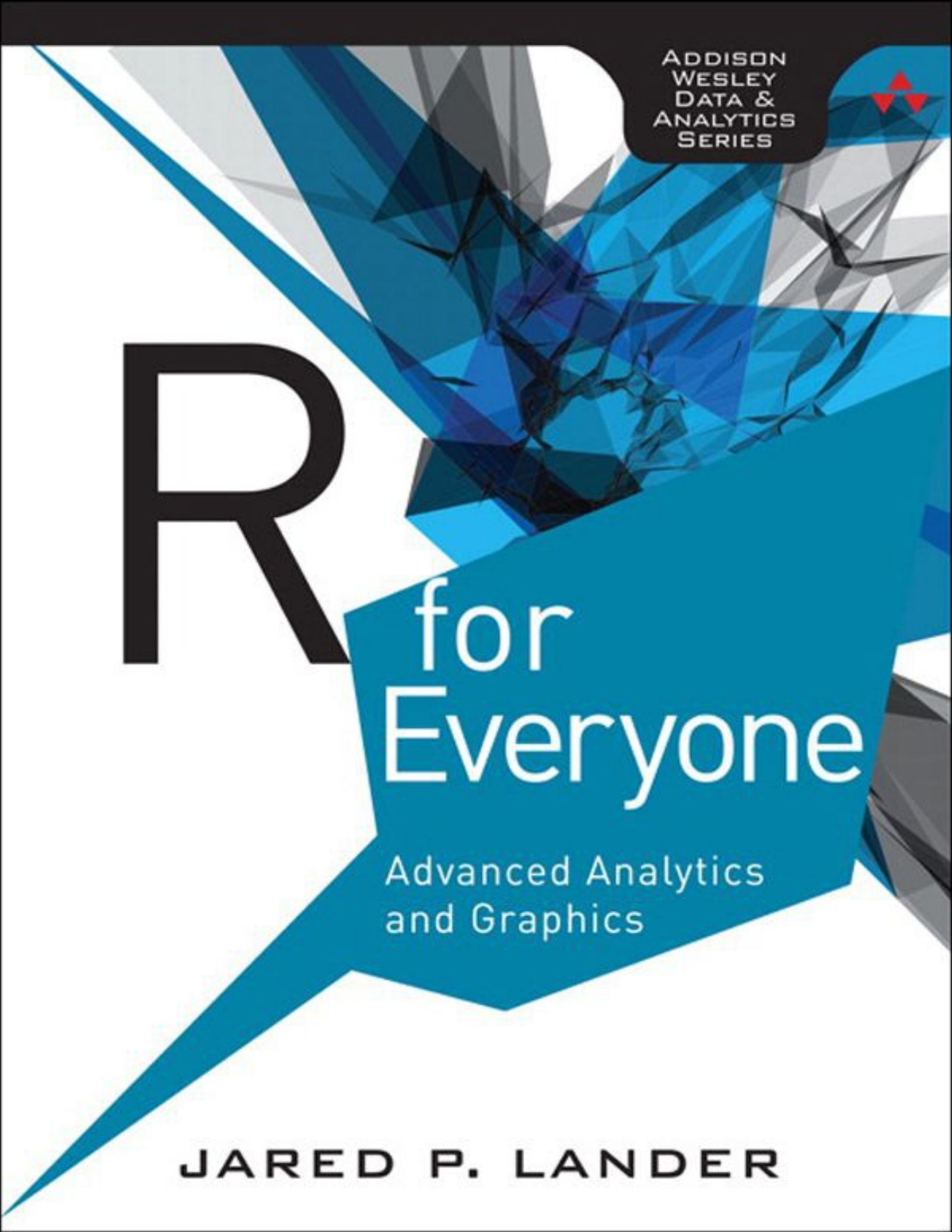
Addison
Wesley
Data &
Analytics
Series
R
for
Everyone
Advanced Analytics
and Graphics
JARED P. LANDER

R for Everyone

Advanced Analytics and Graphics

Jared P. Lander

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

About the Author

Jared P. Lander is the founder and CEO of Lander Analytics, a statistical consulting firm based in New York City, the organizer of the New York Open Statistical Programming Meetup, and an adjunct professor of statistics at Columbia University. He is also a tour guide for Scott's Pizza Tours and an advisor to Brewla Bars, a gourmet ice pop start-up. With an M.A. from Columbia University in statistics and a B.A. from Muhlenberg College in mathematics, he has experience in both academic research and industry. His work for both large and small organizations spans politics, tech start-ups, fund-raising, music, finance, healthcare and humanitarian relief efforts.

He specializes in data management, multilevel models, machine learning, generalized linear models, visualization, data management and statistical computing.

Chapter 1. Getting R

R is a wonderful tool for statistical analysis, visualization and reporting. Its usefulness is best seen in the wide variety of fields where it is used. We alone have used R for projects with banks, political campaigns, tech startups, food startups, international development and aid organizations, hospitals and real estate developers. Other areas where we have seen it used are online advertising, insurance, ecology, genetics and pharmaceuticals. R is used by statisticians with advanced machine learning training and by programmers familiar with other languages, and also by people who are not necessarily trained in advanced data analysis but are tired of using Excel.

Before it can be used it needs to be downloaded and installed, a process that is no more complicated than installing any other program.

1.1. Downloading R

The first step in using R is getting it on the computer. Unlike with languages such as C++, R must be installed in order to run.[1] The program is easily obtainable from the Comprehensive R Archive Network (CRAN), the maintainer of R, at http://cran.r-project.org/. At the top of the page are links to download R for Windows, Mac OS X and Linux.

1. Technically C++ cannot be set up on its own without a compiler, so something would still need to be installed anyway.

There are prebuilt installations available for Windows and Mac OS X while those for Linux usually compile from source. Installing R on any of these platforms is just like installing any other program.

Windows users should click the link Download R for Windows, then base and then Download R 3.x.x for Windows; the x's indicate the version of R. This changes periodically as improvements are made.

Similarly, Mac users should click Download R for (Mac) OS X and then R-3.x.x.pkg; again, the x's indicate the current version of R. This will also install both 32- and 64-bit versions.

Linux users should download R using their standard distribution mechanism whether that is apt-get (Ubuntu and Debian), zypper (SUSE) or another source. This will also build and install R.

1.2. R Version

As of this writing, R is at version 3.0.2, which is a big jump from the previous version, 2.15.3. CRAN follows a one-year release cycle where each major version change increases the middle of the three numbers in the version. For instance, version 3.0.0 was released in 2013. In 2014 the version will be incremented to 3.1.0 with 3.2.0 coming in 2015. The last number in the version is for minor updates to the current major version.

Most R functionality is usually backward compatible with previous versions.

1.3. 32-bit versus 64-bit

The choice between using 32-bit and using 64-bit comes down to whether the computer supports 64-bit—most new machines do—and the size of the data to be worked with. The 64-bit versions can address arbitrarily large amounts of memory (or RAM) so it might as well be used.

This is especially important starting with version 3.0.0, as that adds support for 64-bit integers,

meaning far greater amounts of data can be stored in R objects.

In the past, certain packages required the 32-bit version of R but that is exceedingly rare these days. The only reason for installing the 32-bit version now is to support some legacy analysis or for use on a machine with a 32-bit processor such as Intel's low-power Atom chip.

1.4. Installing

Installing R on Windows and Mac is just like installing any other program.

1.4.1. Installing on Windows

Find the appropriate installer where it was downloaded. For Windows users it will look like Figure 1.1.

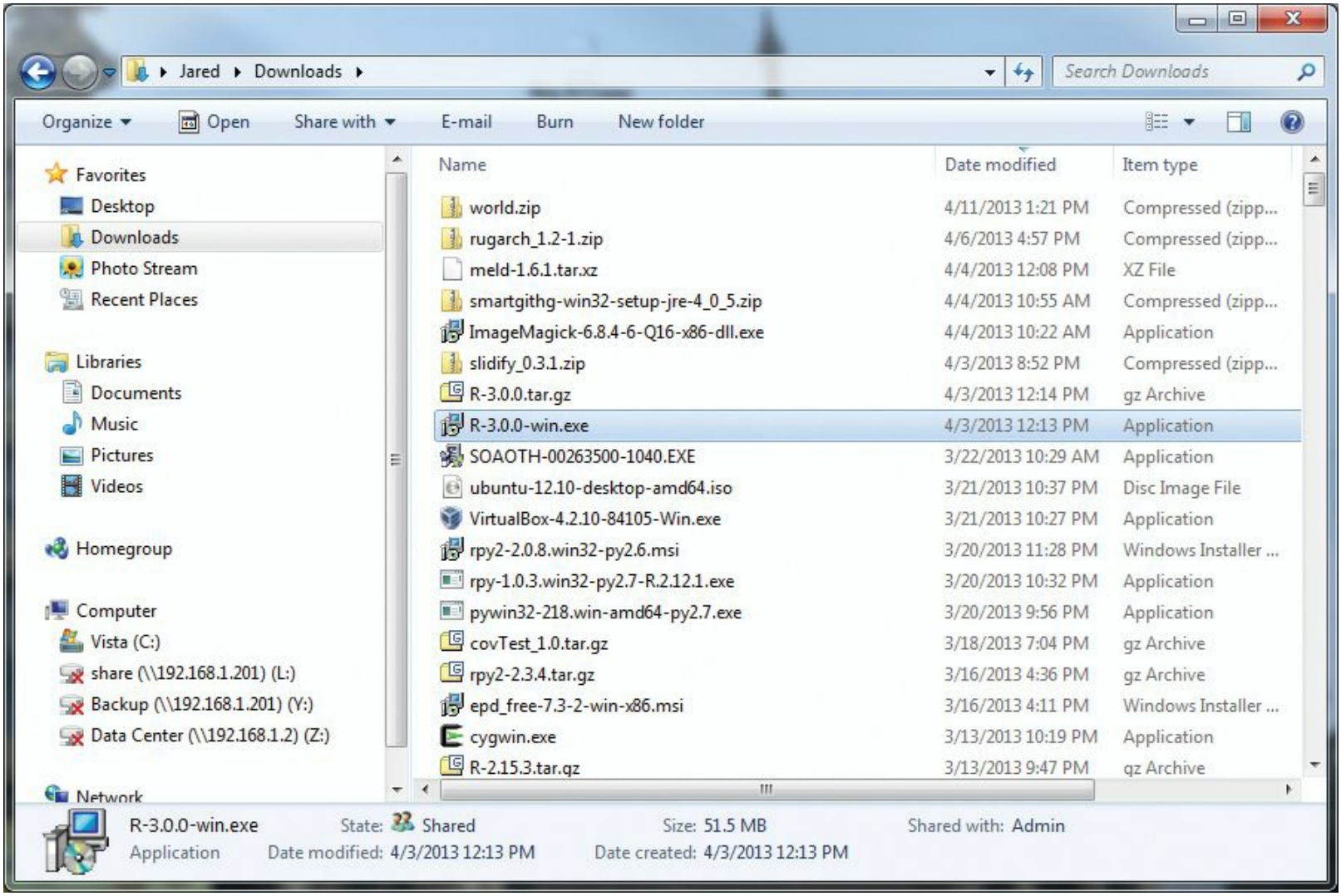

Figure 1.1 Location of R installer.

R should be installed using administrator privileges. This means right-clicking the installer and then selecting Run as Administrator. This brings up a prompt where the administrator password should be entered.

The first dialog, shown in Figure 1.2, offers a choice of language, defaulted at English. Choose the appropriate language and click OK.

Figure 1.2 Language selection.

Next, the caution shown in Figure 1.3 recommends that all other programs be closed. This advice is rarely followed or necessary anymore, so clicking Next is appropriate.

Figure 1.3 With modern versions of Windows, this suggestion can be safely ignored.

The software license is then displayed, as in Figure 1.4. R cannot be used without agreeing to this (important) license, so the only recourse is to click Next.

