CDSA 102-
Programming for Data Science

Certificate in Data Science & Applications

School of Vocational Studies

उत्तराखण्ड मुक्त विश्वविद्यालय

तीनपानी बाईपास रोड, ट्रांसपोर्ट नगर के पास, हल्द्वानी-263139

फोन न.- 05946 - 261122, 261123

टॉल फ्री न.- 18001804025

फैक्स न.- 05946-264232, ई-मेल- info@uou.ac.in

वेबसाइट- www.uou.ac.in

This is wrap up material for study purposes only.

Understanding Programming-

Computer Programming – A Complete Tutorial

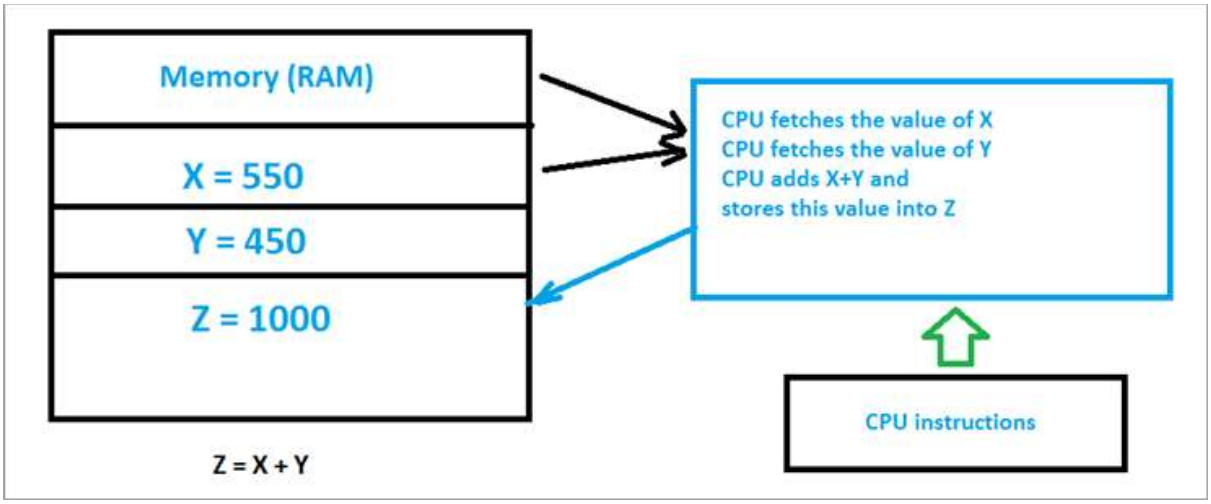
Get ready to dive deep into the world of Computer Programming and know all about the Basics of Programming in detail.

What Is Computer Programming?

Computer Programming is a set of instructions, that helps the developer to perform certain tasks that return the desired output for the valid inputs.

Given below is a Mathematical Expression.

Z = X + Y, where X, Y, and Z are the variables in a programming language. If X = 550 and Y = 450, the value of X and Y are the input values that are called literals. We ask the computer to calculate the value of X+Y, which results in Z, i.e. the expected output.



How Do Computers Work?

A computer is a machine that processes information and this information can be any data that is provided by the user through devices such as keyboards, mouse, scanners, digital cameras, joysticks, and microphones. These devices are called **Input Devices** and the information provided is called input.

The computer requires storage to store this information and the storage is called Memory.

Computer Storage or Memory is of Two Types.

- **Primary Memory or RAM (Random Access Memory)**: This is the internal storage that is used in the computers and is located on the motherboard. RAM can be accessed or modified quickly in any order or randomly. The information that is stored in RAM is lost when the computer is turned off.
- **Secondary Memory or ROM (Read-Only Memory)**: Information (data) stored in ROM is read-only, and is stored permanently. The ROM stored instruction is required to start a computer.

Processing: Operations done on this information (input data) is called Processing. The Processing of input is done in the Central Processing Unit which is popularly known as **CPU**.

Output Devices: These are the computer hardware devices that help in converting information into human-readable form. Some of the output devices include Visual Display Units (VDU) such as a Monitor, Printer, Graphics Output devices, Plotters, Speakers, etc.

A developer can analyze the problem and come up with simple steps to achieve a solution to this problem, for which he/she uses a programming algorithm. This can be compared to a recipe for a food item, where ingredients are inputs and finished delicacy is the output required by the client.

Secret Ingredients

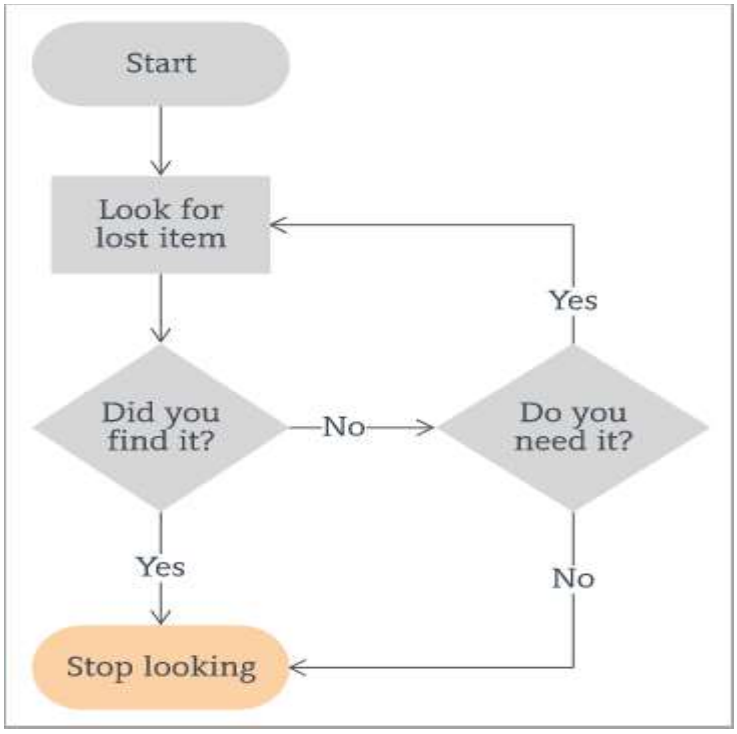
- 8 ounces feta cheese (1-1/3 cups)
- 1/4 cup green onion , minced
- 1 cup whole milk ricotta cheese
- 1/4 cup pine nuts , toasted
- 2 tablespoons brandy
- 3 tablespoons snipped fresh dill

Directions

1. In the bowl of a food processor combine the cheeses and process until smooth.
2. Add the brandy and puree.
3. Fold in the green onions, pine nuts, and dill and continue to fold until the mixture is well blended.
4. Spoon into a 2-cup decorative serving bowl.
5. Cover and chill at least 2 hours.
6. Serve with French bread, crackers or crisp breads.

The Recipe contains ingredients (inputs) and directions (steps) to prepare a food item.

In the development environment, the products, software, and solutions can be designed as scenarios, use cases, and data flow diagrams.



Simple flow chart describing the steps and flow of the solution. Based on the client's requirements, the solution required could be desktop, web or mobile-based.

Basic Programming Concepts

Developers should have essential knowledge on the following concepts to become skilled in Computer Programming,

#1) Algorithm: It is a set of steps or instruction statements to be followed to accomplish specific tasks. A developer can design his algorithm to achieve the desired output.

For Example, a recipe to cook a dessert. The algorithm describes the steps to be followed for completing a specific task, but it does not say how to achieve any of the steps.

#2) Source code: Source code is the actual text that is used to construct the program using the language of choice.

For Example, it is mandatory to have the main method in Java and the text used is as shown below.

```
public static void main(String arg[]) {
//Steps to be performed
}
```

#3) Compiler: Compiler is a software program that helps in converting the source code into binary code or byte code, also called machine language, that is easy for a computer to understand, and can be further executed using an interpreter to run the program.

#4) Data Type: Data used in the applications can be of a different type, it can be a whole number (integer), floating-point (decimal point numbers), characters or objects. **For Example,** double currency = 45.86, where double is a data type used for storing numbers with decimal points.

#5) Variable: Variable is a space holder for the value stored in the memory and this value can be used in the application. **For Example,** int age = 25, where age is a variable.

#6) Conditionals: Knowledge of how to use a certain condition, such that a set of code should execute only if a certain condition is true. In case of a false condition, the program should exit and should not continue the code further.

#7) Array: Array is the variable that stores elements of a similar data type. Knowledge of using an array in coding/programming will be a great benefit.

#8) Loop: Loop is used to execute the series of code until the condition is true. **For Example,** in Java, loops can be used as for loop, do-while, while loop or enhanced for loop.

The code for loop is as shown below:

```
for (int I =0; i&amp;amp;lt;10; i++) {System.out.println(i); }
```

#9) Function: Functions or methods are used to accomplish a task in programming, a function can take parameters and process them to get the desired output. Functions are used to reuse them whenever required at any place repeatedly.

#10) Class: Class is like a template that contains state and behavior, which corresponding to programming is field and method. In Object-Oriented languages like Java, everything revolves around Class and Object.

Essentials of a Programming Language

Just like any other language we use to communicate with others, a programming language is a special language or a set of instructions to communicate with computers. Each programming language has a set of rules (like English has grammar) to follow and it is used to implement the algorithm to produce the desired output.

Top Computer Programming Languages

The below table enlists the top Computer Programming Languages and their applications in real life.

Programming Language	Popularity	Practical Applications of Languages
Java	1	Desktop GUI application (AWT or Swing api), Applets, online shopping sites, internet banking, jar files for secured file handling, enterprise applications, mobile applications, gaming software.
C	2	Operating Systems, Embedded systems, Database management systems, Compiler, gaming and animation.
Python	3	Machine learning, Artificial Intelligence, Data analysis, face detection and image recognition Software.
C++	4	Banking and trading enterprise software, virtual machines and compilers.
Visual Basic .NET	5	Windows services, controls, control libraries, Web applications, Web services.
C#	6	Desktop applications like a file explorer, Microsoft office applications like Word, Excel, Web browsers, Adobe Photoshop.
JavaScript	7	Client side and server-side validations, DOM handling, developing web elements using jQuery (JS library).
PHP	8	Static and dynamic websites and applications, Server-side scripting.
SQL	9	Querying database, CRUD operations in database programming, creating a stored procedure, triggers, database management.
Objective– C	10	Apple's OS X, iOS operating system and APIs, Cocoa and Cocoa Touch.

The selection of particular programming languages depends on many factors such as:

- **Targeted Platform and Project/Solution Requirement:** Whenever a software solution provider comes across the requirement, there are many options to choose an appropriate programming language. **For Example,** if a

user wants the solution to be on mobile, then Java should be the preferred programming language for Android.

- **Influence of Technical Partners with the Organization:** If Oracle is a tech partner with the company, then it is agreed to implement software marketed by Oracle in the solution for every project and product developed. If Microsoft is a tech partner with the company, then ASP can be used as a development framework for building web pages.
- **Competency of available Resources & Learning Curve:** The developers (resources) should be available and competent to quickly learn the selected programming language so that they can be productive for the project.
- **Performance:** The selected language should be scalable, robust, platform-independent, secure and should be efficient in displaying results within the acceptable time limit.
- **Support from the Community:** In the case of open-source programming language, the acceptance, and popularity for the language as well as online support from the growing support group should be available.

Types Of Computer Programming Languages

Computer Programming language can be divided into two types i.e. Low-level Language, and High-level Language.

#1) Low-level Language

- Hardware dependent
- Difficult to understand

Low-level Language can be further divided into two categories,

- **Machine Language:** Machine dependent, difficult to modify or program, **For Example, every** CPU has its machine language. The code written in machine language is the instructions that the processors use.
- **Assembly Language:** Each computer's microprocessor that is responsible for arithmetic, logical and control activities need instructions for accomplishing such tasks and these instructions are in assembly language. The use of assembly language is in device drivers, low-level embedded systems, and real-time systems.

#2) High-level Language

- Independent of hardware
- Their codes are very simple and developers can read, write and debug as they are similar to English like statements.

High-level Language can be further divided into three categories.

- **Procedural Language:** Code in the procedural language is a sequential step by step procedure, that gives information like what to do and how to do. Languages such as Fortran, Cobol, Basic, C, and Pascal are a few examples of procedural language.

- **Non-procedural Language:** Code in non-procedural language specify what to do, but does not specify how to do. SQL, Prolog, LISP are a few examples of non-procedural language.
- **Object-oriented Language:** Use of objects in the programming language, where the code is used to manipulate the data. C++, Java, Ruby, and Python are a few examples of Object-oriented language.

Basic Operations of a Programming Environment

Five basic elements or operations of programming are listed below:

- **Input:** Data can be input using the keyboard, touch screen, text editor, etc. **For Example,** to book a flight, the user can enter his login credentials and then select a departure date and return date, the number of seats, starting place and destination place, Name of Airlines, etc, from desktop, laptop or mobile device.
- **Output:** Once authenticated, and upon receiving the request to book the tickets with the mandatory inputs, a confirmation of booking for the selected date and destination will be displayed on the screen, and a copy of the tickets and invoice information is sent to the user's registered email id and mobile number.
- **Arithmetic:** In case of flight booking, update of the number of seats booked and those seats need some mathematical calculations, further name of the passenger, no. of seats reserved, date of journey, journey start date, and starting place, destination place, etc. should be filled into the airlines server database system.
- **Conditional:** It is required to test if a condition is satisfied or not, based on the condition, the program may execute the function with parameters else it will not get executed.
- **Looping:** It is required to repeat /perform the task until the condition holds. **Types of loops can be While loop, Do-while loop, For loop.**

For Example,

```
for (int i = 0; i < 10; i++)
 {
System.out.println(i);
}
```

Necessary Prerequisites/Skills Required for Programming

#1) Self Reliance: To succeed in coding, you should develop a confidence in yourself, control your impatience, frustration and should refrain from being dependent on someone else to help you in solving your technical problems, rather you should be self-reliant and keep faith on your capabilities, monitor your efforts and remain optimistic and perseverant in learning.

#2) Language: It is an individual's choice to decide which programming languages he/she should learn. A programming language should be selected based on its acceptance in the various domains in software industries. Object-oriented languages like Python and Java, which are free & open-source are widely accepted and used by Google, Yahoo, and NASA.

Java script is another scripting language, a client-side scripting language, but knowing Javascript will highly benefit web-based application developers. Non-procedural language like SQL is mandatory as it is acceptable by all the back-end databases. Click this link for learning an online exercise for SQL.

#3) Logic: As a developer or tester, to excel in the programming language, one must always have conditional and logical thinking. It can be improved as we improve our muscles, there are a few sites where one can prepare and improve logical thinking and prepare for programming language.

- Fresherslive
- The Online Test Centre
- Indiabix

#4) Attention to Detail: A conscientious and alert person with an eye for details will check his/her work for minute details and this will prevent any syntax error, verify if any steps like unit testing or including API /classes, miss associated jar or class files. For some people, meditation might help to improve focus and concentration while for others taking a walk or playing some mind games might help. You need to find out what works for you.

#5) Abstract Thinking: During sprint meeting in an agile environment, the ability to think out of the box, or see things from different angles/perspectives, help to uncover scenarios for requirements and design considerations. This can be improved by a discussion with others.

#6) Patience: At times, it happens as you write a code, for which you are confident about, verified it a couple of types, it works in your machine, but after integration the code snippet does not work, all the effort to identify the fault go in vain, you feel stressed out, frustrated and feel like good for nothing.

During such times, your ability to overcome the situation, try again from scratch and develop patience will prove the developer to be more mature and he/she gets appreciated for the ability to work under pressure environments like releases and acceptance testing or during client demos.

#7) Strong Memory: Being able to understand and visualize the high-level design, data flow, algorithm, data structure, how they interact with each other will separate you from an average coder. Meditation techniques and memory exercises can help with this as well.

How To Start Learning Computer Programming?

As a human, you should have the habit to introspect daily and identify what you have done today, how can you improve yourself, what steps or precautions you will take to avoid difficult situations.

Similarly, consider the below points before learning computer programming.

- Be honest and think about why you want to learn computer programming.
- What is your goal, what will you accomplish in your dream of learning programming?
- Choose the right programming language. **E.g.** Front end programming like JavaScript, PHP, Back end programming like SQL, Java, Python for web-based development.

- Check out some interactive tutorials to get familiar with a programming language. w3schools is good to start understanding many programming languages, and w3resource is good to learn SQL queries interactively.
- Get a book on selected programming language i.e. SQL for Dummies, JavaScript for Dummies.
- Try out some online courses i.e. give a try to Udemy
- Learn Data Structures and Algorithms.
- Make a project using a selected programming language.
- Attempt some certification, and this will make you more confident, knowledgeable and competent.

Where Can We Apply the Skills of Programming?

- **Ability to Communicate:** Communication is an extremely essential quality wherein, you can explain your plan, discuss your doubts, improve your thoughts and exchange information from your superior and your team member. A good communicator can understand and explain the tasks performed in daily reporting, find out how can you improve your thoughts and clear your doubts. During the agile stand-up meeting & sprint meets, you can communicate the plan of action and can lead the team.
- **Problem-solving:** Accepting challenges and accomplishing difficult tasks will build problem-solving skills and this is a prerequisite for a good developer. During development, you may encounter various issues of understanding the business logic and implementing them into your code, integration of the code with application, compatibility issues and many more challenges. Your problem-solving skills will help you to sail through the most critical situations.
- **Collaboration/Teamwork:** Collaboration skills enable you to work with the team members to accomplish some tasks effectively and thereby improve productivity.

Working in a team at times can result in conflict, due to attitude issues. Hence, by understanding the goal to get better products or improve productivity, anyone can play the role of an excellent team player role.

Career Options for Programmers

The career options as a programmer or software developer are many.

The areas or positions for computer programmer are as follows:

- Web Developer
- UI Developer
- User Experience Designer
- SQL Developer
- Quality Assurance
- Automation Test Engineer
- Software Engineer at Test

In the Software Development department his/her responsibilities include the following duties:

- Designing and developing custom and complex solutions using various programming languages wherein he/she should be competent, **For Example, Java**, Python, JavaScript, SQL, oracle.
- Manage project software delivery lifecycle, that includes planning, design, building, testing, and deployment within the company's planned delivery framework.
- Basic knowledge in Networking, ability to work on Integrated Developer Tools such as Eclipse, NetBeans, Atom, etc.
- Should have hands-on working experience with at least one of the CI tools such as Jenkins, Gitlab, Bamboo, etc.
- Should able to use Linux / Unix scripts and shell scripting.
- Excellent communication and people skills.
- Should be a good Team player as well as an Independent Contributor.
- Understanding of agile development environment.

Introduction

R is the open-source statistical language that seems poised to "take over the world" of statistics and data science. R is really more than a statistical package - it is a language or an environment designed to potentiate statistical analysis and production of high quality graphics (for more on information see www.r-project.org/about.html).

Originally developed by two statisticians at the University of Auckland as a dialect of the S statistical language, since 1997 the development of R has been overseen by a core team of some 20 professional statisticians (for more on information see www.r-project.org/contributors.html).

Many new users find that R is initially hard to use. One needs to learn to write code, and there are no (or few) menu tools to make the process easier. In fact, when a grad school friend first excitedly described R to me in 2004 my first thought was "Why would I want to learn that?". I dabbled in R several times following that, but was put off by the difficulties I encountered. I finally began using R in earnest as an environment for some simulation modeling, and then later for statistical and graphical work.

These notes were developed for a short introduction to R for students who already understand basic statistical theory and methods, so the focus is mostly on R itself. Much of the inspiration for these notes comes from "SimpleR" [1] by John Verzani. ("SimpleR" was written to teach both R and introductory statistics together, but I successfully used it as the basis for an introduction to R for several years). As my course has evolved, I felt the need to develop my own materials, but the debt to John Verzani is substantial - many of the good bits in what follows are probably inspired by his work, particularly the didactic style, and the errors are certainly all mine.

A note about formatting in this document: To keep things clear, in this document, R output is shown in a black console (fixed width) font preceded by "#", like this:

```
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   12.00   28.50   51.50   47.90   62.75   83.00
```

[1] "SimpleR" is also still available at http://www.math.csi.cuny.edu/Statistics/R/simpleR/.

while R code is shown in a console font in colored text without the preceding #, like this:

```
summary(c(23, 45, 67, 46, 57, 23, 83, 59, 12, 64))
```

Where we mention R functions or arguments by name, they will appear in the console font, and the names of functions will be followed by parentheses, e.g. `mean()`.

You should be able to download these notes in a Zip file with associated folders. Unzip this to a convenient location and you will have an directory (folder) called "EssentialR". These notes assume that you are using the "EssentialR" directory as the working directory (see Ch 5), and examples point to files in the "Data" directory in "EssentialR".

Getting R

You can download R from CRAN (http://cran.r-project.org/). I also recommend that you then install the excellent RStudio IDE (http://www.rstudio.com/ide/download/) - while not strictly necessary, it makes working in R so much easier that it is worth using.

Other Resources

For an introduction to statistics using R (or a basic R reference), I recommend the following books:

Using R for Introductory Statistics. 2004. John Verzani. Chapman & Hall/CRC. (an extension of SimpleR)

Statistics: An introduction using R. 2005. Michael J. Crawley. Wiley and Sons. (This was useful enough to me when I began learning R that I bought a copy.).

Quick-R (http://www.statmethods.net) is a nice online overview of basic R functions and methods. Useful reference.

Gardner's own (http://www.gardenersown.co.uk/Education/Lectures/R): a nice look at using R for analysis.

R Wikibook (http://en.wikibooks.org/wiki/Statistical_Analysis:_an_Introduction_using_R): an online book for a course like this.

IcebreakeR (http://cran.r-project.org/doc/contrib/Robinson-icebreaker.pdf): another PDF book for a course like this.

Also see the "Contributed Documentation" tab at CRAN (http://cran.r-project.org/doc/contrib) for links to more resources.

The citation for R (type `citation()` to get it) is as follows:

```
R Core Team (2013). R: A language and environment for statistical
  computing. R Foundation for Statistical Computing, Vienna, Austria.
```

```
  URL http://www.R-project.org/.
```

These notes were written in Markdown, and complied using the excellent R package "bookdown" by Yihui Xie - for more information see: https://bookdown.org.

Of course, I would be remiss not to acknowledge the R core team and the other members of the R user community whose efforts have combined to make R such a powerful tool.

Figure 1: CC image

Eric Nord, Greenville IL

Dr.Eric.Nord@gmail.com

```
# [1] "August 12 2020"
```

Chapter 1

Basics

The console, the editor, and basic syntax in R

1.1 The Terminal

One of the things new users find strange about R is that all you have to interact with is a terminal (aka console) and (if you are using a GUI or IDE) an editor. This is very different from Excel or Minitab or similar applications. As we'll see, this has some advantages, but ease of learning might not be one of them. In the meantime, you might find it helpful to imagine that your *workspace* is a bit like a spreadsheet, and the terminal a bit like the "formula bar", where you have to type input to the spreadsheet, or see what is in the spreadsheet.

The R "terminal", or "console" is where commands can be entered and results are displayed. When you start R on your computer, it looks something like this:

```
R version 3.6.2 (2019-12-12) -- "Dark and Stormy Night"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

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

Type 'demo()' for some demos, 'help()' for on-line help, or
```

```
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace restored from /Users/yourname/.RData]
[History restored from /Users/yourname/.Rhistory]

>
```

Notice the ">" at the end of all this? ">" is the R prompt - it means R is waiting for you to do something. If you are using RStudio, the terminal is on the left or lower left pane. Since R is waiting for us, let's try some things. Type (or copy & paste) the following lines (one at a time) into the console. When you press "Enter" you tell R to evaluate each expression.

```
2 + 2
```

```
# [1] 4
```

```
4 * 5
```

```
# [1] 20
```

```
6^2
```

```
# [1] 36
```

```
3 + 5 * 2
```

```
# [1] 13
```

```
(3 + 5) * 2
```

```
# [1] 16
```

```
# 2+2
```

Notice a couple of things:
1. spaces don't matter - 2+2 is the same as 2 + 2
2. the standard operators +,-,*,/,and ^ all function as expected
3. standard order of operations is respected, and can be altered using parentheses
4. all answers are preceded by [1] - we'll see why in a bit
5. an expression preceded by "#" is not evaluated – "#" is the *comment character* in R. Anything between a "#" and the end of a line is treated as a comment and is not evaluated.

A common place new useRs get confused is when the console displays a + rather than the prompt >. This simply means that the last expression entered was incomplete. Try entering 2+ and see what happens. You need to complete the addition before you get a prompt again. You will almost certainly experience this later, as unmatched (will lead to a +.

Let's create a variable (also called an *object* in R), "a" and give it a value of 5.

```
a = 5  # =
```

Here we used the `=` to *assign* the value `5` to `a`. Those of you with some programming experience may wonder about data types. R can handle many types of data (including numerical, character, logical, factors, matrices, arrays, and lists), and we'll discuss them in more detail later. Now we have created the object a, we can see what it is:

```
a
```

```
# [1] 5
```

Notice that the value of `a`, `5`, is returned. We can also calculate with an object:

```
a * 2
```

```
# [1] 10
```

```
a
```

```
# [1] 5
```

```
a = a * 2
```

Notice that when we tell R `a*2` we get the result, but then when we type `a` we can see `a` is not changed. To *change an object* in R we must *assign* it a new value. What do you think we will get if we type `a` now? What if we type `A`? Remember that R is *case sensitive*! (We've just highlighted two of more common errors I see and make with R: 1. forgetting to assign some output that I intended to save to an object and 2. case errors.)

A note about `=` and `<-`. Assignment in R can be done using `=` or using the *assignment operator*: `<-` or `->`. The assignment operator (`<-`) is directional, and the leftward assign is *far* more common than the right. In the case of `=` the "name" is assumed to be on the left - i.e. `=` is equivalent to `<-`. In these notes I generally use `<-`, so that `=` is reserved for passing arguments to functions. (The keyboard shortcut for `<-` in RStudio is "alt+-".)

When you create an object in R it exists in your *environment* or *workspace*. You can see what is in your workspace by typing `ls()`, which is short for "list". If you are using RStudio, you can also go to the "Environment" tab which is usually in the upper right pane.

You can remove things from your workspace using `rm()` - for example `rm(a)` will remove `a` from your workspace. Try `ls()` now. If you type `a` you'll get an error: `Error: object 'a' not found`. This makes sense, since we've removed (deleted) `a`. [1]

A note about the *console* and the *editor*. In RStudio you can go to the `File` menu

[1] Object names in R: In brief they consist of letters, numbers, and the dot and underscore characters. Names must begin with a letter or a dot followed by a letter. The dot has no special meaning in object names in R.

and choose `New>R script` and new pane will open above your console. This is your "Editor", and you can actually have multiple files open here (as tabs). An *R script* file contains R code and comments (see above re: comments) , but not output (though you can paste output into a script file, but you should put it behind comments so R doesn't try to run it). You can easily run code from the editor. Open an new script file and type `# editor demo`. Now press Ctrl+Enter, and R will run that command (since it is a comment, it will just be written to the console). Now type `a\*4` and press Ctrl+Enter (Mac: CMD+Return).

Scripts can easily be saved [2] for future reference or to continue your work later. The console *can't* easily be saved, but it contains the output from your R code. (There is a logic to this - if you have all the code saved, the console output is easily recreated - just run all the commands.) In the console the *up arrow* will retrieve previous commands, which can save you a fair amount of typing (often it is faster to edit a previous command than to type it over).

1.2 Working with Vectors

One of the ways R makes working with data easy is that *R natively handles vectors*, meaning that (almost) *anything you can do to a value in R you can do to a vector of values.* As we'll see, this becomes very useful.

Imagine that I count the number of SMS text messages I receive each day for a week and enter them in R as `sms`:

```
sms <- c(0, 1, 2, 0, 0, 0, 1)
```

Notice:
1. I get very few SMS messages - I like it this way!
2. We use a new *function* (command) here -`c`- it *concatenates* values together into a vector.
3. The function `c` is followed by parentheses `()`. All functions in R are followed by parentheses that contain the *arguments* to the function, like this: `function(argument1,argument2)`.
4. Within the parentheses the different values are separated by commas (`,`) - this is also standard in R - the arguments to a function are separated by commas, and here the values are the arguments to the function `c()`.

R can do many things easily with vectors: mathematical operations, sorting, and sub-setting.

```
sms + 5
```

```
# [1] 5 6 7 5 5 5 6
```

[2] A problem that will prevent saving files from the editor: failure to extract the Essential R folder. In a few cases Windows users have been unable to save anything into the Essential R folder they downloaded. This has always been caused by failure to actually extract ("unzip") the file to create a folder in the user's home folder on the hard-drive of the computer.

```
sms * 5
```

```
# [1]  0  5 10  0  0  0  5
```

```
sms/2
```

```
# [1] 0.0 0.5 1.0 0.0 0.0 0.0 0.5
```

```
sort(sms)
```

```
# [1] 0 0 0 0 1 1 2
```

Notice that if you type `sms` the original data has not been changed - to change the original data you **always need to assign the result** [3]. This is design principle in R - a function should *never change anything in your workspace*; objects in the workspace are only changed by assignment. If you want a function to change an object you must *assign the result* of the function to that object.

1.3 Sub-setting Vectors - the magic "[]"

It is often the case when working with data that we want to select only specific parts of the data (think "filtering" in Excel). In R we do this by "sub-setting" vectors. In R the square brackets `[]` are used for *indexing* vectors, matrices, data tables, and arrays. Here we'll just consider vectors. The more you gain familiarity with R, the more you learn the power of the `[]`.

```
sms[3]
```

```
# [1] 2
```

```
sms[2:4]
```

```
# [1] 1 2 0
```

```
sms[-3]
```

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

```
sms[-(2:3)]
```

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

```
sms[-c(2, 3, 7)]
```

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

Here we've told R to give us the 3rd element of sms and the 2nd-4th elements of sms - the `:` symbol means "through" - you can test this by typing `5:9`:

```
5:9
```

[3] This is a basic point that is very important, and often forgotten.

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

A minus sign "-" in the `[]`, `-` means "all elements except", and this can be used with a range (`2:3`) or a more complex list (`c(2,3,7)`), though it will be necessary to add parentheses as we have done here. This is because `-2:3` returns `-2 -1 0 1 2 3`, and the `-2th` element of a vector does not make sense!

In fact, a more general and useful approach is *logical extraction* - selecting parts of a vector based on some logical test. For example, if we wanted to know the average (mean) number of sms messages I received *on days that I received any sms messages*, we could do this easily with logical extraction. `sms>0` applies the *logical test* `>0` to `sms` and returns a *logical vector* (`TRUE` or `FALSE` for each element of the vector) of the result. This can be used to specify which elements of the vector we want:

```
sms > 0
```

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

```
sms[sms > 0]
```

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

It is worth noting that R treats logical values as `FALSE`=0 and `TRUE`=1. Thus `sum(sms)` will give us the total number of sms messages, but `sum(sms>0)` will give us the number of `TRUE` values in the logical vector created by the logical test `sms>0`. This is equivalent to the number of days where I received any sms messages.

```
sum(sms > 0)
```

```
# [1] 3
```

We can also use the function `which()` for logical extraction, and we can then use the function `mean()` to get the average number of sms messages on days where I received messages:

```
which(sms > 0)
```

```
# [1] 2 3 7
```

```
sms[which(sms > 0)]
```

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

```
mean(sms[which(sms > 0)])
```

```
# [1] 1.333333
```

Notice:
1. The function `which()` returns a *vector of indices* rather than a logical vector.
2. The final line here shows something that is very typical in R - we used one function, `which()` as an argument to another function, `mean()`. This type of

programming can be confusing as there are square brackets and parentheses all over the place. I find that I need to build it up step by step, as you can see we've done here.
3. It is a good practice to develop the skill of reading such a statement from the inside out rather than left to right. `mean(sms[which(sms>0)])` could be read as "use only the values of sms that are greater than 0 and take the mean". 4. A small command like this already has 3 nested sets of parentheses and brackets - it is really easy to drop one by mistake [4]. If you entered `mean(sms[which(sms>0)]` (omitting the closing "`)`") R will notice that your command is incomplete, and instead of a result and a new line with the prompt (`>`) it will give no answer (as the command is not complete) and will give a `+` - this just means that R is waiting for the command to be complete.

You can also use the indexing operator to *change part of vector*:

```
sms
```

```
# [1] 0 1 2 0 0 0 1
```

```
sms[1] <- 1
sms
```

```
# [1] 1 1 2 0 0 0 1
```

This approach could be combined with logical extraction; for example if you wanted to replace all the zero values with 1: `sms[which(sms==0)]<-1`. Notice that *when making a logical test R expects the double equal sign* `==`; this is to differentiate between the assignment or argument use of the equal sign `=` and the logical use `==`. This is a common source of errors.

Recall the mysterious `[1]` that is returned with our results? This just means the first element of the result is shown. If your vector is long enough to go to a second (or third) line, each line will begin showing the element number that begins the line:

```
1:50
```

```
#  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
# [21] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
# [41] 41 42 43 44 45 46 47 48 49 50
```

Notice that each line begins with the index within square brackets (`[]`) for that element.

It is also worth noting that you can use one variable to index another. First we'll make some vectors.

[4] Tip: Many text editing programs (including the editor in RStudio) will balance parentheses and brackets - when you type `(` the matching `)` is added, and the cursor is placed within them. Also, when you place your cursor on a parenthesis or bracket the matching on will be highlighted. If your text editor does not do this, find one that does - it will save you many headaches.

```
x <- c("a", "b", "c", "d", "e", "f", "g", "h", "i")
y <- 21:30
z <- c(2, 4, 6)
```

Now we can try various uses of `[]`:

```
x[y]
```

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

What is going on here? What does the `NA` mean?

```
x[z]
```

```
# [1] "b" "d" "f"
```

The `z-th` element of `x`.

```
y[z]
```

```
# [1] 22 24 26
```

```
z[y]
```

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

The second line here gives many NA's since `z` doesn't have enough elements to match some values of `y`.

```
x[rev(z)]
```

```
# [1] "f" "d" "b"
```

```
y[x]
```

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

`rev()` just reverses its argument. The `x-th` elements of `y` fails because `x` can't be "coerced"" to numeric.

```
y * z
```

```
# Warning in y * z: longer object length is not a multiple of
# shorter object length
```

```
#  [1]  42  88 138  48 100 156  54 112 174  60
```

This warning is important. One might naively assume that R won't do the element-wise addition or multiplication of two vectors of different lengths, but it *does* by *recycling* the shorter vector as necessary, with a warning if the length of longer one is not a multiple of the length of the shorter. *Pay attention* to your vector lengths!

Earlier we learned how we can change elements of a vector using the indexing operator. (e.g. `sms[5]<-3`. There are other tools in R that allow us to edit data. The most useful in RStudio is `edit()`. For example, `edit(sms)` will bring up a small edit window in R studio. If you change the first element from 1 back to zero and press save, the console will show the edited value of `sms`.

However if you type `sms` again, you'll find that the original data hasn't been altered. As we said before, if you want to change values in R *you have to assign the result of you change* [5] -so `sms<-edit(sms)` would actually save the changes. Of course, you might want to make a copy of the data and change that - simple, just assign the result of edit to another name: `sms.2<-edit(sms)`.

The R functions `de()` and `data.entry()` are similar to `edit()`, but aren't supported in RStudio, so we'll skip them here. It is worth noting that these functions (`edit()`,`de()`,and `data.entry()`) are *interactive* - that is they require user input outside of the terminal. There are 2 important things to know about interactive functions in R:
1. When you invoke an interactive function, *R waits for you to be done with it* before R will continue. In RStudio you will see a small red "stop sign" icon just above the console pane. This means R is waiting for you do do something. (The first time this happened to me I thought R had frozen and I forced R to quit and restart).
2. When you use an interactive function any changes you make to data this way are *not recorded in your code* - if you make your changes to your data via the code you write, then saving your code *preserves a record of what you have done.* If you are like me, and have ever found yourself looking at a spreadsheet that you (or someone else) made some time ago, and wondering "what was happening here?", you will see the benefit of having everything you do recorded in your code. It increases the *transparency* of your analysis. [6].

1.4 Other Useful Functions

Some other useful functions for working with vectors of data are:

Function	Description
`sd()`	standard deviation
`median()`	median
`max()`	maximum
`min()`	minimum
`range()`	maximum and minimum
`length()`	number of elements of a vector
`cummin()`	cumulative min (or max `cummax()`)

[5] This is not unique to `edit()`. R will not change something in your environment unless you explicitly tell it to do so. This is actually a good thing if you think about it.

[6] Note that interactive functions like this also will cause problems after we learn how to compile documents in R (Chapter 4).

Function	Description
`diff()`	differences between successive elements of a vector

A couple of additional hints that you'll find helpful:
1. If you type some function in R and press `Return`, but R gives you `+` instead of the answer, it means you have not completed something - most often this means a parenthesis has not been closed.
2. Related to 1.: in RStudio when you type "(" the ")" is automatically placed - this is to help you keep track of your "()"s. Also when your cursor is beside a "(" the matching ")" is highlighted, and vice-versa.
3. You will occasionally see a semicolon (`;`) used - this simply allows two functions to be submitted on one line. This is generally rather rare, since R functions tend to be longer rather than shorter when doing anything complex. 4. The comment character in R is the hash (`#`) - anything between `#` and the end of the line is treated as a comment and is not evaluated. Adding comments in your R code is a good idea - it helps you remember where you are and what you are doing (transparency!)

1.5 A Comment about R Snytax

We can extend the "R as a language" idea introduced in the syllabus in thinking about R syntax. The generic form of an sentence is `noun+verb`. In R that would be rendered as `verb(noun)` - in R the *function* is the *action*, or the *verb*. So we can take a sentence like `Bring me the sandwich` and render it in R as something like this: `bring(to=me,what=sandwich)`. Of course sometimes sentences (and R expressions) are more complicated, such as: `Bring me the sandwich that I left on the counter` which we might render in R as something like this: `bring(to=me,what=left(what=sandwich,where="on the counter"))` [7].

We can reverse this by starting with an R expression, such as `mean (sms [which] (sms > 0))]` (used above). This would translate as `Average the values of "sms" which are greater than zero`. This is kind of a messy example because the second verb ("is", in the comparison "are greater than") may not be immediately obvious.

1.6 Loops in R.

One of the great advantages of R over a point-and-click type analysis tool is that it is so easy to automate repeated tasks. Because R typically handles vectors quite nicely, many (most?) loops can be avoided. For example, there is no need to write a loop to add a two vectors. In addition, code that uses vectorization is

[7] All this talk about sandwiches requires a link to [this cartoon] (https://xkcd.com/149/).

usually much faster than code that uses loops. But sometimes a loop is useful, so we'll consider them here.

Loops are initiated with the function `for()`, with an index a the argument. The loop is usually enclosed in curly braces "{}", but if it all fits on one line it doesn't need the braces.

```
for (i in 1:27) {
    cat(letters[i])
}
```

```
# abcdefghijklmnopqrstuvwxyzNA
```

```
for (i in 1:10) {
    print(paste(i, "squared =", i^2))
}
```

```
# [1] "1 squared = 1"
# [1] "2 squared = 4"
# [1] "3 squared = 9"
# [1] "4 squared = 16"
# [1] "5 squared = 25"
# [1] "6 squared = 36"
# [1] "7 squared = 49"
# [1] "8 squared = 64"
# [1] "9 squared = 81"
# [1] "10 squared = 100"
```

```
for (i in sample(x = 1:26, size = 14)) cat(LETTERS[i])
```

```
# ODKYIVFWTLRQCN
```

These examples are ridiculously trivial, because I want to emphasize the loop syntax, which could be rendered as: `for (every value in the index) {do something with that value}`.

Notice - these loops could all be avoided by passing *vectors* rather than single values to functions - the following code produces (mostly) the same results (not run here, but you should run it to confirm).

```
paste(letters[1:27], collapse = "")  # first example,
# see what happens if paste() is not used
paste(1:10, "squared =", (1:10)^2)  # second example
sample(LETTERS, size = 14)  # third example
```

Also note that `letters` and `LETTERS` are built in. Also note (in the following) that the index does not need to be a number - it can be an object or something that evaluates to a number. Lastly, note that values aren't written to the console when a loop is run unless we specify that they should be using a function such as `cat()` or `print()`.

```
x <- c("Most", "loops", "in", "R", "can", "be", "avoided")
for (i in x) {
    cat(paste(i, "!"))
}
```

```
# Most !loops !in !R !can !be !avoided !
```

A couple of final thoughts on loops: 1) This section is probably of more interest to readers who have had some programming experience in the past.

2) The `apply()` family of functions (See Chapter 10) can be used in place of many loops. We won't explore them now.

3) Some useRs are "hardliners" about avoiding loops and would probably argue that I should not even include this material in chapter 1. I don't view loops as inherently bad, but I try to avoid them when I can in the spirit of "how can I learn to take advantage of more of R's built in tool set". Vectorized R code is usually substantially faster, so there is some advantage to avoiding loops.

1.7 Exercises.

1) Enter the following data in R and give it a name (`P1` for example):
`23,45,67,46,57,23,83,59,12,64`
a) What is the maximum value? b) What is the minimum value? c) What is the mean value?

2) Oh no! The next to last (9th) value was mistyped - it should be 42.
a) Change the 12 to 42. b) How does this change the mean? c) How many values are greater than 40? d) What is the mean of values over 40? *Hint*: how do you see the 9th element of the vector `P1`?

3) Using the data from problem **2** (after the 9th value was changed) find:
a) the sum of `P1`
b) the mean (using the sum and `length(P1)`)
c) the log(base10) of `P1` - use `log(base=10)`
d) the difference between each element of `P1` and the mean of `P1` (Note: you should not need a loop for this problem)

4) If we have two vectors, `a<-11:20` and `b<-c(2,4,6,8)` predict (*without running the code*) the outcome of the following (Write out your predictions as comments in your HW. After you make your predictions, you can check yourself by running the code). Were you correct? a) `a*2`
b) `a[b]`
c) `b[a]`
d) `c(a,b)`
e) `a+b`

Chapter 2

Qualitative Variables

Creating and using categorical variables in R

2.1 Introduction

In the last chapter we saw how we can work with vectors in R. Data that we work with in R is generally stored as vectors (though these vectors are usually part of a *data frame*, which we'll discuss in the next chapter). Of course there are several types of data that we might need to work with - minimally we need qualitative data (categorical data - *factors* in R-speak) and quantitative data (continuous or numerical data - *numeric* in R-speak), but strings (*character* in R) are often useful also. Today we'll consider the first two of these data types, and learn how to work with them in R.

Since there are different types of data, it is very important to know:
A. What type of data you have (by this I mean what it *represents*).

As our focus here is R, I won't dwell on this except to say that it is worth taking time to be clear about this when designing your data. For example, if four replicates are labelled "1, 2, 3, 4", then R is likely to treat replicate is a numerical variable. If they are labelled "A, B, C, D", or "I, II, III, IV", this can be avoided.

B. What R thinks the data is (or how it is *encoded* in R).

The most common data types for vectors in R are: `"logical"`,`"integer"`,`"double"`, and `"character"`. There are several other types that you may never encounter and several types that apply to more complex structures that we'll explore later.

There are a couple of ways to find out how R is storing data. The function `str()` ("structure") will give you the basic data type. The function `summary()` gives summary statistics for numeric variables, but number of levels for factors.

This works well because it also lets you quickly see if you have miss-coded data (e.g typos like "IIi" in place of "III") or extreme outliers.

