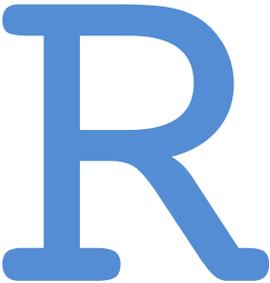


Randall Pruim  
Nicholas J. Horton  
Daniel T. Kaplan

Start  
Teaching with 

Project MOSAIC

# Contents

1	<i>Some Advice on Getting Started With R</i>	10
2	<i>Getting Started with RStudio</i>	15
3	<i>Using R Early in the Course</i>	22
4	<i>What Students Need to Know about R</i>	32
5	<i>What Instructors Need to Know about R</i>	63
6	<i>Getting Interactive with manipulate and shiny</i>	108
7	<i>Bibliography</i>	114
8	<i>Index</i>	115

## About These Notes

These materials were initially created for a workshop entitled *Teaching Statistics Using R* prior to the 2011 United States Conference on Teaching Statistics and revised for the 2013 USCOTS. We organized these workshops to help instructors integrate R (as well as some related technologies) into their statistics courses at all levels. We got great feedback and many wonderful ideas from the participants and those that we've shared this with since the workshops.

We present an approach to teaching introductory and intermediate statistics courses that is tightly coupled with computing generally and with R and RStudio in particular. These activities and examples are intended to highlight a modern approach to statistical education that focuses on modeling, resampling based inference, and multivariate graphical techniques. A secondary goal is to facilitate computing with data through use of small simulation studies, data scraping from the internet and appropriate statistical analysis workflow. This follows the philosophy outlined by Nolan and Temple Lang<sup>1</sup>.

Throughout this book, we introduce multiple activities, some appropriate for an introductory course, others suitable for higher levels, that demonstrate key concepts in statistics and modeling while also supporting the core material of more traditional courses.

### *A Work in Progress*

Consider these notes to be a work in progress. We appreciate any feedback you are willing to share as we continue to work on these materials and the accompanying `mosaic` package. Drop us an email at [pis@mosaic.org](mailto:pis@mosaic.org) with any comments, suggestions, corrections, etc.

Updated versions will be posted at <http://mosaic-web.org>.

### *What's Ours Is Yours – To a Point*

This material is copyrighted by the authors under a Creative Commons Attribution 3.0 Unported License. You are free to *Share* (to copy, distribute and transmit the work) and to *Remix* (to adapt the work) if you attribute our work. More detailed information about the

<sup>1</sup> D. Nolan and D. Temple Lang. Computing in the statistics curriculum. *The American Statistician*, 64(2):97–107, 2010

#### CAUTION!

Despite your best efforts, you WILL find bugs both in this document and in our code. Please let us know when you encounter them so we can call in the exterminators.

licensing is available at this web page: <http://www.mosaic-web.org/go/teachingRlicense.html>.

### *Two Audiences*

The primary audience for these materials is instructors of statistics at the college or university level. A secondary audience is the students these instructors teach. Some of the sections, examples, and exercises are written with one or the other of these audiences more clearly at the forefront. This means that

1. Some of the materials can be used essentially as is with students.
2. Some of the materials aim to equip instructors to develop their own expertise in R and RStudio for their own teaching materials.

Although the distinction can get blurry, and what works “as is” in one setting may not work “as is” in another, we’ll try to indicate which parts of this book fit into each category as we go along.

### *R, RStudio and R Packages*

R can be obtained from <http://cran.r-project.org/>. Download and installation are pretty straightforward for Mac, PC, or linux machines.

RStudio is an integrated development environment (IDE) that facilitates use of R for novice and expert users. We have adopted it as our standard teaching environment because it dramatically simplifies the use of R for instructors and for students. There are several things we use that can only be done in RStudio (mainly things that make use `manipulate()` or RStudio’s support for reproducible research). RStudio is available from <http://www.rstudio.org/>. RStudio can be installed as a desktop (laptop) application or as a server application that is accessible to others via the Internet.

In addition to R and RStudio, we will make use of several packages that need to be installed and loaded separately. The `mosaic` package (and its dependencies) will be used throughout. Other packages appear from time to time as well.

### *Marginal Notes*

Marginal notes appear here and there. Sometimes these are side comments that we wanted to say, but we didn't want to interrupt the flow to mention them in the main text. Others provide teaching tips or caution about traps, pitfalls and gotchas.

Have a great suggestion for a marginal note? Pass it along.

### *Document Creation*

This document was created on May 17, 2014, using `knitr` and R version 3.0.2 Patched (2014-01-13 r64761).

#### DIGGING DEEPER

If you know  $\LaTeX$  as well as R, then `knitr` provides a nice solution for mixing the two. We used this system to produce this book. We also use it for our own research and to introduce upper level students to reproducible analysis methods. For beginners, we introduce `knitr` with RMarkdown, which produces HTML using a simpler syntax.

# *Project MOSAIC*

This book is a product of Project MOSAIC, a community of educators working to develop new ways to introduce mathematics, statistics, computation, and modeling to students in colleges and universities.

The goal of the MOSAIC project is to help share ideas and resources to improve teaching, and to develop a curricular and assessment infrastructure to support the dissemination and evaluation of these approaches. Our goal is to provide a broader approach to quantitative studies that provides better support for work in science and technology. The project highlights and integrates diverse aspects of quantitative work that students in science, technology, and engineering will need in their professional lives, but which are today usually taught in isolation, if at all.

In particular, we focus on:

*Modeling* The ability to create, manipulate and investigate useful and informative mathematical representations of a real-world situations.

*Statistics* The analysis of variability that draws on our ability to quantify uncertainty and to draw logical inferences from observations and experiment.

*Computation* The capacity to think algorithmically, to manage data on large scales, to visualize and interact with models, and to automate tasks for efficiency, accuracy, and reproducibility.

*Calculus* The traditional mathematical entry point for college and university students and a subject that still has the potential to provide important insights to today's students.

Drawing on support from the US National Science Foundation (NSF DUE-0920350), Project MOSAIC supports a number of initiatives to help achieve these goals, including:

*Faculty development and training opportunities*, such as the USCOTS 2011 and 2013 workshops on *Teaching Statistics Using R and RStudio*,

our 2010 Project MOSAIC kickoff workshop at the Institute for Mathematics and its Applications, and our *Modeling: Early and Often in Undergraduate Calculus* AMS PREP workshops offered in 2012 and 2013.

*M-casts*, a series of regularly scheduled webinars, delivered via the Internet, that provide a forum for instructors to share their insights and innovations and to develop collaborations to refine and develop them. A schedule of future M-casts and recordings of past M-casts are available at the Project MOSAIC web site, <http://mosaic-web.org>.

*The development of a “concept inventory” to support teaching modeling.* It is somewhat rare in today’s curriculum for modeling to be taught. College and university catalogs are filled with descriptions of courses in statistics, computation, and calculus. There are many textbooks in these areas and most new faculty teaching statistics, computation, and calculus have a solid idea of what should be included. But modeling is different. It’s generally recognized as important, but few if instructors have a clear view of the essential concepts.

*The construction of syllabi and materials* for courses that teach the MOSAIC topics in a better integrated way. Such courses and materials might be wholly new constructions, or they might be incremental modifications of existing resources that draw on the connections between the MOSAIC topics.

We welcome and encourage your participation in all of these initiatives.

# *Statistical Computation and Computational Statistics*

There are at least two ways in which statistical software can be introduced into a statistics course. In the first approach, the course is taught essentially as it was before the introduction of statistical software, but using a computer to speed up some of the calculations and to prepare higher quality graphical displays. Perhaps the size of the data sets will also be increased. We will refer to this approach as **statistical computation** since the computer serves primarily as a computational tool to replace pencil-and-paper calculations and drawing plots manually.

In the second approach, more fundamental changes in the course result from the introduction of the computer. Some new topics are covered, some old topics are omitted. Some old topics are treated in very different ways, and perhaps at different points in the course. We will refer to this approach as **computational statistics** because the availability of computation is shaping how statistics is done and taught.

In practice, most courses will incorporate elements of both statistical computation and computational statistics, but the relative proportions may differ dramatically from course to course. Where on the spectrum a course lies will be depend on many factors including the goals of the course, the availability of technology for student use, the perspective of the text book used, and the comfort-level of the instructor with both statistics and computation.

Among the various statistical software packages available, R is becoming increasingly popular. The recent addition of RStudio had made R both more powerful and more accessible. Because R and RStudio are free, they have become widely used in research and industry. Training in R and RStudio are often seen as an important additional skill that a statistics course can develop. Furthermore, an increasing number of instructors are using R for their own statistical work, so it is natural for them to use it in their teaching as well. At the same time, the development of R and of RStudio (an optional interface and integrated development environment for R) are making it easier and easier to get started with R.

Nevertheless, those who are unfamiliar with R or who have never used R for teaching are often cautious about using it with students. If you are in that category, then this book is for you. Our goal is to reveal some of what we have learned teaching with R and to make teaching statistics with R as rewarding and easy as possible – for both students and faculty. We will cover both technical aspects of R and RStudio (e.g. how do I get R to do thus and such?) as well as some perspectives on how to use computation to teach statistics. The latter will be illustrated in R but would be equally applicable with other statistical software.

Others have used R in their courses, but have perhaps left the course feeling like there must have been better ways to do this or that topic. If that sounds more like you, then this book is for you, too. As we have been working on this book, we have also been developing the `mosaic` R package (available on CRAN) to make certain aspects of statistical computation and computational statistics simpler for beginners. You will also find here some of our favorite activities, examples, and data sets, as well as answers to questions that we have heard frequently from both students and faculty colleagues. We invite you to scavenge from our materials and ideas and modify them to fit your courses and your students.

# 1

## *Some Advice on Getting Started With R*

Learning R is a gradual process, and getting off to a good start goes a long way toward ensuring success. In this chapter we discuss some strategies and tactics for getting started teaching statistics with R. In subsequent chapters we provide more details about the (relatively few) R commands that students need to know and some additional information about R that is useful for instructors to know. Along the way we present some of our favorite examples that highlight the use of R, including some that can be used very early in a course.

The `mosaic` package includes a vignette outlining a possible minimalist set of R commands for teaching an introductory course.

### *1.1 Strategies*

Each instructor will choose to start his or her course differently, but we offer the following strategies (followed by some tactics and examples) that can serve as a guide for starting the course in a way that prepares students for success with R.

#### 1. Start right away.

Do something with R on day 1. Do something else on day 2. Have students do something by the end of week 1 at the latest.

#### 2. Illustrate frequently.

Have R running every class period and use it as needed throughout the course so students can see what R does. Preview topics by showing before asking students to do things.

#### 3. Teach R as a programming language. (But don't overdo it.)

There is a bit of syntax to learn – so teach it explicitly.

- Emphasize that capitalization (and spelling) matter.
- Explain carefully (and repeatedly) the syntax of functions.

Fortunately, the syntax is very straightforward. It consists of a function name followed by an opening parenthesis, followed by

a comma-separated list of arguments (which may be named), followed by a closing parenthesis.

```
functionname(name1 = arg1, name2 = arg2, ...)
```

Get students to think about what a function does and what it needs to know to do its job. Generally, the function name indicates what the function does. The arguments provide the function with the necessary information to do the task at hand.

- Every object in R has a type (class). Ask frequently: *What type of thing is this?*

Students need to understand the difference between a variable and a data frame and also that there are different kinds of variables (**factor** for categorical data and **numeric** for numerical data, for example). Instructors and more advanced students will want to know about **vector** and **list** objects.

Give more language details in higher level courses.

Upper level students should learn more about user-defined functions and language control structures such as loops and conditionals. Students in introductory courses don't need to know as much about the language.

4. "Less volume, more creativity." [Mike McCarthy, head coach, Green Bay Packers]

Use a few methods frequently and students will learn how to use them well, flexibly, even creatively.

Focus on a small number of data types: numerical vectors, character strings, factors, and data frames. Choose functions that employ a similar framework and style to increase the ability of students to transfer knowledge from one situation to another.

5. Find a way to have computers available for tests.

It makes the test match the rest of the course and is a great motivator for students to learn R. It also changes what you can ask for and about on tests.

One of us first did this at the request of students in an introductory statistics course who asked if there was a way to use computers during the test "*since that's how we do all the homework.*" He now has students bring laptops to class for tests. Another of us has both in-class (without computer) and out-of-class (with computer) components to his assessment.

## 6. Rethink your course.

If you have taught computer-free or computer-light courses in the past, you may need to rethink some things. With ubiquitous computing, some things disappear from your course:

- Reading statistical tables.

Does anyone still consult a table for values of  $\sin$ , or  $\log$ ? All three of us have sworn off the use of tabulations of critical values of distributions (since none of us use them in our professional work, why would we teach this to students?)

- “Computational formulas”.

Replace them with computation. Teach only the most intuitive formulas. Focus on how they lead to intuition and understanding, *not* computation.

- (Almost all) hand calculations.

At the same time, other things become possible that were not before:

- Large data sets
- Beautiful plots
- Simulations and methods based on randomization and resampling
- Quick computations
- Increased focus on concepts rather than calculations

Get your students to think that using the computer is just part of how statistics is done, rather than an add-on.

## 7. Keep the message as simple as possible and keep the commands accordingly simple. Particularly when doing graphics, beware of distracting students with the sometimes intricate details of beautifying for publication. If the default behavior is good enough, go with it.

## 8. Anticipate computationally challenged students, but don’t give in.

Some students pick up R very easily. In every course there will be a few students who struggle. Be prepared to help them, but don’t spend time listening to their complaints. Focus on diagnosing what they don’t know and how to help them “get it”.

In our experience, the computer is often a fall guy for other things the student does not understand. Because the computer gives immediate feedback, it reveals these misunderstandings. For example, if students are confused about the distinctions among

variables, statistics, and observational units, they will have a difficult time providing the correct information to a plotting function. The student may blame R, but that is not the primary source of the difficulty. If you can diagnose the true problem, you will improve their understanding of statistics and fix R difficulties simultaneously.

But even students with a solid understanding of the statistical concepts you are teaching will encounter R errors that they cannot eliminate. Tell students to copy and paste R code and error messages into email when they have trouble. When you reply, explain how the error message helped you diagnose their problem and help them generalize your solution to other situations. See Chapter 5 for some of the common error messages and what they might indicate.

#### TEACHING TIP

When introducing R code to students, we emphasize the following questions: *What do you want R to do for you?* and *What information must you provide, if R is going to do that?* The first question generally determines the function that will be used. The second determines the inputs to that function.

#### TEACHING TIP

Tell your students to copy and paste error messages into email rather than describe them vaguely. It's a big time saver for everyone

## 1.2 Tactics

### 1. Introduce Graphics Early.

Introduce graphics very early, so that students see that they can get impressive output from simple commands. Try to break away from their prior expectation that there is a "steep learning curve."

Accept the defaults – don't worry about the niceties (good labels, nice breaks on histograms, colors) too early. Let them become comfortable with the basic graphics commands and then play (make sure it feels like play!) with fancy things up.

Keep in mind that just because the graphs are easy to make on the computer doesn't mean your students understand how to read the graphs. Use examples that will help students develop good habits for visualizing data.

*Students must learn to see before they can see to learn.*

### 2. Introduce Sampling and Randomization Early.

Since sampling drives much of the logic of statistics, introduce the idea of a random sample very early, and have students construct their own random samples. The phenomenon of a sampling distribution can be introduced in an intuitive way, setting it up as a topic for later discussion and analysis.

## 1.3 Scope of this book

In keeping with this advice, most of the examples in this book fall in the area of exploratory data analysis. The organization is chosen to develop gradually an understanding of R. See the companion volume *A Compendium of Commands to Teach Statistics with R* for a tour of commands used in the primary sorts analyses used in the

first two undergraduate statistics courses. This companion volume is organized by types of data analyses and presumes some familiarity with the R language.

## 2

# Getting Started with RStudio

RStudio is an integrated development environment (IDE) for R that provides an alternative interface to R that has several advantages over other the default R interfaces:

- RStudio runs on Mac, PC, and Linux machines *and looks and feels identical on all of them.*

The default interfaces for R are quite different on the various platforms. This is a distractor for students and adds an extra layer of support responsibility for the instructor.

- RStudio can run in a web browser.

In addition to stand-alone desktop versions, RStudio can be set up as a server application that is accessed via the internet. Installation is straightforward for anyone with experience administering a Linux system. Once set up at your institution, students can start using RStudio by simply opening a website from a browser and logging in. No additional installation or configuration is required.

The web interface is nearly identical to the desktop version. As with other web services, users login to access their account. If students logout and login in again later, even on a different machine, their session is restored and they can resume their analysis right where they left off. With a little advanced set up, instructors can save the history of their classroom R use and students can load those history files into their own workspace.

- RStudio provides support for reproducible research.

RStudio makes it easy to include text, statistical analysis (R code and R output), and graphical displays all in the same document. The RMarkdown system provides a simple markup language and renders the results in HTML. The `knitr`/ $\LaTeX$  system allows users to combine R and  $\LaTeX$  in the same document. The reward for learning this more complicated system is much finer control over

### CAUTION!

The desktop and server version of RStudio are so similar that if you run them both, you will have to pay careful attention to make sure you are working in the one you intend to be working in.

RStudio is like Facebook for statistics.

the output format. Depending on the level of the course, students can use either of these for homework and projects.

We typically introduce students to RMarkdown very early, requiring students to use it for assignments and reports. Handouts, exams, and books like this one are produced using `knitr/LATEX`, and it is relatively easy for interested students to migrate to `knitr` from RMarkdown if they are interested.

- RStudio provides an integrated support for editing and executing R code and documents.
- RStudio provides some useful functionality via a graphical user interface.

RStudio is not a GUI for R, but it does provide a GUI that simplifies things like installing and updating packages; monitoring, saving and loading workspaces; importing and exporting data; browsing and exporting graphics; and browsing files and documentation.

- RStudio provides access to the `manipulate` package.

The `manipulate` package provides a way to create simple interactive graphical applications quickly and easily.

While one can certainly use R without using RStudio, RStudio makes a number of things easier and we highly recommend using RStudio. Furthermore, since RStudio is in active development, we fully expect more useful features in the future.

## 2.1 *Setting up R and RStudio*

R can be obtained from <http://cran.r-project.org/>. Download and installation are pretty straightforward for Mac, PC, or Linux machines. RStudio is available from <http://www.rstudio.org/>. RStudio can be installed as a desktop (laptop) application or as a server application that is accessible to others via the Internet.

### 2.1.1 *RStudio in the cloud*

We primarily use an online version of RStudio. RStudio is a innovative and powerful interface to R that runs in a web browser or on your local machine. Running in the browser has the advantage that you don't have to install or configure anything. Just login and you are good to go. Furthermore, RStudio will "remember" what you were doing so that each time you login (even on a different machine) you can pick up right where you left off. This is "R in the cloud" and works a bit like GoogleDocs or Facebook for R.

To use Markdown or `knitr/LATEX` requires that the `knitr` package be installed on your system. See Section 4.3 for instructions on installing packages.

Your system administrator will likely need to set up your own installation of RStudio for your institution, but we can attest that the process is straightforward and greatly facilitates student and faculty use.

### 2.1.2 RStudio on your computer

There is also a stand-alone version of the RStudio environment that you can install on your desktop or laptop machine. This can be downloaded from <http://www.rstudio.org/>. This assumes that you have a version of R installed on your computer (see below for instructions to download this from CRAN). Even if your students are primarily or exclusively using the server version of RStudio in a browser, instructors may like to have the security blanket of a version that does not require access to the internet. But be warned, the two version look so similar that you may occasionally find yourself working in one of them when you intend to be in the other.

### 2.1.3 Getting R from CRAN

CRAN is the Comprehensive R Archive Network (<http://cran.r-project.org/>). You can download free versions of R for PC, Mac, and Linux from CRAN. (If you use the RStudio stand-alone version, you also need to install R this way first.) All the instructions for downloading and installing are on CRAN. Just follow the appropriate instructions for your platform.

Once you have launched the desktop version of RStudio or logged in to an RStudio server, you will see something like Figure 2.1.

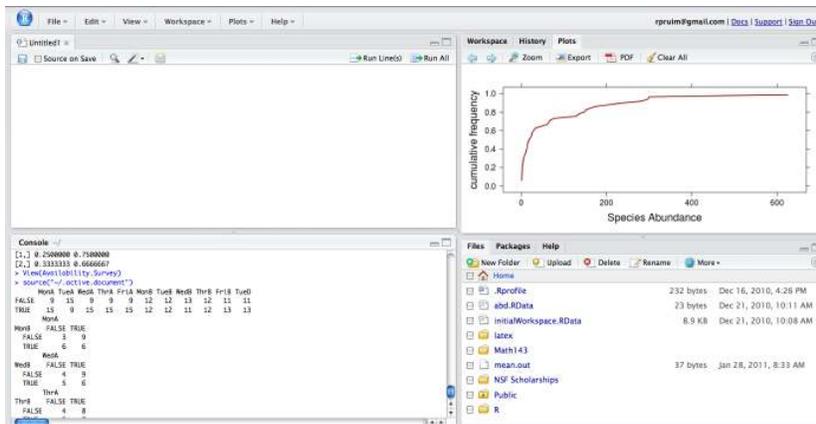


Figure 2.1: Welcome to RStudio.