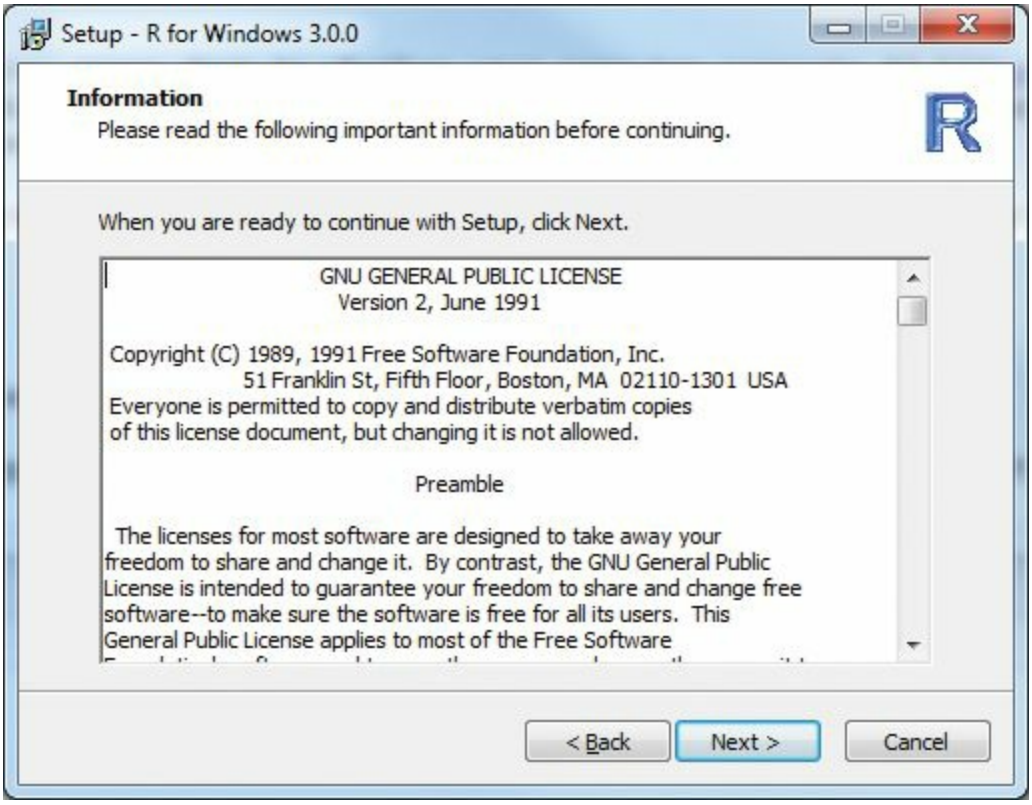

Figure 1.4 The license agreement must be acknowledged to use `R`.

The installer then asks for a destination location. Even though the official advice from CRAN is that `R` should be installed in a directory with no spaces in the name, half the time the default installation directory is `Program Files\R`, which causes trouble if we try to build packages that require compiled code such as C++ for FORTRAN. Figure 1.5 shows this dialog.

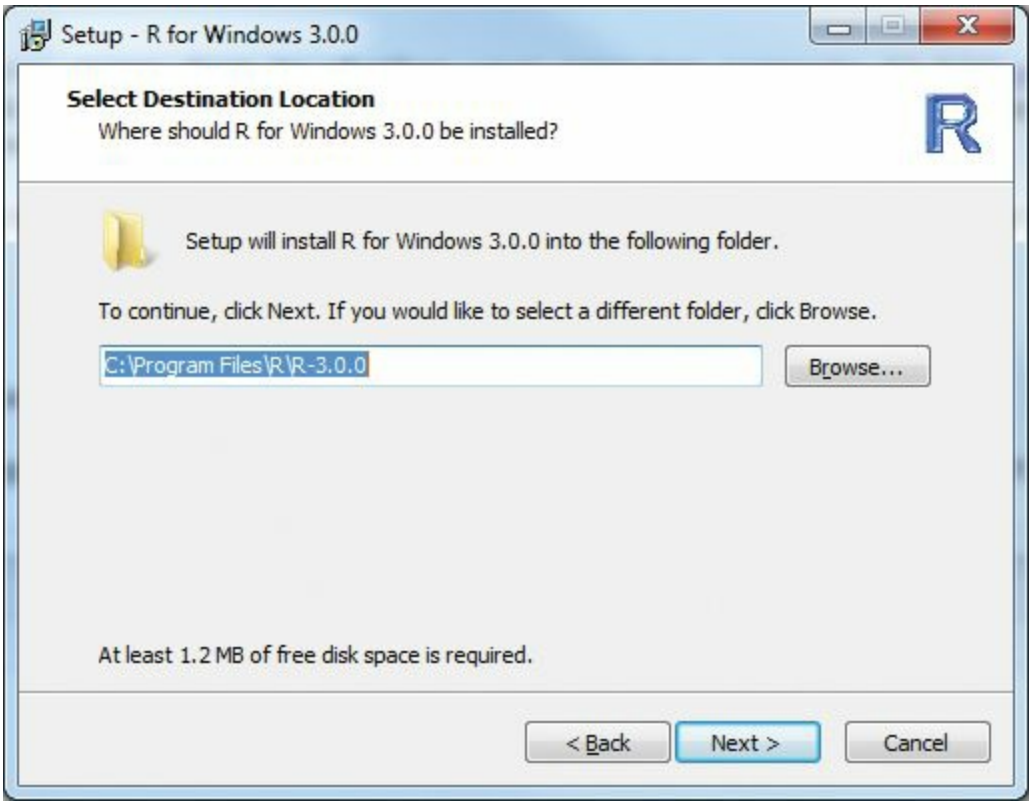

Figure 1.5 It is important to choose a destination folder with no spaces in the name.

If that is the case, click the Browse button to bring up folder options like the ones shown in Figure 1.6.

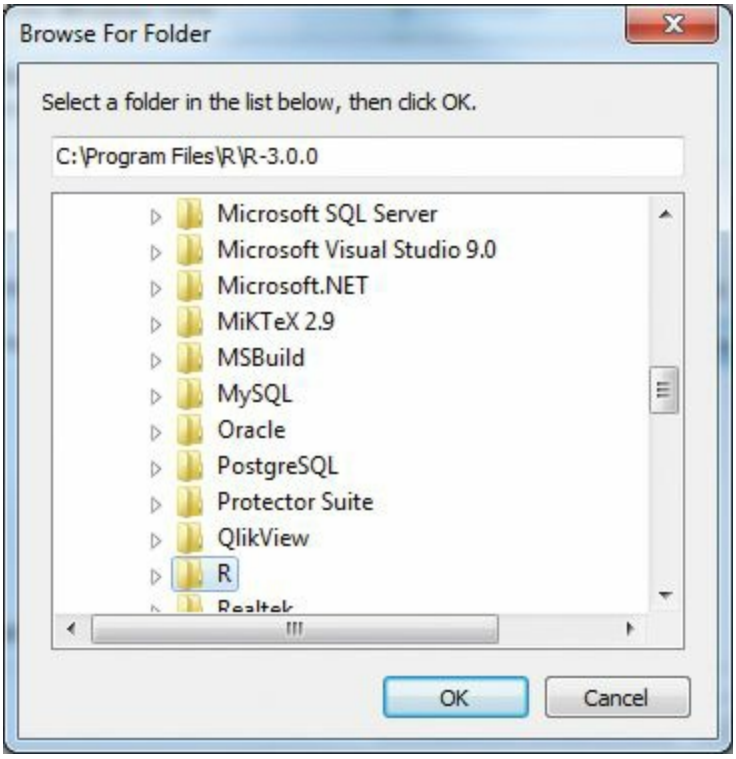

Figure 1.6 This dialog is used to choose the destination folder.

It is best to choose a destination folder that is on the `C:` drive (or another hard disk drive) or inside `My Documents`, which despite that user-friendly name is actually located at `C:\Users\UserName\Documents`, which contains no spaces. Figure 1.7 shows a proper destination for the installation.

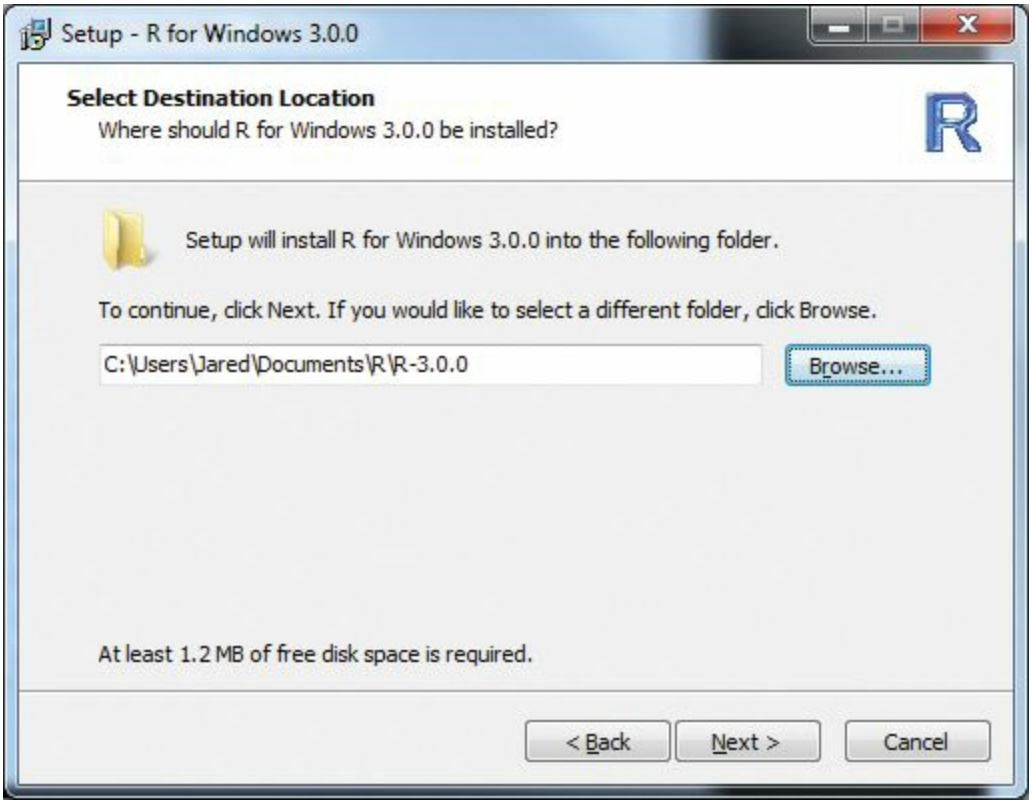

Figure 1.7 This is a proper destination, with no spaces in the name.

Next, Figure 1.8, shows a list of components to install. Unless there is a specific need for 32-bit files, that option can be unchecked. Everything else should be selected.