C. How to ensure that the answers to **A.** and **B.** are the same!

2.2 The Function `factor()`.

Qualitative data, or categorical data is stored in R as *factors*. We can use the function `factor()` to *coerce* (convert) a vector to a factor, as demonstrated here:

```
cols <- c("Blue", "Blue", "Red", "Red", "Blue", "Yellow", "Green")
summary(cols)
```

```
#    Length     Class      Mode
#         7 character character
```

```
cols[2]
```

```
# [1] "Blue"
```

```
cols <- factor(cols)
cols[2]
```

```
# [1] Blue
# Levels: Blue Green Red Yellow
```

Notice that this factor was created as a character variable - the elements still have quotes around them. After we convert it to a factor, even returning one element (`cols[2]`) we can see that there are no quotes and we get the levels reported. The structure (`str(cols)`) reports that it is factor, and shows us the numeric representation of it (we'll discuss this more in a bit). The summary (`summary(cols)`) shows us the frequency for (some of) the levels [1]. Now that we have `cols` as a factor, we can investigate its properties. The function `levels()` shows us all the levels for a factor.

```
str(cols)
```

```
#  Factor w/ 4 levels "Blue","Green",..: 1 1 3 3 1 4 2
```

```
summary(cols)
```

```
#   Blue  Green    Red Yellow
#      3      1      2      1
```

```
levels(cols)
```

```
# [1] "Blue"   "Green"  "Red"    "Yellow"
```

We can use the function `table()` to see a frequency table for our factor. Note: We can use `table()` on character or numeric vectors also - `table()` will coerce

[1] If there were many levels, only the first 5 or 6 would be shown.

its argument(s) to factor if possible (though of course it doesn't *store* the factor - objects are only stored if you explicitly call for them to be stored).

```
table(cols)
```

```
# cols
#   Blue  Green    Red Yellow
#      3      1      2      1
```

```
b <- table(cols)
b[3]
```

```
# Red
#   2
```

```
b[3] * 4
```

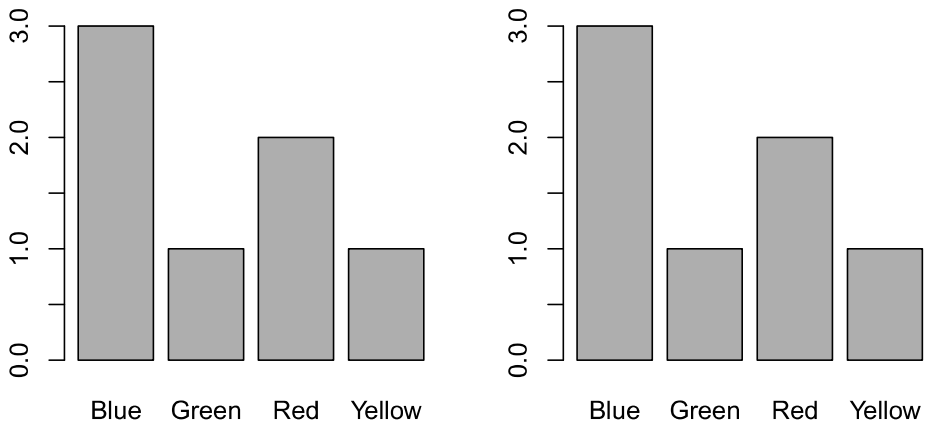
```
# Red
#   8
```

Notice that the frequency table created by `table()` is itself an R *object*, meaning that we can assign it to another name (`b` in this example), and we can access parts of it in the normal way, and use it in further calculations. Using functions to return or store (save) objects is a very common task in R, as you will see, and *many functions return objects.*

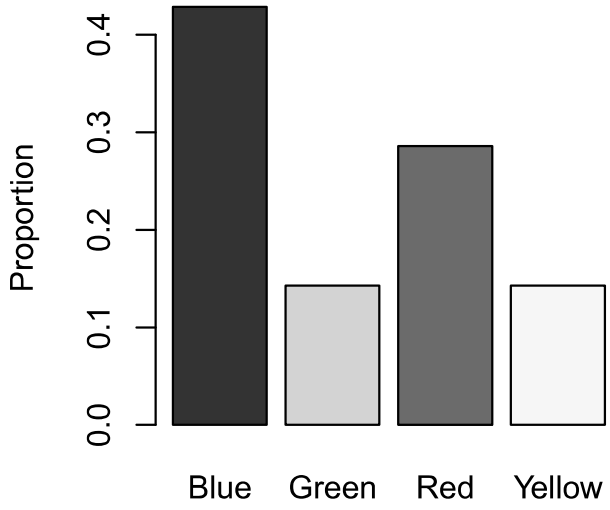
2.3 Visualizing Qualitative Variables.

If we want a graphical summary of a factor, we can make a barplot (`barplot()`). However, we need to use `table()` with `barplot()`, since `barplot()` requires the values it will plot (its `height` argument) in a numeric form (i.e. a vector or a matrix; see `?barplot` for more detail).

```
barplot(table(cols))
plot(cols)
```

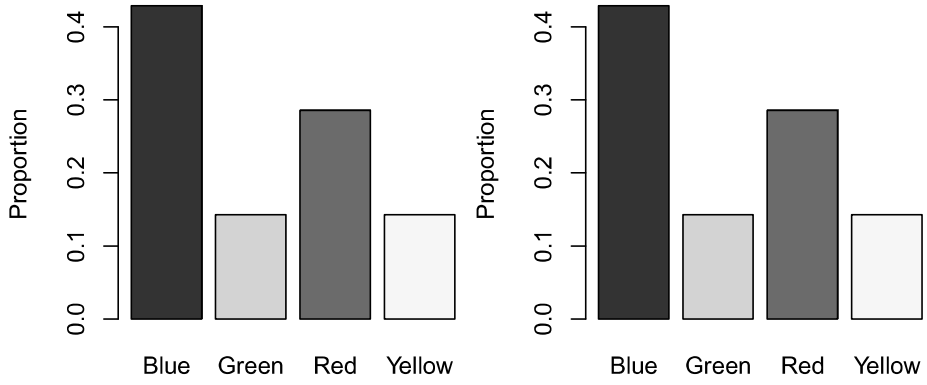


```
barplot(b/length(cols), col = c("blue", "green", "red", "yellow"),
    ylab = "Proportion")
```



Note that `plot(cols)` gives a barplot in the second plot - this is because the data is categorical. In the third plot we used `b` (recall that earlier we assigned `table(cols)` to `b`) either way works.

```
barplot(b/length(cols), col = c("blue", "green", "red", "yellow"),
    ylab = "Proportion")
barplot(table(cols)/length(cols), col = levels(cols), ylab = "Proportion")
```



The first plot here demonstrates how we can easily add color to plots and that we can carry out calculations within the call to `barplot()` (e.g. to calculate proportions). We can also specify the y-axis label (the argument is `ylab`). The second plot demonstrates how we can use the output of `levels()` to specify our colors. This only makes sense in a minority of cases, but it is an example of nesting functions - `table()`, `length()`, and `levels()` are all used to supply arguments to `barplot()`.

> Notice that the `col` argument to `barplot()` is *optional* - barplot works fine if we don't specify `col`, but we have the option to do so if need be. This is a common feature of R functions - minimal arguments are *required*, but there are often many optional arguments, which often have well chosen default values.

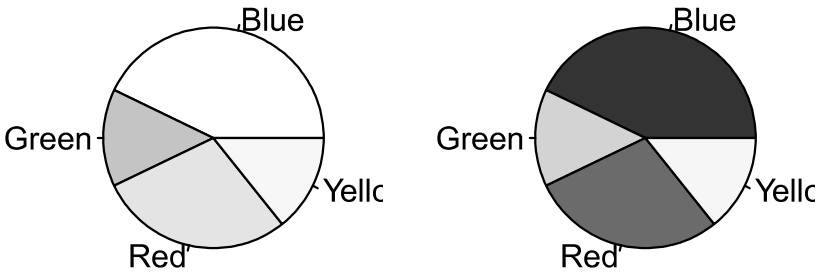
We can also convert a factor to a logical vector (by using a logical test) should we need to for sub-setting:

```
cols == "Red"
```

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

We can also create a pie chart quite easily (though see `?pie` for why this might not be a good idea).

```
pie(table(cols))
pie(b, col = c("blue", "green", "red", "yellow"))
```



Another example, this time with a numerically coded factor.

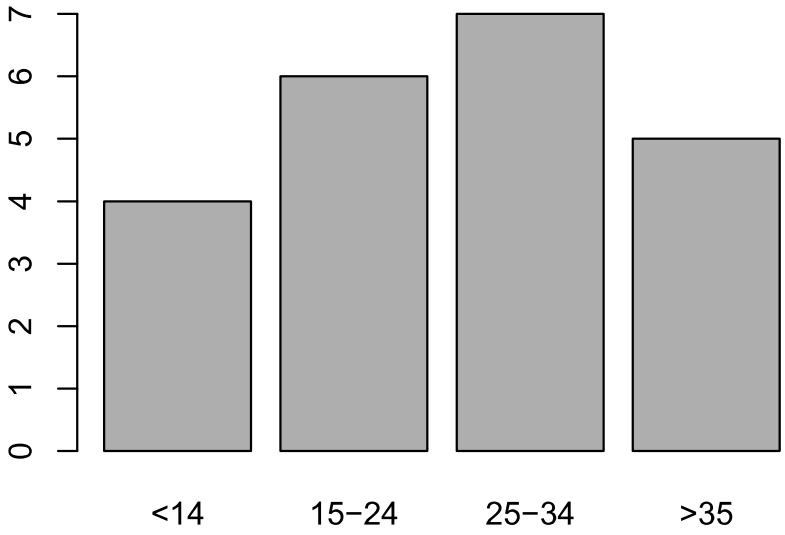
```
a <- factor(scan(text = "2 4 3 3 2 1 1 2 3 4 2 3 3 4 1 3 2 1 4
               3 2 4"))
# scan(text='some text string with spaces separating value')
table(a)
```

```
# a
# 1 2 3 4
# 4 6 7 5
```

```
levels(a) <- c("<14", "15-24", "25-34", ">35")
table(a)
```

```
# a
#   <14 15-24 25-34   >35
#     4     6     7     5
```

```
barplot(table(a))
```



First notice that we introduce the function `scan()` here - when entering a longer list like this it may be easier than using `c()`, as commas don't need to be entered. Second note that we can use the function `levels()` to *set* levels as well as to return them (a number of other R functions display this dual utility as well). For a longer factor variable, it might be faster to enter it this way than by repeatedly entering all the factor levels [2] .

2.4 How Factors are Stored in R

R stores factors as a list of levels and an integer vector representing the level of each element of the factor. So our factor `a`, with values `1,1,4,5` has three levels: `1 4 5`.

```
a <- c(1, 1, 4, 5)
str(a)
```

```
#  num [1:4] 1 1 4 5
```

```
(a <- as.factor(a))
```

```
# [1] 1 1 4 5
# Levels: 1 4 5
```

```
str(a)
```

```
#  Factor w/ 3 levels "1","4","5": 1 1 2 3
```

```
levels(a)
```

[2] In practice I enter rather little data when using R for analysis - mostly I import the data, as we'll see later.

```
# [1] "1" "4" "5"
```

Notice that `str(a)` shows that the original values have been replaced by the level numbers, which are `1,2,3`. This can create an unwelcome surprise if you are trying to use values from a factor variable in a calculation! For this reason, it is probably best to avoid using the integer values from a factor in calculations. Note that while factors levels are stored as a vector of numbers, it does not make sense to treat these as numbers - in this example the level "4" is represented by the value "2". If our levels were "blue","red","green", it clearly makes no sense to assume that because "blue" is level 1, and "green" is level 2 that "green" = twice "blue".

Note that in the third line we put parentheses around the assignment - this is equivalent to `a<-as.factor(a);a` - it both carries out the assignment and shows us the new value of `a`. This is occasionally useful. The function `levels()` returns the levels for a factor (what do you think it does for a non-factor variable?)

We already saw how we can use `factor()` and `as.factor()` will convert character or numeric data to factor data, and `as.numeric()` will (sort of) do the opposite.

```
as.numeric(a)
```

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

As you can see in this case we don't get the original values, we get the integer representation. We can also see this in the output from `str(a)`. If necessary, this can be solved by first converting to character and then to numeric [3] - `as.numeric(as.character(a))` returns: 1, 1, 4, 5.

```
as.numeric(as.character(a))
```

```
# [1] 1 1 4 5
```

Since factor levels are characters, they can be a mix of alphabetic and numeric characters, but that will clearly cause problems if we want to coerce the factor to a numeric vector.

```
(b <- c("-.1", " 2.7 ", "B"))
```

```
# [1] "-.1"   " 2.7 " "B"
```

```
str(b)
```

```
#  chr [1:3] "-.1" " 2.7 " "B"
```

```
as.numeric(b)
```

```
# Warning: NAs introduced by coercion
# [1] -0.1  2.7   NA
```

[3] Note that this is not the most computationally efficient way to accomplish this, but is the easiest to remember. The recommended approach is `as.numeric(levels(a)[a])`, which showcases the power of the `[]` in R.

Here we entered the values with quotes, which created a character variable, (as shown by the `chr` returned by `str()`). When we convert the vector to numeric, the non-numeric value ("B"), can't be coerced to a number, so it is replaced by `NA`, thus the warning `NAs introduced by coercion`. This warning will occasionally show up when a function coerces one of its arguments.

2.5 Changing Factor Levels

Occasionally we need to change the levels of a factor - either to collapse groups together or to correct typos in the data.

```
cols <- factor(c("Blue", "Blue", "Red", "Red", "Bleu", "Yellow", "Green"))
levels(cols)
```

```
# [1] "Bleu"   "Blue"   "Green"  "Red"    "Yellow"
```

Here we have mistyped "Blue" as "Bleu" (I do this kind of thing all the time). We can use the function `levels()` to *set* levels as well as to query them. The key is that since there are currently 5 levels, we must specify 5 levels that correspond to the 5 current levels, but we don't have to specify *unique* levels.

```
levels(cols) <- c("B", "B", "G", "R", "Y")
levels(cols)
```

```
# [1] "B" "G" "R" "Y"
```

Now we have only four levels - by assigning "B" to both "Blue" and "Bleu" we collapsed them together. This is not reversible - if we wanted to get "Bleu" back, we'd have to reload the data. This is where making a copy of the vector before you start tweaking it may be a good idea (though if you are writing all your code in an R script, it is quite easy to get back to where you started - just run all the commands again).

Note that the new levels can be the same as the old levels - in the example above I avoided that just for clarity.

```
cols <- factor(c("Blue", "Blue", "Red", "Red", "Bleu", "Yellow", "Green"))
levels(cols) <- c("Blue", "Blue", "Green", "Red", "Yellow")
levels(cols)
```

```
# [1] "Blue"   "Green"  "Red"    "Yellow"
```

In fact we can supply the same levels in a different order, though that probably doesn't make much sense.

```
cols <- factor(c("Blue", "Blue", "Red", "Red", "Blue", "Yellow", "Green"))
levels(cols)
```

```
# [1] "Blue"   "Green"  "Red"    "Yellow"
```

```
levels(cols) <- c("Yellow", "Blue", "Green", "Red")
levels(cols)
```

```
# [1] "Yellow" "Blue"   "Green"  "Red"
```

Since there are four levels, we must supply a vector of four levels, but they don't need to all be different:

> Note: We could also use the function `replace()` to change factor levels, but to do this we first have to convert the factor to character using `as.character()`, so this method is not generally as useful.

It is worth noting that once a level is created for a factor, the level persists even if the corresponding data is removed.

```
cols
```

```
# [1] Yellow Yellow Green  Green  Yellow Red    Blue
# Levels: Yellow Blue Green Red
```

```
cols[-6]
```

```
# [1] Yellow Yellow Green  Green  Yellow Blue
# Levels: Yellow Blue Green Red
```

Notice the level `Red` is still present, even though there are no values with that level. This is occasionally annoying. The function `droplevels()` can be used to drop unused factor levels you can check this by running `droplevels(cols[-6])`.

Occasionally we want to impose our own order on a factor rather than accept the default (alphanumeric) order given by R. For example, a factor with levels "Low", "Medium" and "High" would be default ordered as "High", "Low", "Medium", which doesn't make sense. We can use the use the `levels` argument of `factor()` to set the orders as we'd like them to be.

```
x <- factor(c("L", "M", "H"))
y <- factor(x, levels = c("L", "M", "H"))
x
```

```
# [1] L M H
# Levels: H L M
```

```
y
```

```
# [1] L M H
# Levels: L M H
```

Notice here that the *content* of `x` and `y` are the same ("L", "M", "H"), when returned by R. However, if you run `str(x);str(y)`, you will see that the underlying encoding is different because the order of levels is different.

In some cases, your interest in the order of factor levels may be for convenience, and not based on an intrinsic order (e.g. ordering categories based on total

frequency, such as might be utilized in a bar plot, or ordering universities on the basis of the number of R users). The method presented above is appropriate for these situations.

In other cases, there is an intrinsic order to the levels (e.g. "Low", "Med", "High"; "Never", "Sometimes", "Always"). In such instances it may be useful to create an *ordered* factor.

```
z <- factor(x, levels = c("L", "M", "H"), ordered = TRUE)
z
```

```
# [1] L M H
# Levels: L < M < H
```

Notice that the levels are listed from lowest to highest, and are shown with the "<" indicating the order. We can also see this when we call `str()` on an ordered factor. Also notice that when we create the ordered factor `z` we begin with `x` that already has the levels `"L","M","H"`. If `x` had other levels we'd need to set the correct levels first via `levels()`, or use the argument `labels` for `factor()`.

```
str(y)
```

```
#  Factor w/ 3 levels "L","M","H": 1 2 3
```

```
str(z)
```

```
#  Ord.factor w/ 3 levels "L"<"M"<"H": 1 2 3
```

The main advantage of an ordered factor is the ability to perform logical tests that depend on ordering on the factor.

```
y > "L"
```

```
# Warning in Ops.factor(y, "L"): '>' not meaningful for factors
# [1] NA NA NA
```

```
z > "L"
```

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

```
sum(z <= "M")  # '<=' is 'less than or equal to'
```

```
# [1] 2
```

2.6 Hypothesis Testing for Factors

We may want to test hypotheses about a qualitative variable. For example, if we roll a die 50 times and get "6" 12 times how likely is it that the die is fair? (This really is a factor - it just has numeric levels.)

We can use the proportion test in R (`prop.test()`) to compare an observed frequency against a hypothesized frequency and calculate a p-value for the

difference. Here our observed frequency is 12 out of 50, and the theoretical probability is 1/6. Our alternative hypothesis is that the probability is greater than 1/6.

```
prop.test(x = 12, n = 50, p = 1/6, alt = "greater")
```

```
#
#  1-sample proportions test with continuity correction
#
# data:  12 out of 50, null probability 1/6
# X-squared = 1.444, df = 1, p-value = 0.1147
# alternative hypothesis: true p is greater than 0.1666667
# 95 percent confidence interval:
#  0.1475106 1.0000000
# sample estimates:
#    p
# 0.24
```

The p-value here is 0.115, so we don't have very strong evidence of an unfair die.

EXTRA: Simulating a hypothesis test for a qualitative variable

Another way to approach this question is to simulate the problem in R. The function `sample()` will randomly choose values, so `sample(1:6)` would be like rolling a fair die, and `sample(1:6,size=50,replace=TRUE)` like rolling the die 50 times. Adding the logical test `==6` asks how many 6's come up, and calling `sum()` on the logical test adds up the number of `TRUE`s (recall from Chapter 1 that logical values can be interpreted as 0 or 1).

```
sample(x = 1:6, size = 50, replace = TRUE)  # rolling a die 50 times
```

```
#  [1] 3 1 1 1 2 5 6 6 1 5 5 4 1 6 4 5 2 5 5 5 3 3 5 3 2 5 2 2 3 1
# [31] 4 1 6 6 1 4 3 4 6 6 2 1 3 3 2 3 4 2 6 5
```

```
sum(sample(1:6, 50, TRUE) == 6)  # how many times is it 6?
```

```
# [1] 8
```

You can easily use the up arrow in the console to repeat this - you'll see that the number of 6's varies. If we repeated this 100 times we could see how frequent a value of 12 or greater is. To do this we'll use a *loop*. First we'll create a vector of `NA`s to store the data, then we'll use a loop to run `sum(sample(1:6,50,TRUE)==6)` 100 times.

```
die <- rep(NA, 100)  # vector to store results
for (i in 1:100) {
    die[i] <- sum(sample(1:6, 50, TRUE) == 6)
}
```

```
table(die)
```

```
# die
#  2  3  4  5  6  7  8  9 10 11 12 13 14 15 17
#  1  2  5  9  9 14 21 13  7  5  5  4  2  2  1
```

```
sum(die >= 12)
```

```
# [1] 14
```

So a value of 12 or greater comes up 14% of the time, which is a bit different from the p-value we got from `prop.test()`. To get a more stable p-value we need to try this 1000 time rather than a hundred (go ahead and try it if you like) We don't have strong enough evidence to conclude that the die is *not* fair. This is much faster than rolling a die 5000 times and recording the results!

> Note: here we created a vector to store the results before running the loop. This is recommended, as it is more efficient, but you *can* "grow" a vector from inside a loop.

2.7 Exercises.

1) The function `rep()` makes repeated series - for example try `rep(4,times=5)` and `rep(c(1,2),each=3)`. Use `rep()` to enter the sequence `1 1 1 1 2 2 2 2 3 3 3 3` repeated 3 times. Now convert it to a factor with the levels "Low","Medium", and "High". Can you change the levels to "Never","Rarely","Sometimes"?

2) Convert the factor from Problem 1 (the final part, with levels "Never","Rarely", and "Sometimes") into a character vector, and save it (assign it a name). Can you convert the character vector to a numeric vector?

3) Earlier we used the factor variable `a` (created by `a<-factor(scan(text="2 4 3 3 2 1 1 2 3 4 2 3 3 4 1 3 2 1 4 3 2 4"))`). Convert `a` into an ordered factor with levels `"Sm","Med","Lg","X-Lg"` (with `1` for "Sm"). How many values are equal to or larger than "Lg"?

4) We can use the output of `table()` with `barplot()` to view the frequency of levels in a factor. Extending our discussion of rolling die, we can use this to view the likelihood of rolling any particular value on one die using `barplot(table(sample(x=1:6,size=1000,replace=TRUE)))`. How does this change if we add the rolls of 2 dice together - i.e. what is the distribution of the sum of two dice? (*Hint:* recall that vectors are added in an element-wise fashion, and that `sample()` returns a vector).

Extra What happens if the two dice have different numbers of sides?

Chapter 3

Quantitative Variables

Creating and using continuous variables in R

3.1 Introduction

In the last chapter, we began working with qualitative data. Now we'll look at how to handle quantitative (continuous, or numerical) data.

3.2 Working with Numeric Data

In the first chapter we saw some functions that can be used with numeric vectors - here we'll demonstrate a bit more. We'll begin with some data that represents the size of a group of mp3 files (in MB), and get some summary statistics:

```
mp3 <- scan(text = "5.3 3.6 5.5 4.7 6.7 4.3 4.3 8.9 5.1 5.8 4.4")
mean(mp3)
```

```
# [1] 5.327273
```

```
var(mp3)
```

```
# [1] 2.130182
```

```
sd(mp3)
```

```
# [1] 1.459514
```

```
median(mp3)
```

```
# [1] 5.1
```

```
summary(mp3)
```

```
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   3.600   4.350   5.100   5.327   5.650   8.900
```

These functions mostly do what we'd expect. The function `fivenum()` gives similar output to `summary()`, but differ slightly, since `fivenum()` returns the upper and lower hinges [1], while `summary()` returns the 1st and 3rd quartiles, and these can differ slightly depending on the number of data points.

```
quantile(mp3, c(0.25, 0.75))
```

```
#  25%  75%
# 4.35 5.65
```

```
quantile(mp3, c(0.18, 0.36, 0.54, 0.72, 0.9))
```

```
#  18%  36%  54%  72%  90%
# 4.30 4.58 5.18 5.56 6.70
```

Notice that the function `quantile()` can return any desired quantile.

3.3 Hypothesis Testing

As we did for the qualitative data we can test hypotheses about quantitative data. For example, if we thought the mean was 4.5, we could test if the data support this by making a t-test. > Recall that `t` is the difference between observed and hypothesized means in units of the standard error, and standard error of the mean is standard deviation divided by the square root of n, and note that we can use `pt()` to calculate probabilities for a t-distribution. See "?Distributions" for more distributions.

```
t <- (mean(mp3) - 4.5)/(sd(mp3)/sqrt(length(mp3)))
pt(t, df = length(mp3) - 1, lower.tail = FALSE) * 2
```

```
# [1] 0.08953719
```

```
# *2 for 2 sided test
```

Recall that `length(mp3)^0.5` is the square root of n; we could also use `sqrt(length(mp3))`.

Of course, R has a built in t-test function that saves us the work:

```
t.test(mp3, mu = 4.5)
```

```
#
#  One Sample t-test
```

[1] The "upper hinge" is the median of the data points above the median. Depending on the number of data points, this may differ from the 3rd quartile.

```
#
# data:  mp3
# t = 1.8799, df = 10, p-value = 0.08954
# alternative hypothesis: true mean is not equal to 4.5
# 95 percent confidence interval:
#  4.346758 6.307788
# sample estimates:
# mean of x
#  5.327273
```

We provide the null value of the mean with the argument `mu`.

3.4 Resistant measures of center and spread

Since the mean and standard deviation can be quite sensitive to outliers, it is occasionally useful to consider some *resistant* measures of center and spread, so-called because they resist the influence of outliers. We'll add an outlier to our `mp3` data and experiment.

```
mp3[8] <- 10.8
mean(mp3)
```

```
# [1] 5.5
```

```
median(mp3)
```

```
# [1] 5.1
```

```
mean(mp3, trim = 0.1)
```

```
# [1] 5.122222
```

The median is substantially lower than the mean, but trimmed mean [2] is much nearer the median. Trimming more of the data will get still closer to the median.

For resistant measures of spread, one candidate is the "Interquartile range" or IQR, defined as the difference between the 3rd and 1st quartiles. Another candidate is the "median absolute deviation" or MAD, defined as the median of the absolute differences from the median, scaled by a constant [3]. If that sounds complex, it is simple in R, since R works easily with vectors.

```
IQR(mp3)
```

```
# [1] 1.3
```

[2] The trimmed mean is taken after removing ("trimming") the upper and lower ends of the data, in this case we specified 10% via the `trim=0.1` argument.

[3] The default value of the constant is 1.4826, as this gives a value comparable to standard deviation for normally distributed data.

```
median(abs(mp3 - median(mp3))) * 1.4826
```

```
# [1] 1.03782
```

```
mad(mp3)
```

```
# [1] 1.03782
```

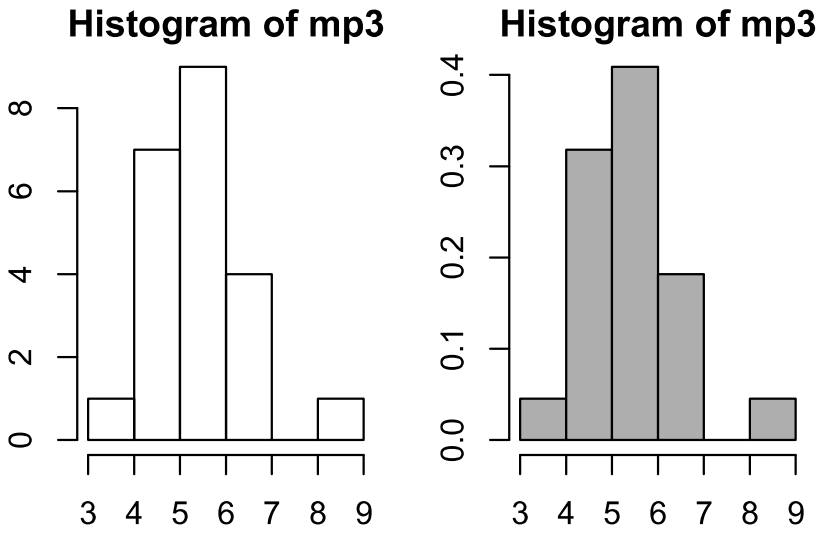
Of course, there is already a function for MAD, so we don't need to do it "by hand".

3.5 Visualizing Quantitative Data

One of the things we often want to do with qualitative data is "have a look". There are several ways to do this in R, and we'll review them here. The first and most common is the **histogram**. First we'll add another album's mp3 file sizes to our data `mp3` - note that `c()` can be used to combine vectors also.

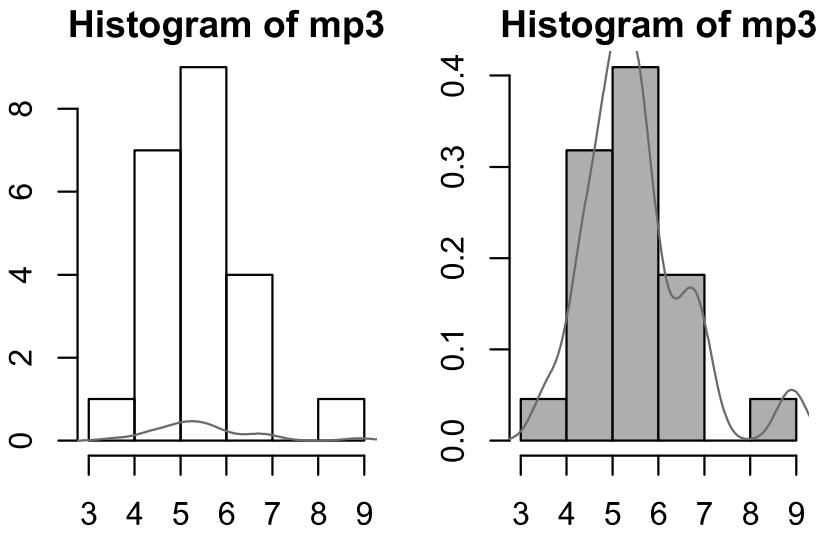
```
mp3[8] <- 8.9
mp3 <- c(mp3, scan(text = "4.9 5 4.9 5.4 6.2 5.6 5.1 5.8 5.5 6.7 7"))
```

```
par(mfrow = c(1, 2))  # split the plot
hist(mp3)
hist(mp3, prob = TRUE, col = "grey")
```



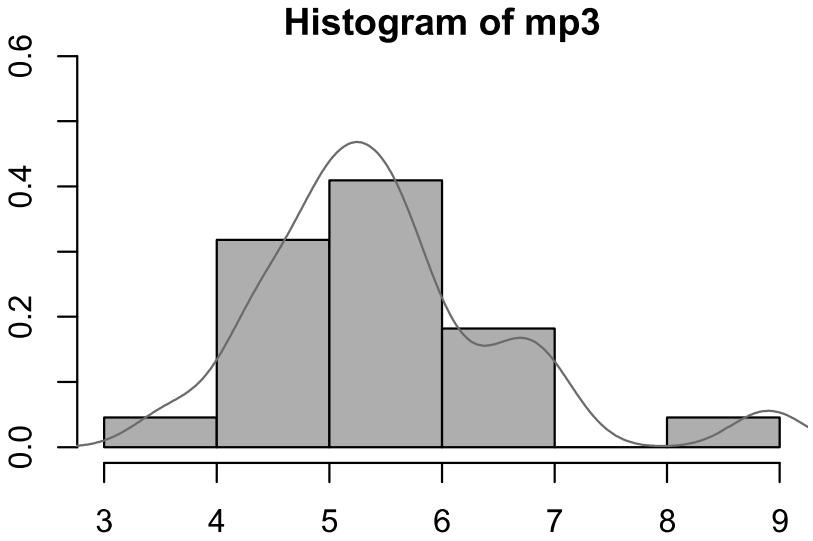
We have 2 versions of the histogram here - in the first, the y-axis is in units of *frequency*, so the scale changes for differing `n`, while the second is in units of *probability*, so distributions with differing `n` can be compared. Another useful visualization is the **kernel density estimate** (KDE), or density estimate, which approximates a probability density function.

```
par(mfrow = c(1, 2))  # split the plot
hist(mp3)
lines(density(mp3), col = "red")
hist(mp3, probability = TRUE, col = "grey")
lines(density(mp3), col = "red")
```



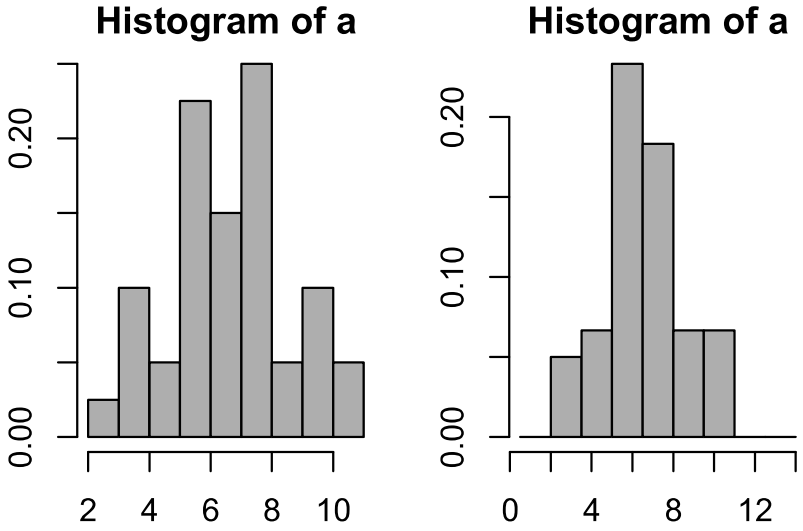
Note that the KDE approximates the histogram (and should have the same area), but for over-plotting on the histogram, the histogram must be in units of probability. In a case like this where our density function is off the scale, we might need to force the histogram to use a longer y-axis, which we can do using the `ylim` argument to specify the y-axis limits. (We'll use this optional argument with many plotting functions)

```
hist(mp3, probability = TRUE, col = "grey", ylim = c(0, 0.6))
lines(density(mp3), col = "red")
```



Here's another example, using `rnorm()`[4] to generate some random data from the normal distribution.

```
par(mfrow = c(1, 2))  # split the plot
a <- rnorm(n = 40, mean = 7, sd = 2)
hist(a, prob = T, col = "grey")
hist(a, prob = T, col = "grey", breaks = seq(0.5, 14, 1.5))
```



Notice that these two histograms represent the same data - this is one of the weaknesses of histograms: the idea they give us about the data depends on

[4]The `r` in `rnorm()` means "random"; `runif()` generates some uniformly distributed random data, and many others are included - see `?Distributions`

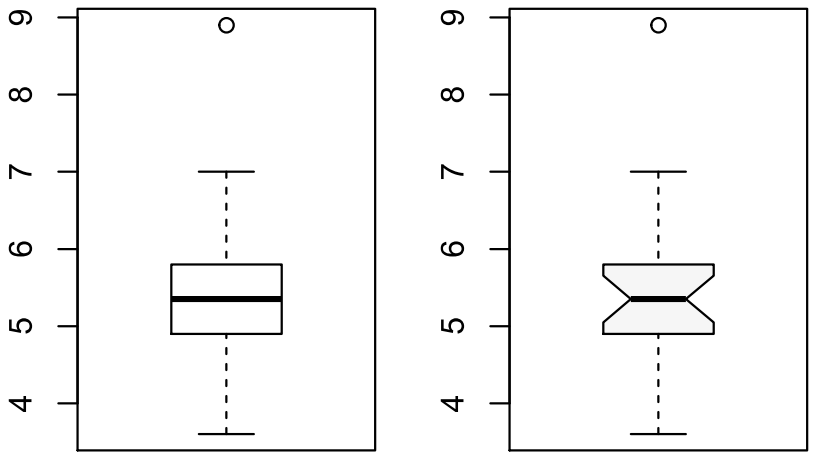
the bins used. This example shows how the `breaks` argument can be used to specify where the "bins" are in the histogram. Here we used the function `seq()` to create a sequence with lower and upper bounds and step size (0.5, 14, and 1.5 in our example). Breaks can also be an arbitrary sequence - try `breaks=c(0,4,5,5.5,6,6.5,7,7.5,8.5,14)`) and see what happens!

Note:

About arguments: Notice that here the argument `probability=TRUE` has been abbreviated as `prob=T`. R is happy to accept *unambiguous* abbreviations for arguments. R is also happy to accept un-named arguments *in the order they are entered* - we did this in our call to `seq()` in the last example - we could have specified `seq(from=0.5,to=14.0,by=1.5)`. For the simpler functions that I use frequently I don't usually spell out the arguments, though here I will tend to spell them out more frequently.

Boxplots are another useful visualization of quantitative data which show the median, lower and upper "hinges" and the upper and lower whiskers. They can also be "notched" to show a confidence interval about the median. Values beyond the whiskers are possible outliers.

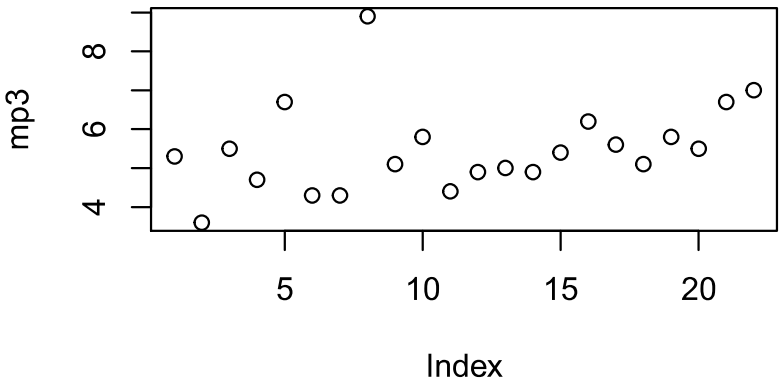
```
par(mfrow = c(1, 2))
boxplot(mp3)
boxplot(mp3, notch = TRUE, col = "cornsilk")
```



The value of `8.9` seems rather suspicious doesn't it?

We can visualize the "raw" data using `plot()`. Since `plot()` requires arguments for both x and y, but we are only providing x, the indices of x will be used for x, and the values of x for y.

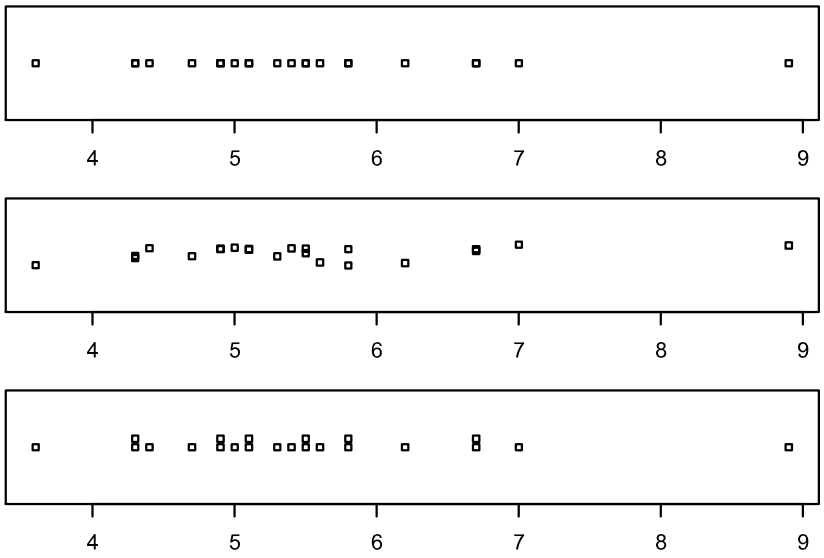
```
plot(mp3)
```



This method doesn't give us summary values like a boxplot does, but it has the advantage of letting us look for structure in the data (though it won't work for very large datasets). For example, here it is evident that the first half of the data set is more variable than the second half. Whether this is important or not depends on nature of the data.

Two other tools that are sometimes useful are the **stripchart** and the **stem and leaf** plot. The "stripchart" is a sort of "one-dimensional scatterplot". The argument `method` tells R how to display values that would over plot.

```
par(mfrow = c(3, 1))
stripchart(mp3)
stripchart(mp3, method = "jitter")
stripchart(mp3, method = "stack")
```



A final trick is the stem and leaf plot, which was originally developed because it could quickly be created with pencil and paper. While it looks simple and a bit

crude, it has the advantages that it preserves the original data - from a stem and leaf plot you can reconstruct the actual values in the data, which you can't do with most of the other visualization tools we've looked at here.

```
stem(mp3)
```

```
#
#   The decimal point is at the |
#
#   2 | 6
#   4 | 3347990113455688
#   6 | 2770
#   8 | 9
```

```
stem(mp3, scale = 2)
```

```
#
#   The decimal point is at the |
#
#   3 | 6
#   4 | 334799
#   5 | 0113455688
#   6 | 277
#   7 | 0
#   8 | 9
```

The stem and leaf chart shows that the lowest value is 3.6, and occurs once, while the maximum value is 8.9, which may be an outlier. Stem and leaf plots are useful for exploratory data analysis (EDA), but I've rarely seen them published; histograms are much more commonly used.

3.6 Converting Quantitative Data to Qualitative

Sometimes we need to take a quantitative variable and "simplify" it by reducing it to categories. The function `cut()` can do this.

```
m.r <- cut(mp3, breaks = c(3:9))  # specify the breaks
m.r
```

```
#  [1] (5,6] (3,4] (5,6] (4,5] (6,7] (4,5] (4,5] (8,9] (5,6] (5,6]
# [11] (4,5] (4,5] (4,5] (4,5] (5,6] (6,7] (5,6] (5,6] (5,6] (5,6]
# [21] (6,7] (6,7]
# Levels: (3,4] (4,5] (5,6] (6,7] (7,8] (8,9]
```

```
m.r[which(mp3 == 5)]  # values of 5.0 coded as (4,5]
```

```
# [1] (4,5]
```

```
# Levels: (3,4] (4,5] (5,6] (6,7] (7,8] (8,9]
```

Note non-matching brackets here: `(4,5]` - this means "greater than 4 and less than or equal to 5", so 4.0 is *not* included, but 5.0 *is* included in the interval `(4,5]`. We can demonstrate that this is so: `m.r[which(mid.rib==5.0)]` returns `(4,5]`.

We can now treat the factor `m.r` like any other factor variable, and assign other names to the levels as we see fit.

```
table(m.r)
```

```
# m.r
# (3,4] (4,5] (5,6] (6,7] (7,8] (8,9]
#     1     7     9     4     0     1
```

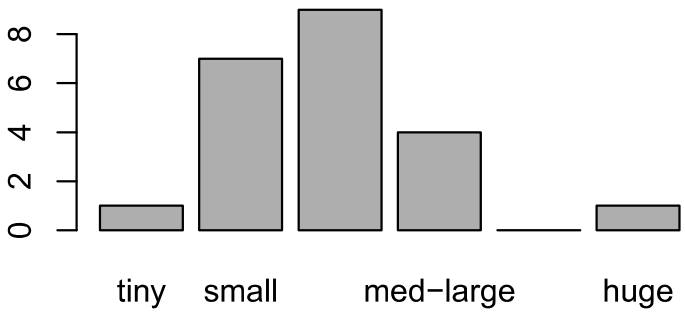
```
levels(m.r)
```

```
# [1] "(3,4]" "(4,5]" "(5,6]" "(6,7]" "(7,8]" "(8,9]"
```

```
levels(m.r) <- c("tiny", "small", "medium", "med-large", "large",
    "huge")
table(m.r)
```

```
# m.r
#      tiny     small    medium med-large     large      huge
#         1         7         9         4         0         1
```

```
plot(m.r)
```



Note that we could use any grouping we want to for `breaks`, just as we saw with `hist()`. Finally notice that in *this* case the function `plot()` creates a barplot. `plot()` is what is known as a *generic function*, meaning that what it does depends on the type of input it is given. For a qualitative variable it will return a barplot. As we progress, we'll see more kinds of output from `plot()`. The use of generic functions in R reduces the number of commands one needs to learn.

3.7 Fitting and Modeling Distributions

t-distribution can be examined with a group of functions: `dx()`, `px()`, `qx()`, and `rx()` giving (respectively) the density, probabilities, quantiles, and random samples from the x distribution; arguments include parameters specific to the distributions. Most common distributions are included, see `?Distributions` for a full listing.