We find it convenient to put the console in the upper left rather than the default location (lower right) so that students can see it better when we project or R session in class.

Notice that RStudio divides its world into four panels. Several of the panels are further subdivided into multiple tabs. Which tabs appear in which panels can be customized by the user.

## 2.2 Using R as a Calculator

R can do much more than a simple calculator, and we will introduce additional features in due time. But performing simple calculations in R is a good way to begin learning the features of RStudio.

### 2.2.1 Calculating in the Console Tab

Commands entered in the Console tab are immediately executed by R. A good way to familiarize yourself with the console is to do some simple calculator-like computations. Most of this will work just like you would expect from a typical calculator. Try typing the following commands in the console panel.

```
5 + 3
[1] 8

15.3 * 23.4
[1] 358

sqrt(16) # square root
[1] 4
```

### 2.2.2 Working with R Script Files

As an alternative, R commands can be stored in a file. RStudio provides an integrated editor for editing these files and facilitates executing some or all of the commands. To create a file, select File, then New File, then R Script from the RStudio menu. A file editor tab will open in the Source panel. R code can be entered here, and buttons and menu items are provided to run all the code (called sourcing the file) or to run the code on a single line or in a selected section of the file.

### 2.2.3 Working with RMarkdown, and knitr/L<sup>A</sup>T<sub>E</sub>X

A third alternative is to take advantage of RStudio's support for reproducible research. If you already know L<sup>A</sup>T<sub>E</sub>X, you will want to investigate the knitr/L<sup>A</sup>T<sub>E</sub>X capabilities. For those who do not already know L<sup>A</sup>T<sub>E</sub>X, the simpler RMarkdown system provides an easy entry into the world of reproducible research methods. It also pro-

vides a good facility for students to create homework and reports that include text, R code, R output, and graphics.

To create a new RMarkdown file, select File, then New File, then RMarkdown. The file will be opened with a short template document that illustrates the mark up language. Click on Compile HTML to convert this to an HTML file. There is a button the provides a brief description of the mark commands supported, and the RStudio web site includes more extensive tutorials on using RMarkdown.

It is important remember that unlike R scripts, which are executed in the console and have access to the console workspace, RMarkdown and `knitr`/ $\text{\LaTeX}$  files do not have access to the console workspace. This is a good feature because it forces the files to be self-contained, which makes them transferable and respects good reproducible research practices. But beginners, especially if they adopt a strategy of trying things out in the console and copying and pasting successful code from the console to their file, will often create files that are incomplete and therefore do not compile correctly.

One good strategy for getting student to use RMarkdown is to provide them with a template that includes the boiler plate you want them to use, loads any R packages that they will need, sets any `knitr` or R settings they way you prefer them, and has placeholders for the work you want them to do.

**CAUTION!**  
RMarkdown, and `knitr`/ $\text{\LaTeX}$  files do not have access to the console workspace, so the code in them must be self-contained.

## 2.3 *The Other Panels and Tabs*

### 2.3.1 *The History Tab*

As commands are entered in the console, they appear in the History tab. These histories can be saved and loaded, there is a search feature to locate previous commands, and individual lines or sections can be transferred back to the console. Keeping the History tab open will allow students to look back and see the previous several commands. This can be especially useful when commands produce a fair amount of output and so scroll off the screen rapidly. History files can be saved and distributed to students so that they can rerun the code illustrated in class. (Before saving the history, you can remove any lines that you don't want saved to spare your students repeating all of your typing errors.)

An alternative is to produce RMarkdown files in class and make those available. This provides a better mechanism for adding additional comments or instructions.

### 2.3.2 *The Files Tab*

The Files tab provides a simple file manager. It can be navigated in familiar ways and used to open, move, rename, and delete files. In the browser version of RStudio, the Files tab also provides a file upload utility for moving files from the local machine to the server.

### 2.3.3 *The Help Tab*

The Help tab is where RStudio displays R help files. These can be searched and navigated in the Help tab. You can also open a help file using the `?` operator in the console. For example

```
?log
```

Will provide the help file for the logarithm function.

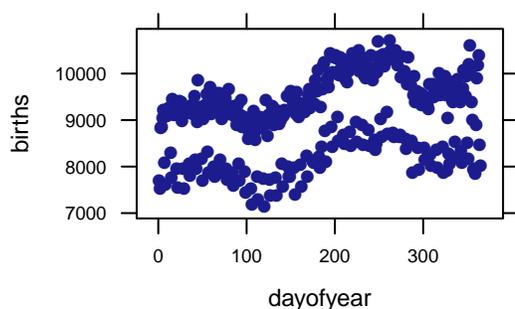
### 2.3.4 *The Workspace Tab*

The Workspace tab shows the objects available to the console. These are subdivided into data, values (non-data frame, non-function objects) and functions. Clicking on a data frame will open it a data viewer. Clicking on other objects will open them in a small editor so you can “fix” them.

### 2.3.5 *The Plots Tab*

Plots created in the console are displayed in the Plots tab. For example,

```
# this will make lattice graphics available to the session
require(mosaic)
xyplot(births ~ dayofyear, data = Births78)
```



---

will display the number of births in the United States for each day in 1978. From the Plots tab, you can navigate to previous plots and also export plots in various formats after interactively resizing them.

### 2.3.6 *The Packages Tab*

Much of the functionality of R is located in packages, many of which can be obtained from a central clearing house called CRAN (Comprehensive R Archive Network). The Packages tab facilitates installing and loading packages. It will also allow you to search for packages that have been updated since you installed them.

## 3

# *Using R Early in the Course*

This chapter includes some of our favorite activities for early in the course. These activities simultaneously provide the students with an introduction to R and to major themes of the course.

### *3.1 Coins and Cups: The Lady Tasting Tea*

There is a famous story about a lady who claimed that tea with milk tasted different depending on whether the milk was added to the tea or the tea added to the milk. The story is famous because of the setting in which she made this claim. She was attending a party in Cambridge, England, in the 1920s. Also in attendance were a number of university dons and their wives. The scientists in attendance scoffed at the woman and her claim. What, after all, could be the difference?

All the scientists but one, that is. Rather than simply dismiss the woman's claim, he proposed that they decide how one should *test* the claim. The tenor of the conversation changed at this suggestion, and the scientists began to discuss how the claim should be tested. Within a few minutes cups of tea with milk had been prepared and presented to the woman for tasting.

At this point, you may be wondering who the innovative scientist was and what the results of the experiment were. The scientist was R. A. Fisher, who first described this situation as a pedagogical example in his 1925 book on statistical methodology <sup>1</sup>. Fisher developed statistical methods that are among the most important and widely used methods to this day, and most of his applications were biological.

You might also be curious about how the experiment came out. How many cups of tea were prepared? How many did the woman correctly identify? What was the conclusion?

Fisher never says. In his book he is interested in the method, not the particular results. But we can use this setting to introduce some key ideas in statistics.

This section is a slightly modified version of a handout R. Pruim has given Intro Stats students on Day 1 after going through the activity as a class discussion.

<sup>1</sup> R. A. Fisher. *Statistical Methods for Research Workers*. Oliver & Boyd, 1925

Let's suppose we decide to test the lady with ten cups of tea. We'll flip a coin to decide which way to prepare the cups. If we flip a head, we will pour the milk in first; if tails, we put the tea in first. Then we present the ten cups to the lady and have her state which ones she thinks were prepared each way.

It is easy to give her a score (9 out of 10, or 7 out of 10, or whatever it happens to be). It is trickier to figure out what to do with her score. Even if she is just guessing and has no idea, she could get lucky and get quite a few correct – maybe even all 10. But how likely is that?

Let's try an experiment. I'll flip 10 coins. You guess which are heads and which are tails, and we'll see how you do.

Comparing with your classmates, we will undoubtedly see that some of you did better and others worse.

Now let's suppose the lady gets 9 out of 10 correct. That's not perfect, but it is better than we would expect for someone who was just guessing. On the other hand, it is not impossible to get 9 out of 10 just by guessing. So here is Fisher's great idea: Let's figure out how hard it is to get 9 out of 10 by guessing. If it's not so hard to do, then perhaps that's just what happened, so we won't be too impressed with the lady's tea tasting ability. On the other hand, if it is really unusual to get 9 out of 10 correct by guessing, then we will have some evidence that she must be able to tell something.

But how do we figure out how unusual it is to get 9 out of 10 just by guessing? We'll learn another method later, but for now, let's just flip a bunch of coins and keep track. If the lady is just guessing, she might as well be flipping a coin.

So here's the plan. We'll flip 10 coins. We'll call the heads correct guesses and the tails incorrect guesses. Then we'll flip 10 more coins, and 10 more, and 10 more, and . . . . That would get pretty tedious. Fortunately, computers are good at tedious things, so we'll let the computer do the flipping for us.

The `rflip()` function can flip one coin

```
require(mosaic)
rflip()

Flipping 1 coin [ Prob(Heads) = 0.5 ] ...

T

Number of Heads: 0 [Proportion Heads: 0]
```

The score is setting up the idea of a test statistic for later.

Have each student make a guess by writing down a sequence of 10 H's or T's while you flip the coin behind a barrier so that the students cannot see the results.

There is a subtle switch here. Before we were asking how many of the students H's and T's matched the flipped coin. Now we are using H to simulate a correct guess and T to simulate an incorrect guess. This makes simulating easier.

or a number of coins

```
rflip(10)

Flipping 10 coins [ Prob(Heads) = 0.5 ] ...

H T H H T H H H T H

Number of Heads: 7 [Proportion Heads: 0.7]
```

Typing `rflip(10)` a bunch of times is almost as tedious as flipping all those coins. But it is not too hard to tell R to `do()` this a bunch of times.

```
do(3) * rflip(10)

  n heads tails prop
1 10     8     2 0.8
2 10     4     6 0.4
3 10     1     9 0.1
```

Notice that `do()` is clever about what information it records. Rather than recording all of the individual tosses, it is only recording the number of flips, the number of heads, and the number of tails.

Let's get R to `do()` it for us 10,000 times and make a table of the results.

```
# store the results of 10000 simulated ladies
random.ladies <- do(10000) * rflip(10)
```

```
tally(~heads, data=random.ladies)

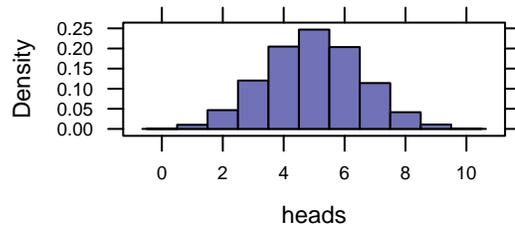
  0    1    2    3    4    5    6    7    8    9   10
5 102  467 1203 2048 2470 2035 1140  415  108    7

# We can also display table using percentages
tally(~heads, data=random.ladies, format='prop')

  0    1    2    3    4    5    6    7
0.0005 0.0102 0.0467 0.1203 0.2048 0.2470 0.2035 0.1140
  8    9   10
0.0415 0.0108 0.0007
```

We can display this table graphically using a plot called a **histogram** with bins of width 1.

```
histogram(~heads, data = random.ladies, width = 1)
```



`histogram()` adds some additional features to `lattice histogram()`. In particular, the `width` and `center` arguments make it easier to control the bins.

You might be surprised to see that the number of correct guesses is exactly 5 (half of the 10 tries) only 25% of the time. But most of the results are quite close to 5 correct. For example, 67% of the results are 4, 5, or 6, for example. About 90% of the results are between 3 and 7 (inclusive). But getting 8 correct is a bit unusual, and getting 9 or 10 correct is even more unusual.

So what do we conclude? It is possible that the lady could get 9 or 10 correct just by guessing, but it is not very likely (it only happened in about 1.2% of our simulations). So *one of two things must be true*:

- The lady got unusually “lucky”, or
- The lady is not just guessing.

Although Fisher did not say how the experiment came out, others have reported that the lady correctly identified all 10 cups! <sup>2</sup>

#### A DIFFERENT DESIGN

Suppose instead that we prepare five cups each way (and that the woman tasting knows this). We give her five cards labeled “milk first”, and she must place them next to the cups that had the milked poured first. How does this design change things?

We could simulate this by shuffling a deck of 10 cards and dealing five of them.

```
cards <- factor(c("M", "M", "M", "M", "M", "T", "T", "T", "T",
                 "T"))
tally(~deal(cards, 5))
```

<sup>2</sup> D. Salsburg. *The Lady Tasting Tea: How statistics revolutionized science in the twentieth century*. W.H. Freeman, New York, 2001

```
M T
3 2
```

The use of `factor()` here let's R know that the possible values are 'M' and 'T', even when only one or the other appears in a given random sample.

```
results <- do(10000) * tally(~deal(cards, 5))
tally(~M, data = results)

  0     1     2     3     4     5 Total
44   993 3966 3927 1028   42 10000

tally(~M, data = results, format = "prop")

  0     1     2     3     4     5 Total
0.0044 0.0993 0.3966 0.3927 0.1028 0.0042 1.0000

tally(~M, data = results, format = "perc")

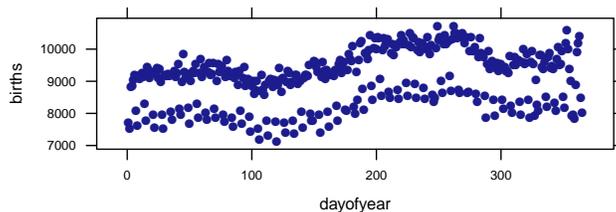
  0     1     2     3     4     5 Total
0.44   9.93 39.66 39.27 10.28   0.42 100.00
```

### 3.1.1 Births by Day

The `Births78` data set contains the number of births in the United States for each day of 1978. A scatter plot of births by day of year reveals some interesting patterns. Let's see how the number of births depends on the day of the year.

The use of the phrase “depends on” is intentional. Later we will emphasize how `~` can often be interpreted as “depends on”.

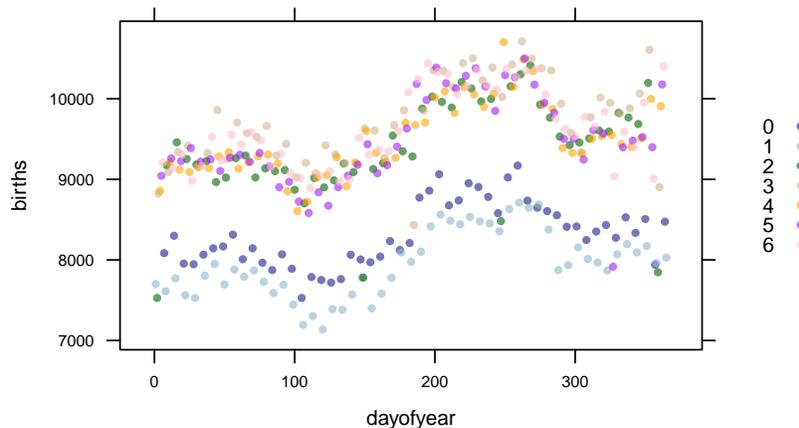
```
xyplot(births ~ dayofyear, data = Births78)
```



When shown this image, students should readily be able to describe two patterns in the data; they should notice both the rise and fall over the course of the year and the two “parallel waves”. Many students will be able to come up with conjectures about the peaks and valleys, but they often struggle to correctly interpret the parallel waves. Having them make conjectures about this will quickly reveal whether they are correctly interpreting the plot.

One conjecture about the parallel waves can be checked using the data at hand. If we display each day of the week with a different symbol or color, we see that there are fewer births on weekends – likely because scheduled births are less likely on weekends. There are a handful of exceptions which are readily seen to be holidays.

```
xyplot(births ~ dayofyear, data=Births78, groups=dayofyear%%7,
       auto.key=list(space="right"))
```



A discussion of this or some other data set that can be explored through graphical displays is a good way to demonstrate “statistical curiosity”, to illustrate the power of R for creating graphs, and to introduce the importance of covariates in statistical analysis.

### 3.2 SAT and Confounding

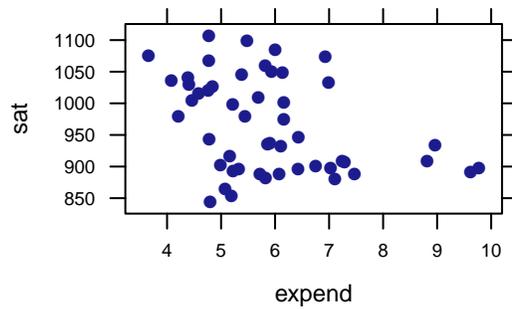
The [SAT](#) data set contains information about the link between SAT scores and measures of educational expenditures. Students are often surprised to see that states that spend more on education do worse on the SAT.

#### TEACHING TIP

This can make a good “think-pair-share” activity. Have students come up with possible explanations, then discuss these explanations with a partner. Finally, have some of the pairs share their explanations with the entire class.

Visualization has been called the “gateway drug” to statistics. It can be a great way to lure students into statistics – and away from their graphing calculators.

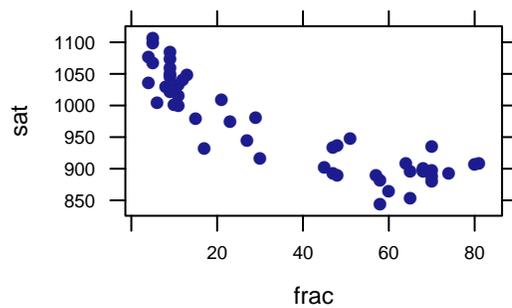
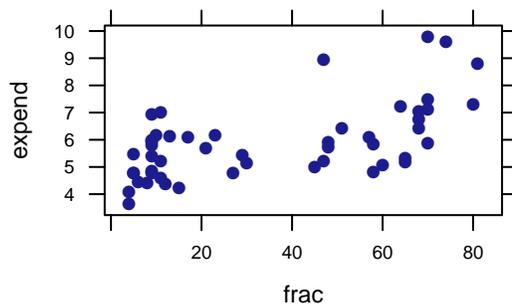
```
xyplot(sat ~ expend, data = SAT)
```



The problem is that expenditures are confounded with the proportion of students who take the exam, and scores are higher in states where fewer students take the exam.

```
xyplot(expend ~ frac, data = SAT)
```

```
xyplot(sat ~ frac, data = SAT)
```



This example can be used to warn against interpreting relationships causally and to illustrate the importance of considering covariates.

### 3.3 *Mites and Wilt Disease*

*This example shows how to build up to statistical inference from first principles.*

Researchers suspect that attack of a plant by one organism induces resistance to subsequent attack by a different organism. Individually potted cotton plants were randomly allocated to two groups: infestation by spider mites or no infestation. After two weeks the mites were dutifully removed by a conscientious research assistant, and both groups were inoculated with *Verticillium*, a fungus that causes Wilt disease. The researchers were hoping the data would shed light on the following big question:

Is there a relationship between infestation and Wilt disease?

The accompanying table shows a cross tabulation the number of plants that developed symptoms of Wilt disease.

```
Mites <- data.frame(
  mites = c(rep("Yes", 11), rep("No", 17),
            rep("Yes", 15), rep("No", 4)),
  wilt = c(rep("Yes", 28), rep("No", 19))
)
tally(~ wilt + mites, Mites)
```

	mites	
wilt	No	Yes
No	4	15
Yes	17	11

Students can begin exploring this data by answering the following questions.

1. Here, what do you think is the explanatory variable? Response variable?
2. What proportion of the plants in the study with mites developed Wilt disease?
3. What proportion of the plants in the study with no mites developed Wilt disease?

4. Relative risk is the ratio of two risk proportions. What is the relative risk of developing Wilt disease, comparing mites to no mites?
5. If there were no association between mites and Wilt disease, what would the relative risk be (in the population as a whole)? How close is the relative risk computed from the data to this value?
6. Let  $X$  be the number of plants in the no mites group that did not develop Wilt disease. What are the possible values for  $X$ ?
7. Assuming a population relative risk of 1, give two possible values for  $X$  that would be more unusual than the value for these data?

Now we can set up a randomization simulation using some cards.

#### Physical Simulation

1. Select 47 cards from your deck, 26 red (mites!) and 21 black
2. Shuffle the cards well
3. Deal out 19 cards, these represent the 19 plants without Wilt disease.
4. Count the number of black cards among those 19. What do these represent?
5. Repeat steps 2–4, five times.

Students can pool their results by recording them in a table on the board at the front of the room. Then have students process the results by answering the following questions.

8. How many black cards would we expect (on average)? Why?
9. What did we observe?
10. How would we summarize these results? What is the big idea?

Once the simulation with cards has been completed, we can use R to do many more simulations very quickly.