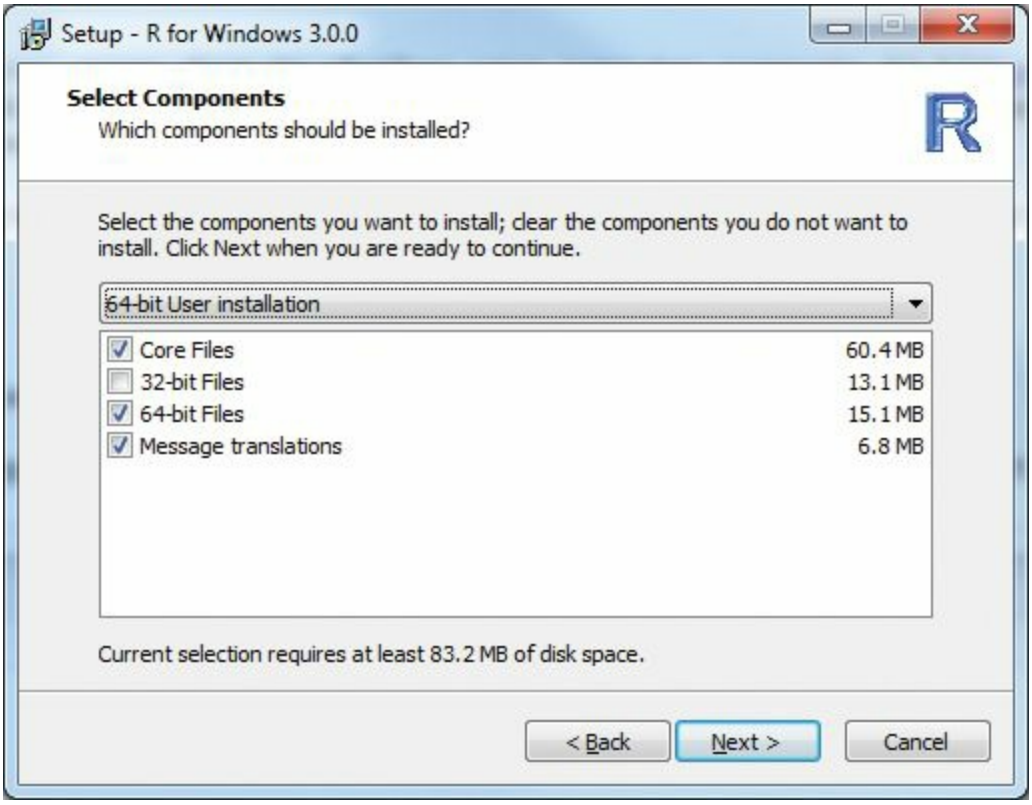

Figure 1.8 It is best to select everything except 32-bit components.

The startup options should be left at the default, No, as in Figure 1.9, because there are not a lot of options and we recommend using RStudio as the front end anyway.

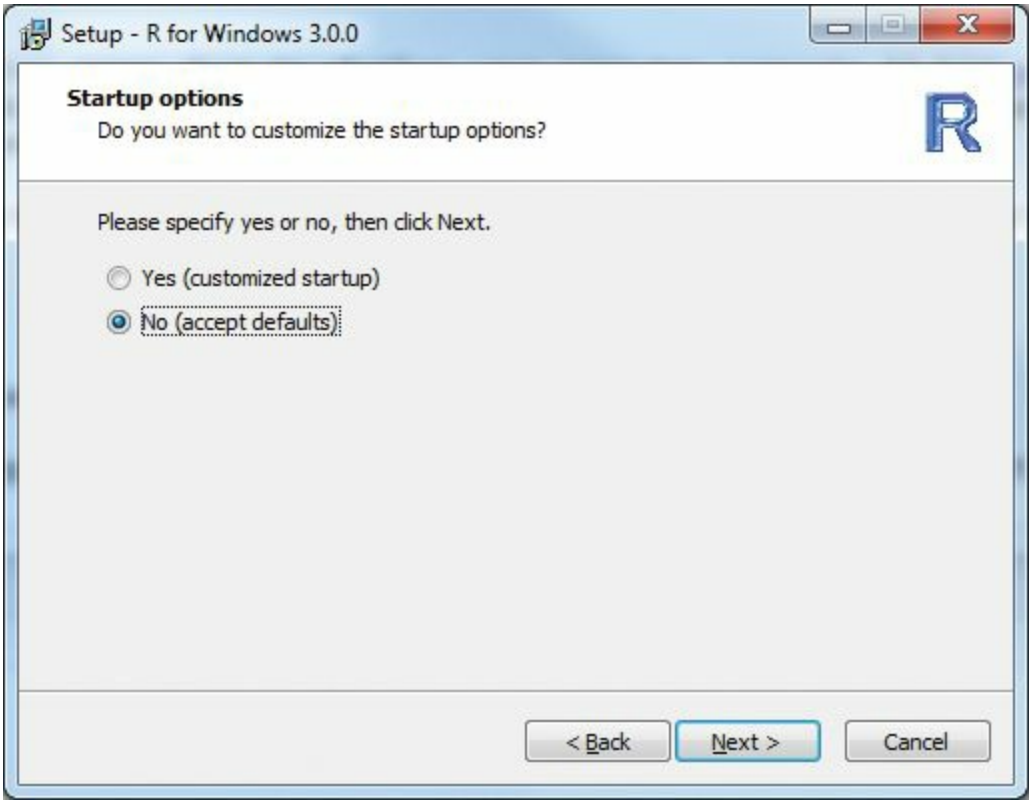

Figure 1.9 Accept the default startup options, as we recommend using RStudio as the front end and these will not be important.

Next, choose where to put the start menu shortcuts. We recommend simply using `R` and putting every version in there as shown in Figure 1.10.

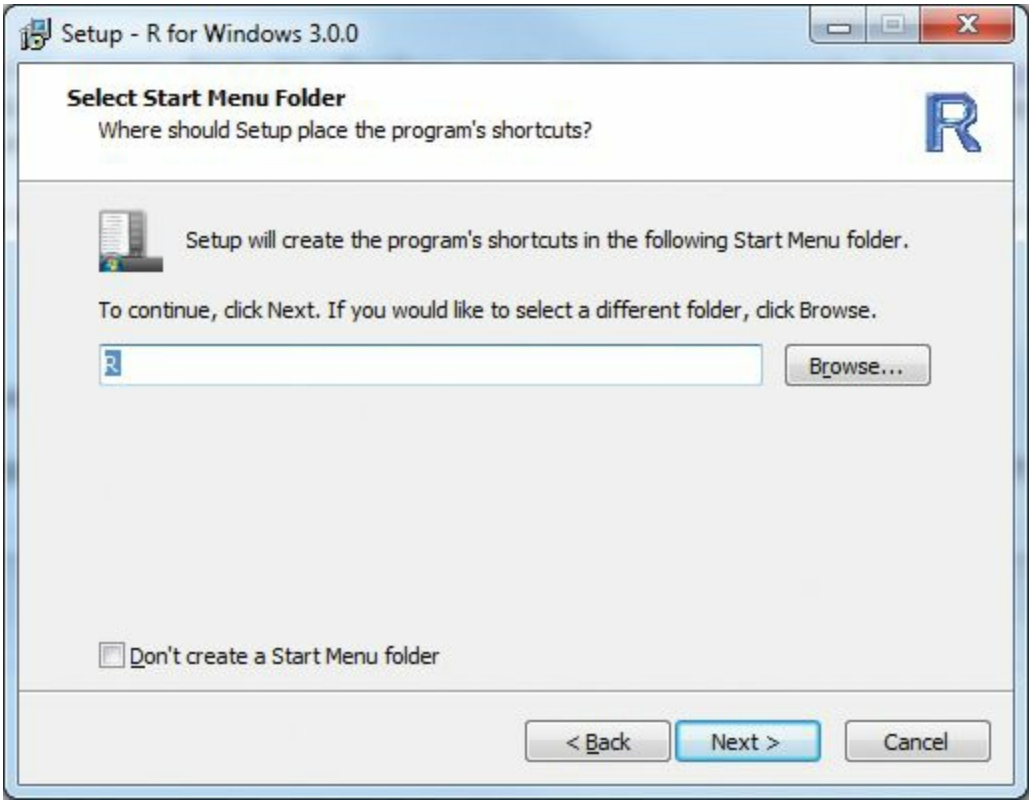

Figure 1.10 Choose the Start Menu folder where the shortcuts will be installed.

We have many versions of `R`, all inside the same Start Menu folder, which allows code to be tested in different versions. This is illustrated in Figure 1.11.

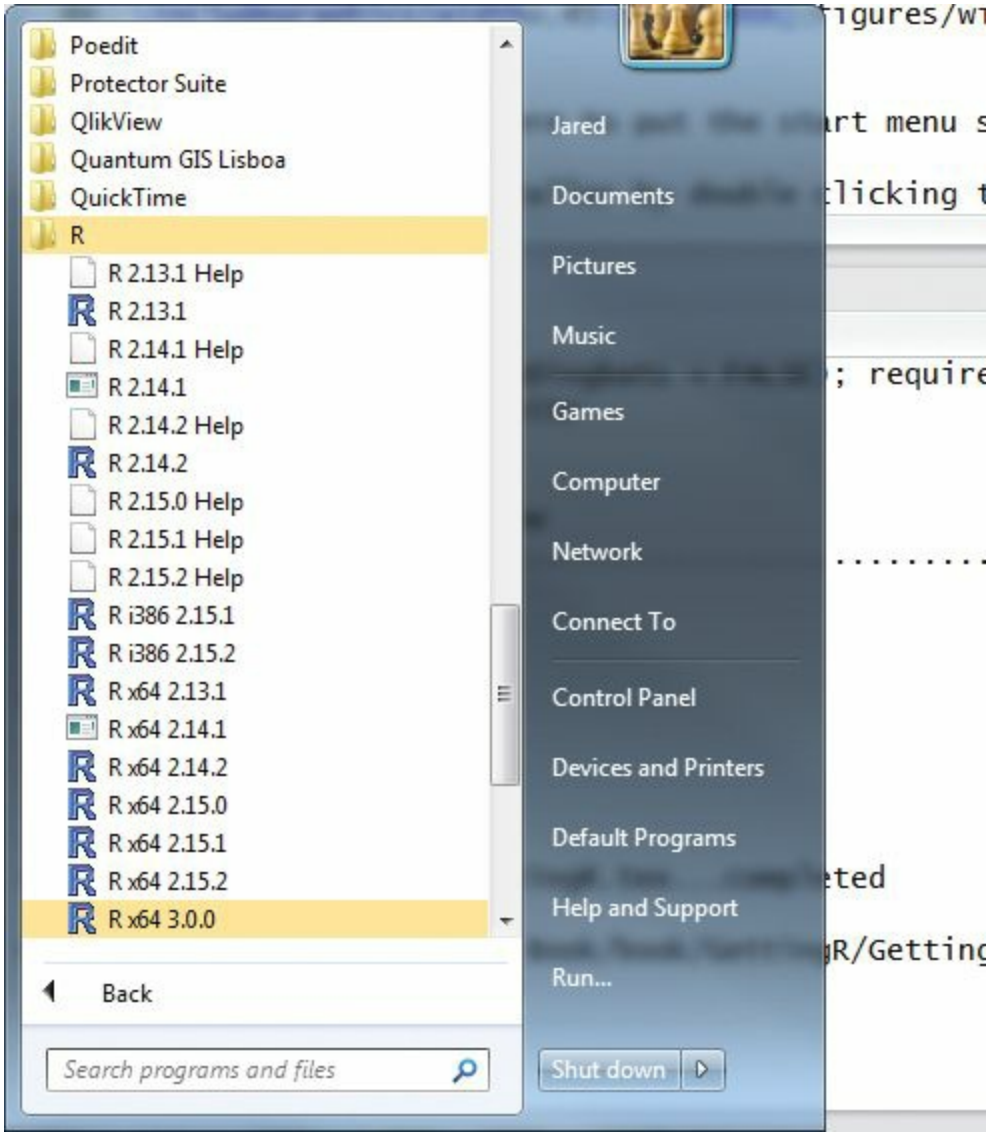

Figure 1.11 We have multiple versions of `R` installed to allow development and testing with different versions.

The last option is choosing whether to complete some additional tasks such as creating a desktop icon (not too useful if using RStudio). We highly recommend saving the version number in the registry and associating `R` with `RData` files. These options are shown in Figure 1.12.