We'll demonstrate some of these functions for the exponential distribution. For example, what would be the probability of value of 3 or more from an exponential distribution with a rate parameter of 1?

```
pexp(q = 3, rate = 1, lower.tail = FALSE)
```

```
# [1] 0.04978707
```

The p-value is pretty close to 0.05, so about 1 of 20 random values from this distribution would be greater than 3. Let's generate 100 values and see how many are greater than or equal to 3.

```
x.exp <- rexp(n = 100, rate = 1)
sum(x.exp >= 3)
```

```
# [1] 5
```

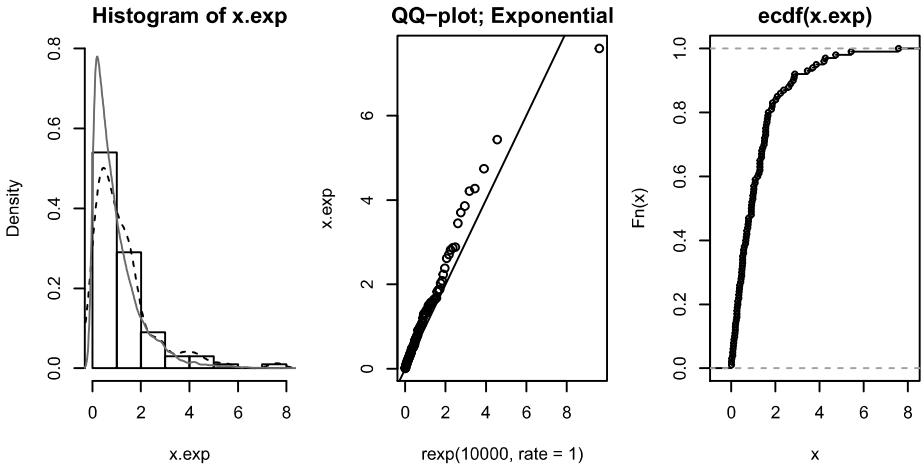
Fairly close to 4.97, and a larger sample size would get even closer:

```
sum(rexp(n = 1e+05, rate = 1) >= 3)/1000
```

```
# [1] 5.047
```

Let's look at how we'd investigate the fit of a distribution. Imagine we have a sample of 100 values, and we think they come from an exponential distribution with a rate of 1.

```
x.exp <- rexp(n = 100, rate = 0.7)
hist(x.exp, prob = TRUE, ylim = c(0, 0.8))
lines(density(x.exp), lty = 2)
lines(density(rexp(10000, rate = 1)), col = "red")
qqplot(x = rexp(10000, rate = 1), y = x.exp, main = "QQ-plot; Exponential")
abline(a = 0, b = 1)
plot(ecdf(x.exp), pch = 21)
```



The first two plots here suggest that the distribution isn't what we hypothesize - rate=1 - (of course in this example we *know* the rate is not 1, our code that generates it shows the value is 0.7). For more ideas on modeling distributions in R see [Fitting Distributions with R] (http://cran.r-project.org/doc/contrib/Ricci-distributions-en.pdf)

3.8 Exercises.

1) The R function `rnorm(n,mean,sd)` generates random numbers from a normal distribution. Use `rnorm(100)` to generate 100 values and make a histogram. Repeat two or three times. Are the histograms the same?

2) Make a histogram from the following data, and add a density estimate line to it. (use `scan()` to enter the data). Try changing the bandwidth parameter for the density estimate (use the argument "adj" for density( ); 1 is the default, 2 means double the bandwidth). How does this change the density estimate?
26 30 54 25 70 52 51 26 67 18 21 29 17 12 18 35 30 36 36 21 24 18 10 43 28 15 26 27 Note: If you are submitting this HW for class in a .txt file, you needn't include the plot, just include a brief description of how changing the bandwidth parameter for `density()` alters the shape of the curve. Note that the bandwidth argument here can be a string that matches on of several methods for bandwidth calculation or a numeric value for the bandwidth.

3) Using the data above compare: a)the mean and median and, b) the standard deviation, IQR and the MAD.

4) Use the function `boxplot()` to find possible outliers in the data from problem 2 (outliers shown as individual points; you don't need to show the boxplot, but you need to show the values that are possible outliers). Compare: a) the mean and median of the data with and without the outliers and b) the standard deviation and MAD for the data with and without the outliers.

Chapter 6

Bivariate Data

Basic approaches to dealing with two variables

6.1 Introduction

A great deal of statistical analysis is based on describing the relationship between two variables. For example - how does planting density (high, medium, or low) alter crop yield? How is home price related to lot size? How are height and foot size related? Is the incidence of heart disease different for men and women? Here we'll consider working with two qualitative variables, one qualitative and one quantitative variable, and two quantitative variables.

6.2 Two Qualitative Variables

We sometimes want to see how two qualitative (factor) variables are related. Here we'll work with some data for number of cylinders [1] (`cyl`) and transmission type (`am`) from 32 models of cars reported in *Motor Trend* magazine in 1974 [2].

```
cyl<-factor(scan(text= "6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4
                 8 8 8 8 4 4 4 8 6 8 4"))
am<-factor(scan(text= "1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
               0 0 0 0 1 1 1 1 1 1 1"))
```

[1] Number of engine cylinders is a nice example of a *numeric factor*. Not only are the values constrained to integer values, but there are only a few values that are common, although there have been a few 3 or 5 cylinder engines. Such a factor could be treated as "ordered", but that is beyond the scope of these notes. This variable might also be treated as numeric - the analyst would have to decide what makes the most sense here.

[2] This data is found in the `mtcars` data set that is included with R: as we'll see later you can access the whole data set using `data(mtcars)`.

```
levels(am)<-c("auto","manual")
table(cyl,am)
```

```
#    am
# cyl auto manual
#   4    3      8
#   6    4      3
#   8   12      2
```

```
table(am, cyl)
```

```
#         cyl
# am        4  6  8
#   auto    3  4 12
#   manual  8  3  2
```

It appears that manual transmissions were more common with smaller numbers of cylinders, while cars with 8 cylinders were far more likely to have automatic transmissions. Notice that our old friend `table()` can be used to give a two-way frequency table as well. Also note that as we discussed in Chapter 2, it is simpler to enter a long factor as level numbers and assign the levels later.

Sometimes we would rather see tables like this expressed as proportions. R can easily do this via the function `prop.table()`.

```
tab <- table(cyl, am)
prop.table(tab, margin = 1)
```

```
#    am
# cyl      auto    manual
#   4 0.2727273 0.7272727
#   6 0.5714286 0.4285714
#   8 0.8571429 0.1428571
```

The `margin=1` tells R that we want the proportions within *rows*. We can see that 85% of 8 cylinder cars have an automatic compared to 27% of four cylinder cars. We can also use `margin=2` to have proportions within *columns*. If we don't include the `margin` argument, the default it to proportions of the entire table.

```
prop.table(tab, margin = 2)
```

```
#    am
# cyl      auto    manual
#   4 0.1578947 0.6153846
#   6 0.2105263 0.2307692
#   8 0.6315789 0.1538462
```

```
prop.table(tab)
```

```
#    am
```

```
# cyl    auto  manual
#   4 0.09375 0.25000
#   6 0.12500 0.09375
#   8 0.37500 0.06250
```

From the first we can see that 63% of cars with automatic transmissions had 8 cylinders. From the second we can see that 38% of cars had both automatic transmission and 8 cylinders.

Finally, note that in these proportion tables R is giving us a larger number of decimal places than we might really want. This can be controlled in several ways - the simplest is via the function `signif()`, which control the number of significant digits printed [3].

```
signif(prop.table(tab), 2)
```

```
#    am
# cyl  auto manual
#   4 0.094  0.250
#   6 0.120  0.094
#   8 0.380  0.062
```

We may want to test whether there is any association between categorical variables. The simplest approach is often to use the χ^2 (Chi^2) test with the null hypothesis that the variables are independent.

```
chisq.test(tab)
```

```
# Warning in chisq.test(tab): Chi-squared approximation may be
# incorrect
```

```
#
#  Pearson's Chi-squared test
#
# data:  tab
# X-squared = 8.7407, df = 2, p-value = 0.01265
```

In this case we have evidence of association between more cylinders and automatic transmissions, but we have a warning - this is because there are too few values in some of our groups - χ^2 is not valid if more than 20% of group have expected values less than 5. We can confirm that this is the problem by capturing the output of `chisq.test()`.

```
names(chisq.test(tab))  # see all the stuff that it produced?
```

```
# Warning in chisq.test(tab): Chi-squared approximation may be
# incorrect
```

[3] You can also change this by changing R's options - see `?options`. The advantage of using `signif()` is that it is temporary and specific to the current command.

```
# [1] "statistic" "parameter" "p.value"   "method"    "data.name"
# [6] "observed"  "expected"  "residuals" "stdres"
```

```
chisq.test(tab)$expected
```

```
# Warning in chisq.test(tab): Chi-squared approximation may be
# incorrect
```

```
#    am
# cyl    auto  manual
#   4 6.53125 4.46875
#   6 4.15625 2.84375
#   8 8.31250 5.68750
```

We have a couple of options to deal with such an violation of assumtions:

1. As long as we have a table that is greater than 2x2 we *may* be able to combine some categories to increase the expected values, though that may not make sense depending on what the categories represent.

2. Perhaps better is the option noted in `? chisq.test` that allows us to *simulate the null distribution*, which lets us avoid the violation of assumptions (and the warning). Of course, you should use a reasonably large number of iterations, which does impose a slight speed penalty. Something like `chisq.test(tab, simulate.p.value=TRUE, B=5000)` should work. The `B=5000` tells R to generate 5000 random scrables of the data that have the same row and column totals.

Note that it is **not** an option to simply multiply the table by some factor that ensures our expected values don't fall below 5. This is because the $/chi^2$ test statistic is calculate on observed and expected *frequencies*, not on *relative frequencies* or *proportions*.

By the way, we could also calculate the expected values by hand - `rowSums()` will give the row sums of an array, so `rowSums(x)/sum(x)` will give the row proportions for an array.

```
sum(tab)  # total
```

```
# [1] 32
```

```
outer(rowSums(tab)/sum(tab), colSums(tab)/sum(tab)) * sum(tab)
```

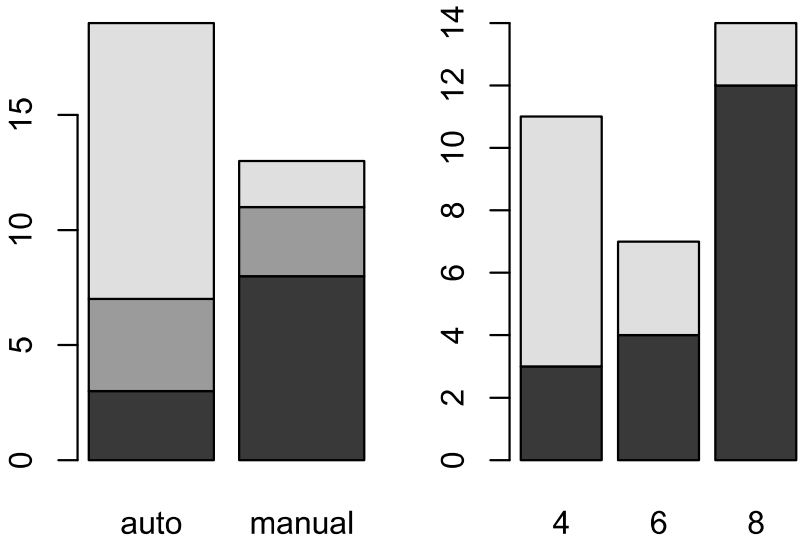
```
#      auto  manual
# 4 6.53125 4.46875
# 6 4.15625 2.84375
# 8 8.31250 5.68750
```

It is nice to see that our "by hand" calculation matches R's calculations!

There are a couple of possible ways to visualize such data. One option is using a

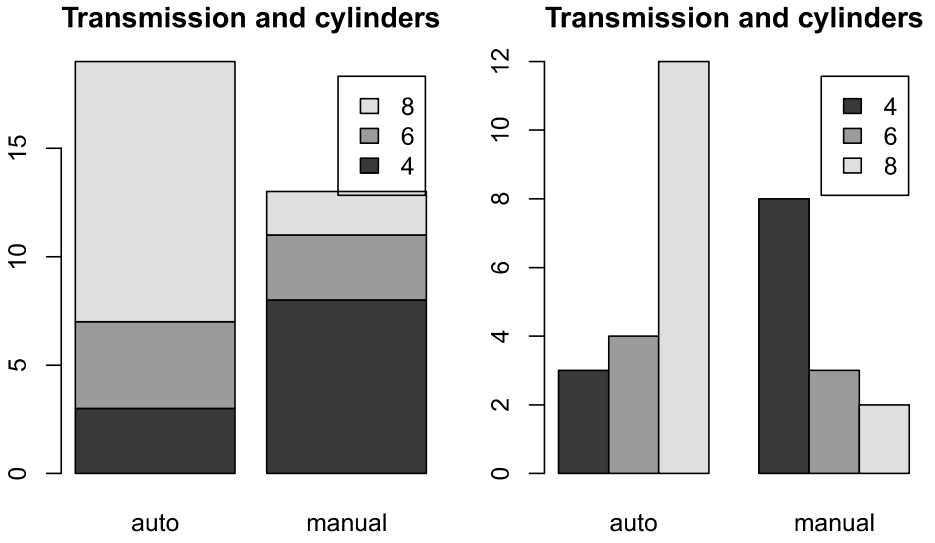
barplot.

```
op = par(mfrow = c(1, 2))
barplot(table(cyl, am))
barplot(table(am, cyl))
```



Note: the function `par()` is used to set graphical *parameters* - in this case we're specifying that the plotting window will be divided into 1 row and 2 columns. We've simultaneously saved the old par settings as `op`. There are a few more options we can use to dress this up. The confusing thing here is that it is the *old* settings that are saved, not the new ones. Also note that the function `options()` we discussed above functions in a similar way with assignment of old options. It is possible that you may never need to set options - it just depends on how you use R.

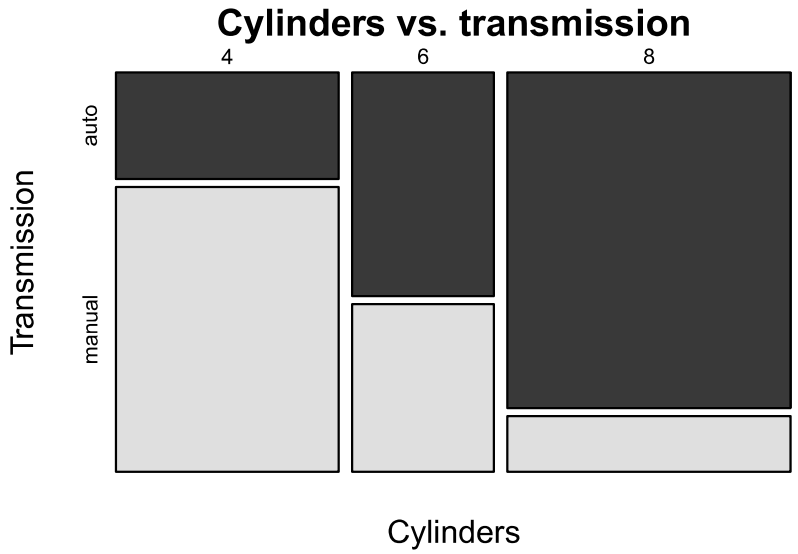
```
par(mfrow=c(1,2),mar=c(3,3,3,0.5))
barplot(table(cyl,am),legend.text=TRUE,main="Transmission and cylinders")
barplot(table(cyl,am),beside=TRUE,legend.text=TRUE,
        main="Transmission and cylinders")
```




```
par(op)
```

Here in addition to dividing the plotting window we've used `par()` to reduce the plot margins. The final line restores the old `par` settings we saved earlier. [4] Another option which is less familiar is the *mosaic plot*, which shows the proportions of each combination of factors.

```
mosaicplot(table(cyl, am), color = T, main = "Cylinders vs. transmission",
    ylab = "Transmission", xlab = "Cylinders")
```



[4] Restoring the old `par` settings is sometimes important - once we split the plotting window it stays split, and we might not want it to.

Note that many of the arguments here are optional. You can try leaving them out to see what they do; the minimum is `mosaicplot(table(cyl,am))`.

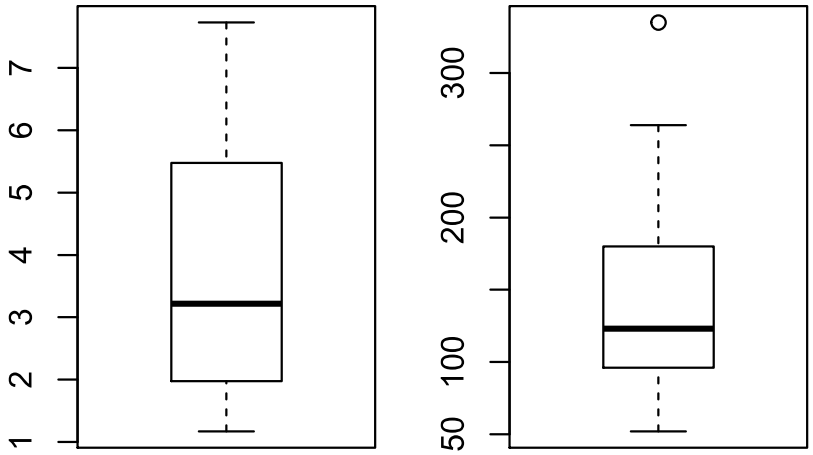
6.3 Two Quantitative Variables

We frequently find that we are looking for association between two quantitative variables. For example, using the motor trend cars data we might wish to look at the association between engine displacement (here in liters) and power output (horsepower).

```
disp=scan(text=
    "2.62 2.62 1.77 4.23 5.90 3.69 5.90 2.40 2.31 2.75 2.75 4.52
    4.52 4.52 7.73 7.54 7.21 1.29 1.24 1.17 1.97 5.21 4.98 5.74
    6.55 1.29 1.97 1.56 5.75 2.38 4.93 1.98")
hp=scan(text=
    "110 110 93 110 175 105 245 62 95 123 123 180 180 180 205 215
    230 66 52 65 97 150 150 245 175 66 91 113 264 175 335 109")
```

6.3.1 Exploring the data

```
op = par(mfrow = c(1, 2))
boxplot(disp)
boxplot(hp)
```




```
par(op)
```

Both variables show a bit of skew, with a larger number of low values. The plot of horsepower shows one possible outlier. We can find which it is using logical extraction:

```
data(mtcars)  # load the whole data set
mtcars[which(mtcars$hp > 250), ]
```

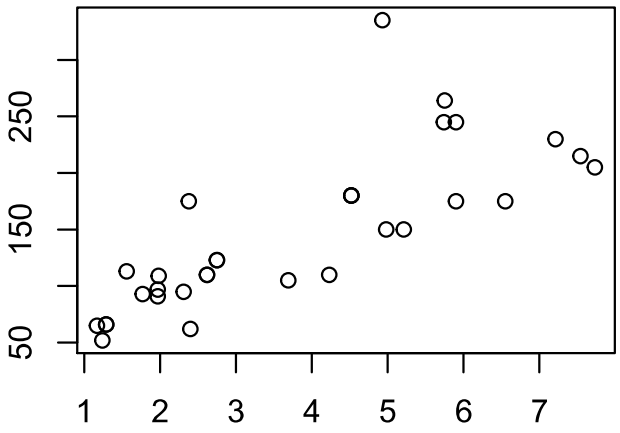
```
#                 mpg cyl disp  hp drat   wt qsec vs am gear carb
# Ford Pantera L 15.8   8  351 264 4.22 3.17 14.5  0  1    5    4
# Maserati Bora  15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
```

This shows only 2 cars with horsepower greater than 250. Notice that here we used the function `data()` to load one of the built-in data sets, and that we used the `$` to specify a variable within a dataset. We'll discuss this in more detail soon. Also notice that within the `[]` we have a comma - the format is `[rownumber,columnnumber]`, and we want the rows with `hp>250`.

6.3.2 Correlation

We might guess that there is some correlation between displacement and power. A simple scatter plot will confirm this:

```
plot(x = disp, y = hp)
```




```
cor(disp, hp)
```

```
# [1] 0.7910567
```

Notice that `plot()` here gives us a scatter plot [5]. The correlation coefficient r is reasonably high at 0.7910567.

By default `cor()` gives us the *pearson* correlation. By setting the `method` argument to `method="spearman"` we can get the *spearman rank* correlation (which is more robust to outliers). It should be noted that the `cor()` function needs to be told how to deal with missing values (`NA`) - this is done via the argument

[5] We could omit the names of the `x` and `y` arguments, the first will be taken as `x` and the second as `y`. Also `plot(hp~disp)` would work.

`use`, which tells R which values to use. A setting of `use="complete.obs"` will often work (see `?cor` for more information).

6.3.3 Regression

Often we want to go a step further and *model* one variable *as a function* of another. With two quantitative variables this is known as linear regression (regression for short). In this case, we might well suspect that larger displacement engines should be more powerful. In R such models are fit using `lm()` (for "linear model"):

```
model = lm(hp ~ disp)
model
```

```
#
# Call:
# lm(formula = hp ~ disp)
#
# Coefficients:
# (Intercept)         disp
#       45.69        26.71
```

```
summary(model)
```

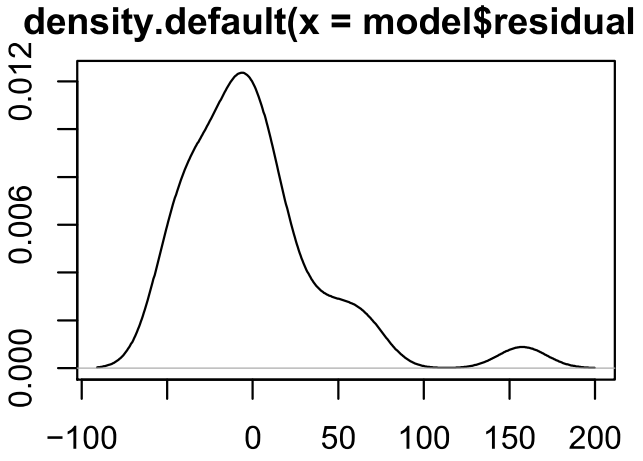
```
#
# Call:
# lm(formula = hp ~ disp)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -48.682 -28.396  -6.497  13.571 157.620
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   45.695     16.128   2.833  0.00816 **
# disp          26.711      3.771   7.083 7.09e-08 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 42.64 on 30 degrees of freedom
# Multiple R-squared:  0.6258,Adjusted R-squared:  0.6133
# F-statistic: 50.16 on 1 and 30 DF,  p-value: 7.093e-08
```

Notice a couple of things here:
1. in our call to `lm()` we specified `hp~disp` - this means hp *as a function of* disp. This type of model notation is used by a number of functions in R.
2. `lm(hp~disp)` returns only the intercept and slope for the model.
3. `lm()` has actually done *much more* - it has created an *"lm object"* that we have

named `model`. Type `names(model)` to see what all is there - you can access all of these - for example `model$residuals` will return the residuals from the model. 4. The function `summary()` when called on an lm object, gives a very helpful summary of the regression. This shows that our model is highly significant, with p-value = 7.093 x 10^-8 .

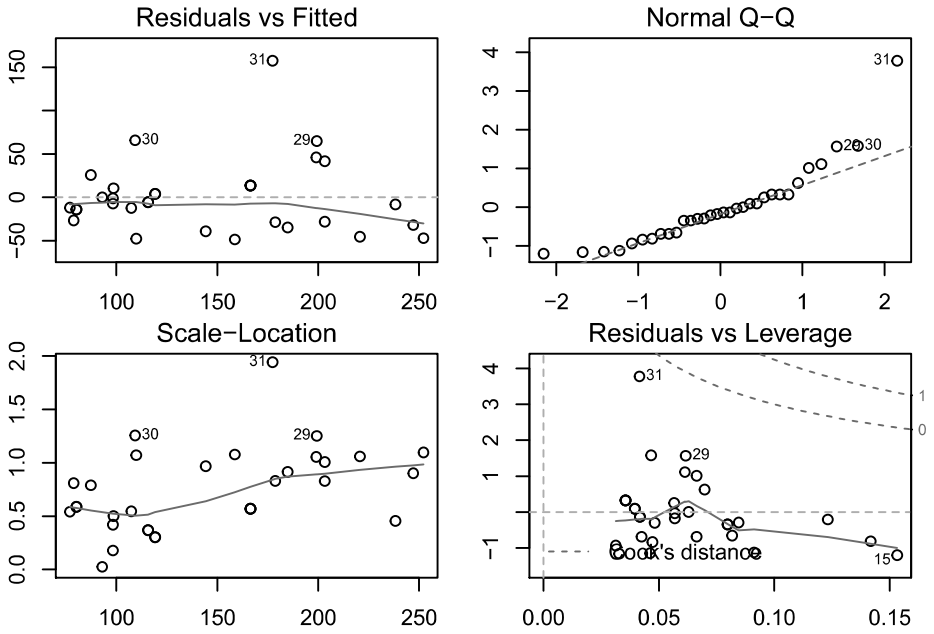
If you recall one of the assumptions in regression is that the residuals are normally distributed. We can check to see if this is true:

```
plot(density(model$residuals))
```



Overall, the residuals are not really normally distributed, but they are probably normal enough for the regression to be valid. Of course, checking model assumptions is a common (and *necessary*) task, so R makes it easy to do.

```
op = par(mfrow = c(2, 2))
plot(model)
```

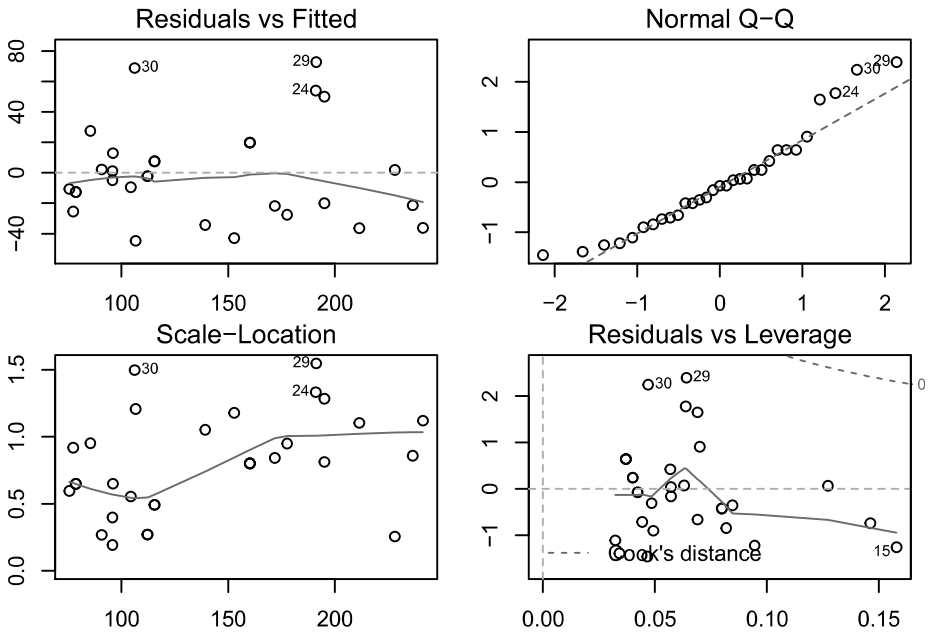


The normal Q-Q plot does show that we may have one outlier, point 31 (The Maserati Bora). We could refit the model without it to see if it fits better.

```
op = par(mfrow = c(2, 2))
model2 <- lm(hp[-31] ~ disp[-31])
summary(model2)
```

```
#
# Call:
# lm(formula = hp[-31] ~ disp[-31])
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -44.704 -21.601  -2.255  16.349  72.767
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   46.146     11.883   3.883 0.000548 ***
# disp[-31]     25.232      2.793   9.033 6.29e-10 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 31.41 on 29 degrees of freedom
# Multiple R-squared:  0.7378,Adjusted R-squared:  0.7287
# F-statistic:  81.6 on 1 and 29 DF,  p-value: 6.291e-10
```

```
plot(model2)
```



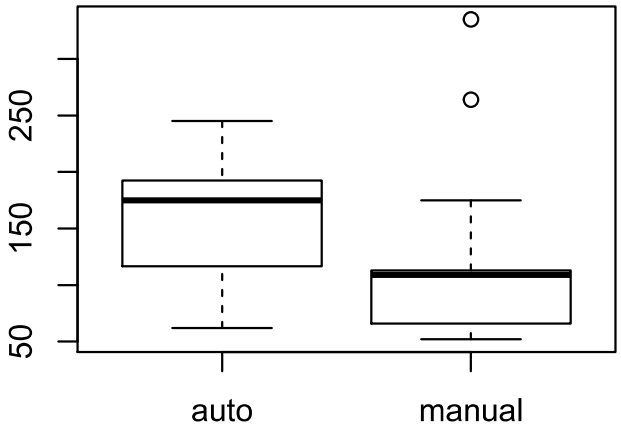

```
par(op)
```

Removing the outlier really improves the model fit - the R^2 increases to 0.729, and the residuals look much more normal (the Q-Q plot is more linear). It is not generally wise to remove a value just because it is an outlier, though if you have reason to beleive the value is erroneous that may be grounds for excluding it from the analysis. It is less clear-cut whether removing outliers to meet assumptions for an analytical method is accepteable - I would probably err on the side of "don't remove". However, I do think it is a good idea to investigate the extent to which unusual values (outliers) influence the findings in a study - in a situation where their presence substantially altered the conclusions of a study, we would want to know that.

6.4 Qualitative and Quantitative Variables

When we have a quantitative and a qualitative variable, we can use similar tools to what we would use for two quantitative variables. Consider the data on cars - do we expect a difference in horsepower between cars with automatic and manual transmissions?

```
plot(am, hp)
```



It appears that more cars with automatic transmissions are generally more powerful, though the two most powerful cars have manual transmissions - we saw these earlier. We can use a two-sample t-test to see if these groups are different.

```
t.test(hp ~ am)
```

```
#
#  Welch Two Sample t-test
#
# data:  hp by am
# t = 1.2662, df = 18.715, p-value = 0.221
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
#  -21.87858  88.71259
# sample estimates:
#   mean in group auto mean in group manual
#             160.2632             126.8462
```

This show that the means are not different - likely the influence of the two "super-cars" with manual transmissions pulls the mean up enough to mask the difference.

6.4.1 ANOVA

Note that if we had more than two groups, we'd need a different approach - we can use `oneway.test()` to do a simple ANOVA. For two groups this is equivalent to the t-test, but it will work for more than two groups also.

Since R uses the function `lm()` for both regression and ANOVA, you may find it helpful to think about ANOVA as a kind of regression, where the predictor variable (x-axis) is *categorical*.

NOTE: `lm()` and `oneway.test()` will return errors if you use a factor as the response variable, so recall that "~" should be read as "as a function of", so that

`cyl~hp` is "cylinders (factor in our case) ~ horsepower" would not work here.

```
oneway.test(hp ~ am)
```

```
#
#  One-way analysis of means (not assuming equal variances)
#
# data:  hp and am
# F = 1.6032, num df = 1.000, denom df = 18.715, p-value =
# 0.221
```

```
oneway.test(hp ~ cyl)
```

```
#
#  One-way analysis of means (not assuming equal variances)
#
# data:  hp and cyl
# F = 35.381, num df = 2.000, denom df = 16.514, p-value =
# 1.071e-06
```

```
summary(lm(hp ~ cyl))
```

```
#
# Call:
# lm(formula = hp ~ cyl)
#
# Residuals:
#    Min     1Q Median     3Q    Max
# -59.21 -22.78  -8.25  15.97 125.79
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)    82.64      11.43   7.228 5.86e-08 ***
# cyl6           39.65      18.33   2.163   0.0389 *
# cyl8          126.58      15.28   8.285 3.92e-09 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 37.92 on 29 degrees of freedom
# Multiple R-squared:  0.7139,Adjusted R-squared:  0.6941
# F-statistic: 36.18 on 2 and 29 DF,  p-value: 1.319e-08
```

We'll dig into ANOVA in more depth in a later chapter.

6.5 Exercises

1) Using the data `cyl` and `am` (transmission type) from Part II, group vehicles based into 8 cylinder and less than 8 cyl. Test whether there is evidence of

association between number of cylinders (8 or <8) and type of transmission. (*Hint* - use `levels()` to re-level `cyl` and then use `chisq.test()`).

2) The built in dataset `faithful` records the time between eruptions and the length of the prior eruption (both in minutes) for 272 inter-eruption intervals (load the data with `data(faithful)`). Examine the distribution of each of these variables with `stem()` or `hist()`. Plot these variables against each other with the length of each eruption (`eruptions`) on the x- axis. How would you describe the relationship? Recall that you can use `faithful$eruptions` to access `eruptions` within `faithful`.

3) Fit a regression of `waiting` as a function of `eruptions` (i.e. `waiting~eruptions`; `waiting` on the y-axis and `eruptions` on the x-axis). What can we say about this regression? Compare the distribution of the residuals (`model$resid` where `model` is your lm object) to the distribution of the variables.

4) The clustering evident in this data might suggest regression is not the best way to analyze it. Might an ANOVA be better? Create a categorical variable from `eruptions` to separate long eruptions from short eruptions (2 groups) and fit a model of `waiting` based on this. (*Hint:* a: use `cut()` to make the categorical variable. The argument 'breaks=' can be a single value - the number of groups to create, or a vector of n values defining the edges of n-1 groups. b: Use `lm()` to fit the model, exactly as you did for the regression. lm() with a categorical predictor variable is an ANOVA.) How does this model compare with that from Ex 3? How did you choose the point at which to cut the data? How might changing the cut-point change the results?

Chapter 7

The Data Frame

The R equivalent of the spreadsheet

7.1 Introduction

Most analytical work involves importing data from outside of R and carrying out various manipulations, tests, and visualizations. In order to complete these tasks, we need to understand how data is stored in R and how it can be accessed. Once we have a grasp of this we can consider how it can be imported (see Chapter 8).

7.2 Data Frames

We've already seen how R can store various kinds of data in vectors. But what happens if we have a mix of numeric and character values? One option is a *list*

```
a <- list(c(1, 2, 3), "Blue", factor(c("A", "B", "A", "B", "B")))
a
```

```
# [[1]]
# [1] 1 2 3
#
# [[2]]
# [1] "Blue"
#
# [[3]]
# [1] A B A B B
# Levels: A B
```

Notice the `[[]]` here - this is the *list element* operator. A list in R can contain an arbitrary number of items (which can be vectors) which can be of different

forms - here we have one numeric, one character, an one factor, and they are all of different lengths.

A list like this may not be something you are likely to want to use often, but in most of the work you will do in R, you will be working with data that is stored as a *data frame* - this is R's most common data structure. A data frame is a special type of list - it is a list of vectors that have the same length, and whose elements correspond to one another - i.e. the 4th element of each vector correspond. Think of it like a small table in a spreadsheet, with the columns corresponding to each vector, and the rows to each record.

There are several different ways to interact with data frames in R. "Built in" data sets are stored as data frames and can be loaded with the function `data()`. External data can be read into data frames with the function `read.table()` and its relative (as we'll see in the next chapter). Existing data can be converted into a data frame using the function `data.frame()`.

```
cyl<-factor(scan(text= "6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4
    8 8 8 8 4 4 4 8 6 8 4"))
am<-factor(scan(text= "1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
    0 0 0 0 1 1 1 1 1 1 1"))
levels(am)<-c("auto","manual")
disp<-scan(text= "2.62 2.62 1.77 4.23 5.90 3.69 5.90 2.40 2.31
    2.75 2.75 4.52 4.52 4.52 7.73 7.54 7.21 1.29 1.24 1.17 1.97
    5.21 4.98 5.74 6.55 1.29 1.97 1.56 5.75 2.38 4.93 1.98")
hp<-scan(text= "110 110  93 110 175 105 245 62 95 123 123 180 180
    180 205 215 230 66 52 65 97 150 150 245 175 66 91 113 264 175
    335 109")
```

Here we've re-created the data on cars that we used in the last chapter.

```
car <- data.frame(cyl, disp, hp, am)
head(car)
```

```
#   cyl disp  hp     am
# 1   6 2.62 110 manual
# 2   6 2.62 110 manual
# 3   4 1.77  93 manual
# 4   6 4.23 110   auto
# 5   8 5.90 175   auto
# 6   6 3.69 105   auto
```

```
summary(car)
```

```
#  cyl         disp             hp             am
#  4:11   Min.   :1.170   Min.   : 52.0   auto  :19
#  6: 7   1st Qu.:1.978   1st Qu.: 96.5   manual:13
#  8:14   Median :3.220   Median :123.0
#         Mean   :3.781   Mean   :146.7
```

```
#          3rd Qu.:5.343   3rd Qu.:180.0
#          Max.   :7.730   Max.   :335.0
```

Now we've created a data frame named `car`. The function `head()` shows us the first 6 rows (by default). Here we see that `summary()`, when called on a data frame, gives the appropriate type of summary for each variable. The variables within the data frame have names, and we can use the function `names()` to retrieve or change these.

```
names(car)
```

```
# [1] "cyl"  "disp" "hp"   "am"
```

```
names(car)[4] <- "trans"
names(car)
```

```
# [1] "cyl"   "disp"  "hp"    "trans"
```

```
car$am
```

```
# NULL
```

```
car$trans
```

```
#  [1] manual manual manual auto   auto   auto   auto   auto
#  [9] auto   auto   auto   auto   auto   auto   auto   auto
# [17] auto   manual manual manual auto   auto   auto   auto
# [25] auto   manual manual manual manual manual manual manual
# Levels: auto manual
```

Data in data frames can be accessed in several ways. We can use the indexing operator `[]` to access parts of a data frame by rows and columns. We can also call variables in a data frame by name using the `$` operator.

```
car[1:3, ]
```

```
#   cyl disp  hp  trans
# 1   6 2.62 110 manual
# 2   6 2.62 110 manual
# 3   4 1.77  93 manual
```

```
car[, 3]
```

```
#  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205
# [16] 215 230  66  52  65  97 150 150 245 175  66  91 113 264 175
# [31] 335 109
```

```
car$hp
```

```
#  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205
# [16] 215 230  66  52  65  97 150 150 245 175  66  91 113 264 175
# [31] 335 109
```

Note that when indexing a data frame we use 2 indices, separated by a comma (e.g. `[2,3]`). Leaving one value blank implies "all rows" or "all columns". Here the first line gives us rows 1:3, the second and third both give us the `hp` variable.

Where we've created a new data frame in this way it is important to note that *R has copied the vectors that make up the data frame.* So now we have `hp` and `car$hp`. It is important to know this because if we change one, the other is not changed.

```
hp[1] == car$hp[1]
```

```
# [1] TRUE
```

```
hp[1] <- 112
hp[1] == car$hp[1]
```

```
# [1] FALSE
```

In a case like this, it might be a good idea to *remove* the vectors we used to make the data frame, just to reduce the possibility of confusion. We can do this using the function `rm()`.

```
ls()
```

```
# [1] "a"    "am"   "car"  "cyl"  "disp" "hp"
```

```
rm(cyl, disp, hp, am)
ls()
```

```
# [1] "a"   "car"
```

Now these vectors are no longer present in our workspace.

It is useful to know that many R functions (`lm()` for one) will accept a `data` argument - so rather than `lm(car$hp~car$cyl)` we can use `lm(hp~cyl,data=car)`. When we specify more complex models, this is very useful. Another approach is to use the function `with()` - the basic syntax is `with(some-data-frame, do-something)` - e.g. `with(car,plot(cyl,hp))`.

7.2.1 Indexing Data Frames

Since our data `car` is a *2-dimensional* object, we ought to use 2 indices. Using the incorrect number of indices can either cause errors or cause unpleasant surprises. For example, `car[,4]` will return the 4th column, as will `car$am` or `car[[4]]`. However `car[4]` will also return the 4th column. If you had intended the 4th row (`car[4,]`) and forgotten the comma, this could cause some surprises.

```
car[[4]]
```

```
#  [1] manual manual manual auto   auto   auto   auto   auto
#  [9] auto   auto   auto   auto   auto   auto   auto   auto
# [17] auto   manual manual manual auto   auto   auto   auto
```

```
# [25] auto   manual manual manual manual manual manual manual
# Levels: auto manual
```

```
head(car[4])
```

```
#    trans
# 1 manual
# 2 manual
# 3 manual
# 4   auto
# 5   auto
# 6   auto
```

However, if we use a single index greater than the number of columns in a data frame, R will throw an error that suggests we have selected *rows* but not columns.

```
car[5]
```

```
# Error in `[.data.frame`(car, 5): undefined columns selected
```

Similarly, if we try to call for 2 indices on a one-dimensional object (vector) we get an "incorrect number of dimensions".

```
car$hp[2, 3]
```

```
# Error in car$hp[2, 3]: incorrect number of dimensions
```

In my experience, these are rather common errors (at least for me!), and you should recognize them.

The function `subset()` is very useful for working with dataframes, since it allows you to extract data from the dataframe based on multiple conditions, and it has an easy to read syntax. For example, we can extract all the records of the `faithful` data with eruptions less than 3 minutes long (`summary()` used here to avoid spewing data over the page).

```
data(faithful)
summary(subset(faithful, eruptions <= 3))
```

```
#    eruptions        waiting
#  Min.   :1.600   Min.   :43.00
#  1st Qu.:1.833   1st Qu.:50.00
#  Median :1.983   Median :54.00
#  Mean   :2.038   Mean   :54.49
#  3rd Qu.:2.200   3rd Qu.:59.00
#  Max.   :2.900   Max.   :71.00
```

7.3 Attaching data

Many R tutorials will use the function `attach()` to *attach* data to the search path in R. This allows us to call variables by name. For example, in this case we have our data frame `car`, but to get the data in `hp` we need to use `car$hp` - any function that calls `hp` directly won't work - try `mean(hp)`. If we use `attach(car)` then typing `hp` gets us the data, and function calls like `mean(hp)` will now work. There are (in my experience) 2 problems with this:

A) When attaching data, R makes copies of it, so if we change `hp`, the *copy* is changed, but the original data, `car$hp` isn't changed unless we explicitly assign it to be changed - i.e. `hp[2]=NA` is *not the same* as `car$hp[2]=NA`. Read that again - `hp` is *not necessarily* the same as `car$hp`! THIS IS A VERY GOOD REASON NOT TO ATTACH DATA.

```
attach(car)
mean(hp)
```

```
# [1] 146.6875
```

```
hp[1] <- 500
hp[1] == car$hp[1]
```

```
# [1] FALSE
```

B) In my experience it is not uncommon to have multiple data sets that have many of the same variable names (e.g. `biomass`). When attaching, these conflicting names cause even more confusion. For example, if we had *not* removed the vectors `cyl, disp, and hp` above, then when we try `attach(car)` R will give us this message:

```
The following object is masked _by_ .GlobalEnv:

    cyl, disp, hp
```

For these reasons I view `attach()` as a convenience for *demonstration* of R use, and not as a "production" tool. I do not use (or only very rarely) `attach()`, and *when I do I am sure to use* ***`detach()`*** as soon as I am done with the data.

7.4 Changing Data Frames

Having imported or created a data frame it is likely that we may want to *alter* it in some way. It is rather simple to remove rows or columns by indexing - `car<-car[-31,]` will *remove* the 31st row of the data and assign the data to its previous name. Similarly `car[,-4]` would remove the 4th column (though here the changed data was not assigned).

It is also very simple to add new columns (or rows) to a data frame - simply index the row (or column) `n+1`, where `n` is the number of rows (or columns).

Alternately, just specifying a new name for a variable can create a new column. Here we'll demonstrate both - to calculate a new variable, displacement per cylinder, we first need cylinders as numeric. We'll use the 'approved' method of converting a factor to numeric - indexing the levels (see Chapter 2).

```
car[, 5] <- as.numeric(levels(car$cyl)[car$cyl])
names(car)
```

```
# [1] "cyl"   "disp"  "hp"    "trans" "V5"
```

```
names(car)[5] <- "cyl.numeric"
```

Our data set now has 5 columns, but until we give the new variable a name it is just `"V5"`, for 'Variable 5'. Let's calculate displacement per cylinder:

```
car$disp.per.cyl <- car$disp/car$cyl.numeric
names(car)
```

```
# [1] "cyl"          "disp"         "hp"           "trans"
# [5] "cyl.numeric"  "disp.per.cyl"
```

This method of creating a new variable is easier because we don't have to bother about the variable name, or about which column in will occupy. Had we used a numeric index of 5, we would overwrite the value in that column.

