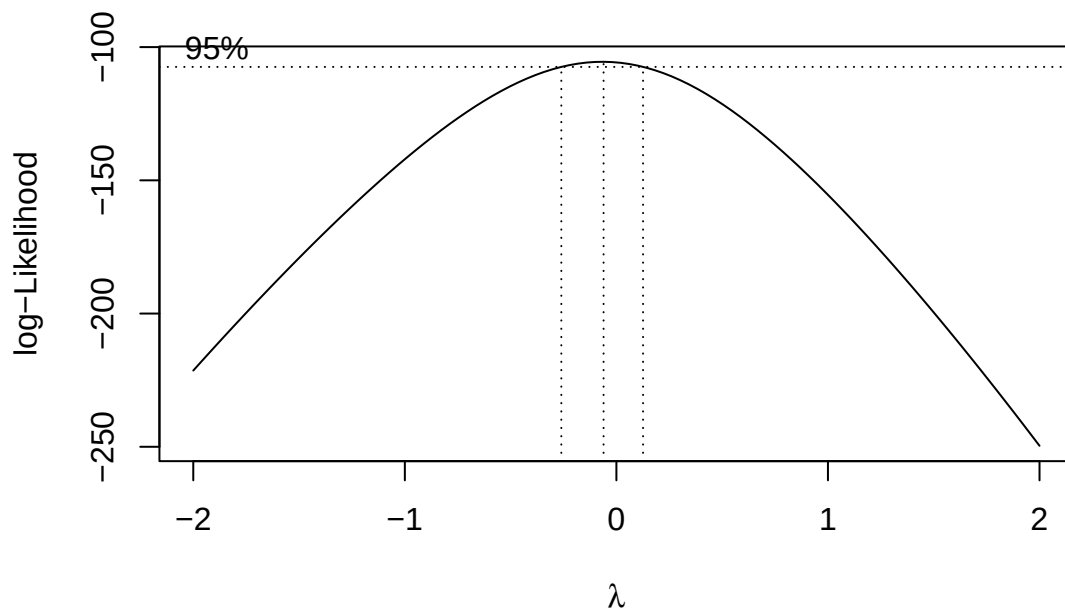
# some model diagnostics
par(mfrow=c(1,2))
plot(m1, which=1)
plot(m1, which=2)
```



Notice how the diagnostic plots show some patterns in the residuals, violating the assumptions of the linear regression model. A Box-Cox transformation might help here.

```
library(MASS)

bc <- boxcox(m1)
```



The Box-Cox function estimated an optimal λ near 0, which corresponds to the log transformation that actually generated the data (see Table 11.1).

Then, we extract the optimal λ :

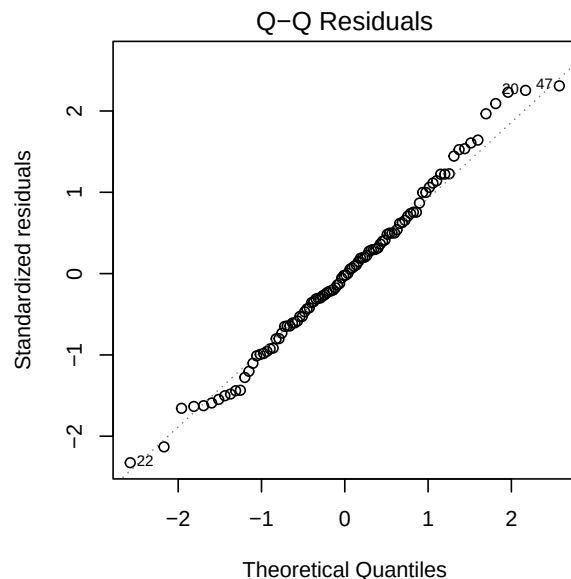
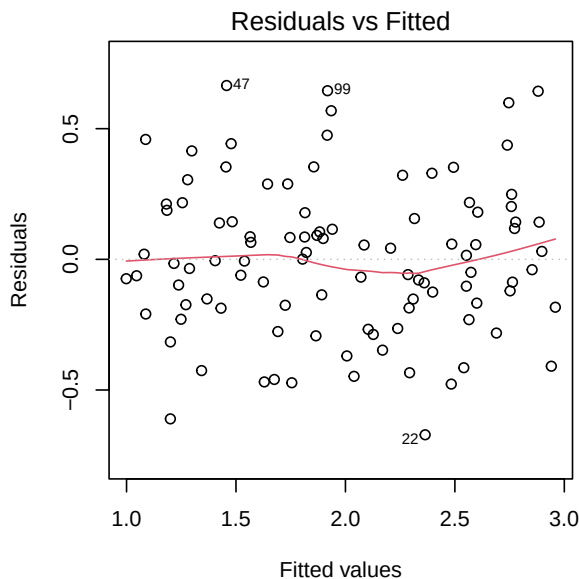
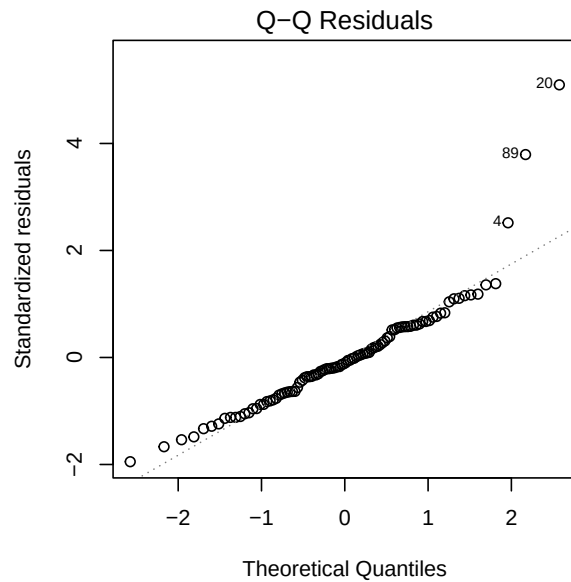
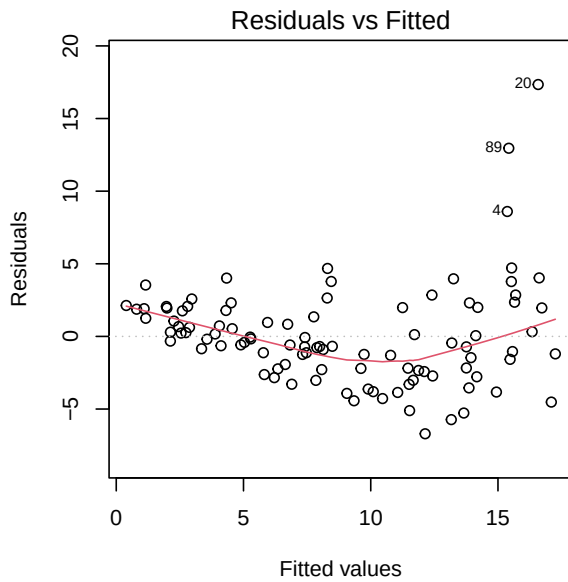
```
use.lambda <- bc$x[which.max(bc$y)]
use.lambda
```

```
[1] -0.06060606
```

You don't need to use the estimated value blindly. In this case, it is close to 0, suggesting a log transform ().

```
# refit the model with the log transform
m2 <- lm(log(y) ~ x)

# compare diagnostic plots
par(mfrow=c(2,2), mar=c(5.1, 5.1, 2.1, 1.1))
plot(m1, which=1)
plot(m1, which=2)
plot(m2, which=1)
plot(m2, which=2)
```



The Box-Cox transformation requires positive response values. In practice, its estimated value of λ is often most useful as a guide to choosing a simple, interpretable transformation rather than as a requirement to use the exact estimated value. As with any transformation, its success should ultimately be evaluated by examining the fitted model and its diagnostic plots.

The Box-Cox procedure was developed when transforming data to satisfy the assumptions of linear models was a common solution to non-normality, nonlinearity, and non-constant variance. It remains useful for identifying appropriate power transformations, but many problems that once motivated Box-Cox transformations can now be addressed more directly using generalized linear models or other models that explicitly represent the distribution and mean-variance relationship of the response.

11.6 Transformations closing thoughts

A transformation should not be chosen simply because it makes a scatterplot look more linear or a histogram look more normal. Before transforming a variable, consider **why** the observed pattern has its shape. A curved relationship may arise because the variables are related on a multiplicative scale, in which case a transformation may be useful. It may instead reflect the mean-variance relationship or support of a particular probability distribution, which may be better handled using a GLM. Alternatively, the curvature may reflect a genuinely nonlinear biological process, in which case a nonlinear model may provide a more meaningful description. The goal is not to make the data fit the model; it is to choose a model and scale that appropriately represent the data and the biological process that generated them.

Chapter 12

Common problems in biological data

Many statistical methods assume that data follow certain distributions, and we have already explored methods for making sure your data follow those distributions. Other problems are not so easy to diagnose or fix. This section will introduce you to some methods for dealing with more subtle data problems. Note that none of these issues have one-size-fits-all solutions. Dealing with these issues will require understanding your study system and thinking through the consequences of potential solutions.

12.1 Outliers and erroneous values

Outliers are values that much larger or smaller than other values in the data set. While these values are theoretically possible in any distribution, they are by their nature extremely unlikely. Outliers can cause problems with many statistical tests because they are often given undue weight. This is because deviations from the mean are often squared.

There are two problems with outliers: how to detect them, and what to do with them.

12.1.1 Detecting outliers

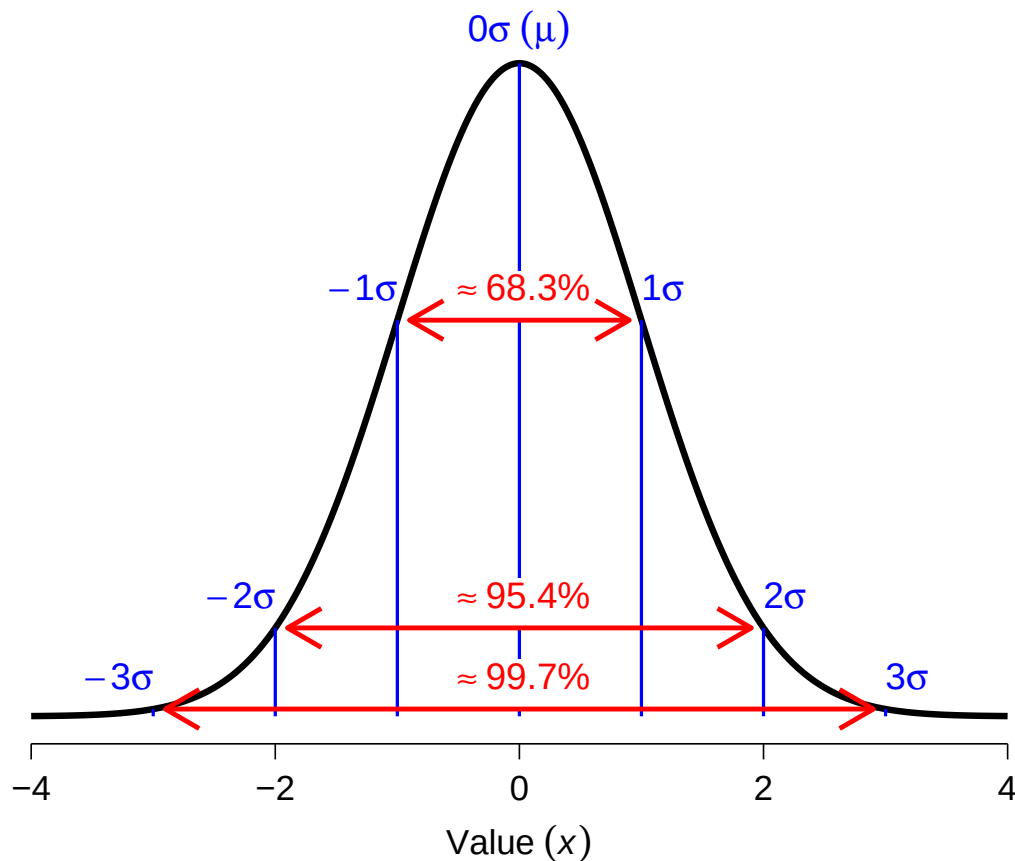
There are several definitions of outlier. Some researchers have suggested simple measures such as Z-scores to detect outliers. Z-scores measure how many standard deviations (SD) an observation is away from the mean. The Z-score of an observation x_i (observation i of variable x) is calculated as:

$$Z(x_i) = \frac{x_i - \bar{x}}{\sigma(x)}$$

where $\bar{x}$ is the mean of x and $\sigma(x)$ is the SD of x . For example, Z-scores of -2.3 and +4.1 indicate observations 2.3 SD below and 4.1 SD above the mean, respectively. Z-scores are also referred to as **sigmas**, referring to the symbol for SD. E.g., a 4σ observation is 4 SD away from the mean. Various σ levels have

been used to detect outliers, in which researchers discard any observation 2σ , 3σ , etc., away from the mean. This threshold varies by discipline, so follow the conventions in your field.

The figure below shows Z-scores relative to the standard normal distribution ($\mu = 0$, $SD = 1$). About 68% of observations are expected to have a Z-score between -1 and 1—i.e., deviate by $\leq 1\sigma$. About 95% of observations should be $\leq 2\sigma$, and over 99% of observations should be $\leq 3\sigma$. The greater the magnitude of a value's Z-score, the less likely it is. Values with > 3 or 4σ should almost never occur. Thus, observations with large-magnitude Z-scores may turn out to be outliers.



It must be noted that a large magnitude Z-score does **not** prove that an observation is an outlier. Like p -values, Z-scores are just a heuristic for making decisions. Don't just delete any observation $|Z| \geq 3$; examine those observations to see if they are legitimate and don't take any action without a good biological reason.

12.1.2 Dealing with outliers

12.1.2.1 Deletion

The easiest way to deal with outliers is simply to delete them. This can be fine if you have a lot of data—many 1000s or even millions of records. I've worked with several statisticians who routinely delete observations with response variables in the top and bottom 5 percentiles. This may not always be an appropriate strategy in biology. For one thing, outliers can occur for completely legitimate reasons and deleting them would be discarding important information about the study system.

Never delete an observation based solely on its magnitude.

Data should only be deleted when there is a biological or experimental reason. These might include experimental errors, inappropriate sampling, extraordinary field conditions, etc. Furthermore, you should always document when observations were deleted and why, and disclose that information in your write up and to reviewers. The possibility of including supplemental information or documents with manuscripts leaves little excuse not to be transparent about what observations you deleted and why.

12.1.2.2 Truncation and censoring

Censoring and **truncation** are options for dealing with extreme values. Censoring data in statistics means analyzing data as either the measured value or the interval in which the value falls. For example, if a mouse is weighed with a scale that goes up to 60 g, and maxes out the scale, we record the mouse as weighing ≥ 60 g. Statistical methods for censored data have been developed but are outside the scope of this course. One popular form of censoring is called **Winsorizing**, where values below the 5th percentile are set to the 5th percentile and values above the 95th percentile are set to the 95th percentile. Censoring is common practice in fields involving certain types of instrumentation. For example, many spectrophotometers (e.g., microplate readers) record optical density (OD) as the amount of light passing through a specimen as a measurement of concentration. Most machines have a lower limit to the OD that they can detect. Observations below this minimum OD might be censored to 0, or to some pre-determined fraction of the minimum detectable OD.

Similar to censoring, outlying data can be removed by **truncation**. Truncation in statistics means ignoring values outside of a certain range. As with deletion, you must be very explicit about what criteria were used to truncate the data. Just as with deletion:

Never remove an observation based solely on its magnitude.

12.1.2.3 Transformation

Another way to deal with outliers is to **transform the data**. Variance-stabilizing transformations, such as the log-transform, often have the side-effect of reducing the influence of outliers in the data.

12.1.2.4 Weighting

Weighting is a strategy where observations can be given greater or lesser weight to adjust how much influence they have. In ordinary least squares regression, every observation contributes equally to the sum of squared residuals. Weighted regression instead minimizes a weighted sum,

$$\sum_i^n w_i (y_i - \hat{y}_i)^2$$

where observations with smaller weights w_i have less influence on the fitted models.

Weighting is most straightforward when the weights are known for some external reason. For example, an observation measured with high precision might reasonably receive more weight than an observation measured with low precision. Choosing weights simply because observations appear to be outliers is more difficult, because arbitrary weights can produce arbitrary results.

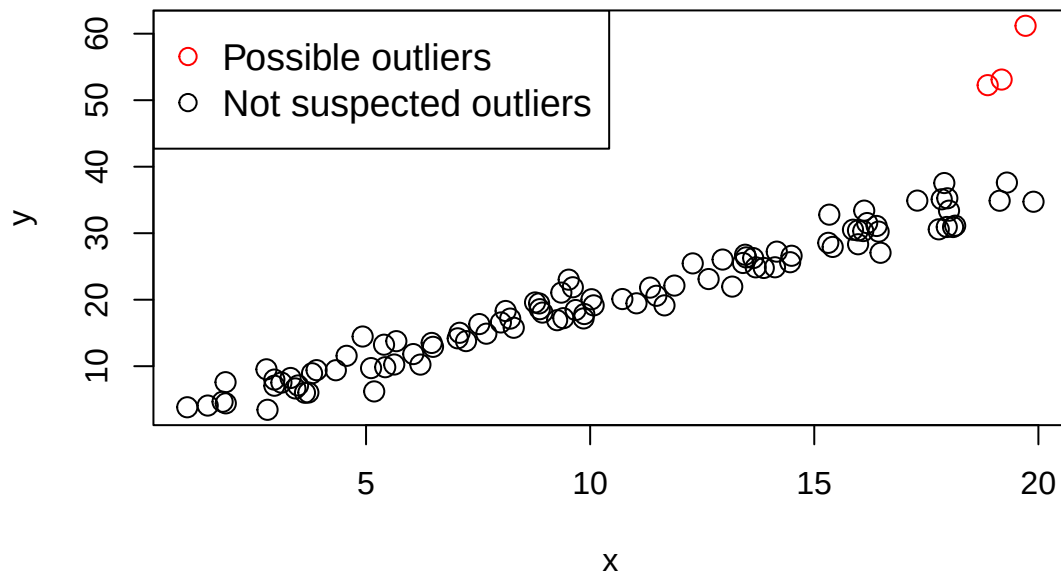
One approach that formalizes this idea is **robust regression**. Robust regression is a category of techniques that reduces the influence of extreme observations. The most famous version is probably **least absolute deviations** regression, which uses absolute rather than squared deviations. Below we see a version called **Huber M-estimator regression**, which uses iteratively-weighted least squares to arrive at a model in which observations with small residuals receive full weight, while observations with larger residuals are progressively downweighted.

Here is an example of using Huber regression to fit a linear model with less weight given to extreme values:

```
# simulate some data
# according to linear model
set.seed(123)
x <- sort(runif(100, 1, 20))
ey <- 1.5 + 1.8*x
y <- rnorm(100, ey, 2)

# add a few extreme observations
outliers <- c(95, 97, 99)
y[outliers] <- y[outliers] + 20
pt.cols <- rep("black", 100)
pt.cols[outliers] <- "red"

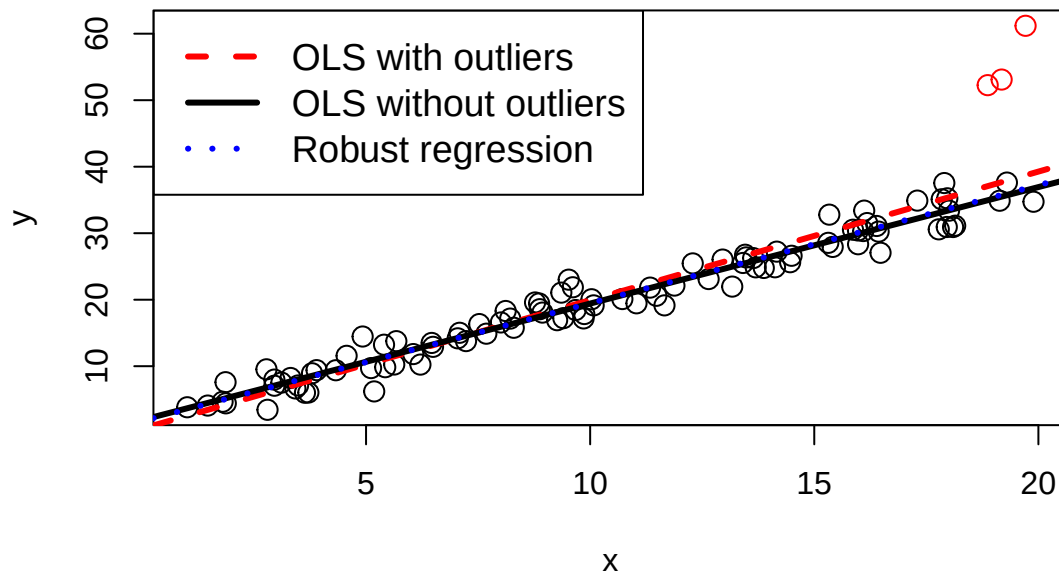
# make a plot
plot(x, y, col=pt.cols, cex=1.4)
legend("topleft", legend=c("Possible outliers", "Not suspected outliers"),
      col=c("red", "black"), cex=1.2, pch=1)
```



Let's compare ordinary regression with the possible outliers (mod1), ordinary regression without the possible outliers (mod2), and robust regression (mod3).

```
my.dat <- data.frame(y=y, x=x)
mod1 <- lm(y ~ x, data=my.dat)
mod2 <- lm(y ~ x, data=my.dat[-outliers,])
mod3 <- MASS::rlm(y ~ x, data=my.dat)

plot(x, y, col=pt.cols, cex=1.4)
abline(mod1, lwd=3, col="red", lty=2)
abline(mod2, lwd=3)
abline(mod3, lwd=3, col="blue", lty=3)
legend("topleft",
      legend=c("OLS with outliers",
               "OLS without outliers",
               "Robust regression"),
      lwd=3, lty=c(2,1,3),
      col=c("red","black","blue"),
      cex=1.2)
```



The robust regression finds a line very similar to the model without the outliers, but does so without discarding data.

12.1.2.5 Alternative strategies

Finally, researchers can use analysis strategies that are less sensitive to outliers. As mentioned above, there are a large class of **robust statistics** that use different strategies to give suspected outliers much less influence than under conventional methods based on sums of squared errors. **Nonparametric methods**, particularly those based on rank order, can also be less sensitive to outliers.

Whatever strategy you choose to deal with your outliers, you need to be aware of how much your decision affected the outcome. It is a good idea to analyze your data both with and without suspected outliers to measure how much influence they really have. If the analysis returns the same results with and without the outliers, leave them in.

12.2 Autocorrelation

Autocorrelation describes a phenomenon where set of values is correlated with itself, with the strength of that correlation determined by the distance between observations. In biology, the “distance” that defines autocorrelation is often spatial, temporal, or phylogenetic.

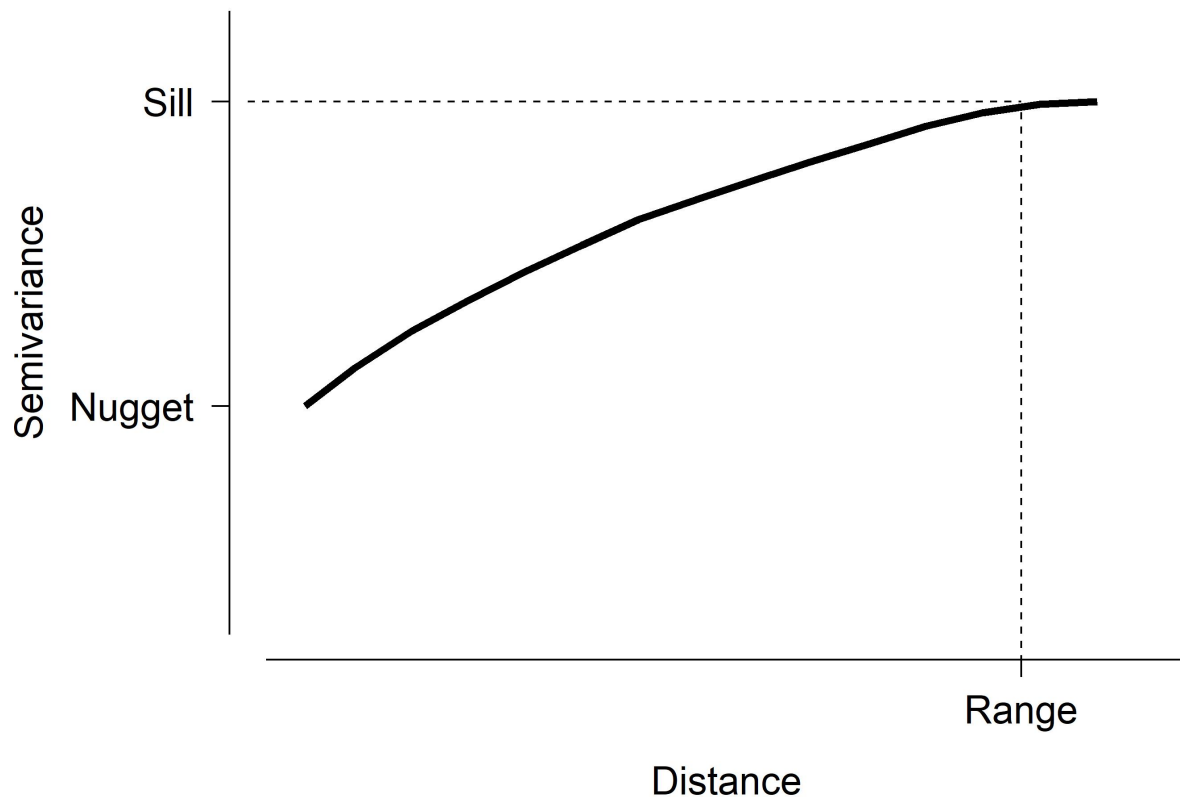
Autocorrelation type	Distance measure	Explanation
Spatial	Space (e.g., km)	Samples from sites that are close together in space are more similar to each other than they are to sites that are farther away, just because they are closer to each other.
Temporal	Time (e.g., days)	Observations taken closer together in time are more similar to each other than they are to observations separated by more time.
Phylogenetic	Relatedness	Taxa that share a more recent common ancestor (i.e., are more closely related) are more similar to each other than they are to taxa that are less closely related to each other.

The main consequence of autocorrelation is similar to the problems associated with pseudoreplication: because observations are not independent, the statistically relevant number of observations (degrees of freedom) is smaller than the nominal number of observations. This means that the power of the test is inflated. A related idea is that the residuals of a statistical model can be autocorrelated. In that case, the model violates the assumption of independent residuals shared by most statistical methods. Autocorrelation can be controlled for if you have some idea of its source and nature.

12.2.1 Graphical methods to identify autocorrelation

One way to explore potential autocorrelation is to construct a variogram. A variogram plots the squared differences between samples against their distance (temporal, spatial, etc.). A semivariogram uses half of that variance. An example semivariogram is below.

This figure shows some important quantities on the semivariogram:



{alt-

text: "Schematic semivariogram with distance on the x-axis and semivariance on the y-axis. A curved line begins above zero at the labeled nugget, increases with distance, and levels off at the labeled sill. A vertical dashed line marks the range, where the curve reaches the sill."}

- **Nugget:** The variance or semivariance at 0 or very short distance. This represents other sources of error, such as sampling error, demographic stochasticity, etc.
- **Sill:** The variance or semivariance at which the curve gets close to its maximum or asymptotic value.
- **Range:** The distance at which the variance or semivariance reaches the sill. Samples at distances beyond the range are considered no longer spatially autocorrelated.

Semivariograms should always increase, at least initially, because closer observations should be similar to each other simply as a matter of proximity. Samples separated by distances less than the range are autocorrelated. Semivariograms sometimes reach a peak and then start decreasing, or varying up and down. The initial peak is usually interpreted as the sill, and the distance when the curve reaches the sill is considered the range. The importance and interpretation of the semivariogram is something that you, as a biologist, have to figure out.

12.2.2 Mantel tests for autocorrelation

Spatial autocorrelation is an ever-present problem for ecologists and environmental biologists: observations that are closer together in space might be similar to each other in terms of biology, just because they are closer together. The cause of that spatial autocorrelation determines how you need to deal with it in your analysis.

The fact that observations close together in space tend to be more similar is not in and of itself a problem. There can be a strong spatial structure in the data that is explained by predictors like latitude, longitude, or some environmental gradient. In other words, the spatial structure may be what you are trying to discover in the first place.

Where spatial structure becomes a problem is in the *residuals* of your analysis. If the residuals contain spatial structure *after accounting for environmental and other predictors*, then there is a problem called “lack of independence”. This means that observations contain less independent information than the model assumes. Consequently the analysis is conducted with more degrees of freedom than there really are. This inflates the power of the test, increasing the possibility of a false positive (a Type I error).

Historically, biologists often used a **Mantel test** to test for spatial autocorrelation in the response variable. Briefly, the Mantel test can be thought of as a multivariate generalization of linear correlation that measures correlation between two matrices. Whereas linear correlation measures how variation in one variable is related to variation in another variable, the Mantel test measures how variation in one matrix is related to variation in another matrix. But given what you just read, you should now recognize that as insufficient:

1. autocorrelation in response *values* is not necessarily a problem if it can be modeled by predictor variables;
2. autocorrelation in model residuals is where the problem lies; and,
3. a Mantel test can fail to detect distance decay in response values, even when residuals are spatially dependent.

Nowadays, we have tests specifically designed for measuring spatial autocorrelations. One of the most common is **Moran's I** . This is a common measure of spatial autocorrelation that is easy to interpret: $I > 0$ means that nearby observations are more similar than expected; $I \approx 0$ means little evidence of spatial autocorrelation; and $I < 0$ means that nearby observations tend to be more dissimilar to each other than expected.

The “than expected” part refers to the amount of similarity expected if the values were distributed randomly with respect to the specified spatial relationships among observations..

We will illustrate use of Moran's I in a future module. For now, the basic procedure is to:

1. Define spatial weights for each pair of observations (a kind of distance matrix). These could be based on a distance threshold, nearest neighbors, or geographic distance.
2. Fit your model (regression, GLM, or whatever).
3. Test for spatial autocorrelation of the model residuals by calculating Moran's I using the spatial weights. One option is the function `spdep::moran.test()`.

12.2.3 Dealing with autocorrelation

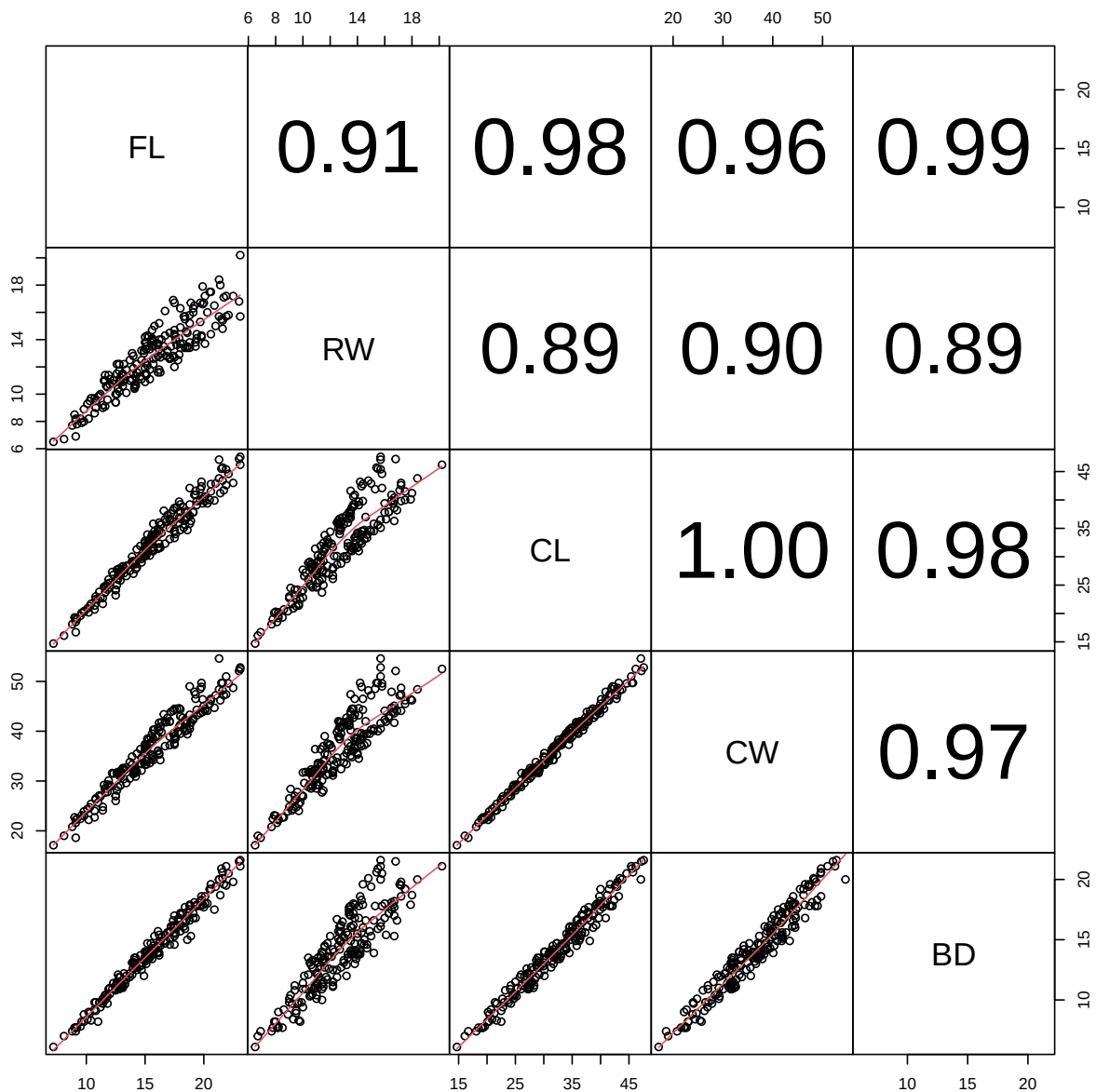
If you detect significant autocorrelation, don't panic. Autocorrelation may or may not be a problem in your dataset. As pointed out by Kuehn and Dormann (2012), there is a difference between autocorrelation in a response variable and autocorrelation in the residuals of a statistical model. Autocorrelation in a response variable can be managed by inclusion of appropriate predictor variables. This works because if autocorrelated values can be predicted by a predictor variable, then it won't be present in the residuals.

Autocorrelation in residuals, however, represents spatial structure that is not accounted for by the model. There are methods for accounting for autocorrelated residuals such as generalized least squares models (GLS) with spatial correlation structures (Zuur *et al.* 2007), conditional autoregressive models (CAR) (Gelfand and Vounatsou 2003), and mixed models that explicitly account for spatial dependence (Bolker *et al.* 2009). Other approaches may incorporate spatial predictors or smooth spatial trends directly into the model. None of these solutions is appropriate for every situation, so consider the likely cause and spatial scale of the autocorrelation, the goals of the analysis, and what your data can support.

12.3 Collinearity

Sometimes we face the problem of having predictor variables that are not only related to the response variable, but also related to each other. This phenomenon is called **collinearity**. In large datasets the problem becomes **multicollinearity**, when many variables are related to each other. Including collinear predictor variables in an analysis can severely bias the parameter estimates of that analysis (e.g., regression coefficients) and as a result make the results misleading or not generalizable. The effects of collinear predictor variables are by definition confounded with each other, meaning that the effects cannot be separated (Dormann *et al.* 2013).

The most straightforward way of detecting collinearity is with scatterplot matrices and correlation coefficients (see Section 8.3.1). This is a technique to visualize how many variables in a dataset relate to each other simultaneously. As shown in the linked section, replacing the upper triangle of boxes with the correlation coefficients can help detect potentially problematic pairs of variables. The figure below shows an example from the `crabs` dataset in package `MASS`. In this dataset, all of the variables are highly correlated with each other. Using more than one of them as a predictor in a model would make it hard for the algorithm to distinguish their effects from each other.



If you have many variables, then a figure like the one above can be hard to read or interpret. One solution might be to create a table of correlation coefficients between every pair of variables. The base function `cor()` can do this, but a better-formatted table takes a little more work.