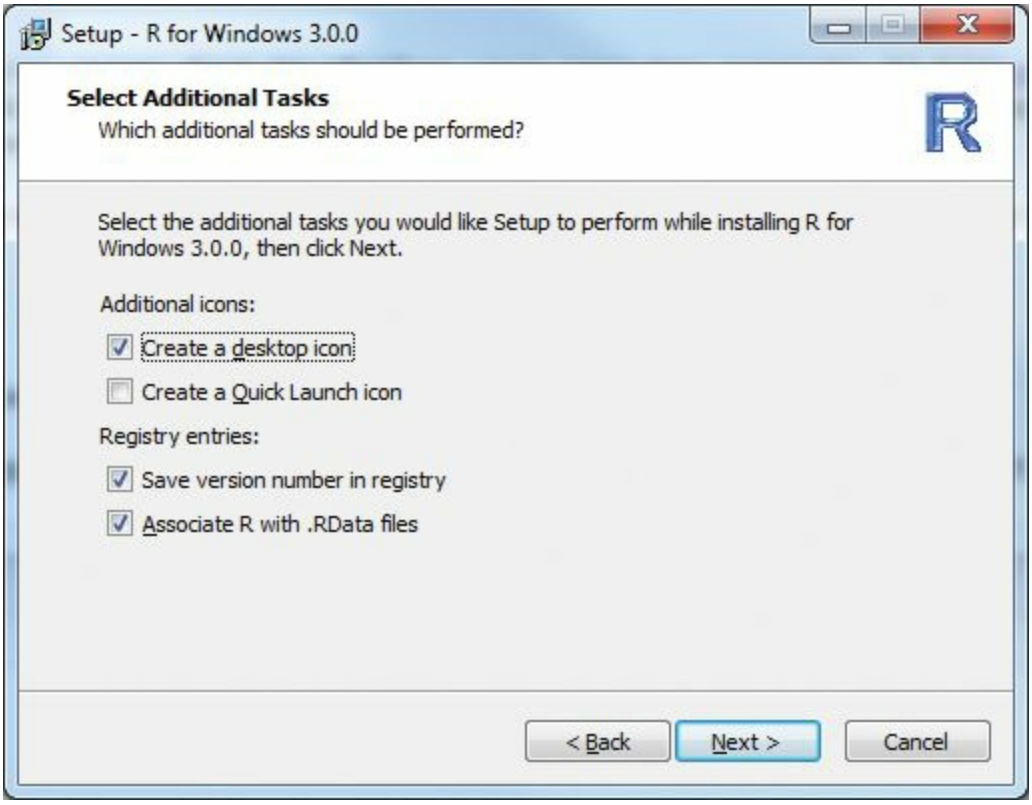

Figure 1.12 We recommend saving the version number in the registry and associating `R` with `RData` files.

Clicking Next begins installation and displays a progress bar, as shown in Figure 1.13.

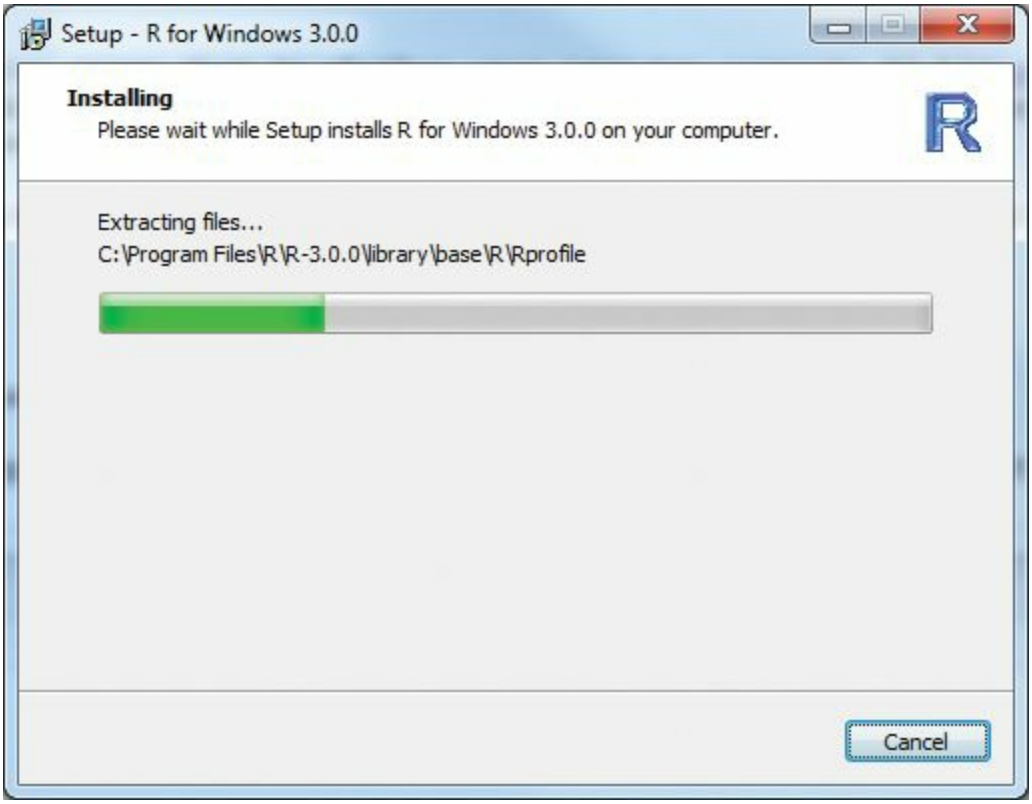

Figure 1.13 A progress bar is displayed during installation.

The last step, shown in Figure 1.14, is to click Finish and the installation is complete.

Figure 1.14 Confirmation that installation is complete.

1.4.2. Installing on Mac OS X

Find the appropriate installer, which ends in `.pkg`, and launch it by double-clicking. This brings up the introduction, shown in Figure 1.15. Click Continue to begin the installation process.

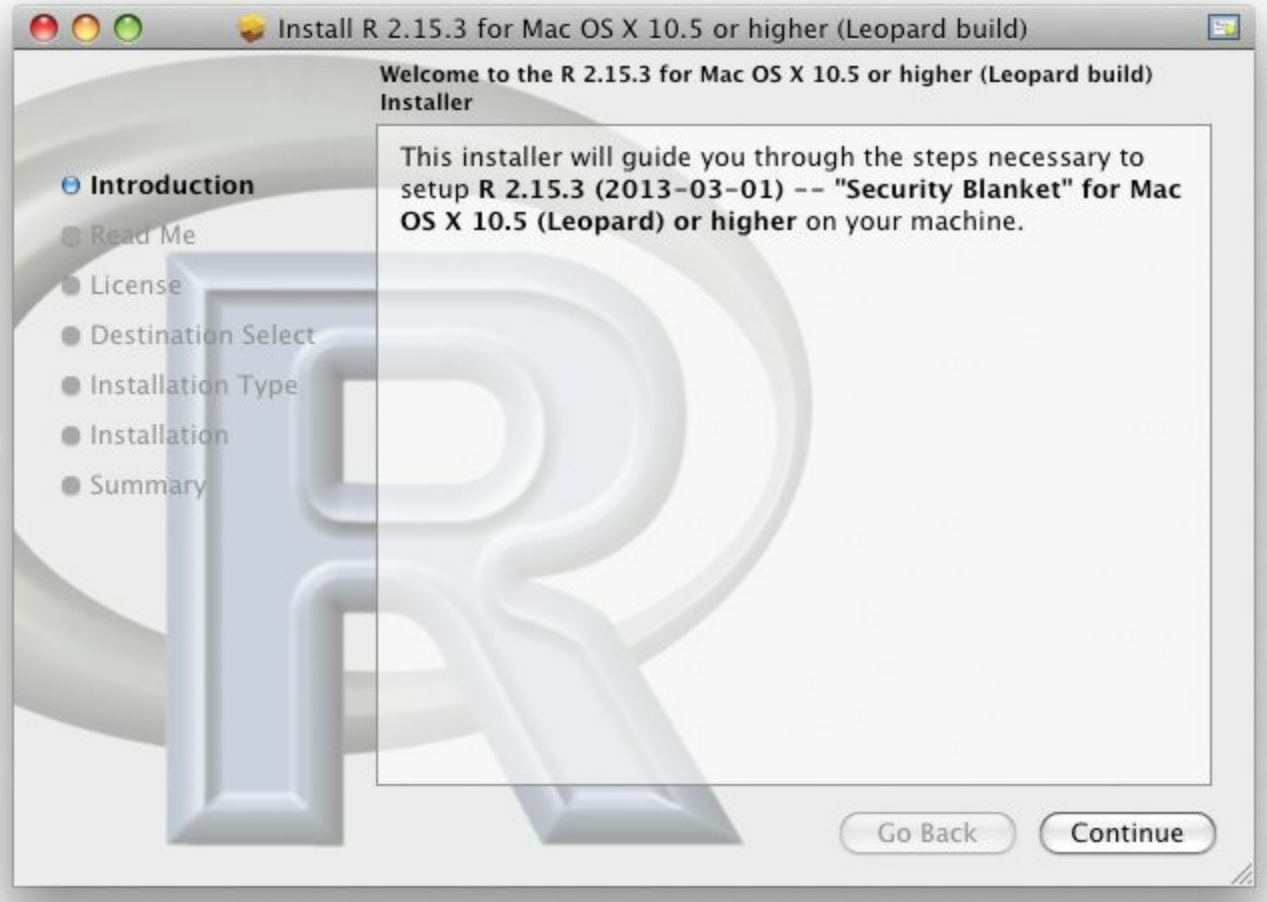

Figure 1.15 Introductory screen for installation on a Mac.

This brings up some information about the version of R being installed. There is nothing to do except click Continue, as shown in Figure 1.16.

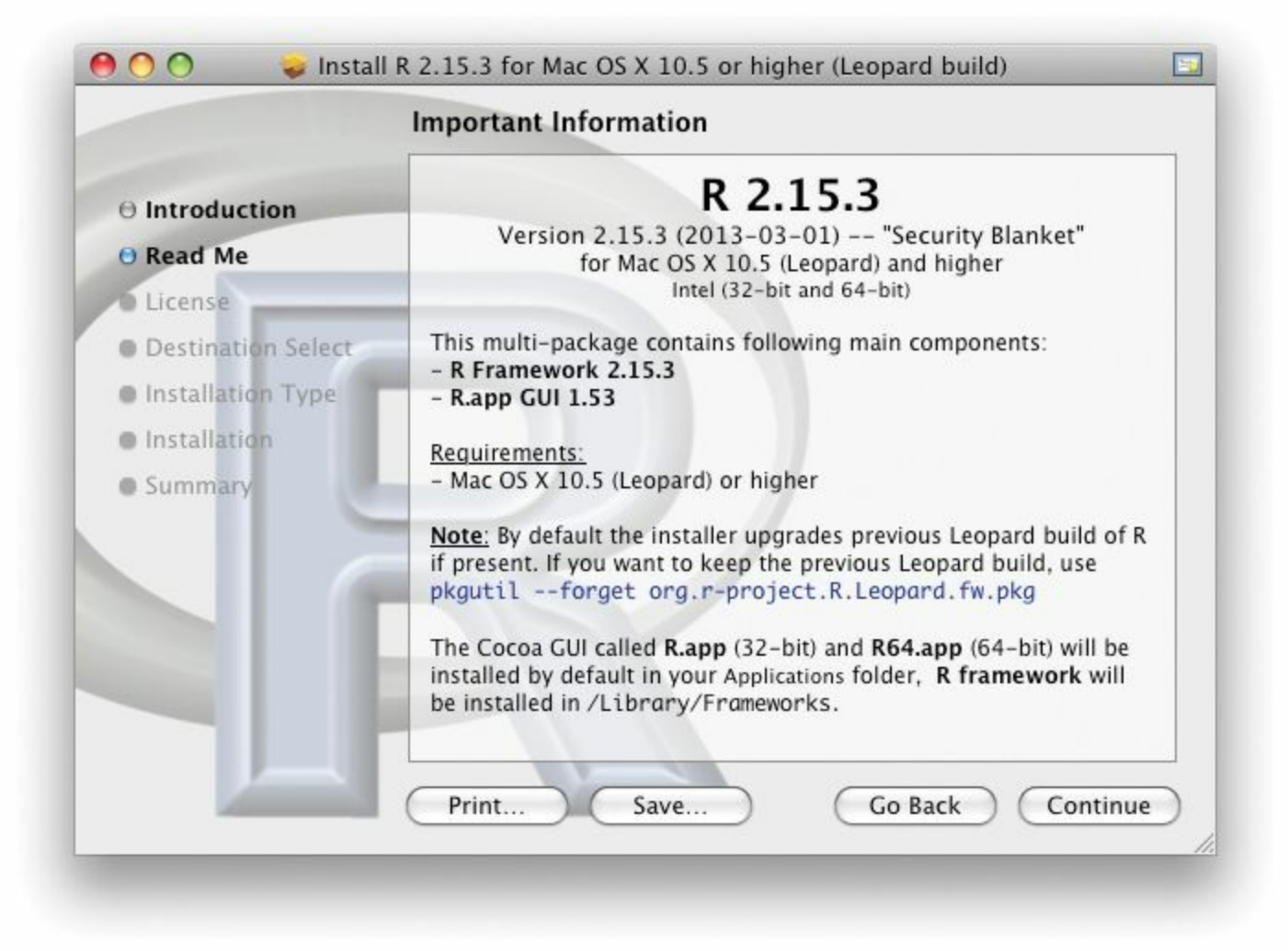

Figure 1.16 Version selection.