Sometimes we might wish to combine 2 data frames together. We can do this using `cbind()` and `rbind()` (for *column* -wise and *row* -wise binding respectively). The dataset `mtcars` contains several variables that are not in our data frame `cars`. We'll use `cbind()` to combine the 2 data sets.

```
data(mtcars)  # load the data
names(mtcars)  # cols 1,5:8,10:11 not in our data
```

```
#  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"
#  [9] "am"   "gear" "carb"
```

```
dim(car)
```

```
# [1] 32  6
```

```
car <- cbind(car, mtcars[, c(1, 5:8, 10:11)])
dim(car)
```

```
# [1] 32 13
```

```
head(car)
```

```
#                   cyl disp  hp  trans cyl.numeric disp.per.cyl
# Mazda RX4           6 2.62 110 manual           6    0.4366667
# Mazda RX4 Wag       6 2.62 110 manual           6    0.4366667
# Datsun 710          4 1.77  93 manual           4    0.4425000
# Hornet 4 Drive      6 4.23 110   auto           6    0.7050000
```

```
# Hornet Sportabout   8 5.90 175   auto           8    0.7375000
# Valiant             6 3.69 105   auto           6    0.6150000
#                    mpg drat    wt  qsec vs gear carb
# Mazda RX4         21.0 3.90 2.620 16.46  0    4    4
# Mazda RX4 Wag     21.0 3.90 2.875 17.02  0    4    4
# Datsun 710        22.8 3.85 2.320 18.61  1    4    1
# Hornet 4 Drive    21.4 3.08 3.215 19.44  1    3    1
# Hornet Sportabout 18.7 3.15 3.440 17.02  0    3    2
# Valiant           18.1 2.76 3.460 20.22  1    3    1
```

Note that the row names from `mtcars` have followed the variables from that data frame. A couple of observations about using adding to data frames: * *Don't use `cbind()` if the rows don't correspond!*- we'll see how to use `merge()` (Chapter 10) which is the right tool for this situation. (Similarly don't use `rbind()` if the columns don't correspond). * `cbind()` and `rbind()` are rather slow - don't use them inside a loop! * If you are writing a loop it is far more efficient to make space for your output (whether in a new data frame or by adding to one) *before* the loop begins, adding a row to your data frame in each iteration of a loop will slow your code down.

7.4.1 EXTRA: Comments

There is an attribute of data frames that is reserved for comments. The function `comment()` allows one to set this. `comment(car)` will return `NULL` because no comment has been set, but we can use the same function to set comments.

```
comment(car) <- "A data set derived from the mtcars dataset. Displacement is in liter
```

Now we have added a comment to this dataset, and `comment(car)` will retrieve it.

7.5 Exercises

1) Use the `mtcars` data (`data(mtcars)`) to answer these questions (if you get confused, review the bit on logical extraction in Chapter 1):
a) Which rows of the data frame contain cars that weigh more than 4000 pounds (the variable is `wt`, units are 1000 pounds).
b) Which cars are these? (*Hint:* since rows are named by car name, use `row.names()`).
c) What is the mean displacement (the variable is`disp`, units are inches^3) for cars with at least 200 horsepower (`hp`)?
d) Which car has the highest fuel economy (`mpg`)?
e) What was the fuel economy for the Honda Civic?

2) Using the `mtcars` data create a new variable for horsepower per unit weight (`hp/wt`). Is this a better predictor of acceleration (`qsec`; seconds to complete

a quarter mile) than raw horsepower? (Hint - check out correlations between these variables and acceleration, or fit regressions for both models).

3) Use the function `subset()` to return the cars with 4 cylinders and automatic transmissions (`am = 0`). (*Hint:* use "&" for logical "AND"; see `?Logic` and select `Logical Operators`).

Chapter 8

Importing Data

Getting your data into R

8.1 Introduction

Now that we understand *how* data frames function in R we'll see how they can be created by importing data. Importing data into R can be a difficult process, and many R learners get stuck here and give up (I quit here several times myself!). In order to avoid getting stuck, we'll consider some of the problems that can arise. While this is among the shortest chapters in this book, I believe it is among the most important.

8.2 Importing Data

The most universal way to import data into R is by using the function `read.table()` or one of it's derivatives (`read.delim()` and `read.csv()`) [1]. R can easily read three types of data files - *comma separated values* (.csv), *tab delimited text* (.txt), and *space delimited text* (.txt). (I've order them here in my order of preference - .csv files are usually the least troublesome). There are other ways to import data - reading data from the web, or from spreadsheet files. These are more advanced or limited to specific installations of R - for now we'll focus on the most common and useful tools.

Typical process:
1) clean up data in spreadsheet.
* a) replace empty cells with NA or other consistent string (absolutely critical for space-delimited files).

[1] If you look at `?read.table` you'll see that `read.delim()` and `read.csv()` are just versions of `read.table()` with different default values. They may be referred to as 'convenience functions'.

* b) fix column names (no spaces, no special characters, '.' and '_' are OK).
* c) save as *.csv or *.txt. (Use `File>Save as` and select `Comma separated values`) 2) Try to import the data `read.csv(...)`.
* a) if errors occur, open the .csv file in a text editor and find and remove problems (missing new lines, tabs, spaces or spurious quotation marks are common culprits) and repeat 1c) and 2). For this step is especially helpful to have the ability to see "invisible" characters - good text editors [2] will do that.
3) Check the data - `names()`, `dim()`, and `summary()` are my favorite tools, but `str()` works also.

8.2.1 On Using the "Working directory"

R wants to know where to save files and where to find files. When saving or opening files you can specify the complete file path (e.g "C:/Documents and Settings/EssentialR/my-file.R"), but this is not usually necessary. R will look for (or create) files by name in the *working directory*, which is just R speak for the folder you choose to have R use. Use `getwd()` to see what your working directory is. To change it use `setwd("file/path/here")`. Alternately, most R GUIs have a tool to do this **-** in RStudio go to "Session>Set Working Directory" and select the folder you want to use. The full command will show up in the console - it is a good idea to copy and paste this into your editor - now when you save your code, you have a note of where the working directory for that code is.

Note From here on, these notes assume that you are using the "Code Files" folder in your "EssentialR" directory as the working directory. Examples point to files in the "Data" directory in "EssentialR", so file paths will resemble `../Data/some-file.csv`; the `../Data/` means "go up one level and find the folder called Data".

8.3 An Example

First we'll examine this data in Excel. Open the file "W101-2010.xls". First we'll make sure there are no empty cells (use `NA`) for empty cells. Next we'll delete unnecessary or partial columns. Finally we'll save it as a .csv file. Now we can import it using the following code - set the file path

```
my.data <- read.csv("~/Dropbox/R_Class/EssentialR/Data/W101-2010.csv",
    comm = "#")
my.data <- read.csv("../Data/W101-2010.csv", comm = "#")
my.data <- read.csv(file.choose(), header = TRUE)  # point the file selection dialog
my.data <- read.table("../Data/W101-2010.csv", header = TRUE, sep = ",",
    quote = " \" ")
```

[2] A good text editor is a wonderful tool. For Windows Notepad++ is the best I know of, but there may be better. On OSX I like Text Wrangler, on Linux Gedit or Geany work pretty well.

These are four nearly equivalent ways to read the same data! The first three are identical except for the way the file path is specified. Note that while the first way is a bit more cumbersome, it does have the advantage of keeping a full record of where the file came from. If you use the second form, it might be a good idea to include a note of the working directory (`getwd()` works for this) in your code. If you use the third, you really should note what the file was, or else when you come back to this in a year you won't know what file you read in (ask me how I know!). Also, note that the third method will fail to import correctly if there are comments in the data file, because we have not told R how to recognize comments.

The fourth method uses `read.table()` rather than `read.csv()`, so several other arguments need to be changed:
1) We don't need to specify a comment character, as "#" is default 2) We need to specify the comma separator (`sep=","`) so each row is parsed into 7 columns as indicated by the location of commas in the text. 3) We need to specify the `header=` argument, because we want the variable names to be included. 4) We need to specify the `quote=` argument - the default is `quote="\'""`, which means either a `'` or a `"` can be interpreted as a quote. We replaced the default with `quote="\""`, meaning that a `'` *won't* be treated as a quote. It is instructive to see what happens if we don't specify this - instead of 270 rows of data, we get 107 - there are a few records here that include an `'`, and if R thinks `'` is a quote, all text between one `'` and the next `'` is treated as a quote, and so all separators (commas) and new line characters are ignored. In the `quote="\""`, the enclosing `" "` are there because R needs a quoted string for the argument. The `\` is an escape character, so the `"` that follows it won't be treated as the *closing* `"`.

8.4 An Easier Way (with Caveats!)

RStudio has a nice built in data import tool. In the "Environment" browser toolbar (upper right pane) there is an icon for `Import Dataset` that features a rather nice dialog for importing data from Text files, from Excel, and from a few other file formats. Importantly this GUI tool writes the code for the action it completes, so it is trivial to copy the code into your document in the editor (so that you can document your work or compile your HW) and add any further arguments or code comments.

Most recent versions of RStudio allow you to specify whether to use `readr` or `base` for importing from a text file, so I encourage use of the `Import Dataset` dialog with Base R [3].

If you click on `Import Dataset>From Text (base)` and navigate to the `W101-2010.csv` file in the `EssentialR/Data` folder, the dialog brings up an import and preview tool.

[3]This will work unless you have massive datasets - for very large data, `read_table()` will be preferable - `readr` creates `tibbles` which are a variant of the data.frame that will work on

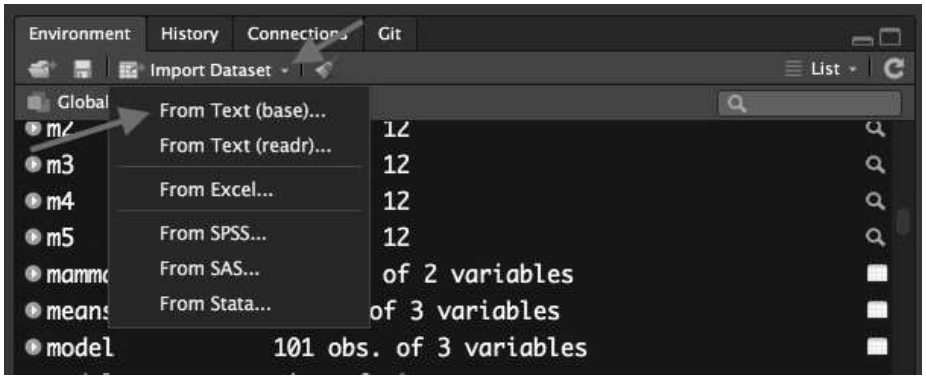


Figure 8.1: ImportDataSet

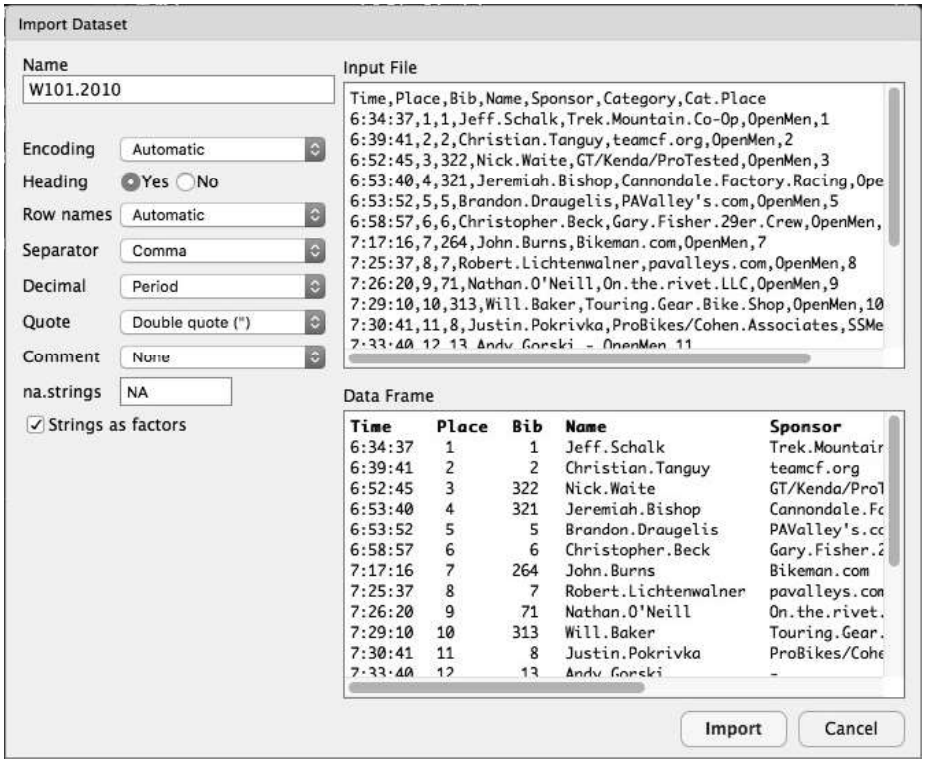


Figure 8.2: ImportDataPreview

You get a preview of the raw input file (top) and the way it will be converted to a data.frame (bottom). On the left are multiple arguments for `read.csv()`, so you can control things like `header=`, `sep=`, `quote=`, ect. This is very useful b/c you can easily see *how* changing these arguments will change how the data is imported. (For example, see what happens if you change `sep=` from comma to something else).

I held off presenting this until later in the chapter for three reasons:
1) I want to be sure you understand how `read.table()` works, because the troubleshooting described below may apply when using the GUI also.
2) Second because recent versions of RStudio default to the function `read_table()` (from the package `readr`) rather than `read.table()`. This can cause some unexpected results in the way factors are imported (`read_table()` imports strings to character rather than factor) [4].

3) When you use the `Import Dataset` dialog in RStudio, the code created includes `View(my.data)`, which gives you a preview in RStudio. This is a great way to check (though I would still use `summary(my.data)` as well!), but if you paste the code into your .R file and try to compile it, it will choke on `View()`. This is because `View()` opens a preview in RStudio, but the compilation is not really "in" RStudio.

8.5 Importing Other Data Types

It is possible to import data files (.csv, etc) from websites, either via ftp or http, simply by replacing the file name with the URL. Note that this will not work with https sites, limiting its usefulness somewhat.

The package `googlesheets` provides tools to read and write data from googlesheets. This is pretty useful.

The packages `xlsx` and `readxl` allow for data to be imported from Microsoft Excel files, though it is worth noting that now one also has to take note of which *worksheet* is imported as well as which file.

There are also tools to allow data files from SAS, SPSS, etc `-` you can find information via Rseek.org.

There are also tools to import data from databases - see this page on Quick R for more information.

Finally note that `write.table()` functions to write R objects to text files, so your can export data as well. See `?write.table` for more details.

---

much larger data sets.

[4] If you use the GUI to import data and you have factors in your data, you can just convert your character variables to factors with `as.factor()` - as long as there area small number of variables to convert this is not too unhandy. Alternately you can just use `Import Dataset>From Text (base)` and `read.table()` will be used.

8.6 Some Typical Problems

I've already noted a couple of common issues (specifying the correct separator, header, and quote characters). Here I have a couple of examples to show what to look for when data import doesn't work.

> **Note** Make sure your working directory is set to your "Code Files" folder.

```
setwd("~/Dropbox/R class/EssentialR/Code Files")
```

```
my.data <- read.table("../Data/Ex1.txt", header = TRUE)
dim(my.data)
```

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

```
summary(my.data)
```

```
#      MeanDM           DMn         Jday            DMse
#  Min.   :380.0   Min.   :8   Min.   :117.0   Min.   :27.66
#  1st Qu.:610.8   1st Qu.:8   1st Qu.:130.2   1st Qu.:29.31
#  Median :673.8   Median :8   Median :137.0   Median :31.65
#  Mean   :689.2   Mean   :8   Mean   :137.2   Mean   :39.50
#  3rd Qu.:794.1   3rd Qu.:8   3rd Qu.:146.0   3rd Qu.:42.78
#  Max.   :984.0   Max.   :8   Max.   :155.0   Max.   :71.01
```

```
my.data <- read.delim("../Data/Ex1.txt", header = TRUE)
dim(my.data)
```

```
# [1] 7 4
```

```
summary(my.data)
```

```
#             MeanDM        DMn             Jday
#  # mean biomass:1   8       :6   117        :1
#  380            :1   sample n:1   128        :1
#  590            :1                137        :2
#  673.25         :1                149        :1
#  674.25         :1                155        :1
#  834            :1                day of year:1
#  984            :1
#             DMse
#  27.664       :1
#  28.663       :1
#  31.262       :1
#  32.033       :1
#  46.363       :1
#  71.014       :1
#  SE of biomass:1
```

These two are different even though the same data was read - why? **Hint** Look at ?read.table and note the default settings for `comment.character`.

Occasionally there are small problems with the file that you can't see in Excel. The file "Ex2.txt" has a missing value with no NA.

```
## my.data <- read.table("../Data/Ex2.txt", header = TRUE)
# produces an error
my.data <- read.table("../Data/Ex2.txt", header = TRUE, sep = "\t")
# no error message, *BUT* --->
dim(my.data)
```

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

```
summary(my.data)
```

```
#      MeanDM           DMn          Jday          DMse
#  Min.   :380.0   Min.   :8   Min.   :117   Min.   :27.66
#  1st Qu.:610.8   1st Qu.:8   1st Qu.:137   1st Qu.:29.31
#  Median :673.8   Median :8   Median :137   Median :31.65
#  Mean   :689.2   Mean   :8   Mean   :139   Mean   :39.50
#  3rd Qu.:794.1   3rd Qu.:8   3rd Qu.:149   3rd Qu.:42.78
#  Max.   :984.0   Max.   :8   Max.   :155   Max.   :71.01
#                              NA's   :1
```

```
# but an NA was introduced.  *Lack of errors is not proof that
# everything worked correctly*
```

Naive use of `read.table()` generates an error because the default value of `sep=" "` does not detect the missing value, which leads to one line having fewer elements than the others (thus the error). Telling R to use a tab (`sep="/t"`) cures this problem because the missing value can now be detected, but there is a missing value in the data (which is correct in this case).

We could just use `read.delim()` since it looks for tab-delimiters, but we may need to specify the `comment.char` argument.

```
my.data <- read.delim("../Data/Ex2.txt", header = TRUE)
head(my.data)
```

```
#           MeanDM      DMn         Jday          DMse
# 1 # mean biomass sample n day of year SE of biomass
# 2            380        8          117        28.663
# 3         674.25        8          137        31.262
# 4            590        8                     27.664
# 5            834        8          149        46.363
# 6         673.25        8          137        32.033
```

```
my.data <- read.delim("../Data/Ex2.txt", header = TRUE, comment.char = "#")
my.data <- read.delim("../Data/Ex2.txt", header = TRUE, comm = "#")
```

```
dim(my.data)
```

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

```
summary(my.data)
```

```
#      MeanDM           DMn          Jday          DMse
#  Min.   :380.0   Min.   :8   Min.   :117   Min.   :27.66
#  1st Qu.:610.8   1st Qu.:8   1st Qu.:137   1st Qu.:29.31
#  Median :673.8   Median :8   Median :137   Median :31.65
#  Mean   :689.2   Mean   :8   Mean   :139   Mean   :39.50
#  3rd Qu.:794.1   3rd Qu.:8   3rd Qu.:149   3rd Qu.:42.78
#  Max.   :984.0   Max.   :8   Max.   :155   Max.   :71.01
#                              NA's   :1
```

Can you see why the `comm="#"` was added?

Another common problem is adjacent empty cells in Excel - an empty cell has had a value in it that has been deleted, and is not the same as a blank cell [5] .These will create an extra delimiter (tab or comma) in the text file, so all rows won't have the same number of values. A really good text editor will show you this kind of error, but when in doubt reopen your .csv file in Excel, copy *only the data* and paste it into a new tab and re-save it as a .csv file. The file "Ex3.txt" includes an error like this. Note that when it is saved as a .csv file this error is visible - this is why .csv files are preferred.

Another problem to watch for is leading quotes - in some cases Excel decides that a line that is commented is actually text and wraps it in quotes. This is invisible in Excel, so you don't know it has happened. When read into R, the leading " causes the comment character (by default #) to be ignored. You can usually diagnose this if you open the .csv or .txt file in a text editor rather than in Excel.

```
my.data <- read.table("../Data/Ex3.txt", header = TRUE)
## my.data <- read.table("../Data/Ex3.txt", header = TRUE, sep = "\t")  # produces an
my.data <- read.delim("../Data/Ex3.txt", header = TRUE, comment.char = "#")  # no err
head(my.data)  # but wrong
```

```
#        MeanDM DMn   Jday DMse
# 380         8 117 28.663   NA
# 674.25      8 137 31.262   NA
# 590         8 128 27.664   NA
# 834         8 149 46.363   NA
# 673.25      8 137 32.033   NA
# 984         8 155 71.014   NA
```

The first line did not produce an error, since the separator is " " (white space).

[5] This is one example of why you should use R whenever possible!

The second line produced an error because of the extra tab. The third line did not give an error message but the data was not imported correctly - the extra tab created a 5th column (with all NAs), but the 4 names were assigned to the last 4 columns, and the values of MeanDM were used for row.names.

Note If you have numeric data that has thousands separators (e.g. 12,345,678) then you will run into trouble using .csv files (with comma separators). There are several ways to address this problem, but I think the easiest is to change the number format in Excel before creating the .csv file. To do this highlight the cells and choose "number" as the format.

8.7 Exercises

1) Find (or invent) some data (not from the "Data" directory supplied with EssentialR) and import it into R. (It is not a bad idea to include a commented line with units for each variable in your .txt or.csv file). a) What did you have to do to "clean it up" so it would read in? b) Are you satisfied with the console output of `summary(yourdata)`? Did all the variables import in the way (format) you thought they should? c) Include the output of `summary(yourdata)` and `head(yourdata)`.

2) The spreadsheet "StatesData.xls" located in the `Data` directory in your EssentialR folder contains some (old) data about the 50 US states, and includes a plot with a regression line. Clean this data up and import it into R. You should be able to fit a regression that mimics the plot in the spreadsheet. What is the p-value for the slope in this regression?

Chapter 9

Manipulating Data

An introduction to data wrangling

9.1 Introduction

Often you find that your data needs to be reconfigured in some way. Perhaps you recorded some measurement in two columns, but you realize it really should be a single variable, maybe the total or mean of the two, or a single value conditioned on another value. Or perhaps you want to make some summary figures, like barplots, that need group means. Often you might be tempted to do this kind of work in Excel because you already know how to do it. However, if you already have the data in R, it is probably faster to do it in R! Even better, when you do it in R it is really reproducible in a way that it is not in Excel.

9.2 Summarizing Data

Frequently we want to calculate data summaries - for example we want to plot means and standard errors for several subsets of a data set [1]. R has useful tools that make this quite simple. We'll look first that the `apply()` family of functions, and then at the very useful `aggregate()`. Fist we'll demonstrate with some data from a study where beans were grown in different size containers [2]. In this study bean plants were grown in varying sized pots (`pot.size`) and either one or two doses of phosphorus (`P.lev`), resulting in varying concentrations of phosphorus (`phos`). Root length and root and shoot biomass were measured.

[1] Excel users will think "Pivot Table".

[2] The data are a simplified form of that reported in: Nord, E. A., Zhang, C., and Lynch, J. P. (2011). Root responses to neighboring plants in common bean are mediated by nutrient concentration rather than self/non-self recognition. Funct. Plant Biol. 38, 941–952.

```
beans <- read.csv("../Data/BeansData.csv", header = TRUE, comm = "#")
dim(beans)
```

```
# [1] 24  8
```

```
summary(beans)
```

```
#     pot.size       phos       P.lev  rep   trt       rt.len
#  Min.   : 4   Min.   : 70.0   H:12   A:6   a:4   Min.   :146.2
#  1st Qu.: 4   1st Qu.:105.0   L:12   B:6   b:4   1st Qu.:243.3
#  Median : 8   Median :175.0          C:6   c:4   Median :280.4
#  Mean   : 8   Mean   :192.5          D:6   d:4   Mean   :301.3
#  3rd Qu.:12   3rd Qu.:210.0                e:4   3rd Qu.:360.3
#  Max.   :12   Max.   :420.0                f:4   Max.   :521.7
#      ShtDM             RtDM
#  Min.   :0.5130   Min.   :0.4712
#  1st Qu.:0.8065   1st Qu.:0.6439
#  Median :1.0579   Median :0.7837
#  Mean   :1.1775   Mean   :0.8669
#  3rd Qu.:1.3159   3rd Qu.:0.9789
#  Max.   :2.7627   Max.   :1.7510
```

The "apply" family of functions are used to "do something over and over again to a subset of some data" (*apply* a function to the data in R-speak). For example we can get means for columns of data:

```
apply(X = beans[, 6:8], MARGIN = 2, FUN = mean, na.rm = TRUE)
```

```
#      rt.len       ShtDM        RtDM
# 301.3179167   1.1774750   0.8668583
```

Here we have applied the function `mean()` to the columns (`MARGIN=2`) 6,7,8 (`6:8`)of `beans` (columns 3,4, and 5 are factors, so`mean()` will give an error, so `apply(X=beans,MARGIN=2,FUN=mean,na.rm=TRUE)` will fail).

Note that we've also specified `na.rm=TRUE` - this is actually an argument to `mean()`, not to `apply()`. If you look at `?apply` you'll find `...` among the arguments. This refers to arguments that are passed to the function specified by `FUN`. Many R functions allow `...` arguments, but they are initially confusing to new users.

In this case there is no missing data, but it is a worthwhile exercise to make a copy of the beans data and introduce an `NA` so you know what happens when there are missing values in the data.

Often we want to summarize data to get group means, for example we want the means for each treatment type.

```
tapply(beans$rt.len, INDEX = list(beans$trt), FUN = mean, na.rm = TRUE)
```

```
#        a        b        c        d        e        f
# 255.2925 400.2000 178.8750 436.2800 226.7575 310.5025
```

In this case, it was easy because there was a variable (trt) that coded all the treatments together, but we don't really need it:

```
tapply(beans$rt.len, list(beans$pot.size, beans$P.lev), mean, na.rm = TRUE)
```

```
#           H        L
# 4  400.2000 255.2925
# 8  436.2800 178.8750
# 12 310.5025 226.7575
```

This gives us a tidy little table of means [3]. If we just wanted a more straightforward list we can use `paste()` to make a *combined factor*.

```
tapply(beans$rt.len, list(paste(beans$pot.size, beans$P.lev)), mean,
    na.rm = TRUE)
```

```
#     12 H     12 L      4 H      4 L      8 H      8 L
# 310.5025 226.7575 400.2000 255.2925 436.2800 178.8750
```

Often we really want to get summary data for multiple columns. The function `aggregate()` is a convenience form of apply that makes this trivially easy.

```
aggregate(x = beans[, 6:8], by = list(beans$pot.size, beans$phos),
    FUN = mean, na.rm = TRUE)
```

```
#   Group.1 Group.2   rt.len    ShtDM     RtDM
# 1      12      70 226.7575 0.823000 0.683700
# 2       8     105 178.8750 0.797575 0.599950
# 3      12     140 310.5025 1.265075 0.958225
# 4       4     210 255.2925 0.944375 0.773750
# 5       8     210 436.2800 1.301950 1.000125
# 6       4     420 400.2000 1.932875 1.185400
```

Notice that `by` must be specified as a list when using variable names, and the output lists `Group.1` and `Group.2` rather than the variable names. If we use column numbers we get nicer output and avoid the need to use `list()`. We'll also extract the standard deviations for each group.

```
aggregate(x = beans[, 6:8], by = beans[c(1, 2)], FUN = mean, na.rm = TRUE)
```

```
#   pot.size phos   rt.len    ShtDM     RtDM
# 1       12   70 226.7575 0.823000 0.683700
# 2        8  105 178.8750 0.797575 0.599950
# 3       12  140 310.5025 1.265075 0.958225
# 4        4  210 255.2925 0.944375 0.773750
# 5        8  210 436.2800 1.301950 1.000125
```

[3] If we were summarizing by three variable rather than two we would get 3-d matrix.

```
# 6        4  420 400.2000 1.932875 1.185400
```

```
aggregate(beans[, 6:8], beans[c(1, 2)], sd, na.rm = TRUE)
```

```
#   pot.size phos    rt.len     ShtDM      RtDM
# 1       12   70  51.51115 0.2491654 0.2204654
# 2        8  105  50.64085 0.2039078 0.1021530
# 3       12  140  34.80957 0.3835852 0.2774636
# 4        4  210  32.37071 0.2066067 0.1503608
# 5        8  210 100.21786 0.3904458 0.2689229
# 6        4  420  86.66397 0.8402876 0.5187955
```

What if we want to specify a `FUN` that doesn't exist in R? Simple - we write our own.

```
aggregate(beans[, 6:8], beans[c(1, 2)], function(x) (sd(x, na.rm = TRUE)/(length(x) -
    sum(is.na(x)))^0.5))
```

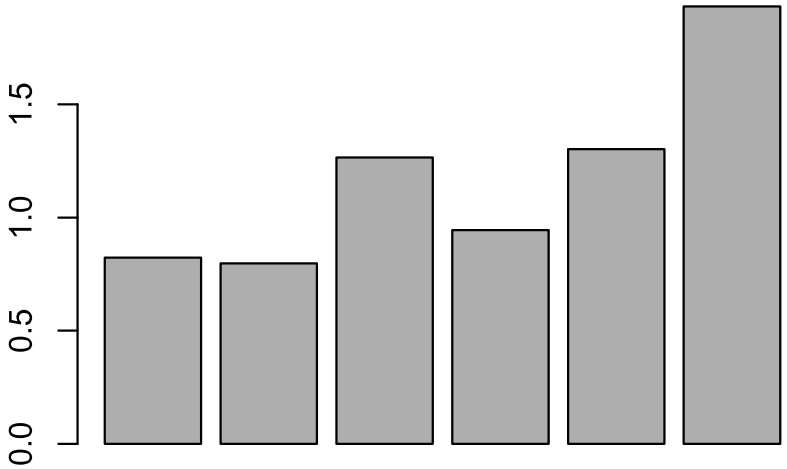
```
#   pot.size phos   rt.len     ShtDM       RtDM
# 1       12   70 25.75557 0.1245827 0.11023268
# 2        8  105 25.32042 0.1019539 0.05107649
# 3       12  140 17.40478 0.1917926 0.13873179
# 4        4  210 16.18536 0.1033033 0.07518041
# 5        8  210 50.10893 0.1952229 0.13446147
# 6        4  420 43.33198 0.4201438 0.25939775
```

Here we've defined the function (`function(x)`) on the fly. We could also begin by first defining a function: `SE=function(x)` followed by the definition of the function, and then just use `FUN=SE` in our call to aggregate.

This is a good example of something I have to look up in my "R Tricks"" file. It is also an example of how lines of R code can get really long (and why auto balancing of parentheses is really nice)! Adding spaces is OK

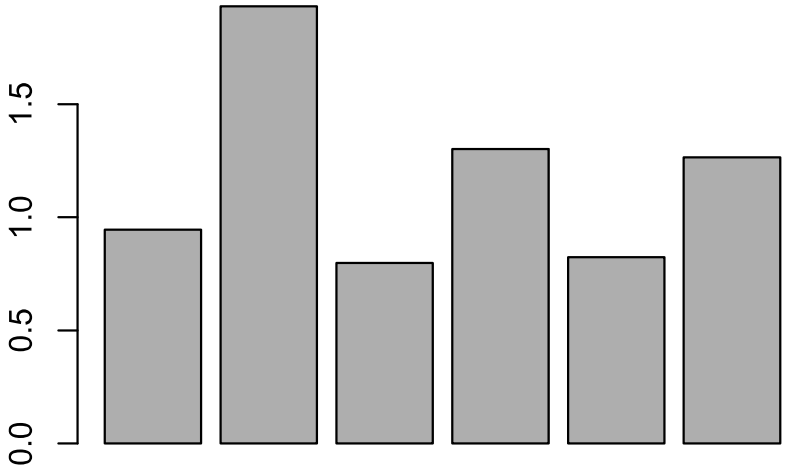
It is easy to see how useful summaries like this could be - let's make a plot from this.

```
beans.means <- aggregate(beans[, 6:8], beans[c(1, 2)], mean, na.rm = TRUE)
barplot(beans.means$ShtDM)
```



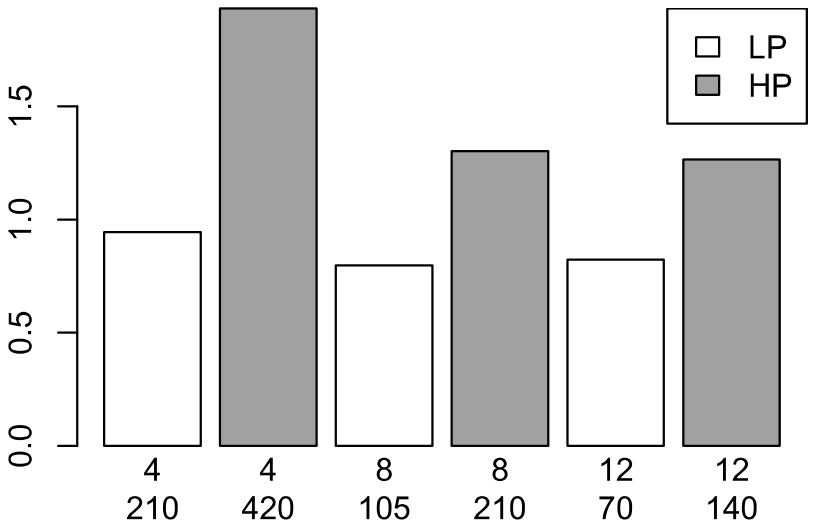
If we don't like the order of the bars here we can use the function `order()` to sort our data (`beans.means`) for nicer plotting.

```
beans.means <- beans.means[order(beans.means$pot.size, beans.means$phos),
    ]
barplot(beans.means$ShtDM)
```



Now we'll add the labels and legend - we'll discuss these in more detail in later lessons.

```
barplot(beans.means$ShtDM,col=c("white","grey70"),names.arg=
    paste(beans.means$pot.size,beans.means$phos,sep="\n"),ylab=
    "Shoot biomass (g)")
legend("topright",fill=c("white","grey70"),legend=c("LP","HP"))
```



You can see that R gives us powerful tools for data manipulation.

9.3 Reformatting Data from "Wide" to "Long"

Sometimes we record and enter data in a different format than that in which we wish to analyze it. For example, I might record the biomass from each of several replicates of an experiment in separate columns for convenience. But when I want to analyze it, I need biomass as a single column, with another column for replicate. Or perhaps I have biomass as a single column and want to break it into separate columns. The R functions `stack()` and `unstack()` are a good place to begin.

```
data(PlantGrowth)
head(PlantGrowth)
```

```
#   weight group
# 1   4.17  ctrl
# 2   5.58  ctrl
# 3   5.18  ctrl
# 4   6.11  ctrl
# 5   4.50  ctrl
# 6   4.61  ctrl
```

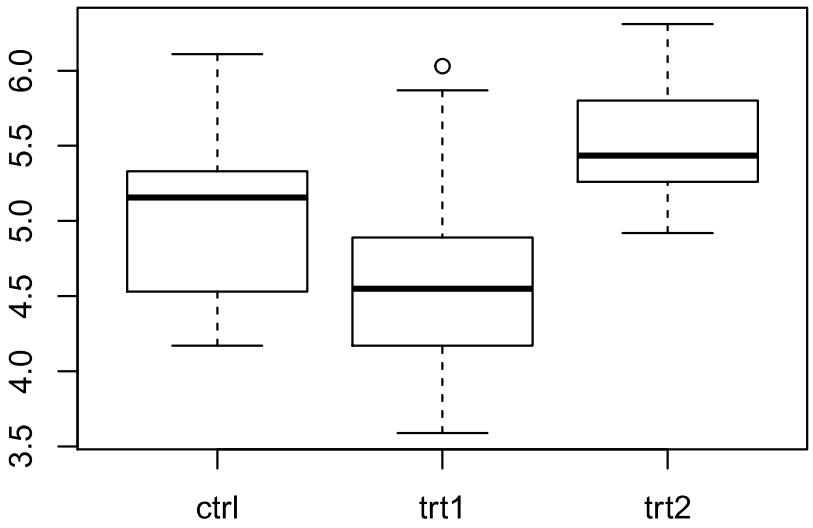
We have 2 variables: weight and group. We can use `unstack()` to make one column for each level of group.

```
unstack(PlantGrowth)
```

```
#    ctrl trt1 trt2
# 1  4.17 4.81 6.31
```

```
# 2  5.58 4.17 5.12
# 3  5.18 4.41 5.54
# 4  6.11 3.59 5.50
# 5  4.50 5.87 5.37
# 6  4.61 3.83 5.29
# 7  5.17 6.03 4.92
# 8  4.53 4.89 6.15
# 9  5.33 4.32 5.80
# 10 5.14 4.69 5.26
```

```
pg <- unstack(PlantGrowth, weight ~ group)
boxplot(pg)
```



This can be useful for plotting (though in this case `boxplot(weight~group,data=PlantGrowth)` would work equally well). We can use `stack()` to go the other way.

```
summary(stack(pg))
```

```
#      values        ind
#  Min.   :3.590   ctrl:10
#  1st Qu.:4.550   trt1:10
#  Median :5.155   trt2:10
#  Mean   :5.073
#  3rd Qu.:5.530
#  Max.   :6.310
```

```
summary(stack(pg, select = c(trt1, trt2)))
```

```
#      values        ind
#  Min.   :3.590   trt1:10
#  1st Qu.:4.620   trt2:10
```

```
#  Median :5.190
#  Mean   :5.093
#  3rd Qu.:5.605
#  Max.   :6.310
```

```
summary(stack(pg, select = -ctrl))
```

```
#      values        ind
#  Min.   :3.590   trt1:10
#  1st Qu.:4.620   trt2:10
#  Median :5.190
#  Mean   :5.093
#  3rd Qu.:5.605
#  Max.   :6.310
```

Notice that we can use the `select` argument to specify or exclude columns when we stack.

Suppose we're interested in comparing any treatment against the control with the PlantGrowth data. We've already seen how this can be done using the function `levels()`. There are three levels, but if we reassign one of them we can make 2 levels.

```
levels(PlantGrowth$group)
```

```
# [1] "ctrl" "trt1" "trt2"
```

```
PlantGrowth$group2 <- factor(PlantGrowth$group)
levels(PlantGrowth$group2) <- c("Ctrl", "Trt", "Trt")
summary(PlantGrowth)
```

```
#      weight       group     group2
#  Min.   :3.590   ctrl:10   Ctrl:10
#  1st Qu.:4.550   trt1:10   Trt :20
#  Median :5.155   trt2:10
#  Mean   :5.073
#  3rd Qu.:5.530
#  Max.   :6.310
```

```
unstack(PlantGrowth, weight ~ group2)
```

```
# $Ctrl
#  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14
#
# $Trt
#  [1] 4.81 4.17 4.41 3.59 5.87 3.83 6.03 4.89 4.32 4.69 6.31 5.12
# [13] 5.54 5.50 5.37 5.29 4.92 6.15 5.80 5.26
```

We can even use `unstack()` to split `weight` based on `group2`, but the output is different as the groups aren't balanced.

9.4 Reshape

For more sophisticated reshaping of data the package "reshape" [4] has powerful tools to reshape data in many ways, but it takes some time to read the documentation and wrap your head around how it works and what it can do.

Basically there are 2 functions here - `melt()` and `cast()` (think working with metal) - once you 'melt' the data you can 'cast' it into any shape you want. `melt()` turns a data set into a series of observations which consist of a variable and value, and `dcast()` reshapes melted data, 'casting' it into a new form. You will probably need to install `reshape` - either from the 'Packages' tab or via `install.packages("reshape")`[5]. We'll demonstrate with the built in data set `iris`, and we'll need to create a unique identifier for each case in the data set.

```
library(reshape)
data(iris)
iris$id <- row.names(iris)
head(melt(iris, id = "id"))
```

```
#   id     variable value
# 1  1 Sepal.Length   5.1
# 2  2 Sepal.Length   4.9
# 3  3 Sepal.Length   4.7
# 4  4 Sepal.Length   4.6
# 5  5 Sepal.Length     5
# 6  6 Sepal.Length   5.4
```

```
tail(melt(iris, id = "id"))
```

```
#      id variable     value
# 745 145  Species virginica
# 746 146  Species virginica
# 747 147  Species virginica
# 748 148  Species virginica
# 749 149  Species virginica
# 750 150  Species virginica
```

```
melt.iris <- melt(iris, id = c("id", "Species"))
dim(melt.iris)
```

```
# [1] 600   4
```

[4] Not to be confused with `reshape2`, by the same author. There are some differences between the original `reshape` and the newer `reshape2` - the newer version is much faster for large datasets, but does not have quite all the funcitonality of `reshape`.

[5] Do recall that you need to then *load* the package either via `library(reshape)`, or via RStudio's packages pane. Note that if you are using knitr, you will need to include the code to load the package (e.g. `library(reshape)`) in you file, since it will need to be loaded into the clean workspace where knitr evaluates the code.

```
head(melt.iris)
```

```
#   id Species     variable value
# 1  1  setosa Sepal.Length   5.1
# 2  2  setosa Sepal.Length   4.9
# 3  3  setosa Sepal.Length   4.7
# 4  4  setosa Sepal.Length   4.6
# 5  5  setosa Sepal.Length   5.0
# 6  6  setosa Sepal.Length   5.4
```

```
tail(melt.iris)
```

```
#      id   Species    variable value
# 595 145 virginica Petal.Width   2.5
# 596 146 virginica Petal.Width   2.3
# 597 147 virginica Petal.Width   1.9
# 598 148 virginica Petal.Width   2.0
# 599 149 virginica Petal.Width   2.3
# 600 150 virginica Petal.Width   1.8
```

Now instead of 150 observations with 6 variables we have 600 observations with 4 variables. We can cast this data using `cast()`. If we specify enough columns from our melted data to account for all the data, then we don't need to specify a `fun.aggregate` - a function with which to aggregate, but we can aggregate the data easily,and with more flexibility than by using aggregate:

```
cast(melt.iris, Species ~ variable, mean)
```

```
#      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
# 1     setosa        5.006       3.428        1.462       0.246
# 2 versicolor        5.936       2.770        4.260       1.326
# 3  virginica        6.588       2.974        5.552       2.026
```

```
cast(melt.iris, Species ~ variable, max)
```

```
#      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
# 1     setosa          5.8         4.4          1.9         0.6
# 2 versicolor          7.0         3.4          5.1         1.8
# 3  virginica          7.9         3.8          6.9         2.5
```

We can get our original data back.

```
head(cast(melt.iris, Species + id ~ variable))
```

```
#   Species id Sepal.Length Sepal.Width Petal.Length Petal.Width
# 1  setosa  1          5.1         3.5          1.4         0.2
# 2  setosa 10          4.9         3.1          1.5         0.1
# 3  setosa 11          5.4         3.7          1.5         0.2
# 4  setosa 12          4.8         3.4          1.6         0.2
```

```
# 5  setosa 13          4.8         3.0          1.4         0.1
# 6  setosa 14          4.3         3.0          1.1         0.1
```

But we can also do other types of reshaping. For example, what if we wanted to separate out Sepal and Petal variables for each record? We can use `strplit()` to split the stings that represent variables, e.g. "Sepal.Width".

```
head(strsplit(as.character(melt.iris$variable), split = ".", fixed = TRUE))
```

```
# [[1]]
# [1] "Sepal"  "Length"
#
# [[2]]
# [1] "Sepal"  "Length"
#
# [[3]]
# [1] "Sepal"  "Length"
#
# [[4]]
# [1] "Sepal"  "Length"
#
# [[5]]
# [1] "Sepal"  "Length"
#
# [[6]]
# [1] "Sepal"  "Length"
```

Notice that this returns a list. We'll have to use `do.call()` to call `rbind()` on the list to bind the list elements into a data frame.

```
head(do.call(rbind, strsplit(as.character(melt.iris$variable), split = ".",
    fixed = TRUE)))
```

```
#      [,1]    [,2]
# [1,] "Sepal" "Length"
# [2,] "Sepal" "Length"
# [3,] "Sepal" "Length"
# [4,] "Sepal" "Length"
# [5,] "Sepal" "Length"
# [6,] "Sepal" "Length"
```

Now this seems a bit esoteric, but we can use `cbind()` to bind these values to our melted iris data

```
melt.iris <- cbind(melt.iris, do.call(rbind, strsplit(as.character(melt.iris$variable),
    split = ".", fixed = TRUE)))
names(melt.iris)[5:6] <- c("Part", "Dimension")
head(melt.iris)
```

```
#   id Species     variable value  Part Dimension
# 1  1  setosa Sepal.Length   5.1 Sepal    Length
# 2  2  setosa Sepal.Length   4.9 Sepal    Length
# 3  3  setosa Sepal.Length   4.7 Sepal    Length
# 4  4  setosa Sepal.Length   4.6 Sepal    Length
# 5  5  setosa Sepal.Length   5.0 Sepal    Length
# 6  6  setosa Sepal.Length   5.4 Sepal    Length
```

Now we can see that we have separated the flower parts (Sepal or Petal) from the dimensions.

```
cast(melt.iris, Species ~ Dimension | Part, mean)
```

```
# $Petal
#      Species Length Width
# 1     setosa  1.462 0.246
# 2 versicolor  4.260 1.326
# 3  virginica  5.552 2.026
#
# $Sepal
#      Species Length Width
# 1     setosa  5.006 3.428
# 2 versicolor  5.936 2.770
# 3  virginica  6.588 2.974
```

```
cast(melt.iris, Species ~ Dimension + Part, mean)
```

```
#      Species Length_Petal Length_Sepal Width_Petal Width_Sepal
# 1     setosa        1.462        5.006       0.246       3.428
# 2 versicolor        4.260        5.936       1.326       2.770
# 3  virginica        5.552        6.588       2.026       2.974
```

So we can talk about the mean Length and Width, averaged over floral parts. In this case it may not make sense to do so, but it demonstrates the type of data reconfiguration that is possible.

```
cast(melt.iris, Species ~ Dimension, mean)
```

```
#      Species Length Width
# 1     setosa  3.234 1.837
# 2 versicolor  5.098 2.048
# 3  virginica  6.070 2.500
```

We can still go back to the original data by just casting the data in a different form:

```
head(cast(melt.iris, Species + id ~ Part + Dimension))
```

```
#   Species id Petal_Length Petal_Width Sepal_Length Sepal_Width
# 1  setosa  1          1.4         0.2          5.1         3.5
```

```
# 2  setosa 10          1.5         0.1         4.9         3.1
# 3  setosa 11          1.5         0.2         5.4         3.7
# 4  setosa 12          1.6         0.2         4.8         3.4
# 5  setosa 13          1.4         0.1         4.8         3.0
# 6  setosa 14          1.1         0.1         4.3         3.0
```

The package `reshape` adds important tools to the R data manipulation toolkit. While they may be a bit tricky to learn, they are very powerful. See ?cast for more examples.