```
data(crabs, package="MASS")
x <- crabs
dat.cols <- 4:8
cor(x[,dat.cols])
```

FL

RW

CL

CW

BD

```

FL 1.0000000 0.9069876 0.9788418 0.9649558 0.9876272
RW 0.9069876 1.0000000 0.8927430 0.9004021 0.8892054
CL 0.9788418 0.8927430 1.0000000 0.9950225 0.9832038
CW 0.9649558 0.9004021 0.9950225 1.0000000 0.9678117
BD 0.9876272 0.8892054 0.9832038 0.9678117 1.0000000

```

```

# same table as a data frame:
vars <- names(x)[dat.cols]

# generate every unique pair of two variables by name:
cx <- data.frame(t(combn(vars, 2)))

# calculate correlation coefficients for each pair:
cx$r <- NA
for(i in 1:nrow(cx)){
  cx$r[i] <- cor(x[,cx$X1[i]], x[,cx$X2[i]])
}
cx

```

	X1	X2	r
1	FL	RW	0.9069876
2	FL	CL	0.9788418
3	FL	CW	0.9649558
4	FL	BD	0.9876272
5	RW	CL	0.8927430
6	RW	CW	0.9004021
7	RW	BD	0.8892054
8	CL	CW	0.9950225
9	CL	BD	0.9832038
10	CW	BD	0.9678117

The resulting table can then be sorted or filtered by `r` to identify pairs of variables that are correlated with each other.

```

# not run:
# filter to |r| > 0.7
cx[which(abs(cx$r) > 0.7),]

```

Dormann *et al.* (2013) suggest to use a cutoff of $|r|$ in between 0.5 and 0.7 to identify collinear pairs of variables. When a pair of variables are identified as collinear, the easiest solution is to simply run the analysis without one of them. This is because the variables are redundant, and any conclusions made using one would also be obtained using the other.

12.4 Missing data

The last target of exploratory data analysis that we will explore is **missing data**. Data can be missing for many reasons. Equipment failures, human error, natural disasters, and other causes can cause values in your dataset to go missing or otherwise be invalid. How to handle the missing data is up to you as a biologist: *there is no one-size-fits-all statistical solution*. The two basic ways to deal with missing data are to (1) not deal with it and (2) estimate the missing values.

12.4.1 Option 1: Ignore missing data

The easiest solution is simply to ignore missing data. Consider [this dataset](#), which contains morphological data on 63 species of bats (Hutcheon *et al.* 2002). Notice that in this version the brain masses (`brw`) are missing for 3 species of bats.

```
# replace with your working directory and path!
in.name <- "../datasets/missing-data-example.csv"
dat <- read.csv(in.name, header=TRUE)
```

Because there are a lot of data, one option for analyzing this dataset would be to simply ignore the missing observations. For example, if someone was interested in the relationship between body size (variable `bow`) and brain mass, they could just fit a model and R would (for most methods) automatically ignore missing values. Note that in this example both variables are log-transformed to ensure normality and positivity.

```
dat$logbw <- log(dat$bow)
dat$logbr <- log(dat$brw)

mod1 <- lm(logbr~logbw, data=dat)
summary(mod1)
```

Call:

```
lm(formula = logbr ~ logbw, data = dat)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-0.82879	-0.16408	0.05009	0.19548	0.35881

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	3.96599	0.09476	41.85	<2e-16	***
logbw	0.75322	0.02881	26.14	<2e-16	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2616 on 58 degrees of freedom

(3 observations deleted due to missingness)

Multiple R-squared: 0.9218, Adjusted R-squared: 0.9204

F-statistic: 683.4 on 1 and 58 DF, p-value: < 2.2e-16

The parameter estimates of intercept = 3.9659 and slope = 0.7532 are very close to the values obtained using the full dataset of 3.9685 and 0.7516. In this case, ignoring the missing data worked just fine.

12.4.2 Option 2: Interpolate or impute missing data

If you need to replace the missing data, then the best option is to **interpolate** or **impute** them. How exactly to do this depends on the question being asked. If there is a clear relationship between the variable with missing values and another variable, then you can use that relationship to estimate the missing values. You can also use **multiple imputation** to estimate the missing values using information from several related variables at once. The example below illustrates simple interpolation (aka: single imputation) to replace the missing brain weights using known body masses.

```
# flag missing observations
flag <- which(is.na(dat$brw))

# log transform (if not done already)
dat$logbw <- log(dat$bow)
dat$logbr <- log(dat$brw)

# fit model
mod1 <- lm(logbr~logbw, data=dat)

# predict missing logbr at logbw
px <- data.frame(logbw=dat$logbw[flag])
pr <- predict(mod1, newdata=data.frame(px))

# insert predicted values
dat$logbr[flag] <- pr

# back-transform to original scale
dat$brw[flag] <- exp(dat$logbr[flag])
```

The interpolated values are similar to the true values, but with quite a bit of error. The term of art here is “correct to an order of magnitude”.

```
dat$brw[flag]
```

```
[1] 729.7784 507.7160 2138.6809
```

```
true.vals <- c(597, 543, 2070)
```

```
# percentage bias in interpolated values:  
100 * (dat$brw[flag]-true.vals) / true.vals
```

```
[1] 22.240932 -6.497972 3.317919
```

Interpolation can be improved by predicting the missing values using additional variables. Multiple imputation, like any statistical model, can suffer from the effects of multicollinearity in the “predictors”, so be mindful that you don’t obtain—and use—biased estimates of your missing values. Penone *et al.* (2014) reviewed some strategies for imputing missing data in the context of life history traits, but the approaches could be applied to other areas of biology.

Chapter 13

Planning your data analysis

One of the main questions that biologists have after collecting data is, “What statistical test do I use?” Ignoring the fact that they should have thought about that ***before collecting their data***, the answer to that question depends on the answers to two other questions:

1. What question are you trying to answer?
2. What kind of data do you have?

After **exploring the data**, **summarizing the data**, **thinking about distributions**, **checking data distributions**, and gaining an understanding of the **patterns** and **problems** in a dataset, it is time to run the main analysis. This guide is designed to help you use what you learned in exploring the data to choose an appropriate analytical strategy.

13.1 How to use this guide

The sections below list some common analysis strategies that you might use for some common types of biological questions, and guides you through the selection process. This document is meant to serve as a primer and rough guide; it is not meant to be exhaustive. This document is also very much a work in progress. Not every situation is covered, and there might be issues in your dataset that make the recommendations below problematic.

Each section is structured as a dichotomous key, because biologists love dichotomous keys. At each step, choose the option that best applies to your situation. If you don't know the answer to a question or which option to choose, you may need to go back and do some more exploratory data analysis. R functions are provided for most methods.

13.2 What question are you trying to answer?

13.2.1 Quantifying your question

The first thing to nail down is what *statistical question* you are trying to answer. This does not mean what *biological question* you are investigating, but two are related. An example biological question might be:

How do small mammals (rodents) adapt to live in urban environments?

This is a good biological question, but it is *not* a question that you can answer with statistics. In order to answer the overarching biological question, we usually need to break it down into smaller chunks: hypotheses that can either be true or false, and predictions (patterns in the data) that would be observed if the hypothesis is true and not observed if a hypothesis is false. For the question above, we might have some working hypothesis like:

Rodents living in urban environments shift their diet to take advantage of anthropogenic food sources.

This is a good scientific hypothesis, because it is **falsifiable**. For this or any other hypothesis we need to be able to define:

1. What would we observe if the hypothesis is not true?
2. What would we observe if the hypothesis is true?

For this statement, these are relatively straightforward. We are closer to a statistical hypothesis, but we're not quite there yet. What does it mean to "shift their diet"? Compared to what? How do we quantify a "shift"?

To make a statistical hypothesis we need to think of *specific quantitative or categorical comparisons that would support or contradict the hypothesis*. For example, here are a few of the potential predictions implied by the hypothesis above:

- Rodents in urban environments will have a **greater proportion** of artificial foods in their feces than conspecific rodents in rural or natural environments.
- Rodents in urban environments will consume a **smaller number** of different kinds of food than rodents in rural environments because of the availability of calorie-dense foods associated with humans.
- Rodents in urban environments will spend **less** time foraging and make **shorter** foraging movements than rodents in rural or natural environments.
- Stable isotope ratios of whole-body tissues in urban animals will be **different than** those ratios in rural animals.
- Serum triglyceride concentrations will be **correlated with** amount of impervious surface within 150 m of their home range.

These predictions are statements of the kinds of patterns that we should observe if the hypothesis was true. Note the bolded parts: these are quantitative comparisons that can be true or not true.

We can also define their logical negation: what would we observe if the hypothesis was not true? Critically, these predictions include comparison phrases like **greater than** or **smaller number of** or **less time than**. Those comparisons are the sorts of patterns that a statistical model can actually evaluate. Knowing what kind of pattern you are testing is a key step in planning your data analysis.

13.2.2 Types of questions

Broadly speaking, statistical hypotheses in biology tend to fall into just a few categories:

1. Testing for a *difference in mean or location* between groups.
2. Testing for a *continuous relationship* between a response variable (or multiple variables) and one or more predictor variables. Also referred to as “predicting” or “modeling” a continuous variable.
3. Testing for (lack of) *independence between categories* of observations.
4. *Classifying* observations based on predictor variables.

These categories are not mutually exclusive. Many questions phrased as one category can be reframed into another. How you frame your question depends on many factors and at some point will involve some discretion on your part. *There is often more than one right way analyze a dataset*, but the right ways are a vanishingly small subset of the possible ways to analyze your data.

13.3 Testing for a difference in mean or location

Dichotomous key for testing for a difference in mean or location:

1. Single response variable (or one variable tested at a time) → **3**
- 1'. Multiple response variables tested at once → **2**
2. Data are appropriate for a parametric multivariate analysis → Try **MANOVA**.
R function: `manova()`.
- 2'. Data are not appropriate for MANOVA, or differences are being measured using a distance or dissimilarity measure → Try **permutational MANOVA (PERMANOVA)**.
R function: `vegan::adonis2()`.
3. Need to compare values in one group to a target value → **4**
- 3'. Need to compare values in one group to values in at least one other group → **5**
4. Data are appropriate for a parametric test → Try a one-sample *t*-test.
R function: `t.test()`.
- 4'. Data are not appropriate for a parametric test → Try a one-sample Wilcoxon signed-rank test.
R function: `wilcox.test()`.
5. Comparing two groups in a simple design → **6**
- 5'. Comparing more than two groups, or using a more complex design involving multiple factors, interactions, blocking, or repeated measures → **7**

6. Data are appropriate for a parametric test → Try a **two-sample *t*-test**, including a paired *t*-test if observations are paired or matched.
R function: `t.test()`.
- 6'. Data are not appropriate for a parametric test → Try a **Mann-Whitney test (Wilcoxon rank sum test)**, or a Wilcoxon signed-rank test for paired data.
R function: `wilcox.test()`.
7. Data are appropriate for a parametric test → Try **ANOVA**. This includes designs with multiple factors, interactions, blocking, and repeated measures.
R functions commonly include `lm()` and `aov()`.
- 7'. Data are not appropriate for a parametric test → For a simple independent-groups design, try the **Kruskal-Wallis test**. More complex designs may require another method appropriate for the study design.
R function: `kruskal.test()`.

13.3.1 Additional considerations

Once you reach a decision from the key above, there are a few more options to consider:

- Are there any **interactions** (see Section 15.5.1.3 and Section 16.5.1) between your predictor variables? This means that the effects of two predictors are not additive; i.e., the effect of one variable affects the effect of another variable. Most of the methods above can accommodate interactions. You can use R function `interaction.plot()` to explore potential interactions visually.
- Is there a hierarchical structure to your data, with the higher-level groups representing variation that you are interested in explaining? This is often called “blocking” and needs to be included in your model. Usually the blocking factor can be included as a predictor in the model (see Section 15.5.1.4).
- Is there a hierarchical structure to your data, with the higher-level groups representing variance you are not interested in but still need to account for? If so, you may want to consider adding random effects to your model. This creates a **mixed model** (see Chapter 23).

13.4 Testing for a continuous relationship between two or more variables

Dichotomous key for testing for a continuous relationship between two or more variables:

1. Need to measure whether two variables vary together, without distinguishing between predictor and response variables → **2**
- 1'. Need to model or predict a response variable using one or more predictor variables → **3**
2. Interested in the strength of a linear relationship → Use **Pearson's product-moment correlation, *r***.
R functions: `cor()` or `cor.test()`.

- 2'. Interested in the strength of a monotonic relationship, or data are ordinal or otherwise inappropriate for Pearson's correlation → Use **Spearman's rank correlation coefficient, ρ** .
R functions: `cor()` or `cor.test()` with `method="spearman"`.
3. Response variable is continuous and can reasonably be modeled using a normal distribution → 4
- 3'. Response variable is better described by another probability distribution, such as binary, proportion, or count data → Use a **generalized linear model (GLM)**.
R function: `glm()`.
4. Relationship between the response and predictor variables can reasonably be represented by a linear model → Use **linear regression** or **multiple linear regression**.
R function: `lm()`.
- 4'. Relationship between the response and predictor variables is nonlinear → 5
5. Have a biologically or mathematically meaningful function describing the expected relationship → Use **nonlinear regression**.
R function: `nls()`.
- 5'. Shape of the relationship is unknown or needs to be estimated flexibly from the data → Try a **generalized additive model (GAM)** or **regression tree**.
R functions: `mgcv::gam()` or `rpart::rpart()`. ### Additional considerations {#sec-plan-cont-addl}

Once you reach a decision from the key above, there are a few more options to consider:

- Is there autocorrelation in your data? If so, you may want to consider including an autocorrelation structure.
- Is there a hierarchical structure to your data, with the higher-level groups representing variance you are not interested in but still need to account for? If so, you may want to consider adding **random effects** to your model. This creates a **mixed model**:
 - Linear models become **linear mixed models (LMM)**. R function `lme4::lmer()`.
 - Generalized linear models become **generalized linear mixed models (GLMM)**. R function `lme4::glmer()`.
 - Additive models become **generalized additive mixed models (GAMM)**. R function `mgcv::gamm()`.
 - NLS models become **nonlinear mixed models (NLME)**. R function `nlme::nlme()` or `lme4::nlmer()`.
- Do your data meet most of the assumptions for linear regression, but with heteroscedasticity (non-constant variance in residuals)? You might consider using generalized least squares (GLS) with a different variance structure. R function `nlme::gls()`.
- Is there severe **multicollinearity** among your predictor variables? If so, try dropping some of the collinear predictors or try a regularization technique such as ridge or lasso regression. Various R packages.
- Do you not have a clear idea of the form of the relationship between your continuous response variable and the predictors? Or, do you not want to assume a data model and just let the data speak for themselves? Try a machine learning technique like boosted regression trees. R function `gbm::gbm()`.

13.5 Relationships between categorical variables

When all variables are categorical, the question being asked is likely something like, “Do the observed frequencies differ between groups?”. Such data are usually summarized in a contingency table (R function `ftable()`; see Section 7.3.2) and analyzed with a χ^2 test (“chi-squared test”). But, the χ^2 test isn’t the only option:

Dichotomous key for tests of independence:

Dichotomous key for testing independence between categorical variables:

1. Observations are independent → **2**
- 1'. Observations are paired or matched, with a binary categorical response measured twice → Use **McNemar’s test**.
R function: `mcnemar.test()`.
2. Expected cell counts are sufficiently large → Use a **χ^2 test of independence**.
R function: `chisq.test()`.
- 2'. Some expected cell counts are too small for the χ^2 approximation to be reliable → Use **Fisher’s exact test**.
R function: `fisher.test()`.

13.6 Classifying observations

Dichotomous key for classifying observations:

1. Outcome has two possible categories → **2**
- 1'. Outcome has more than two possible categories → **3**
2. Want to model how predictor variables are related to the probability of an outcome → Use **logistic regression**.
R function: `glm()` with `family=binomial`.
- 2'. Want a flexible method primarily for classification or prediction → Try a **classification tree**.
R function: `rpart::rpart()`.
3. Outcome categories have a meaningful order → Use ordinal logistic regression (Section 21.3). R functions include `MASS::polr()` and `ordinal::clm()`.
- 3'. Outcome categories do not have a meaningful order → **4**
4. Want to model how predictor variables are related to the probabilities of different outcomes → Try multinomial logistic regression.
R function: `nnet::multinom()`. These methods are beyond the scope of this course.
- 4'. Want a flexible method primarily for classification or prediction → Try a **classification tree**.
R function: `rpart::rpart()`.

13.7 Conclusions

This guide was designed to cover a lot of possible pathways for biological data analysis, but of course there are some situations that aren't covered. By the time you have deconstructed your problem enough to answer the diagnostic questions above, you should be well on your way to figuring out what analysis to use. Whatever you do, make sure that you document your decision and its justification...this will be very helpful later!

Part IV

Introductory statistics with R

Everything you definitely learned in undergraduate statistics, but could use a refresher on.

Chapter 14

Introductory statistics with R

In this module, we will review some basic statistical methods that are usually covered introductory statistics or biostatistics course. These tests are classics for a reason: they are powerful and collectively can be applied to many situations. However, as we shall see, these tests have a lot of assumptions and conditions that must be met in order to use them...and those conditions are not always easy to meet.

First, we'll review the basics of how biological data are stored, then what a frequentist statistical test really is—including what *P*-values are and what they represent. Then we'll review the classic frequentist tests in 4 categories:

1. Tests for differences in mean or location.
2. Tests for patterns in continuous data.
3. Tests for proportions and contingency (aka: tests for independence)
4. Classification.

Most biological problems fall into one of those four categories. Most introductory statistical courses cover the first 3 categories pretty well. Classification can be more complicated. When your problem does not, you'll need to move on to one of the more advanced analyses we'll cover later this semester.

14.1 Datasets and variables

When biologists collect data in an experiment, they create a **dataset**. This often takes the form of a table. Different software packages may refer to this as a spreadsheet, data frame, or matrix. Whatever you call them, a typical biological dataset usually looks like this:

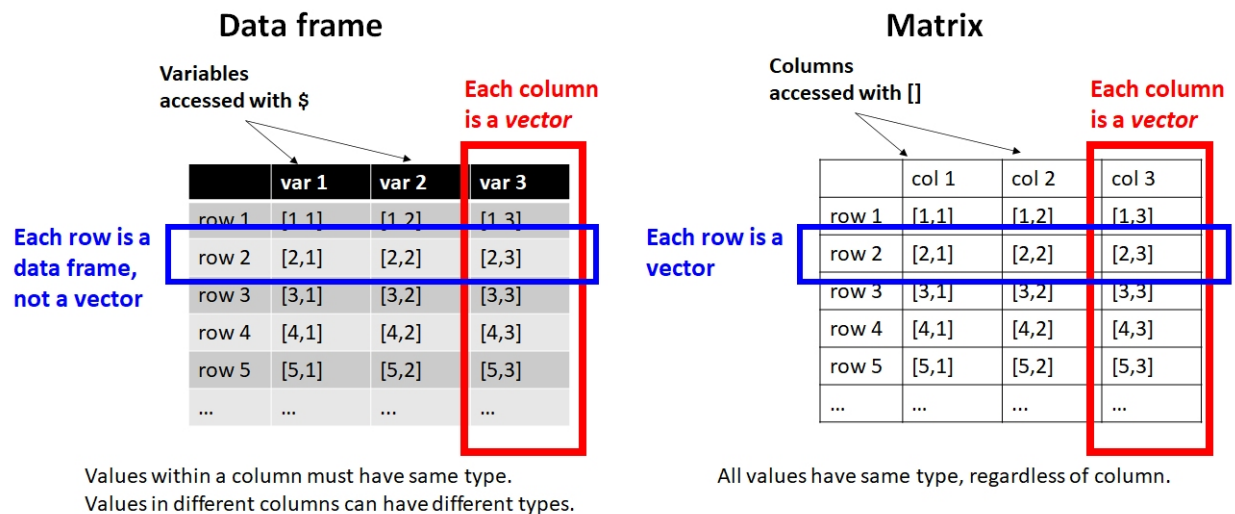
Figure 14.1: Structure of a typical biological dataset. Observations are stored on rows, and variables that describe those observations are stored in columns. All information about an observation is contained in 1 row. All observations of one variable are contained in 1 column.

Observations	Variables					
	Var. 1	Var. 2	Var. 3	Var. 4	Var. 5	Var. 6
	Observation 1					
	Observation 2					
	Observation 3					
	Observation 4					
	Observation 5					
	Observation 6					
	Observation 7					
	Observation 8					
	Observation 9					
	Observation 10					
	Observation 11					
	Observation 12					

- Each row of the dataset is called an **observation**, or sometimes **record**. Each entity about which data is collected corresponds to 1 row of the dataset. For example, the dataset for a study of limb morphology in bats might have one row for each species.
 - If someone asks what your “sample size” or “number of samples” is, the answer is likely the number of rows in your dataset (minus any blanks or missing values, of course).
- Each column is called a **variable**, or sometimes a **feature**.
 - Each variable contains 1 and only 1 kind of value. For example, a cell entry like “12.2 cm” is not legitimate, because it contains 2 kinds of values: a measurement (12.2) and a unit (cm). If the units need to be recorded, they should be in a separate column.

In R, datasets are almost always stored as objects called **data frames**. Data frames have rows and columns, and function a bit like a spreadsheet. Unlike a **matrix**, rows and columns are not interchangeable: a row of a data frame is a data frame with 1 row, while a column is really a vector (i.e., not a data frame).

Figure 14.2: Comparison of the structure of data frames and matrices in R.



Variables in a dataset come in three varieties. Understanding these is crucial to being able to think about the structure of your dataset and what kind of statistical test you need.

- **Response:** outcomes that you want to explain or predict using other variables. AKA: dependent variables.
- **Predictor:** variables that are used to model, or predict, response variables. AKA: independent or explanatory variables.
- **Key:** values which are used to relate 2 or more data tables to each other. Fundamental in database design.

In this course, every analysis will have at least 1 response variable and at least 1 predictor variable. Understanding the nature of your response variable and predictor variable(s) is key to choosing an appropriate analysis.

Variables can be either **categorical** or **numeric**:

- **Categorical:** defines group membership; can be encoded as numbers, dummy values, or text. Also called **qualitative** variables. E.g., “control” vs. “treated”, “group 1” vs. “group 2”, “herbivore” vs. “carnivore”, and so on.
- **Numeric:** numbers. Also called **quantitative** variables. E.g., 1, 2, 3, and so on.

When categorical variables are used as explanatory variables, they are called **factors**. Each unique value of a factor is called a **level**. For example, the factor “treatment” might have two levels, “control” and “exposed”. Or, the factor “diet” might have 3 levels: “herbivore”, “omnivore”, and “carnivore”.

Numeric variables can be **continuous** or **discrete**:

- **Continuous** variables can take on any real value within their domain. Just because a value is rounded off to integers, does not mean they are not continuous. For example, if mice are weighed on a scale with precision of 1 g, then all measured weights will be positive whole numbers. These should be treated as continuous in the analysis.
- **Discrete** variables can take on only integer values, and in most cases only positive or non-negative integers. Counts are a common example of a discrete value.

Some kinds of numeric variables need to be treated with care, because they have a particular domain and set of possible values. Most of these should probably not be modeled (i.e., treated as the response variable) using the basic statistical methods in this module. We'll explore models in **Module 18** for these variables.

Type of data	Domain	How to model
Counts	Non-negative integers	GLM or GLMM
Proportions	The closed interval $[0, 1]$, or possibly the open interval $(0, 1)$ depending on the context	GLM or GLMM; possibly a χ^2 test or other test for independence
Measurements (mass, length, volume, etc.)	Positive real numbers for dimensions; real values for temperature (except Kelvin)	Linear model; transformation or an appropriate GLM/GLMM if needed
Censored or truncated data	Usually real numbers with a well-defined upper or lower limit, often related to limits of detection or measurement	Special methods such as Tobit models (beyond the scope of this course...for now)
Categorical outcomes	One of a set of possible categories. Categories may be binary, ordered, or unordered. Although often coded as numbers, they should not generally be treated as numerical measurements	Logistic, ordinal, or multinomial regression; GLMMs for non-independent observations; classification trees (CART)

14.2 Frequentist statistics

Classical statistics, such as those taught in introductory statistics or biostatistics courses, are also **frequentist** statistics. This means that these methods rely on probability distributions (also called frequency distributions) of derived values called **test statistics**, and make assumptions about the applicability of those test statistics. These assumptions can be quite strong, and possibly restrictive, so you need to be careful when using frequentist tests.

If your data do not meet the assumptions of a test, the test might be invalid.

The basic procedure of any frequentist test is as follows:

1. Collect data.
2. Calculate a summary of the effect size (aka: quantity of interest, such as difference in means between groups), which scales the effect size by other considerations such as sample size or variability. This summary is the **test statistic**.
3. Compare the test statistic to the distribution of test statistics expected given the sample size and assuming a true effect size of 0. The probability of a test statistic that extreme given a true effect size of 0 is the **p-value**.
4. If the value of the test statistic is rare enough (i.e., the *p*-value is small enough), assume that it was not that large due to chance alone, and reject the null hypothesis of no effect. I.e., declare a “significant” test result.

Frequentist tests vary widely in their assumptions, the nature of their test statistics, and so on, but the procedure above describes most of them.

Another way to think of frequentist statistics is the problem of distinguishing **signal** from **noise**. The biological patterns we want to discover are the signal. Everything else that we measure that is influenced by random variation in the subject or the environment or the measurement process is the noise. Humans are very good at [detecting patterns in noise](#), even if there isn't one.

A real biological pattern should be reliably produce a signal that we can detect. Noise, on the other hand, can produced apparent signals but not reliably or repeatedly. Frequentist statistics is a formalized way of asking the question “How likely is it that this pattern is due to noise?”. If the answer is a sufficiently low probability, then the inference is that the pattern is a signal.

14.2.1 *p*-values and NHST

When scientists design experiments, they are testing one or more **hypotheses**: proposed explanations for a phenomenon. Contrary to popular belief, the statistical analysis of experimental results usually does not test a researcher's hypotheses directly. Instead, statistical analysis of experimental results focuses on comparing experimental results to the results that might have occurred if there was no effect of the factor under investigation. This paradigm is called **null hypothesis significance testing (NHST)** and, for better or worse, is the bedrock of modern scientific inference.

In other words, traditional statistics sets up the following:

- **Null hypothesis (H_0)**: default assumption that some effect (or quantity, or relationship, etc.) is zero and does not affect the experimental results
- **Alternative hypothesis (H_a or H_1)**: assumption that the null hypothesis is false. Usually this is the researcher's scientific hypothesis, or proposed explanation for a phenomenon.

In (slightly) more concrete terms, if researchers want to test the hypothesis that some factor X has an effect on measurement Y, then they would run an experiment comparing observations of Y across different values of X. They would then have the following set up:

- **Null hypothesis:** Factor X does not affect Y.
- **Alternative hypothesis:** Factor X affects Y.