### Computational Simulation

```
tally(~ wilt + mites, data=Mites)

      mites
wilt  No Yes
No    4  15
Yes  17  11

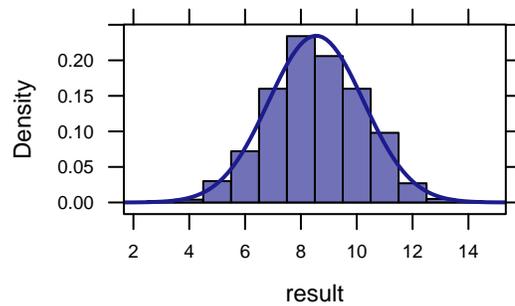
X <- tally(~ wilt + mites, data=Mites)["No","No"]
X
[1] 4

nullDist <- do(1000) *
  tally(~ wilt + shuffle(mites), data=Mites)["No","No"]

Loading required package: parallel

histogram(~ result, data=nullDist,
          width=1, type="density", fit="normal")

Loading required package: MASS
```



## 4

# *What Students Need to Know About R & How to Teach It*

### *4.1 Using R as a Calculator*

Most students are familiar and comfortable with their hand-held calculators, so it is natural to begin an introduction to R by showing them how R can be used to perform typical calculator operations.

Try typing the following commands in the console panel.

```
5 + 3  
[1] 8  
15.3 * 23.4  
[1] 358  
sqrt(16) # square root function  
[1] 4
```

This last example demonstrates how functions are called within R as well as the use of comments. Comments are prefaced with the # character. Comments can be very helpful when writing scripts with multiple commands or to annotate example code for your students.

You can save values to named variables for later reuse.

```
product = 15.3 * 23.4 # save result  
product # display the result  
[1] 358  
product <- 15.3 * 23.4 # <- can be used instead of =  
product  
[1] 358
```

Once variables are defined, they can be referenced in other operations and functions.

```
0.5 * product           # half of the product
[1] 179

log(product)           # (natural) log of the product
[1] 5.881

log10(product)        # base 10 log of the product
[1] 2.554

log(product, base=2)  # base 2 log of the product
[1] 8.484
```

The semi-colon can be used to place multiple commands on one line. One frequent use of this is to save and print a value all in one go:

```
15.3 * 23.4 -> product; product # save result and show it
[1] 358
```

## 4.2 Four Things to Know About R

As is true for most computer languages, R has to be used on its terms. R does not learn the personality and style of its users. Getting along with R is much easier if you keep in mind (and remind your students about) a few key features of the R language.

### 1. R is case-sensitive

If you mis-capitalize something in R it won't do what you want. Unfortunately, there is not a consistent convention about how capitalization should be used, so you just have to pay attention when encountering new functions and data sets.

### 2. Functions in R use the following syntax:

#### TEACHING TIP

It's probably best to settle on using one or the other of the right-to-left assignment operators rather than to switch back and forth. The authors have different preferences: two of us find the equal sign to be simpler for students and more intuitive, while the other prefers the arrow operator because it represents visually what is happening in an assignment, because it can also be used in a left to right manner, and because it makes a clear distinction between the assignment operator, the use of = to provide values to arguments of functions, and the use of == to test for equality.

#### TEACHING TIP

Some students will be slow to catch on to the importance of capitalization. So you may have to remind them several times early on.

```
functionname(argument1, argument2, ...)
```

- The arguments are always surrounded by (round) parentheses and separated by commas.  
Some functions (like `data()`) have no required arguments, but you still need the parentheses.
  - If you type a function name without the parentheses, you will see the *code* for that function (this generally isn't what you want unless you are curious about how something is implemented).
3. TAB completion and arrows can improve typing speed and accuracy.  
If you begin a command and hit the TAB key, R and RStudio will show you a list of possible ways to complete the command. If you hit TAB after the opening parenthesis of a function, RStudio will display the list of arguments it expects.  
The up and down arrows can be used to retrieve past commands when working in the console.
4. If you see a `+` prompt, it means R is waiting for more input.  
Often this means that you have forgotten a closing parenthesis or made some other syntax error. If you have messed up and just want to get back to the normal prompt, press the escape key and start the command fresh.

### 4.3 Installing and Using Packages

R is open source software. Its development is supported by a team of core developers and a large community of users. One way that users support R is by providing **packages** that contain data and functions for a wide variety of tasks. As an instructor, you will want to select a few packages that support the way you want to teach your course.

If you need to install a package, most likely it will be on CRAN, the Comprehensive R Archive Network. Before a package can be used, it must be **installed** (once per computer or account) and **loaded** (once per R session). Installing downloads the package software and prepares it for use by compiling (if necessary) and putting its components in the proper location for future use. Loading makes a previously installed package available for use in an R session.

For example, to use the `mosaic` package, we must first install it:

```
install.packages("mosaic") # fetch package from CRAN
```

and then load it:

#### TEACHING TIP

Introduce functions by emphasizing the questions *What do we want the computer to do?* and *What information does the computer need to compute this?* The answer to the first question determines the function to use. The answer to the second question determines what the arguments must be.

#### CAUTION!

Your students will sometimes find themselves in a syntactic hole from which they cannot dig out. Teach them about the ESC key early.

#### TEACHING TIP

If you set up an RStudio server, you can install all of the packages you want to use. You can even configure the server to autoload packages you use frequently. Students who use R on their desktop machines will need to know how to install and load these packages, however.

```
require(mosaic) # load the package before use.
```

The Packages tab in RStudio makes installing and loading packages particularly easy and avoids the need for `install.packages()` for packages on CRAN, and makes loading packages into the console as easy as selecting a check box. The `require()` function is still needed to load packages within RMarkdown, `knitr`/ $\text{\LaTeX}$ , and script files.

If you are running on a machine where you don't have privileges to write to the default library location, you can install a personal copy of a package. If the location of your personal library is first in `R_LIBS`, this will probably happen automatically. If not, you can specify the location manually:

```
install.packages("mosaic", lib = "~/R/library")
```

Occasionally you might find a package of interest that is not available via a repository like CRAN. Typically, if you find such a package, you will also find instructions on how to install it. If not, you can usually install directly from the zipped up package file.

```
# repos = NULL indicates to use a file, not a repository
install.packages("some-package.tar.gz", repos = NULL)
```

From this point on, we will assume that the `mosaic` package has been installed and loaded.

## 4.4 Getting Help

If something doesn't go quite right, or if you can't remember something, it's good to know where to turn for help. In addition to asking your friends and neighbors, you can use the R help system.

### 4.4.1 ?

To get help on a specific function or data set, simply precede its name with a `?`:

```
?log # help for the log function
```

### CAUTION!

Remember that in RMarkdown and Rnw files, any packages you use must be loaded within the file.

```
?HELPrct # help on a data set in the mosaic package
```

This will give you the documentation for the object you are interested in.

#### 4.4.2 *apropos()*

If you don't know the exact name of a function, you can give part of the name and R will find all functions that match. Quotation marks are mandatory here.

```
apropos("tally") # must include quotes. single or double.
[1] "statTally" "tally"
```

#### 4.4.3 *?? and help.search()*

If that fails, you can do a broader search using `??` or `help.search()`, which will find matches not only in the names of functions and data sets, but also in the documentation for them. Quotation marks are optional here.

#### 4.4.4 *Examples and Demos*

Many functions and data sets in R include example code demonstrating typical uses. For example,

```
example(histogram)
```

will generate a number of example plots (and provide you with the commands used to create them). Examples such as this are intended to help you learn how specific R functions work. These examples also appear at the end of the documentation for functions and data sets.

The `mosaic` package (and some other packages as well) also includes demos. Demos are bits of R code that can be executed using the `demo()` command with the name of the demo. To see how demos work, give this a try:

```
demo(lattice)
```

Demos are intended to illustrate a concept, a method, or some such thing, and are independent of any particular function or data set.

You can get a list of available demos using

Not all package authors are equally skilled at creating examples. Some of the examples are nonexistent or next to useless, others are excellent.

```
demo() # all demos
demo(package = "mosaic") # just demos from mosaic package
```

## 4.5 Data

### 4.5.1 Data in Packages

Data sets in R packages are the easiest to deal with. In section 4.8, we'll describe how to load your own data into R and RStudio, but we recommend starting with data in packages, and that is what we will do here, too. Once students know how to work with data and what data in R are supposed to look like, they will be better prepared to import their own data sets.

Many packages contain data sets. You can see a list of all data sets in all loaded packages using

```
data()
```

You can optionally choose to restrict the list to a single package:

```
data(package = "mosaic")
```

Typically (provided the author of the package allowed for lazy loading of data) you can use data sets by simply typing their names. But if you have already used that name for something or need to refresh the data after making some changes you no longer want, you can explicitly load the data using the `data()` function with the name of the data set you want.

```
data(Births78)
```

There is no visible effect of this command, but the `Births78` data frame has now been reloaded from the `mosaic` package and is ready for use. Anything you may have previously stored in a variable with this same name is no longer available.

### 4.5.2 Data Frames

Data sets are usually stored in a special structure called a **data frame**.

#### TEACHING TIP

Start out using data in packages and show students how to import their own data once they understand how to work with data.

#### TEACHING TIP

Students who collect their own data, especially if they store it in Excel, are unlikely to put data into the correct format unless explicitly taught to do so.

Data frames have a 2-dimensional structure.

- Rows correspond to **observational units** (people, animals, plants, or other objects we are collecting data about).
- Columns correspond to **variables** (measurements collected on each observational unit).

The `Births78` data frame contains three variables measured for each day in 1978. There are several ways we can get some idea about what is in the `Births78` data frame.

```
head(Births78)
```

```
  date births dayofyear
1 1/1/78  7701         1
2 1/2/78  7527         2
3 1/3/78  8825         3
4 1/4/78  8859         4
5 1/5/78  9043         5
6 1/6/78  9208         6
```

```
summary(Births78)
```

```
   date      births      dayofyear
1/1/78 : 1  Min.   : 7135  Min.   : 1
1/10/78: 1  1st Qu.: 8554  1st Qu.: 92
1/11/78: 1  Median : 9218  Median :183
1/12/78: 1  Mean   : 9132  Mean   :183
1/13/78: 1  3rd Qu.: 9705  3rd Qu.:274
1/14/78: 1  Max.   :10711  Max.   :365
(Other):359
```

In interactive mode, you can also try

```
?Births78
```

to access the documentation for the data set. This is also available in the `Help` tab. Finally, the `Workspace` tab provides a list of data in the workspace. Clicking on one of the data sets brings up the same data viewer as

#### TEACHING TIP

To help students keep variables and data frames straight, and to make it easier to remember the names, we have adopted the convention that data frames in the `mosaic` package are capitalized and variables (usually) are not. This convention has worked well, and you may wish to adopt it for your data sets as well.

```
View(Births78)
```

We can gain access to a single variable in a data frame using the `$` operator using the syntax

```
dataframe$variable
```

For example,

```
Births78$births
```

shows the contents of the `births` variable in `Births78` data set. Listing the entire set of values for a particular variable isn't very useful for a large data set. We would prefer to compute numerical or graphical summaries. We'll do that shortly.

### 4.5.3 The Perils of `attach()`

The `attach()` function in R can be used to make objects within data frames accessible in R with fewer keystrokes, but we strongly discourage its use, as it often leads to name conflicts and other complications. The Google R Style Guide<sup>1</sup> echoes this advice, stating that

*The possibilities for creating errors when using `attach()` are numerous. Avoid it.*

It is far better to directly access variables using the `$` syntax or to use functions that allow you to avoid the `$` operator.

## 4.6 Graphical Summaries of Data

After introducing students to the calculator functions of the R console and teaching them how to inspect a data frame, we like to get them making graphical summaries of data. This gives the students access to functionality where R really shines (and is certainly much better than a hand-held calculator). It also begins to develop their ability to interpret graphical representations of data, to think about distributions, and to pose statistical questions.

### 4.6.1 Lattice Graphics

There are several ways to make graphs in R. One approach is a system called `lattice` graphics. Whenever the `mosaic` package is loaded, the `lattice` package is also loaded. One of the attractive aspects of `lattice` plots is that they make use of a **formula interface**

An alternative is to use the `with()` function.

As we will see, there are relatively few instances where one needs to use the `$` operator.

**CAUTION!**  
Avoid the use of `attach()`.

<sup>1</sup> <http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html>

similar to that used for numerical summaries (using `mosaic`) and modeling.

All the plots we need can be created with essentially the same syntactic structure:

```
plotname( y ~ x | z, data=dataframe,
          groups=grouping_variable, ...)
```

- Here are the names of several `lattice` plots:
  - `dotPlot()` (notice the capital P)<sup>2</sup>
  - `histogram()` or `histogram()` (for histograms)<sup>3</sup>
  - `densityplot()` (for density plots)
  - `freqpolygon()` (for frequency polygons)
  - `bwplot()` (for boxplots)
  - `xyplot()` (for scatter plots)
  - `qqmath()` (for quantile-quantile plots)
- `x` is the name of the variable that is plotted along the horizontal ( $x$ ) axis.
- `y` is the name of the variable that is plotted along the vertical ( $y$ ) axis. (For some plots, such as histograms, this slot is empty because R computes these values from the values of `x`.)
- `z` is a conditioning variable used to split the plot into multiple subplots called **panels**.
- `grouping_variable` is used to display different groups differently (different colors or symbols, for example) within the same panel.
- `...` There are many additional arguments to these functions that let you control just how the plots look. (But we'll focus on the basics for now.)

#### 4.6.2 The shape of a distribution: Histograms and their kin

A number of plots can be used to visualize the “shape” of a distribution, including dot plots, histograms, density plots and frequency polygons. Of these, the histogram is probably the most commonly used, but there are situations when one of the others is a better alternative. Each can be created in R using a simple command structure

```
plotname(~x, data = ...)
```

#### TEACHING TIP

Emphasize repeatedly the commonalities between the various commands that use this formula interface.

<sup>2</sup> `dotPlot()` is in the `mosaic` package and was created because `dotplot()` in the `lattice` package makes something different from what we call a dot plot.

<sup>3</sup> `histogram()` is a `mosaic` function that adds some extra functionality to `histogram()`.

#### CAUTION!

Even when `y` is missing, the `~` is still mandatory.

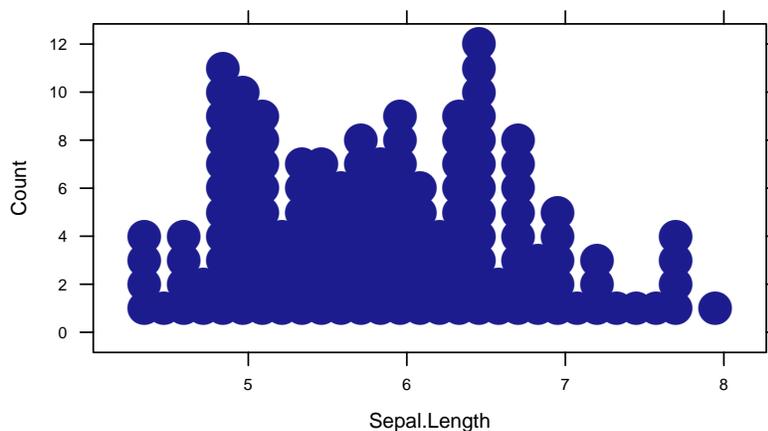
Since for each of these plots the values along the  $y$  axis are computed rather than provided in the data, the  $y$  slot in the formula is empty.

DOT PLOTS: `dotPlot()`

A **dot plot** represents each value of a quantitative variable with a dot. The values are rounded a bit so that the dots line up neatly, and dots are stacked up into little towers when the data values cluster near each other. Dot plots are primarily used with modestly sized data sets and can be used as a bridge to the other plots, where there is no longer a direct connection between a component of the plot and an individual observation.

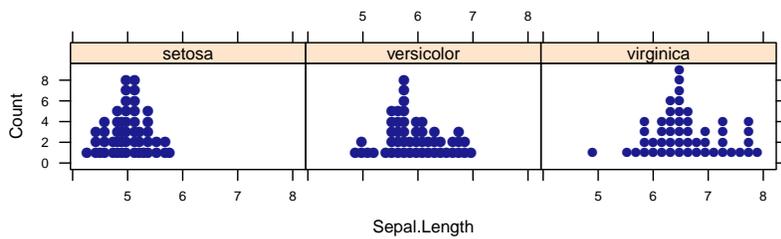
Here is an example using the sepal lengths recorded in the `iris` data set.

```
# n = 30 gives approx 30 columns
dotPlot(~Sepal.Length, data = iris, n = 30)
```



We can use a conditional variable to give us separate dot plots for each of the three species in this data set.

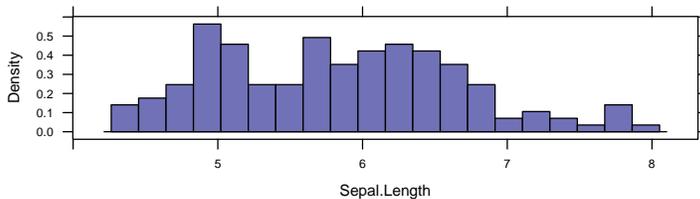
```
dotPlot(~ Sepal.Length | Species, data=iris, n=20,
        cex=.6, # cex used to shrink the dots a bit
        layout=c(3,1))
```



HISTOGRAMS: `histogram()`

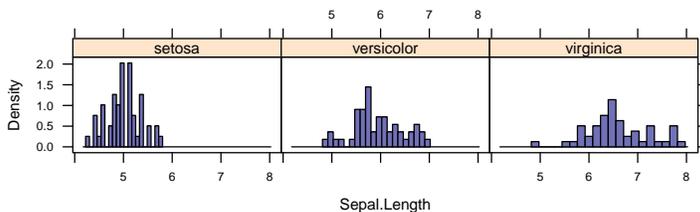
**Histograms** are a lot like dot plots, but the towers of dots are replaced by vertical bars. Again the `y` component of the formula is empty since we let R compute the heights of the bars

```
# n= 20 gives approx. 20 bars
histogram(~Sepal.Length, data = iris, n = 20)
```



We can use a conditional variable to give us separate histograms for each species.

```
histogram(~ Sepal.Length | Species, data=iris, n=20,
          layout=c(3,1)) # 3 columns (x) and 1 row (y)
```



This reveals the true story of the data – the distribution of sepal length is different for the different species of iris.

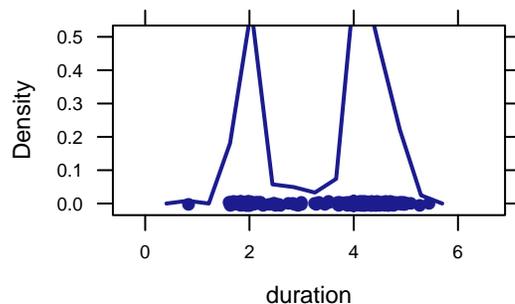
In lattice lingo, the three subplots are called **panels** and the labels at the top are called **strips**.

FREQUENCY POLYGONS: `freqpolygon()`

**Frequency polygons** and **density plots** provide alternatives to histograms that make it easier to overlay the representations of multiple subsets of the data. A frequency polygon is created from the same data summary (bins and counts) as a histogram, but instead of representing each bin with a bar, it is represented by a point (at the center of where the top of the histogram bar would have been).

These points are then connected with line segments. Here is an example that shows the distribution of Old Faithful eruptions times from a sequence of observations

```
require(MASS)
freqpolygon(~duration, data = geysers, n = 15)
```



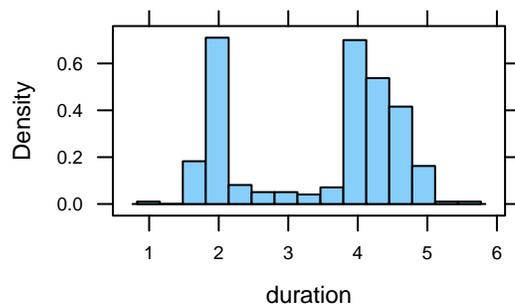
**CAUTION!**

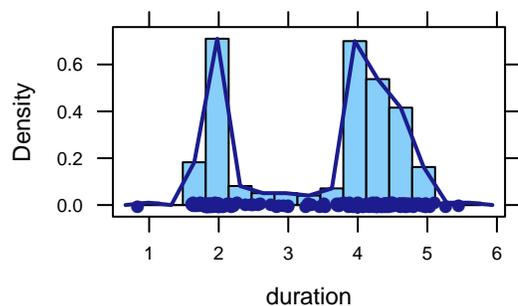
The `faithful` data set contains similar data, but the variable names in that data frame are poorly chosen. The `geyser` data set in the `MASS` package has better names and more data.

**TEACHING TIP**

Point out that an interesting feature of this distribution is its clear bimodality. In particular, the mean and median eruption time are not a good measures of the duration of a “typical” eruption since almost none of the eruption durations are near the mean and median.