9.5 Merging Data Sets

Another data manipulation task is merging two data sets together. Perhaps you have field and laboratory results in different files, and you want to merge them into one file. Here we'll use an example from the `merge()` help file.

```
authors <- data.frame(surname = c("Tukey", "Venables", "Tierney",
    "Ripley", "McNeil"), nationality = c("US", "Australia", "US",
    "UK", "Australia"), deceased = c("yes", rep("no", 4)))
books <- data.frame(name = c("Tukey", "Venables", "Tierney", "Ripley",
    "Ripley", "McNeil", "R Core"), title = c("Expl. Data Anal.", "Mod. Appl. Stat ...",
    "LISP-STAT", "Spatial Stat.", "Stoch. Simu.", "Inter. Data Anal.",
    "An Intro. to R"), other.author = c(NA, "Ripley", NA, NA, NA,
    NA, "Venables & Smith"))
authors
```

```
#    surname nationality deceased
# 1    Tukey          US      yes
# 2 Venables   Australia       no
# 3  Tierney          US       no
# 4   Ripley          UK       no
# 5   McNeil   Australia       no
```

```
books
```

```
#       name               title     other.author
# 1    Tukey    Expl. Data Anal.             <NA>
# 2 Venables Mod. Appl. Stat ...           Ripley
# 3  Tierney           LISP-STAT             <NA>
# 4   Ripley       Spatial Stat.             <NA>
# 5   Ripley        Stoch. Simu.             <NA>
# 6   McNeil   Inter. Data Anal.             <NA>
# 7   R Core      An Intro. to R Venables & Smith
```

We've created 2 small data frames for demonstration purposes here. Now we can use `merge()` to merge them.

```
(m1 <- merge(authors, books, by.x = "surname", by.y = "name"))
```

```
#    surname nationality deceased               title other.author
# 1   McNeil   Australia       no   Inter. Data Anal.         <NA>
# 2   Ripley          UK       no       Spatial Stat.         <NA>
# 3   Ripley          UK       no        Stoch. Simu.         <NA>
# 4  Tierney          US       no           LISP-STAT         <NA>
# 5    Tukey          US      yes    Expl. Data Anal.         <NA>
# 6 Venables   Australia       no Mod. Appl. Stat ...       Ripley
```

```
(m2 <- merge(books, authors, by.x = "name", by.y = "surname"))
```

```
#       name               title other.author nationality deceased
# 1   McNeil   Inter. Data Anal.         <NA>   Australia       no
# 2   Ripley       Spatial Stat.         <NA>          UK       no
# 3   Ripley        Stoch. Simu.         <NA>          UK       no
# 4  Tierney           LISP-STAT         <NA>          US       no
# 5    Tukey    Expl. Data Anal.         <NA>          US      yes
# 6 Venables Mod. Appl. Stat ...       Ripley   Australia       no
```

Notice that the order of the columns mirrors the order of columns in the function call - in the first line we asked for `authors,books` and the columns are the three columns of `authors` and the all but the first column of `books`, because that column (`name`) is the `by.y` column. If both data frames had the column `surname` we could just say `by=surname`. Notice that "R core" (`books[7,]`) is not included in the combined data frame - this is because it does not exist in *both* of them. We can override this, but `NA`s will be introduced. Also note that `by`, `by.x` and `by.y` can be vectors of more than one variable -useful for complex data sets.

```
merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)
```

```
#    surname nationality deceased               title
# 1   McNeil   Australia       no   Inter. Data Anal.
# 2   Ripley          UK       no       Spatial Stat.
# 3   Ripley          UK       no        Stoch. Simu.
# 4  Tierney          US       no           LISP-STAT
# 5    Tukey          US      yes    Expl. Data Anal.
# 6 Venables   Australia       no Mod. Appl. Stat ...
# 7   R Core        <NA>     <NA>      An Intro. to R
#       other.author
# 1             <NA>
# 2             <NA>
# 3             <NA>
# 4             <NA>
# 5             <NA>
# 6           Ripley
# 7 Venables & Smith
```

This is a good example of something that is much easier to do in R than in Excel.

9.6 More about Loops

Sometimes we want to have R do something over and over again. Often a *loop* is the simplest [6] way to do this. AS we saw earlier the general syntax for a loop in R is: `for(index) {do something}`.

The curly braces are optional if it fits on one line, but required if it goes over one line.

Here are a couple of examples:

```
for (i in 1:5) print(i)
```

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

```
x <- c(2, 5, 7, 23, 6)
for (i in x) {
    cat(paste("i^2=", i^2, "\n"))
}
```

```
# i^2= 4
# i^2= 25
# i^2= 49
# i^2= 529
# i^2= 36
```

Another example might be converting multiple numeric columns to factors. Imagine we wanted to convert columns 3,5,and 6 of a data frame from numeric to factor. We could run (nearly) the same command three times:

```
# df[,3]=factor(df[,3]); df[,5]=factor(df[,5]);
# df[,6]=factor(df[,6]);
```

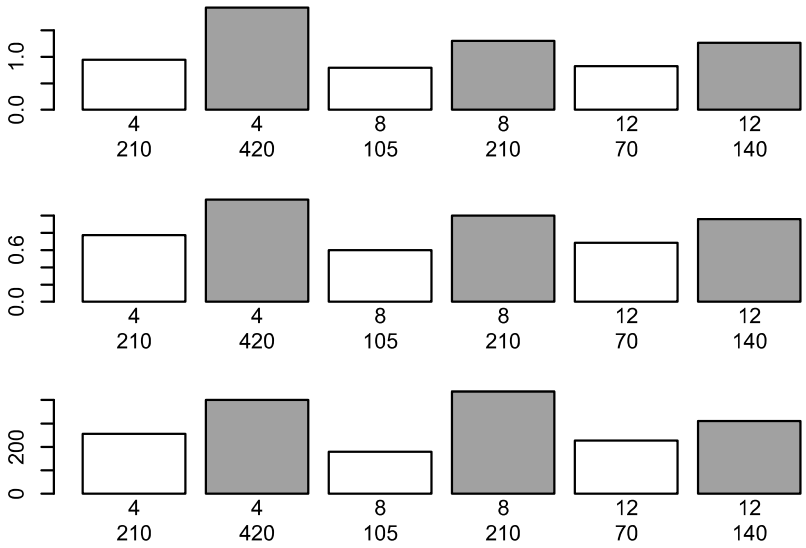
However, particularly if we have more than two columns that need to be converted it may be easier to use a loop - just use the vector of the columns to be converted as the index:

```
# for(i in c(3,5,6)) df[,i]<-factor(df[,i])
```

[6] Though possibly not the fastest - advanced R users will sometimes say that you should use an `apply()` function rather than a loop because it will run faster. This is probably only a real concern if you have very large data sets.

Loops can be used in many ways - for example the code we used to plot shoot biomass could be put in a loop to plot all the response variables in the data.

```
par(mfrow=c(3,1),mar=c(3,3,0.5,0.5))
for (p in c(4,5,3)){
  barplot(beans.means[,p],col=c("white","grey70"),
    names.arg=paste(beans.means$pot.size,beans.means$phos,sep="\n"))
 # plot the pth column of beans.means
  }
```



Obviously, this plot isn't perfect yet, but it is good enough to be useful. Note - by convention, the code *inside* a loop is indented to make it easier to see where a loop begins and ends.

9.7 Exercises

1) Load the data set "ufc" (the file is ufc.csv). This data shows diameter at breast height (Dbh) and Height for forest trees. Can you use `unstack()` to get the diameter data for white pine (`WP`)? Start by unstacking all the diameter data. Can you also get this data by logical extraction? (*Hint:* use the function `which()`. If you really only wanted the data for one species logical extraction would probably be better.)

2) For the data set ufc find the mean Dbh and Height for each species. (*Hint:* aggregate is your friend for more than one response variable.)

3) Make a barplot showing these mean values for each species. Use beside =TRUE (stacking two different variables wouldn't make sense...). (*Hint:* this will be easier if you make a new variable for the means from Q2. Look at

?barplot for the data type "height" must have - `as.matrix()` can be used to make something a matrix.)

4) The barplot in Q3 suggests a fair correlation between Dbh and height. Plot Height~DBH, fit a regression, and plot the line. What is the R^2?

Chapter 10

Working with multiple variables

Some basic tools for multivariate data

10.1 Introduction

Now that we've discussed the data frame in R, and seen how data can be imported, we can begin to practice working with multiple variables. Rather than a full introduction to multivariate methods, here we'll cover some basic tools.

10.2 Working with Multivariate Data

Working with more than two variables becomes more complex, but many of the tools we've already learned can also help us here. We'll use the `mtcars` data we referred to in the Chapters 4 & 5, but now we'll load it directly.

```
data(mtcars)
summary(mtcars)
```

```
#       mpg             cyl             disp             hp
#  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
#  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
#  Median :19.20   Median :6.000   Median :196.3   Median :123.0
#  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
#  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
#  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
#       drat             wt             qsec
#  Min.   :2.760   Min.   :1.513   Min.   :14.50
```

```
#  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89
#  Median :3.695   Median :3.325   Median :17.71
#  Mean   :3.597   Mean   :3.217   Mean   :17.85
#  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90
#  Max.   :4.930   Max.   :5.424   Max.   :22.90
#        vs               am              gear
#  Min.   :0.0000   Min.   :0.0000   Min.   :3.000
#  1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:3.000
#  Median :0.0000   Median :0.0000   Median :4.000
#  Mean   :0.4375   Mean   :0.4062   Mean   :3.688
#  3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:4.000
#  Max.   :1.0000   Max.   :1.0000   Max.   :5.000
#       carb
#  Min.   :1.000
#  1st Qu.:2.000
#  Median :2.000
#  Mean   :2.812
#  3rd Qu.:4.000
#  Max.   :8.000
```

If you look at the "Environment" tab in RStudio you should now see `mtcars` under "Data". We'll convert some of this data to factors, since as we discussed before, the number of cylinders, transmission type (and number of carburetors, V/S, and number of forward gears) aren't really continuous.

```
for (i in c(2, 8, 9, 10, 11)) {
    mtcars[, i] = factor(mtcars[, i])
}
names(mtcars)[9] <- "trans"
levels(mtcars$trans) <- c("auto", "manual")
summary(mtcars)
```

```
#       mpg         cyl          disp             hp
#  Min.   :10.40   4:11   Min.   : 71.1   Min.   : 52.0
#  1st Qu.:15.43   6: 7   1st Qu.:120.8   1st Qu.: 96.5
#  Median :19.20   8:14   Median :196.3   Median :123.0
#  Mean   :20.09          Mean   :230.7   Mean   :146.7
#  3rd Qu.:22.80          3rd Qu.:326.0   3rd Qu.:180.0
#  Max.   :33.90          Max.   :472.0   Max.   :335.0
#       drat             wt             qsec       vs
#  Min.   :2.760   Min.   :1.513   Min.   :14.50   0:18
#  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1:14
#  Median :3.695   Median :3.325   Median :17.71
#  Mean   :3.597   Mean   :3.217   Mean   :17.85
#  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90
#  Max.   :4.930   Max.   :5.424   Max.   :22.90
#     trans    gear   carb
```

```
#  auto  :19   3:15   1: 7
#  manual:13   4:12   2:10
#              5: 5   3: 3
#                     4:10
#                     6: 1
#                     8: 1
```

Notice the `for(){}` loop here; just as we saw in the last chapter this is faster than writing 5 lines of code. I prefer to have informative factor names, so we'll change that, and then check `summary()` to make sure it is all OK.

Note that we could achieve the same result using the list version of `apply()`, `lapply()`. This will run faster, but the code is slightly longer than the loop version above.

```
mtcars[, c(2, 8, 9, 10, 11)] <- lapply(mtcars[, c(2, 8, 9, 10, 11)],
    FUN = function(x) (factor(x)))
```

In the last lesson we used `table()` for two-way tables, but we can also use it for three-way tables.

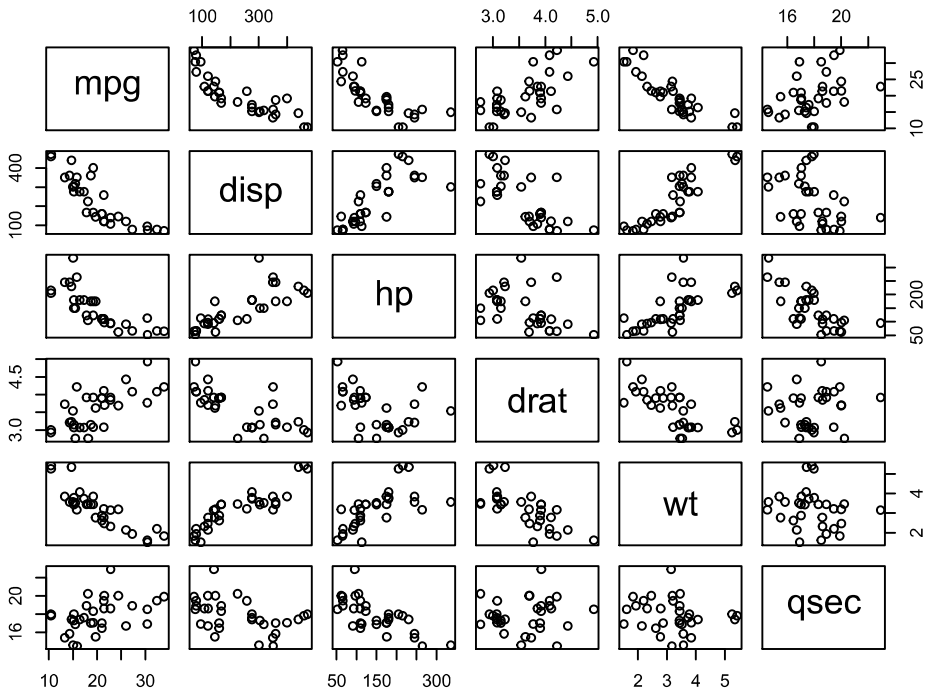
```
with(mtcars, table(carb, cyl, trans))
```

```
# , , trans = auto
#
#     cyl
# carb 4 6 8
#    1 1 2 0
#    2 2 0 4
#    3 0 0 3
#    4 0 2 5
#    6 0 0 0
#    8 0 0 0
#
# , , trans = manual
#
#     cyl
# carb 4 6 8
#    1 4 0 0
#    2 4 0 0
#    3 0 0 0
#    4 0 2 1
#    6 0 1 0
#    8 0 0 1
```

This give us bit more insight - cars with automatic transmissions don't seem to have more than 4 carburetors, but cars with manual transmissions might have as many as 8, but not more carburetors than cylinders.

Another tool that is often useful to explore multivariate data is the "pairs plot", which shows correlations for all pairs. Since we have 3 factors here, lets exclude them so we can focus on the numeric variables.

```
pairs(mtcars[, -c(2, 8, 9, 10, 11)])
```



Now we can see what patterns of correlation exist in this dataset. Fuel economy ("mpg") is relatively well correlated (negatively) with displacement, though perhaps not in a linear fashion. Quarter-mile time (qsec) is not strongly correlated with any of the other variables (but there may be a weak negative correlation with hp, which stands to reason).

10.2.1 Lattice Graphics

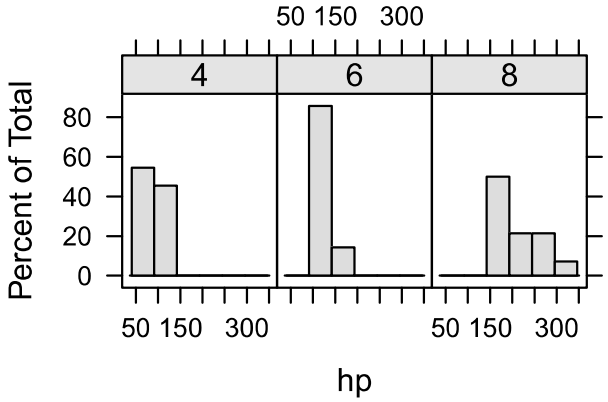
The "Lattice" package includes nice functionality for making plots conditioned on a third variable. Either use `install.packages ("lattice")` if your are connected to the internet or `install.packages (repos=NULL, type = "source", pkgs = file.choose())` and select the package file for "lattice". Remember that to use it you have to load it: `library (lattice)`

```
# Loading required package: lattice
```

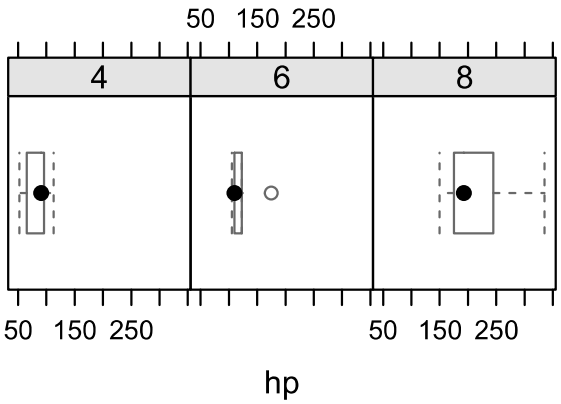
Alternately, you can go to the "Packages" tab in RStudio and click "Install Packages". Note that while you should only need to *install* a package once, you will need to *load* it (via the function `library()`) each session that you wish to use it - this is to keep R from getting to bloated.

The `lattice` package has functions that plot data *conditioned* on another variable, which is specified by the `|` operator.

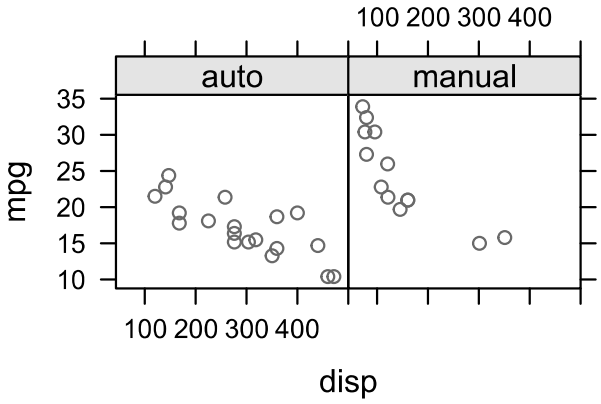
```
histogram(~hp | cyl, data = mtcars)
```




```
bwplot(~hp | cyl, data = mtcars)
```



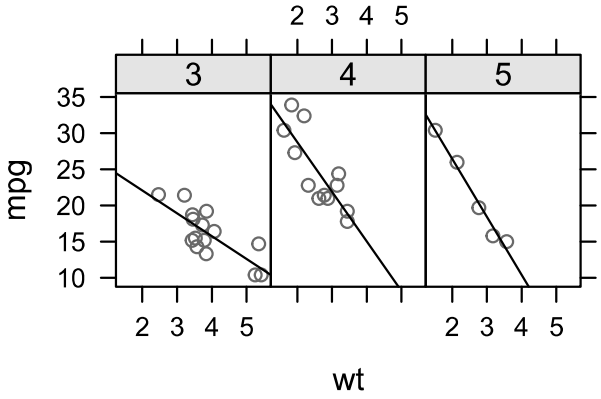

```
xyplot(mpg ~ disp | trans, data = mtcars)
```



10.2.2 EXTRA: Customizing Lattice Plots

By defining custom functions, we can customize lattice plots. Here we'll define a custom function using `panel.` functions from `lattice`. The actual definition of the function is simple, knowing the pieces needed to define this particular function is less so.

```
plot.regression = function(x, y) {
    panel.xyplot(x, y)
    panel.abline(lm(y ~ x))
}
xyplot(mpg ~ wt | gear, panel = plot.regression, data = mtcars)
```



10.3 An Example

For example let us consider some data from the pilot study on root anatomy that we looked at in the last chapter. 123 root cross sections were analyzed for a suite of 9 anatomical traits. There are several sources of variation here - samples are from different genotypes (12), grown in different media (2), and

from different locations in the root system (2).

In the last chapter we saw how to covert several columns to factors, how to change factor levels, and how to calculate new variables.

```
getwd()  ## mine is 'EssentialR/Chapters', YMMV
```

```
# [1] "/Users/enord/Dropbox/R_Class/EssentialR/Chapters"
```

```
anat <- read.table("../Data/anatomy-pilot-simple.txt", header = TRUE,
    sep = "\t")
summary(anat)
```

```
#      Gtype    media.rep     sample      Loc        RSXA.mm2
#  C      :12   r1:31     Min.   :1.000   L1:60   Min.   :0.0681
#  D      :12   R1:29     1st Qu.:1.000   L2:63   1st Qu.:0.9682
#  F      :12   r2:32     Median :2.000           Median :1.1354
#  G      :12   R2:31     Mean   :1.943           Mean   :1.1225
#  I      :12             3rd Qu.:3.000           3rd Qu.:1.2789
#  B      :11             Max.   :3.000           Max.   :1.6347
#  (Other):52
#     TCA.mm2           AA.mm2       Cort.Cell.Num
#  Min.   :0.0545   Min.   :0.0057   Min.   : 510
#  1st Qu.:0.7560   1st Qu.:0.1153   1st Qu.:1542
#  Median :0.9045   Median :0.2073   Median :1817
#  Mean   :0.8881   Mean   :0.2098   Mean   :1857
#  3rd Qu.:1.0176   3rd Qu.:0.2837   3rd Qu.:2136
#  Max.   :1.3882   Max.   :0.5084   Max.   :3331
#
#     XVA.mm2            Per.A          CellSize.1
#  Min.   :0.00240   Min.   : 1.359   Min.   :0.0000610
#  1st Qu.:0.04080   1st Qu.:15.428   1st Qu.:0.0003300
#  Median :0.04960   Median :23.258   Median :0.0004830
#  Mean   :0.05079   Mean   :22.886   Mean   :0.0006254
#  3rd Qu.:0.06070   3rd Qu.:29.313   3rd Qu.:0.0008675
#  Max.   :0.08990   Max.   :46.262   Max.   :0.0017030
#
#    CellSize.2          CellSize.3          CellSize.4
#  Min.   :0.0000680   Min.   :0.0000470   Min.   :0.0000280
#  1st Qu.:0.0003025   1st Qu.:0.0001610   1st Qu.:0.0001150
#  Median :0.0005520   Median :0.0001950   Median :0.0001390
#  Mean   :0.0007299   Mean   :0.0002052   Mean   :0.0001400
#  3rd Qu.:0.0009215   3rd Qu.:0.0002380   3rd Qu.:0.0001615
#  Max.   :0.0036640   Max.   :0.0004380   Max.   :0.0002710
#
#            Comments
#                :98
```

```
#   mc           :13
#   wried anatomy: 3
#   blur image   : 2
#   a little blur: 1
#   dull blade   : 1
#  (Other)       : 5
```

```
for (i in c(1, 3)) anat[, i] <- factor(anat[, i])
# cols 1,3 to factors
levels(anat$media.rep) <- c("R1", "R1", "R2", "R2")
# r1,2 to R1,2
anat$CellSize.avg <- rowMeans(anat[, 11:14])  # avgCellSize
anat$m.Loc <- factor(paste(anat$media.rep, anat$Loc))
# combined levels
```

We've fixed that. Now that we have estimates for average CellSize let's remove the original cell size values as well as the comments column.

```
anat <- anat[, -(11:15)]
names(anat)
```

```
#  [1] "Gtype"         "media.rep"     "sample"
#  [4] "Loc"           "RSXA.mm2"      "TCA.mm2"
#  [7] "AA.mm2"        "Cort.Cell.Num" "XVA.mm2"
# [10] "Per.A"         "CellSize.avg"  "m.Loc"
```

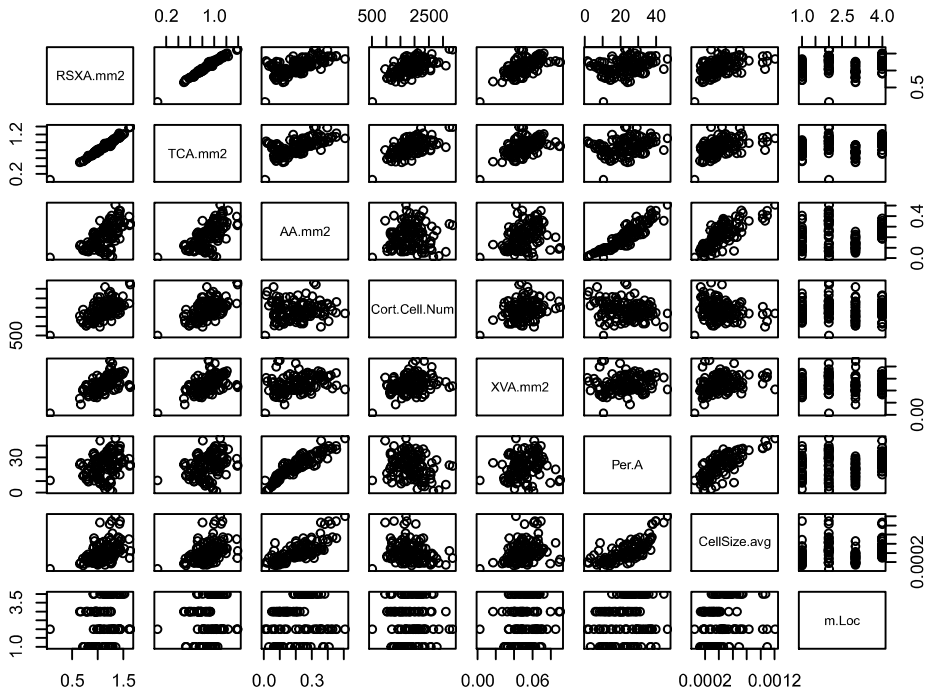
```
summary(anat)
```

```
#      Gtype    media.rep sample Loc        RSXA.mm2
#  C      :12   R1:60     1:43   L1:60   Min.   :0.0681
#  D      :12   R2:63     2:44   L2:63   1st Qu.:0.9682
#  F      :12             3:36           Median :1.1354
#  G      :12                            Mean   :1.1225
#  I      :12                            3rd Qu.:1.2789
#  B      :11                            Max.   :1.6347
#  (Other):52
#     TCA.mm2           AA.mm2       Cort.Cell.Num
#  Min.   :0.0545   Min.   :0.0057   Min.   : 510
#  1st Qu.:0.7560   1st Qu.:0.1153   1st Qu.:1542
#  Median :0.9045   Median :0.2073   Median :1817
#  Mean   :0.8881   Mean   :0.2098   Mean   :1857
#  3rd Qu.:1.0176   3rd Qu.:0.2837   3rd Qu.:2136
#  Max.   :1.3882   Max.   :0.5084   Max.   :3331
#
#     XVA.mm2            Per.A         CellSize.avg
#  Min.   :0.00240   Min.   : 1.359   Min.   :0.0000510
#  1st Qu.:0.04080   1st Qu.:15.428   1st Qu.:0.0002739
#  Median :0.04960   Median :23.258   Median :0.0003802
```

```
#  Mean   :0.05079   Mean   :22.886   Mean   :0.0004251
#  3rd Qu.:0.06070   3rd Qu.:29.313   3rd Qu.:0.0005346
#  Max.   :0.08990   Max.   :46.262   Max.   :0.0012030
#
#    m.Loc
#  R1 L1:29
#  R1 L2:31
#  R2 L1:31
#  R2 L2:32
#
#
#
```

We're down to 12 variables, but this is probably too many for a pairs plot. We'll exclude the first 4 variables as they are factors.

```
pairs(anat[, -(1:4)])
```



That is a useful figure that permits us to quickly see how seven variables are related to each other. What would happen if we used `pairs(anat)`? For more variables than 7 or 8 the utility of such a figure probably declines. We can see that RXSA (cross section area) is tightly correlated with TCA (cortical area), and that AA (aerenchyma area) is correlated with Per.A (aerenchyma area as percent of cortex). These are not surprising, as they are mathematically related. XVA (xylem vessel area) does not seem to be strongly correlated with any other

measure. Per.A (Cortical aerenchyma as percent of root cortex) is correlated with average cell size (which is interesting if you are a root biologist!).

Sometimes it is useful to enlarge a plot to see it better - especially if it is a complex one like this pairs plot. In RStudio, you can click the "zoom" button above the plots to enlarge a plot for viewing.

We can also use the function `by()` for multi-way data summaries over factor variables.

```
by(data = anat[, 7:10], INDICES = list(anat$media.rep, anat$Loc),
    FUN = colMeans)
```

```
# : R1
# : L1
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  1.712069e-01  1.808207e+03  4.722414e-02  2.038428e+01
# ------------------------------------------------
# : R2
# : L1
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  1.368774e-01  1.740452e+03  4.993548e-02  1.864896e+01
# ------------------------------------------------
# : R1
# : L2
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  2.577355e-01  2.009871e+03  5.344839e-02  2.447344e+01
# ------------------------------------------------
# : R2
# : L2
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  2.688344e-01  1.865281e+03  5.228438e-02  2.772134e+01
```

Note that if we are including more than one factor in the argument `INDICES` they must be in a `list()` - the syntax is similar to that for `aggregate()` that we saw in chapter 9.

10.4 PCA

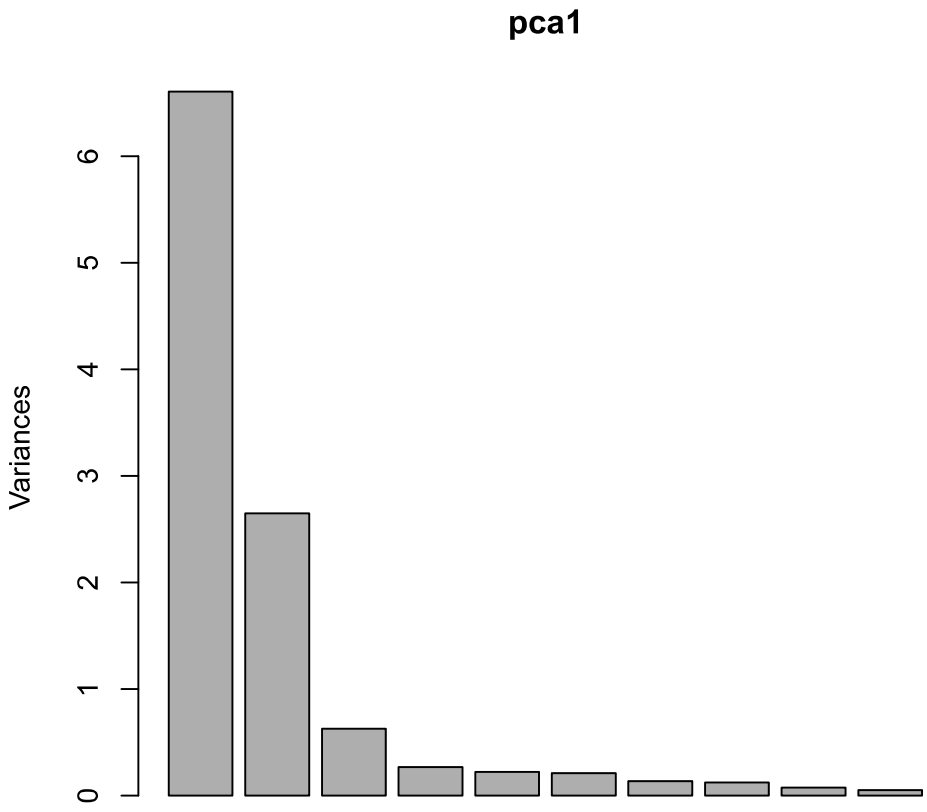
Since we live in a three dimensional world, our perceptual ability can generally cope with three dimensions, but we often have difficult time visualizing or understanding higher dimensional problems. Principal Components Analysis, or PCA, is a tool for reducing the dimensions of a data set.

In the most basic terms, PCA rotates the data cloud to produce new axes, or dimensions, that maximize the variability. These are the main axes of variation in the data, or the "Principal Components". They can be related to the original variables only by rotation. Here is an example with the `mtcars` data:

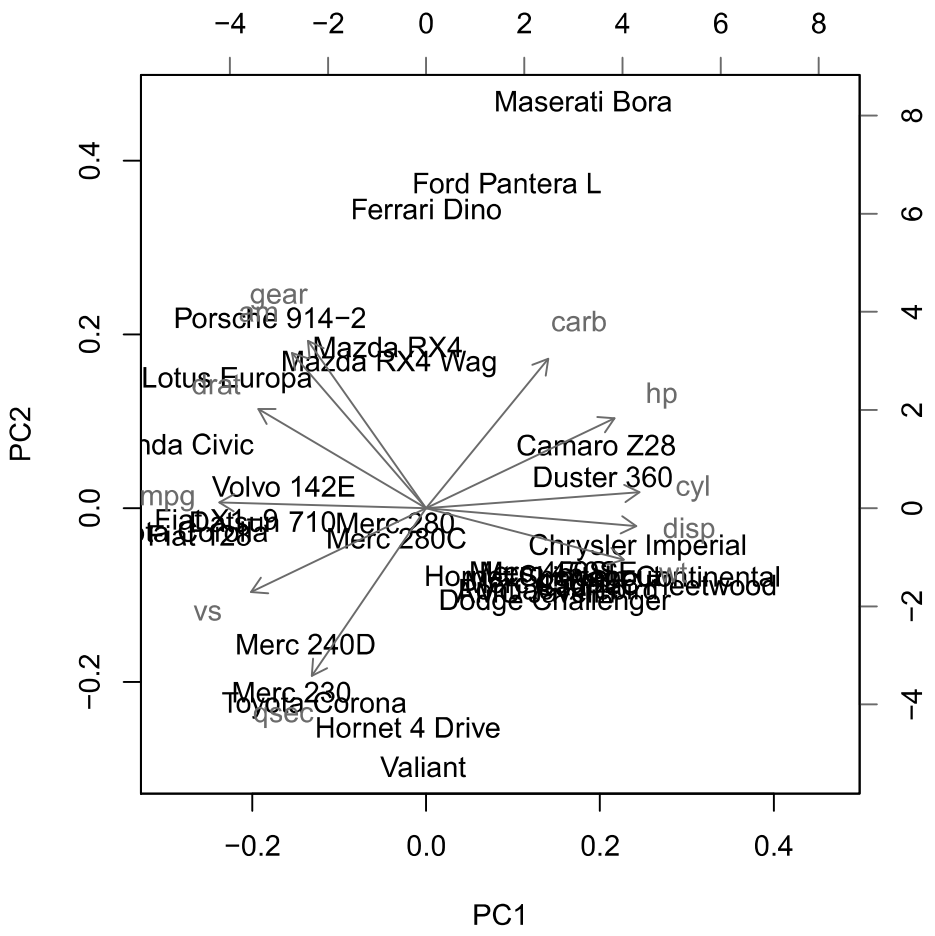
```
data(mtcars)
pca1 <- prcomp(mtcars, center = TRUE, scale. = TRUE)
summary(pca1)
```

```
# Importance of components:
#                           PC1    PC2     PC3     PC4     PC5
# Standard deviation     2.5707 1.6280 0.79196 0.51923 0.47271
# Proportion of Variance 0.6008 0.2409 0.05702 0.02451 0.02031
# Cumulative Proportion  0.6008 0.8417 0.89873 0.92324 0.94356
#                           PC6    PC7     PC8    PC9    PC10
# Standard deviation     0.46000 0.3678 0.35057 0.2776 0.22811
# Proportion of Variance 0.01924 0.0123 0.01117 0.0070 0.00473
# Cumulative Proportion  0.96279 0.9751 0.98626 0.9933 0.99800
#                          PC11
# Standard deviation     0.1485
# Proportion of Variance 0.0020
# Cumulative Proportion  1.0000
```

```
screeplot(pca1)
```



```
biplot(pca1)
```




```
pca1$rotation
```

```
#              PC1         PC2         PC3          PC4         PC5
# mpg  -0.3625305  0.01612440 -0.22574419 -0.022540255  0.10284468
# cyl   0.3739160  0.04374371 -0.17531118 -0.002591838  0.05848381
# disp  0.3681852 -0.04932413 -0.06148414  0.256607885  0.39399530
# hp    0.3300569  0.24878402  0.14001476 -0.067676157  0.54004744
# drat -0.2941514  0.27469408  0.16118879  0.854828743  0.07732727
# wt    0.3461033 -0.14303825  0.34181851  0.245899314 -0.07502912
# qsec -0.2004563 -0.46337482  0.40316904  0.068076532 -0.16466591
# vs   -0.3065113 -0.23164699  0.42881517 -0.214848616  0.59953955
# am   -0.2349429  0.42941765 -0.20576657 -0.030462908  0.08978128
# gear -0.2069162  0.46234863  0.28977993 -0.264690521  0.04832960
# carb  0.2140177  0.41357106  0.52854459 -0.126789179 -0.36131875
```

```
#              PC6          PC7          PC8          PC9
# mpg  -0.10879743  0.367723810 -0.754091423  0.235701617
# cyl   0.16855369  0.057277736 -0.230824925  0.054035270
# disp -0.33616451  0.214303077  0.001142134  0.198427848
# hp    0.07143563 -0.001495989 -0.222358441 -0.575830072
# drat  0.24449705  0.021119857  0.032193501 -0.046901228
# wt   -0.46493964 -0.020668302 -0.008571929  0.359498251
# qsec -0.33048032  0.050010522 -0.231840021 -0.528377185
# vs    0.19401702 -0.265780836  0.025935128  0.358582624
# am   -0.57081745 -0.587305101 -0.059746952 -0.047403982
# gear -0.24356284  0.605097617  0.336150240 -0.001735039
# carb  0.18352168 -0.174603192 -0.395629107  0.170640677
#             PC10         PC11
# mpg   0.13928524 -0.124895628
# cyl  -0.84641949 -0.140695441
# disp  0.04937979  0.660606481
# hp    0.24782351 -0.256492062
# drat -0.10149369 -0.039530246
# wt    0.09439426 -0.567448697
# qsec -0.27067295  0.181361780
# vs   -0.15903909  0.008414634
# am   -0.17778541  0.029823537
# gear -0.21382515 -0.053507085
# carb  0.07225950  0.319594676
```

Notice:

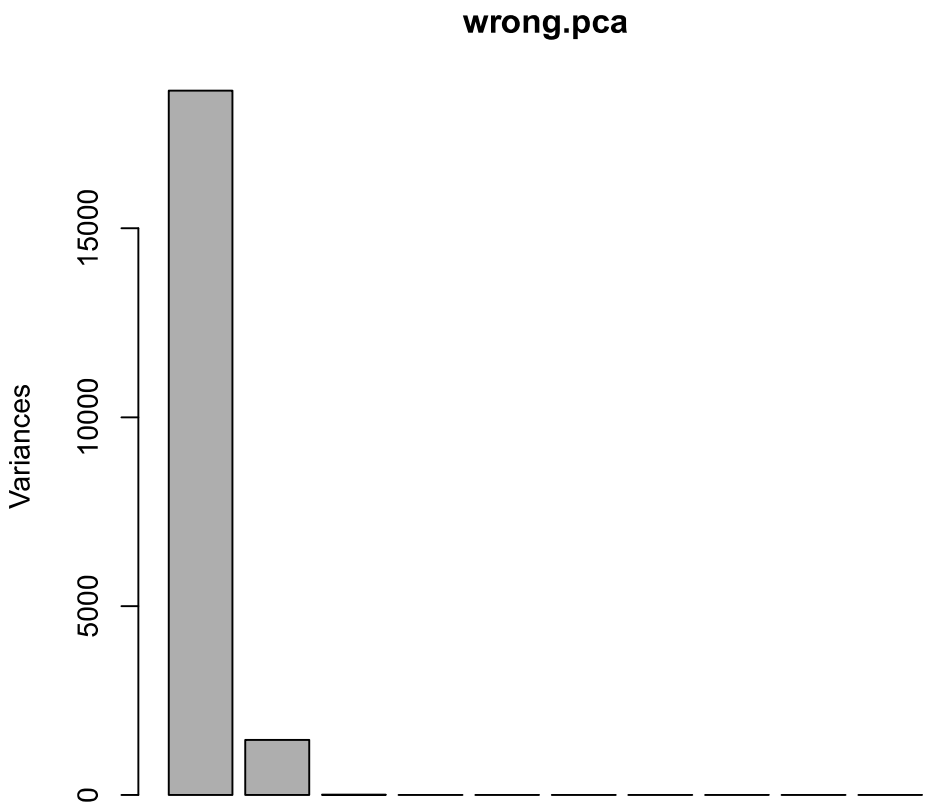
1. `prcomp()` requires numeric variables only; we must exclude any non-numeric variables in the data.
2. Because the variables may be scaled differently, it is almost always necessary to subtract the mean and scale by the standard deviation, so all variables share the same scale, hence the arguments `center` and `scale.` For more evidence see the next code block.
3. `summary()` on an object of type `prcomp` gives us a brief description of the principal components. Here we see that there are 11 PCs - the full number of PCs is always the number of dimensions in the data, in this case 11.
4. The "reduction of dimensions" can be seen in the "Cumulative Proportion" row - the first three PCs explain about 90% of the variation in the data, so we can consider three rather than 11 variables (although those three represent a synthesis of all 11). The `screeplot()` is basically a barplot of the the proportion of variance explained. This data set is rather nicely behaved, the first 2 or 3 PCs capture most of the variation. This is not always the case.
5. The `rotation`, sometimes called "loadings" shows the contribution of each variable to each PC. Note that the signs (directions) are arbitrary here.

6. PCA is often visualized with a "biplot", in which the data points are visualized in a scatterplot of the first two PCs, and the original variables are also drawn in relation to the first two PCs. Here one can see that vehicle weight (`wt`) is rather well correlated with number of cylinders and displacement, but that fuel economy (`mpg`) in inversely correlated with number of cylinders and displacement. Variables that are orthogonal are not well correlated - forward gears (`gear`) and horsepower (`hp`). In this case the first 2 PCs explain about 85% of the variation in the data, so these relationships between variables are not absolute.