Notice that alternative hypothesis is the **logical negation** of the null hypothesis, not a statement about another variable (e.g., “Factor Z affects Y”). This is an easy mistake to make and one of the weaknesses of NHST: the word “hypothesis” in the phrase “null hypothesis” is not being used in the sense that scientists usually use it. This unfortunate choice of vocabulary is responsible for generations of confusion among researchers and statistics learners.

In order to test an “alternative hypothesis”, what researchers usually do is estimate how likely the data would be if the null hypothesis were true. If a pattern seen in the data is highly unlikely, this is taken as evidence that the null hypothesis should be rejected in favor of the alternative hypothesis. This is not the same as “accepting” or “proving” the alternative hypothesis. Instead, we “reject” the null hypothesis. In the other case, we do not “accept” the null hypothesis, we “fail to reject” it.

The null hypothesis is usually translated to a prediction like one of the following:

- The mean of Y is the same for all levels of X.
- The mean of Y does not differ between observations exposed to X and observations not exposed to X.
- Outcome Y is equally likely to occur when X occurs as it is when X does not occur.
- Rank order of Y values does not differ between levels of X.

All of these predictions have something in common: **Y is independent of X**. That’s what the null hypothesis means. To test the predictions of the null hypothesis, we need a way to calculate what the data might have looked like if the null hypothesis was correct. Once we calculate that, we can compare the data to those calculations to see how “unlikely” the actual data were. Data that are sufficiently unlikely under the assumption that the null hypothesis is correct—a “significant difference”—are a kind of evidence that the null hypothesis is false.

Most classical statistical tests calculate a summary statistic called a **p-value** to estimate how likely some observed pattern in the data would be if the null hypothesis was true. This is the value that researchers often use to make inferences about their data. The p-value depends on many factors, as we’ll see, but the most important are:

- The magnitude of some **test statistic**: a summary of the pattern of interest.
- Sample size, usually expressed as degrees of freedom (DF)
- The variability of the data

- The consistency, sign, and magnitude of the observed pattern

When I teach statistics to biology majors, I try to avoid the terms “null hypothesis” and “alternative hypothesis” at first, because most of my students find them confusing. Instead, I instruct students to ask two questions about a hypothesis:

1. What should we observe if the hypothesis is not true?
2. What should we observe if the hypothesis is true?

In order to be scientific hypothesis, both of those questions have to be answerable. Otherwise, we have no way of distinguishing between a situation where a hypothesis is supported, or a situation where it is not, because experimental results could always be due to random chance (even if the probability of that is very small). All we can really do is estimate how likely it is that random chance drove the outcome.

When interpreting p -values, there are **3 extremely important things to remember**:

1. When $p < 0.05$, the hypothesis is “supported”. When $p \geq 0.05$, the hypothesis is “not supported”. Notice that the terms are “supported” or “not supported”. Words like “proven” or “disproven” are not appropriate when discussing p -values.
2. The p -value is **not** the probability that your hypothesis is false, or the probability that a hypothesis is true. It is a heuristic for assessing how well data support or do not support a hypothesis.
3. p -values depend on many factors, including sample size, effect size, variation in the data, and how well the data fit the assumptions of the statistical test used to calculate the p -value. This means that p -values should not be taken at face value and **never** used as the sole criterion by which a hypothesis is accepted or rejected.

When a statistical test returns a p -value less than 0.05, we say that the results are “statistically significant” and we “reject the null hypothesis”. For example, a “statistically significant difference” in means between two groups, or a “statistically significant correlation” between two numeric variables.

When a statistical test returns a p -value of 0.05 or greater, we say that the test (or difference, or whatever) is “not statistically significant” and we “fail to reject the null hypothesis”. Notice that we do not “accept the null hypothesis”—that would require a different kind of analysis.

14.3 Alternatives to frequentist statistics

14.3.1 Information-theoretic methods

One of the main alternatives to NHST is information-theoretic (IT) modeling. IT methods borrow insights from information theory to draw conclusions about the extent to which statistical models describe data (Burnham

and Anderson 2002). Rather than evaluating individual hypotheses primarily using p -values, IT methods can be used to compare different models in terms of how much information is expected to be lost by using each model to represent the underlying process (Hobbs and Hilborn 2006). In other words, whereas NHST often asks whether particular effects are supported by the data, IT methods emphasize comparing the relative support for entire models.

There are several information criteria in use, but many balance two components: one representing how well the model fits the observed data, based on its likelihood, and another that penalizes the model for complexity (i.e., number of parameters). These information criteria are used to compare models to each other, not to determine whether any particular model is “significant” or whether any of its terms are significant. The complexity penalty is important because adding parameters will generally improve, or at least not worsen, the fit of a model to the observed data, even when those additional parameters have little value for describing the underlying process. Without such a penalty, model selection would therefore tend to favor overly complex models...a phenomenon called **overfitting**.

The most common information criterion is Akaike’s Information Criterion (AIC) (Burnham and Anderson 2002):

$$AIC = -2 \log(\hat{L}) + 2K$$

where $\hat{L}$ is the maximum value of the likelihood function and K is the number of estimated parameters in the model (e.g., slopes and intercepts). Lower values of AIC indicate a better balance between model fit and model complexity. The multiplier of 2 on the K term is the penalty for complexity (i.e., adding additional parameters). This makes adding another parameter supported only if it improves the likelihood enough to overcome the penalty.

Other information criteria exist, such as the AIC corrected for small sample sizes, AIC_C :

$$AIC_C = AIC + \frac{2K(K+1)}{n-K-1}$$

Where n is the number of observations used to fit the model. The correction becomes smaller as sample size increases relative to the number of parameters, so AIC_C converges toward AIC as sample size increases. Exactly what constitutes a “small” sample size is subjective, but Burnham and Anderson (2002) suggest that AIC_C should be used when $n/K < 40$.

Many other information criteria exist, such as $QAIC$ and $QAIC_C$ for use when overdispersion is suspected (Lebreton *et al.* 1992), Bayesian information criterion (BIC) which has a stronger complexity penalty; and deviance information criterion (DIC) which is often used in Bayesian inference.

14.3.1.1 Likelihood?

Before we jump into an example IT data analysis, we also need to understand the concept of the **likelihood function**. A likelihood function describes how well different possible parameter values are supported

by the observed data under a particular statistical model. A statistical model specifies the mathematical relationships among variables and the probability distribution of the observations; examples include a linear regression model or a Poisson GLM.

In symbols, we can write the likelihood function as:

$$\mathcal{L}(\theta|x)$$

where x represents the observed data and θ represents one or more unobserved parameters. For the data we actually observed, the likelihood function asks: which possible values of θ would make these data more or less plausible?

Importantly, likelihood is **not** the probability that a particular parameter value or model is correct. Although $\mathcal{L}(\theta|x)$ is written with the parameters to the left of the vertical bar, it is based on the probability of the data given the parameters, $p(x|\theta)$, not the probability of the parameters given the data, $p(\theta|x)$. The latter is a Bayesian posterior probability (see Section 14.3.2).

The takeaway is that probability asks about data when the model is known, while likelihood asks about parameter values after the data have been observed.

Likelihoods are most famously used in modern data analysis as **maximum likelihood explanation (MLE)**. This procedure attempts to estimate the parameters of a model most likely to produce the observed data.

Here's a quick example: imagine you have measured 5 squirrel femurs and found lengths of 63.6, 66.5, 64.2, 67.6, and 68.3 mm, yielding an observed sample mean and SD of 66.0 ± 2.1 mm. What estimate of the population mean μ that makes this set of measurements most plausible?

```
a <- c(63.6, 66.5, 64.2, 67.6, 68.3)
mean(a)
```

```
[1] 66.04
```

```
sd(a)
```

```
[1] 2.067124
```

Consider 3 candidates for μ : 64, 66, and 68. For any proposed value of μ , `dnorm()` tells us the probability density of observing each femur length under a normal distribution with that mean:

```
samp.sd <- 2.1
dnorm(a, mean=64, sd=samp.sd)
```

```
[1] 0.18655737 0.09352817 0.18911291 0.04370627 0.02334791
```



```
dnorm(a, mean=66, sd=samp.sd)
```

```
[1] 0.09887122 0.18466340 0.13156914 0.14211406 0.10428277
```

```
dnorm(a, mean=68, sd=samp.sd)
```

```
[1] 0.02115483 0.14719781 0.03695464 0.18655737 0.18804388
```

Because we observed more than 1 measurement, and assume that they are independent, the joint likelihood is the product of their densities:

```
prod(dnorm(a, mean=64, sd=samp.sd))
```

```
[1] 3.367191e-06
```

```
prod(dnorm(a, mean=66, sd=samp.sd))
```

```
[1] 3.560036e-05
```

```
prod(dnorm(a, mean=68, sd=samp.sd))
```

```
[1] 4.036932e-06
```

It looks like 66 has the greatest joint likelihood across all observed data. But we can get a fuller picture if we test more values:

```
# series of possible mu to test
test.mu <- seq(60, 72, by=0.01)

# number of possible mu
n <- length(test.mu)

# set up vector to hold likelihood
# function of each candidate mu
lik <- numeric(n)

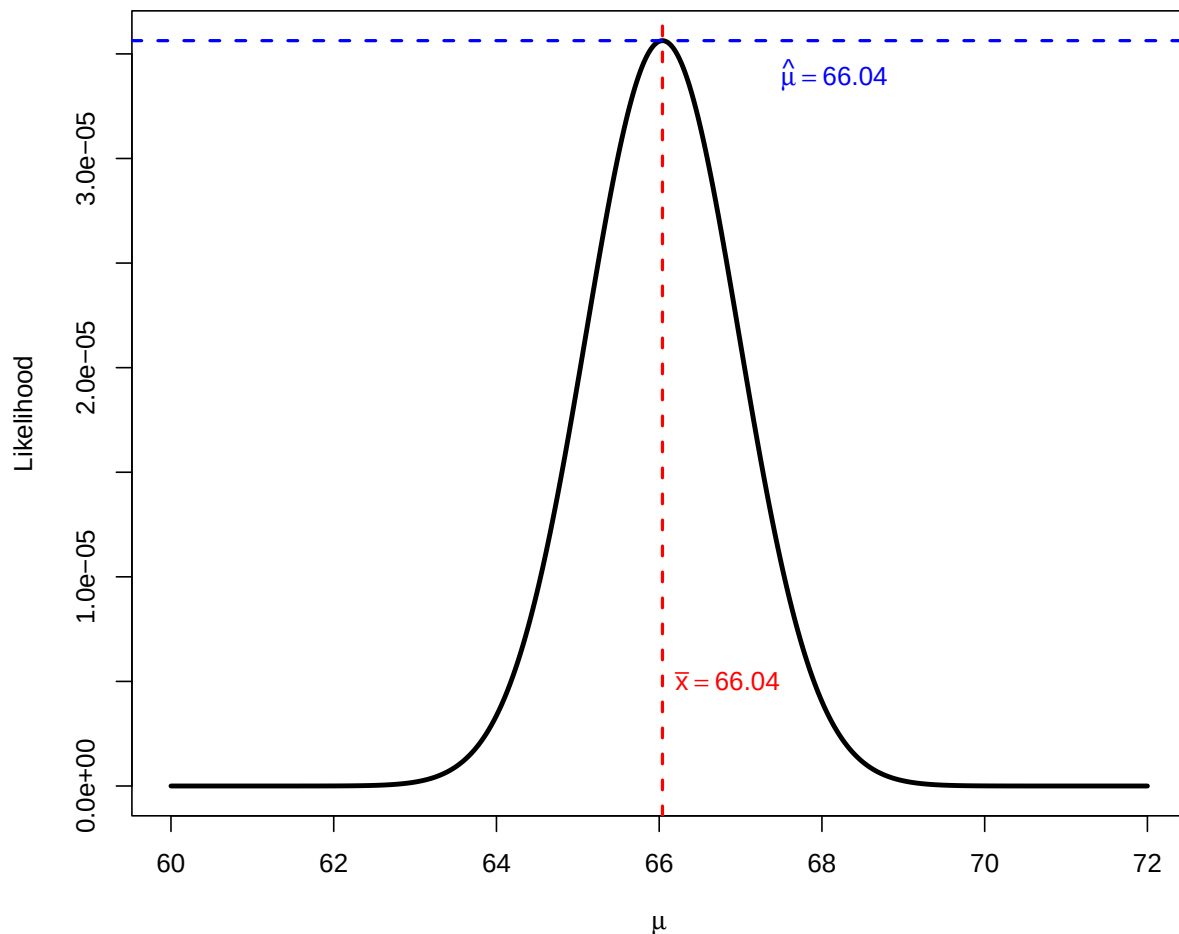
# calculate likelihoods of mu's in a loop
for(i in 1:n){lik[i] <- prod(dnorm(a, test.mu[i], sd=samp.sd))}#i

# now plot the likelihood function vs. mu
```

```

plot(test.mu, lik, type="l", lwd=3,
     xlab=expression(mu),
     ylab="Likelihood")
abline(v=mean(a), lwd=2, lty=2, col="red")
text(66.2, 0.5e-5, expression(bar(x)==66.04), adj=0, col="red")
abline(h=lik[which.max(lik)], lty=2, lwd=2, col="blue")
text(67.5, 3.4e-5, expression(hat(mu)==66.04), adj=0, col="blue")

```



In this case, the sample mean $\bar{x} = 66.04$ (`mean(a)`) is the same as the μ value that maximized the joint likelihood of the 5 observed values, $\hat{\mu} = 66.04$ (`test.mu[which.max(lik)]`). But, the code above only worked because we held the SD to be known and the same as the sample SD. If we didn't, we would have to search over a range of possible SD values, and find the combination of $\hat{\mu}$ and $\hat{\sigma}$ that maximized the likelihood. In that case, the profile of likelihoods would have been a surface, not a line. The maximum likelihood estimates for something like a linear regression model, or a Poisson GLM, are found by similar procedure, only with multiple parameters making the profiling much more complicated.

14.3.2 Bayesian inference

Bayesian inference is a framework for evaluating evidence and *updating beliefs based on evidence*. When conducting Bayesian inference, researchers start with some initial idea about their model system: a **prior**. They then collect evidence (data), and update their idea based on the evidence. The updated idea is called the **posterior**.

- When the prior is strong, or the evidence weak, their ideas will not be updated much and the posterior will be strongly influenced by the prior.
- When the evidence is strong, or the prior weak, the posterior will be updated more and thus less influenced by the prior.

Some researchers prefer to use naïve or uninformative priors, so that their conclusions are influenced mostly by the evidence. Bayesian statistical methods will return essentially the same parameter estimates (e.g., differences in means or slopes) as other methods when uninformative priors are used. This means that almost any traditional statistical analysis can be replaced with an equivalent Bayesian one. The statistical models that are fit, such as linear regression or ANOVA, are the same. All that really changes is how the researcher interprets the relationship between the model and the data.

In frequentist inference, researchers consider parameters like slopes, intercepts, effect sizes, etc., as fixed, unknown parameters and consider the data to occur stochastically following probability distributions. Bayesian inference flips that script, treating the data as fixed, and assuming that the model parameters are unknown quantities following probability distributions.

Bayesian inference depends on **Bayes' theorem**:

$$p(H|E) = \frac{P(E|H) P(H)}{P(H) P(E|H) + P(\neg H) P(E|\neg H)} = \frac{P(E|H) P(H)}{P(E)}$$

where

- H is a hypothesis that can be true or false
- $p(H)$ is the prior, an estimate of the probability of the hypothesis being true before any new data (E) are observed
- E is evidence, specifically new data used to update the prior
- $p(H|E)$ is the probability of H given E . I.e., the probability of the hypothesis being true after new evidence is observed. This is also called the **posterior probability**.
- $p(E)$ is the probability of E regardless of H . I.e., the probability of observing the evidence whether or not the hypothesis is true. $p(E)$ is called the **marginal likelihood**.
- $p(E|H)$ is the probability of observing the evidence E given H . I.e., the probability of observing the evidence if the hypothesis is true. $p(E|H)$ is also called the **likelihood**.

Bayes' theorem is powerful because it is very general. The precise nature of E and H do not matter. All that matters is that one could sensibly define an experiment or set of observations in which E^* might depend on H and vice versa.

Bayesian inference is attractive to many biologists because it mimics the way that scientists actually think: starting with one idea (a prior), collecting evidence, then updating that idea (posterior). Part of the Bayesian calculation is $p(H|E)$, or the probability of the hypothesis being true given the evidence that is observed. Contrast this with the frequentist p -value, which is $p(E|\neg H)$. Most people instinctively want the first, but have to settle with the almost backwards logic of the p -value.

Here is a quick example of Bayesian reasoning. Imagine you go to the doctor and get tested for a rare condition, [Niemann-Pick disease](#). Your doctor tells you that this disease affects about 1 in 250000 people. Unfortunately, you test positive. But, the doctor tells you not to worry because the test is only 99% reliable. Why is the doctor so sanguine, and what is the probability that you have Niemann-Pick disease?

The probability of having this disease can be estimated using Bayes' theorem. First, use what you know to assemble the terms:

- The overall incidence is 1 in 250000 people, or 0.0004%. So, the prior probability $P(H)$ is 0.000004.
- The probability of a positive test given that you have the disease, $P(E|H)$, is 0.99 because the test is 99% reliable.
- The denominator, $P(E)$, is the unconditional probability of a positive test.

The denominator includes two possibilities: either a positive test when someone has the disease, or a positive test when someone doesn't have the disease. That is, the probability of a true positive or a false positive. This can be calculated as:

$$p(H)(E|H) + p(\neg H)p(E|\neg H)$$

The first term is the probability of a true positive ($p(E|H)$) scaled by the probability of having the disease ($p(H)$). The second is the probability of a false positive ($p(E|\neg H)$) scaled by the probability of not having the disease ($p(\neg H)$). We can simplify a bit by reasoning that $p(\neg H)$ is really $1 - p(H)$.

Plug every thing into Bayes' rule:

$$p(H|E) = \frac{P(E|H) P(H)}{P(H) P(E|H) + (1 - P(H)) P(E|\neg H)} = p(H|E) = \frac{(0.99)(0.000004)}{(0.000004)(0.99) + (0.999996)(0.01)}$$

and simplify:

$$p(H|E) = \frac{0.00000396}{0.01000392} = 0.000396$$

That's not very worrying! In this example, the test evidence can't overcome the fact that the disease is so rare. In other words, even among people who get a positive test, it's far more likely that you don't have the disease but got a false positive than that you have the disease and got a true positive. This is an example of a very strong prior being more influential than the evidence.

There's an issue with this calculation, though. The prior probability of 0.000004 was based on the assumption that the test was conducted on a random person from the population. Do physicians conduct tests for vanishingly rare diseases on random people? Of course not. Maybe this physician only orders this test if they think that there is a 10% chance that you really have the disease. This might be because, in their experience, 10% of people with your symptoms turn out to have the disease. In that case, the calculation becomes:

$$(H|E) = \frac{(0.99)(0.1)}{(0.1)(0.99) + (0.9)(0.01)}$$

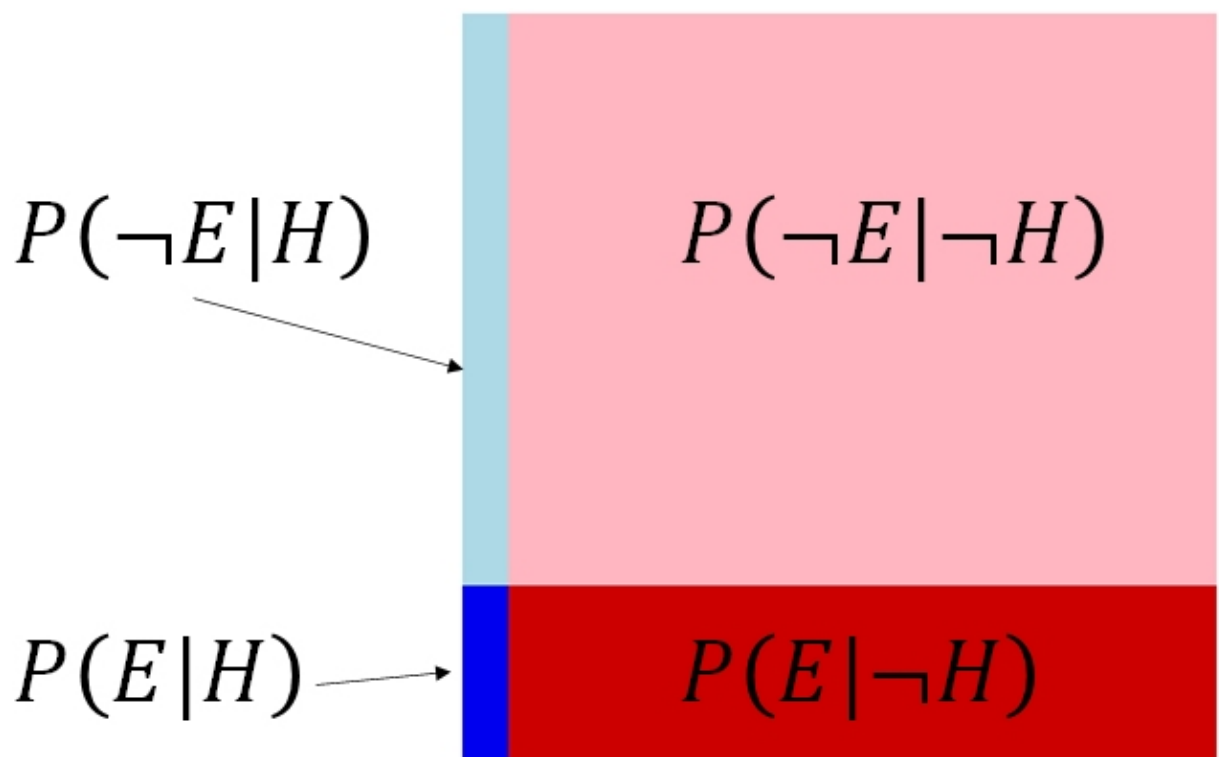
which simplifies to:

$$p(H|E) = \frac{0.099}{0.108} = 0.916$$

91.6% is a lot more worrying than 0.0004%. This example shows that the prior probability can have a huge effect on the posterior probability.

One way to visualize Bayes' rule is to think of the entire probability space as occupying a square with area 1. The square has 4 sections:

Figure 14.3: Bayes theorem helps us reason about the relative probabilities of observing a piece of evidence E or not when a hypothesis H is true or not true. The area of all four sections must add up to 1. According to Bayes' theorem, the probability of a hypothesis H being true, given evidence E, is the ratio of $P(E|H)$ to $P(E|H) + P(E|\neg H)$.



Notice that the prior probability $p(H)$ (the two left, blue sections) is much smaller than the probability of $\neg H$. Bayes' rule restricts our focus to the cases where we observe the evidence—the bottom two sections. Given that the evidence was observed, the sections with $\neg E$ are irrelevant. The calculation then becomes:

$$\frac{P(E|H)}{P(E|H) + P(E|\neg H)}$$

Besides its epistemological neatness, another advantage of Bayesian inference in practice is that parameters in Bayesian models (e.g. slopes, intercepts, etc.) are estimated with **credible intervals (CRI)**, not **confidence intervals (CI)**.

- A frequentist 95% confidence interval is a statement about the long run performance of a series of repeated experiments. If we repeatedly collected new samples and calculated intervals the same way, about 95% of those intervals would contain the true parameter. This is not the same as saying that there is a 95% probability that the true parameter value lies within our estimated CI.
- A Bayesian 95% credible interval is a statement about the parameter itself. Given the data, model, and prior, there is a 95% probability that the parameter lies within the interval. That is, a credible interval does what most people mistakenly think the confidence interval does!

I'm a big fan of Bayesian inference. But, I'll admit that there is a downside: Bayesian model-fitting is not available in base R, and requires specialized packages and sometimes external software. Should you go through all of this Bayesian trouble? There are a few reasons where Bayesian inference might be preferable to traditional frequentist statistics.

1. The model to be fit is very complex. Because the user can specify any model they want, they can fit any model they want.
2. Maximum-likelihood methods often require explicit derivations of model likelihoods, whereas an MCMC approach does not (MCMC, or Markov-chain Monte Carlo, is a common method for fitting Bayesian models). This means that models whose likelihoods have no closed form solution, or have computationally difficult solutions, can be estimated more easily with Bayesian methods than maximum likelihood (sometimes).
3. The researcher has prior information that they would like to incorporate into the analysis. This can reduce the amount of data needed to reach certain conclusions.
4. Maximum likelihood methods are not effective in some situations.
5. Error propagation is very easy under Bayesian inference (especially MCMC).

6. The researcher prefers the philosophy of Bayesian inference.

Bayesian inference is beyond the scope of this course, but if you're interested head on over to my course notes for BIOL 7950: Bayesian Inference for Biologists.

14.4 Linear models

14.4.1 What is “linear”?

Linear models are called *linear* models because they can be expressed as **linear combinations** of a set of estimated parameters. A linear combination is an expression formed by multiplying each term by a constant, and adding the products together. For example,

$$ax + by$$

is a linear combination of x and y , where a and b are constants. The x and the y could be single values, or, as commonly used in statistics, **vectors** of values. A vector in this sense is an ordered set of values. One example of a linear combination that you're already familiar with is a **weighted mean**. The weighted mean of a set of numbers $[x_1, x_2, \dots, x_n]$ is given by the sum

$$\bar{x}_w = \sum_i^n \left(\frac{w_i}{\sum_i^n w_i} \right) x_i$$

Because the sum of all weights must be 1, this simplifies to:

$$\bar{x}_w = \sum_i^n (w_i) x_i$$

meaning that the weights are the coefficients, and the values of x are the terms. Actually, the mean itself is a linear combination, where all weights are equal to $1/n$.

To generalize, a linear model consists of the sum of the products of an input vector and a series of coefficients of the same length. In a simple linear regression, the coefficient is the slope: the expected change in Y associated with a unit increase in X .

Importantly, a linear model must be *linear in its parameters*, not necessarily in the predictor variables themselves. For example, the model $y = \beta_0 + \beta_1 x + \beta_2 x^2$ is nonlinear in x because the x^2 forces a curved graph. But, if x^2 increases by 1, then y will increase by β_2 . This is what it means to be “linear in its parameters”.

All linear models can be written very compactly in matrix notation:

$$\mathbf{Y} = \mathbf{X}\beta + \varepsilon$$

The boldface variables indicate that each term is a matrix or a vector, not a variable in the usual sense. In most biological analyses, $\mathbf{Y}$, β , and ε are column vectors, which are like matrices with 1 column. The matrix $\mathbf{X}$ is the **design matrix**, which contains the values of the explanatory variables used to fit the model.

The notation above may be compact, but expanding everything out makes it clearer what is going on:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

where n is the number of observations, and p is the number of predictors. The leftmost column in $\mathbf{X}$, consisting of all 1s, are what gets multiplied by β_0 to produce the intercept. For example, the expected value for the first value of the response y_1 is equal to:

$$E(y_1) = \beta_0(1) + \beta_1(x_{11}) + \beta_2(x_{12}) + \dots + \beta_p(x_{1p})$$

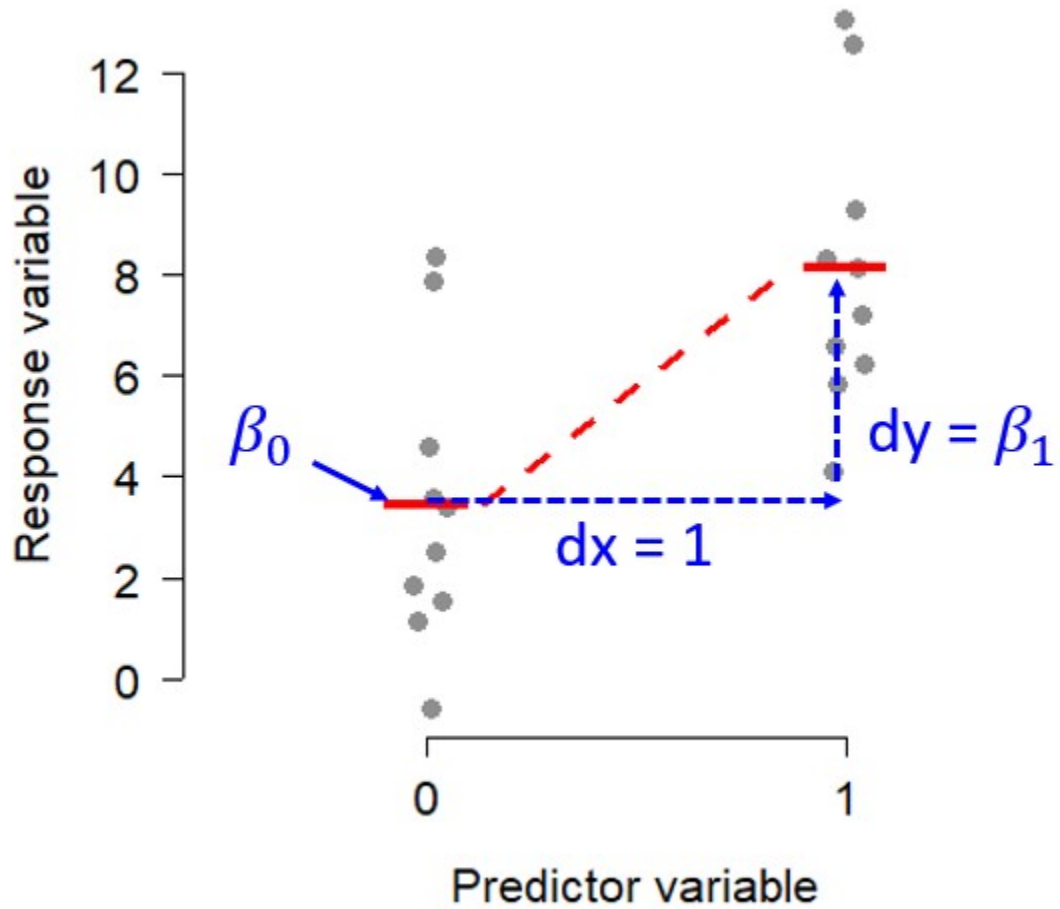
where x_{ij} is the x value for observation i , predictor variable j . The observed value of y_i the expected value plus some random deviation ε_i , which comes from a normal distribution with mean 0 and SD σ_{res} (“res” is for “residual”).