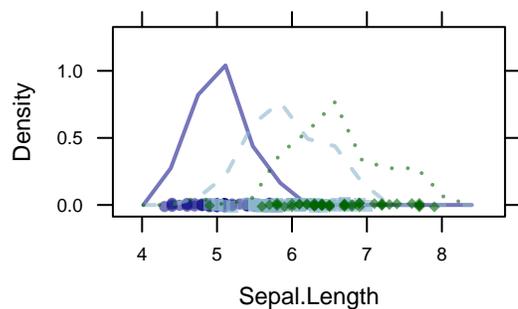
Numerically, the data are being summarized and represented in exactly the same way as for histograms, but visually the horizontal and vertical line segments of the histogram are replaced by sloped line segments.





This may give a more accurate visual representation in some situations (since the distribution can “taper off” better). More importantly, it makes it much easier to overlay multiple distributions.

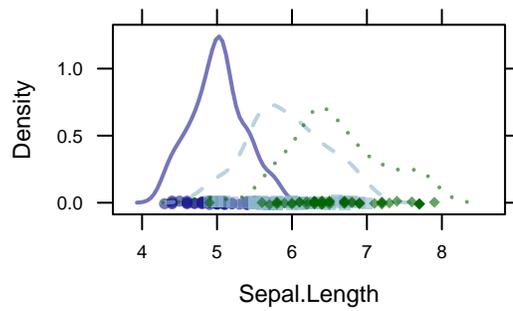
```
freqpolygon(~Sepal.Length, data = iris, groups = Species)
```



DENSITY PLOTS: `densityplot()`

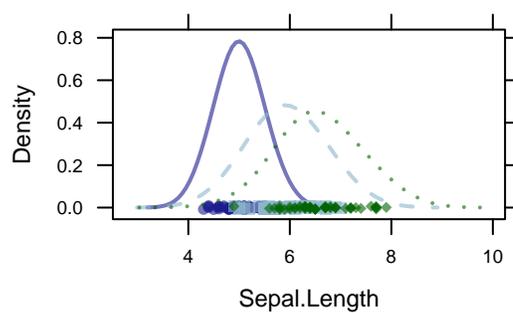
**Density plots** are similar to frequency polygons, but the piecewise linear representation is replaced by a smooth curve.

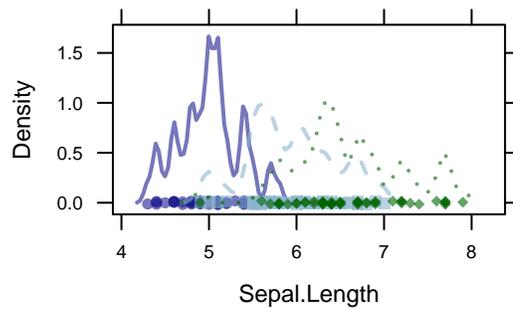
```
densityplot(~Sepal.Length, data = iris, groups = Species)
```



Beginners do not need to know the details of how that smooth curve is generated, but should be introduced to the `adjust` argument which controls the degree of smoothing. It is roughly equivalent to choosing wider or narrower bins for a histogram or frequency polygon. The default value is 1. Higher values smooth more heavily; lower values, less so.

```
densityplot( ~ Sepal.Length, data=iris, groups=Species,
             adjust=3)
densityplot( ~ Sepal.Length, data=iris, groups=Species,
             adjust=1/3)
```





### THE DENSITY SCALE

There are three scales that can be used for the plots in the preceding section: `count`, `percent`, and `density`. Beginning students will be most familiar with the `count` scale and perhaps also the `percent` scale, but most will not have seen the `density` scale. The density scale captures the most important aspect of all of these plots

Area is proportional to frequency.

The density scale is chosen so that the constant of proportionality is 1, in which case we have

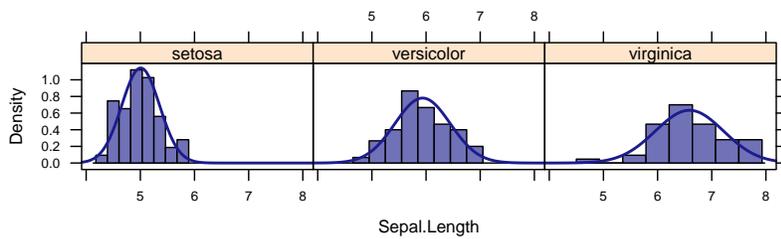
Area equals proportion.

This is the only scale available for `densityplot()` and is the most suitable scale if one is primarily interested in the *shape* of the distribution. The vertical scale is affected very little by the choice of bin widths or `adjust` multipliers. It is also the appropriate scale to use when overlaying a density function.

```
histogram( ~ Sepal.Length | Species, data=iris,
           fit="normal" )
```

### TEACHING TIP

Create some histograms or frequency polygons with a density scale and see if your students can determine what the scale is. Choosing convenient bin widths (but not 1) and comparing plots with multiple bin widths and multiple scale types can help them reach a good conjecture about the density scale.

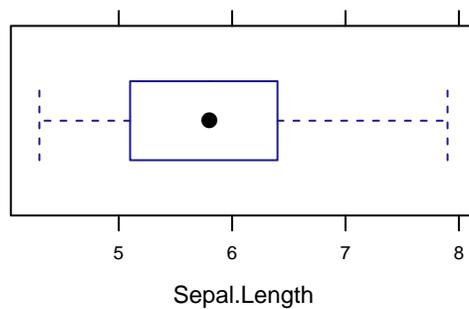


The other scales are primarily of use when one wants to be able to read off bin counts or percents from the plot.

#### 4.6.3 Boxplots: `bwplot()`

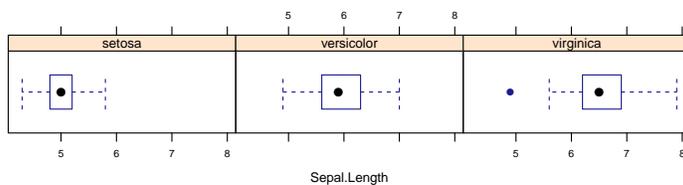
Boxplots are constructed in much the same manner as the plots in the previous section:

```
bwplot(~Sepal.Length, data = iris)
```



As we did for histograms, we can use conditioning to show boxplots for multiple groups within the data set:

```
bwplot(~Sepal.Length | Species, data = iris, layout = c(3, 1))
```

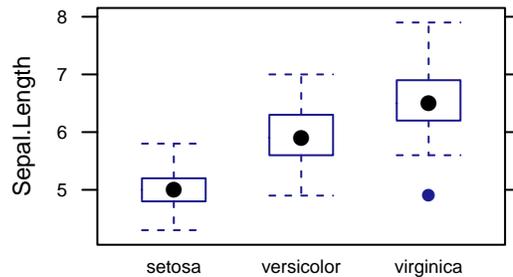


#### CAUTION!

This is not a particularly good plot for displaying this information. We'll illustrate a better way in a moment.

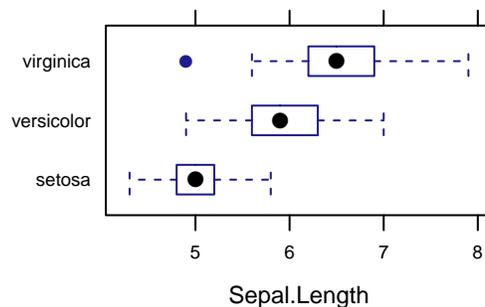
But there are better ways to do this.

```
bwplot(Sepal.Length ~ Species, data = iris)
```



The boxplots are vertical when the categorical variable is on the right hand side of the `~`.

```
bwplot(Species ~ Sepal.Length, data = iris)
```



The boxplots are horizontal when the categorical variable is on the left hand side of the `~`.

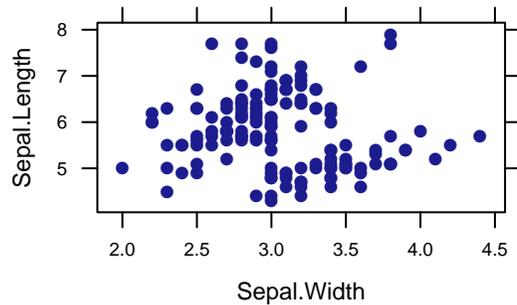
#### CAUTION!

If numbers are used to label the categories of a categorical variable, R will consider the data to be quantitative. You can convert numerical values into a categorical factor by wrapping it in the function `factor()`, e.g., `factor(x)`.

#### 4.6.4 Scatterplots: `xypplot()`

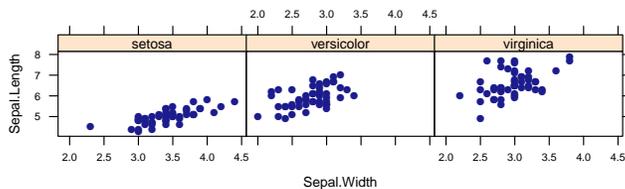
Scatterplots are made with `xypplot()`. The formula and modeling interface is used in the same manner as before. Just remember that the “y variable” comes first. (Its label is also farther left on the plot, if that helps you remember.)

```
xyplot(Sepal.Length ~ Sepal.Width, data = iris)
```



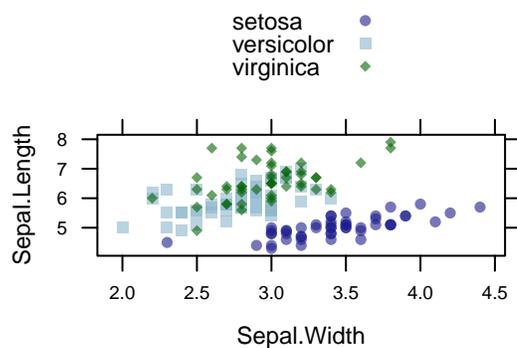
Again, we can use conditioning to make a panel for each species.

```
xyplot(Sepal.Length ~ Sepal.Width | Species, data=iris,
  layout=c(3,1)) # layout controls number of columns and rows
```



Even better (for this example), we can use the `groups` argument to indicate the different species using different symbols on the same panel.

```
xyplot(Sepal.Length ~ Sepal.Width, groups=Species,
  auto.key=TRUE, data=iris)
```



#### 4.6.5 Saving Your Plots

There are several ways to save plots in RStudio, but the easiest is probably the following:

1. In the Plots tab, click the “Export” button.
2. Copy the image to the clipboard using right click.
3. Go to your document (e.g. Microsoft Word) and paste in the image.
4. Resize or reposition your image as needed.

The `pdf()` function can be used to save plots as pdf files. See the documentation of this function for details and links to functions that can be used to save graphics in other file formats.

You can save all of this exporting and copying and pasting if you use RMarkdown, or `knitr`/`LaTeX` to prepare your documents.

#### 4.7 Numerical Summaries

Numerical summaries are computed using a syntax that matches the `lattice` graphics syntax. Most of the numerical summaries already familiar to you have obvious names. Here are a few examples.

```
mean(~births, data = Births78)
[1] 9132

median(~births, data = Births78)
[1] 9218

sd(~births, data = Births78)
[1] 817.9

max(~births, data = Births78)
[1] 10711

min(~births, data = Births78)
[1] 7135
```

In the current version of the `mosaic` package, the `~` can be omitted in the one-variable numerical summary functions. This is not true for the `lattice` plots, so we recommend that you use the `~` and be consistent across multiple functions.

The `favstats()` function in the `mosaic` package computes several numerical summaries all at once.

```
favstats(~births, data = Births78)

  min   Q1 median   Q3   max mean   sd   n missing
7135 8554   9218 9705 10711 9132 817.9 365      0
```

The formula interface for these functions provided by the `mosaic` package

- Removes the need for the `$` operator.
- Simplifies computing summary statistics separately in each of several groups. For example,

```
# these are equivalent
mean(length ~ sex, data = KidsFeet)

  B     G
25.11 24.32

mean(~length | sex, data = KidsFeet)

  B     G
25.11 24.32
```

- Makes computing numerical summaries syntactically identical to creating graphical summaries with `lattice` or fitting linear models.

#### 4.7.1 Tabulating Categorical Data

The Current Population Survey (CPS) is used to supplement census information between census years. The `CPS85` data set consists of a random sample of persons from the CPS, with information on wages and other characteristics of the workers, including sex, number of years of education, years of work experience, occupational status, region of residence and union membership.

```
head(CPS85, 3)

  wage educ race sex hispanic south married exper union age sector
1  9.0   10  W  M      NH     NS Married   27  Not  43  const
2  5.5   12  W  M      NH     NS Married   20  Not  38  sales
3  3.8   12  W  F      NH     NS Single    4  Not  22  sales
```

Categorical variables are often summarized in a table. R can make a table for a categorical variable using `tally()`.

```
tally(~race, CPS85)
```

```
NW  W
67 467
```

```
tally(~sector, CPS85)
```

```
clerical  const  manag  manuf  other  prof  sales  service
      97      20      55      68      68     105      38      83
```

If we prefer to have the results presented as proportions or percents, we just have to ask.

```
tally(~sector, CPS85, format = "perc")
```

```
clerical  const  manag  manuf  other  prof  sales  service
 18.165   3.745  10.300  12.734  12.734  19.663  7.116  15.543
```

```
tally(~sector, CPS85, format = "prop")
```

```
clerical  const  manag  manuf  other  prof  sales  service
0.18165  0.03745  0.10300  0.12734  0.12734  0.19663  0.07116  0.15543
```

We can make a **cross-table** (also called a **contingency table** or a **two-way table**) summarizing this data with `tally()`. This often provides a useful display of the relationship between two categorical variables.

```
tally(~race + sector, CPS85) # 2-way cross-table
```

```
      sector
race  clerical  const  manag  manuf  other  prof  sales  service
NW      15      3      6      11      5      7      3      17
W       82     17     49     57     63     98     35     66
```

Alternatively, we can select one of our categorical variables to be a conditional variable. The report is then done using proportions by default.

```
tally(~race | sector, CPS85) # conditional on sector
```

sector	race	clerical	const	manag	manuf	other	prof	sales	service
NW		0.15464	0.15000	0.10909	0.16176	0.07353	0.06667	0.07895	0.20482
W		0.84536	0.85000	0.89091	0.83824	0.92647	0.93333	0.92105	0.79518

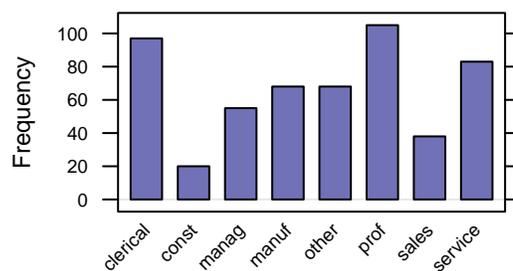
```
tally(~sector | race, CPS85) # conditional on race
```

sector	race	NW	W
clerical		0.22388	0.17559
const		0.04478	0.03640
manag		0.08955	0.10493
manuf		0.16418	0.12206
other		0.07463	0.13490
prof		0.10448	0.20985
sales		0.04478	0.07495
service		0.25373	0.14133

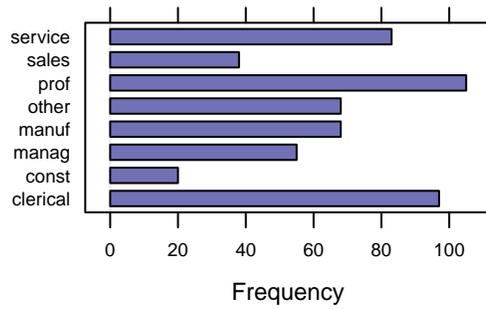
#### 4.7.2 Graphing Categorical Data

The `mosaic` function `bargraph()` can display these tables as bar graphs.

```
bargraph(~ sector, data=CPS85,
         scales=list(x=list(rot=45)))
```

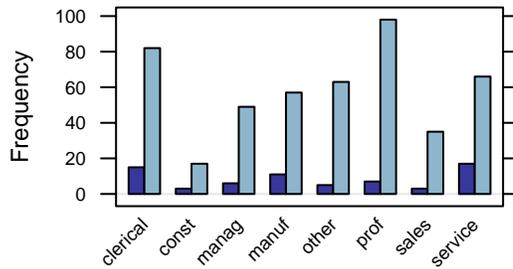


```
# horizontal bars
bargraph(~sector, data = CPS85, horizontal = TRUE)
```

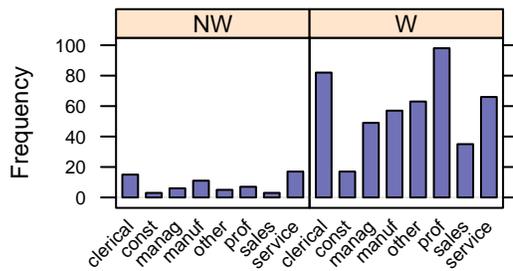


As with the other lattice plots, we can add grouping or conditioning to our plot.

```
bargraph(~ sector, data=CPS85, groups=race,
          scales=list(x=list(rot=45)))
```

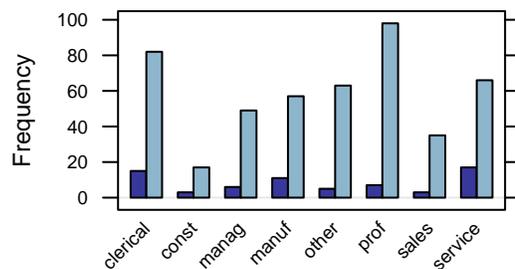


```
bargraph(~ sector | race, data=CPS85,
          scales=list(x=list(rot=45)))
```



Bar graphs can also be used to display two-way tables.

```
bargraph( ~ sector, groups=race, data=CPS85,
          scales=list(x=list(rot=45)))
```



### 4.7.3 Working with Pretabulated Data

Because categorical data is so easy to summarize in a table, often the frequency or contingency tables are given instead. You can enter these tables manually using a combination of `c()`, `rbind()` and `cbind()`:

```
myrace <- c(NW = 67, W = 467) # c for combine or concatenate
myrace
  NW  W
67 467
```

```
mycrosstable <- rbind(
  NW = c(clerical=15, const=3, manag=6, manuf=11,
         other=5, prof=7, sales=3, service=17),
  W = c(82,17,49,57,63,98,35,66)
)
mycrosstable
  clerical const manag manuf other prof sales service
NW      15     3     6    11     5     7     3     17
W       82    17    49    57    63    98    35    66
```

#### TEACHING TIP

This is an important technique if you use a text book that presents categorical data in tables.

Replacing `rbind()` with `cbind()` will allow you to give the data column-wise instead.

This arrangement of the data would be sufficient for applying the Chi-squared test, but it is not in a format suitable for plotting with `lattice`. Our cross table is still missing a bit of information – the names of the variables being stored. We can add this information if we convert it to a table

#### TEACHING TIP

If plotting pre-tabulated categorical data is important, you probably want to provide your students with a wrapper function to simplify all this. We generally avoid this situation by providing the data in raw format or by presenting an analysis of the data in tables without using graphical summaries.

```
class(mycrosstable)

[1] "matrix"

mycrosstable <- as.table(mycrosstable)
# mycrosstable now has dimnames, but they are unnamed
dimnames(mycrosstable)

[[1]]
[1] "NW" "W"

[[2]]
[1] "clerical" "const" "manag" "manuf" "other"
[6] "prof" "sales" "service"

# let's add meaningful dimnames
names(dimnames(mycrosstable)) <- c("race", "sector")
mycrosstable

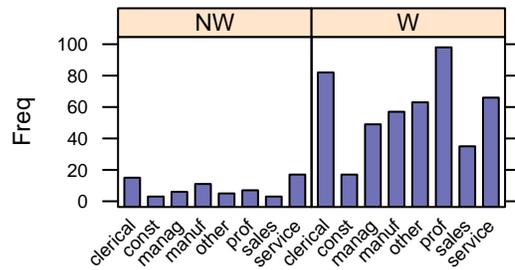
      sector
race clerical const manag manuf other prof sales service
NW      15     3     6    11     5     7     3     17
W       82    17    49    57    63    98    35    66
```

We can use `barchart()` instead of `bargraph()` to plot data already tabulated in this way, but first we need yet one more transformation.

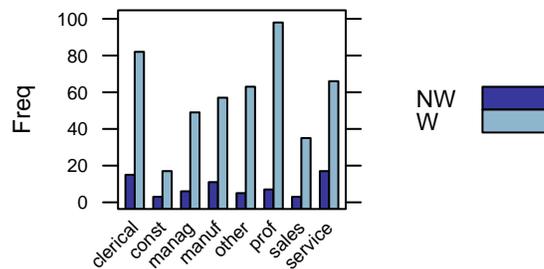
```
head(as.data.frame(mycrosstable))

  race  sector Freq
1  NW clerical  15
2   W clerical  82
3  NW   const   3
4   W   const  17
5  NW  manag   6
6   W  manag  49
```

```
barchart( Freq ~ sector | race,
  data=as.data.frame(mycrosstable),
  auto.key=list(space='right'),
  scales=list(x=list(rot=45))
  )
```



```
barchart( Freq ~ sector, groups=race,
  data=as.data.frame(mycrosstable),
  auto.key=list(space='right'),
  scales=list(x=list(rot=45))
  )
```



## 4.8 Using Your Own Data

Eventually, students will want to move from using example data sets in R packages to using data they find or collect themselves. When

### TEACHING TIP

Start out using data from packages and focusing on what R can do with the data. Later, once students are familiar with R and understand the format required for data, teach students how to import their own data.

this happens will depend on the type of students you have and the type of course you are teaching.