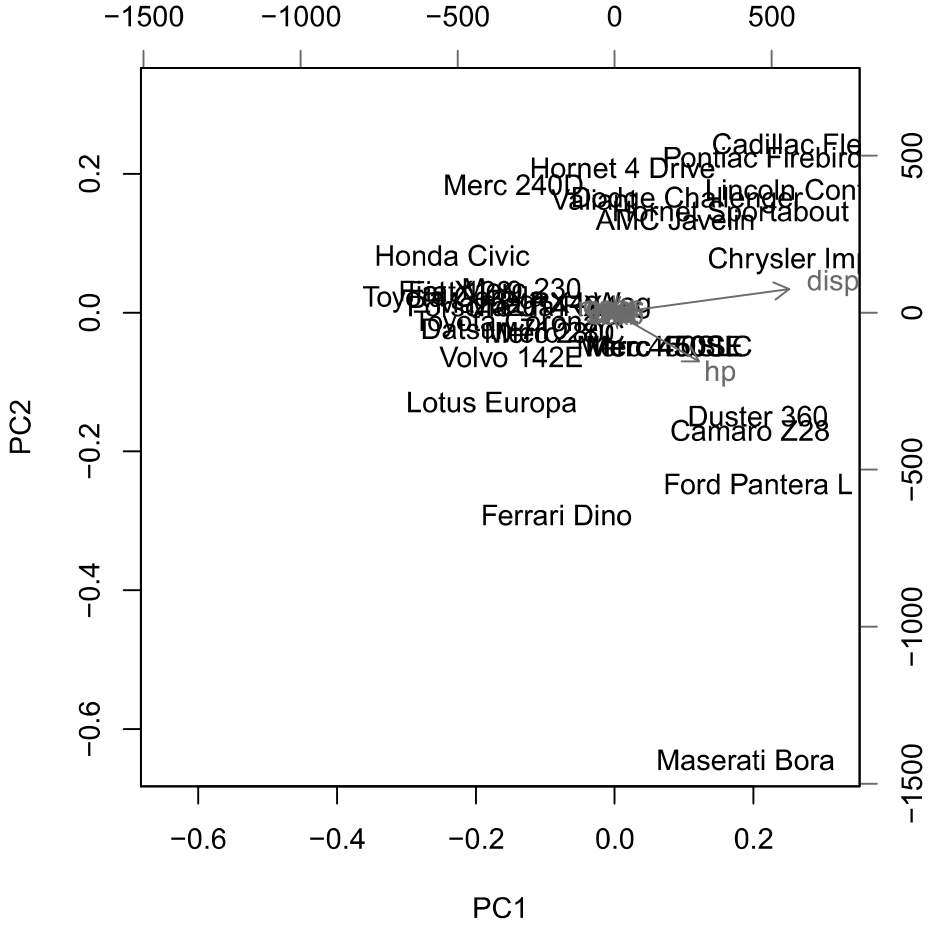
```
wrong.pca <- prcomp(mtcars)
summary(wrong.pca)
```

```
# Importance of components:
#                             PC1      PC2     PC3     PC4     PC5
# Standard deviation     136.533 38.14808 3.07102 1.30665 0.90649
# Proportion of Variance   0.927  0.07237 0.00047 0.00008 0.00004
# Cumulative Proportion    0.927  0.99937 0.99984 0.99992 0.99996
#                             PC6    PC7   PC8    PC9   PC10   PC11
# Standard deviation     0.66354 0.3086 0.286 0.2507 0.2107 0.1984
# Proportion of Variance 0.00002 0.0000 0.000 0.0000 0.0000 0.0000
# Cumulative Proportion  0.99998 1.0000 1.000 1.0000 1.0000 1.0000
```

```
screeplot(wrong.pca)
```




```
biplot(wrong.pca)
```



To highlight the importance of scaling, we demonstrate here the unscaled pca of the mtcars data. Note that for the *unscaled* `wrong.pca` the first 2 PCs explain nearly all the variance - this is simply because a small number (2) of variables are scaled much "larger" than the others and so contribute a much larger share of the variance. A look at `summary(mtcars)` should show you that re-scaling `mpg`, `disp`, and `hp` (simply dividing by 10, 100, and 100, respectively) would yield a different pca than the original scaled pca - I leave it for you to test.

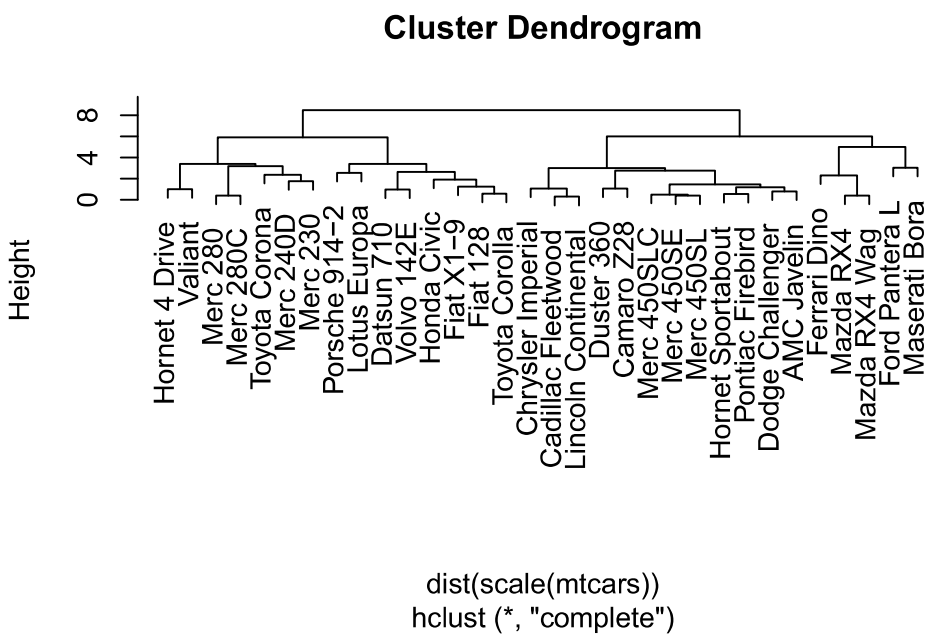
10.5 Clustering

Another way to look at multivariate data is to ask how data points may be related to each other. A glance at the biplot above shows that there are some small clusters of data points. There are a number of ways to approach this question. Here we'll demonstrate hierarchical clustering. This begins by finding the points that are "closest" to each other. In R "Closeness" is actually calculated by the function `dist()` and can be defined various ways - the simplest is the

euclidean distance, but there are other options also (see `?dist`). Data points are then grouped into clusters based on their "closeness" (or distance), and (as you might expect by now) there are different methods of grouping ("agglomeration") supported - see `?hclust` for more.

Now we can use `dist()` and `hclust()` to cluster the data. However, as we saw with PCA, differences in variable scaling can be very important. We can use the function `scale()` to scale and center the data before clustering. As with PCA, it is instructive to compare the this dendrogram with that derived from unscaled data (again, I leave this to the reader).

```
plot(hclust(dist(scale(mtcars))))
```



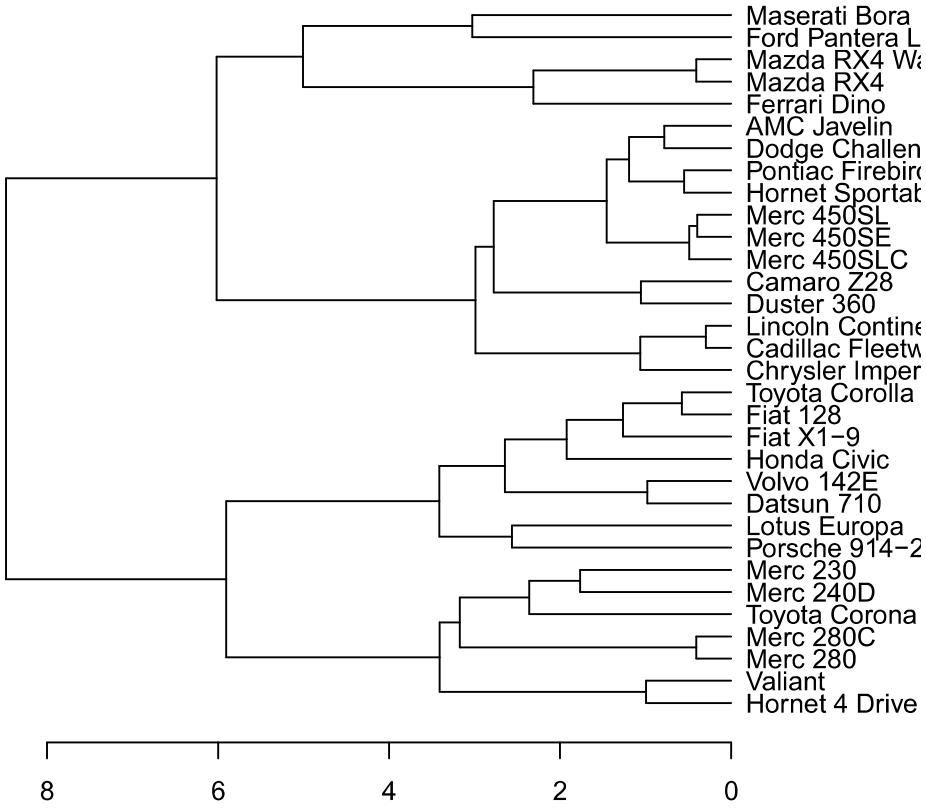
Note that the three Merc 450 models cluster together, and the Cadillac Fleetwood and Lincoln Continental also cluster. The dendrogram for the unscaled data makes less sense (though if you don't happen to remember cars from 1973 it is possible that *none* of these clusters look "right" to you).

10.5.1 Plotting a dendrogram horizontally.

If we want to view the dendrogram printed horizontally, we have to save our `hclust` object and create a `dendrogram` object from it. We'd also need to tweak the plot margins a bit, but it might make the dendrogram easier to read:

```
hc <- hclust(dist(scale(mtcars)))
dendro <- as.dendrogram(hc)
par(mar = c(3, 0.5, 0.5, 5))
```

```
plot(dendro, horiz = TRUE)
```



Another clustering tool is "k-means clustering". K-means clustering does require that we provide the number of clusters, and it is sensitive to the starting location of the clusters (which can be user specified or automatic). This means that K-means will not always return exactly the same clustering. Here we'll demonstrate using the `iris` data, which has floral dimensions for three species of iris. We'll specify three clusters to see how well we can separate the three species.

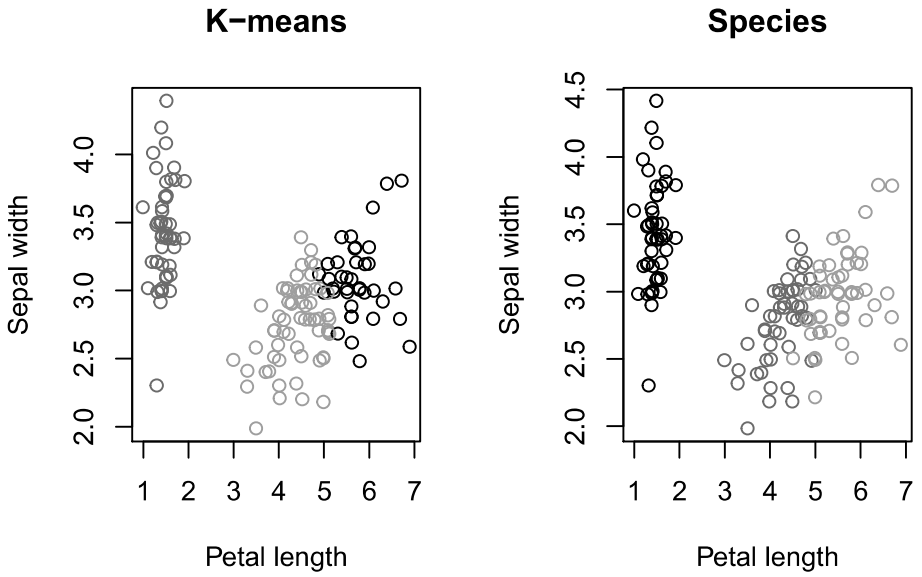
```
data(iris)
kcl <- kmeans(iris[, -5], 3, nstart = 3)
table(kcl$cluster, iris$Species)
```

```
#
#     setosa versicolor virginica
#   1      0          2        36
#   2     50          0         0
#   3      0         48        14
```

Based on the table, it seems like "setosa" can be reliably separated but "versicolor" and "virginica" are somewhat more difficult to separate, although even "virginica"

is identified 72% of the time. Let's examine this data a bit more. Since the data is rounded to the nearest 0.1 cm, we'll add a bit of noise to the data, using `jitter()`

```
plot(jitter(iris$Petal.Length), jitter(iris$Sepal.Width), col = kcl$cluster,
    main = "K-means", xlab = "Petal length", ylab = "Sepal width")
plot(jitter(iris$Petal.Length), jitter(iris$Sepal.Width), col = iris$Species,
    main = "Species", xlab = "Petal length", ylab = "Sepal width")
```



Note that the colors in the above plots are essentially arbitrary. To better see where the `kmeans()` result doesn't match the original species, it is worth "zooming in" on the are where there is some confusion. Try running the above code and adding "`xlim=c(4,6),ylim=c(2.3,3.4)`" to both `plot()` calls. It is also worth repeating these plots without the use of `jitter()` to better appreciate how it helps in this case.

For more useful descriptions of functions, see the CRAN Task View on multivariate analysis.

10.6 Exercises

1) The built in dataset `iris` has data on four floral measurements for three different species of iris. Make a pairs plot of the data. What relationships (correlations) look strongest?

2) The grouping evident with the `Species` variable in the last plot should make you curious. Add the argument `col=iris$Species` to the last plot you made. Does this change your conclusions about correlations between any of the

relationships? Can you make a lattice plot (`xyplot()`) showing `Sepal.Length` as a function of `Sepal.Width` for the different species?

3) The built in data `state.x77` (which can be loaded via `data(state)`) has data for the 50 US states. Fit a principal components analysis to this data. What proportion of variation is explained by the first three principal components? What variable has the greatest (absolute value) loading value on each of the first three principal components? (*Note:* the dataset `state` is a list of datasets one of which is a matrix named `state.x77`)

4) The `state.x77` can also be approached using hierarchical clustering. Create a cluster dendrogram of the first 20 states (in alphabetical order, as presented in the data) using `hclust()`. (*Hint:* given the length of the names, it might be worth plotting the dendrogram horizontally). Do any clusters stand out as surprising?

Chapter 11

Linear Models I

Linear regression

11.1 Introduction

Regression is one of the most basic but fundamental statistical tools. We have a *response* ("dependent") variable - y that we want to model, or *predict* as a function of the *predictor* ("independent") variable - x. (y~x in R-speak).

We assume a linear model of the form $y = B_0 + B_1 * x + e$ where B_0 is the intercept, B_1 is the slope, and e is the *error*. Mathematically we can estimate B_0 and B_1 from the data. Statistical inference requires that we assume that: *the errors are independent & normal with mean 0 and var* $= s^2$.

Notice that we **don't** have to assume that y or x are normally distributed, only that the **error** (or *residuals*) are normally distributed [1].

11.2 Violation of Assumptions and Transformation of Data

We'll begin with some data showing body and brain weights for 62 species of mammals (if you haven't installed MASS, do `install.packages("MASS")`). We'll load the data and fit a regression.

```
library(MASS)
data(mammals)
```

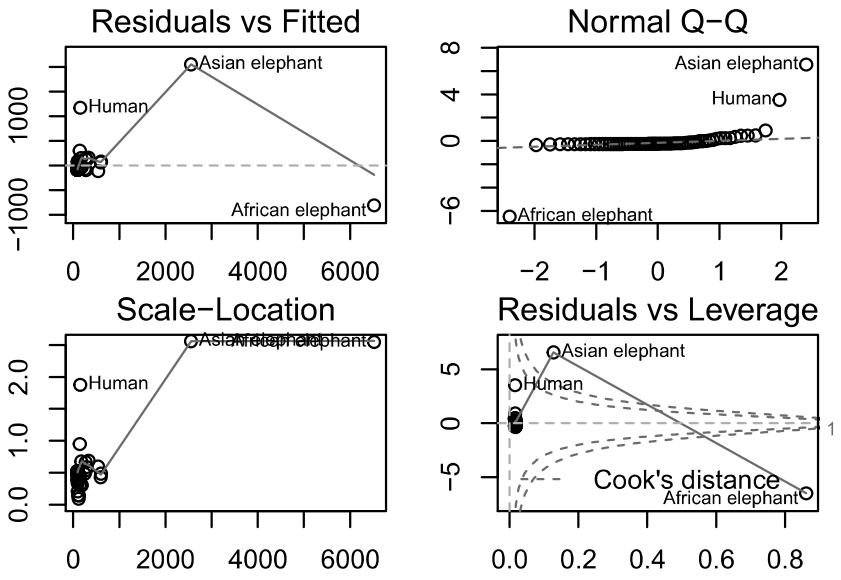
[1] In practice, regression and ANOVA methods are fairly robust to some degree of non-normality in the residuals, but substantial non-random patterning in the residuals is cause for concern as it indicates either need for transformation of the response *or* an inadequate model (or both).

```
model1 <- lm(brain ~ body, data = mammals)
summary(model1)
```

```
#
# Call:
# lm(formula = brain ~ body, data = mammals)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -810.07  -88.52  -79.64  -13.02 2050.33
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 91.00440   43.55258    2.09   0.0409 *
# body         0.96650    0.04766   20.28   <2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 334.7 on 60 degrees of freedom
# Multiple R-squared:  0.8727,Adjusted R-squared:  0.8705
# F-statistic: 411.2 on 1 and 60 DF,  p-value: < 2.2e-16
```

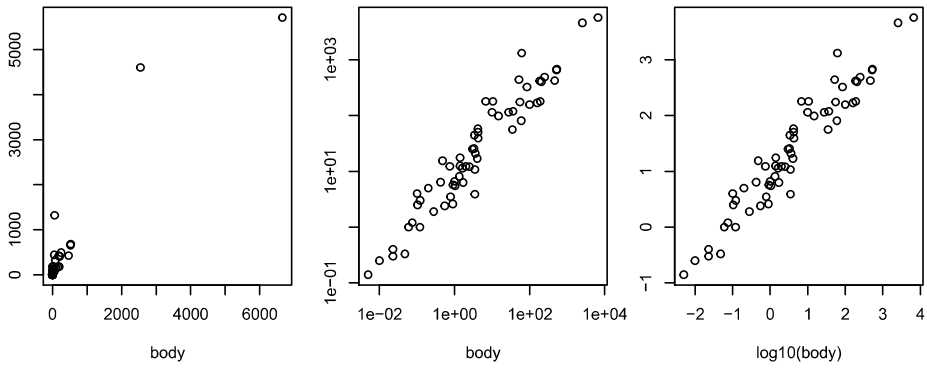
So far this looks pretty satisfactory - R^2 is , and the p-value is vanishingly small ($< 2.2*10^{-16}$). But let's have a look at the distribution of the residuals to see if we're violating any assumptions:

```
plot(model1)
```



The diagnostic plots show some problems - the residuals don't seem to be normally distributed (Normal Q-Q plot shows large departure from linear). This suggests that transformation may be needed. Let's look at the data, before and after log transformation:

```
with(mammals, plot(body, brain))
with(mammals, plot(body, brain, log = "xy"))
with(mammals, plot(log10(body), log10(brain)))
```



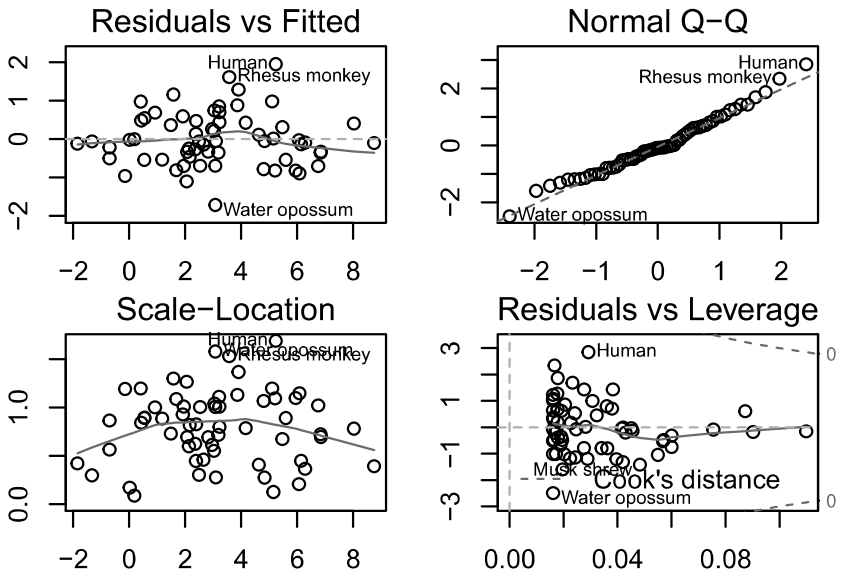
Here it is clear that this data may need *log transformation* (as is often the case when the values occur over several orders of magnitude). The log/log plot of body and brain mass shows a much stronger linear association. Let's refit the model.

```
model2 <- lm(log(brain) ~ log(body), data = mammals)
summary(model2)
```

```
#
# Call:
# lm(formula = log(brain) ~ log(body), data = mammals)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -1.71550 -0.49228 -0.06162  0.43597  1.94829
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  2.13479    0.09604   22.23   <2e-16 ***
# log(body)    0.75169    0.02846   26.41   <2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.6943 on 60 degrees of freedom
# Multiple R-squared:  0.9208,Adjusted R-squared:  0.9195
# F-statistic: 697.4 on 1 and 60 DF,  p-value: < 2.2e-16
```

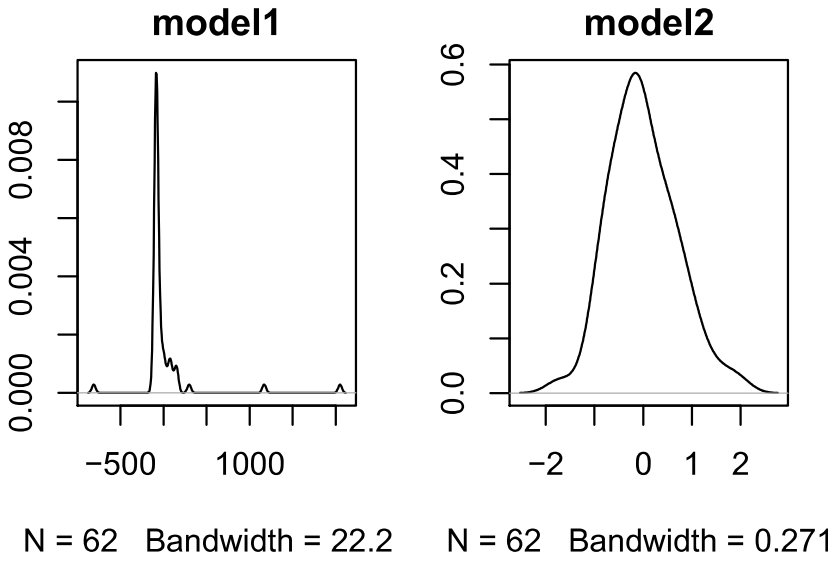
We see a bump in the R^2. The p-value isn't appreciably improved (but it was already *very* significant).

```
plot(model2)
```



This is much more satisfactory - the residuals are much nearer normal distribution than in the raw data. We can confirm this by inspecting the residuals. From the summary we can see that `log(brain)` is predicted to be near `2.13 + 0.752 * log(body)`, so each unit increase in `log(body)` yields an increase of `0.752` in `log(brain)`.

```
op <- par(mfrow = c(1, 2), mar = c(4, 3, 2, 1))
plot(density(model1$resid), main = "model1")
plot(density(model2$resid), main = "model2")
```



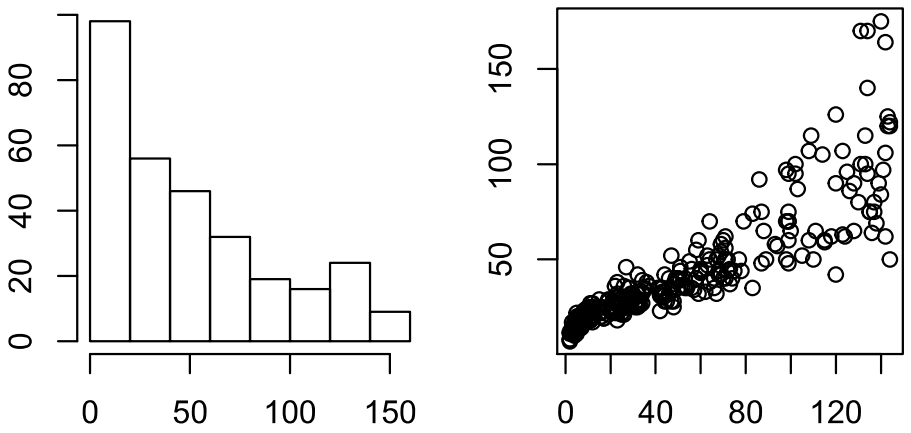

```
par(op)
```

Note that we *must refit the model* to check this - the residuals from the log transformed model *are not the same* as the log of the residuals! They can't be - the residuals are centered about zero (or should be), so there are many negative values, an the log of a negative number can't be computed.

Notice that `lm()` creates an "lm object" from which we can extract things, such as residuals. Try `coef(model1)` and `summary(model2)$coeff`.

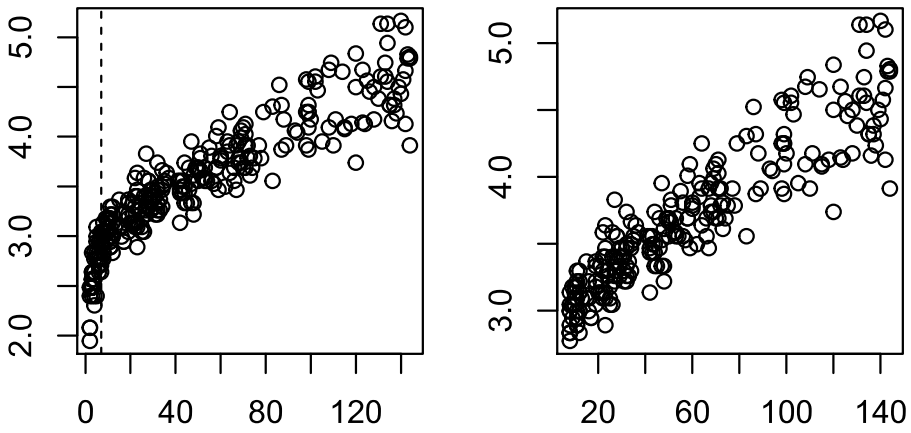
Another example of violation of assumptions. This is weight and age (in months) data for 5337 children <12 years old.

```
Weights <- read.csv("../Data/WeightData.csv", comm = "#")
hist(Weights$age, main = "")
plot(weight ~ age, data = Weights)
```



We can see that there appears to be increasing *variation in weight* as children age. This is pretty much what we'd expect - there is more space for variation (in absolute terms) in the weight of 12 year-olds than in 12 week-olds. However this suggests that for a valid regression model transformation of the weight might be needed.

```
op <- par(mfrow = c(1, 2), mar = c(2, 3, 1.5, 0.5))
plot(log(weight) ~ age, data = Weights)
abline(v = 7, lty = 2)
plot(log(weight) ~ age, data = subset(Weights, age > 7))
```

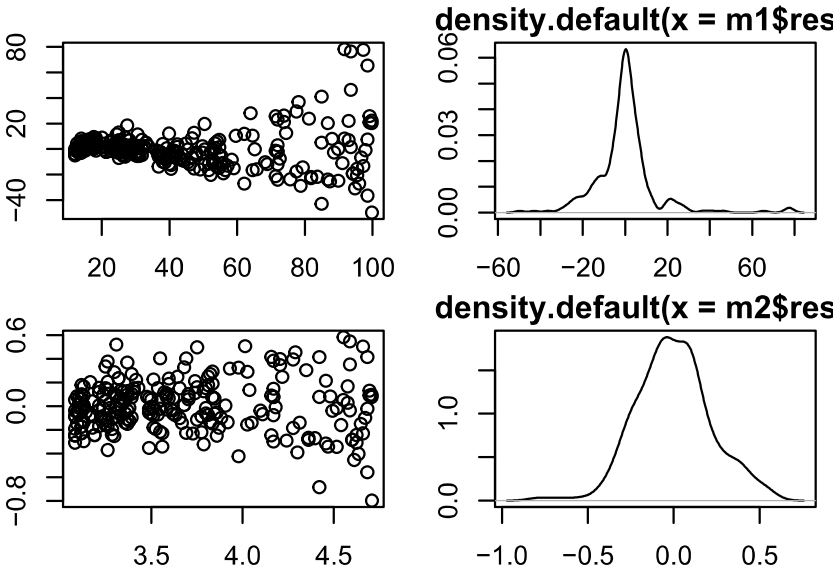


The variation appears more consistent in the first plot, apart from some reduced variation in weights (and a steeper slope) at the very low end of the age range (<7 months, indicated by the dashed line). In the second plot we've excluded this data. Let's fit a regression to both raw and transformed data and look at the residuals.

```
m1 <- lm(weight ~ age, data = Weights)
m2 <- lm(log(weight) ~ age, data = subset(Weights, age > 7))
```

You can use `plot(m1)` to look at the diagnostic plots. Let's compare the distribution of the residuals, though here we'll only look at the first two plots from each model.

```
op <- par(mfrow = c(2, 2), mar = c(2, 3, 1.5, 0.5))
plot(m1$fitted.values, m1$resid)
plot(density(m1$resid))
plot(m2$fitted.values, m2$resid)
plot(density(m2$resid))
```




```
par(op)
```

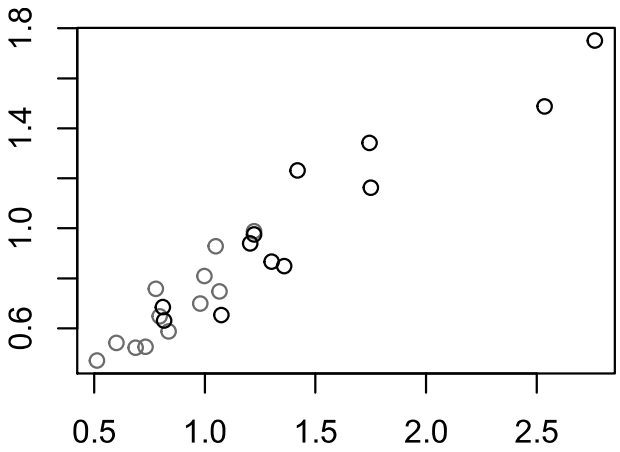
Both of the plots of residuals vs fitted values and the kernel density estimates of the residuals show that the transformed data (with the <7 month subjects excluded) more nearly meets the regression assumptions, though there is still a bit of change in variation, they are certainly sufficiently close to normal and constant to meet regression assumptions.

11.3 Hypothesis Testing

IN our analysis of the 'beans" data, there was a strong correlation between Shoot and Root biomass [2], and it seems to apply to both the High P and Low P plants.

[2]This data shows strong "root:shoot allometry" - the slope of log(Root)~log(Shoot) is constant across treatments, indicating that while a treatment might reduce overall *size*, it doesn't alter the fundamental growth pattern. See work by Karl Niklas and others on this topic.

```
beans <- read.csv("../Data/BeansData.csv", comm = "#")
plot(RtDM ~ ShtDM, data = beans, col = P.lev)
```

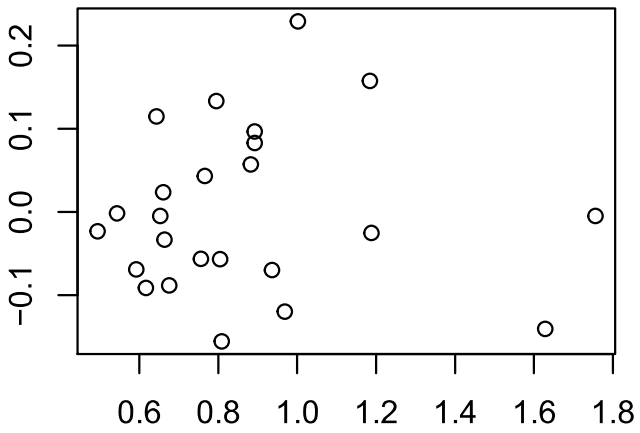


Let's have a look at the regression fit to this data. Note that a factor (like `P.lev`) can be used directly to color points. They are colored with the colors from the color palette that correspond to the factor levels (see `palette()` to view or change the default color palette).

```
m1 <- lm(RtDM ~ ShtDM, data = beans)
summary(m1)
```

```
#
# Call:
# lm(formula = RtDM ~ ShtDM, data = beans)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -0.15540 -0.06922 -0.01391  0.06373  0.22922
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  0.20659    0.04892   4.223  0.00035 ***
# ShtDM        0.56075    0.03771  14.872 5.84e-13 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.1007 on 22 degrees of freedom
# Multiple R-squared:  0.9095,Adjusted R-squared:  0.9054
# F-statistic: 221.2 on 1 and 22 DF,  p-value: 5.838e-13
```

```
plot(m1$resid ~ m1$fitted.values)
```



The residuals don't show any particular pattern, and they are approximately normal (see `plot(density(m1$resid))`). The summary shows an intercept of `signif(summary(m1)$coef[1,1],3)` and slope (the `ShtDM` term) of `signif(summary(m1)$coef[2,1],3)`. Standard errors are given for each parameter, as well as a t-value and a p-value (columns 3 and 4).

The t-value and p-values given by `summary(m1)`[3] are for the null hypothesis that B0 (or B1) is equal to 0. In this case the p-values are both very small, indicating a high degree of certainty that both parameters are *different* from 0 (note that this is a 2-sided test).

In some cases, we want to know more. For example, we might have the hypothesis that the *slope* of the root:shoot relationship should be 1. We can test this by calculating the t-value and p-value for this test. Recall that t is calculated as the difference between the observed and hypothesized values scaled by (divided by) the standard error.

```
B1 <- summary(m1)$coeff[2, 1:2]
t <- (B1[1] - 1)/B1[2]
t
```

```
#  Estimate
# -11.64947
```

For convenience we've stored the estimate and SE for B_1 (slope) as `B1` - we could just as well have used `t=(summary(m1)$coeff[2,1]-1)/summary(m1)$coeff[2,2]`. The t value is very large and negative (we can read this value as "B1 is 11.65 standard errors smaller than the hypothesized value"), so the slope estimate is smaller than the hypothesized value. If we want to get a p-value for this, we use the function `pt()`, which give probabilities for the t distribution.

```
pt(t, df = 22, lower.tail = TRUE)
```

[3]This applies to the t- and p-values shown in the summary for any lm object.

```
#     Estimate
# 3.505015e-11
```

You can see we specify 22 degrees of freedom - this is the same as error degrees of freedom shown in `summary(m1)`. We specified `lower.tail=TRUE` because we have a large negative value of `t` and we want to know how likely a lower (more negative) value would be - in this case it is pretty unlikely! Since our hypothesis is a 2-sided hypothesis, we'll need to multiply the p-value by 2.

We can use the same approach to test hypothesized values of B_0 - but be sure to use the SE for B_0, since the estimates of SE are specific to the parameter. Note: Here is one way to check your work on a test like this: Calculate a new y-value that incorporates the hypothesis. For example - here our hypothesis is B_1=1, mathematically `1 * beans$ShtDM`. So if we *subtract* that from the root biomass, what we have left is *the difference between our hypothesized slope and the observed slope.* In this case we calculate it as `RtDM-1*ShtDM`. If the hypothesis was *correct* then the slope of a regression of this new value on the predictor would be zero. Have a look at `summary(lm(RtDM-1*ShtDM~ShtDM,data=beans))` - the slope will be quite negative, showing that our hypothesis is not true, and the p-value very low. Note though that this p-value is 2-sided, and in fact is twice the p-value we calculated above.

R actually has a built-in function to do such tests - `offset()` can be used to specify offsets within the formula in a call to `lm()`.

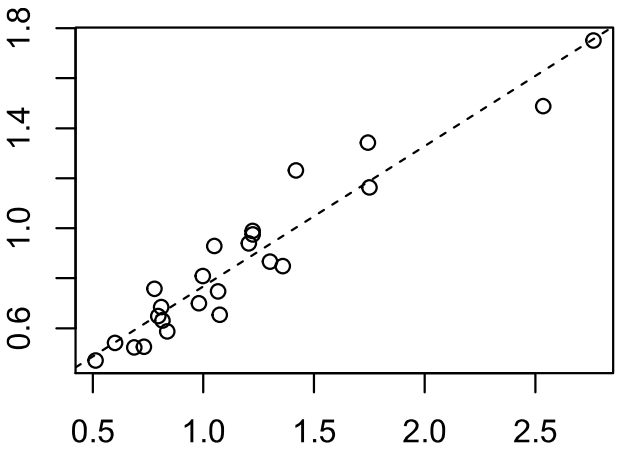
```
summary(lm(RtDM ~ ShtDM + offset(1 * ShtDM), data = beans))
```

```
#
# Call:
# lm(formula = RtDM ~ ShtDM + offset(1 * ShtDM), data = beans)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -0.15540 -0.06922 -0.01391  0.06373  0.22922
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  0.20659    0.04892   4.223  0.00035 ***
# ShtDM       -0.43925    0.03771 -11.649 7.01e-11 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.1007 on 22 degrees of freedom
# Multiple R-squared:  0.9095,Adjusted R-squared:  0.9054
# F-statistic: 221.2 on 1 and 22 DF,  p-value: 5.838e-13
```

The syntax is a bit odd, since we have both `ShtDM` and `1*ShtDM` in our model, but this is how we specify a fixed hypothetical value.

It is useful to know that the line plotting function `abline()` can take an lm object as an argument (The argument `lty` specifies line type - 2 is a dotted line).

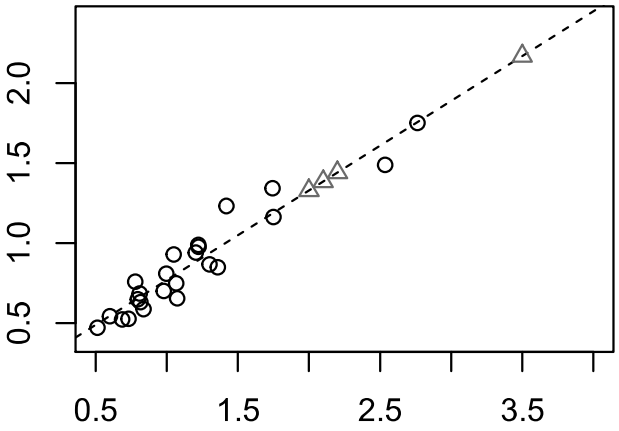
```
plot(RtDM ~ ShtDM, data = beans)
abline(m1, lty = 2)
```



11.4 Predictions and Confidence Intervals from Regression Models

In some cases we want to use a regression model to predict values we haven't (or can't) measure, or we would like to know how confident we are about a regression line. The function `predict()` can make predictions from lm objects.

```
new.vals <- c(2, 2.1, 2.2, 3.5)
preds = predict(m1, newdata = data.frame(ShtDM = new.vals))
points(new.vals, preds, col = "red", pch = 24)
```



A key detail to notice: `predict()` requires the `newdata` *as a data frame* - this

is to allow prediction from more complex models [4]. The predicted values should (and do) fall right on the line. Also notice that the final value for which we wanted a prediction is well beyond the range of the data. This is not wise, and such predictions should not be trusted (but R will not warn you - always engage the brain when analyzing!).

We can think about two types of "confidence intervals" for regressions. The first can be thought of as describing the certainty about the location of the regression line (the average location of `y` given `x`). R can calculate this with the `predict()` function if we ask for the "confidence" interval.

```
ci <- predict(m1, data.frame(ShtDM = sort(beans$ShtDM)), level = 0.95,
    interval = "confidence")
head(ci)
```

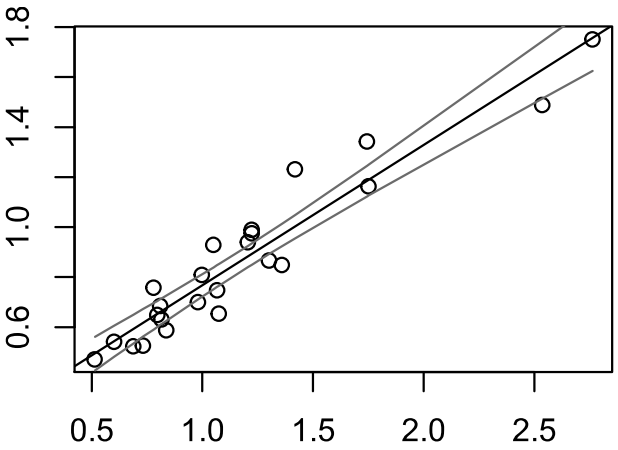
```
#         fit       lwr       upr
# 1 0.4942547 0.4270534 0.5614560
# 2 0.5432641 0.4811956 0.6053326
# 3 0.5919371 0.5346135 0.6492607
# 4 0.6169465 0.5618943 0.6719988
# 5 0.6434139 0.5906208 0.6962070
# 6 0.6530588 0.6010482 0.7050694
```

Notice there are three values - the "fitted" value, and the lower and upper CI. Also notice that we can specify a confidence level, and that we used our predictor variable (`beans$ShtDM`) as the `new` data, but *we used `sort()` on it* - this is to aid in plotting the interval.

We can plot this interval on our scatterplot using the function `lines()`. Since we've sorted the data in `ci` (y-axis), we need to sort the x-axis values also. (Note: `sort()` also removes any `NA` values.)

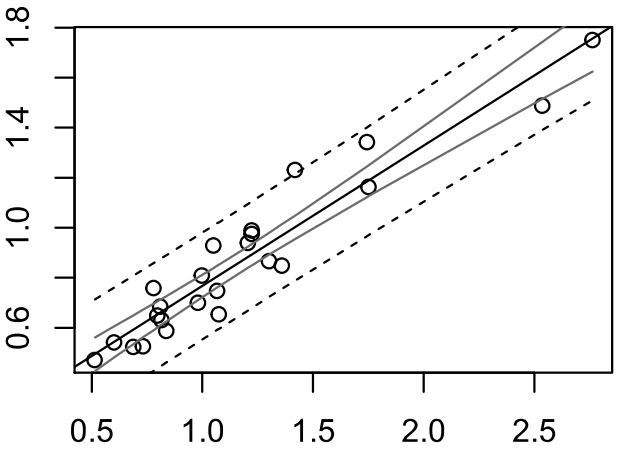
```
plot(RtDM ~ ShtDM, data = beans)
abline(m1, lty = 1)
lines(sort(beans$ShtDM), ci[, 2], col = "red")
lines(sort(beans$ShtDM), ci[, 3], col = "red")
```

[4] More precisely, the `newdata=` argument for `predict()` needs to be a data.frame *with the same variable names* as the predictors in the model for which predictions are being made.



It shouldn't surprise us that the confidence interval here is quite narrow - the p-values for B0 and B1 are very small. Notice that a fair number of the data points are outside the bands. This is because this "confidence interval" applies *to the regression line as a whole*. If we want to predict individual values, the uncertainty is a bit greater - this is called a "prediction interval".

```
lines(sort(beans$ShtDM), ci[, 2], col = "red")
lines(sort(beans$ShtDM), ci[, 3], col = "red")
pri <- predict(m1, data.frame(ShtDM = sort(beans$ShtDM)), level = 0.95,
    interval = "prediction")
lines(sort(beans$ShtDM), pri[, 2], lty = 2)
lines(sort(beans$ShtDM), pri[, 3], lty = 2)
```



When we plot this notice that ~95% of our data points are within this interval - this is consistent with the meaning of this interval.

11.5 Exercises

1) For the `beans` data test how effective root biomass (`RtDM`) is as a predictor of root length (`rt.len`).

2) For the `beans` data, test the hypothesis that the slope of the relationship of root biomass ~ shoot biomass ($B1$) is 0.5.

3) We worked with the dataset `mammals` earlier in this chapter, and concluded that it needed to be log-transformed to meet regression assumptions. Use `predict()` to calculate the confidence interval and regression line for this regression and graph it on both the log/log plot and on the un-transformed data (this will require that you back-transform the coordinates for the line and confidence intervals).