14.4.1.1 Factors in linear models

This notion of linearity is what makes many analyses with **factors** (categories or grouping variables) as predictors “linear”. For example, the t -test is a linear model, but it doesn’t really describe a line on a plot of y vs. x the way that linear regression does. However, if you encode the factor (grouping variable) it should be clear what the difference in group means really means:

$$E(y_i) = (1 \times \beta_0) + (x_i \times \beta_1)$$

For group 1, $x_1 = 0$, so the expected value is β_0 . For group 2, $x_i = 1$, so the β_1 term comes into play as the difference in means between groups 1 and 2. This is what R, SAS, and other programs are doing internally when you analyze data with a factor variable. Here is what that looks like visually:



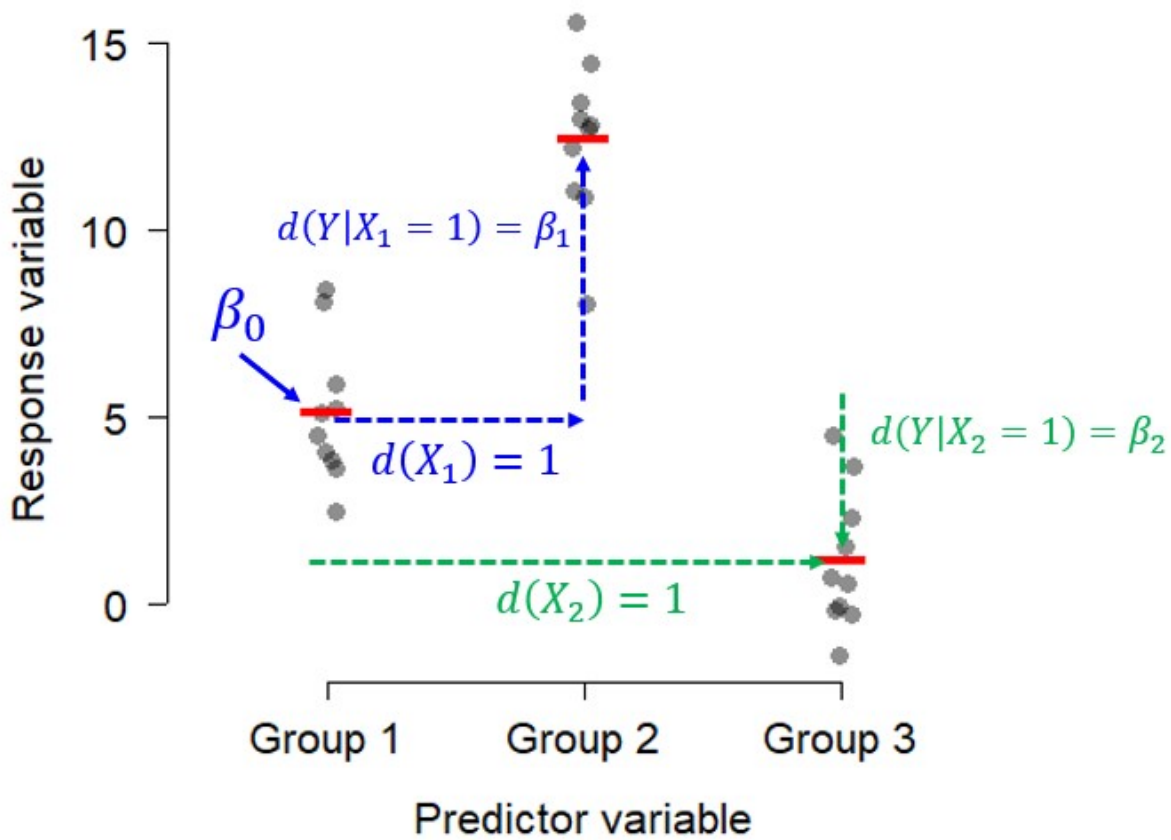
one-way ANOVA with 3 levels can be thought of this way:

Similarly, a

$$E(y_i) = (1 \times \beta_0) + (x_{1,i} \times \beta_1) + (x_{2,i} \times \beta_2)$$

And visualized this way:

Figure 14.4: Illustration of how a one-way ANOVA with 3 levels qualifies as a linear model.



14.4.1.2 Linear models and sums of squares

There are tangible and intangible benefits to using linear models. First, as shown above, linear models can be written very compactly in the language of linear algebra (i.e., the branch of mathematics that deals with certain types of operations on matrices and vectors). From a practical standpoint, this forces us to organize our datasets sensibly, into matrices with rows that describe observations and columns that describe variables. From a computational standpoint, this makes the computer's job much more efficient because of the architecture of modern computers.

Second, and less obvious, is that the model coefficients of a linear model can be **solved for**, rather than approximated. When we fit a linear model to data, we are really after the values in the coefficients matrix. A modern statistical program can solve for these values rather than having to estimate them. For example, a simple linear regression with 1 predictor variable X can be solved by finding the intercept β_0 and β_1 that together minimize the sum of squared residuals SS_{res} . These coefficients are:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

to yield the effect of x_1, β_1 . The intercept is then:

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

The caret symbol $\hat{}$ indicates that a value is an “estimate”, even when it is solved for from data (e.g., $\hat{\beta}_1$ is pronounced “beta-one-hat” and is the estimated slope). Similarly, the bar above a letter indicates the sample mean (e.g., $\bar{x}$ is pronounced “x bar” and is the mean of all observed x).

14.4.1.3 Assumptions of linear models

Linear models have some other features and assumptions that you need to be aware of. If your data or variables violate the assumptions below, you may need to use something other than linear models to analyze your data.

14.4.1.3.1 Assumption 1: Normal residuals

The linear model assumes that model residuals, the random errors representing the differences between observed and predicted values, follow a normal distribution. Models for data that do not have normal residuals exist, but they are not linear models. We’ll explore those in Chapter 18.

The assumption of normal residuals explains the last term in the linear model equation, ε_i :

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

In this equation, y_i is observation i of the response variable y ; β_0 is the intercept; β_1 is the slope; x_i is the predictor variable x for observation i , and ε_i is the residual for observation i (ε is the Greek letter “epsilon” and usually stands for error or residual terms). The residual term can be expanded like this:

$$y_i = \beta_0 + \beta_1 x_i + \text{Normal}(0, \sigma_{res})$$

This expansion makes it clear that each residual ε_i comes independently from a normal distribution with mean 0 and standard deviation (SD) σ_{res} . For convenience, we sometimes separate the deterministic part and the stochastic part of the model and write the **state space** form of the linear regression model:

$$y_i \sim \text{Normal}(\eta_i, \sigma_{res})$$

$$\eta_i = \beta_0 + \beta_1 x_i$$

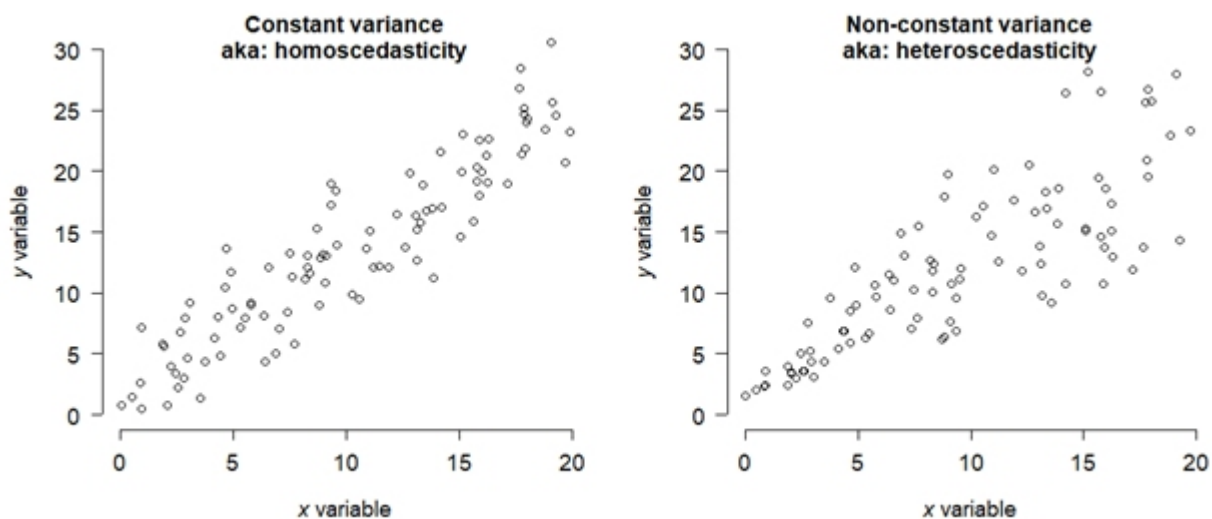
In this version, the symbol $\sim$ means “distributed as” and denotes a **stochastic** relationship—a relationship that contains some element of randomness. This is in contrast to a **deterministic** relationship, which does not (equality, aka: identity, $=$ is the most famous kind of deterministic relationship).

14.4.1.3.2 Assumption 2: Homogeneity of variance

Linear models assume **homoscedasticity**, or constant variance. This means that the residual variance does not depend on either x or y . If the variance in the response variable depends on some predictor variable, then this should be incorporated into the model (resulting in something other than a linear model). If the variance appears to depend on the response variable, then there is probably an issue with the assumed response distribution.

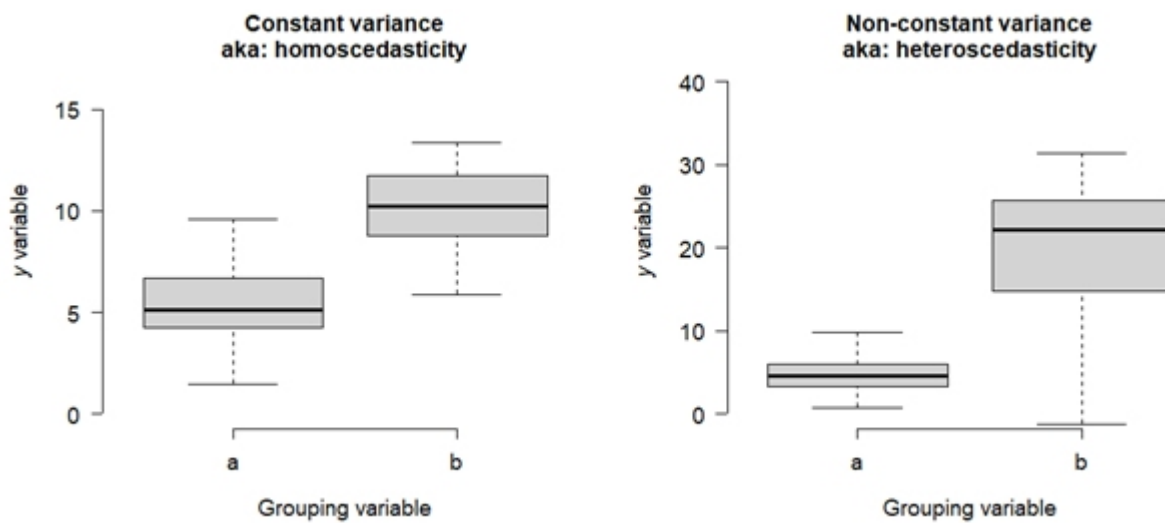
The figure below shows two datasets with a linear relationship between x and y . In the left panel, the variance is the same everywhere (homoscedastic). The right panel shows a heteroscedastic relationship where the variance increases at larger x .

Figure 14.5: Example of heteroscedasticity, or unequal variance, across the domain of x . The left panel shows homoscedasticity, where residual variance is approximately the same for all x . The right panel shows heteroscedasticity, where residual variance increases with x .



The figure below shows two datasets with a categorical predictor, suitable for a t -test. The left panel shows a relationship with equal variances in each group, while the right panel shows a situation with unequal variances.

Figure 14.6: Example of heteroscedasticity, or unequal variance, between two levels *a* and *b* of some factor. The left panel shows homoscedasticity, where residual variance is approximately the same in group *a* and group *b*. The right panel shows heteroscedasticity, where residual variance is different in each group.



Heteroscedasticity is a serious problem for linear models because it leads to biased parameter estimates and standard errors (SE) of those estimates. The latter issue means that significance tests on parameters will be incorrect. Bayesian statistics does not involve significance tests, but heteroscedasticity that is unaccounted for will still lead to biased estimates. The usual solution is to either apply a variance-stabilizing transformation (like the log-transform), incorporate variance parameters that can be estimated, or be clever about choosing a response distribution.

14.4.1.3.3 Assumption 3: Fixed and independent predictors

Linear models assume that predictor values are precisely known (“fixed”) and independent of each other. If there is uncertainty in the predictor variables, this adds uncertainty to the response values that linear models cannot account for. Simply put, linear models have a term for uncertainty in y , σ_{res} , but no term for uncertainty in x .

14.4.1.3.4 Assumption 4: Independently and identically distributed errors

(*i.i.d.*)

The assumption of independently and identically distributed (*i.i.d.*) errors is **very** important. It means that the residual for each observation depends only on the parameters of the residual distribution and **not** on predictor variables or other observations. When the assumption of independence is violated, the degrees of freedom in the analysis is artificially inflated because the number of unique pieces of information is smaller than the number used in the analysis. In frequentist analyses, this deflates the p -value and increases the chance of a type I error (false positive).

14.4.1.3.5 Final note about linear model assumptions

Real data almost always violate the linear model assumptions to some degree, so the answer to “do my data meet the assumptions of linear models?” is “No.”. The more useful question is, “do my data violate the assumptions of linear models *enough to be a problem?*”. Linear models can be robust to *mild* violations of these assumptions, but you need to be careful. Many of these assumptions can be tested directly—for example, testing residuals for normality or homoscedasticity—and if there is any doubt about whether your data meet the assumptions, you should perform those tests. Actually, you should perform the tests anyway just to be safe.

14.5 Analysis of variance (ANOVA)

The term **analysis of variance (ANOVA)** is used in two related ways, and we need to clear them up before the next module. In introductory statistics, ANOVA usually refers to a linear model used to compare means among groups, such as comparing mean body mass among three species of frog. If someone says something like, “we compared the treatments using ANOVA”, this is the sense they mean. We will explore this sense of ANOVA in Section [15.5.1](#).

More generally, ANOVA refers to a framework for evaluating how variation is attributed to different components of a statistical model. This is called “partitioning the variance” or “partitioning sums of squares”. For example, an ANOVA table can be used to test a linear regression even when the predictor is continuous and there are no groups to compare. The function of ANOVA in that case is to determine how much variation is explained by the different predictors—be they continuous or factors—in the model and how much variation is left over (residual). We will explore this sense of ANOVA in Section [15.5.1.1](#).

This broader usage is reflected in R’s `anova()` function, which can be applied to many kinds of models. For ordinary linear models, ANOVA tests are based on partitioning sums of squares. For generalized linear models, analogous tests are commonly based on changes in **deviance** rather than sums of squares. Deviance is a generalization of variance (which is calculated from sums of squares), so sums of squares are a special case of deviance (Section [18.4](#)). This reflects the fact that linear models based on sums of squares are a special case of generalized linear models (Chapter [18](#)).

Chapter 15

Tests for differences in mean or location

In a test for difference in mean or location, we are interested in whether observations differ between groups. Most of the time we are testing whether the central tendency differs between groups. A test of this kind evaluates a prediction like, “a typical value in group A is different from a typical value in group B”.

Alternatively, we can test for a difference in **location**: the relative position of random values from 2 or more groups. This is a slightly different statement: “values in group A tend to be greater than (or less than) values in group B”. Put another way: “If we pick a random value from group A, it will be reliably greater (or less than) than a random value from group B”. Tests for location are performed on the **rank order** of the values, not the values themselves.

When comparing groups, the data are usually best shown using a **boxplot**, also known as a **box-and-whiskers** plot. The exact implementation varies between software packages, but boxplots usually show the median, interquartile range (approximately), extreme values, and possible outliers (if any). Run the command `?boxplot.stats` to see a description of how boxplots are calculated in R.

15.1 Tests for 1 group

When all of the values come from the same set of observations, we can ask questions such as:

- Is the mean in this group equal to, or different from, some arbitrary value?
- Is the mean in this group greater than, or not greater than, some arbitrary value?

These questions are the domain of **one-sample tests**. The most important is the **one-sample t-test**, which comes in two varieties: **one-tailed** and **two-tailed**. All one-sample tests compare the mean of a set of values to some quantity (hypothetical mean) of interest, called μ (the Greek letter “mu”). In a one-tailed test,

the null hypothesis is that the underlying population mean is either $\leq \mu$ or $\geq \mu$. In a two-tailed test, the null hypothesis is that the population mean is equal to μ .

A note on symbology: sample vs. population parameters

In statistics, we analyze **samples** (randomly-selected subsets of observed entities) to make inferences about **populations** (the hypothetical set of all possible entities which could be included in a sample). For example, if you wanted to know how the population of a nation of 100 million citizens felt about some political issue, it would be time-consuming and expensive to ask each citizen what they thought. Instead, you would sample a randomly selected set of 1000 citizens and then assume that those results reflected the opinion of the entire citizenry. For this assumption to work, your sample must be both random and representative of the entire population.

Similarly, in science we cannot collect an infinite amount of data, or measure every squirrel in the forest, or any other type of exhaustive and comprehensive data collection. We instead take representative samples of all the possible samples—i.e., the population—and rely on good practices in experimental design to enable us to assume that our sample represents the population.

The reason I bring this up here is that we need to keep in mind the difference between sample statistics and population statistics. **Sample statistics** are calculated from your data. **Population statistics** are **inferred** from your data using statistical models. This is why statistical tests are commonly referred to as **inferential statistics**, and why simple calculations from data are referred to as **descriptive statistics**.

The most common descriptive statistics are shown below:

Parameter	Sample	Population
Mean	$\bar{x}$	μ
Variance	s^2	σ^2
Standard deviation	s	σ

15.1.1 One-sample, one-tailed *t*-test

In a one-tailed one-sample test, we are interested in whether the underlying population mean (μ) of a set of values is not equal to some value of interest...either at most some value, or at least some value.

Question: Is the mean value less than OR greater than some value x ?

Null hypothesis: True mean is $= x$.

Alternative hypothesis: True mean is $< x$ or true mean is $> x$.

Example use case: Fred needs to check whether the mean measurement error across his 12 technicians is less than 0.5 mm.

Here is an example where we simulate some data in R and run a one-tailed one-sample *t*-test.


```
# simulate some values with nominal mean 5 and sd 3
set.seed(123)
a <- rnorm(20, 5, 3)

# 1 sample 1 tailed t-test
# Example 1: Is the underlying mean <= 0?
t.test(a, alternative="greater")
```

One Sample t-test

```
data: a
t = 8.3142, df = 19, p-value = 4.707e-08
alternative hypothesis: true mean is greater than 0
95 percent confidence interval:
 4.29664      Inf
sample estimates:
mean of x
 5.424871
```

```
# Example 2: Is the underlying mean <= 4?
t.test(a, alternative="greater", mu=4)
```

One Sample t-test

```
data: a
t = 2.1838, df = 19, p-value = 0.02086
alternative hypothesis: true mean is greater than 4
95 percent confidence interval:
 4.29664      Inf
sample estimates:
mean of x
 5.424871
```

Interpretation: In the example above, the researcher can reject the null hypothesis that the true mean is ≤ 4 , and conclude that the true mean is > 4 .

```
# Example 3: Is the underlying mean >= 5.5?
t.test(a, alternative="less", mu=5.5)
```

One Sample t-test

```
data:  a
t = -0.11514, df = 19, p-value = 0.4548
alternative hypothesis: true mean is less than 5.5
95 percent confidence interval:
      -Inf 6.553102
sample estimates:
mean of x
  5.424871
```

Interpretation: In example 3, the researcher cannot reject the null hypothesis that $\mu \geq 5.5$. This makes sense, because the true mean is about 5.4 ± 2.9 (verify with `mean(a)` ; `sd(a)`).

15.1.2 One-sample, two-tailed *t*-test

In a two-tailed test, we are interested in whether the underlying population mean of a set of values is **equal** to some value of interest.

Question: Is the underlying true mean equal to, or different from, some value μ ?

Null hypothesis: True mean = μ .

Alternative hypothesis: True mean $\neq \mu$

Example use case: Wilma needs to test whether the mean pH of her prepared stock solution is equal to, or different from, the target pH of 7.7.

Here is an example where we simulate some data and conduct a one-sample, two-tailed *t*-test.

```
# simulate data
set.seed(123)
a <- rnorm(20, 3, 1)

# 1 sample 2-tailed t-test:
## Example 1: mu = 0 (default)
t.test(a)
```

One Sample *t*-test

```
data:  a
t = 14.445, df = 19, p-value = 1.067e-11
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
  2.686402 3.596845
sample estimates:
```

```
mean of x
3.141624
```

Interpretation: In the example above, the researcher can reject the null that the true mean = 0, and conclude that the true mean is significantly different from 0.

Here's another example with a different dataset and test.

```
# simulate data
set.seed(123)
a <- rnorm(20, 3, 1)

# 1 sample 2-tailed t-test:
## Example 2: mu = 5
t.test(a, mu=5)
```

One Sample t-test

```
data: a
t = -8.5445, df = 19, p-value = 6.218e-08
alternative hypothesis: true mean is not equal to 5
95 percent confidence interval:
 2.686402 3.596845
sample estimates:
mean of x
3.141624
```

Interpretation: The researcher can reject the null hypothesis that the mean = 5, and conclude that the true mean is not 5. Note that this includes both < 5 and > 5 as possibilities, but the direction is obvious from the sample mean.

15.2 Tests for 2 groups

15.2.1 Student's *t*-test (Welch) (2-tail, 2-sample)

When most people say “*t*-test”, what they usually mean is a two-sample, two-tailed *t*-test that tests for a **difference in means between two groups**. The original version was introduced as the **Student's *t*-test** by William Sealy Gosset in 1908¹; modern software uses a modified version called the **Welch *t*-test** (or sometimes the “**Welch-Satterthwaite *t*-test**”, but this is not quite correct). The newer version is better at accounting for non-normality and unequal variances.

¹Gosset published his work under the pen name “Student” because his employer at the time, the Guinness Brewery, didn't want their competitors to know that they used the *t*-test in their quality control procedures.

The test statistic, t , is basically the difference in sample means scaled by the variance in each group and the sample size in each group. It is calculated as:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

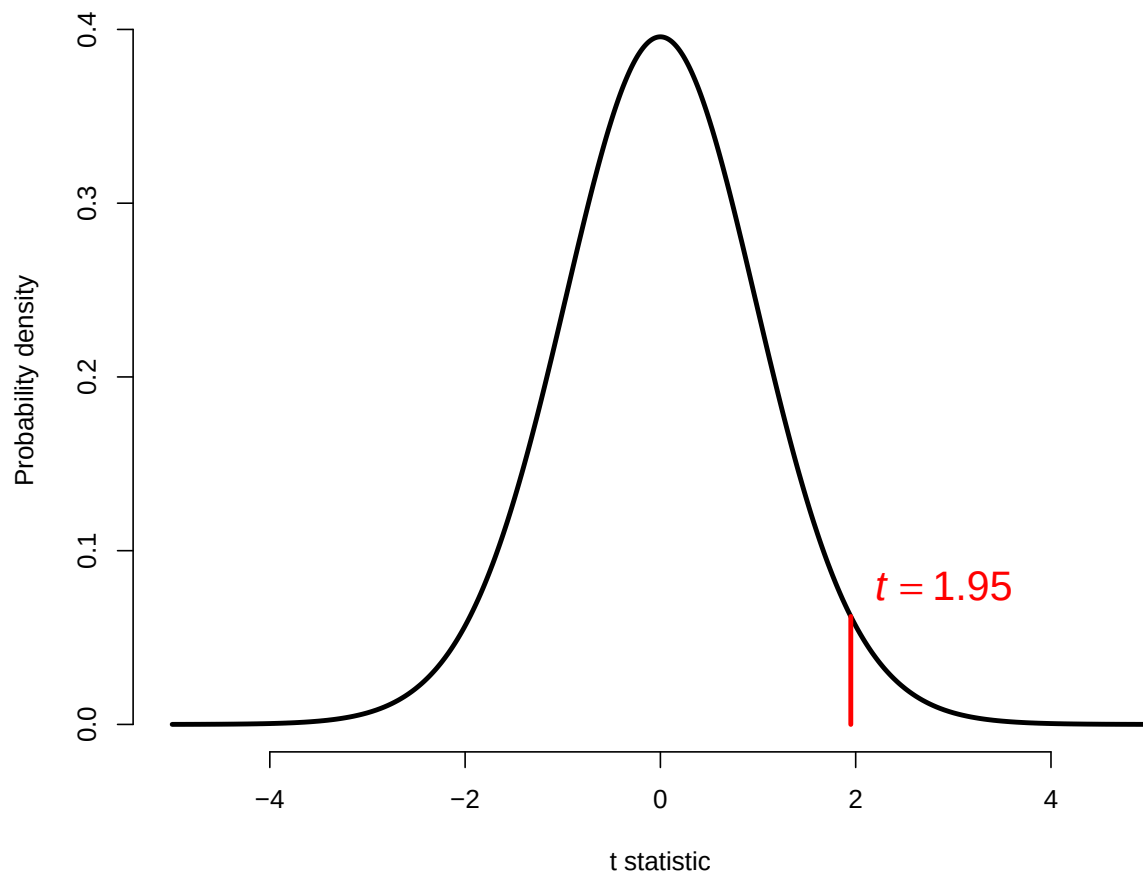
where $\bar{x}_1$ and $\bar{x}_2$ are the sample means, s_1^2 and s_2^2 are the sample variances, and n_1 and n_2 are the sample sizes. Think through that equation...what would increasing the difference in means (the numerator) do to t ? What about increasing one or both sample sizes? One or both sample variances?

Basically, the numerator of t is the quantity of interest: the difference in means between two groups. The denominator is the standard error (SE) of that estimate.

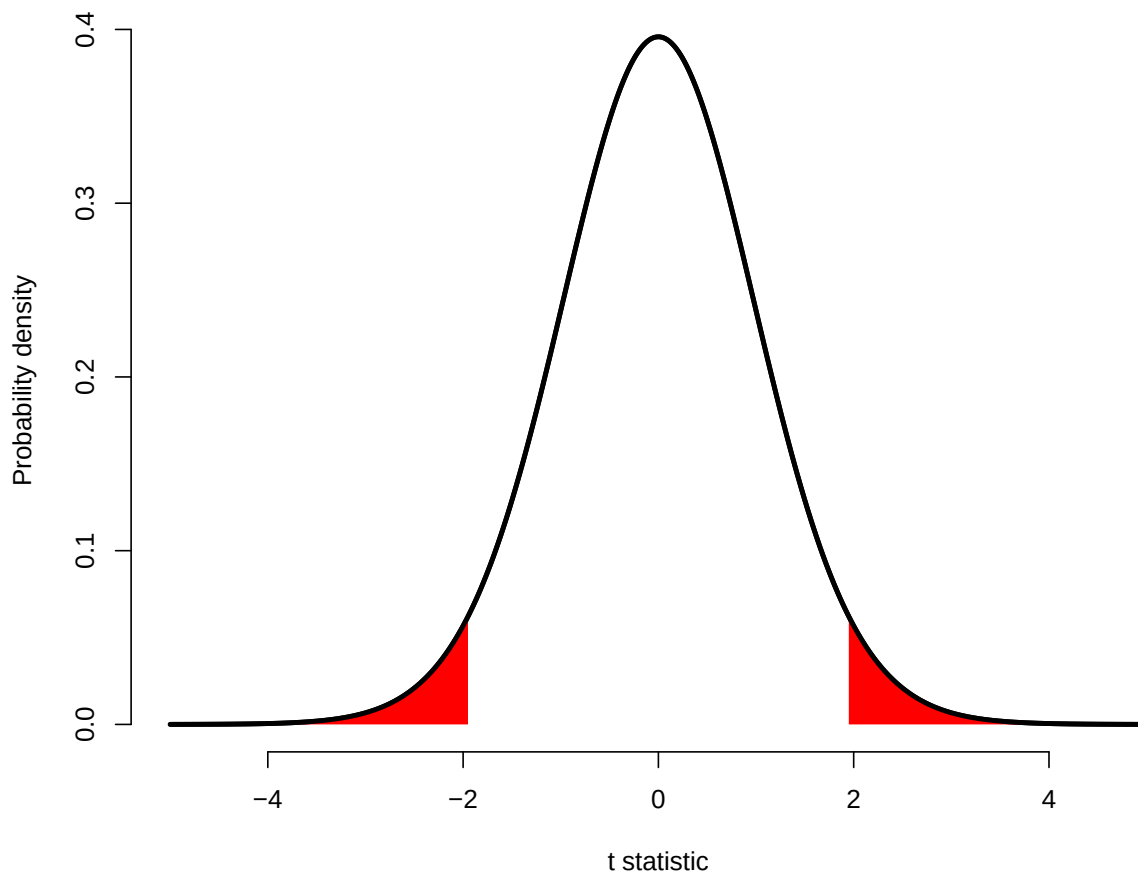
A t value by itself doesn't tell you much, though, because what we're interested in is how likely that t score is relative to the number of degrees of freedom (DF) in the dataset. While the early users would simply use $n-1$ as the DF, modern t -statistics and t distributions are defined by a newer version of the degrees of freedom calculation:

$$df = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{(s_1^2/n_1)^2}{n_1-1} + \frac{(s_2^2/n_2)^2}{n_2-1}}$$

Say you perform a test comparing groups A and B, and the t statistic comes out to be 1.95. We have to compare that to the t distribution with the number of DF contained in the data, which might be 31.2:



The question for the hypothesis test is: in a dataset with 31.2 DF, how often should we observe t values at least as large as 1.95 (i.e., $|t| \geq 1.95$)? That proportion is the area shaded below:



The area of the left shaded area is $\text{pt}(-1.95, 31.2)$, which is 0.0301. The distribution is symmetrical, so $p(|t| \geq 1.95) = 0.0301 \times 2 = 0.0602$. Thus, the test was not significant ($t = 1.95$, 31.2 d.f., $p = 0.06$).

Question: Do two groups have the same underlying population mean?

Null hypothesis: The true difference in means = 0. i.e., $\mu_1 = \mu_2$.

Alternative hypothesis: True difference in means is $\neq 0$. i.e., $\mu_1 \neq \mu_2$.

Example use case: Barney needs to test whether tomato plants treated with a new pesticide have a different yield (g tomato / plant) than plants in the untreated control group.

Below is an example where we simulate some data and run a two-sample, two-tailed test to compare the means of two groups.

```
# simulate data
set.seed(42)
a <- rnorm(100, 5, 2)
b <- rnorm(100, 3, 2)
```

```
# example 1: difference in means
t.test(a,b)
```

Welch Two Sample t-test

```
data: a and b
t = 8.1211, df = 194.18, p-value = 5.241e-14
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1.696004 2.783990
sample estimates:
mean of x mean of y
 5.065030  2.825033
```

Interpretation: The researcher should conclude that there is a statistically significant difference between the means of groups a and b. In a manuscript, you should report the test statistic, the DF, and the p -value. E.g., “There was a significant difference in means between group a and group b ($t = 8.121$, 194.18 d.f., $p < 0.001$).” Notice that values are rounded to 2 or 3 decimal places, which is usually enough for biology. This is especially true for the p -value, which can be approximated by R down to infinitesimally small values. I always round those to 0.001 or 0.0001.

A t -test can also be performed using the **formula** interface, which is an R programming construct for specifying a response variable and its predictors. Notice that the formula interface here is the same as used in `boxplot()` (Section 8.1.3) and `aggregate()` (Section 7.1.1).

Syntax note: Formula syntax has the response variable on the left, then a `~`, then the predictor or predictors. For predictors, `a + b` means an additive model; while `a * b` specifies an interaction. For a t -test, there can be only 1 predictor. If you want to test > 1 predictor at a time, you need to use ANOVA instead.

Here are some common formula examples:

Table 15.1: Common R model formulas and their meanings.

Formula	Meaning
<code>y ~ x</code>	y is modeled as a function of x .
<code>y ~ x1 + x2</code>	y is modeled as a function of $x1$ and $x2$, with their effects treated as additive (i.e., no interaction between them is included).
<code>y ~ x1 * x2</code>	y is modeled as a function of $x1$, $x2$, and their interaction . An interaction means that the effect of one predictor depends on the value of the other predictor. This is shorthand for <code>y~x1 + x2 + x1:x2</code> .

Formula	Meaning
$y \sim 1$	y is modeled using only a constant (the intercept), with no predictors. This is often called an intercept-only or null model .
$y \sim 0 + x$	y is modeled as a function of x , but without an intercept (i.e., the intercept is constrained to 0).