R provides the functions `read.csv()` (for comma separated values files), `read.table()` (for white space delimited files) and `load()` (for loading data in R's native format). The `mosaic` package includes a function called `read.file()` that uses slightly different default settings and infers whether it should use `read.csv()`, `read.table()`, or `load()` based on the file name.

Since most software packages can export to csv format, this has become a sort of *lingua franca* for moving data between packages. Data in excel, for example, can be exported as a csv file for subsequent reading in R. If you have python installed on your system, you can also use `read.xls()` from the `gdata` package to read read directly from Excel files without this extra step.

Each of these functions accepts a URL as well as a file name, which provides an easy way to distribute data via the Internet:

```
births <-
  read.table('http://www.calvin.edu/~rpruim/data/births.txt',
            header=TRUE)
head(births) # live births in the US each day of 1978.
```

	date	births	datenum	dayofyear
1	1/1/78	7701	6575	1
2	1/2/78	7527	6576	2
3	1/3/78	8825	6577	3
4	1/4/78	8859	6578	4
5	1/5/78	9043	6579	5
6	1/6/78	9208	6580	6

We can omit the `header=TRUE` if we use `read.file()`

```
births <-
  read.file('http://www.calvin.edu/~rpruim/data/births.txt')
```

#### 4.8.1 Importing Data in RStudio

The RStudio interface provides some GUI tools for loading data. If you are using the RStudio server, you will first need to upload the data to the server (in the Files tab), and then import the data into your R session (in the Workspace tab). If you are running the desktop version, the upload step is not needed.

**CAUTION!**  
There is a conflict between the `resample()` functions in `gdata` and `mosaic`. If you want to use `mosaic`'s `resample()`, be sure to load `mosaic` after you load `gdata`.

Even if you use RStudio GUI for interactive work, you will want to know how to use functions like `read.csv()` for working in RMarkdown, or `knitr`/ $\LaTeX$  files.

**TEACHING TIP**  
Remind students that the 2-step process (upload, then import) works much like images in Facebook. First you upload them to Facebook, and once they are there you can include them in posts, etc.

#### 4.8.2 *Developing Good Data Habits*

However you teach students to collect and import their data, students will need to be trained to follow good data organization practices:

- Choose good variables names.
- Put variables names in the first row.
- Use each subsequent row for one observational unit.
- Give the resulting data frame a good name.

Scientists may be disappointed that R data frames don't keep track of additional information, like the units in which the observations are recorded. This sort of information should be recorded, along with a description of the protocols used to collect the data, observations made during the data recording process, etc. This information should be maintained in a lab notebook or a **codebook**.

## 4.9 Review of R Commands

Here is a brief summary of the commands introduced in this chapter.

```

require(mosaic)           # load the mosaic package
answer <- 42              # store the number 42 in a variable named answer
log(123); log10(123); sqrt(123)  # some standard numerical functions
x <- c(1,2,3)            # make a vector containing 1, 2, 3 (in that order)
data(iris)               # (re)load the iris data set
names(iris)              # see the names of the variables in the iris data
summary(iris)            # summarize each variables in the iris data set
str(iris)                # show the structure of the iris data set
head(iris)               # first few rows of the iris data set

mydata <- read.table("file.txt")  # read data from a text file
mydata <- read.csv("file.csv")    # read data from a csv file
mydata <- read.file("file.txt")   # read data from a text or csv file

tally( ~ sector, data=CPS85 )     # frequency table
tally( ~ sector + race, data=CPS85 ) # cross tabulation of sector by race
mean( ~ age, data = HELPrct )     # mean age of HELPrct subjects
mean( ~ age | sex, data = HELPrct ) # mean age of male and female HELPrct subjects
mean( age ~ sex, data = HELPrct ) # mean age of male and female HELPrct subjects
median(x); var(x); sd(x);    # more numerical summaries
quantile(x); sum(x); cumsum(x) # still more summaries
favstats( ~ Sepal.Length, data=iris ) # compute favorite numerical summaries

dotPlot( ~ Sepal.Length | Species, data=iris ) # dot plots for each species
histogram( ~ Sepal.Length | Species, data=iris ) # histograms (with extra features)
densityplot( ~ Sepal.Length, groups = Species, data=iris ) # overlaid densityplots
freqpolygon( ~ Sepal.Length, groups = Species, data=iris ) # overlaid frequency polygons
bwplot( Sepal.Length ~ Species, data = iris ) # side-by-side boxplots
xyplot( Sepal.Length ~ Sepal.Width | Species, data=iris ) # scatter plots for each species
bargraph( ~ sector, data=CPS85 ) # bar graph
qqmath( ~ age | sex, data=CPS85 ) # quantile-quantile plots

```

## 4.10 Exercises

**4.1** What is the average (mean) *width* of the sepals in the `iris` data set?

**4.2** Determine the average (mean) sepal width for each of the three species in the `iris` data set.

**4.3** The `Utilities2` data set in the `mosaic` package contains information about the bills for various utilities at a residence in Minnesota collected over a number of years. Since the number of days in a billing cycle varies from month to month, variables like `gasbillpday` (`elecbillpday`, etc.) contain the gas bill (electric bill, etc.) divided by the number of days in the billing cycle.

- a) Use the documentation to determine what the `kwh` variables contains.
- b) Make a scatter plot of `gasbillpday` vs. `monthsSinceY2K` using the command

```
xypLOT(gasbillpday ~ monthsSinceY2K, data=Utilities2,
        type='l')      # the letter l
```

What pattern(s) do you see?

- c) What does `type='l'` do? Make your plot with and without it. Which is easier to read in this situation?
- d) What happens if we replace `type='l'` with `type='b'`?
- e) Make a scatter plot of `gasbillpday` by `month`. What do you notice?
- f) Make side-by-side boxplots of `gasbillpday` by `month` using the `Utilities2` data frame. What do you notice?

Your first try probably won't give you what you expect. The reason is that `month` is coded using numbers, so R treats it as numerical data. We want to treat it as categorical data. To do this in R use `factor(month)` in place of `month`. R calls categorical data a **factor**.

- g) Make any other plot you like using this data. Include both a copy of your plot and a discussion of what you can learn from it.

4.4 The table below is from a study of nighttime lighting in infancy and eyesight (later in life).

	no myopia	myopia	high myopia
darkness	155	15	2
nightlight	153	72	7
full light	34	36	3

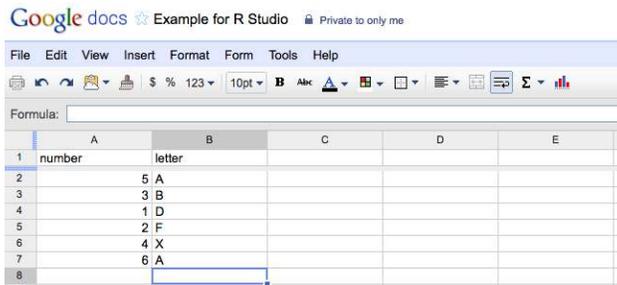
a) Recreate the table in R.

b) What percent of the subjects slept with a nightlight as infants?

There are several ways to do this. You could use R as a calculator to do the arithmetic. You can save some typing if you use the function `tally()`. See `?tally` for documentation.

c) Create a graphical representation of the data. What does this plot reveal?

4.5 Enter the following small data set in an Excel or Google spreadsheet and import the data into RStudio.



The screenshot shows a Google Docs spreadsheet with the following data:

	A	B	C	D	E
1	number	letter			
2		5 A			
3		3 B			
4		1 D			
5		2 F			
6		4 X			
7		6 A			
8					

# 5

## *What Instructors Need to Know about R*

We recommend keeping the amount of R that students need to learn to a minimum, and choosing functions that support a formula interface whenever possible to keep the required functions syntactically similar. But there are some additional things that instructors should know about R. We outline some of these things in this chapter.

You may find that some of these things are useful for your students to know as well. That will depend on the goals for your course and the abilities of your students. In higher level courses, much of the material in this chapter is also appropriate for students.

### *5.1 Some Workflow Suggestions*

Our workflow advice can be summarized in one short sentence:

*Think like a programmer.*

It doesn't take sophisticated programming skills to be good at using R. In fact, most uses of R for teaching statistics can be done working one step at a time, where each line of code does one complete and useful task. After inspecting the output (and perhaps saving it for further computation later), one can proceed to the next operation.

Nevertheless, we can borrow from the collective wisdom of the programming community and adopt some practices that will make our experience more pleasurable, more efficient, and less error-prone.

- Store your code in a file.

It can be tempting to do everything in the console. But the console is ephemeral. It is better to get into the habit of storing code in files. Get in the habit (and get your students in the habit) of working with R scripts, RMarkdown files, and Rnw files.

You can execute all the code in an R script file using

```
source("file.R")
```

RStudio has additional options for executing some or all lines in a file. See the buttons in the panel for any R script, RMarkdown or Rnw file. (You can create a new file in the main File menu.)

R can be used to create executable scripts. Option parsing and handling is supported with the `optparse` package.

If you work at the console’s interactive prompt and later wish you had been putting your commands into a file, you can save your past commands with

```
savehistory("someRCommandsIalmostLost.R")
```

In RStudio, you can selectively copy portions of your history to a script file (or the console) using the History tab.

- Use meaningful names.
  - Rarely should objects be named with a single letter.
  - Adopt a personal convention regarding case of letters. This will mean you have one less thing to remember when trying to recall the name of an object. For example, in the `mosaic` package, all data frames begin with a capital letter. Most variables begin with a lower case letter (a few exceptions are made for some variables with names that are well-known in their capitalized form).
- Adopt reusable idioms.
  - Computer programmers refer to the little patterns that recur throughout their code as idioms. For example, here is a “compute, save, display” idiom.

```
# compute, save, display idiom
footModel <- lm( length ~ width, data=KidsFeet ); footModel

Call:
lm(formula = length ~ width, data = KidsFeet)

Coefficients:
(Intercept)      width
      9.82         1.66
```

Often there are multiple ways to do the same thing in R, but if you adopt good programming idioms, it will be clearer to both you and your students what you are doing.

- Write reusable functions.
  - Learning to write your own functions (see Section ??) will greatly increase your efficiency and also help you understand better how R works. This, in turn, will help you debug your students error messages. (More on error messages in ??.) It also makes it possible for you to simplify tasks you want your students to be able to do in R. That is how the `mosaic` package originated – as a collection of tools we had assembled over time to make teaching and learning easier.

- Comment your code.

It's amazing what you can forget. The comment character in R is `#`. If you are working in RMarkdown or Rnw files, you can also include nicely formatted text to describe what you are doing and why.

## 5.2 Primary R Data Structures

### 5.2.1 Modes and other attributes

In R, data is stored in objects. Each **object** has a *name*, *contents*, and also various *attributes*. Attributes are used to tell R something about the kind of data stored in an object and to store other auxiliary information. Two important attributes shared by all objects are **mode** and **length**.

```
w <- 2.5
x <- c(1, 2)
y <- "foo"
z <- TRUE
abc <- letters[1:3]
```

```
mode(w)
[1] "numeric"

length(w)
[1] 1

mode(x)
[1] "numeric"

length(x)
[1] 2

mode(y)
[1] "character"

length(y)
[1] 1
```

```

y[1]

[1] "foo"

y[2] # not an error to ask for y[2]

[1] NA

mode(z)

[1] "logical"

length(z)

[1] 1

abc

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

mode(abc)

[1] "character"

length(abc)

[1] 3

abc[3]

[1] "c"

```

Each of the objects in the example above is a **vector**, an ordered container of values that all have the same mode.<sup>1</sup> The `c()` function concatenates vectors (or lists). Notice that `w`, `y`, and `z` are vectors of length 1. Missing values are coded as `NA` (not available). Asking for an entry “off the end” of a vector returns `NA`. Assigning a value “off the end” of a vector results in the vector being lengthened so that the new value can be stored in the appropriate location.

```

q <- 1:5
q

[1] 1 2 3 4 5

q[10] <- 10
q

[1] 1 2 3 4 5 NA NA NA NA 10

```

<sup>1</sup> There are other modes in addition to the ones shown here, including `complex` (for complex numbers), `function`, `list`, `call`, and `expression`.

There are important ways that R has been optimized to work with vectors since they correspond to variables (in the sense of statistics). For categorical data, a **factor** is a special type of vector that includes an additional attribute called *levels*. A factor can be ordered or un-ordered (which can affect how statistics are done and graphs are made) and its elements can have mode **numeric** or **character**.

A **list** is similar to a vector, but its elements may be of different modes (including **list**, **vector**, etc.). A **data frame** is a list of vectors (or factors), each of the same length, but not necessarily of the same mode. This is R's primary way of storing data sets. An **array** is a multi-dimensional table of values that all have the same mode. A **matrix** is a 2-dimensional array.

The access operators `[ ]` for vectors, matrices, arrays, and data frames, and `[[ ]]` for lists – are actually *functions* in R. This has some important consequences:

- Accessing elements is slower than in a language like C/C++ where access is done by pointer arithmetic.
- These functions also have named arguments, so you can see code like the following

```
xm <- matrix(1:16, nrow = 4)
xm
      [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

xm[5]

[1] 5

xm[, 2] # this is 1-d (a vector)

[1] 5 6 7 8

xm[, 2, drop = FALSE] # this is 2-d (still a matrix)

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

Many objects have a **dim attribute** that stores the dimension of the object. You can change it to change the shape (or even the number of dimensions) of a vector, matrix, or array. You can see all of the non-intrinsic attributes (mode and length are intrinsic) using `attributes()`, and you can set attributes (including new ones you make up) using `attr()`. Some attributes, like dimension, have special functions for accessing or setting. The `dim()` function returns the dimensions of an object as a vector. The number of rows and columns can be obtained using `nrow()` and `ncol()`.

```
ddd <- data.frame(number = 1:5, letter = letters[1:5])
attributes(ddd)

$names
[1] "number" "letter"

$row.names
[1] 1 2 3 4 5

$class
[1] "data.frame"
```

```
dim(ddd)
```

```
[1] 5 2
```

```
nrow(ddd)
```

```
[1] 5
```

```
ncol(ddd)
```

```
[1] 2
```

```
names(ddd)
```

```
[1] "number" "letter"
```

```
row.names(ddd)
```

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

### 5.2.2 What is it?

R provides a number of functions for testing the mode or class of an object.

```
mode(xm); class(xm)

[1] "matrix"

c(numeric=is.numeric(xm), char=is.character(xm),
  int=is.integer(xm), logical=is.logical(xm))

numeric   char   int logical
   TRUE   FALSE   TRUE  FALSE

c(vector=is.vector(xm), matrix=is.matrix(xm), array=is.array(xm))

vector matrix array
  FALSE   TRUE   TRUE
```

### 5.2.3 Changing modes (coercion)

If R is expecting an object of a certain mode or class but gets something else, it will often try to **coerce** the object to meet its expectations. You can also coerce things manually using one of the many `as.???` functions.

```
apropos("^as\\.")[1:10] # just a small sample

[1] "as.array"           "as.array.default"
[3] "as.call"            "as.character"
[5] "as.character.condition" "as.character.Date"
[7] "as.character.default" "as.character.error"
[9] "as.character.factor" "as.character.hexmode"

xm

      [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

# convert numbers to strings (this drops attributes,
# including dimension)
as.character(xm)
```

```
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11"
[12] "12" "13" "14" "15" "16"
```

```
# convert matrix to vector
as.vector(xm)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

as.logical(xm)

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

```
alpha <- c("a", "1", "b", "0.5")
mode(alpha)

[1] "character"
```

```
as.numeric(alpha) # note the NAs when coercion isn't possible
Warning: NAs introduced by coercion

[1] NA 1.0 NA 0.5

as.integer(alpha) # notice coercion of 0.5 to 0
Warning: NAs introduced by coercion

[1] NA 1 NA 0
```

### 5.3 More About Vectors

Vectors are so important in R that they deserve some additional discussion. In Section 5.4.5 we learned how to generate some simple vectors. Here we will learn about some of the operations and functions that can be applied to vectors.

#### 5.3.1 Names and vectors

We can give each position in a vector a name. This can be very handy for certain uses of vectors.

```
myvec <- 1:5
myvec

[1] 1 2 3 4 5

names(myvec) <- c("one", "two", "three", "four", "five")
myvec

  one  two three  four  five
1    2    3    4    5
```

Names can also be specified as a vector is created using `c()`.

```
another <- c(mean = 10, sd = 2, `trimmed mean` = 9.7)
another

      mean      sd trimmed mean
1  10.0    2.0      9.7
```

### 5.3.2 Vectorized functions

Many R functions and operations are “vectorized” and can be applied not just to an individual value but to an entire vector, in which case they are applied componentwise and return a vector of transformed values. Most traditional mathematics functions are available and work this way.

```
x <- 1:5; y <- seq(10, 60, by=10);
x

[1] 1 2 3 4 5

y

[1] 10 20 30 40 50 60

y + 1

[1] 11 21 31 41 51 61

x * 10

[1] 10 20 30 40 50

x < 3
```

The `seq()` function creates an arithmetic sequence of numbers from a specified starting point, upper bound, and increment.

```
[1] TRUE TRUE FALSE FALSE FALSE

x^2

[1] 1 4 9 16 25

log(x)           # natural log

[1] 0.0000 0.6931 1.0986 1.3863 1.6094

log(x, base=10) # base 10 log

[1] 0.0000 0.3010 0.4771 0.6021 0.6990

log10(x)        # base 10 log again

[1] 0.0000 0.3010 0.4771 0.6021 0.6990
```

Vectors can be combined into a matrix using `rbind()` or `cbind()`. This can facilitate side-by-side comparisons.

```
# compare round() and signif() by binding rowwise into matrix
z <- rnorm(5); z

[1] -0.56048 -0.23018 1.55871 0.07051 0.12929

rbind(round(z, digits=3), signif(z, digits=3))

      [,1] [,2] [,3] [,4] [,5]
[1,] -0.56 -0.23 1.559 0.0710 0.129
[2,] -0.56 -0.23 1.560 0.0705 0.129
```

### 5.3.3 Functions that act on vectors as vectors

Other functions, including many statistical functions, are designed to work on the vector as a vector. Often these return a single value (technically a vector of length 1), but other return types are used as appropriate.

```
x <- 1:10
z <- rnorm(100)
mean(z)

[1] 0.06073

sd(z)
```

```

[1] 0.9089

var(z)

[1] 0.826

median(z) # basic statistical functions

[1] -0.01139

range(z) # range returns a vector of length 2

[1] -2.309 2.187

sum(x)

[1] 55

prod(x) # sums and products

[1] 3628800

```

```

z <- rnorm(5)
z

[1] -0.04503 -0.78490 -1.66794 -0.38023 0.91900

sort(z)

[1] -1.66794 -0.78490 -0.38023 -0.04503 0.91900

rank(z)

[1] 4 2 1 3 5

order(z)

[1] 3 2 4 1 5

rev(x) # reverse x

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

```

The following functions produce vectors, but not by applying a function component by component.

```
x
[1] 1 2 3 4 5 6 7 8 9 10
diff(x) # pairwise differences
[1] 1 1 1 1 1 1 1 1 1
cumsum(x) # cumulative sum
[1] 1 3 6 10 15 21 28 36 45 55
cumprod(x) # cumulative product
[1]      1      2      6     24    120    720   5040
[8]  40320 362880 3628800
```

Whether a function is vectorized or treats a vector as a unit depends on its implementation. Usually, things are implemented the way you would expect. Occasionally you may discover a function that you wish were vectorized and is not. When writing your own functions, give some thought to whether they should be vectorized, and test them with vectors of length greater than 1 to make sure you get the intended behavior.

Some additional useful functions are included in Table 5.2.

#### 5.3.4 Recycling

When vectors operate on each other, the operation is done componentwise, recycling the shorter vector to match the length of the longer.

```
x <- 1:5
y <- seq(10, 70, by = 10)
x + y

Warning: longer object length is not a multiple of shorter
object length

[1] 11 22 33 44 55 61 72
```

In fact, this is exactly how things like  $x + 1$  actually work. If  $x$  is a vector of length  $n$ , then 1 (a vector of length 1) is first recycled into a vector of length  $n$ ; then the two vectors are added componentwise. Some vectorized functions that take multiple vectors as arguments will first use recycling to make them the same length.

cumsum() cumprod() cummin() cummax()	Returns vector of cumulative sums, products, minima, or maxima.
pmin(x,y,...) pmax(x,y,...)	Returns vector of parallel minima or maxima where <i>i</i> th element is max or min of x[i], y[i], ...
which(x)	Returns a vector of indices of elements of x that are true. Typical use: which(y > 5) returns the indices where elements of y are larger than 5.
any(x)	Returns a logical indicating whether any elements of x are true. Typical use: if ( any(y > 5) ) { ...}.
na.omit(x)	Returns a vector with missing values removed.
unique(x)	Returns a vector with repeated values removed.
table(x)	Returns a table of counts of the number of occurrences of each value in x. The table is similar to a vector with names indicating the values, but it is not a vector.
paste(x,y,..., sep=" ")	Pastes x and y together componentwise (as strings) with sep between elements. Recycling applies.