Then the license information is displayed, as in Figure 1.17. Click Continue to proceed, the only viable option in order to use R.

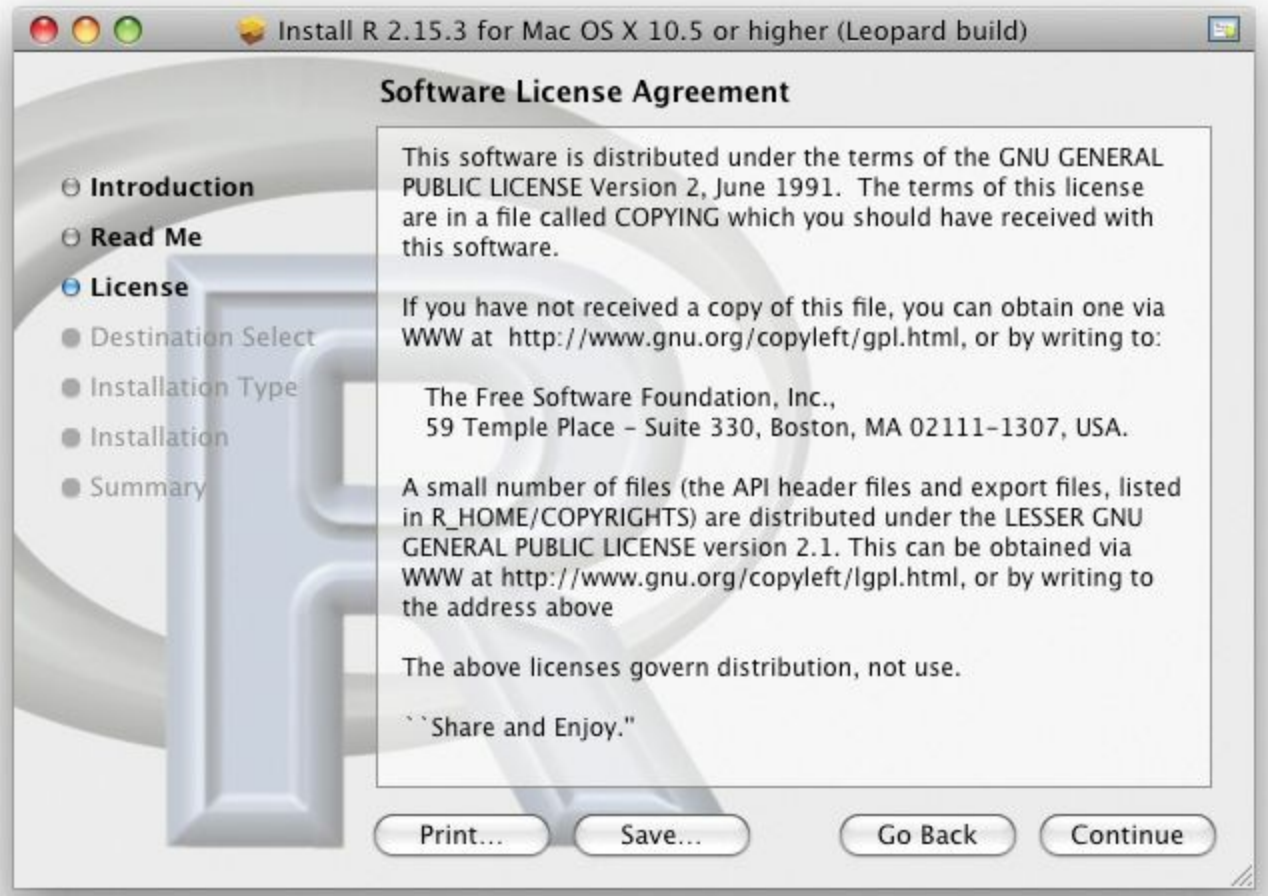

Figure 1.17 The license agreement, which must be acknowledged to use R.

Click Agree to confirm that the license is agreed to, which is mandatory to use R as is evidenced in Figure 1.18.

Figure 1.18 The license agreement must also be agreed to.

To install R for all users, click Install; otherwise, click Change Install Location to pick a different location. This is shown in Figure 1.19.

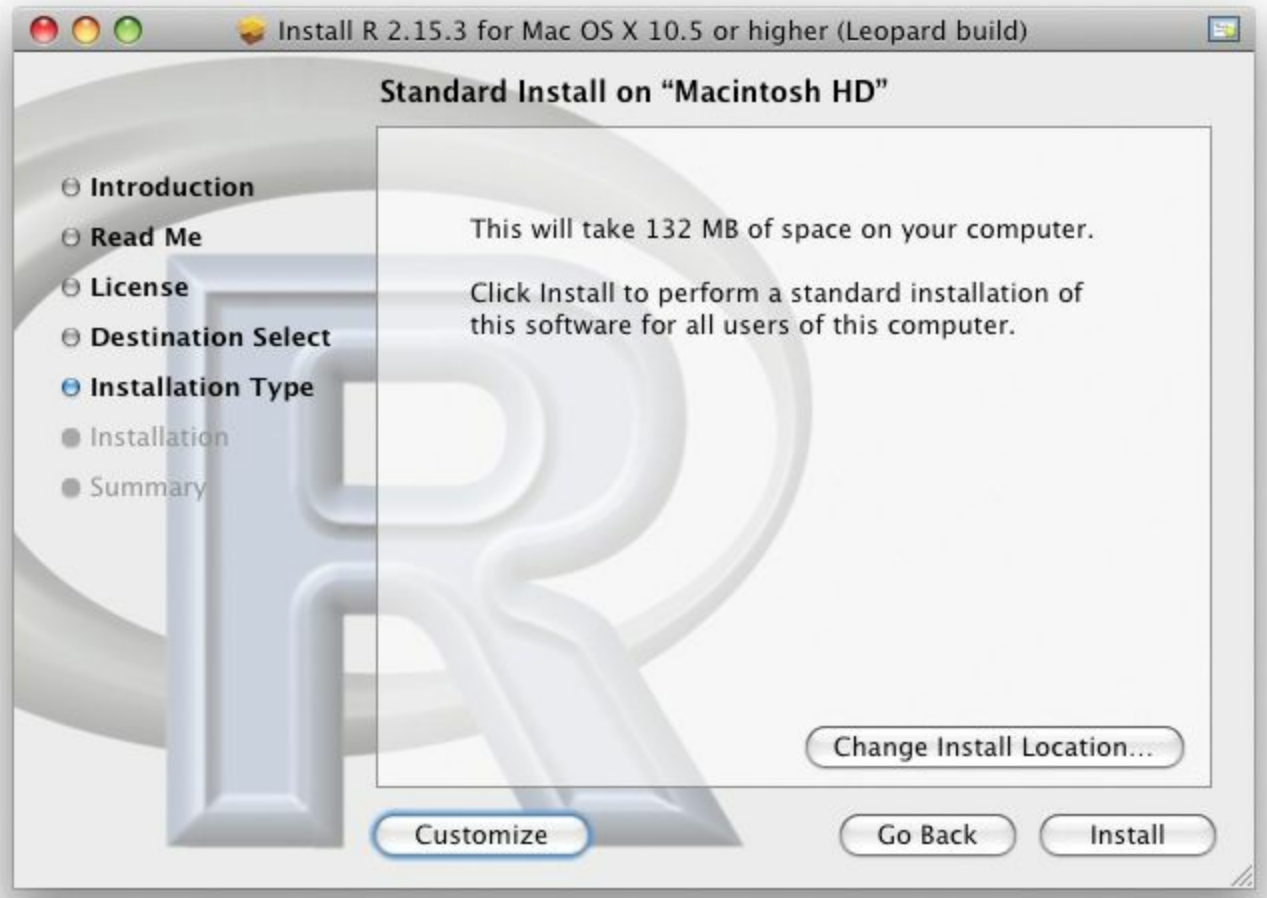

Figure 1.19 By default `R` is installed for all users, although there is the option to choose a specific location.

If prompted, enter the necessary password as shown in Figure 1.20.

Figure 1.20 The administrator password might be required for installation.

This starts the installation process, which displays a progress bar as shown in Figure 1.21.

Figure 1.21 A progress bar is displayed during installation.

When done, the installer signals success as Figure 1.22 shows. Click Close to finish the installation.

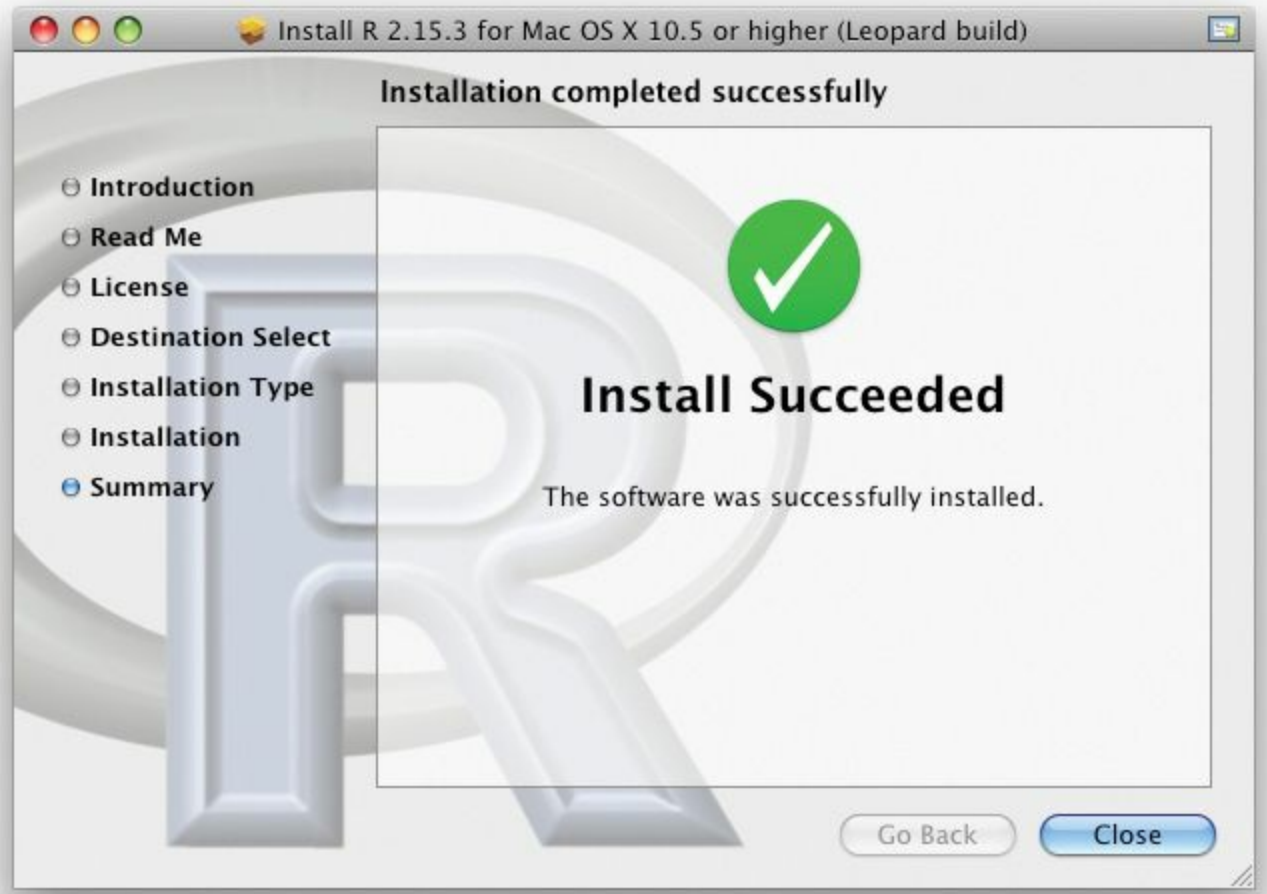

Figure 1.22 This signals a successful installation.