The formula interface is used in several contexts in R, including plotting, modeling, and aggregating.

```
# pull some data from example dataset iris
# i.e., make a new data frame with only 2 groups
dx <- iris[which(iris$Species %in% c("setosa", "versicolor")),]

# perform the test.
t.test(Petal.Length~Species, data=dx)
```

Welch Two Sample t-test

```
data: Petal.Length by Species
t = -39.493, df = 62.14, p-value < 2.2e-16
alternative hypothesis: true difference in means between group setosa and group versicolor
95 percent confidence interval:
 -2.939618 -2.656382
sample estimates:
 mean in group setosa mean in group versicolor
           1.462             4.260
```

Interpretation: The researcher should conclude that there is a statistically significant difference between the mean petal length of *Iris setosa* and *Iris versicolor*.

15.2.2 Two-sample, one-tailed t-tests

Question: Is the true difference in means between 2 groups greater than, or less than, some target value μ ?

Null hypothesis: The true difference in means between 2 groups is μ . I.e., $\mu_1 - \mu_2 = \mu$.

Alternative hypothesis: The true difference in means between 2 groups is either greater than $> \mu$ or $< \mu$. I.e., either $\mu_1 - \mu_2 > \mu$ or $\mu_1 - \mu_2 < \mu$.

Example use case: Betty needs to compare two stock solutions and check that their pH differs by no more than 0.2.

The 2-sample, 1-tailed *t*-test that can be used in several ways:

1. Test whether the mean in one group is greater or less than the mean in another group.
2. Test whether the the mean in one group is greater or less than the mean in another group, and that the difference is at least some specified value.

For example, if you know that the mean of group 1 must be less than the mean in group 2, and will not even consider that the mean of group 2 is less than that of group 1, then you could run a 2-sample 1-tailed test.

```
# simulate data
set.seed(789)
x1 <- rnorm(20, 10, 1)
x2 <- rnorm(20, 12, 1)

# is mean(x1) > mean(x2)?
t.test(x1, x2, alternative="greater")
```

Welch Two Sample t-test

```
data:  x1 and x2
t = -8.8728, df = 37.99, p-value = 1
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -2.402592      Inf
sample estimates:
mean of x mean of y
 9.689982 11.708940
```

```
# is mean(x1) < mean(x2)?
t.test(x1, x2, alternative="less")
```

Welch Two Sample t-test

```
data:  x1 and x2
t = -8.8728, df = 37.99, p-value = 4.259e-11
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
 -Inf -1.635323
sample estimates:
mean of x mean of y
 9.689982 11.708940
```

```
# is mean(x1) - mean(x2) at least 1?
t.test(x1, x2, alternative="greater", mu=1)
```

Welch Two Sample t-test

```
data: x1 and x2
t = -13.267, df = 37.99, p-value = 1
alternative hypothesis: true difference in means is greater than 1
95 percent confidence interval:
 -2.402592      Inf
sample estimates:
mean of x mean of y
 9.689982 11.708940
```

```
# is mean(x1) - mean(x2) at most 1?
t.test(x1, x2, alternative="less", mu=1)
```

Welch Two Sample t-test

```
data: x1 and x2
t = -13.267, df = 37.99, p-value = 3.867e-16
alternative hypothesis: true difference in means is less than 1
95 percent confidence interval:
 -Inf -1.635323
sample estimates:
mean of x mean of y
 9.689982 11.708940
```

an uncommon variant used to test whether two groups have a difference in means that is less than or greater than some specified value. For example, if you produce two batches of bacteria by inoculation, incubation, and then serial dilution, you may want to confirm that the difference in cell count between them is less than some acceptable value. You could use a 2-sample, 1-tailed test with the “alternative” hypothesis being that the difference is “greater” than your benchmark.

```
set.seed(789)
x1 <- rnorm(30, 5, 2)
x2 <- rnorm(30, 5.5, 2)

# is the true difference in means <1?
t.test(x1, x2, alternative="greater")
```

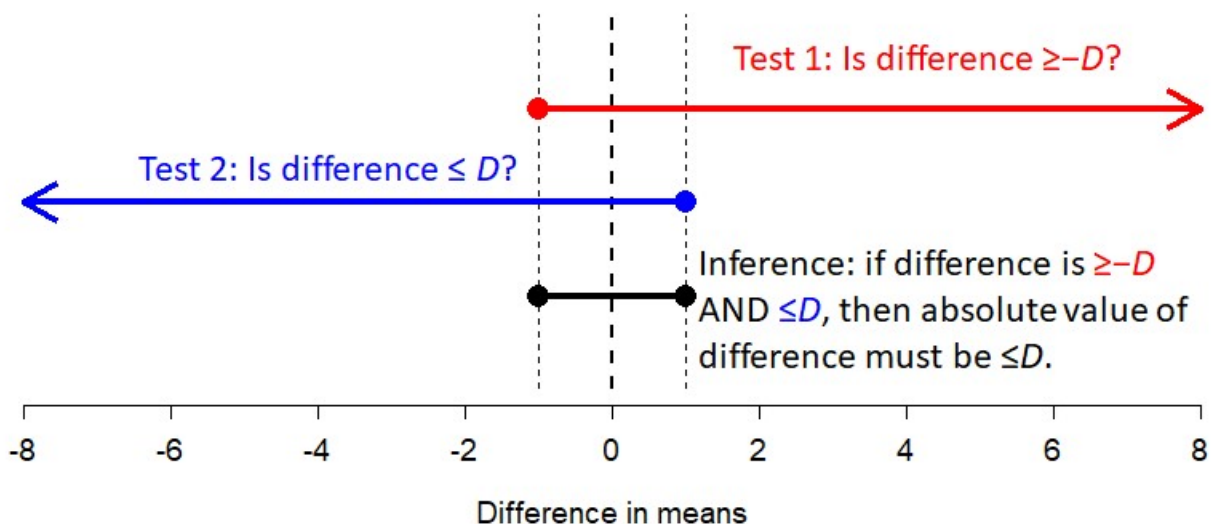
Welch Two Sample t-test

```
data:  x1 and x2
t = -2.6676, df = 50.08, p-value = 0.9949
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -2.048126      Inf
sample estimates:
mean of x mean of y
 4.438104  5.696001
```

15.2.3 Equivalence testing

It is not possible to demonstrate statistically that a difference in means is truly 0, or that the underlying mean of a set of data truly is equal to a target value μ (the converse of a one-sample, two-tailed t -test). With a 1-sample, 2-tailed test the best you can do is “fail to reject” the null hypothesis that the mean is 0. However, we can show that the difference is arbitrarily close to 0. This approach is called **equivalence testing**, and one method is the **two one-sided tests (TOST)** method. The method works like this:

Figure 15.1: Illustration of the two one-sided test (TOST) strategy for establishing that a sample, or set of differences, has an underlying mean arbitrarily close to some value μ ($\mu = 0$ in this example.)



Here is a worked TOST equivalence test example in R:

```
# simulate data
set.seed(42)
a <- rnorm(100, 5, 1)
```

```

b <- rnorm(100, 5, 1)

# let target tolerance = 1
# note that total tolerance range is 1, so
# amount for both one-sided tests is 1/2==0.5
D <- 0.5

# test 1: is difference >-D?
t.test(a, b, alternative = "greater", mu=-D)

```

Welch Two Sample t-test

```

data: a and b
t = 4.4956, df = 194.18, p-value = 5.955e-06
alternative hypothesis: true difference in means is greater than -0.5
95 percent confidence interval:
 -0.1079329      Inf
sample estimates:
mean of x mean of y
 5.032515  4.912516

```

```

# test 2: is difference <D?
t.test(a, b, alternative = "less", mu=D)

```

Welch Two Sample t-test

```

data: a and b
t = -2.7554, df = 194.18, p-value = 0.00321
alternative hypothesis: true difference in means is less than 0.5
95 percent confidence interval:
 -Inf 0.34793
sample estimates:
mean of x mean of y
 5.032515  4.912516

```

Interpretation: Test 1 showed that the difference in means is > -0.5 . Test 2 showed that the difference in means is < 0.5 . So, the difference in means is in the interval $(-0.5, 0.5)$, which means the difference in means is at most 0.5 in either direction.

15.2.4 Wilcoxon rank-sum test

Question: Are values in one group consistently greater than or less than those in another group? OR: Is the median rank in one group different from that in another group?

Null hypothesis: For randomly selected values x and y from two groups, $p(x > y) = p(y > x)$.

Alternative hypothesis: For randomly selected values X and Y from two groups, $p(x > y) \neq p(y > x)$.

Example use case: Homer is interested in whether plots where invasive plants have been removed have more squirrels than plots where invasive plants are not removed, but does not want to model the actual number of squirrels.

The **Mann-Whitney U test**, also known as the **Wilcoxon test**, is a nonparametric alternative to the two-sample t -test. **Nonparametric** means that the test does not make restrictive assumptions about the distribution of the data or a test statistic. Nonparametric tests are often based on the **ranks** of the data: the smallest value is rank 1, the next smallest is rank 2, and so on. Because of this, nonparametric tests allow you make statements about how consistent a pattern is, rather than about the actual values.

This test has two common use cases:

- The researcher is only interested in whether values in one group are consistently larger than those in another group.
- The researcher needs to perform a t -test, but the data do not meet the assumptions of the t -test.

The syntax for the Wilcoxon test is fairly similar to that of a t -test.

Example Wilcoxon test in R:

```
# simulate some data:
set.seed(123)
a <- runif(20, 1, 6)
b <- runif(20, 4, 9)

# run the test
wilcox.test(a,b)
```

Wilcoxon rank sum exact test

data: a and b

W = 35, p-value = 1.126e-06

alternative hypothesis: true location shift is not equal to 0

Here is the same kind of test, but using the formula interface. You can use the formula interface for t -tests as well.

```
# pull some data from example dataset iris
# i.e., make a new data frame with only 2 groups
use.species <- c("versicolor", "virginica")
dx <- iris[which(iris$Species %in% use.species),]
dx$Species <- factor(dx$Species, levels=use.species)

# perform the test.
wilcox.test(Petal.Length~Species, data=dx)
```

Wilcoxon rank sum test with continuity correction

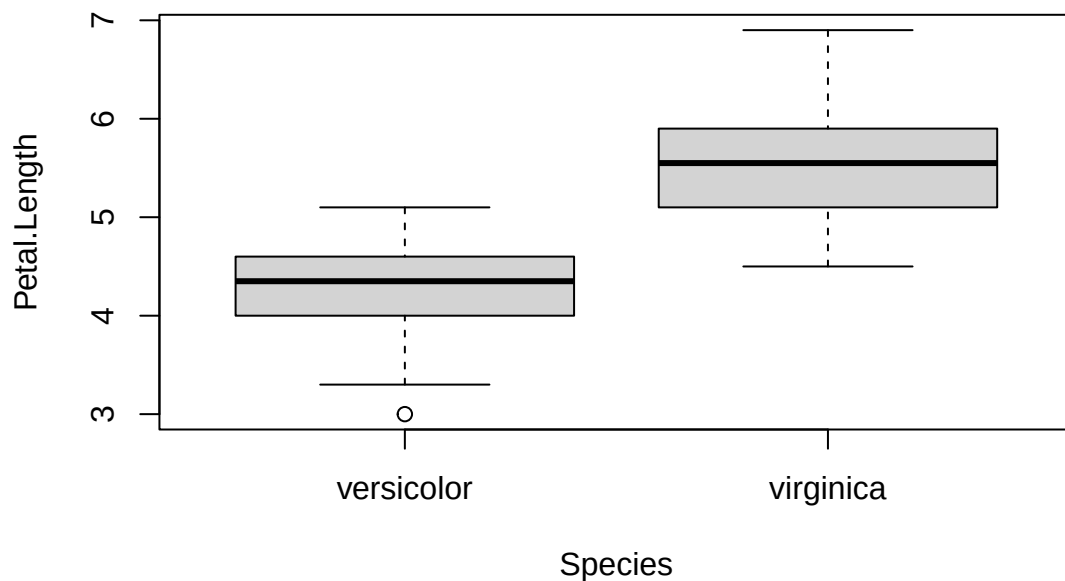
data: Petal.Length by Species

W = 44.5, p-value < 2.2e-16

alternative hypothesis: true location shift is not equal to 0

Interpretation: The researcher can say that values in one group are consistently greater than values in the other group. Which way that goes should be obvious from looking at the sample medians or a boxplot.

```
# follow up the test with a boxplot
# note that the formula interface is used here as well!
boxplot(Petal.Length~Species, data=dx)
```



When sample sizes or effect sizes are large, the Wilcoxon test and the t -test will often give the same answers in terms of significance vs. nonsignificance.

Because the t -test tests for a difference in means, and the Wilcoxon is its rank-based alternative, some people think that the Wilcoxon test is a test for a difference in medians...but this is **not correct**. It's more accurate to say that the Wilcoxon test tests for a median difference.

15.3 Tests for paired data

15.3.1 Paired t -test

Question: Is the mean of the differences of paired observations = 0, or different from 0?

Null hypothesis: Mean of pairwise differences = 0.

Alternative hypothesis: Mean of pairwise differences $\neq 0$.

Example use case: Pebbles needs to test for a mean change in the body mass of individual mice before and after a feeding trial. Each mouse is weighed both before and after the trial.

Many biological experiments involved **paired samples**, where a single unit or entity was measured twice. For example, a single mouse might be weighed before and after a feeding trial. In a **paired t -test**, we test whether or not the mean of differences is 0. This is subtly different than the test for differences in means evaluated in a two-sample test. Consequently, a paired t -test is tantamount to a one-sample test on the pairwise differences. Compare the null hypotheses tested by the two-sample t -test and paired t -test below:

Two-sample t -test: $\frac{\sum_{i=1}^{n_1} x_{1,i}}{n_1} = \frac{\sum_{i=1}^{n_2} x_{2,i}}{n_2}$

Paired t -test: $\frac{\sum_{i=1}^n x_{1,i} - x_{2,i}}{n} = 0$

Let's simulate some data, conduct a paired t -test, and convince ourselves that the paired t -test is equivalent to a one-sample test on pairwise differences.

```
# Example paired t-test in R
# simulate data
set.seed(42)
a <- rnorm(100, 5, 2)
# add pair-wise differences
b <- a + rnorm(100, 2, 0.2)
```

Interpretation: The researcher should conclude that the mean difference for each subject was significantly different from 0, at -1.98 with a 95% CI of [-2.02, -1.95].

Note that the paired t -test gives essentially the same result as conducting a one-sample test on the pair-wise differences.

```
# compare to one sample t-test on differences:

dif <- b - a
t.test(dif)
```

One Sample t-test

```
data:  dif
t = 109.63, df = 99, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 1.946622 2.018385
sample estimates:
mean of x
 1.982503
```

15.4 t-test summary

As we have seen, the *t*-test in all of its variations is a very useful and flexible family of tests that can be applied to many biological problems. One of the reasons that *t*-tests are so widely applicable—and why there are so many *t*-tests—is that the test statistic *t* is so elegant in its construction: it represents the quantity of interest directly in its numerator, and then the denominator scales that quantity according to the level of uncertainty in the data. It is thus a kind of **signal-to-noise ratio** that predated information theory by almost 50 years.

As a review, the table below summarizes the different *t*-tests and their use cases:

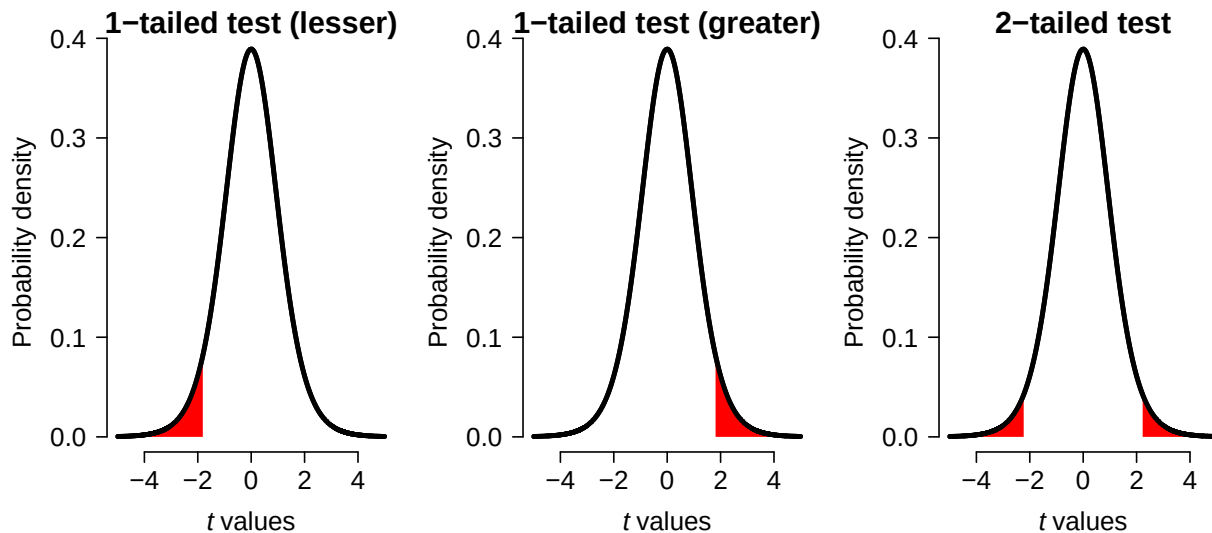
Table 15.2: Common types of *t*-tests and the hypotheses they evaluate.

Type	Use	Section
One-sample <i>t</i> -test, one-tailed	Tests whether the population mean of a single set of values is less than or greater than a specified value μ_0 . Appropriate when differences in only one direction are of interest.	Section 15.1.1
One-sample <i>t</i> -test, two-tailed	Tests whether the population mean is equal to a specified value μ_0 (null hypothesis) or differs from it (alternative hypothesis).	Section 15.1.2

Type	Use	Section
Two-sample t -test, two-tailed	Tests whether the population means of two groups (μ_1 and μ_2) differ. This can also be thought of as testing whether the difference between the means is equal to 0. Welch's t -test, which does not assume equal variances, should generally be used and is the default in R.	Section 15.2.1
Two-sample t -test, one-tailed	Tests whether the population mean of one group (μ_1) is less than or greater than the mean of another group (μ_2). Appropriate when differences in only one direction are of interest.	Section 15.2.2
Paired t -test	Tests whether the mean difference between paired observations differs from a specified value D_0 , usually 0. The test may be one- or two-tailed depending on the hypothesis.	Section 15.3.1

The number of “samples” in a t -test refers to how many sets of observations are being compared. There can be either 1 sample (all observations compared to some value) or 2 samples (2 sets of observations compared to each other).

The number of “tails” refers to how sides (left (negative) and/or right (positive)) are of interest in the test statistic distribution. In a two-tailed test, or two-sided test, an extreme test statistic could fall in either the right or the left tail of the distribution. The figure below shows the range of t values which would be declared significant for each kind of test. Notice that for a one-tailed test, the significance region extends closer to the center, because its area must be equal to the same α as a two-tailed test.



The t -test is usually one of the first methods taught because its use cases are simple, and its test statistic is pretty easy to understand. Consider the original t statistic introduced to the English statistics literature by Gosset and popularized as “Student’s t ” by Fisher:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_{x_1}^2 + s_{x_2}^2}{2}} \sqrt{\frac{2}{n}}}$$

The numerator is the quantity of interest: the difference in sample means. It should make sense that the larger the difference in sample means, the greater the probability that the difference will be statistically significant. However, considering only the sample means $\bar{x}_1$ and $\bar{x}_2$ is not enough. We also need to account for variation within each sample, and for the total sample size. Variance should *decrease* t , while sample size should increase it. Both terms are put in the denominator so that the difference in means is relative to them. Notice that in the denominator, the variances $s_{x_1}^2$ and $s_{x_2}^2$ are in a numerator. Thus, increasing them will decrease t . On the other hand, the sample size n is in a numerator in the denominator (which is tantamount to being in the overall numerator). Thus, increasing sample size will increase t .

But as elegant as Student’s t is, it has some drawbacks. It can produce biased estimates and p -values when the two groups have unequal sample sizes, or unequal variances. For these reasons, a more modern version of the t statistic was developed by Welch in the 1940s. This is sometimes called the “Welch’s t -test”, or less precisely the “Welch-Satterthwaite t -test”. Satterthwaite’s contribution was not to the t statistic, but rather an equation to better calculate the degrees of freedom to use for the test².

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1}{\sqrt{n_1}} + \frac{s_2}{\sqrt{n_2}}}}$$

This newer version has the same numerator, but instead of pooling the variances and sample sizes, keeps

²This is why t -tests in R often report non-integer degrees of freedom.

them separate. This allows for more flexibility when dealing with unequal sample sizes or (mildly) heteroscedastic data.

15.5 Tests for 3 or more groups

15.5.1 Analysis of variance

In this section, we will explore the “test comparing group means” sense of analysis of variance (ANOVA) (Section 14.5). The linear model version of ANOVA assesses significance using an F -test. Much like t , F is a test statistic that summarizes a quantity of interest in a test that can be compared to the typical strength of a pattern produced by random noise (see Section 15.2.1). Whereas t is usually a difference in means scaled by sample size and variance, F compares variances. The basic idea of an F statistic is:

$$F = \frac{\text{explained variance}}{\text{unexplained variance}}$$

In the case of a one-way ANOVA, this translates to:

$$F = \frac{\text{between-group variability}}{\text{within-group variability}}$$

The numerator is essentially the variance in group means:

$$\sum_{i=1}^K n_i \frac{(\bar{Y}_i - \bar{Y})^2}{(K - 1)}$$

where $\bar{Y}_i$ is the sample mean in group i , n_i is the number of observations in group i , $\bar{Y}$ is the overall mean of the data, and K is the number of groups.

The denominator is close to the the sum of within-group variances:

$$\frac{\sum_{i=1}^K \sum_{j=1}^{n_i} (Y_{ij} - \bar{Y}_i)^2}{N - K}$$

where Y_{ij} is observation j in group i (out of K groups), and N is the overall sample size.

The F statistic is a ratio of *variance explained* to *variance not explained*. F values follow the F distribution, which is defined by degrees of freedom $d_1 = K - 1$ and $d_2 = N - K$ under the null hypothesis. Practically, F will be large when there is more explained variance than unexplained variance, which is unlikely if the group means are not different.

Question: Do the means differ between groups (with number of groups ≥ 2).

Null hypothesis: All group means are equal.

Alternative hypothesis: At least one group mean is different from the others.

Example use case: Marge needs to test whether cucumber mass is greater in plants grown in potting mix, sandy soil, or clayey soil.

The nature of the F statistic belies the other meaning of ANOVA: the partitioning of variation into its different sources. In other words, what proportion of variation in the Y variable does each factor explain? How much variation is due to random chance (i.e., residual variation)? ANOVA in this sense can be applied to many models, including all linear models.

The first step in a typical ANOVA is the **omnibus test**, which assesses whether *any* of the group means differ from each other. Then, a **post-hoc test** is performed to check *which* groups differ from each other.

ANOVAs are typically described as “X-way ANOVA”, where X is the number of factors. For example, a test of mean stomach size in herbivorous, carnivorous, and omnivorous mammals would be a “one-way ANOVA” because it has one grouping variable, “diet”. The three groups in the factor diet are the **levels** of the factor.

15.5.1.1 One-way ANOVA

When there is one explanatory **factor**, or grouping variable, the test is called a **one-way ANOVA** regardless of how many **levels** or groups that factor contains. Here is an example using the built-in example dataset `iris`. There are two common ways to fit an ANOVA in R.

```
# method 1: lm()
mod1 <- lm(Petal.Length~Species, data=iris)

# method 2: aov()
mod2 <- aov(Petal.Length~Species, data=iris)
```

Once the model is fit, we request the **anova table** so we can see the contributions of each source of variation.

```
# method 1: using lm
anova(mod1)
```

Analysis of Variance Table

Response: Petal.Length

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Species	2	437.10	218.551	1180.2	< 2.2e-16 ***
Residuals	147	27.22	0.185		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```
# method 2: using aov
summary(mod2)
```

```

              Df Sum Sq Mean Sq F value Pr(>F)
Species        2  437.1   218.55    1180 <2e-16 ***
Residuals     147   27.2    0.19
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

No matter which method we use, the ANOVA table shows us that the factor *Species* uses 2 degrees of freedom (variously abbreviated as DF, Df, or d.f.) to account for 437.10 of the squared errors; the residuals use 147 d.f. to account for 27.22 of the squared errors. The ratio of the mean squared error per d.f. associated with each source, $218.551 / 0.185 = 1180.2$, is the test statistic F . This is used to calculate the P -value (much like t , F follows a particular distribution defined by DF, which allows $p(F)$ to be calculated). We can also see that about 94.1% of variation ($437.10 / (437.10 + 27.22)$) is associated with species. This is the model's coefficient of determination, or R^2 . The R^2 tells you the proportion of variation in Y explained by the model.

Tip

Rounding errors

R is somewhat idiosyncratic in choosing how many digits it prints, and this can lead to confusion if values are rounded to too few decimal places. In the ANOVA example above, the MSE for the two sources of variation should divide to the F statistic. However, $218.55 / 0.19 \neq 1180$. If you look at the raw output of `mod1`, however, you that the sums of squares for *Species* and *Residuals* are 437.1028 and 27.2226, respectively. With more decimal places, you can then recalculate $(437.1028 / 2) / (27.2226 / 147) = 1180.161$, the correct F ratio.

Remember that the all that the omnibus test p -value tells you is that the means of at least one pair of species differ. To find out which pair or pairs, we need to use a post-hoc test. My favorite is the Tukey's honest significant difference (HSD) test because it automatically adjusts for multiple comparisons.

```
# method 1:
TukeyHSD(aov(mod1))
```

```

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

Fit: aov(formula = mod1)

$Species
              diff          lwr          upr p adj

```

```
versicolor-setosa      2.798 2.59422 3.00178      0
virginica-setosa       4.090 3.88622 4.29378      0
virginica-versicolor  1.292 1.08822 1.49578      0
```

```
# method 2:
TukeyHSD(mod2)
```

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

```
Fit: aov(formula = Petal.Length ~ Species, data = iris)
```

```
$Species
              diff      lwr      upr p adj
versicolor-setosa  2.798 2.59422 3.00178      0
virginica-setosa   4.090 3.88622 4.29378      0
virginica-versicolor 1.292 1.08822 1.49578      0
```

The Tukey output shows us that every group differs from every other, with $p < 0.001$ (here p is rounded to 0, although it can never be 0). The differences in group means are presented with their **95% confidence intervals (CI)**. For example, the difference between the group means of `versicolor` and `setosa` is 2.798 cm, with 95% CI = [2.594, 3.002]. The fact that the difference is > 0 means that the mean in `versicolor` is greater than the mean in `setosa`. If the difference was negative, then that would mean that the mean in `setosa` was greater.

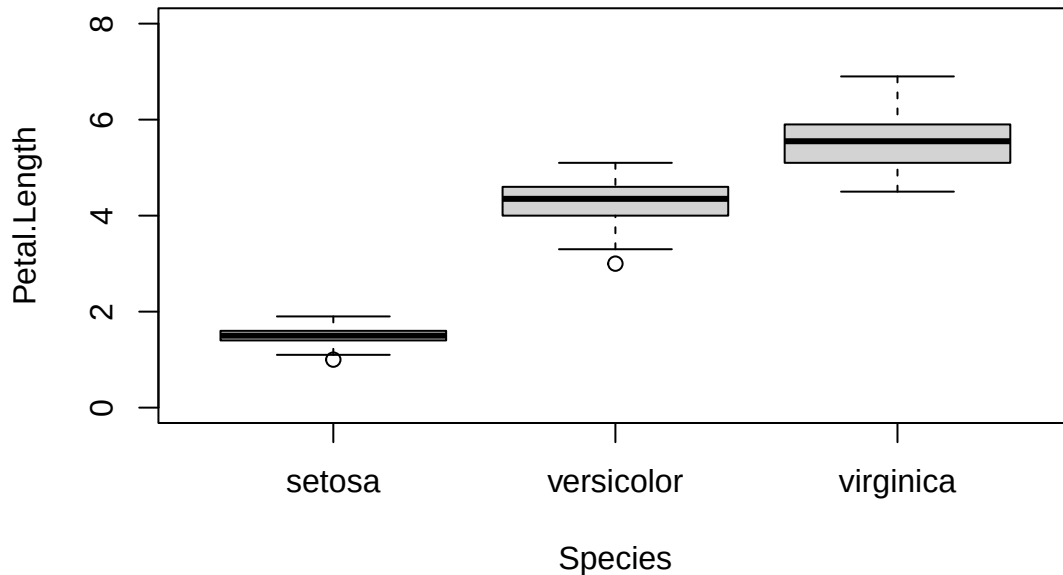
Notice that none of these CI include 0: this suggests that a true difference of 0 between any two groups is very unlikely. Even without examining p -values, CI provide a useful way to evaluate both the direction and magnitude of differences.

One of the nice things about the Tukey HSD test is that the p -values are automatically adjusted for **multiple comparisons**, a phenomenon where the probability of a false positive goes up as a [researcher performs more tests](#), using a customized version of the probability distribution for the test statistic. This controls the **family wise error rate**, or total false positive probability for this set of three related tests.

As with a t -test, a boxplot is usually the best way to show the results of an ANOVA.

```
boxplot(Petal.Length~Species, data=iris, ylim=c(0, 8))
```

Figure 15.2: Boxplot showing that petal length is greater in *Iris versicolor* than in *Iris setosa*, and greater still in *Iris virginica*.



15.5.1.2 Two- or more way ANOVA

When there are 2 or more explanatory factors, the test is called a **two-way ANOVA**, or **three-way ANOVA**, and so on. We can add additional factors to an ANOVA with the + sign.

```
# make a copy of iris with an extra grouping variable
iris2 <- iris
iris2$Flower <- c("purple", "pink")

# fit the two-way ANOVA without interaction
mod3 <- aov(Petal.Length~Species+Flower, data=iris2)
summary(mod3)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
Species	2	437.1	218.55	1174.229	<2e-16	***
Flower	1	0.0	0.05	0.261	0.61	
Residuals	146	27.2	0.19			

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Adding the colors that way just repeated them until the column was full: purple, pink, purple, pink, and so on until the new vector had length 150 (the number of rows in `iris`). This essentially distributes the colors randomly with respect to the species and thus the petal lengths, so it should be no surprise that flower color had no effect on petal length ($p = 0.61$). A Tukey test confirms the effects of species, and the non-effect of `Flower`. Technically, you don't need to do or report a Tukey test if the main effect is nonsignificant.

```
TukeyHSD(mod3)
```

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

```
Fit: aov(formula = Petal.Length ~ Species + Flower, data = iris2)
```

```
$Species
```

	diff	lwr	upr	p adj
versicolor-setosa	2.798	2.593692	3.002308	0
virginica-setosa	4.090	3.885692	4.294308	0
virginica-versicolor	1.292	1.087692	1.496308	0

```
$Flower
```

	diff	lwr	upr	p adj
purple-pink	0.036	-0.1032347	0.1752347	0.6101255

15.5.1.3 ANOVA with interaction

Another important question is whether there is an **interaction** between two variables. An interaction is a situation where one variable changes the effect of another. To test for an interaction between two variables, use the `*` symbol instead of the `+`. The result `y~x1*x2` is shorthand for `y~x1+x2+x1:x2`. The individual effects of the variables `x1` and `x2` are called their **main effects**. The effect of both together `x1:x2` is the **interaction effect**. If a significant interaction effect is detected, then the main effects should not be interpreted by themselves.