Chapter 12

Linear Models II

ANOVA

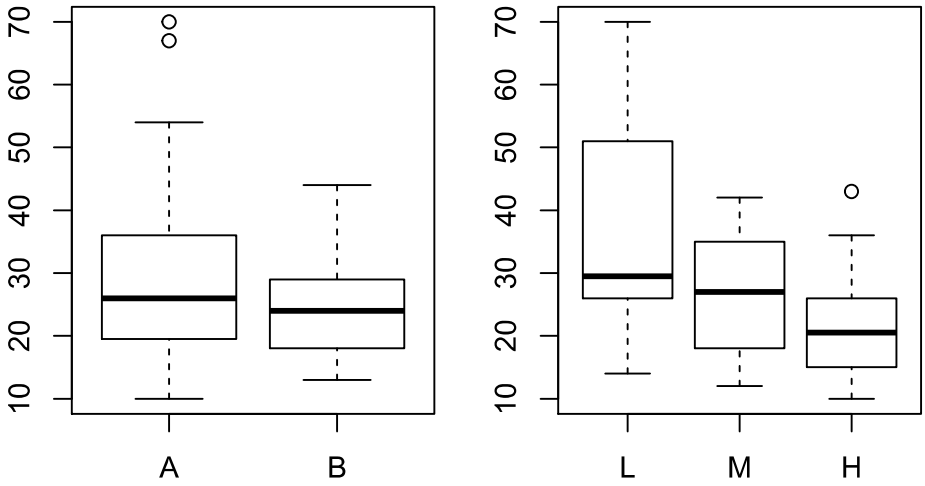
12.1 I. Introduction

It is often the case that we have one or more factors that we'd like to use to model some response. This is where we turn to Analysis of Variance, or ANOVA. Remember that, despite the fancy name, one-way ANOVA is basically just another form of regression - the continuous predictor variable is replaced by a factor. Since this is the case, it should not be surprising that the function `lm()` can be used for this type of analysis also.

12.2 One-way ANOVA

To explore this type of model we'll load some data on how the type of wool and the loom tension affects the number of breaks in wool yarn being woven.

```
data(warpbreaks)
boxplot(breaks ~ wool, data = warpbreaks)
boxplot(breaks ~ tension, data = warpbreaks)
```



Do the two types of wool have differing average numbers of breaks? The boxplot does not suggest much difference. One way to check would be to use a two-sample t-test.

```
t.test(breaks ~ wool, data = warpbreaks)
```

```
#
#  Welch Two Sample t-test
#
# data:  breaks by wool
# t = 1.6335, df = 42.006, p-value = 0.1098
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
#  -1.360096 12.915652
# sample estimates:
# mean in group A mean in group B
#        31.03704        25.25926
```

Of course, as we saw in Lesson 3, if we have more than two groups, we need something different.

```
oneway.test(breaks ~ tension, data = warpbreaks)
```

```
#
#  One-way analysis of means (not assuming equal variances)
#
# data:  breaks and tension
# F = 5.8018, num df = 2.00, denom df = 32.32, p-value =
# 0.007032
```

This strongly suggests that `tension` affects `breaks`. We can also use `lm()` to fit a linear model.

```
summary(lm(breaks ~ tension, data = warpbreaks))
```

```
#
# Call:
# lm(formula = breaks ~ tension, data = warpbreaks)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -22.389  -8.139  -2.667   6.333  33.611
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)    36.39       2.80  12.995  < 2e-16 ***
# tensionM      -10.00       3.96  -2.525 0.014717 *
# tensionH      -14.72       3.96  -3.718 0.000501 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 11.88 on 51 degrees of freedom
# Multiple R-squared:  0.2203,Adjusted R-squared:  0.1898
# F-statistic: 7.206 on 2 and 51 DF,  p-value: 0.001753
```

The coefficients shown in the output from summary begin with the "Intercept". This is the mean for the first level of the factor variable [1] - in this case `tension="L"`. The coefficient given for the next level is for the difference between the second and the first level, and that for the third is for the difference between first and third.

We can use the function `anova()` on an lm object to see an analysis of variance table for the model.

```
anova(lm(breaks ~ tension, data = warpbreaks))
```

```
# Analysis of Variance Table
#
# Response: breaks
#           Df Sum Sq Mean Sq F value   Pr(>F)
# tension    2 2034.3 1017.13  7.2061 0.001753 **
# Residuals 51 7198.6  141.15
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This shows very strong support (p=0.0018) for a significant effect of `tension` on

[1] By default factor levels are assigned by alpha-numeric order, so the default order for levels "H", "M", and "L" would be "H=1; L=2, M=3". This doesn't make sense in this case (though it wouldn't change the estimates of group means or differences between them). We saw how to fix this in Chapter 7.

`breaks`. Note that `summary(aov(breaks~tension,data=warpbreaks))`[2] will give the same result with slightly different format.

If we want to test all the differences between groups, we can use `TukeyHSD()` to do so - but we must use it on an an object created by `aov()`, it won't work with an `lm()` object.

```
TukeyHSD(aov(breaks ~ tension, data = warpbreaks))
```

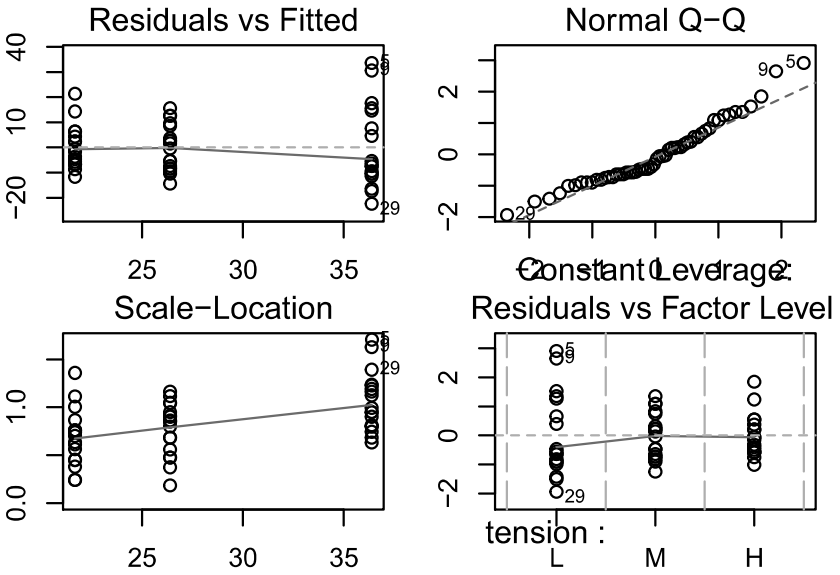
```
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = breaks ~ tension, data = warpbreaks)
#
# $tension
#           diff       lwr        upr     p adj
# M-L -10.000000 -19.55982 -0.4401756 0.0384598
# H-L -14.722222 -24.28205 -5.1623978 0.0014315
# H-M  -4.722222 -14.28205  4.8376022 0.4630831
```

This lists all the pairwise differences between groups showing the estimated difference between groups, the lower and upper confidence limits for the difference, and the p-value for the difference - a significant p-value means the difference is real. We can see that both `H` and `M` are different from `L`, but not from each other.

Of course, just as in regression it is good practice to check our diagnostic plots for violations of assumptions.

```
plot(lm(breaks ~ tension, data = warpbreaks))
```

[2]The syntax is potentially confusing — unfortunately `anova(lm(y~x, data=df))`, `summary(lm(y~x, data=df))`, and `aov(y~x,data=df)` are so confusingly similar.



There is evidence of modest difference in variance between groups (though not enough to cause concern), and the normal QQ plot shows that the residuals are near enough to normal.

12.3 For violations of assumptions.

For a one-way ANOVA (i.e. a single factor) where assumptions are violated, we do have a few options. The function `oneway.test()` we used above does not assume equal variance, so it can be used with unequal variance. If the residuals are strongly non-normal, the Kruskal-Wallace test is a non-parametric alternative.

```
kruskal.test(breaks ~ wool, data = warpbreaks)
```

```
#
#  Kruskal-Wallis rank sum test
#
# data:  breaks by wool
# Kruskal-Wallis chi-squared = 1.3261, df = 1, p-value =
# 0.2495
```

Another option is transformation of the data. However (as we'll see) a common cause of violation of regression assumptions is that there are sources of variation not included in the model. One of the brilliant features of the linear model is that it can accommodate multiple predictors, and the inclusion of the right predictors sometimes allows the regression assumptions to be met [3].

[3] Of course, this assumes that you knew or guessed what the "right" predictors might be and measured them.

12.4 Multi-Way ANOVA - Understanding `summary(lm())`

Particularly in designed experiments we often do have more than one factor that we need to include in our model. For example in the `warpbreaks` data we looked at both `tension` and `wool` separately, but we might need to *combine* them to understand what is going on. The *formula interface* lets us tell R *how* to use multiple predictors.

R formula	Y as a function of:
`Y ~ X1`	X1
`Y ~ X1 + X2`	X1 and X2
`Y ~ X1 * X2`	X1 and X2 and the X1xX2 interaction
`Y ~ X1 + X2 + X1:X2`	Same as above, but the interaction is explicit
`(Y ~ (X1 + X2 +X3)^2)`	X1, X2, and X3, with only 2-way interactions
`Y ~ X1 +I(X1^2)`	X1 and X1 squared (use `I()`for a literal power)
`Y ~ X1|X2`	X1 for each level of X2

```
summary(lm(breaks ~ wool + tension, data = warpbreaks))
```

```
#
# Call:
# lm(formula = breaks ~ wool + tension, data = warpbreaks)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -19.500  -8.083  -2.139   6.472  30.722
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   39.278      3.162  12.423  < 2e-16 ***
# woolB         -5.778      3.162  -1.827 0.073614 .
# tensionM     -10.000      3.872  -2.582 0.012787 *
# tensionH     -14.722      3.872  -3.802 0.000391 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 11.62 on 50 degrees of freedom
# Multiple R-squared:  0.2691,Adjusted R-squared:  0.2253
# F-statistic: 6.138 on 3 and 50 DF,  p-value: 0.00123
```

The model coefficients here are understood nearly as before - the intercept now is the first level of *each* factor (e.g. `wool=A` & `tension=L`). The `woolB` estimate is the difference between `wool=A` and `wool=B`. Because we have not included the `wool x tension` interaction here, we assume that the influence of `wool` is the

same for all levels of `tension`. As before we can use `anova()` to see an ANOVA table showing the estimate of the effect for each factor, and `TukeyHSD()` on an `aov()` fit to test group-wise differences.

```
anova(lm(breaks ~ wool + tension, data = warpbreaks))
```

```
# Analysis of Variance Table
#
# Response: breaks
#           Df Sum Sq Mean Sq F value   Pr(>F)
# wool       1  450.7  450.67  3.3393 0.073614 .
# tension    2 2034.3 1017.13  7.5367 0.001378 **
# Residuals 50 6747.9  134.96
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
TukeyHSD(aov(breaks ~ wool + tension, data = warpbreaks))
```

```
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = breaks ~ wool + tension, data = warpbreaks)
#
# $wool
#          diff       lwr       upr     p adj
# B-A -5.777778 -12.12841 0.5728505 0.0736137
#
# $tension
#           diff       lwr        upr     p adj
# M-L -10.000000 -19.35342 -0.6465793 0.0336262
# H-L -14.722222 -24.07564 -5.3688015 0.0011218
# H-M  -4.722222 -14.07564  4.6311985 0.4474210
```

Notice that the p-value for the ANOVA and the Tukey comparisons are the same for the factor `wool` but not for `tension` - that is because there are only two levels of `wool` but 3 levels of `tension`.

With more than one factor we also need to think about *interactions* between them, and what they mean. I this case we can understand the interaction as asking:
Does changing the tension have the same effect on breaks for both wool A and wool B?

```
summary(lm(breaks ~ wool + tension + wool:tension, data = warpbreaks))
```

```
#
# Call:
# lm(formula = breaks ~ wool + tension + wool:tension, data = warpbreaks)
#
```

```
# Residuals:
#      Min       1Q   Median       3Q      Max
# -19.5556  -6.8889  -0.6667   7.1944  25.4444
#
# Coefficients:
#                Estimate Std. Error t value Pr(>|t|)
# (Intercept)      44.556      3.647  12.218 2.43e-16 ***
# woolB           -16.333      5.157  -3.167 0.002677 **
# tensionM        -20.556      5.157  -3.986 0.000228 ***
# tensionH        -20.000      5.157  -3.878 0.000320 ***
# woolB:tensionM   21.111      7.294   2.895 0.005698 **
# woolB:tensionH   10.556      7.294   1.447 0.154327
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 10.94 on 48 degrees of freedom
# Multiple R-squared:  0.3778,Adjusted R-squared:  0.3129
# F-statistic: 5.828 on 5 and 48 DF,  p-value: 0.0002772
```

The output of `summary()` is similar to the above. The `woolB` estimate now is only for `tension=L`, since the last two estimates show the effect of wool on breaks for the other tensions. Since this estimate is negative for `tension=L` and positive for the higher levels of `tension` this is likely to be a significant difference.

12.5 Multi-Way ANOVA - Calculating group means

If we wanted to calculate the means for each for the groups we could do so by adding coefficients together - for example the estimate for wool=B; tension=H would be

```
Intercept (wool=A;tension=L)+woolB+tensionH+woolB:tensionH
```

It is not too complicated to calculate this from the coefficients.

```
m1 <- lm(breaks ~ wool * tension, data = warpbreaks)
sum(summary(m1)$coeff[c(1, 2, 4, 6), 1])
```

```
# [1] 18.77778
```

But it would be tedious (and error-prone) to do this for all factor levels. Fortunately we don't need to - we can use the function `predict()` to do it for us. We just need to give it all the possible factor levels as `"newdata"`. We can use the function `unique()` to give us the unique combinations of factor levels:

```
unique(warpbreaks[, 2:3])
```

```
#    wool tension
```

```
# 1     A       L
# 10    A       M
# 19    A       H
# 28    B       L
# 37    B       M
# 46    B       H
```

We can then use these as the `"newdata"` argument to `predict()` to get our predicted values. First we'll create the lm object, and then we'll create an object for this `"newdata"`, and then we'll calculate the predicted values.

```
m1 <- lm(breaks ~ wool + tension + wool:tension, data = warpbreaks)
m1.pv = unique(warpbreaks[, 2:3])
m1.pv$predicted = predict(m1, newdata = unique(warpbreaks[, 2:3]))
m1.pv
```

```
#    wool tension predicted
# 1     A       L  44.55556
# 10    A       M  24.00000
# 19    A       H  24.55556
# 28    B       L  28.22222
# 37    B       M  28.77778
# 46    B       H  18.77778
```

In some cases [4] we could also get these means is by using `tapply()` to apply the function `mean()`.

```
tapply(m1$fitted, list(warpbreaks$wool, warpbreaks$tension), mean)
```

```
#          L        M        H
# A 44.55556 24.00000 24.55556
# B 28.22222 28.77778 18.77778
```

12.6 Multi-Way ANOVA - Getting a handle on interactions

It can be hard (or nearly impossible) to understand what an interaction really means. In this example it means that the effect of changing tension of rate of breakage differs for the two types of wool.

```
anova(lm(breaks ~ wool + tension + wool:tension, data = warpbreaks))
```

```
# Analysis of Variance Table
#
# Response: breaks
```

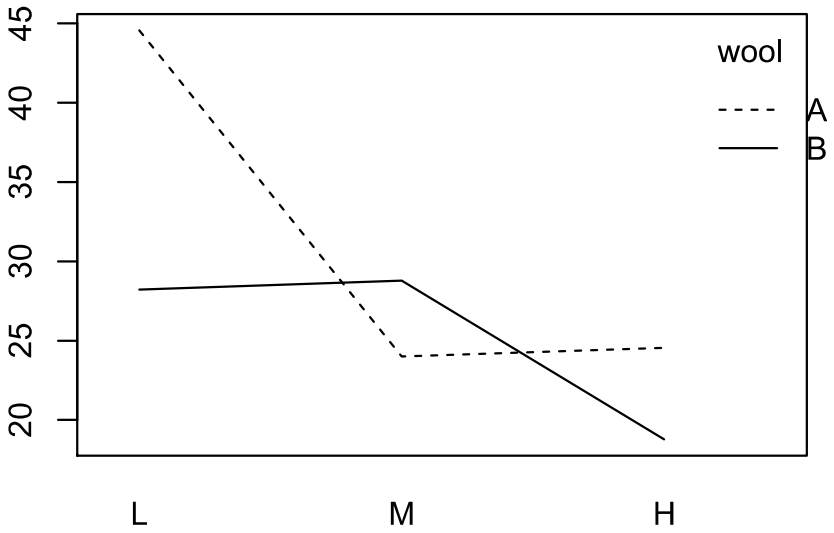
[4] Using `tapply()` in this way will only produce the same values as `predict()` for a *saturated* model, i.e. one that contains *all factors and interactions*! Also note that if there were any NA values in the data they would propagate as we haven't added `na.rm=TRUE`.

```
#              Df Sum Sq Mean Sq F value    Pr(>F)
# wool          1  450.7  450.67  3.7653 0.0582130 .
# tension       2 2034.3 1017.13  8.4980 0.0006926 ***
# wool:tension  2 1002.8  501.39  4.1891 0.0210442 *
# Residuals    48 5745.1  119.69
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Notice that here adding the interaction moved the effect of `wool` from p=0.074 to p=0.058. The interaction was strong enough that ignoring it increased the size of the residuals, and so reduced the magnitude of the F value.

Often it is helpful to visualize interactions to better understand them, and the function `interaction.plot()` gives us a quick way to do this.

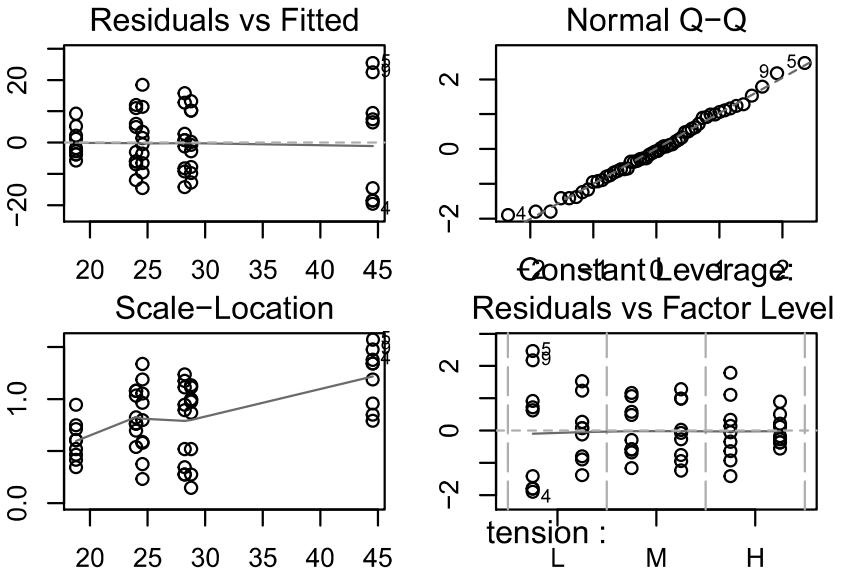
```
with(warpbreaks, interaction.plot(x.factor = tension, wool, response = breaks))
```



The syntax here is a bit different: the first and third arguments are the x and y axes, and the second is the grouping factor (`trace.factor`). This clearly and quickly shows us that the biggest difference between wools is at low tension.[5]

```
plot(lm(breaks ~ tension * wool, data = warpbreaks))
```

[5] A more complete interpretation is that increasing tension reduces the number of breaks, but this occurs at a lower tension for wool A than it does for wool B.



A check the diagnostic plots shows minimal evidence of unequal variance ("heteroscedasticity") or departure from normal distribution. As hinted above note how improving the model by accounting for more sources of variation (adding the second factor and the interaction) improved the agreement with regression assumptions.

12.7 Multi-Way ANOVA - Tukey HSD and family-wise error

When we introduce an interaction we now have many more groups. In this example we have 6 groups (2 wools * 3 tensions). This gives many more pairwise comparisons.

```
TukeyHSD(aov(breaks ~ wool + tension + wool:tension, data = warpbreaks))
```

```
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = breaks ~ wool + tension + wool:tension, data = warpbreaks)
#
# $wool
#          diff       lwr       upr    p adj
# B-A -5.777778 -11.76458 0.2090243 0.058213
#
# $tension
#           diff       lwr       upr     p adj
# M-L -10.000000 -18.81965 -1.180353 0.0228554
```

```
# H-L -14.722222 -23.54187 -5.902575 0.0005595
# H-M  -4.722222 -13.54187  4.097425 0.4049442
#
# $`wool:tension`
#                diff       lwr        upr     p adj
# B:L-A:L -16.3333333 -31.63966  -1.027012 0.0302143
# A:M-A:L -20.5555556 -35.86188  -5.249234 0.0029580
# B:M-A:L -15.7777778 -31.08410  -0.471456 0.0398172
# A:H-A:L -20.0000000 -35.30632  -4.693678 0.0040955
# B:H-A:L -25.7777778 -41.08410 -10.471456 0.0001136
# A:M-B:L  -4.2222222 -19.52854  11.084100 0.9626541
# B:M-B:L   0.5555556 -14.75077  15.861877 0.9999978
# A:H-B:L  -3.6666667 -18.97299  11.639655 0.9797123
# B:H-B:L  -9.4444444 -24.75077   5.861877 0.4560950
# B:M-A:M   4.7777778 -10.52854  20.084100 0.9377205
# A:H-A:M   0.5555556 -14.75077  15.861877 0.9999978
# B:H-A:M  -5.2222222 -20.52854  10.084100 0.9114780
# A:H-B:M  -4.2222222 -19.52854  11.084100 0.9626541
# B:H-B:M -10.0000000 -25.30632   5.306322 0.3918767
# B:H-A:H  -5.7777778 -21.08410   9.528544 0.8705572
```

Tukey's HSD shows us all 15 pairwise differences between the 6 combinations of `wool` and `tension`. This a situation where Fishers LSD does not perform well. The probability of detecting at least *one* difference where none exist is $(1-(1-\alpha)^n)$, with $\alpha = 0.05$ and n=15 this is 0.537. This shows why Tukey's multiple comparisons is important – this "family-wise" error rate can get quite high quickly.

Sorting out which groups really differ is not always simple, though in this case we can rather quickly see that the other five groups differ from woolA:tensionL, but those 5 groups don't differ from each other.

12.8 HSD.test - a useful tool for ANOVA

The package `agricolae` has some nice tools to make multiple comparisons a bit easier. Install the package to follow along.

```
library(agricolae)
data(sweetpotato)
```

Now we can fit a model to the data and make comparisons.

```
model <- aov(yield ~ virus, data = sweetpotato)
out <- HSD.test(model, "virus", group = TRUE)
out$means
```

```
#       yield      std r  Min  Max   Q25  Q50   Q75
# cc 24.40000 3.609709 3 21.7 28.5 22.35 23.0 25.75
```

```
# fc 12.86667 2.159475 3 10.6 14.9 11.85 13.1 14.00
# ff 36.33333 7.333030 3 28.0 41.8 33.60 39.2 40.50
# oo 36.90000 4.300000 3 32.1 40.4 35.15 38.2 39.30
```

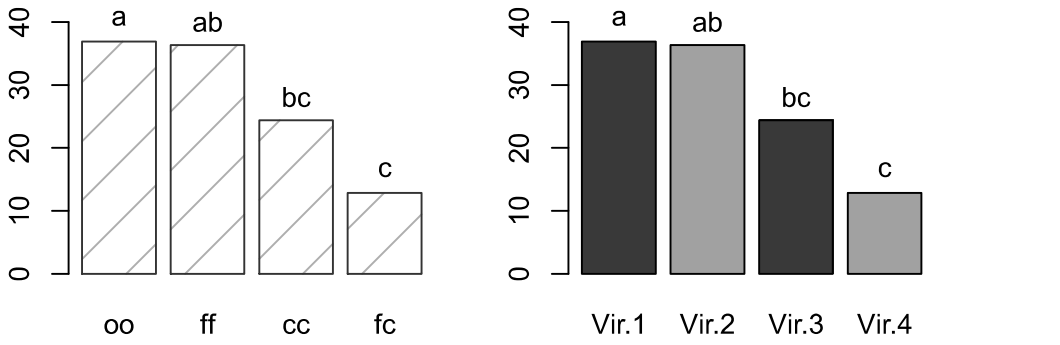
```
out$groups
```

```
#       yield groups
# oo 36.90000      a
# ff 36.33333     ab
# cc 24.40000     bc
# fc 12.86667      c
```

`HSD.test()` has nicely calculated the mean and standard deviations for each group and a description of how the means are grouped, with the groups denoted by letter such that groups that share a letter aren't different. In this case `oo` differs from `cc` and `fc`, and `fc` differs from `oo` and `ff`.

A convenience function is provided which we can use to make a barplot of such an analysis, and all the arguments to `barplot()` can be used to modify this plot.

```
bar.group(out$groups, ylim = c(0, 45), density = 4, border = "blue")
bar.group(out$groups, ylim = c(0, 45), col = c("grey30", "grey70"),
    names.arg = c("Vir.1", "Vir.2", "Vir.3", "Vir.4"), ylab = "some units")
```



Here we demonstrate again on the `warpbreaks` data:

```
model2 <- (lm(breaks ~ wool + tension + wool:tension, data = warpbreaks))
HSD.test(model2, trt = "tension", group = TRUE, main = "Wool x Tension")$groups
```

```
#     breaks groups
# L 36.38889      a
# M 26.38889      b
# H 21.66667      b
```

Note that while the `HSD.test()` function does not work for interactions, we can create an interaction variable using `intearction()` and then use `HSD.test()`.

```
txw <- with(warpbreaks, interaction(wool, tension))
model3 <- aov(breaks ~ txw, data = warpbreaks)
```

```
library(agricolae)
HSD.test(model3, "txw", group = TRUE)$groups
```

```
#       breaks groups
# A.L 44.55556      a
# B.M 28.77778      b
# B.L 28.22222      b
# A.H 24.55556      b
# A.M 24.00000      b
# B.H 18.77778      b
```

Note that this output is much more compact than the output from `TukeyHSD(aov(breaks~wool+tension,data=warpbreaks))` shown earlier, but the conclusion is identical.

12.9 Exercises

1) For the `beans` data we used in Chapter 11 model shoot biomass as a function of phosphorus (phos). Make sure `phos` is coded *as a factor*. Check the coefficient estimates `summary(lm(...))` - what do they suggest?

2) Use Tukey to compare the different levels of `phos` for the model in Problem 1. How does this confirm your answer to #1? Is the responsed similar for Root biomass? Does this same general pattern hold?

3) I kept track of my electric bill every month for over 7 years. The data set ("electric bill.txt") is located in the "Data" directory in essential R, and includes variable for `month`, `year`, the amount of electricity used in kilowatt hours (`kwh`), the number of days in the billing cycle (`days`), and the average temperature during the billing cycle (`avgT`). There are also variables that describe whether the bill was based on an estimated reading or actual reading (`est`, with levels `e` and `a` for estimated or actual), `cost` (in dollars), and energy use per day (`kWhd.1`).

Fit a model of `kwhd.1` as a function of `avgT`. What is the R^2? Test the hypothesis that the slope is -0.25 kwh per day per degree F increase in average monthly temperature. What is the p-value for this test?

4) My old house did not have AC, but we ran several fans in the summer, and the refrigerator and freezer certainly worked harder during the warmer months, so there could be a minimum in energy use at moderate temperatures. Include a quadratic (squared) term for average temperature. How does this change the R^2 value of the model? What is the p-value for the quadratic term? Do the residuals suggest that one model should be favored?

EXTRA) Graph both the linear model (a line) and the quadratic model (a curve) over the data. *Hint* you can use `predict()` and `lines()` like we used

for confidence intervals to add a non-linear regression line, or use the function `curve()` to add curves to plots.

Chapter 15

Visualizing Data I

Enhancing scatter plots

15.1 Introduction

So far we have used R's graphics in fairly straightforward ways to examine data or look for model violations. But sometimes we need to do more. While creating publication quality plots is not something *everyone* will do, being able to create more complex graphics can be a great assist in data analysis, and the ability to customize graphics to visualize data is one of the strengths of R [1].

In this chapter and the next we build up some pretty complex figures from basic building blocks. Some of the examples are fairly complex - but my hope is that you can follow along to see what is possible by combining basic building blocks.

I chose the title "visualizing data" here because while one goal of making figures is communication, I often find (especially with more complex data) that I can understand the data better when I find the right way to visualize it. A resource that may be useful when considering how to visualize some data is the R graph gallery.

There are some R packages that provide more "advanced"" plotting interfaces (e.g. `ggplot2` and `lattice`), and you may want to have a look at these. Here, in the spirit of learning "essential" R, we'll focus on base R. Learning to use the basic plotting functions and graphical parameters can provide great flexibility in visualizing data. In this session we'll focus on scatter plots.

[1] In fact, I often see info-graphics online or in print publications that are almost certainly made with R, in places like the New York Times

15.2 Basic Scatter Plots

15.2.1 A simple scatter plot

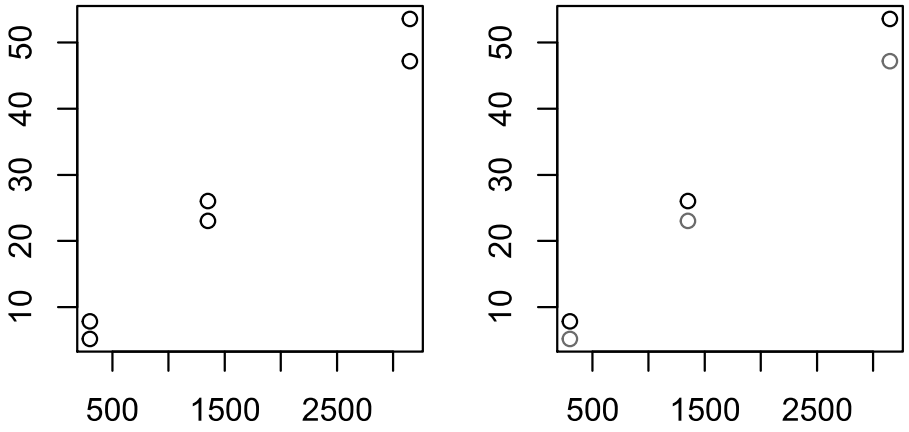
Here we'll look at the relationship between weed seed-bank density (number of weed seeds in soil) and weed density (number of growing weed plants) in conventional and herbicide-free plots. (This data set only contains the means).

```
DT <- read.delim("../Data/DataVizEx1.txt")
summary(DT)
```

```
#  Den   Manag    SeedDen           TotDM             Den1
#  H:2   C:3   Min.   : 300.0   Min.   :  4.84   Min.   : 5.208
#  L:2   O:3   1st Qu.: 562.5   1st Qu.: 26.15   1st Qu.:11.630
#  M:2         Median :1350.0   Median : 45.05   Median :24.521
#              Mean   :1600.0   Mean   : 66.91   Mean   :27.142
#              3rd Qu.:2700.0   3rd Qu.: 85.81   3rd Qu.:41.896
#              Max.   :3150.0   Max.   :187.29   Max.   :53.583
#       Den2             Den3             DenF
#  Min.   : 4.292   Min.   : 0.800   Min.   : 2.120
#  1st Qu.: 7.375   1st Qu.: 3.799   1st Qu.: 4.942
#  Median :15.333   Median : 6.868   Median :10.197
#  Mean   :17.733   Mean   : 7.959   Mean   :11.297
#  3rd Qu.:26.604   3rd Qu.:11.674   3rd Qu.:17.609
#  Max.   :36.354   Max.   :17.167   Max.   :22.000
```

We have seed-bank density as a factor (`Den`) and as a continuous variable (`SeedDen`), and weed density (weeds per meter^2) at four time points (`Den1` to `DenF`). We'll look at the relationship between seed-bank density and weed density at the first count. We'll use `with()` to avoid repetition of `DT$`.

```
with(DT, plot(SeedDen, Den1))  # make the plot
plot(Den1 ~ SeedDen, data = DT, col = Manag)
```

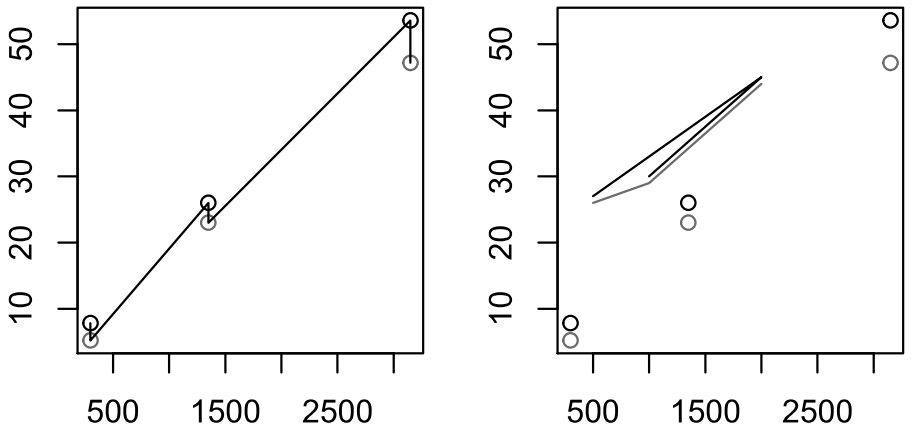


We have two points at each of three levels of weed seed-bank density. Is there a consistent difference between our two treatments - O(rganic) vs. C(onventional)? In the second plot we coded point color by treatment.

We can see that one set of points is consistently above the other. Which is which? They are coded as `C` and `O`, and the level that is first alphabetically (`C` in this case) will be represented in black, the next in red, and so on through all the colors in the palette (use `palette()` to view or change the palette).

Let's try adding some lines here - we'll use `lines()` draws lines (defined by vectors `x=` and `y=`) on an existing plot.

```
with(DT, plot(SeedDen, Den1, col = Manag))
with(DT, lines(SeedDen, Den1))
with(DT, plot(SeedDen, Den1, col = Manag))  # second plot
lines(x = c(1000, 2000, 500), y = c(30, 45, 27))  # not same as
lines(x = c(500, 1000, 2000), y = c(27, 30, 45) - 1, col = "red")
```



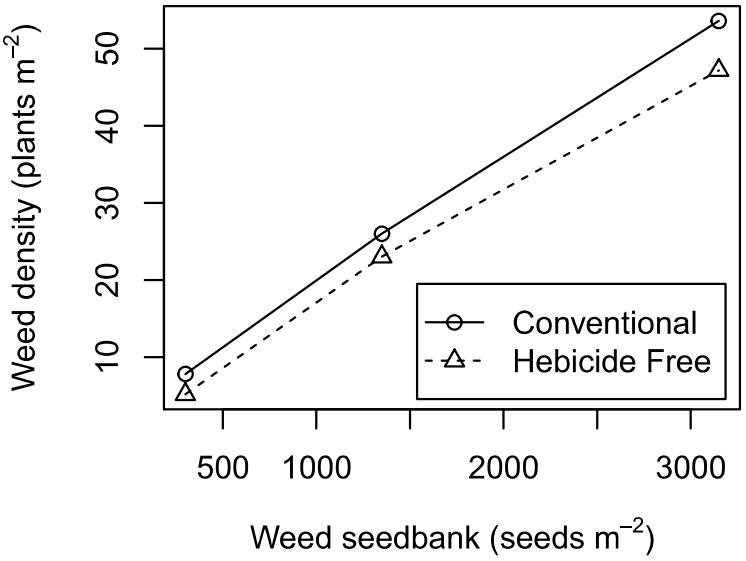

```
# -1 y-axis offset on the red to avoid overplotting
```

This line was not exactly what we had in mind. A quick look at the data is instructive - 300, 300, 1350, 1350, 3150, 3150 - there are 2 values for each x value. The line created by `lines()` follows the `x=` vector literally - 2 at 300, 2 at 1350, etc. The *order of elements* in the `x=` and `y=` arguments matters. We can confirm this by plotting some arbitrary lines on the second plot (the second is offset `-1` in the y direction for clarity). The same three points are plotted but in different order.

15.2.2 A simple scatter plot revisited

Let's try again with the same data. We'll use different symbols for our treatments, and we'll add proper axis labels. We'll fully parse the code following the code block and plot.

```
op <- par(mar = c(4, 4.5, 0.5, 0.5))
sym <- c(21, 24)  # define vector of symbols
with(DT, plot(SeedDen, Den1, pch = sym[Manag], ylab = expression(paste("Weed density"
    " (plants ", m^-2, ")")), xlab = expression(paste("Weed seedbank (seeds ",
    m^-2, ")"))))
# plot with nice axis labels
lines(DT$SeedDen[DT$Manag == "C"], DT$Den1[DT$Manag == "C"], lty = 1)
# add line for trt C
lines(DT$SeedDen[DT$Manag == "O"], DT$Den1[DT$Manag == "O"], lty = 2)
# add line for trt O
legend("bottomright", inset = 0.025, pch = sym, lty = 1:2, c("Conventional",
    "Hebicide Free"))  # add legend
```



That is a basic, but clean plot that clearly shows the differences. There are quite a number of things to note here since we are including many common elements of quality plots (most of which we'll use repeatedly in this chapter and the next): 1. We selected 2 plotting symbols in our vector `sym`, and indexed these based on levels of the variable `Manag` (the `pch=sym[Manag]` in the plot command, analogous to the use of `col=Manag` in the preceding plot). "`pch=`" means "plotting character"; see `?points` for a list of the valid plotting characters. 2. We used `expression()` to allow a superscript in our axis label, and `paste` to combine the mathematical expression (`m^-2`) with the plain text (`"Weed density (plants "` and `")"`) parts of the label [2].
3. We used logical extraction (`[DT$Manag=="C"]`) to specify which part of the data to plot for each of the two lines.

[2]Note that the text string `Weed density","(plants "` is broken into 2 pieces - this is not necessary for R, but results in neater line wrapping in these notes.

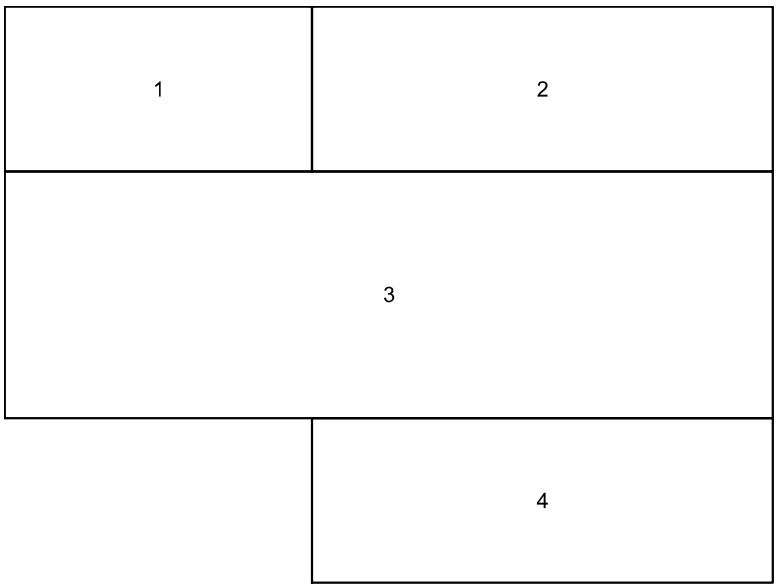
4. `lines()` creates a line from point to point, not a trend-line - we could use `abline(lm(De1~Den2,data=DT)` with the appropriate logical extraction for that).
5. while `legend()` can be complex, in many cases (like this one) it is pretty simple - we just specify line type (`lty=`), plotting symbol (`pch=`), and the legend text.
6. We used `par(mar=c(bottom,left,top,right))` to set the *margin* around each plot. `mar=` is expressed in *lines of text*, so it changes as `cex=` is changed (this assures that margins will be large enough for axis labels). Here we use low values for `top` and `bottom`.

Note: It is a worthwhile exercise is to take the code for this last plot and go through it line by line, commenting out lines and changing arguments to see how they work.

15.3 Multi-Panel Plots I: Layout

It is often the case that we want to show multiple graphs at once in a combined layout. We've seen how `par(mfrow=c(rows,columns))` can be used to split the plotting window. However, the function `layout()` provides a much more flexible tool to do this.

```
layout(matrix(c(1, 2, 3, 3, 0, 4), nrow = 3, byrow = TRUE), heights = c(1,
    1.5, 1), widths = c(1, 1.5))
layout.show(4)  # show the first 4 plots in the layout
```



A couple of things to notice here: 1. `layout()` takes as it's first argument a matrix of plot numbers

2. Plot widths and heights can be manipulated
3. Not all parts of the plotting window need to contain plots
4. `layout.show()` lets us see the layout

Since this layout contains 4 plots, the next 4 times the plot command is called (unless `new=TRUE` is used to force over-plotting - see `?par` or the discussion of adding a second y axis near the end of this chapter for more).

When developing complex graphics, I often find it very useful to force my plot to a specified size. This lets me be sure that my choices for symbol and text size, legend placement, margins, etc. all work together the way I want them to[3]. There are several ways to do this. For developing a plot I use the function `quartz(title,height,width)` [4] (`quartz()` is OSX only) or `x11(title,height,width)` (Linux or Windows) to open a new plotting window whose size I can control. Alternately, in RStudio you can choose "Save plot as image" from the "Export" menu on the "Plots" tab, and then specify the dimensions you want for the plot. If using markdown the chunk options `fig.width=` and `fig.height=` allow you to control plot size in the final document [5].

Here we'll make a multi-panel figure with 3 plots. Each panel will be similar to the scatter plot we made in part 2, but will show other response variables, and the response variables are called by column number rather than by name. As before, we'll parse the code after the plot.

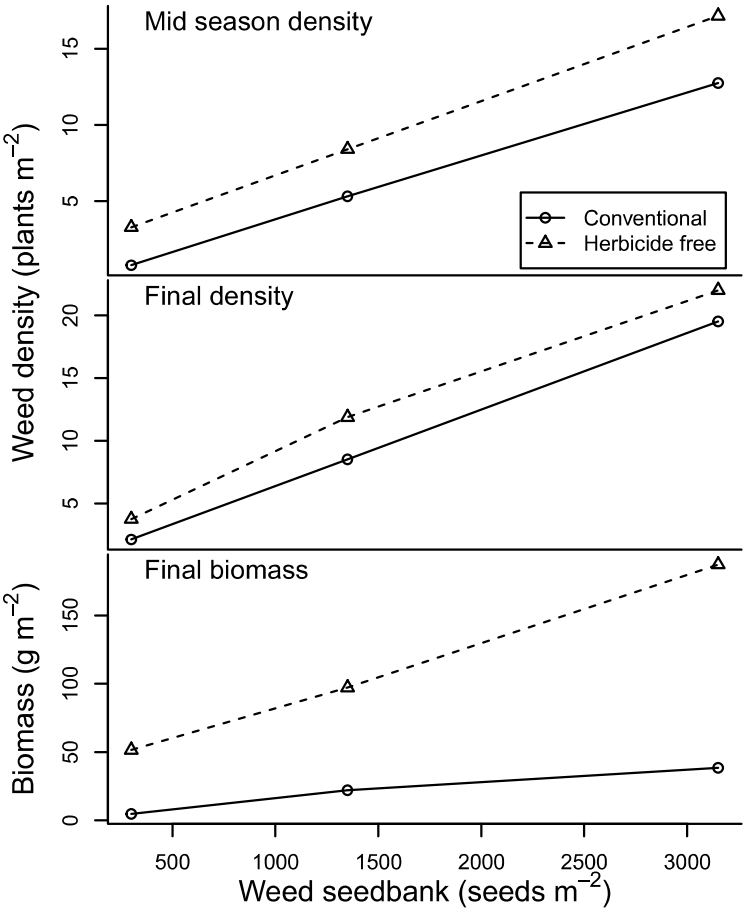
```
layout(matrix(c(1, 2, 3, 0), nrow = 4), heights = c(1, 1, 1, 0.5))
sym <- c(21, 24)  # plotting characters to use
par(mar = c(0.1, 4.3, 0.1, 1), bty = "l")
# set margins and plot frame type plot 1
with(DT, plot(SeedDen, DT[, 7], pch = sym[Manag], xaxt = "n", xlab = "",
    ylab = ""))
lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", 7], lty = 1)
lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", 7], lty = 2)
text(300, max(DT[, 7]) * 0.97, "Mid season density", pos = 4, cex = 1.2)
mtext(side = 2, line = 2.5, at = -1, text = expression(paste("Weed density (plants ",
    m^-2, ")")), cex = 0.9)
legend("bottomright", inset = 0.025, pch = sym, lty = 1:2, c("Conventional",
    "Herbicide free"))
## plot 2
with(DT, plot(SeedDen, DT[, 8], pch = sym[Manag], xaxt = "n", xlab = "",
    ylab = ""))
```

[3] When you change the size of the plot window, graphic elements are re-sized, and sometimes this doesn't make you figure more readable. You can see this by observing how a plot changes when you click the "Zoom" button above the plot and re-size the window.