1.4.3. Installing on Linux

Retrieving R from its standard distribution mechanism will download, build and install R in one step.

1.5. Revolution R Community Edition

Revolution Analytics offers a community version of its build of R featuring an Integrated Development Environment based on Visual Studio and built with the Intel Matrix Kernel Library (MKL), allowing for much faster matrix computations. It is available for free at http://www.revolutionanalytics.com/products/revolution-r.php. They also offer a paid version that provides specialized algorithms to work on very large data. More information is available at http://www.revolutionanalytics.com/products/revolution-enterprise.php.

1.6. Conclusion

At this point R is fully usable and comes with a crude GUI. However, it is best to install RStudio and use its interface, which is detailed in Section 2.2. The process involves downloading and launching an installer, just as with any other program.

Chapter 2. The R Environment

Now that R is downloaded and installed, it is time to get familiar with how to use R. The basic R interface on Windows is fairly Spartan as seen in Figure 2.1. The Mac interface (Figure 2.2) has some extra features and Linux has far fewer, being just a terminal.

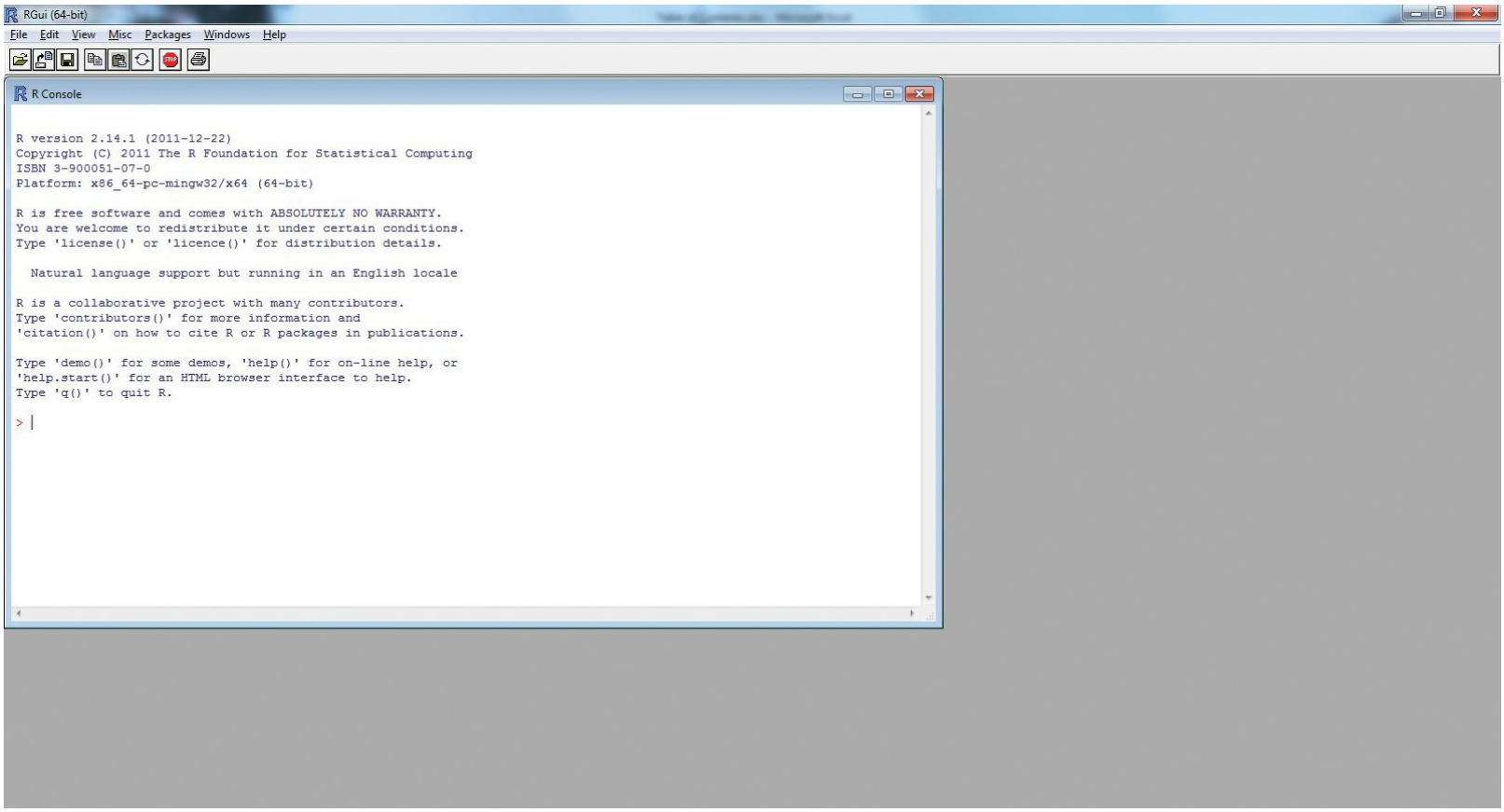

Figure 2.1 The standard R interface in Windows.

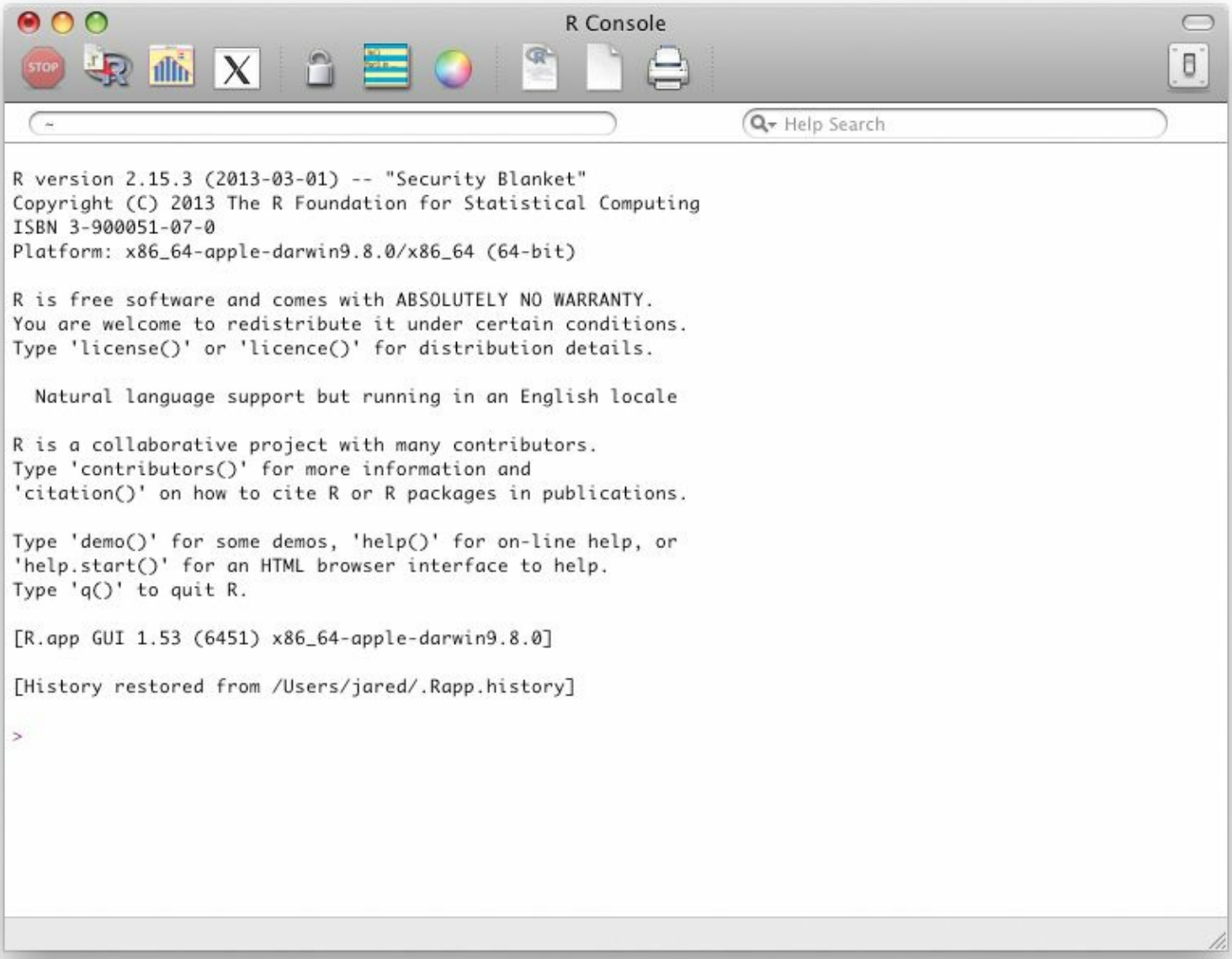

Figure 2.2 The standard R interface on Mac OS X.

Unlike other languages, R is very interactive. That is, results can be seen one command at a time. Languages such as C++ require that an entire section of code be written, compiled and run in order to see results. The state of objects and results can be seen at any point in R. This interactivity is one of the most amazing aspects of working with R.

There have been numerous Integrated Development Environments (IDEs) built for R. For the purposes of this book we will assume that RStudio is being used, which is discussed in Section 2.2.

2.1. Command Line Interface

The command line interface is what makes R so powerful, and also frustrating to learn. There have been attempts to build point-and-click interfaces for R, such as Rcmdr, but none have truly taken off. This is a testament to how typing in commands is much better than using a mouse. That might be hard to believe, especially for those coming from Excel, but over time it becomes easier and less error prone.

For instance, fitting a regression in Excel takes at least seven mouse clicks, often more: `Data >> Data Analysis >> Regression >> OK >> Input Y Range >> Input X`

Range >> OK. Then it may need to be done all over again to make one little tweak or because there are new data. Even harder is walking a colleague through those steps via email. In contrast, the same command is just one line in R, which can easily be repeated and copied and pasted. This may be hard to believe initially, but after some time the command line makes life much easier.

To run a command in R, type it into the console next to the > symbol and press the Enter key. Entries can be as simple as the number 2 or complex functions, such as those seen in Chapter 8.

To repeat a line of code, simply press the Up Arrow key and hit Enter again. All previous commands are saved and can be accessed by repeatedly using the Up and Down Arrow keys to cycle through them.

Interrupting a command is done with Esc in Windows and Mac and Ctrl-C in Linux.

Often when working on a large analysis it is good to have a file of the code used. Until recently, the most common way to handle this was to use a text editor[1] such as TextPad or UltraEdit to write code and then copy and paste it into the R console. While this worked, it was sloppy and led to a lot of switching between programs.

1. This means a programming text editor as opposed to a word processor such as Microsoft Word. A text editor preserves the structure of the text whereas word processors may add formatting that makes it unsuitable for insertion into the console.

2.2. RStudio

While there are a number of IDEs available, the best right now is RStudio, created by a team led by JJ Allaire whose previous products include ColdFusion and Windows Live Writer. It is available for Windows, Mac and Linux and looks identical in all of them. Even more impressive is the RStudio server, which runs an R instance on a Linux server and allows the user to run commands through the standard RStudio interface in a Web browser. It works with any version of R (greater than 2.11.1) including Revolution R from Revolution Analytics. RStudio has so many options that it can be a bit overwhelming. We will cover some of the most useful or frequently used features.