```
# fit the two-way ANOVA WITH interaction
mod3 <- aov(Petal.Length~Species*Flower, data=iris2)
summary(mod3)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Species	2	437.1	218.55	1161.375	<2e-16 ***

Flower	1	0.0	0.05	0.258	0.612
Species:Flower	2	0.1	0.04	0.201	0.818
Residuals	144	27.1	0.19		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

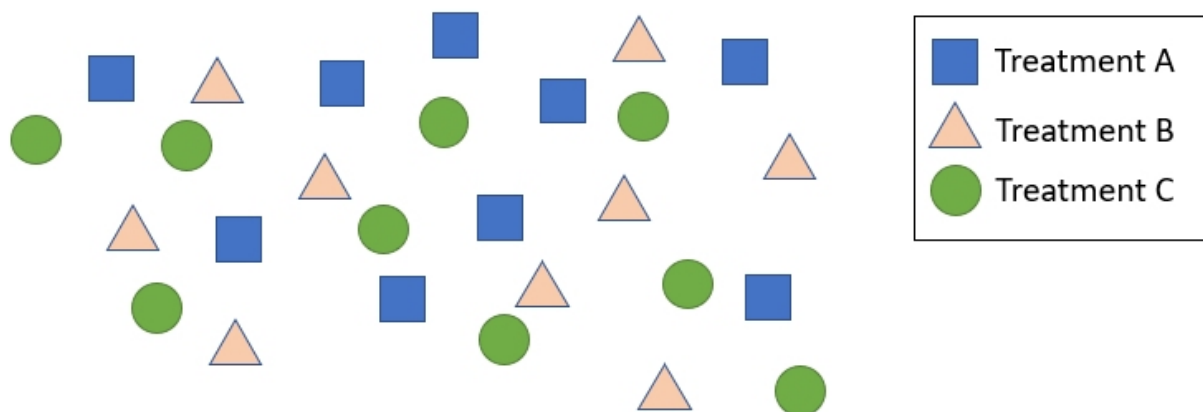
The interaction is nonsignificant, so we should instead report the main effects in the model without the interaction (`mod2`). If there is an interaction present, then it becomes impossible to discuss the effects of one variable without discussing the effects of the other. We will explore interactions more in the section on ANCOVA below.

15.5.1.4 Blocking designs

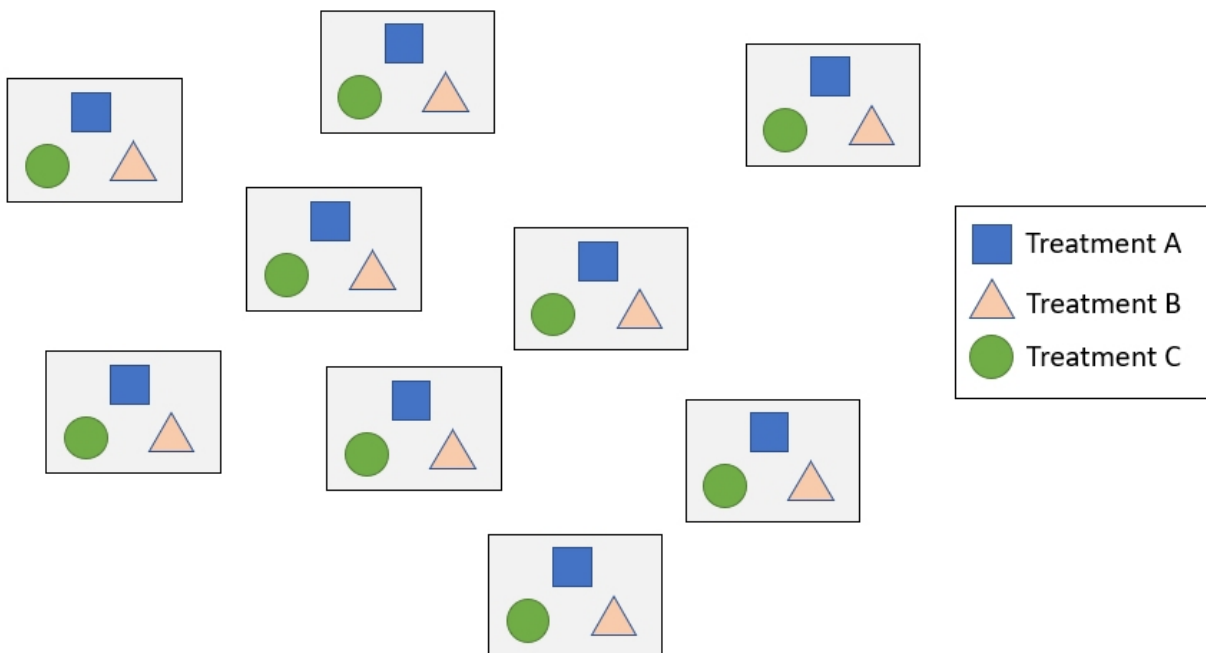
Blocking is a strategy in experimental design that attempts to control for variation by grouping experimental units (observations) into groups called **blocks**. Blocks are defined by observations that are similar to each other in some attribute—like location, time, sex, species, etc.—that contributes to overall variation and that the researcher needs to control for.

Consider the example below.

Figure 15.3: Illustration of a field experiment with 3 researcher-applied treatments, A, B, and C, spread out over a landscape.



This experiment appears to have randomly distributed its treatments across space, which is good. However, there will still be variation among observations that is caused by different environments or habitats in different places. Rather than model those location-specific differences explicitly, the research could instead distributed **blocks** of observations randomly across space, with each block containing 1 observation of each of the 3 treatments.



{fig-

alt="fig-alt="Schematic of a blocked experimental design with three treatments. Experimental units assigned to Treatment A, Treatment B, and Treatment C are represented by blue squares, orange triangles, and green circles, respectively. The units are arranged into nine blocks, shown as rectangles, with each block containing one unit from each treatment."}

Now, variation attributable to spatial factors can be associated with the blocks and no longer confounds the effects of treatment. Observations within a block should be close enough that they experience the same environmental conditions.

In the example code below, we are going to add an imaginary factor to the `iris` dataset, `color`. The example analysis shows how a researcher would test for a difference in petal length between flowers of different colors, while accounting for the fact that different species have different petal lengths. In this analysis, the researcher is interested in differences between colors, and considers species a potential confounding variable.

This example works because we force the color effect with an artificial variable, and because the setup says explicitly that the researcher considers species a potential confound. From an experimental design standpoint, blocking is not synonymous with “a second categorical factor in an ANOVA”, although that is how it is treated mathematically.

```
# spare copy
dx <- iris

# make the largest flowers within each species purple,
# and the smallest flowers blue.
# scale within species
z <- tapply(dx$Petal.Length, dx$Species, scale)
```

```

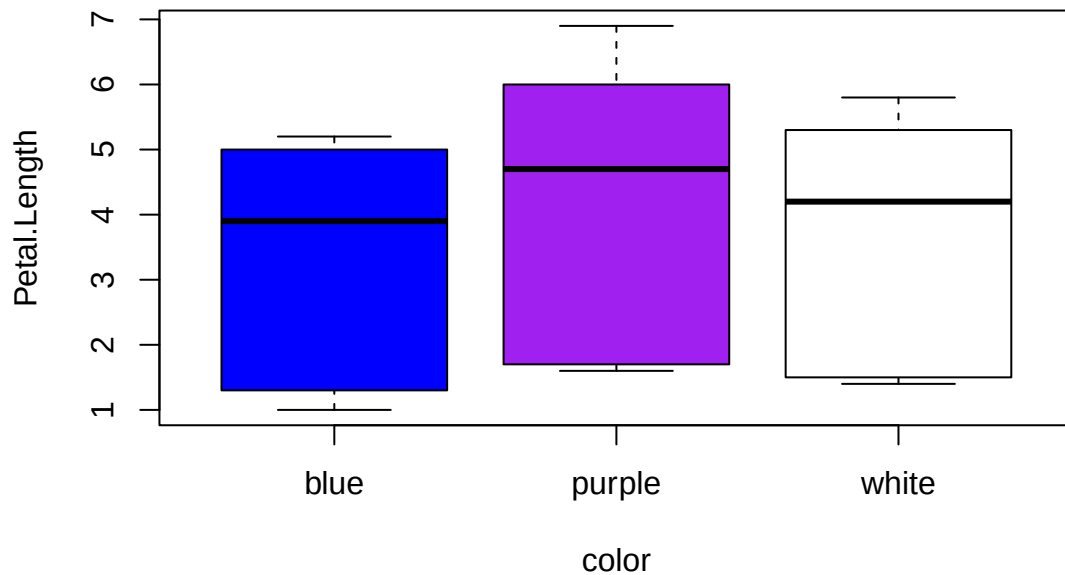
dx$z <- do.call(c, z)

# any flower >= 0.6 SD away from mean
# is a non-white color
dx$color <- "white"
dx$color[which(dx$z <= -0.6)] <- "blue"
dx$color[which(dx$z >= 0.6)] <- "purple"

# vector of colors
cols <- c("blue", "purple", "white")

boxplot(Petal.Length~color, data=dx, col=cols)

```



The boxplot shows that the sample medians for blue, white, and purple are the smallest, intermediate, and greatest, respectively. But is that a significant difference?

```

# test for effect of color on petal length
summary(aov(Petal.Length~color, data=dx))

```

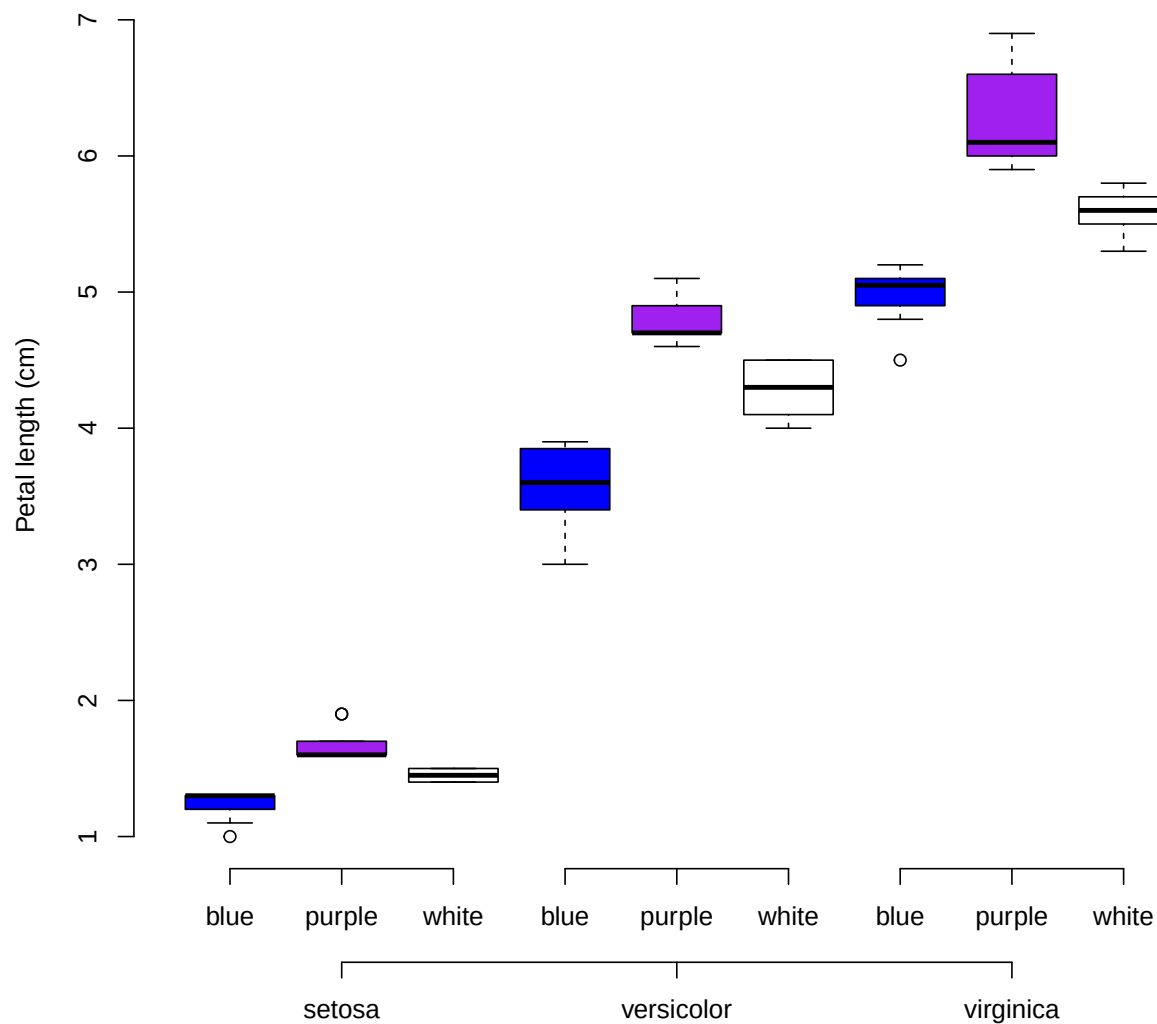
	Df	Sum Sq	Mean Sq	F value	Pr(>F)
color	2	13.6	6.805	2.219	0.112
Residuals	147	450.7	3.066		

No, it's not. But at this point the researcher should suspect that a lot of variation in size is really due to species. If she controls for species by treating it as a blocking factor, then she might be better able to detect an effect of color.

Below is the same petal length data broken up by color within species. Now, we can see that within each species, blue flowers are smaller and purple flowers are larger. The first ANOVA couldn't detect that because it didn't consider species.

Also, note the use of `axis()` to create a more elegant set of labels for the x-axis.

```
par(mar=c(6.1, 4.1, 1.1, 1.1), bty="n")
boxplot(Petal.Length~color+Species, data=dx, col=cols,
        xaxt="n", xlab="", ylab="Petal length (cm)")
axis(side=1, at=1:3, labels=cols)
axis(side=1, at=4:6, labels=cols)
axis(side=1, at=7:9, labels=cols)
axis(side=1, at=c(2, 5, 8), labels=levels(dx$Species), line=3)
```



Here is the ANOVA rerun with species as a blocking factor:

```
anovatable <- summary(aov(Petal.Length~color+Species, data=dx))
anovatable
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
color	2	13.6	6.81	130.6	<2e-16 ***
Species	2	443.2	221.58	4251.0	<2e-16 ***
Residuals	145	7.6	0.05		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

This shows that there is a significant effect of color, after species has been accounted for. We can also see why species covered the effect of color: there are considerably more sums of squares associated with species than with color.

```
anovatable[[1]][,2]/sum(anovatable[[1]][,2])
```

```
[1] 0.02931143 0.95441113 0.01627743
```

The total variance is approximately $13.61 + 443.16 + 7.56 = 464.33$. Most of that variance ($100(443.16/464.33) \approx 95.4\%$) is attributed to differences between species. Only about 3% ($13.61/464.33$) is attributed to color. It's a small effect, but a significant one, and only detectable after accounting for species.

15.5.1.5 Repeated measures ANOVA

Coming soon!

15.5.2 Kruskal-Wallis test

Question: Are values in one or more groups consistently greater or less than values in another group? (number of groups ≥ 3); essentially extends Wilcoxon test to more groups)

Null hypothesis: For randomly selected values x and y from any 2 groups, $p(x > y) = p(y > x)$.

Alternative hypothesis: For randomly selected values x and y from any 2 groups, $p(x > y) \neq p(y > x)$

Example use case: Bart is studying whether tomato yields tend to be greater in plants treated with no pesticide, pesticide A, or pesticide B. He is not interested in predicting the actual tomato yield.

Because ANOVA is a linear model, the data used in ANOVA must meet the **assumptions of the linear model**. If the data do not meet the assumptions, or cannot be transformed to do so, then a nonparametric alternative to ANOVA is needed. The most common alternative is the **Kruskal-Wallis test**. Some texts refer to this as a “nonparametric ANOVA”, but this is not correct because at no point is variance actually partitioned. Like the Wilcoxon test, the Kruskal-Wallis test is based on ranks of the data. A significant Kruskal-Wallis test indicates that values in at least one group are consistently greater than values in at least one other group. As with ANOVA, a post-hoc test is needed to determine which groups differ significantly from each other.

Example Kruskal-Wallis test in R

```
kruskal.test(Petal.Length~Species, data=iris)
```

```
Kruskal-Wallis rank sum test
```

```
data: Petal.Length by Species
```

```
Kruskal-Wallis chi-squared = 130.41, df = 2, p-value < 2.2e-16
```

The significant result means that petal length is consistently greater for at least species compared to the other. This is analogous to the omnibus ANOVA test. If we want to know which groups differ, we need a post-hoc test. The most common option is the Dunn's test.

```
library(dunn.test)
dunn.test(iris$Petal.Length, iris$Species, method="bonferroni")
```

Kruskal-Wallis rank sum test

data: x and group

Kruskal-Wallis chi-squared = 130.411, df = 2, p-value = 0

Comparison of x by group
(Bonferroni)

Col Mean-		
Row Mean	setosa	versicol
-----+-----		
versicol	-5.862996	
	0.0000*	
virginic	-11.41838	-5.555388
	0.0000*	0.0000*

alpha = 0.05

Reject Ho if p <= alpha/2

Interpretation: A posthoc Dunn's test showed that all three species differed from each other in terms of the rank order of petal lengths.

15.6 Tests with ordered factors

Some factors can represent group membership, but still have an inherent ordering to them. For example, soft drinks come in sizes "small", "medium", and "large". These are categories, but categories in order: small < medium < large. In contrast, the three species in the `iris` dataset have no ordering.

You may have data with a factor for age, like "juvenile", "subadult", "adult". These categories correspond to ages, but cannot be treated as literal numbers of years or months. It would also be a mistake to treat the comparison of "juvenile vs. subadult" the same as the comparison of "juvenile vs. adult".

If you analyze an ordered factor, R will estimate polynomial terms of order up to $k - 1$, where k is the number of factor levels. The polynomial function is a way to fit different patterns across the ordered levels. For example, a linear pattern captures a monotonic increase or decrease across the levels. A quadratic

pattern can describe a situation where the middle of three levels has greater (or smaller) values than the first and third levels. Let's create a toy dataset to illustrate working with an ordered factor.

```
ages <- c("juvenile", "subadult", "adult")
dx <- data.frame(age=rep(ages, each=20))
# tell R that this is an ordered factor
dx$age <- factor(dx$age, levels=ages, ordered=TRUE)

# notice the levels statement:
dx$age
```

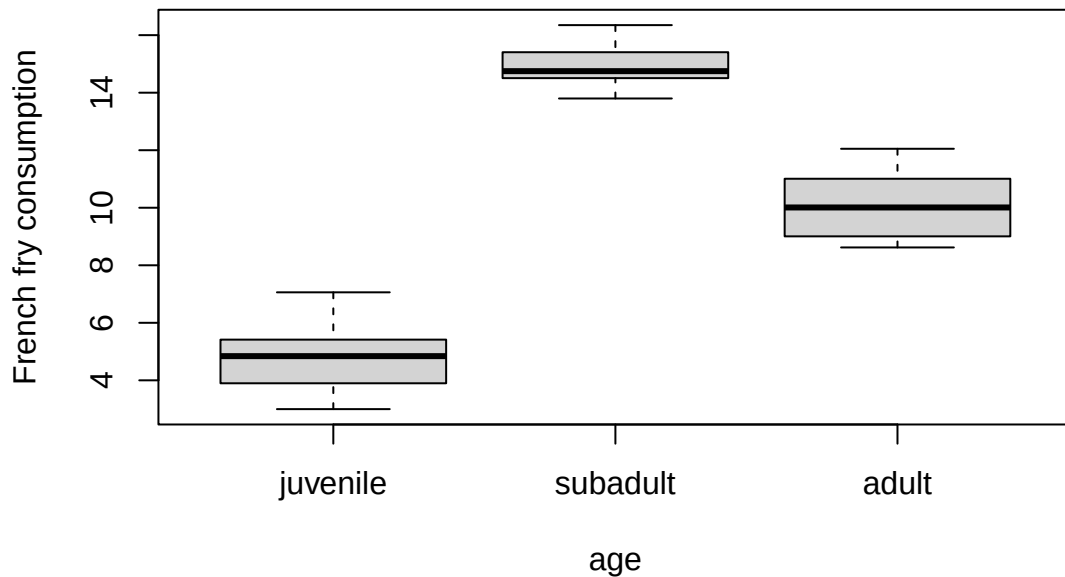
```
[1] juvenile juvenile juvenile juvenile juvenile juvenile juvenile juvenile
[9] juvenile juvenile juvenile juvenile juvenile juvenile juvenile juvenile
[17] juvenile juvenile juvenile juvenile subadult subadult subadult subadult
[25] subadult subadult subadult subadult subadult subadult subadult subadult
[33] subadult subadult subadult subadult subadult subadult subadult subadult
[41] adult adult adult adult adult adult adult adult
[49] adult adult adult adult adult adult adult adult
[57] adult adult adult adult
Levels: juvenile < subadult < adult
```

Here we use our new factor to plot and analyze french fry consumption across age categories:

```
# add y values
dx$ff <- NA
dx$ff[which(dx$age=="juvenile")] <- rnorm(20, 5, 1)
dx$ff[which(dx$age=="subadult")] <- rnorm(20, 15, 1)
dx$ff[which(dx$age=="adult")] <- rnorm(20, 10, 1)

# inspect the data
boxplot(ff~age, data=dx, ylab="French fry consumption")
```


Figure 15.4: Boxplot showing that french fry consumption is greatest in subadults, less in adults, and lowest in juveniles.



Now fit the model using the ordered factor age:

```
mod11 <- aov(ff~age, data=dx)
summary(mod11)
```

```

              Df Sum Sq Mean Sq F value Pr(>F)
age              2 1008.8    504.4    526.2 <2e-16 ***
Residuals       57   54.6     1.0
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The omnibus test shows us that age is significant. However, because we have an ordered factor, we don't want something like a Tukey test. Group means are not of interest; rather, the pattern in group means across groups is. So, we what we want is to see which patterns across age are significant.

```
summary.lm(mod11)
```

Call:

```
aov(formula = ff ~ age, data = dx)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-1.88500	-0.78190	-0.08528	0.59410	2.17546

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	9.9419	0.1264	78.66	<2e-16	***
age.L	3.6275	0.2189	16.57	<2e-16	***
age.Q	-6.1060	0.2189	-27.89	<2e-16	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9791 on 57 degrees of freedom

Multiple R-squared: 0.9486, Adjusted R-squared: 0.9468

F-statistic: 526.2 on 2 and 57 DF, p-value: < 2.2e-16

The coefficients for `age.L` and `age.Q` are the linear and quadratic trends, respectively. In this case, there is a positive linear trend (3.5113), so Y generally increases with age category. There is also a significant negative quadratic trend (-6.2328), meaning that the relationship is curved. These trends describe the *shape* of the trend with respect to the age categories, not literally the values of Y given X (as in a polynomial equation), so you need to inspect the boxplot or a plot of means.

This analysis was simple because there were only 3 levels of `age`. When an ordered factor has many levels, you may observe “significant” polynomial trends that are biologically unimportant or difficult to interpret. It is up to you as a biologist to exercise some judgement in deciding how many terms to interpret. For example, a quadratic pattern can be readily interpreted biologically. But what about a quintic? Or an octic?

Chapter 16

Tests for continuous patterns

This module discusses classic tests for patterns where we relate changes in one continuous variable to another continuous variable. By convention, the **response** or **dependent** variable is plotted on the y-axis, and is taken to be the “effect” being studied. The **predictor** or **independent** or **explanatory** variable the one interpreted as the cause. Predictors are plotted on the x-axis. Such a plot or model is said to be of “y versus x”.

However, as we see in the section on correlation, we can also investigate patterns between variables without assuming that one causes the other—only that they reliably change together.

16.1 Correlation

Correlation is a statistical relationship where changes in one variable are reliably associated with changes in another variable. This relationship need not be causal, hence the famous mantra, “correlation does not equal causation”. As such, correlation is usually a **descriptive technique**, which lets researchers characterize patterns, rather than an **inferential technique**, which lets researchers make estimates about unknown parameters (i.e., infer unobserved quantities).

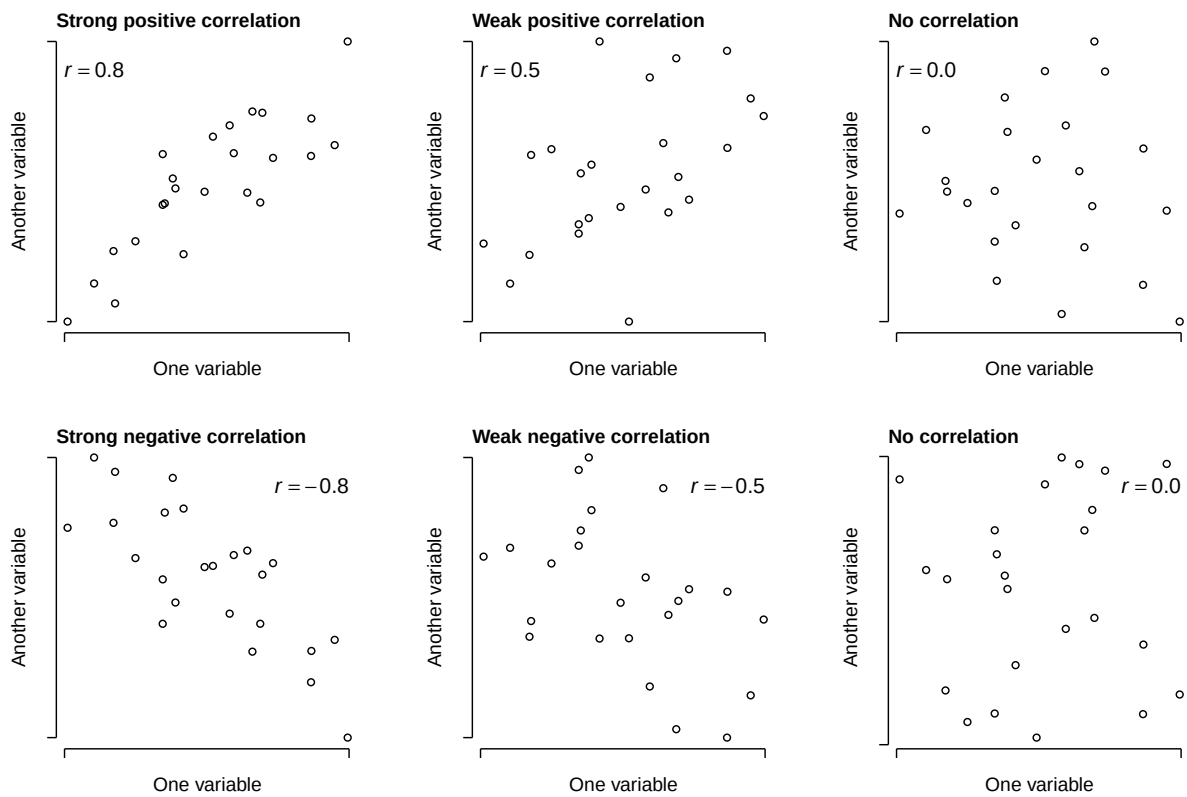
Correlations are typically expressed as a **correlation coefficient**, a value that indicates the strength of the correlation by its magnitude and the direction of the correlation by its sign.

- **Positive correlation:** as one variable gets larger, the other variable does too. Correlation coefficient approaches $+1$.
- **Negative correlation:** as one variable gets larger, the other variable gets smaller. Correlation coefficient approaches -1 .

When a correlation coefficient is near 0, this indicates no correlation.

Correlation coefficients are often calculated when we want to see if two values are related, without going as far as to infer a causal relationship. Because science is all about finding explanations (i.e., causes) for

Figure 16.1: Illustration of possible positive and negative relationships between pairs of variables, with different levels of consistency (r).



natural phenomena, a correlation analysis is usually not presented on its own. However, correlation analysis can be part of a larger workflow:

- Correlations can indicate if some variables in a dataset are *redundant*—that is, any analysis of one would say the same as analyzing the other.
- Correlations can indicate association without inference—e.g., it might be sufficient to know that your residuals are correlated with some other variable, without being able to model the relationship.
- Correlations can help identify proxy variables—variables that can stand in for one another. E.g., correlation analysis may reveal that an inexpensive, easily obtainable variable is a good stand-in for an expensive, but less obtainable one.
- Correlations can screen large numbers of variables for possible patterns and inform future investigations. For example, in some analyses I will start by testing many variables for correlations, and only invest time fitting inferential models (e.g., GLMs) for the variables that show some correlation with the response variable.

16.1.1 Linear correlation

Question: As values in one set change, do values in another set change at a constant rate?

Null hypothesis: N/A (descriptive statistic)

Alternative hypothesis: N/A (descriptive statistic)

Example use case: Lisa is studying whether the size of a mammal's cecum is related to the proportion of plant material in its diet, and suspects that the relationship is linear.

When we suspect that two variables are correlated, and that the shape of that relationship is a straight line, we can use **Pearson's product moment correlation**, also called **Pearson's r** . This is probably the most commonly used correlation coefficient, and can be calculated in many ways. One way is this:

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

In this expression, r_{xy} is the correlation between vectors x and y ; n is the total number of observations; x_i and y_i are the i -th value of x and y , respectively; $\bar{x}$ and $\bar{y}$ are the sample means of x and y . This expression shows that r gets larger as the slope of the relationship between x and y gets larger, regardless of sign; and that increasing variance decreases r .

The numerator is also called the **covariance** of x and y . You might recognize the denominator as the product of the SD of x and y . So, the formula for Pearson's r can be written very compactly as:

$$r_{xy} = \frac{\text{cov}(x, y)}{s_x s_y}$$

The base function `cor()` calculates correlation coefficients, including r .

```
# calculate linear correlation coefficient
cor(iris$Petal.Length, iris$Petal.Width)
```

```
[1] 0.9628654
```

You can also request a significance test on the coefficient, although some might consider this unnecessary because correlation is inherently a descriptive technique. If you are using correlation as a hypothesis test, however, you do need a p -value.

```
cor.test(iris$Petal.Length, iris$Petal.Width)
```

```
Pearson's product-moment correlation

data:  iris$Petal.Length and iris$Petal.Width
t = 43.387, df = 148, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9490525 0.9729853
sample estimates:
      cor
0.9628654
```

You can also get correlations between a set of variables. The result is a correlation matrix (see Section 6.4.1.3).