Table 5.2: Some useful R functions.

### 5.3.5 Accessing elements of vectors

R allows for some very interesting and useful methods for accessing elements of a vector that combine the ideas above. First, recall that the `[ ]` operator is actually a function. Furthermore, it is vectorized.

```
x <- seq(2, 20, by = 2)
x[1:5]

[1] 2 4 6 8 10

x[c(1, 4, 7)]

[1] 2 8 14
```

`[ ]` accepts logical (i.e. boolean) arguments well. The boolean values (recycled, if necessary) are used to select or deselect elements of the vector.

```
x <- seq(2, 20, by = 2)
x[c(TRUE, TRUE, FALSE)] # skips every third (note recycling)

[1] 2 4 8 10 14 16 20
```

```
x[x > 10] # more typical use of boolean in selection
[1] 12 14 16 18 20
```

Negative indices are used to omit elements.

```
x[-c(2, 4)] # all but 2nd and 4th
[1] 2 6 10 12 14 16 18 20
```

Here are some more examples.

```
notes <- toupper(letters[1:7])
a <- 1:5
b <- seq(10, 100, by = 10)
toupper(letters[5:10])

[1] "E" "F" "G" "H" "I" "J"

paste(letters[1:5], 1:3, sep = "-")

[1] "a-1" "b-2" "c-3" "d-1" "e-2"

a + b

[1] 11 22 33 44 55 61 72 83 94 105

(a + b)[a + b > 50]

[1] 55 61 72 83 94 105

length((a + b)[a + b > 50])

[1] 6

table(a + b > 50)

FALSE TRUE
 4     6
```

## 5.4 Working with Data

In Section 4.5 we discussed using data in R packages, and in Section 4.8 we discussed methods for bringing your own data into R.

In both of these scenarios, we have assumed that the data had been entered and cleaned in some other software and focussed primarily on data import. In this section we discuss ways to create and manipulate data within R. But first discuss a few more details regarding importing data.

#### 5.4.1 *Finer control over data import*

The `na.strings` argument can be used to specify codes for missing values. The following can be useful, for example:

```
someData <- read.file('file.csv',
  na.strings=c('NA', '', '.', '-', 'na'))
```

because SAS uses a period (.) to code missing data, and some csv exporters use '-'. By default R reads these as string data, which forces the entire variable to be of character type instead of numeric.

By default, R will recode character data as a factor. If you prefer to leave such variables in character format, you can use

```
somData <- read.file('file.csv',
  na.strings=c('NA', '', '.', '-', 'na'),
  stringsAsFactors=FALSE)
```

Even finer control can be obtained by manually setting the class (type) used for each column in the file. In addition, this speeds up the reading of the file. For a csv file with four columns, we can declare them to be of class integer, numeric, character, and factor with the following command.

```
someData <- read.file('file.csv',
  na.strings=c('NA', '', '.', '-', 'na'),
  colClasses=c('integer', 'numeric', 'character', 'factor'))
```

#### 5.4.2 *Manually typing in data*

The `c()` function combines elements into a single list or vector.

```
x <- c(1,1,2,3,5,8,13); x
[1] 1 1 2 3 5 8 13
```

Even if you primarily use the RStudio interface to import data, it is good to know about the command line methods since these are required to import data into scripts, RMarkdown, and `knitr`/LaTeX files.

The `scan()` function can speed up data entry in the console by allowing you to avoid the commas. Individual values are separated by white space or new lines. A blank line is used to signal the end of the data. By default, `scan()` is expecting decimal data (which it calls **double**, for double precision), but it is possible to tell `scan()` to expect something else, like **character** data (i.e., text). There are other options for data types, but numerical and text data handle the most important cases. See `?scan` for more information and examples.

Be sure when using `scan()` that you remember to save your data somewhere. Otherwise you will have to type it again.

### 5.4.3 *Creating data frames from vectors*

The `c()` and `scan()` functions put data into a **vector**, not a **data frame**. We can build a data frame for our data as follows.

```
color <- c("red", "green", "blue")
number <- c(3, 5, 4)
myDataFrame <- data.frame(color = color, letters = number)
myDataFrame
```

	color	letters
1	red	3
2	green	5
3	blue	4

### 5.4.4 *Getting data from mySQL data bases*

The **RMySQL** package allows direct access to data in MySQL data bases. This can be convenient when dealing with subsets of very large data sets. The example below queries a the UCSC genome browser to find all the known genes on chromosome 1.

```

require(RMySQL)

Loading required package: RMySQL
Loading required package: DBI

# Set up a connection to the public MySQL server
# for the UCSC genome browser (no password)
ucscData <- dbConnect(MySQL(),
                      user="genome",
                      host="genome-mysql.cse.ucsc.edu")

# Function to make it easier to query
query <- function(...) dbGetQuery(ucscData, ...)

# Get the UCSC gene name, start and end sites for genes on Chromosome 1
chrom1 <-
  query("SELECT name, chrom, txStart, txEnd FROM mm9.knownGene WHERE chrom='chr1';")
dim(chrom1)

[1] 3056    4

head(chrom1,2)

      name chrom txStart  txEnd
1 uc007aet.1 chr1 3195984 3205713
2 uc007aeu.1 chr1 3204562 3661579

```

#### 5.4.5 Generating data

The following code shows a number of ways to generate data systematically. This can be useful for designing experiments, for creating illustrations, or for performing simulations.

```

# all integers in a range
x <- 5:20; x

[1] 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# structured sequences
seq(0, 50, by=5)

[1] 0 5 10 15 20 25 30 35 40 45 50

seq(0, 50, length=7)

[1] 0.000 8.333 16.667 25.000 33.333 41.667 50.000

```

```

rep(1:5, each=3)

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

rep(1:5, times=3)

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

# c() concatenates vectors
c(1:5, 10, 3:5)

[1] 1 2 3 4 5 10 3 4 5

```

R can also sample from several different distributions.

```

# random draws from normal distribution
rnorm(10, mean = 10, sd = 2)

[1] 8.849 11.216 6.764 9.889 11.039 10.602 10.211 8.719
[9] 8.301 7.952

x <- 5:20 # all integers in a range
# random sample of size 5 from x (no replacement)
sample(x, size = 5)

[1] 13 14 7 20 8

# a different random sample of size 5 from x (no replacement)
sample(x, size = 5)

[1] 16 10 18 17 13

# random sample of size 5 from x (with replacement)
resample(x, size = 5)

[1] 9 8 14 9 13

```

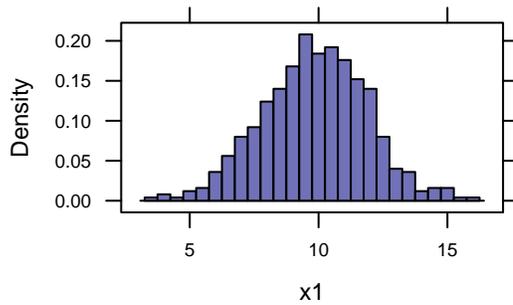
Functions for sampling from other distributions include `rbinom()`, `rchisq()`, `rt()`, `rf()`, `rhyper()`, etc.

**Example 5.1.** For teaching purposes it is sometimes nice to create a histogram that has the approximate shape of some distribution. One way to do that was illustrated above.

```

x1 <- rnorm(500, mean = 10, sd = 2)
histogram(~x1, width = 0.5)

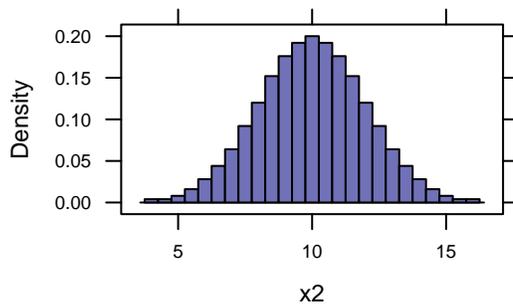
```



This works, but the resulting plot has a fair amount of noise.

The `ppoints()` function returns evenly spaced probabilities and allows us to obtain theoretical quantiles of the normal distribution instead. The resulting plot now illustrates the idealized sample from a normal distribution.

```
x2 <- qnorm(ppoints(500), mean = 10, sd = 2)
histogram(~x2, width = 0.5)
```



This is not what real data will look like (even if it comes from a normal population), but it can be better for illustrative purposes to remove the noise. ◇

#### 5.4.6 Saving Data

`write.table()` and `write.csv()` can be used to save data from R into delimited flat files.

```
ddd <- data.frame(number = 1:5, letter = letters[1:5])
write.table(ddd, "ddd.txt")
write.csv(ddd, "ddd.csv")
```

Data can also be saved in native R format. Saving data sets (and other R objects) using `save()` has some advantages over other file formats:

- Complete information about the objects is saved, including attributes.
- Data saved this way takes less space and loads much more quickly.
- Multiple objects can be saved to and loaded from a single file.

The downside is that these files are only readable in R.

```
abc <- "abc"
ddd <- data.frame(number = 1:5, letter = letters[1:5])
# save both objects in a single file
save(ddd, abc, file = "ddd.rda")
# load them both
load("ddd.rda")
```

For more on importing and exporting data, especially from other formats, see the *R Data Import/Export* manual available on CRAN.

## 5.5 *Sharing With and Among Your Students*

Instructors often have their own data sets to illustrate points of statistical interest or to make a particular connection with a class. Sometimes you may want your class as a whole to construct a data set, perhaps by filling in a survey or by contributing their own small bit of data to a class collection. Students may be working on projects in small groups; it's nice to have tools to support such work so that all members of the group have access to the data and can contribute to a written report.

There are now many technologies that support such sharing. For the sake of simplicity, we will emphasize three that we have found particularly useful both in teaching statistics and in our professional collaborative work. These are:

- Within RStudio server.
- A web site with minimal overhead, such as provided by Dropbox.
- The services of Google Docs.
- A web-based RStudio server for R.

The first two are already widely used in university environments and are readily accessible simply by setting up accounts. Setting up an RStudio web server requires some IT support, but is well within the range of skills found in IT offices and even among some individual faculty.

### 5.5.1 Using RStudio server to share files

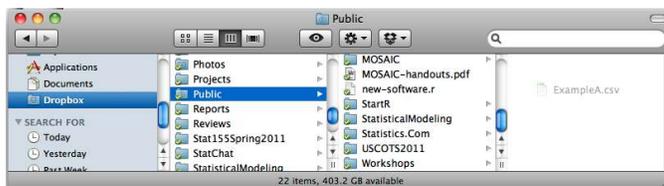
The RStudio server runs on a Linux machine. Users of RStudio have accounts on the underlying Linux file system and it is possible to set up shared directories with permissions that allow multiple users to read and/or write files stored there. This has to be done outside of RStudio, but if you are familiar with the Linux operating system or have a system administrator who is willing to help you out, this is not difficult to do.

### 5.5.2 Your own web site

You may already have a web site. We have in mind a place where you can place files and have them accessed directly from the Internet. For sharing data, it's best if this site is public, that is, it does not require a login. That rules out most "course support" systems such as Moodle or Blackboard.

The Dropbox service for storing files in the "cloud" provides a very convenient way to distribute files over the web. (Go to [dropbox.com](http://dropbox.com) for information and to sign up for a free account.) Dropbox is routinely used to provide automated backup and coordinated file access on multiple computers. But the Dropbox service also provides a **Public** directory. Any files that you place in that directory can be accessed directly by a URL.

To illustrate, suppose you wish to share some data set with your students. You've constructed this data set in a spreadsheet and stored it as a csv file, let's call it `example-A.csv`. Move this file into the **Public** directory under Dropbox — on most computers Dropbox arranges things so that its directories appear exactly like ordinary directories and you'll use the ordinary, familiar file management techniques as in Figure 5.1.



**TEACHING TIP**  
When accounts are set up on the RStudio server for a new class at Calvin, each user is given a symbolic link to a directory where the instructor can write files and students can only read files. This provides an easy way to make data, R code, or history files available to students from inside RStudio.

Our discussion of Dropbox is primarily for those who do not already know how to do this other ways.

Figure 5.1: Dragging a csv file to a Dropbox Public directory

Dropbox also makes it straightforward to construct the web-location identifying URL for any file by using mouse-based menu commands to place the URL into the clipboard, whence it can be copied to your course-support software system or any other place for distribution to students. For a csv file, reading the contents of the file into R can be done with the `read.csv()` function, by giving it the quoted URL:

```
a <- read.csv("http://dl.dropbox.com/u/5098197/USCOTS2011/ExampleA.csv")
```

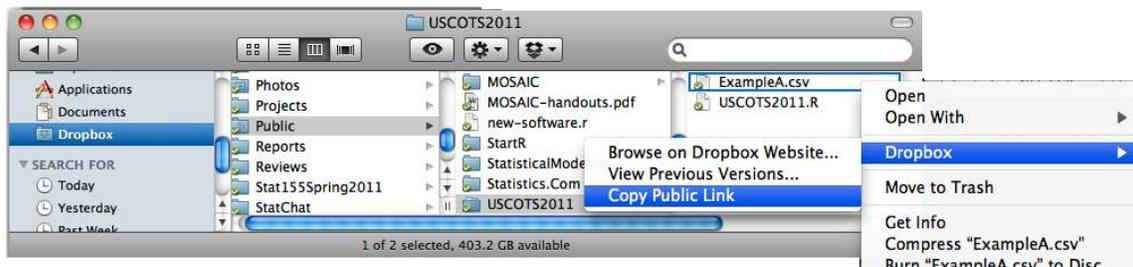


Figure 5.2: Getting the URL of a file in a Dropbox Public directory

This technique makes it easy to distribute data with little advance preparation. It's fast enough to do in the middle of a class: the csv file is available to your students (after a brief lag while Dropbox synchronizes). It can even be edited by you (but not by your students).

The same technique can be applied to all sorts of files like R workspaces or R scripts (files containing code). Of course, your students need to use the appropriate R command: `load()` for a workspace or `source()` for a script.

The example below will source a file that will print a welcoming message for you.

```
source("http://mosaic-web.org/go/R/hello.R")
```

```
Hello there. You just sourced a file over the web!
```

But you can put any R code you like in the files you have your students source. You can install and load packages, retrieve or modify data sets, define new functions, or anything else R allows.

Many instructors will find it useful to create a file with your course-specific R scripts, adding on to it and modifying it as the course progresses. This allows you to distribute all sorts of special-purpose functions, letting you distribute new R material to your students. That brilliant new idea you had at 2 am can be programmed up and put in place for your students to use the next morning in

class. Then as you identify bugs and refine the program, you can make the updated software immediately available to your students.

If privacy is a concern, for instance if you want the data available only to your students, you can effectively accomplish this by giving files names known only to your students, e.g., [Example-A78r423.csv](#).

### 5.5.3 GoogleDocs

The Dropbox technique (or any other system of posting files to the Internet) is excellent for broadcasting: taking files you create and distributing them in a read-only fashion to your students. But when you want two-way or multi-way sharing of files, other techniques are called for, such as provided by the GoogleDocs service.

GoogleDocs allows students and instructors to create various forms of documents, including reports, presentations, and spreadsheets. (In addition to creating documents *de novo*, Google will also convert existing documents in a variety of formats.)

Once on the GoogleDocs system, the documents can be edited *simultaneously* by multiple users in different locations. They can be shared with individuals or groups and published for unrestricted viewing and even editing.

For teaching, this has a variety of uses:

- Students working on group projects can all simultaneously have access to the report as it is being written and to data that is being assembled by the group.
- The entire class can be given access to a data set, both for reading and for writing.
- The Google Forms system can be used to construct surveys, the responses to which can populate a spreadsheet that can be read back into RStudio by the survey creators.
- Students can “hand in” reports and data sets by copying a link into a course support system such as Moodle or Blackboard, or emailing the link.
- The instructor can insert comments and/or corrections directly into the document.

An effective technique for organizing student work and ensuring that the instructor (and other graders) have access to it, is to create a separate Google directory for each student in your class (Dropbox can also be used in this manner). Set the permission on this directory to share it with the student. Anything she or he drops into the directory is automatically available to the instructor. The student can also share with specific other students (e.g., members of a project group).

#### CAUTION!

*Security through Obscurity* of this sort will not generally satisfy institutional data protection regulations nor professional ethical requirements, so nothing truly sensitive or confidential should be “protected” in this manner.

We will illustrate the entire process in the context of the following example.

**Example 5.2.** One exercise for students starting out in a statistics course is to collect data to find out whether the “close door” button on an elevator has any effect. This is an opportunity to introduce simple ideas of experimental design. But it’s also a chance to teach about the organization of data.

Have your students, as individuals or small groups, study a particular elevator, organize their data into a spreadsheet, and hand in their individual spreadsheet. Then review the spreadsheets in class. You will likely find that many groups did not understand clearly the distinction between cases and variables, or coded their data in ambiguous or inconsistent ways.

Work with the class to establish a consistent scheme for the variables and their coding, e.g., a variable `ButtonPress` with levels “Yes” and “No”, a variable `Time` with the time in seconds from a fiducial time (e.g. when the button was pressed or would have been pressed) with time measured in seconds, and variables `ElevatorLocation` and `GroupName`. Create a spreadsheet with these variables and a few cases filled in. Share it with the class.

Have each of your students add their own data to the class data set. Although this is a trivial task, having to translate their individual data into a common format strongly reinforces the importance of a consistent measurement and coding system for recording data.

Once you have a spreadsheet file in GoogleDocs, you will want to open it in R. This can be exported as a csv file, then open it using the csv tools in R, such as `read.csv()`.

Direct communication with GoogleDocs requires facilities that are not present in the base version of R, but are available through the `RCurl` package. In order to make these readily available to students, the `mosaic` package contains a function that takes the quoted (and cumbersome) string with the Google-published URL and reads the corresponding file into a data frame. `RCurl` needs to be installed for this to work, and will be loaded if it is not already loaded when `fetchGoogle()` is called.

```
elev <- fetchGoogle(
  "https://spreadsheets.google.com/spreadsheet/pub?
  hl=en&hl=en&key=0Am13enSal074dEVzMGJSMU5TbTc2eWlWakppQlpjcGc&
  single=TRUE&gid=0&output=csv")
```

```
head(elev)

  StudentGroup Elevator CloseButton Time Enroute
1          HA Campus Center         N 8.230      N
2          HA Campus Center         N 7.571      N
3          HA Campus Center         N 7.798      N
4          HA Campus Center         N 8.303      N
5          HA Campus Center         Y 5.811      N
6          HA Campus Center         Y 6.601      N

LagToPress
1          0
2          0
3          0
4          0
5          0
6          0
```

Of course, you'd never want your students to type that URL by hand; you should provide it in a copy-able form on a web site or within a course support system. ◇

## 5.6 Manipulating Data Frames

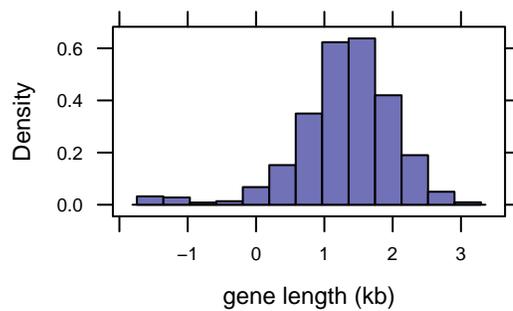
### 5.6.1 Adding new variables to a data frame

The `transform()` function can be used to add or modify variables in a data frame.

```
chrom1 <- transform(chrom1, length=(txEnd - txStart)/1000)
head(chrom1, 2)

  name chrom txStart  txEnd  length
1 uc007aet.1 chr1 3195984 3205713  9.729
2 uc007aeu.1 chr1 3204562 3661579 457.017

histogram( ~log10(length), data=chrom1,
           xlab="gene length (kb)" )
```



Here we show how to modify the `Births78` data frame so that it contains a new variable `day` that is an ordered factor. (Details about some of the functions involved will be presented later in this chapter).

```
data(Births78)
weekdays <- c("Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat")
Births <- transform( Births78,
  day = factor(weekdays[1 + (dayofyear - 1) %% 7],
    ordered=TRUE, levels = weekdays) )
head(Births,3)
```

	date	births	dayofyear	day
1	1/1/78	7701	1	Sun
2	1/2/78	7527	2	Mon
3	1/3/78	8825	3	Tue

The `CPS85` data frame contains data from a Current Population Survey (current in 1985, that is). Two of the variables in this data frame are `age` and `educ`. We can estimate the number of years a worker has been in the workforce if we assume they have been in the workforce since completing their education and that their age at graduation is 6 more than the number of years of education obtained.

```
CPS85 <- transform(CPS85, workforce.years = age - 6 - educ)
favstats(~workforce.years, data = CPS85)
```

	min	Q1	median	Q3	max	mean	sd	n	missing
	-4	8	15	26	55	17.81	12.39	534	0

In fact this is what was done for all but one of the cases to create the

```
xyplot( births ~ dayofyear, Births, groups=day, auto.key=list(space='right') )
```

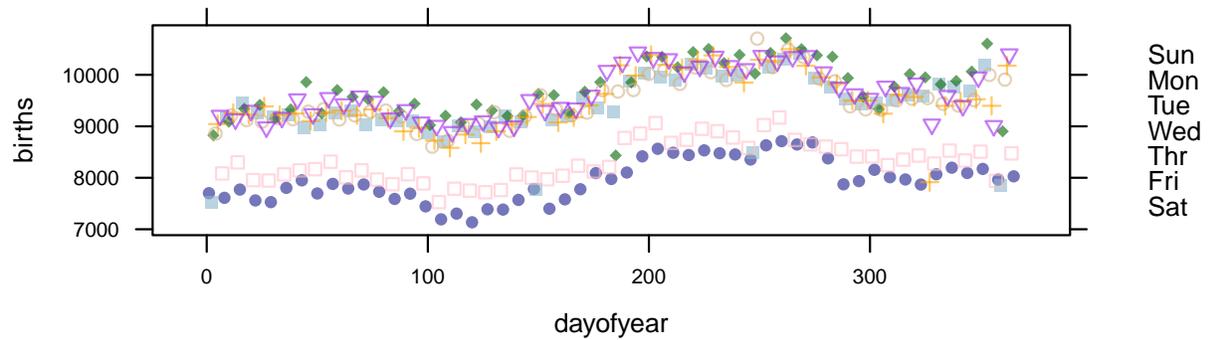


Figure 5.3: Number of US births in 1978 colored by day of week.