RStudio is highly customizable but the basic interface looks roughly like Figure 2.3. In this case the lower left pane is the R console, which can be used just like the standard R console. The upper left pane takes the place of a text editor but is far more powerful. The upper right pane holds information about the workspace, command history, files in the current folder and Git version control. The lower right pane displays plots, package information and help files.

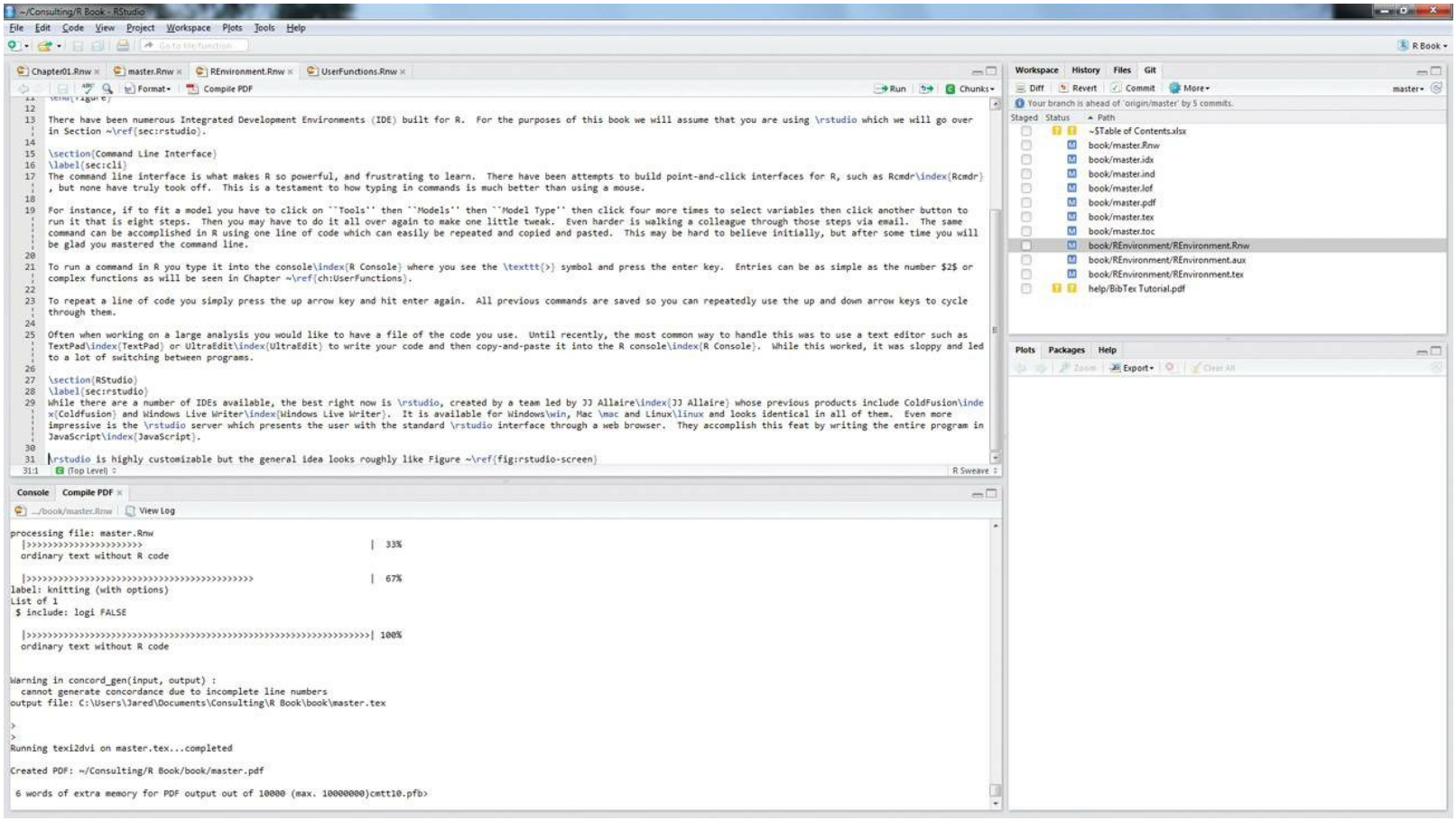

Figure 2.3 The general layout of RStudio.

There are a number of ways to send and execute commands from the editor to the console. To send one line place the cursor at the desired line and press `Ctrl+Enter` (`Command+Enter` on Mac). To insert a selection, simply highlight the selection and press `Ctrl+Enter`. To run an entire file of code, press `Ctrl+Shift+S`.

When typing code, such as an object name or function name, hitting `Tab` will autocomplete the code. If more than one object or function matches the letters typed so far, a dialog will pop up giving the matching options as shown in Figure 2.4.

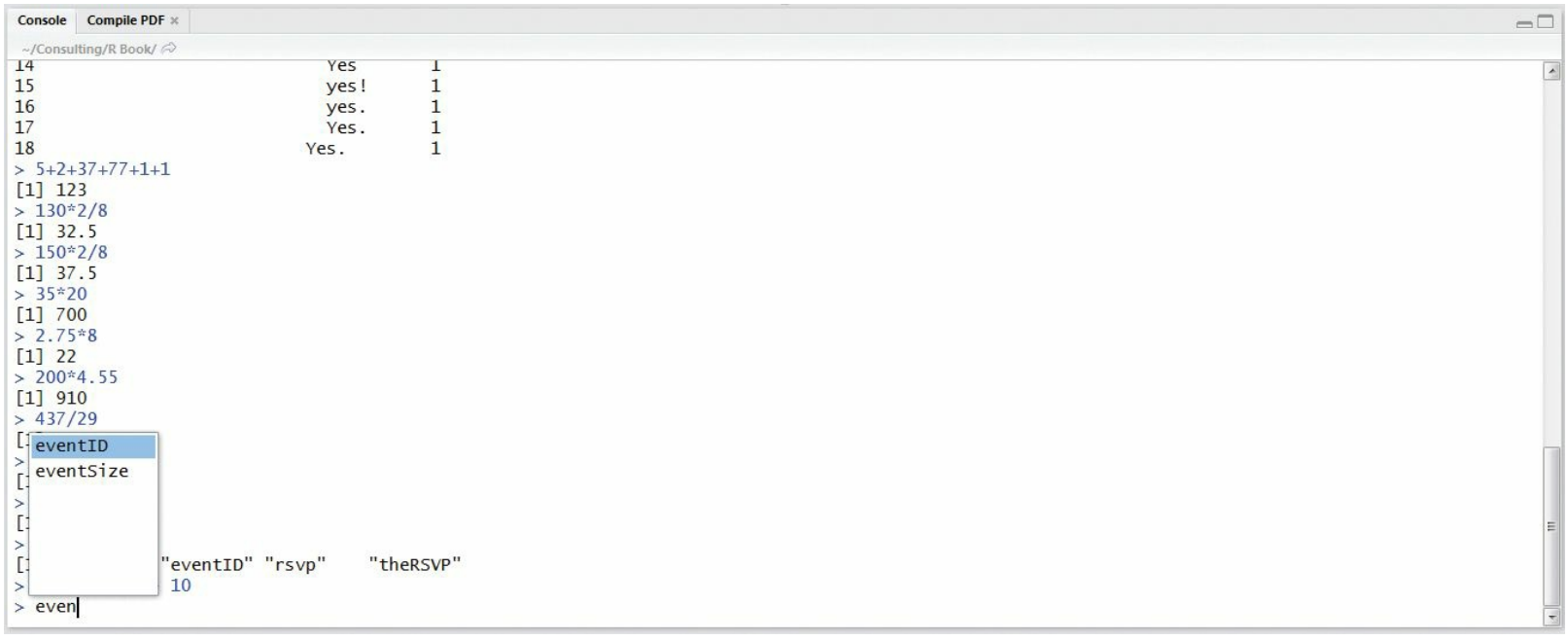

Figure 2.4 Object Name Autocomplete in RStudio.

Typing `Ctrl+1` moves the cursor to the text editor area and `Ctrl+2` moves it to the console. To move to the previous tab in the text editor, press `Ctrl+Alt+Left` in Windows, `Ctrl+PageUp` in Linux and `Ctrl+Option+Left` on Mac. To move to the next tab in the text editor, press `Ctrl+Alt+Right` in Windows, `Ctrl+PageDown` in Linux and `Ctrl+Option+Right` on Mac. For a complete list of shortcuts click `Help >> Keyboard Shortcuts`.

2.2.1. RStudio Projects

A primary feature of RStudio is projects. A project is a collection of files—and possibly data, results and graphs—that are all related to each other.[2] Each package even has its own working directory. This is a great way to keep organized.

2. This is different from an `R` session, which is all the objects and work done in `R` and kept in memory for the current usage period, which usually resets upon restarting `R`.

The simplest way to start a new project is to click `File >> New Project` as in Figure 2.5.

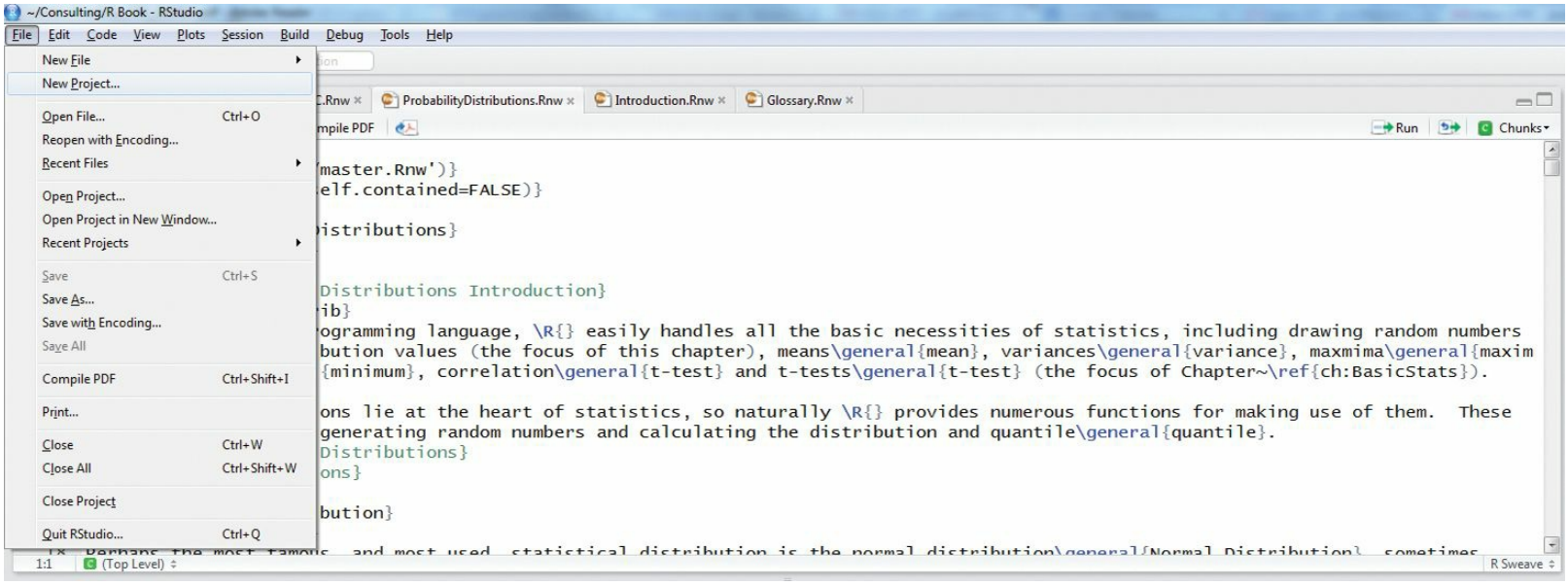

Figure 2.5 Clicking `File >> New Project` begins the project creation process.