[4] The units are inches - I have no idea why inches and not cm!

[5] When developing complex graphics, I usually include a call to `quartz()` that specifies the for the figure as the first line in the plot chunk. I also include the the chunk options `fig.width=` and `fig.height=` specifying the same dimensions as the call to `quartz()`, and the chunk option `eval= -1` so the call to `quartz()` is ignored when compiling.

```
lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", 8], lty = 1)
lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", 8], lty = 2)
text(300, max(DT[, 8]) * 0.97, "Final density", pos = 4, cex = 1.2)
## plot 3
with(DT, plot(SeedDen, DT[, 4], pch = sym[Manag], ylab = "", xlab = ""))
lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", 4], lty = 1)
lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", 4], lty = 2)
text(300, max(DT[, 4]) * 0.97, "Final biomass", pos = 4, cex = 1.2)
mtext(side = 1, line = 2.5, text = expression(paste("Weed seedbank (seeds ",
    m^-2, ")")), cex = 0.9)
mtext(side = 2, line = 2.5, text = expression(paste("Biomass (g ",
    m^-2, ")")), cex = 0.9)
```



This is a fair amount of code for just one figure (though a lot of it is repeated between panels). Things to notice: 1. The call to `layout()` creates a space for the common x-axis below the bottom plot - the 0th plot - try

`layout.show(3)`. Alternately we could have skipped `layout()` and used the arguments `oma=c(4.2,0,0,0), mfrow=c(3,1)` in the initial call to `par()`. The approach we used here is a bit more flexible, as we can size the plots differently if we wanted to, which we couldn't do using `mfrow=`. 2. `legend()` was only needed once because all the plots have the same legend.
3. the graphical argument `xaxt="n"` was used to suppress the plotting of the x axis in the first 2 plots.
4. To keep x and y axis labels from printing we used `xlab=""` and `ylab=""` in all the plots. We could have added the axis label for the final plot using `xlab=`, but we would also need to add the argument `xpd=NA` to allow plotting to overflow into the adjacent area.
5. The "Weed density" y axis label was added in plot 2 using `mtext()` - in principle it could have been added in any of the plots, since the `at=` argument allows us to specify where on the axis it will be centered. `at=2` specifies centering at y=2. (Panels 1 & 2 have the same y axis units so we've centered the label over both of them).

15.3.1 Loops for multi-panel figures.

The preceding plot required ~30 lines of code. This is typical for multi-panel plots, at least using base R. `ggplot2` and `lattice` have functions that make something like this simpler, but I usually stick with base R, partly because I know I can add elements to the plot. Notice that when you look at the code, most of it appears repeatedly with only minor variations. This might suggest using a loop. We won't re-create the plot here for brevity's sake, as it is essentially identical to the above plot . But I encourage you to run this code and look at how it works - you'll want to size the plotting window as discussed above.

```
par(mfrow = c(3, 1), oma = c(4.1, 0, 1, 0), mar = c(0.1, 4.3, 0.1,
    1), bty = "l")
vars <- c(7, 8, 4)  # the column # for each response variable
sym <- c(21, 24)  # plotting characters to use
labs <- c("", "", "", "Final biomass", "Rye", "Rye termination", "Mid season density"
    "Final density")  # plot labels
for (i in vars) {
    # begin loop for panels
    with(DT, plot(SeedDen, DT[, i], pch = sym[Manag], xaxt = "n",
        xlab = "", ylab = ""))  # plot the ith column
    lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", i], lty = 1)
    # add lines
    lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", i], lty = 2)
    text(300, max(DT[, i]) * 0.97, labs[i], pos = 4, cex = 1.2)
    if (i == 4) {
        # add x axis for the last plot only (i==4)
        axis(side = 1, at = seq(500, 3000, 500))
        mtext(side = 1, line = 2.5, text = expression(paste("Weed seedbank (seeds ",
```

```
            m^-2, ")")), cex = 0.9)
        mtext(side = 2, line = 2.5, text = expression(paste("Biomass (g ",
            m^-2, ")")), cex = 0.9)
    }
    if (i == 7)
        {
            mtext(side = 2, line = 2.5, at = -1, text = expression(paste("Weed density",
                "(plants ", m^-2, ")")), cex = 0.9)
            # y axis label for first 4 plots
            legend("bottomright", inset = 0.02, legend = c("Conventional",
                "Herbicide free"), pch = sym, lty = c(1, 2), ncol = 2)
        }  # y axis for last plot
}  # end loop for panels
```

Things to notice: 1. the loop is indexed by the vector `vars` which refers to the column numbers of the response variables.
2. the legend in the first plot is side-by-side rather than stacked - the argument `ncol=2` allows this.
3. the code to create the x-axis for the final plot could have been moved outside the loop, avoiding another `if()`.

Was it worth it? The code was somewhat reduced - 18 vs 24 lines of code. I'm not sure that justifies the added complexity. The main benefit (in my experience) is that the "guts" of the plot are only here once - this makes it *much* simpler to change something (plotting character or line type) and keep all the panels consistent. This suggests that it is more likely to be worth using a loop for a figure if all panels are very similar and when there are many panels.

15.4 Adding a Secondary y-axis

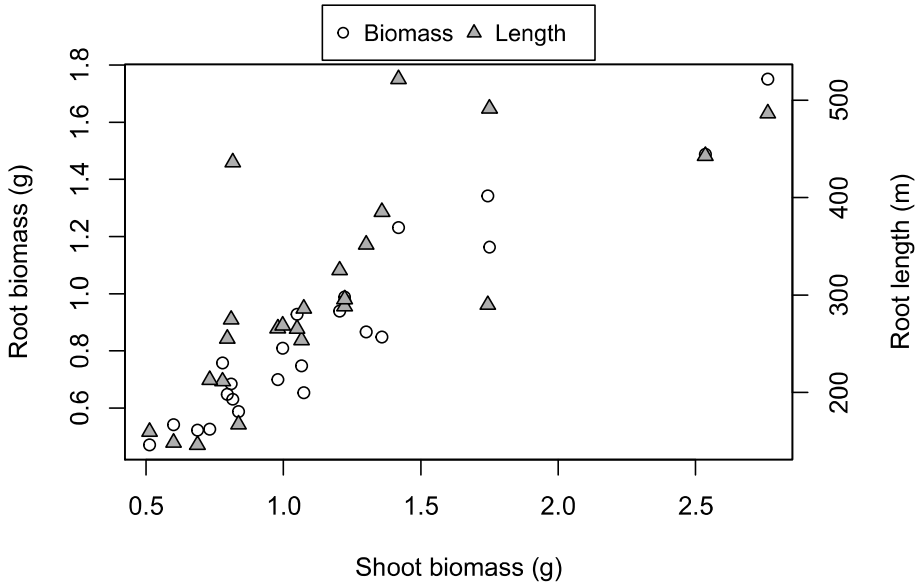
Occasionally we want to add a second y-axis to a plot to plot more than one response variable. Since `plot()` also creates a new coordinate space with appropriate units, this does not seem that it would help us.

One way this can be done is using `points()` to add the second variable, but first we'd need to convert the second response variable to the same scale as the first, and we might need to fool around with the `ylim` argument in the initial plot command. Workable, but probably a hassle.

There is an easier way, though it may seem counter-intuitive. One of the arguments to `par()` is `new=`. A value of `new=TRUE` (counter-intuitively) tells R to treat the current graphics device *as if it were a new device*. This means that a new coordinate space is calculated, and a second plot can be made without the first having been erased.

```
par(mar = c(4.1, 4.1, 3, 4.1))
beans <- read.csv("../Data/BeansData.csv", comm = "#")  # load data
```

```
with(beans, plot(ShtDM, RtDM, xlab = "Shoot biomass (g)", ylab = "Root biomass (g)"))
par(new = TRUE)
with(beans, plot(ShtDM, rt.len, xaxt = "n", yaxt = "n", ylab = "",
    xlab = "", pch = 24, bg = "grey"))
axis(side = 4)
mtext(side = 4, line = 3, "Root length (m)")
legend("top", inset = c(0, -0.15), pch = c(21, 24), pt.bg = c("white",
    "grey"), legend = c("Biomass", "Length"), ncol = 2, xpd = NA,
    cex = 0.9)
```



Notice the `inset=c(0,-0.15)` and `xpd=NA` in the call to `legend()` - This allows the legend to plot in the marginal space, outside the plot area. If you run this command without the `xpd=NA` you will see why it was added. NOTE: These two y variables may not really good candidates for this type of presentation!

In the last two sessions we have covered the most commonly used graphical tools in R (well, in base-R anyway). These are most of the tools you need to make most of the figures you might need to make, and enough base to learn how to make others.

15.5 Summary

In this chapter we've looked at several scatterplots and at how lines, points, and secondary axes can be added to plots. In addition we've explored creating multi-panel plots. These are pretty basic tools which can be applied to a wide range of graphics.

15.6 Exercises

1) We'll revisit the electric bill data once more. In the Chapter 13 exercises we fit an ANCOVA to this data. Plot this ANCOVA (not the residuals), showing the two curves and two parts of the data with distinct symbols, and with properly formatted axes and labels (i.e., Kilowatt hours per day should be shown as "KWHd^{-1}".)

2) Using the electric bill data, plot daily energy use (`kWhd.1`) as a function of average temperature (`avgT`)[6]. Add a second y axis to show cost. Include a legend.

[6]To clarify, this means average temperature is on the x-axis.

Chapter 16

Visualizing Data II

Errorbars and polygons

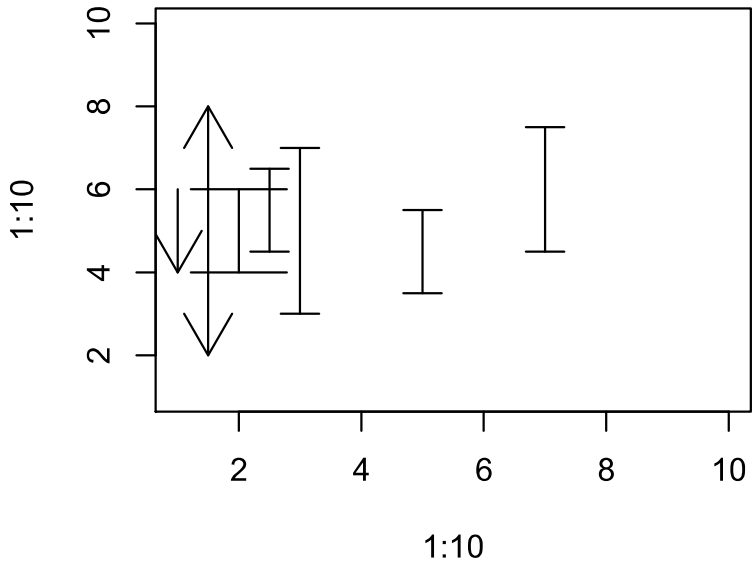
16.1 Introduction

In the last chapter we built some rather complex multiple-panel scatter plots. Here we'll consider some additional ways that scatter plots (x,y plots) can be enhanced, principally by adding error bars and polygons or ribbons.

16.2 Scatter Plot with Error Bars

This figure will illustrate an ANCOVA between the biomass of a rye crop and the the date at which it was cut (covariate) for two different rye planting dates (factor variable). In this figure we'll show the variability of biomass at each cutting date using error bars, and show the ANCOVA line for each of the two levels of planting date in two years. First we'll introduce the function `arrows()` which we'll use to make error bars.

```
op <- par(mar = c(4, 4, 0.5, 0.5))
plot(1:10, 1:10, type = "n")
arrows(x0 = 1, y0 = 6, x1 = 1, y1 = 4)
arrows(1.5, 8, 1.5, 2, code = 3)
arrows(2, 6, 2, 4, code = 3, angle = 90)
arrows(2.5, 6.5, 2.5, 4.5, code = 3, angle = 90, length = 0.1)
x <- c(3, 5, 7)
y <- c(5, 4.5, 6)
z <- c(2, 1, 1.5)
arrows(x, y - z, x, y + z, code = 3, angle = 90, length = 0.1)
```



Things to notice: 1. `arrows()` takes four points, `x0=,y0=,x1=,y1=` that define the ends of the arrows.
2. `code=3` puts arrows at both ends of the arrow
3. `angle=90` makes the arrow look like an error bar
4. `length` controls the length of the arrowhead or crossbar
5. `arrows()` is happy to take a vector as an argument, allowing one call to `arrows()` to create multiple error bars.

To make the plot we first need to load the data and examine it. We'll use a dataset for the plot that includes the means and standard errors, and we'll an anova table to the plot that shows an ANCOVA fit on the full data.

```
RyeMeans <- read.delim("../Data/Rye ANCOVA.txt", comment = "#")
head(RyeMeans)  # examine the data
```

```
#   Term.DOY    YrPd MeanDM   DMsd  DMse n
# 1      117 2008 P1  380.0  81.07 28.66 8
# 2      137 2008 P1  674.3  88.42 31.26 8
# 3      128 2008 P1  590.0  78.25 27.66 8
# 4      149 2008 P1  834.0 131.10 46.36 8
# 5      137 2008 P1  673.3  90.60 32.03 8
# 6      155 2008 P1  984.0 200.90 71.01 8
```

```
RyeMeans$Term.DOY  # not in order
```

```
#  [1] 117 137 128 149 137 155 117 137 128 149 137 155 118 142 128
# [16] 152 140 160 118 142 128 152 140 160
```

```
RyeMeans <- RyeMeans[order(RyeMeans$Term.DOY), ]  # sort the data by Term.DOY
## PLOT
```

```
range(RyeMeans$MeanDM + RyeMeans$DMse)  # ~110 to ~1100)
```

```
# [1]  111.885 1115.230
```

```
range(RyeMeans$Term.DOY)  # find x range (~115-160)
```

```
# [1] 117 160
```

```
levels(RyeMeans$YrPd)
```

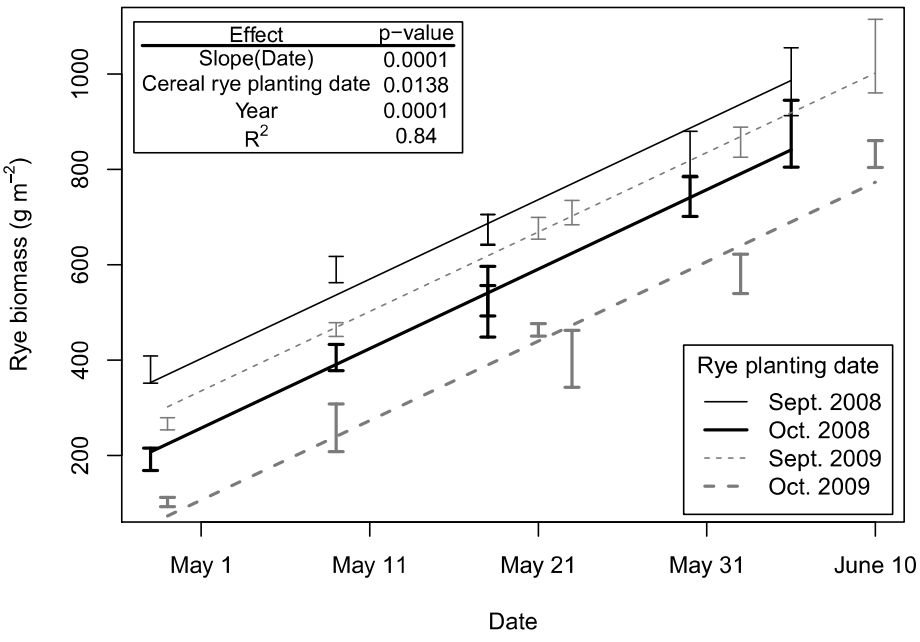
```
# [1] "2008 P1" "2008 P2" "2009 P1" "2009 P2"
```

Things to notice: 1. We sorted the data frame using `RyeMeans<-RyeMeans[order(RyeMeans$Term.DOY),]`. This is just standard R indexing: before the comma we specify the rows; here we just say the rows given by the function `order()` applied to `RyeMeans$Term.DOY`. 2. The variable `YrPd` lists the year and the planting date together as a single factor. This is a convenience for plotting. The original ANCOVA was fit with the year and rye planting date as separate factors, and included their interaction, (which was not significant).

Now we'll plot the data. As before, if you run this code you may want to set your figure size to `width=6.75,height=5.25`.

```
## quartz(height = 5.25, width = 6.75)
# only run in console, not evaluated on compile pdf(file =
# 'RyeAncova.pdf', height=5.25,width=6.75)
par(mar = c(5, 5, 3, 1))  # set margin settings.
with(subset(RyeMeans, YrPd == "2008 P1"), plot(Term.DOY, MeanDM, ylim = c(100,
    1100), xlim = range(RyeMeans$Term.DOY), ylab = expression(paste("Rye biomass (g ",
    m^{
        -2
    }, ")")), xlab = "Date", type = "n", xaxt = "n"))  # subset not necessary here
## for dates on X axis, use this, and add xaxt='n' to above call to
## plot()
axis(side = 1, at = seq(120, 160, by = 10), labels = c("May 1", "May 11",
    "May 21", "May 31", "June 10"))
## add error bars for termination date (Term.DOY) in each treatment
## group (YrPd).
for (i in 1:4) {
    with(subset(RyeMeans, as.numeric(YrPd) == i), arrows(Term.DOY,
        MeanDM + DMse, Term.DOY, MeanDM - DMse, length = 0.05, angle = 90,
        code = 3, lwd = c(1, 2, 1, 2)[i], col = c("black", "black",
            "grey57", "grey57")[i]))
}  # using this loops avoids 4 identical calls to arrows()
legend("bottomright", inset = 0.015, legend = c("Sept. 2008", "Oct. 2008",
    "Sept. 2009", "Oct. 2009"), lwd = c(1, 2, 1, 2), col = c("black",
    "black", "grey57", "grey57"), lty = c(1, 1, 2, 2), title = "Rye planting date")
## ADD lines
```

```
endpoints <- data.frame(YrPd = rep(c("2008 P1", "2008 P2", "2009 P1",
    "2009 P2"), each = 2), Term.DOY = c(117, 155, 117, 155, 118, 160,
    118, 160))
# create df of x value endpoints for lines
endpoints <- cbind(endpoints, RyeDM = predict(lm(MeanDM ~ Term.DOY +
    YrPd, data = RyeMeans), newdata = endpoints))
# create df of x and y value for line endpoints
endpoints <- cbind(endpoints[c(1, 3, 5, 7), ], endpoints[c(2, 4, 6,
    8), -1])
# reorganize this so each row is coordinates for one line
segments(x0 = endpoints[, 2], y0 = endpoints[, 3], x1 = endpoints[,
    4], y1 = endpoints[, 5], col = c("black", "black", "grey57", "grey57"),
    lwd = c(1, 2, 1, 2), lty = c(1, 1, 2, 2))
# draw the lines ADD ANOVA table
legend(121, 1137, legend = c("Effect", "Slope(Date)", "Cereal rye planting date",
    "Year", expression(R^2)), bty = "n", adj = 0.5, cex = 0.9)
# adj=0.5:centered text
legend(130.5, 1137, legend = c("p-value", "0.0001", "0.0138", "0.0001",
    "0.84"), bty = "n", adj = 0.5, cex = 0.9)
rect(116, 835, 135.5, 1110)
lines(x = c(116.5, 135), y = c(1060, 1060), lwd = 2)
```




```
## dev.off()
```

There is an informative plot that shows the variability in the y-axis for each

group at each point in the x-axis. While you never may need to make a plot like this, some elements of this are likely to be useful - for example a regression line that doesn't extend beyond the x-axis range of the data would be generally useful.

Things to notice: 1. the use of `type="n"` to create a blank plot - even though nothing has been plotted, the coordinate space has been created, so `points()`, `lines()`, and `arrows()` can be used.
2. The use of a `for()` loop to draw the errorbars avoids 4 nearly identical calls to `arrows()`. 3. To draw the lines of the ANCOVA model so that they don't extend beyond the data, we can't use `abline()` (in any case, it would not gracefully deal with a complex model like this). We need to use `predict()` with our model, which means we need a data.frame of that includes the terms of the model. It is worth viewing the object `endpoints` when it is created and each of the times it is modified.
4. The modified form of `endpoints` simplifies using `segments()` to draw 4 lines at once. 5. `"grey57"` means a grey that is 57% white, so `"grey90"` is a light grey and `"grey10"` a dark grey.
6. Adding the ANCOVA table to the plot with `legend()` required a bit of trial en error to get the spacing right. The function `addtable2plot()` in the package "plotrix" might make this easier.[1]
7. There are two lines commented out with ###: `pdf(...)` and `dev.off()`. If they were run they would open a pdf graphic device (`pdf()`) of specified size and close that device (`dev.off()`), and all code between them would be sent to that device. Rather than create a plot you can view while making it, they would create a .pdf file. RStudio has nice tools for exporting plots, but is is good to know how to write directly to pdf, in case you need to script making multiple plots (or you are not using RStudio)

16.3 Scatter Plots with Confidence Ribbons

One of the wonderful things about R graphics is how anything is possible. Here we'll make a scatter plot of some data and visualize the confidence interval for around that data with a ribbon, which we'll make using `polygon()` for drawing arbitrary polygons. This example is a plot showing the effect of limestone gravel on the pH of acidic forest soils in PA. Soil was sampled at varying distances (`dist`) from the road[2] on several forest roads in PA, which were surfaced either with shale or limestone gravel.

```
pHmeans <- read.table("../Data/ph data.txt", header = TRUE, sep = "\t")
pHmeans <- pHmeans[pHmeans$side == "down", -3]  # simplify the data
head(pHmeans)
```

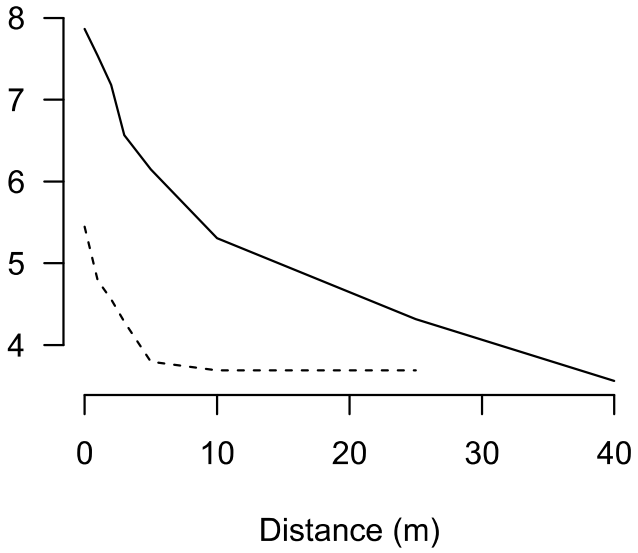
[1] Though using `legend()` is an improvement over just using `text()`, which is how I did it at first.

[2] The measurements were made on both sides of the road (up and down hill), but here we'll use just one.

```
#    trt dist     mean     stdev count      sterr      CI95
# 1    L    0 7.864583 0.2946167    12 0.08504851 0.1666951
# 3    L    1 7.536000 0.4277008    12 0.12346659 0.2419945
# 5    L    2 7.180917 0.4435590    12 0.12804447 0.2509672
# 7    L    3 6.564250 0.5775116    12 0.16671324 0.3267580
# 9    L    5 6.147750 0.7776220    12 0.22448014 0.4399811
# 11   L   10 5.306583 1.0053788    12 0.29022787 0.5688466
```

The data table here includes mean, standard deviation, standard error, and the 95% confidence interval for the mean. First we'll make a basic version of the plot. As above you'll want to set height of the graphics device to 5 inches and the width to 4.

```
xl <- range(pHmeans$dist)  # x limits
par(mar = c(4.1, 4.1, 1, 1))  # set par()
with(subset(pHmeans, trt == "L"), plot(dist, mean, xlab = "Distance (m)",
    ylab = "", type = "l", ylim = range(pHmeans$mean), frame = FALSE,
    las = 1))  # plot
with(subset(pHmeans, trt == "S"), lines(dist, mean, lty = 2))
```

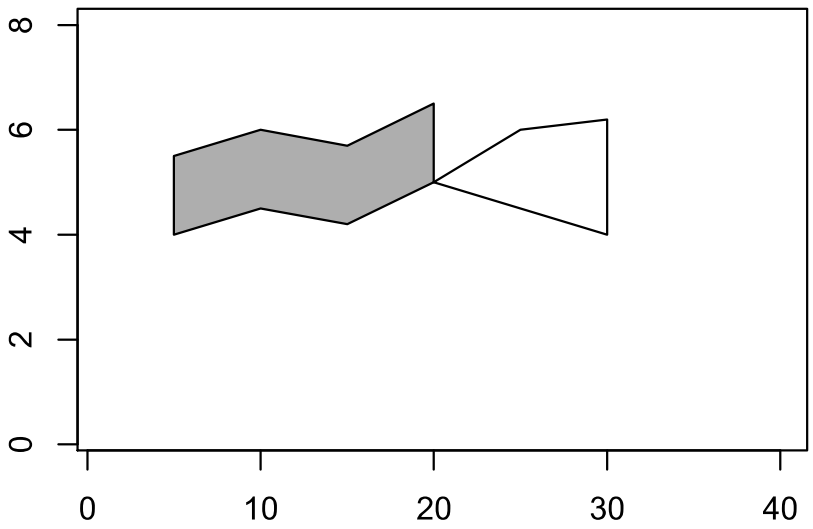


Some things to notice: 1. The argument `las=` forces axis tick labels to be horizontal. 2. the `frame=FALSE` argument suppresses the "box" around the plot.

In order to show the variability around the mean we'll use the function `polygon()`, which creates arbitrary polygons bounded by a series of points of given `x=` and `y=` coordinates, as shown here.

```
plot(1:40, (1:40)/5, type = "n")
polygon(x = c(20, 25, 30, 30, 20), y = c(5, 6, 6.2, 4, 5))
x <- c(5, 10, 15, 20)
```

```
y <- c(4, 4.5, 4.2, 5)
polygon(x = c(x, rev(x)), y = c(y, rev(y + 1.5)), col = "grey")
```



The second call to `polygon()` shows how we can create a "ribbon" using `c()` and `rev()` (which reverses its argument). We'll use the same approach to calculate the ribbons for the limestone and shale roads on the up-slope and down-slope sides of the road. We'll start by calculating the y- and x- values for the polygons.

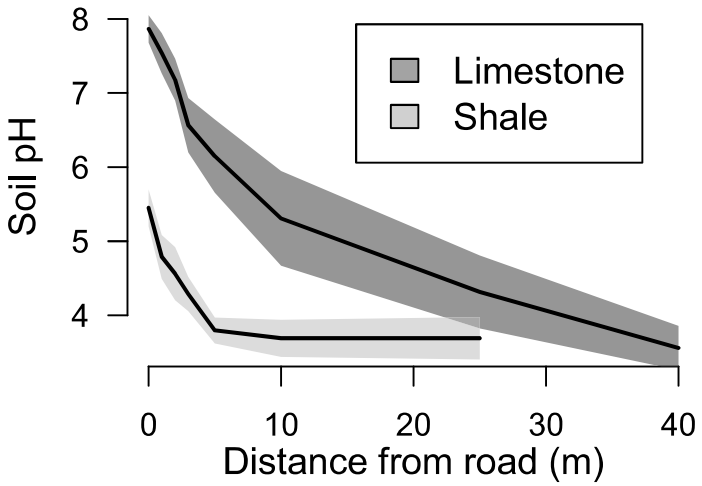
```
L <- with(subset(pHmeans, trt == "L"), c(mean + sterr * qt(p = 0.975,
    df = count - 1), rev(mean - sterr * qt(p = 0.975, df = count -
    1))))
# pH for limestone
S <- with(subset(pHmeans, trt == "S"), c(mean + sterr * qt(p = 0.975,
    df = count - 1), rev(mean - sterr * qt(p = 0.975, df = count -
    1))))
# pH for shale
dds <- with(subset(pHmeans, trt == "S"), c(dist, rev(dist)))
# distances for limestone
ddl <- with(subset(pHmeans, trt == "L"), c(dist, rev(dist)))
# distances for shale
```

Notice that we're using `subset()` to select the relevant part of the data, and then using `c()` and `rev()` to put together the top and the bottom y values (lines 2-5) and to create the x values (last 2 lines). In my head "L" stands for limestone and "S" for shale, and "dds" for distances for shale. The call to `qt()` here is returning the appropriate t-value for a 95% confidence interval given the degrees of freedom.

```
lime = rgb(t(col2rgb("grey44")), alpha = 128, max = 255)
# lime='#70707080'
```

```
shal = rgb(t(col2rgb("darkolivegreen2")), alpha = 128, max = 255)
# '#BCEE6880'
xl <- range(pHmeans$dist)
op <- par(mar = c(4.1, 4.1, 1, 1))

with(subset(pHmeans, trt == "L"), plot(dist, mean, xlab = "", ylab = "",
    type = "n", ylim = c(3.5, 8.25), xlim = xl, las = 1, frame = FALSE))
polygon(ddl, L, col = lime, border = NA)
polygon(dds, S, col = shal, border = NA)
with(subset(pHmeans, trt == "L"), lines(dist, mean, xlab = "", lty = 1,
    lwd = 2))
with(subset(pHmeans, trt == "S"), lines(dist, mean, xlab = "", lty = 1,
    lwd = 2))
legend("topright", inset = 0.1, fill = c(rgb(r = 0.5, g = 0.5, b = 0.5,
    alpha = 0.5), rgb(r = 0.73, g = 0.93, b = 0.41, alpha = 0.7)),
    legend = c("Limestone", "Shale"), cex = 1.2)
mtext(text = "Soil pH", side = 2, cex = 1.2, line = 2.2)
mtext(text = "Distance from road (m)", side = 1, line = 2, cex = 1.2)
```



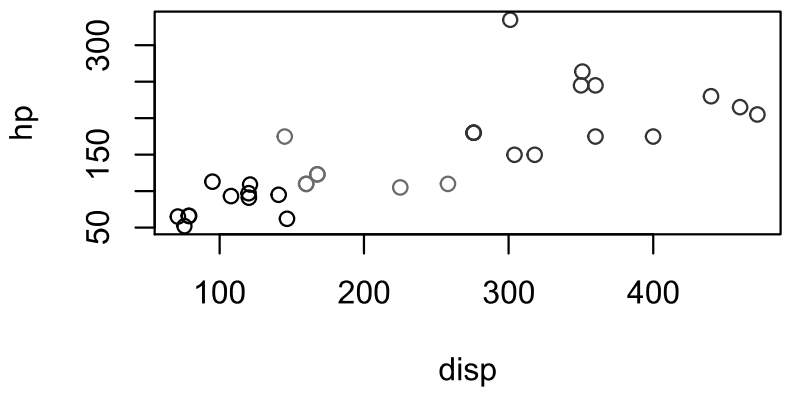
Things to notice here: 1. We want semi-transparent colors in case our ribbons overlap. We use `col2rgb()` to get the rgb colors that correspond to the R colors, and `rgb()` to specify colors with transparency (`alpha<255`, with `max=255`).
2. Since `col2rgb` returns rows and `rgb()` requires columns, `t()` transposes the rows of `col2rgb()` to columns.
3. `col2rgb` returns values from 0-255, so we tell `rgb()` that `max=255` - the default for `rgb()` is a 0-1 scale.
4. We created the polygons first and plotted the lines on top.

The package `ggplot2` has tools for automatically adding ribbons, but now you know how to manipulate arbitrary polygons.

16.4 Error Bars in 2 Dimensions

Sometimes we might want to show errorbars in two dimensions. This is not particularly difficult, it just uses more of what we learned last chapter. We'll demonstrate with the `mtcars` data, and look at horsepower (`hp`) and displacement (`disp`) for cars with differing number of cylinders (`cyl`).

```
data(mtcars)
cols = c("black", "red", "blue")
with(mtcars, plot(disp, hp, col = cols[cyl/2 - 1]))
```

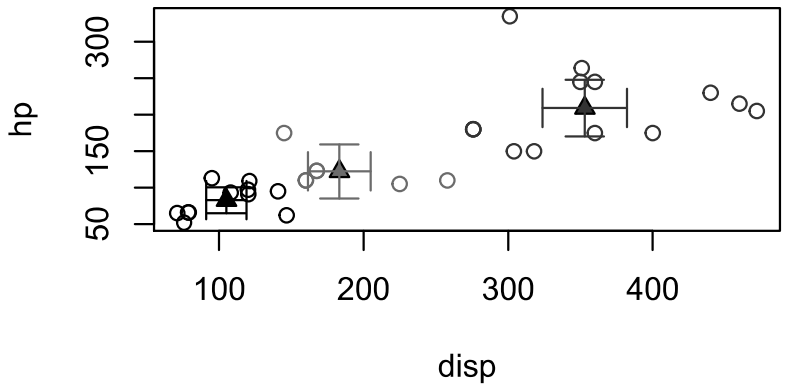


As a quick check we can plot the data and see we have three clusters. How different are they - are their means "significantly different"? First we'll calculate means and standard errors for each group (see Chapter 9).

```
car.means <- aggregate(mtcars[, 3:4], by = mtcars[2], mean)
car.means[, 4:5] <- aggregate(mtcars[, 3:4], by = mtcars[2], function(x) qt(0.025,
    df = length(x), lower.tail = FALSE) * sd(x)/sqrt(length(x)))[2:3]
names(car.means)[4:5] <- c("hp.CI", "disp.CI")
```

Note that here we've actually (correctly) queried the t-distribution for calculating our 95% CI, rather than just using a value of 1.96 from the Z distribution. Now we can plot this data and add errorbars. For interest, let's plot it on top of our previous plot.

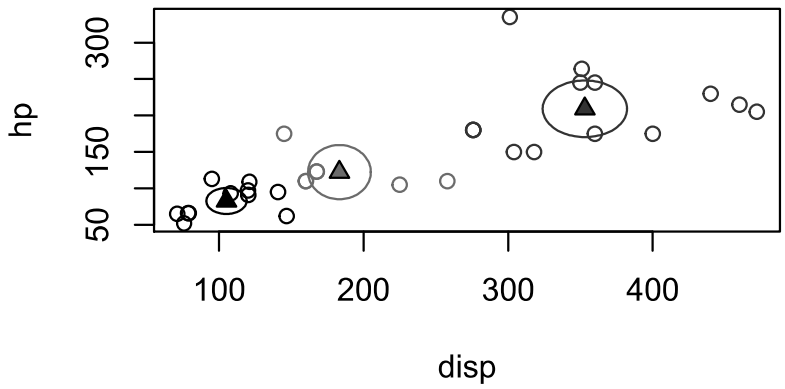
```
with(mtcars, plot(disp, hp, col = cols[cyl/2 - 1]))
with(car.means, points(disp, hp, pch = 24, bg = cols[cyl/2 - 1]))
with(car.means, arrows(disp, hp - hp.CI, disp, hp + hp.CI, code = 3,
    length = 0.1, angle = 90, col = cols[cyl/2 - 1]))
# y-axis error bars
with(car.means, arrows(disp - disp.CI, hp, disp + disp.CI, hp, code = 3,
    length = 0.1, angle = 90, col = cols[cyl/2 - 1]))
```




```
# x-axis error bars
```

An alternate way to show the errorbars in 2 dimensions is using an ellipse. The package `plotrix` has a function for plotting ellipses that we can use. You'll need to run `install.packages("plotrix")` if you haven't already done so.

```
library(plotrix)
with(mtcars, plot(disp, hp, col = cols[cyl/2 - 1]))
with(car.means, points(disp, hp, pch = 24, bg = cols[cyl/2 - 1]))
with(car.means, draw.ellipse(x = disp, y = hp, a = disp.CI, b = hp.CI,
    border = cols, lwd = 1))
```



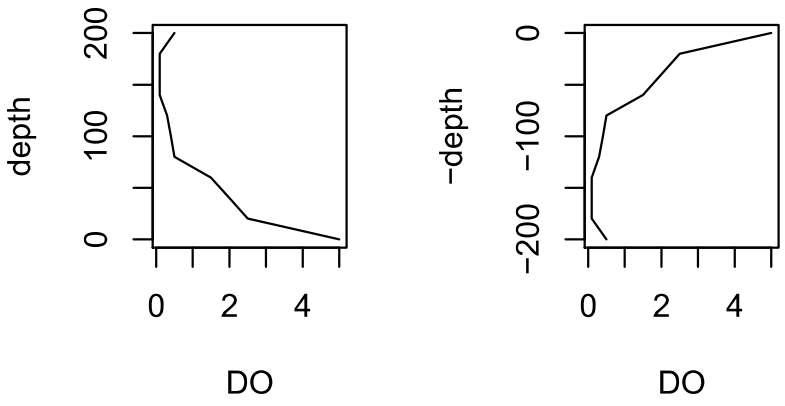
There are other ways to do draw an ellipse, like [directly coding an ellipse] (https://stat.ethz.ch/pipermail/r-help/2006-October/114652.html).

It is important to recall that the ellipses or error bars in these last figures are for the *means*. We don't expect that new samples drawn from these populations will fall within these bounds - in fact, few of our individual samples fall within them. (See discussion of *confidence* vs *prediction* intervals in /@ref(#regression))

16.5 Reversing Axes

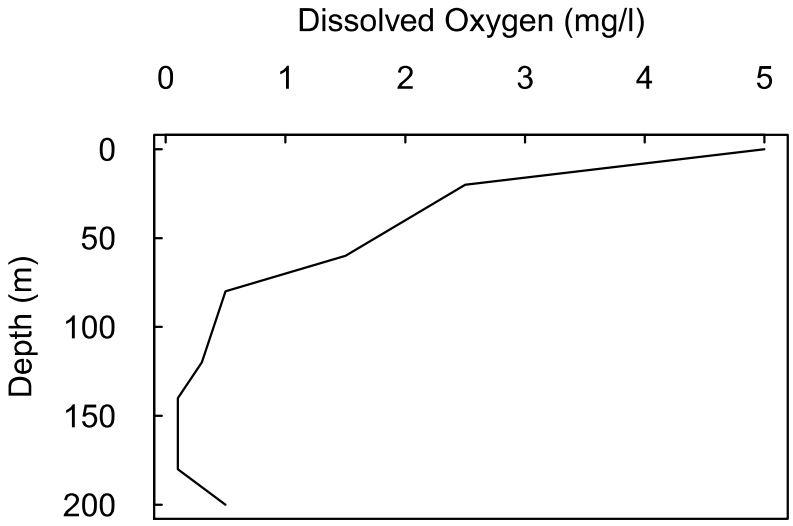
On occasion it make sense to reverse an axis for some reason - maybe to plot something that varies with depth (dissolved oxygen in ocean water [for example] (http://www.oceannetworks.ca/sites/default/files/images/pages/data/Saanich_oxygen_profile.gif)). We'll produce a (crude) facsimile of this figure to demonstrate how to approach this. We'll begin by making up some data.

```
depth <- seq(0, 200, 20)
DO <- c(5, 2.5, 2, 1.5, 0.5, 0.4, 0.3, 0.1, 0.1, 0.1, 0.5)
plot(DO, depth, type = "l")
plot(DO, -depth, type = "l")
```



A simple call to plot is fine for basic visualization, but it is "upside down", rather than what we really want to see. Even plotting the negative of depth only gets us partway to our goal - the axis labels on the y axis should be positive, and the x-axis should be on top.

```
par(mar = c(1, 5, 4, 1), tck = 0.02)  # set margins
plot(DO, depth, type = "l", xaxt = "n", xlab = "", ylab = "Depth (m)",
    ylim = c(200, 0), las = 1)
axis(side = 3)
mtext(side = 3, line = 2.5, "Dissolved Oxygen (mg/l)")
```



Note that here we also reversed the default outside placement of tick marks using the argument `tck=0.02` - this is just to demonstrate that you can change pretty much anything in your plots. The value 0.02 is fraction of the plot width. We also placed the tick mark labels horizontally using the argument `las=1`.Also note that `mtext()` can take vectors as inputs (hardly surprising since this is R). One would use the same approach we used here for reversing the y-axis to reverse the x-axis.

16.6 Summary

In this chapter we've looked at several scatterplots and at how lines, error bars, and polygons can be added to plots. We've also learned how to create semitransparent colors. These are pretty basic tools which can be applied to a wide range of graphics. In the next chapter we'll continue with visualizing data. We've also looked at how we can create multi-panel figures.

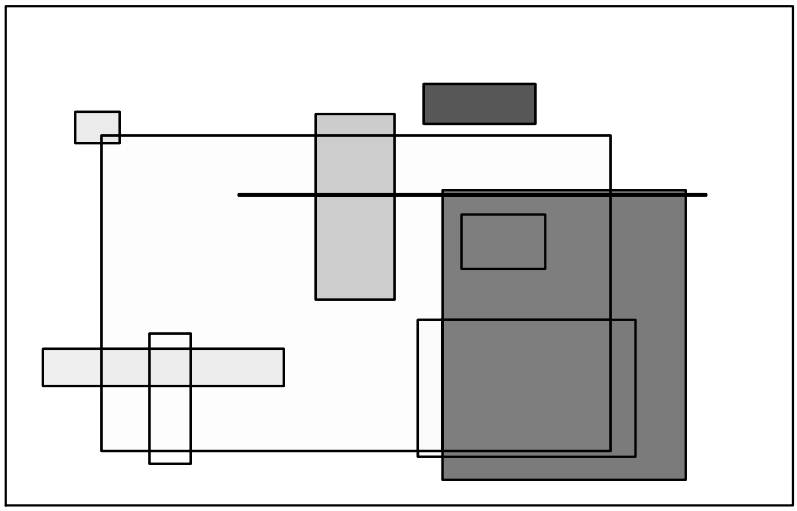
16.7 Fun with R graphics

I created this at one point to see what kind of "art" (or perhaps I should say "aRt"?) I could generate in R with random rectangles and random (semitransparent) fill. It mostly uses things we've looked at in this lesson. Here I've defined it as a function because that makes it easier to run it repeatedly until I like the output. We'll look more at writing functions later. (I make no claim that this is useful, or even art, but it amused me and I learned some things).

```
random.color.boxes = function(n = 10) {
    cols <- colors()[c(1, 26, 552, 652, 454)]
    cols <- col2rgb(cols)  # to find out the rgb code for a color
```

```
    als <- c(rep(0, 5), rep(10, 5), seq(0, 220, 20))
    # rgb(red,green,blue,alpha,max) # to specify color

    par(mar = c(1, 1, 1, 1))
    plot(0:10, 0:10, type = "n", xaxt = "n", yaxt = "n", xlab = "",
        ylab = "")
    cs <- sample(1:5, n, rep = T)
    as <- sample(als, n, rep = T)
    a <- runif(n) * 10
    b <- runif(n) * 10
    c <- runif(n) * 10
    d <- runif(n) * 10
    rect(a, b, c, d, border = "black", col = rgb(cols[1, cs], cols[2,
        cs], cols[3, cs], as, max = 255))
    rect(a, b, c, d, border = "black")  # replot borders
}
random.color.boxes()
```



16.8 Exercises

1) In Chapter 12 we fit a simpler model (only the average temperature and the quadratic term, no factor for insulation). Use `polygon()` to plot the confidence interval for this model and then plot the points over it. *Hint* Use `predict()` to generate the confidence interval.

2) The "ufc" data we examined in last chapter's exercises can be loaded from ufc.csv (in the "data" directory of EssentialR). This contains data on forest trees, including `Species`, diameter (in cm, measured 4.5 feet above ground and known as "diameter at breast height" or `Dbh`), and `height` (in decimeters). Make a

scatterplot showing average `Height` as a function of average `Dbh` for each species. Include x and y errorbars on the plot showing the standard errors.