`exper` variable that is already in the `CPS85` data.

```
tally(~(exper - workforce.years), data = CPS85)
```

```
  0  4
533 1
```

### 5.6.2 Renaming variables

Both the column (variable) names and the row names of a data frames can be changed by simple assignment using `names()` or `row.names()`.

```
ddd # small data frame we defined earlier

  number letter
1      1      a
2      2      b
3      3      c
4      4      d
5      5      e

row.names(ddd) <- c("Abe", "Betty", "Claire", "Don", "Ethel")
ddd # row.names affects how a data.frame prints
```

	number	letter
Abe	1	a
Betty	2	b
Claire	3	c
Don	4	d
Ethel	5	e

More interestingly, it is possible to reset just individual names with the following syntax.

```
# misspelled a name, let's fix it
row.names(ddd)[2] <- "Bette"
row.names(ddd)

[1] "Abe"      "Bette"    "Claire"  "Don"     "Ethel"
```

The `faithful` data set (in the `datasets` package, which is always available) has very unfortunate names.

```
names(faithful)

[1] "eruptions" "waiting"
```

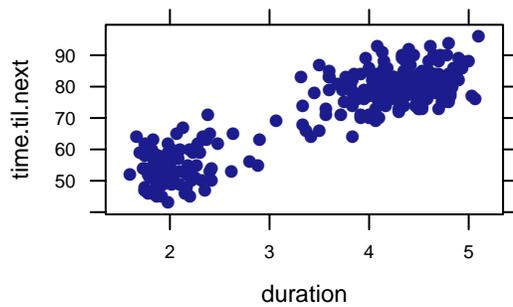
The measurements are the duration of an eruption and the time until the subsequent eruption, so let's give it some better names.

```
names(faithful) <- c("duration", "time.til.next")
head(faithful, 3)

  duration time.til.next
1   3.600           79
2   1.800           54
3   3.333           74
```

```
xyplot(time.til.next ~ duration, faithful)
```

An alternative solution is to use the `geyser` data set in the `MASS` package. The `geyser` data frame has better names and more data. But here we want to illustrate how to repair the damage in `faithful`.



We can also rename a single variable using `names()`. For example, perhaps we want to rename `educ` (the second column) to `education`.

```
names(CPS85)[2] <- "education"
CPS85[1, 1:4]

  wage education race sex
1    9         10   W   M
```

If we don't know the column number (or generally to make our code clearer), a few more keystrokes produces

```
names(CPS85)[names(CPS85) == "education"] <- "educ"
CPS85[1, 1:4]

  wage educ race sex
1    9   10   W   M
```

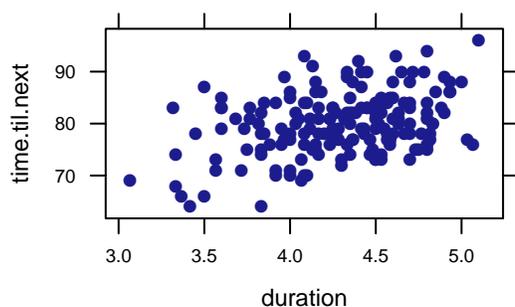
If the variable containing a data frame is modified or used to store a different object, the original data from the package can be recovered using `data()`.

See Section 5.3 for information that will make it clearer what is going on here.

### 5.6.3 Creating subsets

We can use `subset()` to select only certain rows from a data frame.

```
# any logical can be used to create subsets
faithfulLong <- subset(faithful, duration > 3)
xyplot(time.til.next ~ duration, faithfulLong)
```



Of course, if all we want to do is produce a graph, there is no reason to create a new data frame. The plot above could also be made with

```
xypLOT( time.til.next ~ duration, faithful,
        subset=duration > 3 )
```

#### 5.6.4 Dropping variables

Since we already have `educ`, there is no reason to keep our new variable `workforce.years`. Let's drop it. Notice the clever use of the minus sign.

```
CPS1 <- subset(CPS85, select = -workforce.years)
```

Any number of variables can be dropped or kept in this manner by supplying a vectors of variables names.

```
CPS1 <- subset(CPS85, select = -c(workforce.years, exper))
```

If we only want to work with the first few variables, we can discard the rest in a similar way. Columns can be specified by number as well as name (but this can be dangerous if you are wrong about where the columns are):

```
CPSsmall <- subset(CPS85, select = 1:4)
head(CPSsmall, 2)
```

	wage	educ	race	sex
1	9.0	10	W	M
2	5.5	12	W	M

### 5.6.5 Merging datasets

The `fusion1` data frame in the `fastR` package contains genotype information for a SNP (single nucleotide polymorphism) in the gene *TCF7L2*. The `pheno` data frame contains phenotypes (including type 2 diabetes case/control status) for an intersecting set of individuals. We can merge these together to explore the association between genotypes and phenotypes using `merge()`.

```
require(fastR)

Loading required package: fastR
Attaching package: 'fastR'
The following object is masked from 'package:graphics':
  panel.smooth

head(fusion1, 3)

   id   marker markerID allele1 allele2 genotype Adose
1  9735 RS12255372      1      3      3      GG      0
2 10158 RS12255372      1      3      3      GG      0
3  9380 RS12255372      1      3      4      GT      0
  Cdose Gdose Tdose
1     0     2     0
2     0     2     0
3     0     1     1

head(pheno, 3)

   id  t2d  bmi sex  age smoker chol waist weight
1 1002 case 32.86 F 70.76 former 4.57 112.0 85.6
2 1009 case 27.39 F 53.92 never 7.32 93.5 77.4
3 1012 control 30.47 M 53.86 former 5.02 104.0 94.6
  height whr sbp dbp
1 161.4 0.9868 135 77
2 168.1 0.9397 158 88
3 176.2 0.9327 143 89
```

```
# merge fusion1 and pheno keeping only id's that are in both
fusion1m <- merge(fusion1, pheno, by.x='id', by.y='id',
                 all.x=FALSE, all.y=FALSE)
head(fusion1m, 3)
```

```

  id      marker markerID allele1 allele2 genotype Adose
1 1002 RS12255372      1      3      3      GG      0
2 1009 RS12255372      1      3      3      GG      0
3 1012 RS12255372      1      3      3      GG      0
  Cdose Gdose Tdose   t2d  bmi sex  age smoker chol
1     0     2     0  case 32.86  F 70.76 former 4.57
2     0     2     0  case 27.39  F 53.92  never 7.32
3     0     2     0 control 30.47  M 53.86 former 5.02
  waist weight height   whr sbp dbp
1 112.0   85.6  161.4 0.9868 135 77
2  93.5   77.4  168.1 0.9397 158 88
3 104.0   94.6  176.2 0.9327 143 89

```

In this case, since the values are the same for each data frame, we could collapse `by.x` and `by.y` to `by` and collapse `all.x` and `all.y` to `all`. The first of these specifies which column(s) to use to identify matching cases. The second indicates whether cases in one data frame that do not appear in the other should be kept (`TRUE`) or dropped (filling in `NA` as needed) or dropped from the merged data frame.

Now we are ready to begin our analysis.

```

tally(~t2d + genotype + marker, data = fusion1m)

, , marker = RS12255372

      genotype
t2d    GG  GT  TT
case   737 375 48
control 835 309 27

```

## 5.7 Functions in R

Functions in R have several components:

- a **name** (like `histogram`)<sup>2</sup>
- an ordered list of named **arguments** that serve as inputs to the function

These are matched first by name and then by order to the values supplied by the call to the function. This is why we don't always include the argument name in our function calls. On the other hand, the availability of names means that we don't have to remember the order in which arguments are listed.

<sup>2</sup> Actually, it is possible to define functions without naming them; and for short functions that are only needed once, this can actually be useful.

Arguments often have **default values** which are used if no value is supplied in the function call.

- **a return value**

This is the output of the function. It can be assigned to a variable using the assignment operator (`=`, `<-`, or `->`).

- **side effects**

A function may do other things (like make a graph or set some preferences) that are not necessarily part of the return value.

When you read the help pages for an R function, you will see that they are organized in sections related to these components. The list of arguments appears in the **Usage** section along with any default values. Details about how the arguments are used appear in the **Arguments** section. The return value is listed in the **Value** section. Any side effects are typically mentioned in the **Details** section.

Now let's try writing our own function. Suppose you frequently wanted to compute the mean, median, and standard deviation of a distribution. You could make a function to do all three to save some typing. Let's name our function `mystats()`. The `mystats()` will have one argument, which we are assuming will be a vector of numeric values. Here is how we could define it:

```
mystats <- function(x) {
  mean(x)
  median(x)
  sd(x)
}
mystats((1:20)^2)
[1] 127.9
```

The first line says that we are defining a function called `mystats()` with one argument, named `x`. The lines surrounded by curly braces give the code to be executed when the function is called. So our function computes the mean, then the median, then the standard deviation of its argument.

But as you see, this doesn't do exactly what we wanted. So what's going on? The value returned by the last line of a function is (by default) returned by the function to its calling environment, where it is (by default) printed to the screen so you can see it. In our case, we computed the mean, median, and standard deviation, but only the standard deviation is being returned by the function and hence displayed. So this function is just an inefficient version of `sd()`. That isn't really what we wanted.

Even if you do not end up writing many functions yourself, writing a few functions will give you a much better feel for how information flows through R code.

There are ways to check the **class** of an argument to see if it is a data frame, a vector, numeric, etc. A really robust function should check to make sure that the values supplied to the arguments are of appropriate types.

We can use `print()` to print out things along the way if we like.

```
mystats <- function(x) {
  print(mean(x))
  print(median(x))
  print(sd(x))
}
mystats((1:20)^2)

[1] 143.5
[1] 110.5
[1] 127.9
```

Alternatively, we could use a combination of `cat()` and `paste()`, which would give us more control over how the output is displayed.

```
altmystats <- function(x) {
  cat(paste(" mean:", format(mean(x), 4), "\n"))
  cat(paste(" edian:", format(median(x), 4), "\n"))
  cat(paste(" sd:", format(sd(x), 4), "\n"))
}
altmystats((1:20)^2)

 mean: 143.5
 edian: 110.5
 sd: 127.9
```

Either of these methods will allow us to see all three values, but if we try to store them ...

```
temp <- mystats((1:20)^2)

[1] 143.5
[1] 110.5
[1] 127.9

temp

[1] 127.9
```

A function in R can only have one return value, and by default it is the value of the last line in the function. In the preceding example we only get the standard deviation since that is the value we calculated last.

We would really like the function to return all three summary statistics. Our solution will be to store all three in a vector and return the vector.<sup>3</sup>

<sup>3</sup> If the values had not all been of the same mode, we could have used a list instead.

```
mystats <- function(x) {
  c(mean(x), median(x), sd(x))
}
mystats((1:20)^2)

[1] 143.5 110.5 127.9
```

Now the only problem is that we have to remember which number is which. We can fix this by giving names to the slots in our vector. While we're at it, let's add a few more favorites to the list. We'll also add an explicit `return()`.

```
mystats <- function(x) {
  result <- c(min(x), max(x), mean(x), median(x), sd(x))
  names(result) <- c("min", "max", "mean", "median", "sd")
  return(result)
}
mystats((1:20)^2)

  min    max  mean median    sd
1.0 400.0 143.5 110.5 127.9

summary(Sepal.Length ~ Species, data = iris, fun = mystats)

Length  Class  Mode
      3 formula call

aggregate(Sepal.Length ~ Species, data = iris, FUN = mystats)

  Species Sepal.Length.min Sepal.Length.max
1  setosa           4.3000           5.8000
2 versicolor           4.9000           7.0000
3 virginica           4.9000           7.9000
 Sepal.Length.mean Sepal.Length.median Sepal.Length.sd
1           5.0060           5.0000           0.3525
2           5.9360           5.9000           0.5162
3           6.5880           6.5000           0.6359
```

Notice how nicely this works with `aggregate()` and with the `summary()` function from the `Hmisc` package. You can, of course, define your own favorite function to use with `summary()`. The `favstats()` function in the `mosaic` package includes the quartiles, mean, standard, deviation, sample size and number of missing observations.

```
favstats(Sepal.Length ~ Species, data = iris)

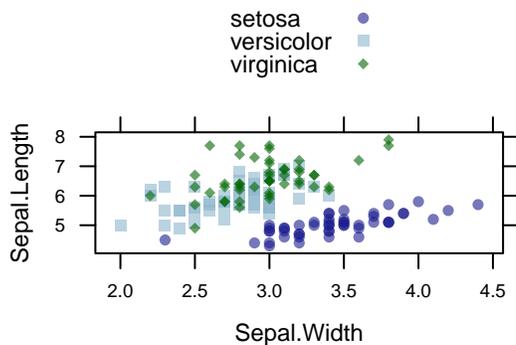
      .group min    Q1 median  Q3 max  mean    sd  n
1   setosa 4.3 4.800    5.0 5.2 5.8 5.006 0.3525 50
2 versicolor 4.9 5.600    5.9 6.3 7.0 5.936 0.5162 50
3 virginica 4.9 6.225    6.5 6.9 7.9 6.588 0.6359 50
missing
1      0
2      0
3      0
```

### 5.8 A Few Graphical Bells and Whistles

There are lots of arguments that control how lattice plots look. Here are just a few examples.

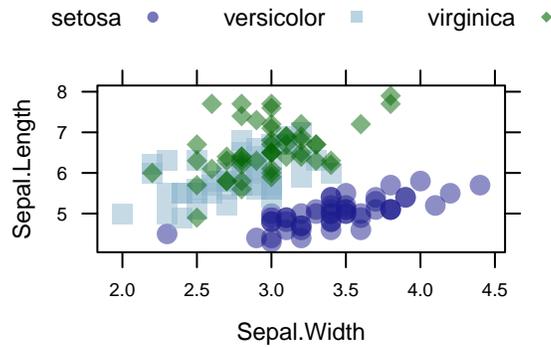
**AUTO.KEY** When using overlaid groups in a plot, it is often useful to have a legend. `auto.key=TRUE` turns on a simple legend. (There are ways to have more control, if you need it.)

```
xyplot(Sepal.Length ~ Sepal.Width, groups=Species,
       data=iris, auto.key=TRUE)
```



**ALPHA, CEX** Sometimes it is nice to have elements of a plot be partly transparent. When such elements overlap, they get darker, showing us where data are “piling up.” Setting the `alpha` argument to a value between 0 and 1 controls the degree of transparency: 1 is completely opaque, 0 is invisible. The `cex` argument controls “character expansion” and can be used to make the plotting “characters” larger or smaller by specifying the scaling ratio.

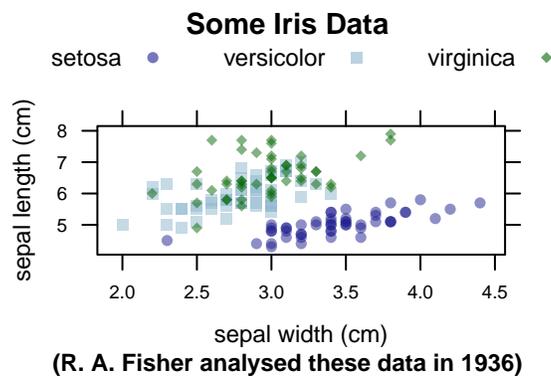
```
xyplot(Sepal.Length ~ Sepal.Width, data=iris,
       groups=Species,
       auto.key=list(columns=3),
       alpha=.5, cex=1.3)
```



MAIN, SUB, XLAB, YLAB

You can add a title or subtitle, or change the default labels of the axes.

```
xyplot(Sepal.Length ~ Sepal.Width, groups=Species, data=iris,
       main="Some Iris Data",
       sub="(R. A. Fisher analysed these data in 1936)",
       xlab="sepal width (cm)",
       ylab="sepal length (cm)",
       alpha=.5,
       auto.key=list(columns=3))
```

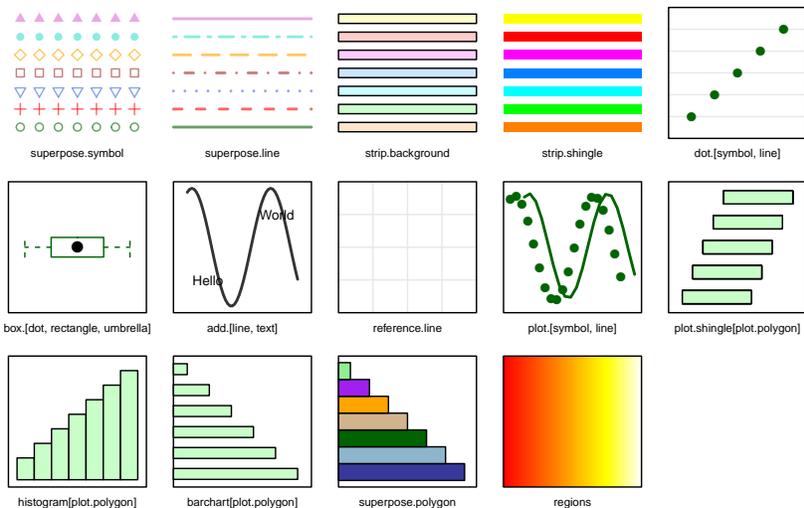


TRELLIS.PAR.SET() Default settings for lattice graphics are set using `trellis.par.set()`. Don't like the default font sizes? You can change to a 7 point (base) font using

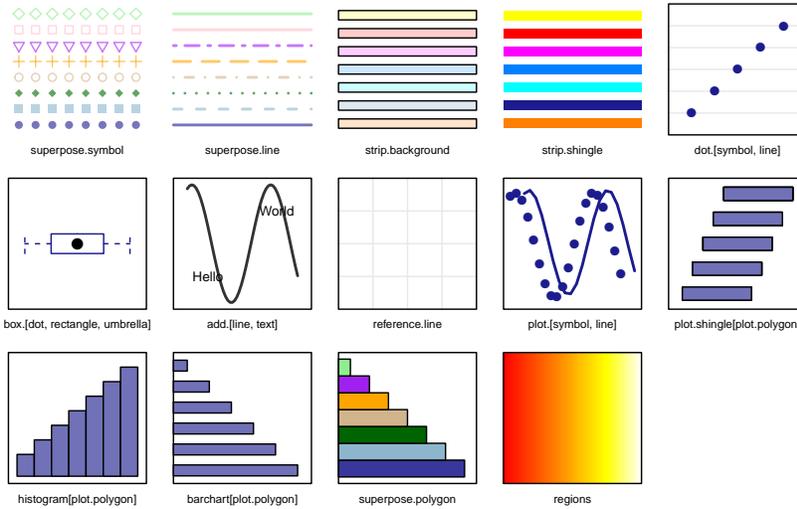
```
# set the base size for text to 7 point
trellis.par.set(fontsize = list(text = 7))
```

Nearly every feature of a lattice plot can be controlled: fonts, colors, symbols, line thicknesses, colors, etc. Rather than describe them all here, we'll mention only that groups of these settings can be collected into a theme. `show.settings()` will show you what the theme looks like.