```
# correlations between several variables at once:
# i.e., a correlation matrix
cor(iris[,1:4])
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Sepal.Length	1.0000000	-0.1175698	0.8717538	0.8179411
Sepal.Width	-0.1175698	1.0000000	-0.4284401	-0.3661259
Petal.Length	0.8717538	-0.4284401	1.0000000	0.9628654
Petal.Width	0.8179411	-0.3661259	0.9628654	1.0000000

There isn't a built-in way to check for significant correlations between many pairs of variables at once. This is because `cor.test()` only works with one pair at a time. Fortunately, it's not hard to do in a `for()` loop:

```

# data frame with just the variables we need
dx <- iris[,1:4]

# set up data frame to hold results
## get the names of the variables
var.names <- names(dx)
## get every unique combination of 2 of those names
var.pairs <- combn(var.names, 2)
## transpose the result (swap rows and columns)
## so it is oriented like a data frame, and convert
## from matrix to data frame
cor.df <- as.data.frame(t(var.pairs))
cor.df$r <- NA
cor.df$p <- NA

#### NOTE: you can create the object cor.df more      ##
##### quickly by combining commands like this:      ##
##### not run:                                       ##
##### cor.df <- as.data.frame(t(combn(names(dx),2))) ##

# find the r and p values in a loop
for(i in 1:nrow(cor.df)){
  # test once per iteration
  curr <- cor.test(dx[,cor.df$V1[i]], dx[,cor.df$V2[i]])
  # pull out values we need from correlation test result
  # and round to 3 decimal places
  cor.df$r[i] <- round(curr$estimate, 3)
  cor.df$p[i] <- round(curr$p.value, 3)
}#close i loop

# print result
cor.df

```

	V1	V2	r	p
1	Sepal.Length	Sepal.Width	-0.118	0.152
2	Sepal.Length	Petal.Length	0.872	0.000
3	Sepal.Length	Petal.Width	0.818	0.000
4	Sepal.Width	Petal.Length	-0.428	0.000
5	Sepal.Width	Petal.Width	-0.366	0.000
6	Petal.Length	Petal.Width	0.963	0.000

It's not pretty, but we can see right away that every pair is significantly correlated except sepal width and

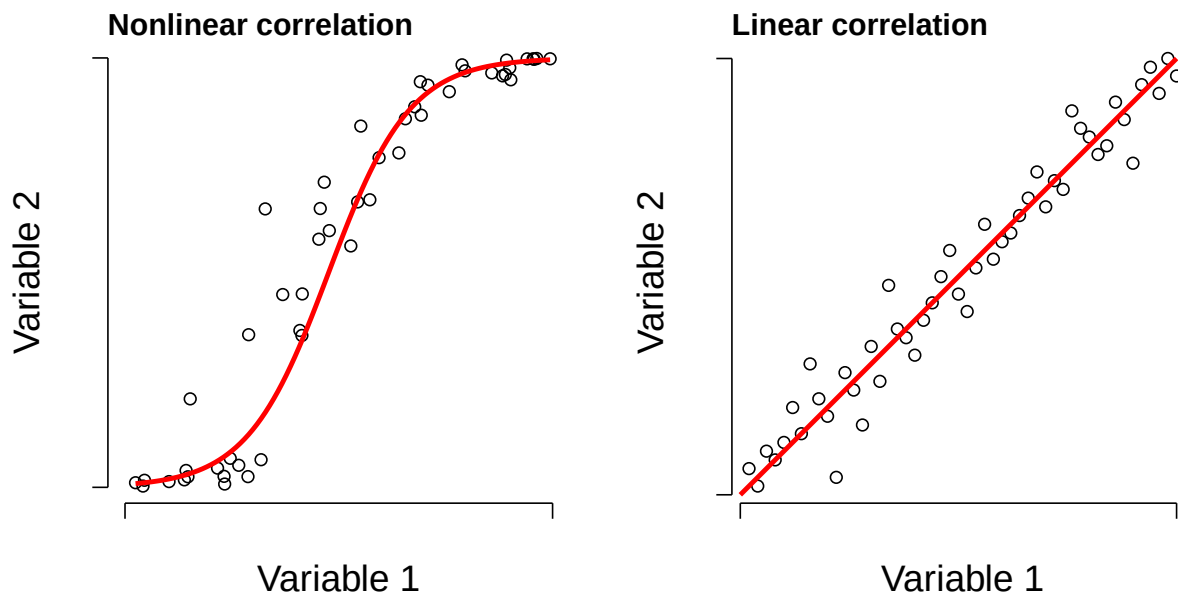
sepal length. Remember that the p -values printed as 0.000 are not literally 0; report them as “<0.001”.

Another way to explore correlations between many pairs of variables is to visualize them in a scatterplot matrix (see Section 6.4.1.3).

16.1.2 Nonlinear correlation

Many processes and relationships in biology are not linear, and but we still want to study correlations in those processes. So, instead of asking whether or not the association is linear, we ask whether it is **monotonic**, meaning that the changes in one variable associated with changes in another tend to have the same sign. For example, as X increases, Y either consistently increases or consistently decreases, but does not go up and down across the domain of X .

Consider the figure below. The left panel clearly shows some relationship between X and Y , but it is not linear and using a linear correlation coefficient might miss something. The right panel shows the same data points after a **rank transform**.



Often when two variables share a nonlinear relationship, a rank transform will make that relationship appear more linear. A “rank-transform” assigns the smallest value value “0”, the second smallest value to “1”, and so on. Rank-transforming these data lets us calculate a nonparametric, nonlinear correlation.

16.1.2.1 Spearman's ρ

The nonparametric equivalent of Pearson's r is Spearman's ρ (“rho”). Spearman's ρ is calculated as the Pearson's r linear correlation coefficient of the rank-transformed values.

$$\rho_{xy} = \frac{\text{cov}(R[x], R[y])}{s_{R[x]}s_{R[y]}}$$

When observations have identical values, their ranks may be tied, which creates some minor issues in the calculation of ρ and its p -value. For this reason, R may throw a warning when calculating a Spearman correlation. You can ignore this warning.

It can be obtained using code similar to what was done for linear correlation.

```
# just the coefficient
cor(iris$Petal.Length, iris$Sepal.Width, method="spearman")
```

```
[1] -0.3096351
```

```
# confirming the connection to Pearson's r
cor(rank(iris$Petal.Length), rank(iris$Sepal.Width))
```

```
[1] -0.3096351
```

```
# can also get a significance test for rho
cor.test(iris$Petal.Length, iris$Sepal.Width, method="spearman")
```

Spearman's rank correlation rho

```
data: iris$Petal.Length and iris$Sepal.Width
S = 736637, p-value = 0.0001154
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
-0.3096351
```

Just as with linear correlation, you can get a correlation matrix of nonlinear ρ correlation coefficients, and a scatterplot matrix with `pairs()`. See Section [6.4.2.1](#).

16.1.2.2 Kendall's τ

There is another nonparametric rank-based correlation coefficient, Kendall's τ ("tau"). Kendall's τ is based not upon a correlation coefficient between ranked observations, but instead on the concordance or agreement between the rankings of x and y . This means that for any pair of observations (x_i, y_i) and (x_j, y_j) , the ranking order of (x_i, x_j) and (y_i, y_j) agree. Put another way, if both $x_i > x_j$ and $y_i > y_j$ **or** $x_i < x_j$ and $y_i < y_j$, then the pair of observations i and j is **concordant**. Any other finding is **discordant**. The original version of Kendall's τ , now called τ_a is thus calculated:

$$\tau_{xy} = \frac{(\text{number of concordant pairs}) - (\text{number of discordant pairs})}{(\text{number of pairs})}$$

There is a [more modern version](#) of τ , called τ_b , that can account for ties. This is what the R functions `cor()` and `cor.test()` calculate.

In practice, Kendall's τ and Spearman's ρ tend to give qualitatively similar results, so you can often choose freely between them. However, there are some situations where τ might be preferable. First, the magnitude of τ has a much more intuitive meaning than that of ρ . It is related to the degree to which concordant pairs are more common than discordant pairs. E.g., $\tau = 0.6$ indicates roughly that concordant pairs are 60% more common than discordant pairs. When $\rho = 0.6$, on the other hand...well, $\rho = 0.6$.

Second, τ can work better at small sample sizes because its sampling distribution and exact tests are well developed.

Finally, τ may be preferred when the data are genuinely ordinal, because of how comparison between ranks is central to the calculation of τ . This is philosophical rather than computational: ρ works just fine on ordinal data, too.

That last thing I'll add is that while ρ and τ address the same question, they go about it in different ways and so they are on different scales. This is why they give qualitatively similar results even when the numbers are different. In fact, ρ and τ have a known relationship under certain conditions:

$$\tau = \frac{2}{\pi} \arcsin(\rho)$$

You can obtain Kendall's τ in R with the `method` argument.

```
# just the coefficient
cor(iris$Petal.Length, iris$Sepal.Width, method="kendall")
```

```
[1] -0.1859944
```

```
# can also get a significance test for rho
cor.test(iris$Petal.Length, iris$Sepal.Width, method="kendall")
```

```
Kendall's rank correlation tau
```

```
data: iris$Petal.Length and iris$Sepal.Width
z = -3.2197, p-value = 0.001283
alternative hypothesis: true tau is not equal to 0
sample estimates:
      tau
-0.1859944
```

16.2 Linear models

We reviewed the definition and properties of linear models in Section 14.4. Linear models are an important part of classical, frequentist statistics because of their power and, historically, their ease of computation. Many methods that are taught in introductory statistics classes are just special cases of the linear model: **t-tests**, **ANOVA**, **linear regression**, and so on. As we will see in Chapter 18, linear models themselves are a special case of an even more powerful, broader framework called the **generalized linear model (GLM)**.

16.2.1 Lots of analyses are linear models in disguise (or can be)

One of the reasons that linear models are so powerful is that many models or curves that people fit to data, even some whose curves are nonlinear, are actually linear models in a statistical sense. A “linear model” here does not mean that the graph must be a straight line. It means that the model is built by adding together predictor variables that are multiplied by constants (slopes), and these slopes are used only as multipliers of the predictor variables. For example, both $y = \beta_0 + \beta_1 x$ and $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$ are linear models because the coefficients (*beta* terms) are used only as multipliers of the predictor variables.

Other models that appear nonlinear can *become* linear models through transformation. For example, an exponential model $y = ae^{\beta_1 x}$ is nonlinear, but log-transforming both sides yields the linear expression $\log(y) = \log(a) + \beta_1 x$. The section below illustrates a few of these “linear models in disguise” so you can recognize and apply these methods on your own data.

16.2.1.1 Ordinary linear models

The basic linear regression model is written as

$$y = \beta_0 + \beta_1 x$$

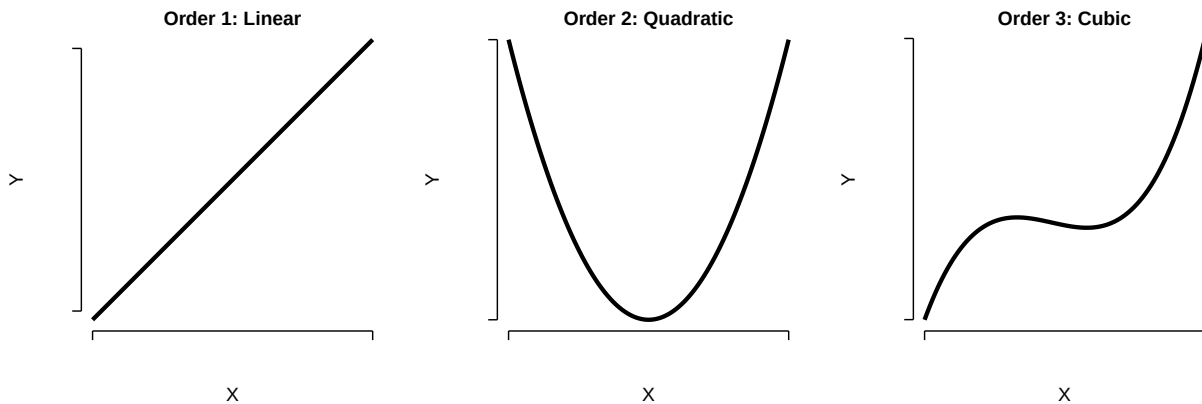
and is the same as the equation for a line that you learned in high school or college algebra: $y = mx + b$. Statisticians just use β_0 for the y -intercept and β_1 for the slope instead of b and m .

Why it’s linear: the parameters β_0 and β_1 appear only as multipliers or additive constants, and are combined by addition.

16.2.1.2 Polynomials

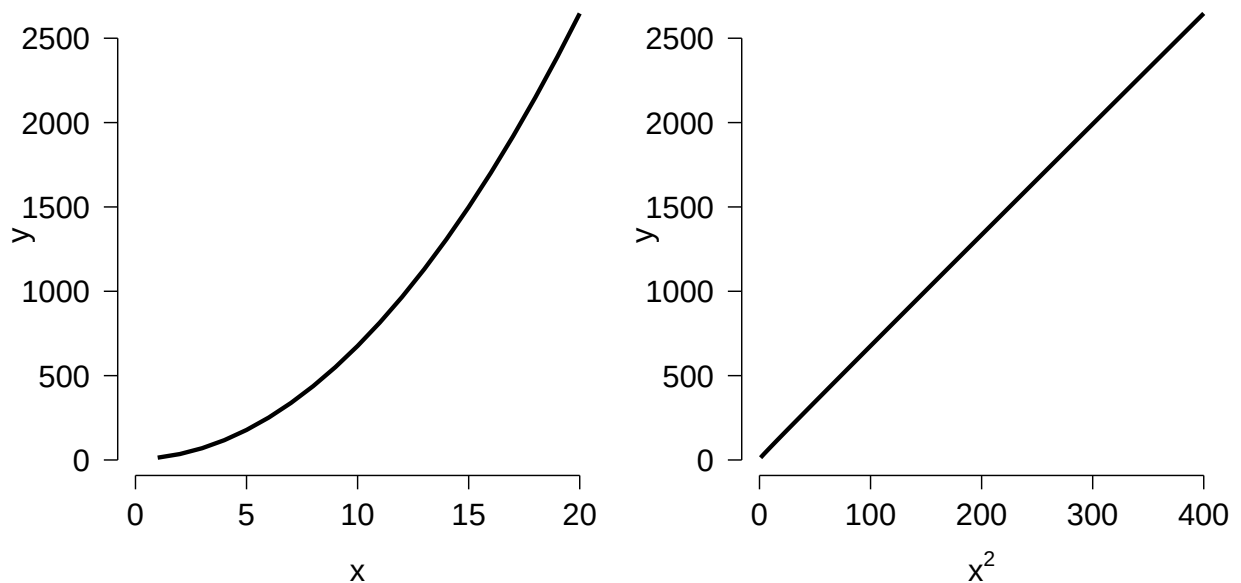
A polynomial is a function such as $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \dots + \beta_k X^k$, where k is the **order** of the polynomial. A polynomial of order 2 is a quadratic equation. A polynomial of order 3 is a cubic, and so on. Polynomials have well-known shapes, such as the parabola of the quadratic function. Any polynomial with order ≥ 2 will not be a straight line.

Figure 16.2: Curve shapes produced by polynomial equations of order 1, 2, and 3: linear, quadratic, and cubic, respectively.



Why it's linear: although the graph may be curved, the parameters β_0 , β_1 , β_2 , and so on appear only as multipliers on predictor variables, are combined by addition, and are not raised to powers, exponentiated, or use inside other functions. So, a polynomial is linear in its predictors. Look what happens in the figure below. On the left, the polynomial $5 + 2.1x + 6.5x^2$ is plotted vs. x and appears nonlinear. On the right, y is plotted against x^2 and is linear.

Figure 16.3: Left: a plot of the nonlinear polynomial function $y = 5 + 2.1x + 6.5x^2$. Right: The same function plotted against x^2 produces a straight line.



16.2.1.3 Exponentials

The exponential function $y = ae^{bx}$ is ubiquitous in biology because it describes compounding growth (like compound interest). This function is not linear, but it can be turned into and analyzed as linear by

log-transforming both sides.

Figure 16.4: Rick and Morty meme saying this sounds like linear with extra steps.



Why it *can be* linear: While the exponential function appears nonlinear, it can be made linear by log-transforming both sides:

$$\log(y) = \log(ae^{bx})$$

$$\log(y) = \log(a) + \log(e^{bx})$$

and because $\log(e^z) = z$:

$$\log(y) = \log(a) + bx$$

which is the form of a straight line with intercept $\log(a)$ and slope b .

💡 Tip

Logarithm bases

Mathematicians and statisticians almost always work with the **natural logarithm**, which uses Euler's number e (≈ 2.7182) as its base. The natural logarithm can be written $\log(x)$, $\log_e(x)$, or $\ln(x)$. Natural log is used because many [equations related to growth or change become easier](#) when e is the base. In this tutorial, and any math or statistics text, assume natural logarithm.

In contrast, scientists and engineers often use other bases, like $\log_{10}$ or $\log_2$, because these are more intuitive for humans. You can use whatever base you like so long as you are clear about what you did. If you are working with R, you will have an easier time working in natural logs because many functions are set up to use it.

16.2.1.4 Power laws

Power laws of the form $Y = aX^b$ are also ubiquitous in biology, as they describe proportional growth. Power laws are similar to exponential functions, but with a key difference: in a power law, X is in the base and is raised to a power, while in an exponential, X is in the exponent of some other base.

Why it *can be* linear: Like an exponential, a power law can be linearized by taking the logarithm of both sides.

$$\log(Y) = \log(aX^b)$$

$$\log(Y) = \log(a) + b \times \log(X)$$

In this form, the power law is linear with intercept $\log(a)$ and slope b .

16.3 Linear regression

The linear regression model describes a pattern where the response variable Y changes as a linear function of the predictor variable X . It looks like this:

$$Y = \beta_0 + \beta_1 X + \varepsilon$$

where β_0 is the y -intercept, β_1 is the slope (effect of a unit increase in X on Y), and ε is residual variation in Y not explained by X —the spread of points above or below the line. In a linear model, residual variation follows a normal distribution with mean 0 and some unknown SD σ_{res} . Each residual ε_i follows the same distribution:

$$\varepsilon_i \sim \mathcal{N}(0, \sigma_{\text{res}})$$

where $\mathcal{N}$ signifies the normal distribution.

The R syntax for linear regression is basically the same as that for ANOVA and the t -test we've seen before, or a boxplot for that matter, but this time the right hand side of the formula has a continuous predictor rather than a factor.

```
# fit a linear regression model
mod5 <- lm(Petal.Length~Petal.Width, data=iris)
```

```
# test for significance of terms
anova(mod5)
```

Analysis of Variance Table

Response: Petal.Length

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Petal.Width	1	430.48	430.48	1882.5	< 2.2e-16 ***
Residuals	148	33.84	0.23		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```
# see estimates of parameter coefficients
summary(mod5)
```

Call:

```
lm(formula = Petal.Length ~ Petal.Width, data = iris)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.33542	-0.30347	-0.02955	0.25776	1.39453

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.08356	0.07297	14.85	<2e-16 ***
Petal.Width	2.22994	0.05140	43.39	<2e-16 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4782 on 148 degrees of freedom

Multiple R-squared: 0.9271, Adjusted R-squared: 0.9266

F-statistic: 1882 on 1 and 148 DF, p-value: < 2.2e-16

The `summary()` output tells you a lot about the regression: the coefficients (and their p -values), the omnibus ANOVA for the model (the F test at the bottom), and the coefficient of determination (Adjusted R -squared). Coefficients are identified as the y -intercept β_0 (Intercept) and by the variable names (e.g., the slope with respect to petal width is `Petal.Width`). Each coefficient is presented as its estimate and SE (e.g., the intercept is 1.08 ± 0.07). The test statistic t for each coefficient is the ratio of the

estimate to the SE, and the p -value for t calculated from a t -distribution (much as in a t -test). The coefficient of determination R^2 is the proportion of variation in Y explained by the model. In this case, about 93% of variation in Petal.Length is explained by Petal.Width.

When you present a linear regression, you should present the parameters, their SE, test statistics, and p -values in a table. The omnibus F test and R^2 can be reported in Results text. Finally, you should present the data as a scatterplot, with the predicted values as a trendline and the 95% CI of that trendline if feasible. Here is how to do that for the model above:

```
# generate x values over which to draw trendline
n <- 100
minx <- min(iris$Petal.Width, na.rm=TRUE)
maxx <- max(iris$Petal.Width, na.rm=TRUE)
px <- seq(minx, maxx, length=n)

# put in a data frame with same variable names as in the model
prx <- data.frame(Petal.Width=px)

# calculate predicted values and their SE
pred <- predict(mod5, newdata=prx,
                se.fit=TRUE, interval="confidence")

# best way to get expected value and 95% confidence limits:
# (use this with linear models)
prx$mn <- pred$fit[, "fit"]
prx$lo <- pred$fit[, "lwr"]
prx$up <- pred$fit[, "upr"]

# when using methods other than linear regression, you can
# approximate the 95% CI using the predicted values and
# their SE, by drawing quantiles from the normal distribution
# defined by predicted value (pred$fit) and SEs (pred$se.fit).
# When using this with GLMs, you need to keep track of what
# values are on the link scale, and which are on the
# data scale.
### not run:
### prx$mn <- qnorm(0.5, mean=pred$fit, sd=pred$se.fit)
### prx$lo <- qnorm(0.025, mean=pred$fit, sd=pred$se.fit)
### prx$up <- qnorm(0.975, mean=pred$fit, sd=pred$se.fit)

# make the plot
## include some par options for a more professional figure
```



```

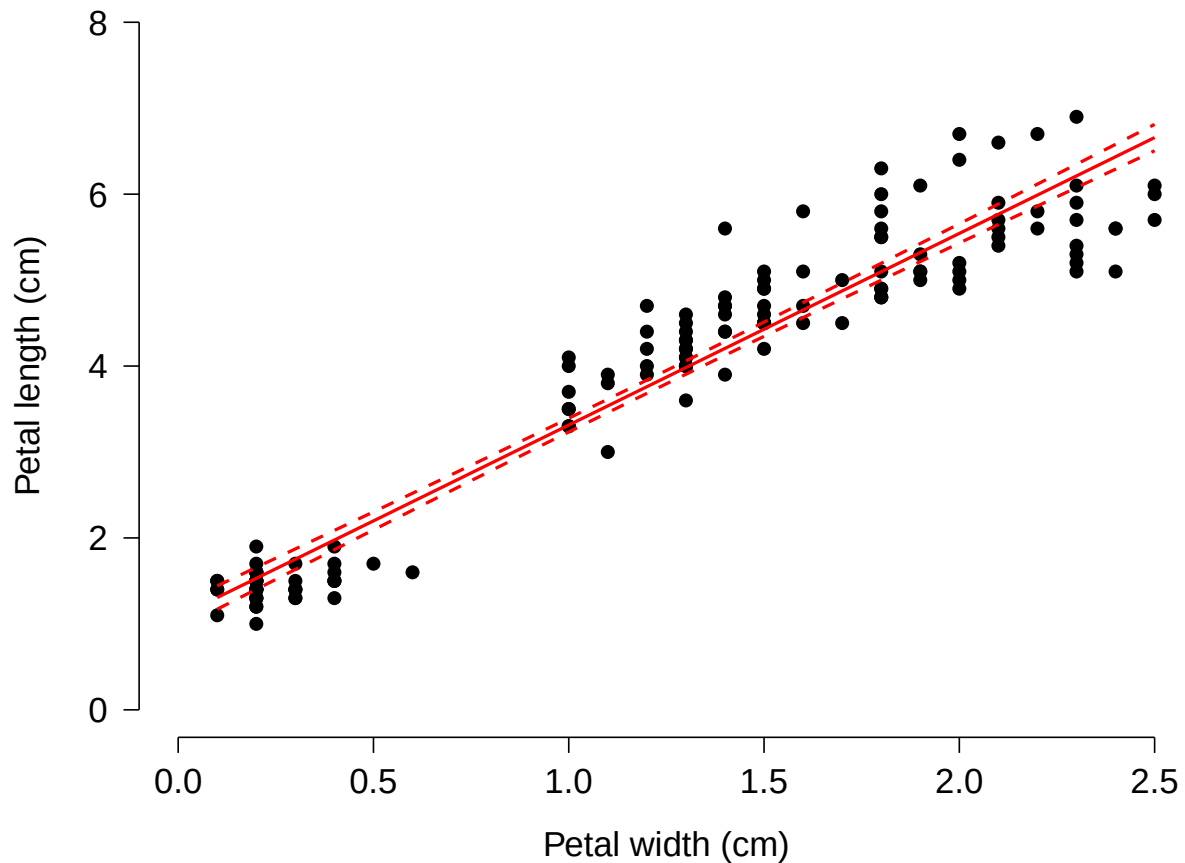
## see ?par for more options
par(mfrow=c(1,1),          # 1 x 1 figure layout
    mar=c(5.1, 5.1, 1.1, 1.1), # margin sizes in lines of text
    lend=1,                # flat ends on lines
    las=1,                 # axis text in reading orientation
    bty="n",               # no box around figure
    cex.axis=1.3,          # size of axis text
    cex.lab=1.3)           # size of axis labels

## make the base plot
plot(iris$Petal.Width,      # x values
     iris$Petal.Length,    # y values
     pch=16,               # type of points (see ?points)
     cex=1.2,              # size of points
     xlab="Petal width (cm)", # x axis label
     ylab="Petal length (cm)", # y axis label
     xlim=c(0, 2.5),        # limits of x axis
     ylim=c(0, 8))          # limits of y axis

## add trendline and 95% confidence limits as lines
## with points(...,type="l")
points(prx$Petal.Width,    # x values
       prx$lo,             # y values
       type="l",           # lines (see ?base::plot)
       lwd=2,              # line width (1 is default)
       col="red",          # line color ("black" is default)
       lty=2)              # dashed line (1, solid line, is default)
points(prx$Petal.Width, prx$up, type="l", lwd=2, col="red", lty=2)
points(prx$Petal.Width, prx$mn, type="l", lwd=2, col="red")

```

Figure 16.5: Data plotted with the trendline (solid) and its 95% confidence interval (dashed).



16.4 Multiple regression

Linear regression is easily extended to **multiple linear regression**, where there are >1 continuous predictors. Just as with ANOVA, additional predictor variables are added to the model formula using +.

```
mod6 <- lm(Petal.Length~Petal.Width + Sepal.Width, data=iris)
summary(mod6)
```

Call:

```
lm(formula = Petal.Length ~ Petal.Width + Sepal.Width, data = iris)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.33753	-0.29251	-0.00989	0.21447	1.24707

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	2.25816	0.31352	7.203	2.84e-11	***
Petal.Width	2.15561	0.05283	40.804	< 2e-16	***
Sepal.Width	-0.35503	0.09239	-3.843	0.00018	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4574 on 147 degrees of freedom
Multiple R-squared: 0.9338, Adjusted R-squared: 0.9329
F-statistic: 1036 on 2 and 147 DF, p-value: < 2.2e-16

Adding predictors to a multiple regression model will almost always increase the model fit (e.g., proportion of variance explained, R^2), but this does not mean you should keep adding predictors. Multiple regression models are vulnerable to **overfitting**, where random noise (i.e., residual variation) is modeled as if it was part of the deterministic part of the model. Multiple regression models are also highly vulnerable to **collinearity**, where the predictors are correlated with each other. When collinear predictors are included in a model, it is impossible for the model to unambiguously partition the sums of squares associated with each predictor.

When plotting the results of a multiple linear regression, you should present the effects of all significant predictors. This is done by plotting the expected y value across a range of each predictor, while holding the other predictor (or predictors) constant at their median. This will show the marginal effect of each predictor on its own. Here is a quick example:

```
# simulate some data
## random number seed
set.seed(42)

## number of observations
n <- 100

## x variables
x1 <- runif(n, 1, 20)
x2 <- runif(n, 1, 30)

## coefficients
beta0 <- 121
beta1 <- 6.1
beta2 <- -3.1

## expected y values
ey <- beta0 + beta1*x1 + beta2*x2
```

```
# add residual variation
sigma.res <- 7.8
y <- ey + rnorm(n, 0, sigma.res)

# put in a data frame
dx <- data.frame(y=y, x1=x1, x2=x2)

# fit the model
mod7 <- lm(y~x1+x2, data=dx)

# check significance of terms
anova(mod7)
```

Analysis of Variance Table

Response: y

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
x1	1	125985	125985	2486.3	< 2.2e-16 ***
x2	1	62984	62984	1243.0	< 2.2e-16 ***
Residuals	97	4915	51		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

And check the coefficients:

```
summary(mod7)
```

Call:

```
lm(formula = y ~ x1 + x2, data = dx)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-15.2269	-4.4098	0.0963	4.2975	21.6348

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	120.23351	2.11508	56.85	<2e-16 ***
x1	6.06715	0.12478	48.62	<2e-16 ***
x2	-3.07237	0.08715	-35.26	<2e-16 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.118 on 97 degrees of freedom
Multiple R-squared: 0.9746, Adjusted R-squared: 0.9741
F-statistic: 1865 on 2 and 97 DF, p-value: < 2.2e-16

Now make a figure showing $y \sim x_1$ and $y \sim x_2$. In each panel, hold the other x value at its median to show the marginal effect of the plotted x value.

```
# define sequences of each x value
plot.n <- 100

minx1 <- min(dx$x1, na.rm=TRUE)
maxx1 <- max(dx$x1, na.rm=TRUE)
px1 <- seq(minx1, maxx1, length=plot.n)

minx2 <- min(dx$x2, na.rm=TRUE)
maxx2 <- max(dx$x2, na.rm=TRUE)
px2 <- seq(minx2, maxx2, length=plot.n)

# medians of x1 and x2
medx1 <- median(x1, na.rm=TRUE)
medx2 <- median(x2, na.rm=TRUE)

# assemble data frames with all predictors in the model
prx1 <- data.frame(x1=px1, x2=medx2)
prx2 <- data.frame(x1=medx1, x2=px2)

# generate predictions
pred1 <- predict(mod7, newdata=prx1,
                  interval="confidence", se.fit=TRUE)
pred2 <- predict(mod7, newdata=prx2,
                  interval="confidence", se.fit=TRUE)

# put expected values and 95% CI in their
# respective data frames
prx1$mn <- pred1$fit[, "fit"]
prx1$lo <- pred1$fit[, "lwr"]
prx1$up <- pred1$fit[, "upr"]

prx2$mn <- pred2$fit[, "fit"]
prx2$lo <- pred2$fit[, "lwr"]
prx2$up <- pred2$fit[, "upr"]
```

```

# now make the plot

## include some par options for a more professional figure
## see ?par for more options
par(mfrow=c(1,2),          # 1 x 2 figure layout
    mar=c(5.1, 5.1, 1.1, 1.1), # margin sizes in lines of text
    lend=1,                  # flat ends on lines
    las=1,                   # axis text in reading orientation
    bty="n",                  # no box around figure
    cex.axis=1.3,            # size of axis text
    cex.lab=1.3,             # size of axis labels
    xpd=NA)                  # allows points to print in margin

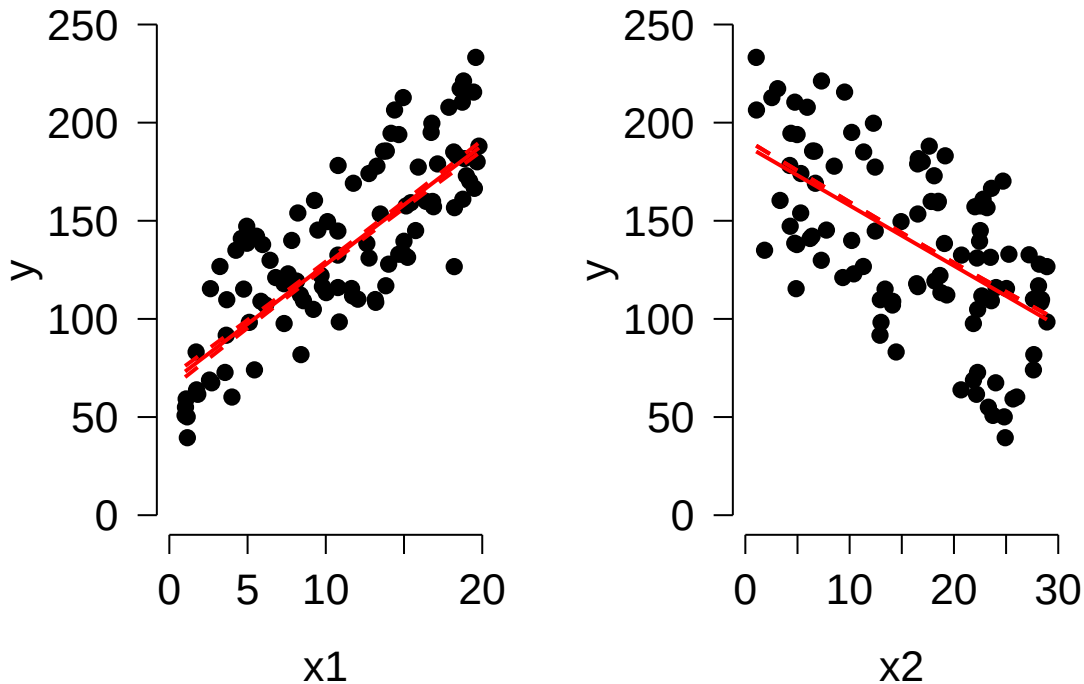
## make the first panel
plot(dx$x1,                 # x values
     dx$y,                  # y values
     pch=16,                # type of points (see ?points)
     cex=1.2,               # size of points
     xlab="x1",             # x axis label
     ylab="y",              # y axis label
     xlim=c(0, 20),         # limits of x axis
     ylim=c(0, 250))        # limits of y axis

## add trendline and 95% confidence limits as lines
## with points(...,type="l")
points(prx1$x1, # x values
       prx1$lo, # y values
       type="l", # lines (see ?base::plot)
       lwd=2,    # line width (1 is default)
       col="red", # line color ("black" is default)
       lty=2)    # dashed line (1, solid line, is default)
points(prx1$x1, prx1$up, type="l", lwd=2, col="red", lty=2)
points(prx1$x1, prx1$mn, type="l", lwd=2, col="red")

# make the second panel
## make the first panel
plot(dx$x2, dx$y,
     pch=16, cex=1.2,
     xlab="x2", ylab="y",
     xlim=c(0, 30), ylim=c(0, 250))
points(prx2$x2, prx2$up, type="l", lwd=2, col="red", lty=2)

```

```
points(prx2$x2, prx2$up, type="l", lwd=2, col="red", lty=2)
points(prx2$x2, prx2$mn, type="l", lwd=2, col="red")
```



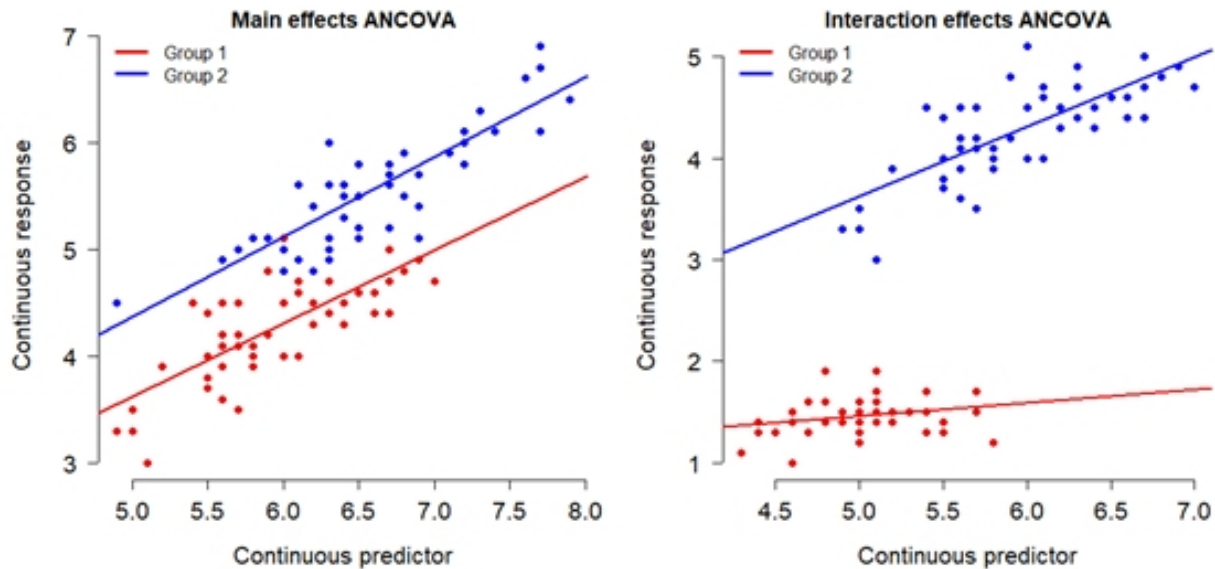
16.5 Analysis of covariance (ANCOVA)

Sometimes biologists want to test for effects of one variable while controlling for another. For example, the slope between Y and X might differ between groups. Or, groups might differ only after accounting for some continuous covariate. Here are some specific examples:

- Herbivorous mammals tend to have longer intestines than carnivorous mammals, but only after accounting for the effect of body mass on intestine length. The researcher wants to see how groups differ, but needs to control for a continuous variable (body length) first.
- Humerus length and femur length both increase with body mass in birds, but humerus length increases faster than femur length (i.e., their slopes are different). The researcher wants to test for a difference in a continuous relationship, but needs to control for a factor (which bone) first.