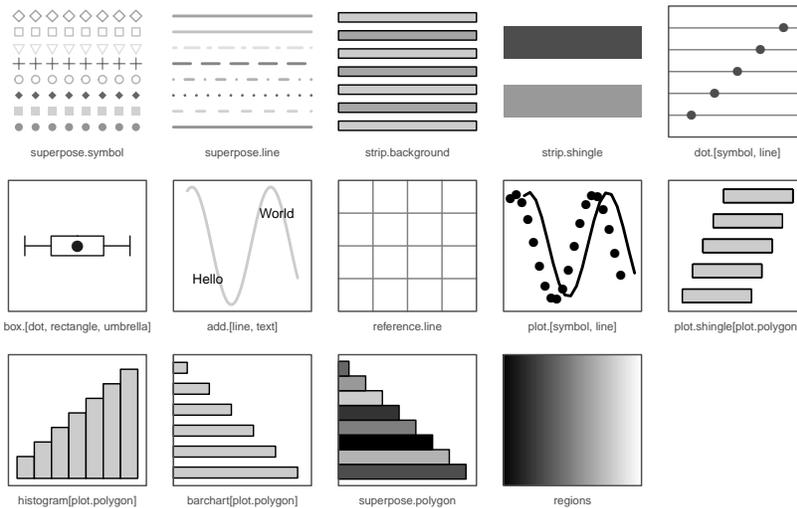
```
# a theme in the lattice package
trellis.par.set(theme = col.whitebg())
show.settings()
```



```
# a theme in the mosaic package
trellis.par.set(theme = col.mosaic())
show.settings()
```



```
# black and white version of previous theme
trellis.par.set(theme = col.mosaic(bw = TRUE))
show.settings()
```



```
# back to the mosaic theme
trellis.par.set(theme = col.mosaic())
# and back to a larger font
trellis.par.set(fontsize = list(text = 9))
```

#### DIGGING DEEPER

The `RColorBrewer` package provides several palettes of colors that are highly distinguishable and aesthetically pleasing.

## 5.9 Additional Notes on R Syntax

### 5.9.1 Text and Quotation Marks

For the most part, text in R must be enclosed in either single or double quotations. It usually doesn't matter which you use, unless you want one or the other type of quotation mark *inside* your text. Then you should use the other type of quotation mark to mark the beginning and the end.

```
# apostrophe inside requires double quotes around text
text1 <- "Mary didn't come"
# this time we flip things around
text2 <- "Do you use \"scare quotes\"?"
```

## 5.10 Common Error Messages and What Causes Them

### 5.10.1 Error: Object not found

R reports that an object is not found when it cannot locate an object with the name you have used. One common reason for this is a typing error. This is easily corrected by retyping the name with the correct spelling.

```
histogram(~aeg, data = HELPrct)

Error: object 'aeg' not found
```

Another reason for an object-not-found error is using unquoted text where quotation marks were required.

```
text3 <- hello

Error: object 'hello' not found
```

In this case, R is looking for some object named `hello`, but we meant to store a string:

```
text3 <- "hello"
```

### 5.10.2 Error: unexpected ...

If while R is parsing a statement it encounters something that does not make sense it reports that something is “unexpected”. Often this is the result of a typing error – like omitting a comma.

```
c(1,2 3) # missing a comma
Error: unexpected numeric constant in "c(1,2 3"
```

### 5.10.3 Error: object of type 'closure' is not subsettable

The following produces an error if `time` has not been defined.

```
time[3]
Error: object of type 'closure' is not subsettable
```

There is a function called `time()` in R, so if you haven't defined a vector by that name, R will try to subset the `time()` function, which doesn't make sense.

Typically when you see this error, you have a function in a place you don't mean to have a function. The message can be cryptic to new users because of the reference to a closure.

### 5.10.4 Other Errors

If you encounter other errors and cannot decipher them, often pasting the error message into a google search will find a discussion of that error in a context where it stumped someone else.



## 5.11 Review of R Commands

Here is a brief summary of the commands introduced in this chapter.

```

source( "file.R" ) # execute commands in a file

x <- 1:10 # create vector with numbers 1 through 10
M <- matrix( 1:12, nrow=3 ) # create a 3 x 4 matrix
data.frame(number = 1:26, letter=letters[1:26] ) # create a data frame

mode(x) # returns mode of object x
length(x) # returns length of vector or list
dim(HELPrct) # dimension of a matrix, array, or data frame
nrow(HELPrct) # number of rows
ncol(HELPrct) # number of columns
names( HELPrct ) # variable names in data frame
row.names( HELPrct ) # row names in a data frame
attributes(x) # returns attributes of x

toupper(x) # capitalize
as.character(x) # convert to a character vector
as.logical(x) # convert to a logical (TRUE or FALSE)
as.numeric(x) # convert to numbers
as.integer(x) # convert to integers
factor(x) # convert to a factor [categorical data]
class(x) # returns class of x

smallPrimes <- c(2,3,5,7,11) # create a (numeric) vector
rep(1, 10) # ten 1's
seq(2, 10, by=2) # evens less than or equal to 10
rank(x) # ranks of items in x
sort(x) # returns elements of x in sorted order
order(x) # x[ order(x) ] is x in sorted order
rev(x) # returns elements of x in reverse order
diff(x) # returns differences between consecutive elements
paste( "Group", 1:3, sep="" ) # same as c("Group1", "Group2", "Group3")

write.table(HELPrct, file="myHELP.txt") # write data to a file
write.csv(HELPrct, file="myHELP.csv") # write data to a csv file
save(HELPrct, file="myHELP.Rda") # save object(s) in R's native format

modData <- transform( HELPrct, old = age > 50 ) # add a new variable to data frame
women <- subset( HELPrct, sex=='female' ) # select only specified cases
favs <- subset( HELPrct, select=c('age','sex','substance') ) # keep only 3 columns

trellis.par.set(theme=col.mosaic()) # choose theme for lattice graphics
show.settings() # inspect lattice theme
fetchGoogle( ... ) # get data from google URL

```

## 5.12 Exercises

**5.1** Using `faithful` data frame, make a scatter plot of eruption duration times vs. the time since the previous eruption.

**5.2** The `fusion2` data set in the `fastR` package contains genotypes for another SNP. Merge `fusion1`, `fusion2`, and `pheno` into a single data frame.

Note that `fusion1` and `fusion2` have the same columns.

```
names(fusion1)
[1] "id"      "marker"  "markerID" "allele1" "allele2"
[6] "genotype" "Adose"   "Cdose"    "Gdose"   "Tdose"

names(fusion2)
[1] "id"      "marker"  "markerID" "allele1" "allele2"
[6] "genotype" "Adose"   "Cdose"    "Gdose"   "Tdose"
```

You may want to use the `suffixes` argument to `merge()` or rename the variables after you are done merging to make the resulting data frame easier to navigate.

Tidy up your data frame by dropping any columns that are redundant or that you just don't want to have in your final data frame.

**5.3** The `Jordan8687` data set (in the `fastR` package) contains the number of points Michael Jordan scored in each game of the 1986–87 season.

- Make a histogram of this data. Add an appropriate title.
- How would you describe the shape of the distribution?
- In approximately what percentage of his games, did Michael Jordan score less than 20 points? More than 50? (You may want to add `breaks=seq(0, 70, by=5)` to your command to neaten up the bins.)

**5.4** Cuckoos lay their eggs in the nests of other birds. Is the size of cuckoo eggs different in different host species nests? The cuckoo data set (in `fastR`) contains data from a study attempting to answer this question.

- a) When were these data collected? (Use `?cuckoo` to get information about the data set.)
- b) What are the units on the length measurements?
- c) Make side-by-side boxplots of the length of the eggs by species.
- d) Calculate the mean length of the eggs for each host species.
- e) What do you think? Does it look like the size differs among the different host species? Refer to your R output as you answer this question. (We'll learn formal methods to investigate this later in the semester.)

## 6

# *Getting Interactive with manipulate and shiny*

One very attractive feature of RStudio is the `manipulate()` function (in the `manipulate` package, which is only available within RStudio). This function makes it easy to create a set of controls (such as sliders, checkboxes, drop down selections, etc.) that can be used to dynamically change values within an expression. When a value is changed using these controls, the expression is automatically re-executed and any plots created as a result are redrawn. This can be used to quickly prototype a number of activities and demos as part of a statistics lecture.

`shiny` is a new web development system for R being designed by the RStudio team. `shiny` uses a reactive programming model to make it relatively easy for an R programmer to create highly interactive, well designed web applications using R without needing to know much about web programming. Programming in `shiny` is more involved than using `manipulate`, but it offers the designer more flexibility. One of the goals in creating `shiny` was to support corporate environments, where a small number of statisticians and programmers can create web applications that can be used by others within the company without requiring them to know any R. This same framework offers many possibilities for educational purposes as well. Some have even suggested implementing fairly extensive GUI interfaces to commonly used R functionality using `shiny`.

### *6.1 Getting Started with manipulate*

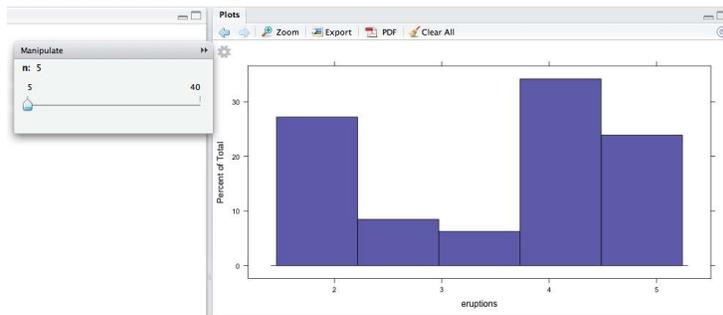
The `manipulate()` function and the various control functions that are used with it are only available after loading the `manipulate` package, which is only available in RStudio.

```
require(manipulate)
```

### 6.1.1 Sliders

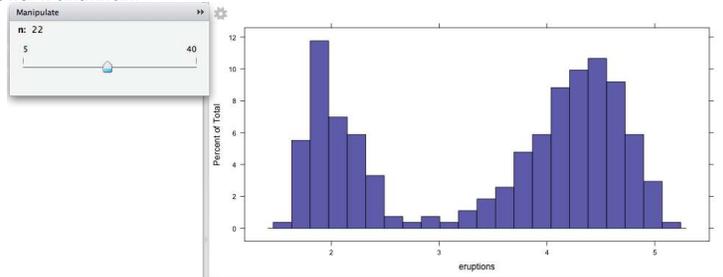
```
manipulate(
  histogram( ~ eruptions, data=faithful, n=N),
  N = slider(5,40)
)
```

This generates a plot along with a slider ranging from 5 bins to 40.



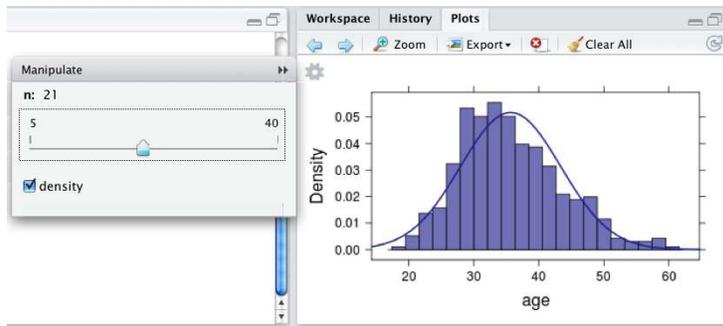
We find it useful to capitalize the inputs to the manipulated expression that are hooked up to `manipulate` controls. This helps avoid naming collisions and signals how the main manipulated expression is being used.

When the slider is changed, we see a clearer view of the eruptions of Old Faithful.



### 6.1.2 Check Boxes

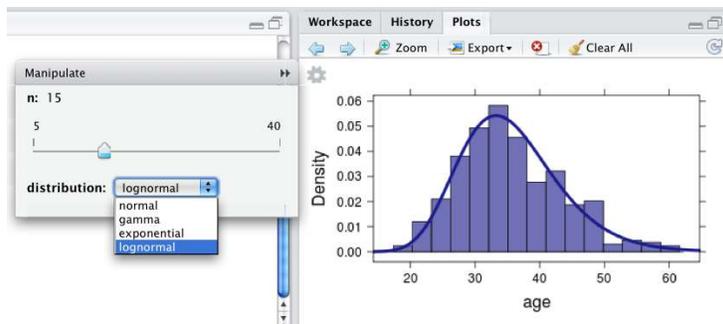
```
manipulate(
  histogram( ~ age, data=HELPrct, n=N, density=DENSITY),
  N = slider(5,40),
  DENSITY = checkbox()
)
```



### 6.1.3 Drop-down Menus

Drop-down menus can be added using the `picker()` function.

```
manipulate(
  histogram( ~ age, data=HELPrct, n=N,
             fit=DISTRIBUTION, dlwd=4),
  N = slider(5,40),
  DISTRIBUTION =
    picker('normal', 'gamma', 'exponential', 'lognormal',
           label="distribution")
)
```

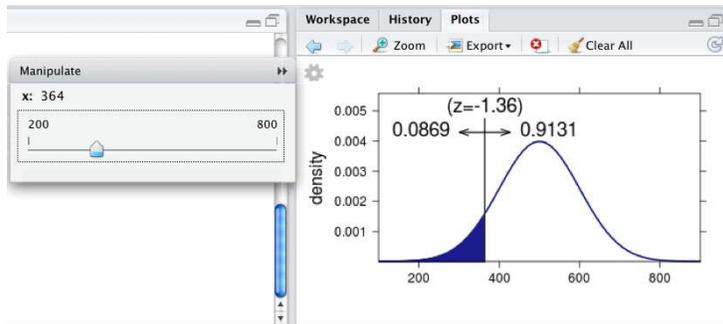


### 6.1.4 Visualizing Normal Distributions

In this section we will gradually build up a small `manipulate` example that shows the added flexibility that comes from writing a function that returns a `manipulate` object. Such functions can be distributed to students to allow them to explore interactively in a more flexible way.

We begin by creating an illustration of tail probabilities in a normal distribution.

```
manipulate(
  xpnorm( X, 500, 100, verbose=FALSE, invisible=TRUE ),
  X = slider(200,800) )
```



The version below can be used to investigate central probabilities and tail probabilities.

```
manipulate(
  xpnorm( c(-X,X), 500, 100, verbose=FALSE, invisible=TRUE ),
  X = slider(200,800) )
```

These examples work with a fixed distribution. Here is a fancier version in which a function returns a manipulate object. This allows us to easily create illustrations like the ones above for any normal distribution.

```
mNorm <- function( mean=0, sd=1 ) {
  lo <- mean - 5*sd
  hi <- mean + 5*sd
  manipulate(
    xpnorm( c(A,B), mean, sd, verbose=FALSE, invisible=TRUE ),
    A = slider(lo, hi, initial=mean-sd),
    B = slider(lo, hi, initial=mean+sd)
  )
}
mNorm( mean=100, sd=10 )
```

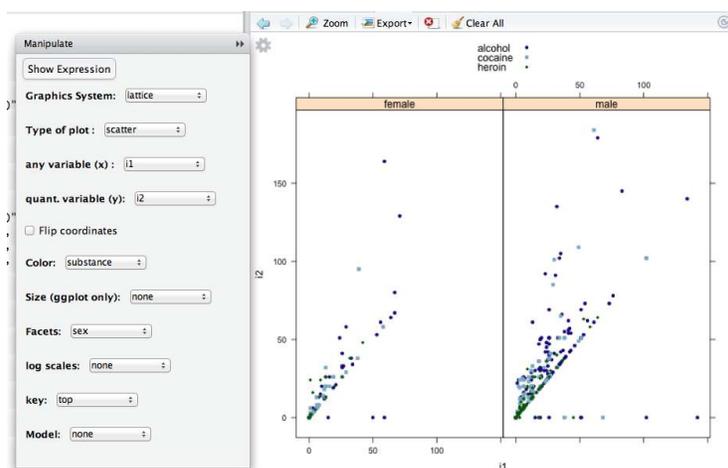
## 6.2 mPlot()

The `mosaic` package provides the `mPlot()` function which allows users to create a wide variety of plots using either `lattice` or `ggplot2`. Furthermore, the code used to generate these plots can be displayed upon request. This facilitates learning these commands, allows users

to make further modifications that are not possible in the manipulate interface, and provides an easy copy-and-paste mechanism for dropping these plots into other documents.

The available plots come in two clusters, depending on whether the underlying plot is essentially two-variable or one-variable. Additional variables can be represented using color, size, and sub-plots (facets).

```
# These are essentially 2-variable plots
mPlot(HELPrct, "scatter") # start with a scatter plot
mPlot(HELPrct, "boxplot") # start with boxplots
mPlot(HELPrct, "violin") # start with violin plots
# These are essentially 1-variables plots
mPlot(HELPrct, "histogram") # start with a histogram
mPlot(HELPrct, "density") # start with a density plot
mPlot(HELPrct, "frequency polygon") # start with a frequency polygon
```



### 6.3 Shiny

**shiny** is a package created by the RStudio team to, in their words,

[make] it incredibly easy to build interactive web applications with R. Automatic “reactive” binding between inputs and outputs and extensive pre-built widgets make it possible to build beautiful, responsive, and powerful applications with minimal effort.

These web applications can, of course, run R code to do computations and produce graphics that appear in the web page.

The level of coding skill required to create this is beyond the scope of this book, but those with a little more programming background can easily learn the necessary toolkit to make beautiful interactive

web pages. More information about `shiny` and some example applications are available at <http://www.rstudio.com/shiny/>.

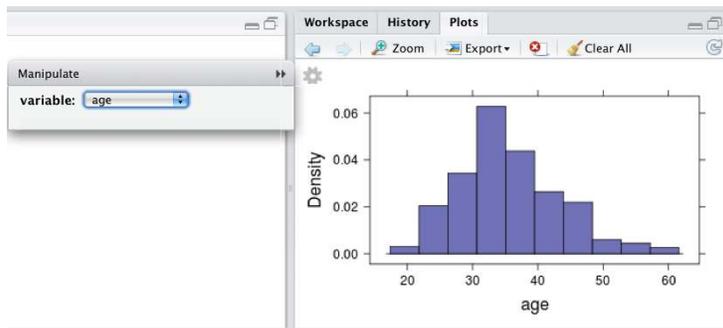
## Exercises

**6.1** The following code makes a scatterplot with separate symbols for each sex.

```
xypLOT(cesd ~ age, data = HELPrct, groups = sex)
```

Build a `manipulate` example that allows you to turn the grouping on and off with a checkbox.

**6.2** Build a `manipulate` example that uses a picker to select from a number of variables to make a plot for. Here's an example with a histogram:



**6.3** Design your own interactive demonstration idea and implement it using RStudio `manipulate` tools.

# 7

## *Bibliography*

- [Fis25] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver & Boyd, 1925.
- [Fis70] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver & Boyd, 14th edition, 1970.
- [NT10] D. Nolan and D. Temple Lang. Computing in the statistics curriculum. *The American Statistician*, 64(2):97–107, 2010.
- [Sal01] D. Salsburg. *The Lady Tasting Tea: How statistics revolutionized science in the twentieth century*. W.H. Freeman, New York, 2001.

# 8

## Index

->, 95  
<-, 95  
=, 95  
[ ], 65, 67  
[[ ]], 67  
#, 65  
  
any(), 74  
argument of an R function, 94  
as.integer(), 69  
as.numeric(), 69  
attr(), 65, 68  
attributes(), 65, 68  
  
c(), 79  
cat(), 96  
cbind(), 72  
comment character in R (#), 65  
cummax(), 74  
cummin(), 74  
cumprod(), 74  
cumsum(), 74  
  
data.frame(), 78  
diff(), 74  
dim(), 68  
  
favstats(), 97  
Fisher, R. A., 22  
  
function(), 95  
functions in R, 94  
  
install.packages(), 34  
is.integer(), 69  
is.numeric(), 69  
  
length(), 65  
letters[], 68  
log(), 71  
  
matrix(), 67  
mean(), 71  
median(), 71  
mode(), 65  
mystats(), 95  
  
na.omit(), 74  
names(), 68  
ncol(), 68  
nrow(), 68  
  
order(), 74  
  
paste(), 74, 96  
pmax(), 74  
pmin(), 74  
prod(), 74  
  
rank(), 74  
  
rbind(), 72  
recycling, 74  
rep(), 79  
require(), 34  
return(), 97  
rev(), 74  
RMySQL, 78  
rnorm(), 79  
round(), 72  
row.names(), 68  
  
sample(), 79  
scan(), 78  
sd(), 71  
seq(), 79  
signif(), 72  
sort(), 74  
SQL, 78  
sum(), 74  
  
table(), 74  
tolower(), 76  
toupper(), 76  
  
unique(), 74  
  
var(), 71  
  
which(), 74