The method for both situations is called **analysis of covariance**, or **ANCOVA**. You can think of ANCOVA as combining linear regression and ANOVA. ANCOVA is especially useful for studying **interactions**, where one variable changes the effect of another.

Figure 16.6: Illustration of ANCOVA without an interaction (different intercepts, same slope) and with an interaction (different intercepts and slopes.)



So that we really understand what an ANCOVA does, let's simulate our own dataset where groups A and B have different slopes.

```
# simulation parameters
set.seed(42)
n <- 30
beta0_a <- 2
beta0_b <- 6
beta1_a <- 1.4
beta1_b <- 2.5
sigma_res <- 2

# groups A and B start in separate data frames
data_a <- data.frame(group="A", col="red", pch=15, x=runif(n, 5, 20))
data_b <- data.frame(group="B", col="blue", pch=16, x=runif(n, 5, 20))

# add expected values
data_a$ey <- beta0_a + beta1_a * data_a$x
data_b$ey <- beta0_b + beta1_b * data_b$x

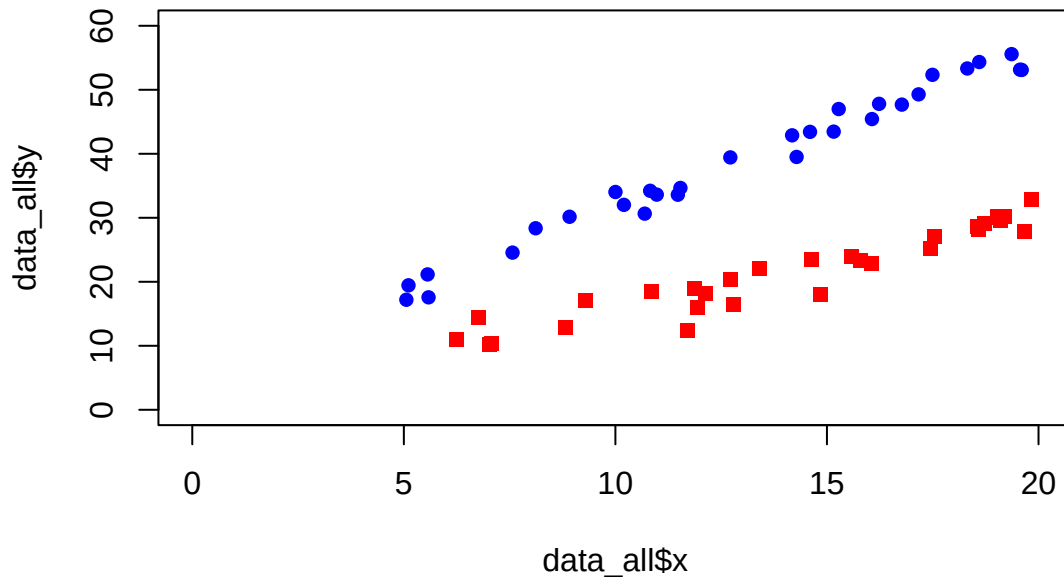
# combine to single data frame
data_all <- rbind(data_a, data_b)

# add residuals
```



```
data_all$y <- data_all$ey + rnorm(n*2, mean=0, sd=sigma_res)

# and make a plot to inspect our data
plot(data_all$x, data_all$y, col=data_all$col, pch=data_all$pch,
      xlim=c(0, 20), ylim=c(0, 60))
```



Now we can fit the ANCOVA and see the results.

```
# fit the model. Notice the * to specify an interaction
mod7 <- lm(y~x*group, data=data_all)

# test significance of terms
anova(mod7)
```

Analysis of Variance Table

Response: y

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
x	1	3507.2	3507.2	1012.289	< 2.2e-16	***
group	1	5664.0	5664.0	1634.827	< 2.2e-16	***
x:group	1	311.0	311.0	89.778	3.108e-13	***
Residuals	56	194.0	3.5			

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Now check the model parameters

```
summary(mod7)
```

Call:

```
lm(formula = y ~ x * group, data = data_all)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-5.5122	-0.9491	0.4801	1.1691	3.8166

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	0.44990	1.17053	0.384	0.702170
x	1.49194	0.07883	18.925	< 2e-16 ***
groupB	5.62958	1.55677	3.616	0.000642 ***
x:groupB	1.03148	0.10886	9.475	3.11e-13 ***

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Residual standard error: 1.861 on 56 degrees of freedom

Multiple R-squared: 0.9799, Adjusted R-squared: 0.9789

F-statistic: 912.3 on 3 and 56 DF, p-value: < 2.2e-16

R treats group A as the control or **baseline** because it is alphabetically first. Any effect involving group B is relative to the effect in group A. Here:

- The intercept in group A is (Intercept), 0.4499.
- The slope in group A is x, 1.4919.
- The intercept in group B is the group A intercept plus the effect of being in group B: $0.4499 + 5.6295 = 6.0794$.
- The slope in group B is the slope in group A plus the effect of being in group B on the slope: $1.4919 + 1.0314 = 2.5233$

Making a plot with trendlines for an ANCOVA follows much the same procedure as we used to make the plot for linear regression, but, you have to take care to include each group. Notice that when plotting a line for each group, we make sure to define the line only for the range of x values present in that group in the data.

```

# get x sequence within the x domain of each group
pxa <- seq(min(data_all$x[data_all$group=="A"]),
           max(data_all$x[data_all$group=="A"]),
           length=100)
pxb <- seq(min(data_all$x[data_all$group=="B"]),
           max(data_all$x[data_all$group=="B"]),
           length=100)
# put in data frames
prxa <- data.frame(group="A", x=pxa)
prxb <- data.frame(group="B", x=pxb)

# calculate predictions
preda <- predict(mod7, newdata=prxa, se.fit=TRUE)
predb <- predict(mod7, newdata=prxb, se.fit=TRUE)

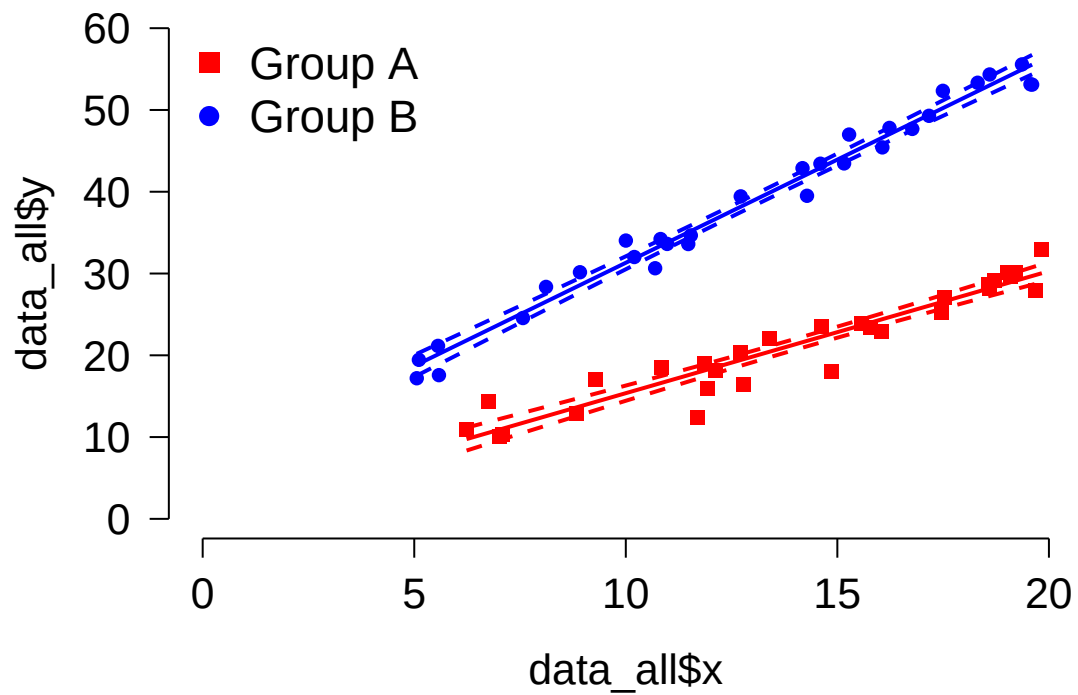
# add to data frames
prxa$mn <- qnorm(0.5, preda$fit, preda$se.fit)
prxa$lo <- qnorm(0.025, preda$fit, preda$se.fit)
prxa$up <- qnorm(0.975, preda$fit, preda$se.fit)

prxb$mn <- qnorm(0.5, predb$fit, predb$se.fit)
prxb$lo <- qnorm(0.025, predb$fit, predb$se.fit)
prxb$up <- qnorm(0.975, predb$fit, predb$se.fit)

# make the plot
par(mar=c(5.1, 5.1, 1.1, 1.1), bty="n", las=1, lend=1,
    cex.axis=1.3, cex.lab=1.3)
plot(data_all$x, data_all$y, col=data_all$col, pch=data_all$pch,
     xlim=c(0, 20), ylim=c(0, 60))
points(prxa$x, prxa$mn, type="l", lwd=2, col="red")
points(prxa$x, prxa$lo, type="l", lwd=2, col="red", lty=2)
points(prxa$x, prxa$up, type="l", lwd=2, col="red", lty=2)
points(prxb$x, prxb$mn, type="l", lwd=2, col="blue")
points(prxb$x, prxb$lo, type="l", lwd=2, col="blue", lty=2)
points(prxb$x, prxb$up, type="l", lwd=2, col="blue", lty=2)
legend("topleft", legend=c("Group A", "Group B"),
     pch=c(15, 16), col=c("red", "blue"), cex=1.4,
     bty="n")

```

Figure 16.7: Plot of Y vs. X with groups A and B in the data, where the groups have different slopes. This illustrates ANCOVA with an interaction between X and group.



16.5.1 Main effects and interaction effects

Chapter 17

Tests for contingency

Tests for contingency, or tests for independence, are tests where both the predictor and the response variables are **categorical**. The question being asked is usually some version of, “If an observation is in category A, does it also tend to be in category B?”.

Consider the (made up) data from a pesticide exposure experiment below:

Table 17.1: Pollen production by pesticide treatment.

Pollen production	Unexposed	Exposed
None	17	6
Reduced	3	4
Normal	4	14

This table tallies the fates of 48 plants that were *exposed* or *not exposed* to an experimental pesticide. The outcomes were classified as *no pollen production*, *reduced pollen production*, and *normal pollen production*. Rather than asking a question about the mean pollen production in each group, the researchers might ask whether the fate of a plant (in terms of pollen production) is independent of whether or not it was treated. This is why the tests in this module are sometimes called **tests of independence**.

A good clue for whether or not you need to do a test for independence is whether or not your data are best summarized in a contingency table (Section 7.3.2). A contingency table shows how often observations fall into each category within a dataset.

17.1 Fisher’s exact test

Fisher’s exact test is a test for independence in contingency tables that is usually only used when expected values are small ($\lesssim 5$) or when the χ^2 test gives questionable approximations.

Question: Are the proportions of data described by two or more categorical variables random or nonrandom?

Null hypothesis: Membership in one of one set of categories is independent of membership in one of another set of categories.

Alternative hypothesis: Membership in one of one set of categories is not independent of membership in one of another set of categories.

Example use case: Rick is studying if whether a fish spawning or not spawning is related to whether it resides in main channel or off-channel habitat, and he has dozens of fish in his dataset.

Fisher's exact test is a test for independence in contingency tables that is usually only used when the some of the expected values are 5 or less. Although historically it was not used at large sample sizes because the p -value calculation involves numerous factorials that were intractable with older computers, modern algorithms can use computational shortcuts to shorten this substantially. In fact, the Fisher's exact test is valid at any sample size, but tends to be more conservative than alternatives like the χ^2 test.

```
# make a matrix from the table above
a <- matrix(c(17,6,3,4,4,14), byrow=TRUE, nrow=3)
a
```

```
      [,1] [,2]
[1,]   17    6
[2,]    3    4
[3,]    4   14
```

The expected values, ironically, can be calculated by the `chisq.test()` function:

```
chisq.test(a)$expected
```

```
      [,1] [,2]
[1,] 11.5 11.5
[2,]  3.5  3.5
[3,]  9.0  9.0
```

Notice that some of the expected cell counts are less than 5. In such situations, the χ^2 approximation can be suspect and the Fisher's test is preferred.

```
fisher.test(a)
```

Fisher's Exact Test for Count Data

```
data:  a
p-value = 0.002849
alternative hypothesis: two.sided
```

Because Fisher's is an exact test (i.e., the P -value is an direct, exact calculation), there are no test statistics or degrees of freedom to report.

17.2 Chi-squared test

The chi-square (χ^2) test is a more powerful, flexible, and computationally efficient cousin of the Fisher's exact test. It works by comparing the input contingency table to what the contingency table would be expected to look like under the assumption of independence. That is, if the null hypothesis were true. The test statistic, χ^2 , measures how different the two contingency tables are. The test works better with large sample sizes; with small sample sizes, the approximations use to estimate the p -value can break down. A good rule of thumb is that every cell (or most cells) of the hypothetical null contingency table should have ≥ 5 observations.

To meet the $E(n) \geq 5$ criterion, we need to collect more data for our imaginary pesticide experiment above. Consider the (made up) data from a pesticide exposure experiment below:

Table 17.2: Pollen production by pesticide treatment.

Pollen production	Unexposed	Exposed
None	33	15
Reduced	8	13
Normal	9	22

```
# make a matrix from the table above
b <- matrix(c(33, 15, 8, 13, 9, 22), byrow=TRUE, nrow=3)
b
```

```
      [,1] [,2]
[1,]   33  15
[2,]    8  13
[3,]    9  22
```

The χ^2 test assumes that every outcome category is equally likely within every input category. In other words, input and outcome categories are independent of each other. The expected cell counts can be calculated by scaling observed cell counts by the marginal and grand total:

Pollen production	Unexposed	Exposed	Row totals
None	33	15	48
Reduced	8	13	21
Normal	9	22	31

Pollen production	Unexposed	Exposed	Row totals
Column totals	50	50	100

The expected count in the cell in row i , column j , is:

$$E_{ij} = \frac{R_i C_j}{T}$$

where R_i is the total for row i , C_j is the total for column j , and T is the grand total. For example, for row 1, column 2, the expected value is:

$$E_{1,2} = \frac{48 \cdot 50}{100} = 24$$

So the entire table is:

Pollen production	Unexposed	Exposed	Row totals
None	$(48 \cdot 50) / 100 = 24$	$(48 \cdot 50) / 100 = 24$	48
Reduced	$(21 \cdot 50) / 100 = 10.5$	$(21 \cdot 50) / 100 = 10.5$	21
Normal	$(31 \cdot 50) / 100 = 15.5$	$(31 \cdot 50) / 100 = 15.5$	31
Column totals	50	50	100

Once we have the expected counts, we can calculate the χ^2 statistic as:

$$\chi^2_{stat} = \sum_{i=1}^{n_{row}} \sum_{j=1}^{n_{col}} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

The χ^2 statistic is then compared to a χ^2 distribution with $(n_{row} - 1)(n_{col} - 1)$ degrees of freedom. In this case, $DF = (3 - 1)(2 - 1) = 2$. The p -value is then calculated as:

```
1-pchisq(13.392, 2)
```

```
[1] 0.001235845
```

Fortunately, we don't have to calculate all of that by hand. The `chisq()` function includes the expected counts in its output object, and prints everything you need to report in your write-up. The χ^2 test is not an exact test, so you need to report the test statistic and degrees of freedom.


```
# chi squared test
chi.b <- chisq.test(b)
chi.b
```

Pearson's Chi-squared test

```
data:  b
X-squared = 13.392, df = 2, p-value = 0.001236
```

In your manuscript, thesis, or whatever, the test would be reported as, “pollen production class was not independent of pesticide application ($\chi^2 = 13.39$, 2 d.f., $p = 0.001$)”.

Let's check the expected counts to make sure none are < 5 .

```
# check the expected counts
# if >1 are <5, consider a Fisher's exact test
chi.b$expected
```

```
      [,1] [,2]
[1,] 24.0 24.0
[2,] 10.5 10.5
[3,] 15.5 15.5
```

17.3 G-test

The G-test is another test for independence that can be used in situations where the χ^2 test is appropriate. Its main advantage is that it is more computationally efficient at very large sample sizes (1000s or more). Its test statistic is a bit different:

$$G = 2 \sum_i^n O_i \log\left(\frac{O_i}{E_i}\right)$$

Where the terms are the same as in the equation for χ^2 and “log” is the natural logarithm function. Further, the total observed count must equal the total expected count:

$$\sum_i^n O_i = \sum_i^n E_i$$

The G-test should be used when sample sizes are large, because it is more computationally efficient than χ^2 , and in situations where $O_i > 2E_i$ for at least some cells.

Despite that fact that it has been recommended in literature for over 3 decades, the G-test is not available in the base R `stats` package. It is implemented in several add-on packages such as `AMR`.

17.4 Contingency tests and odds ratios

One interesting way to express the results of a test for independence is to use the **odds ratio** of one outcome relative to another. The **odds** of some event k with probability p is:

$$Odds(k) = \frac{p(k)}{1 - p(k)}$$

So the odds ratio of two events k and m is simply the odds of k divided by the odds of m .

$$OR(k, m) = \frac{\frac{p(k)}{1-p(k)}}{\frac{p(m)}{1-p(m)}}$$

Odds ratios make it easier to interpret relative likelihoods than probabilities. For example, if the control treatment in some experiment results in 96% mortality, and the other treatment has 98% mortality, the effect size of the treatment is not 2%. It is really $((0.98) / (0.02)) / ((0.96) / (0.04)) = 2.04$. That is, the treatment more than doubled the likelihood of mortality! Expressing that result as an odds ratio feels like it is overstating things, until you realize that the proportion of non-mortalities was cut in half. If the mortality rates were lower, say 52% vs. 54%, the odds ratio would be 1.08, meaning that mortality was 8% more likely than in the control.

Let's calculate the odds ratios for our pesticide study. Collapse the 3 outcomes into 2: normal pollen, and less than normal pollen (which includes no pollen):

Pollen production	Unexposed	Exposed	Row totals
Reduced	41	28	69
Normal	9	22	31
Column totals	50	50	100

The odds ratio for reduced pollen in the exposed treatment compared to control is:

$$\frac{\frac{28}{22}}{\frac{41}{9}} = \frac{1.273}{4.556} = 0.279$$

This means that plants in the exposed treatment were 0.279 times as likely to have reduced pollen production as plants in the control treatment. Or, you could reverse the calculation and find that exposed plants were 3.58 times as likely to have normal pollen counts.

Part V

Generalized linear models (GLM)

A link function to the past.

Chapter 18

Introduction to generalized linear models (GLM)

A link function to the past.

18.1 Prelude with linear models

18.2 GLM basics

18.2.1 Overall structure

18.2.2 Response distributions

18.2.3 Linear predictors

18.2.4 Link functions

18.3 Common GLMs

18.3.1 GLMs for continuous data

18.3.1.1 Linear regression as a GLM

18.3.1.2 GLMs for overdispersed data

18.3.2 GLMs for count data

18.3.3 GLMs for binary and proportional data

18.4 Deviance and other GLM diagnostics

18.4.1 Deviance

18.4.2 Pseudo- R^2 or not?

18.4.3 AIC and likelihood functions

Chapter 19

GLM for continuous response data

19.1 Gaussian GLM

19.2 Log-normal GLM

19.3 Gamma GLM

Chapter 20

GLM for count data

1! Hah ha ha

2! Hah ha ha

20.1 Poisson GLM for counts

20.2 Negative binomial GLM for overdispersed counts

Chapter 21

GLM for categorical and proportional responses

Even our curves have curves.

It's what's under the curve that counts.

21.1 Logistic GLM for binary responses

21.1.1 Odds ratios

21.2 Binomial GLM for proportions

21.3 Ordinal logistic regression

Part VI

Extensions of GLMs: nonlinear and mixed models

Even our curves have curves.

Chapter 22

Nonlinear methods for biology

Things are not always linear.

22.1 Nonlinear least squares

22.2 Generalized additive models (GAM)

22.3 Generalized nonlinear models (GNM)

22.4 Regression trees for nonlinear patterns

Chapter 23

Random effects and mixed models

So random.

23.1 Fixed and random effects

23.2 Linear mixed effects models (LMM)

23.3 Generalized linear mixed effects models (GLMM)

23.4 Nonlinear mixed effects models (NLME)

23.5 Additive mixed models (GAMM)

Part VII

Multivariate data analysis

The needs of the many outweigh the needs of the few, or the one.

Chapter 24

Introduction to multivariate data and data analysis

How far is too far?

Chapter 25

Dissimilarity and distance

How far is too far?

25.1 Mantel tests

Chapter 26

Clustering for biologists

Two is company, but three's a crowd.

Chapter 27

Ordination

We're all just points on a graph.

27.1 Principal components analysis

Part VIII

Alternatives to p -values: information-theoretic and Bayesian methods

... until now, you have become the very thing you swore to destroy.

Chapter 28

Multivariate hypothesis testing

Anything is permuted.

28.1 Comparing groups

28.1.1 MANOVA

28.1.2 PERMANOVA

28.1.3 MRPP

28.2 Analyzing dissimilarity

28.2.1 Mantel tests

Chapter 29

Information theoretic model-based inference

Ain't AIC grand?

Part IX

Bonus material

Material that didn't quite fit anywhere else. Future chapters will likely be added here.

Chapter 30

Introduction to Bayesian inference for biologists

Bayes bayes bayes.

Chapter 31

Advanced R graphics

Base graphics are the best graphics.

Chapter 32

Classification and regression trees

Barking up the wrong tree

Appendix A

Appendix A

NEON small mammal data

Bolker BM. 2008. Ecological models and data in R. Princeton, New Jersey, USA: Princeton University Press.

Bolker BM, Brooks ME, Clark CJ, Geange SW, Poulsen JR, Stevens MHH, White J-SS. 2009. Generalized linear mixed models: A practical guide for ecology and evolution. *Trends in Ecology & Evolution* 24(3):127–135. <https://doi.org/10.1016/j.tree.2008.10.008>

Box GEP, Cox DR. 1964. An analysis of transformations. *Journal of the Royal Statistical Society: Series B (Methodological)* 26(2):211–243. <https://doi.org/10.1111/j.2517-6161.1964.tb00553.x>

Broman KW, Woo KH. 2018. Data organization in spreadsheets. *The American Statistician* 72(1):2–10. <https://doi.org/10.1080/00031305.2017.1375989>

Burnham K, Anderson D. 2002. Model selection and multi-model inference: a practical information-theoretic approach. New York: Springer-Verlag.

Dalgaard P. 2002. Introductory statistics with R. Springer Science+Business Media.

Dormann CF, Elith J, Bacher S, Buchmann C, Carl G, Carré G, Marquéz JRG, Gruber B, Lafourcade B, Leitão PJ *et al.* 2013. Collinearity: A review of methods to deal with it and a simulation study evaluating their performance. *Ecography* 36(1):27–46. <https://doi.org/10.1111/j.1600-0587.2012.07348.x>

Gelfand AE, Vounatsou P. 2003. Proper multivariate conditional autoregressive models for spatial data analysis. *Biostatistics* 4(1):11–15. <https://doi.org/10.1093/biostatistics/4.1.11>

Green NS, Wildhaber ML, Albers JL. 2019. [Potential responses of the Lower Missouri River shovelnose sturgeon \(*Scaphirhynchus platyrhynchus*\) population to a commercial fishing ban.](#) *Journal of Applied*

Ichthyology 35(1):370–377.

Green NS, Wildhaber ML, Albers JL, Pettit TW, Hooper MJ. 2020. [Efficient mammal biodiversity surveys for ecological restoration monitoring](#). Integrated Environmental Assessment and Management.

Grolemund G, Wickham H. 2011. [Dates and times made easy with lubridate](#). Journal of Statistical Software 40(3):1–25.

Hobbs NT, Hilborn R. 2006. [Alternatives to statistical hypothesis testing in ecology: a guide to self teaching](#). Ecological Applications 16(1):5–19.

Hutcheon JM, Kirsch JAW, Jr TG. 2002. A comparative analysis of brain size in relation to foraging ecology and phylogeny in the Chiroptera. Brain, behavior and evolution 60(3):165–180. <https://doi.org/10.1159/000065938>

James G, Witten D, Hastie T, Tibshirani R. 2013. [An introduction to statistical learning](#). 7th edn. Vol. 112. Springer.

Kéry M. 2010. Introduction to WinBUGS for ecologists: Bayesian approach to regression, ANOVA, mixed models and related analyses. Academic Press.

Kirchner BN, Green NS, Sergeant DA, Mink JN, Wilkins KT. 2011. [Responses of small mammals and vegetation to a prescribed burn in a tallgrass blackland prairie](#). The American Midland Naturalist 166(1):112–125.

Kuehn I, Dormann CF. 2012. [Less than eight \(and a half\) misconceptions of spatial analysis](#). Journal of Biogeography 39(5):995–998.

Lebreton J-D, Burnham KP, Clobert J, Anderson DR. 1992. [Modeling survival and testing biological hypotheses using marked animals: a unified approach with case studies](#). Ecological Monographs 62(1):67–118.

Legendre P, Legendre L. 2012. Numerical ecology. Amsterdam: Elsevier.

McCune B, Grace JB, Urban DL. 2002. Analysis of ecological communities. MjM software design, Gleneden Beach, Oregon, USA.

National Ecological Observatory Network (NEON). 2025. Small mammal box trapping (DP1.10072.001). <https://doi.org/10.48443/YRH2-6058>

Penone C, Davidson AD, Shoemaker KT, Di Marco M, Rondinini C, Brooks TM, Young BE, Graham CH, Costa GC. 2014. [Imputation of missing data in life-history trait datasets: Which approach performs the best?](#) Methods in Ecology and Evolution 5(9):961–970.

- Ritz C, Baty F, Streibig JC, Gerhard D. 2015. Dose-response analysis using R. PLoS ONE 10(12):e0146021. <https://doi.org/10.1371/journal.pone.0146021>
- Spector P. 2008. Data manipulation with R. New York: Springer Science+Business Media.
- Warton DI, Hui FKC. 2011. The arcsine is asinine: The analysis of proportions in ecology. Ecology 92(1):3–10. <https://doi.org/10.1890/10-0340.1>
- Wasserstein RL, Lazar NA. 2016. [The ASA statement on p-values: Context, process, and purpose](#). The American Statistician 70:129–133.
- Wickham H. 2014. Tidy data. Journal of Statistical Software 59(10). <https://doi.org/10.18637/jss.v059.i10>
- Wickham H, Averick M, Bryan J, Chang W, McGowan LD, François R, Grolemund G, Hayes A, Henry L, Hester J *et al.* 2019. Welcome to the tidyverse. Journal of Open Source Software 4(43):1686. <https://doi.org/10.21105/joss.01686>
- Wolodzko T. 2026. [extraDistr: Additional univariate and multivariate distributions](#).
- Zuur AF, Ieno EN, Smith GM. 2007. Analysing ecological data. New York: Springer Science+Business Media.
- Zuur AF, Ieno EN, Walker NJ, Saveliev AA, Smith GM *et al.* 2009a. Mixed effects models and extensions in ecology with R. Vol. 574. New York: Springer.
- Zuur A, Ieno EN, Meesters E. 2009b. A beginner's guide to R. New York: Springer Science+Business Media.