Three options are available, shown in Figure 2.6: starting a new project in a new directory, associating a project with an existing directory or checking out a project from a version control repository such as Git or SVN. In all three cases a `.Rproj` file is put into the resulting directory and keeps track of the project.

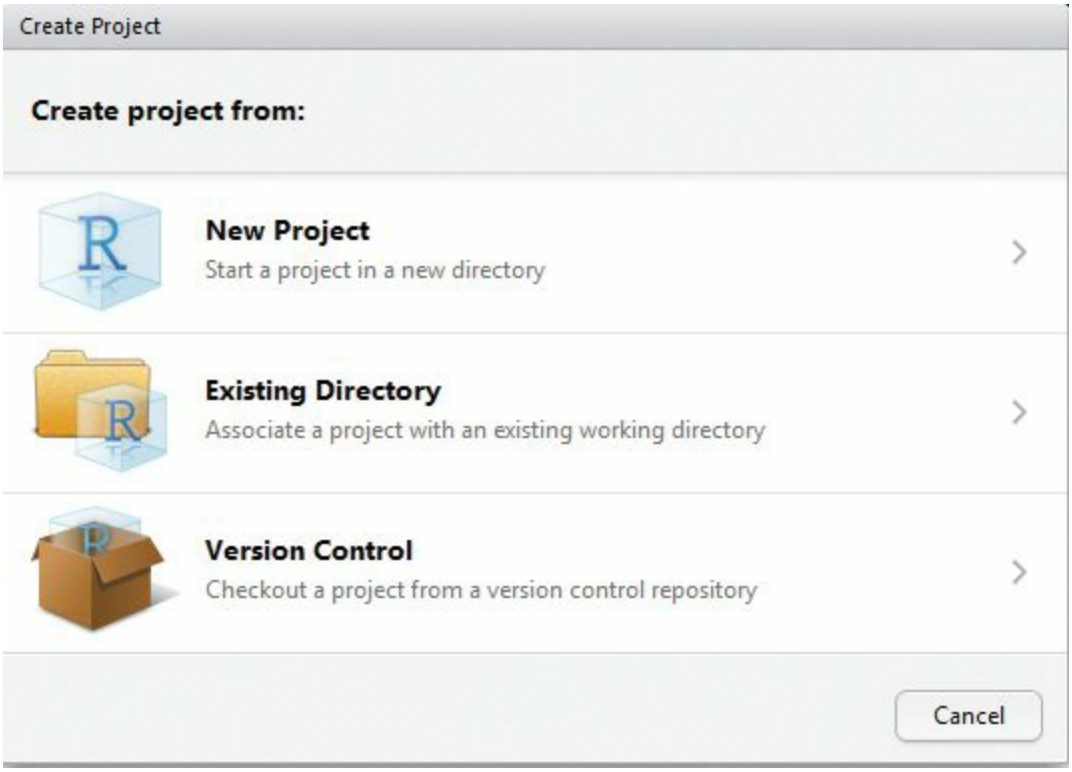

Figure 2.6 Three options are available to start a new project: a new directory, associating a project with an existing directory or checking out a project from a version control repository.

Choosing to create a new directory brings up a dialog, shown in Figure 2.7, that requests a project name and where to create a new directory.

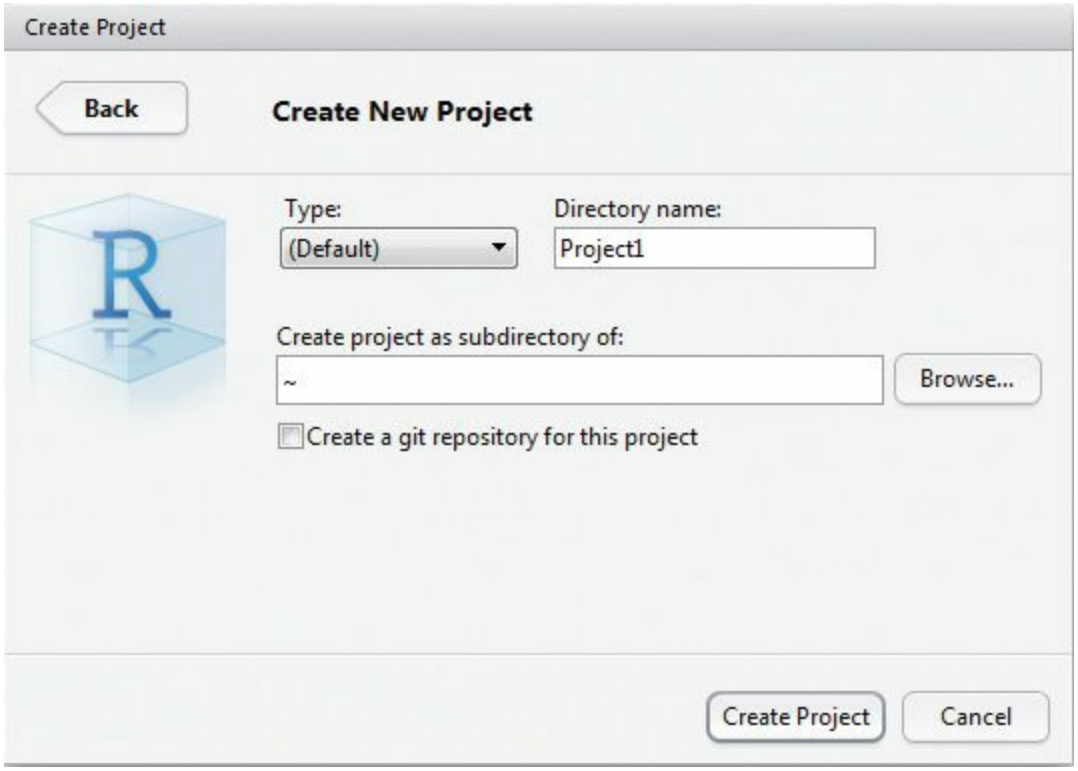

Figure 2.7 Dialog to choose the location of a new project directory.

Choosing an existing directory asks for the name of the directory, seen in Figure 2.8.

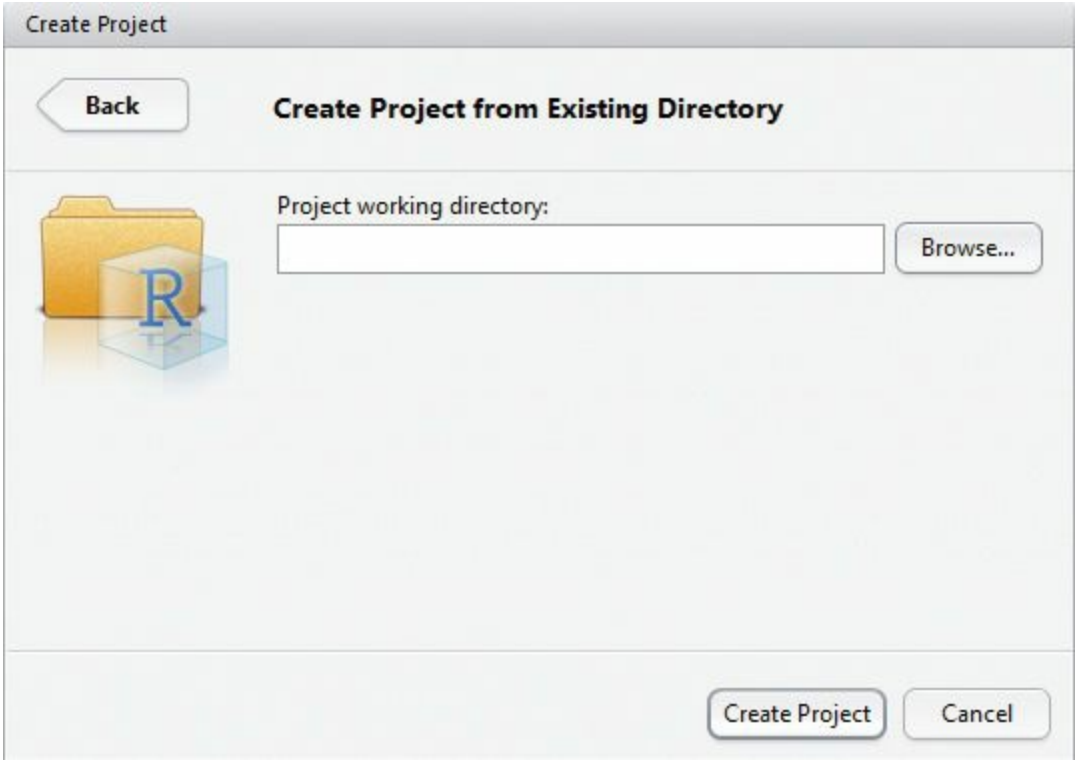

Figure 2.8 Dialog to choose an existing directory in which to start a project.

Choosing to use version control (we prefer Git) firsts asks whether to use Git or SVN as in Figure 2.9.

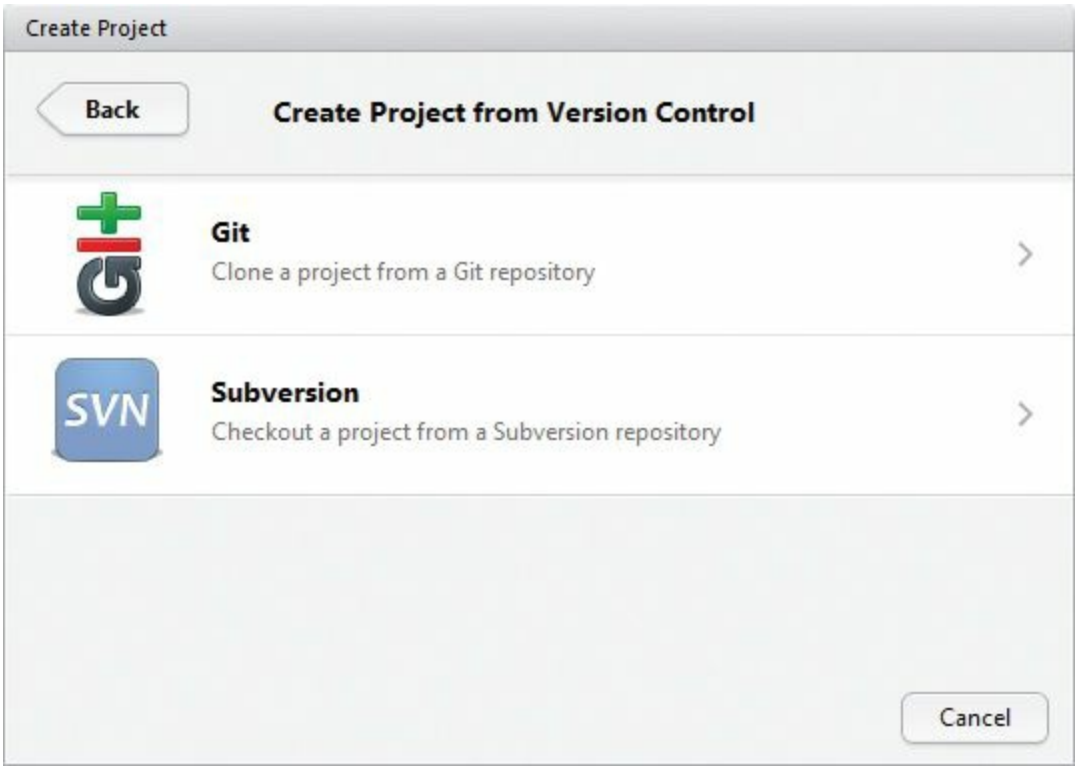

Figure 2.9 Here is the option to choose which type of repository to start a new project from.

Selecting Git asks for a repository URL, such as `git@github.com:jaredlander/coefplot.git`, which will then fill in the project directory name, as shown in Figure 2.10. As with creating a new directory, this will ask where to put this new directory.

Figure 2.10 Enter the URL for a Git repository, as well as the folder where this should be cloned to.

2.2.2. RStudio Tools

RStudio is highly customizable with a lot of options. Most are contained in the Options dialog accessed by clicking `Tools >> Options`, as seen in Figure 2.11.

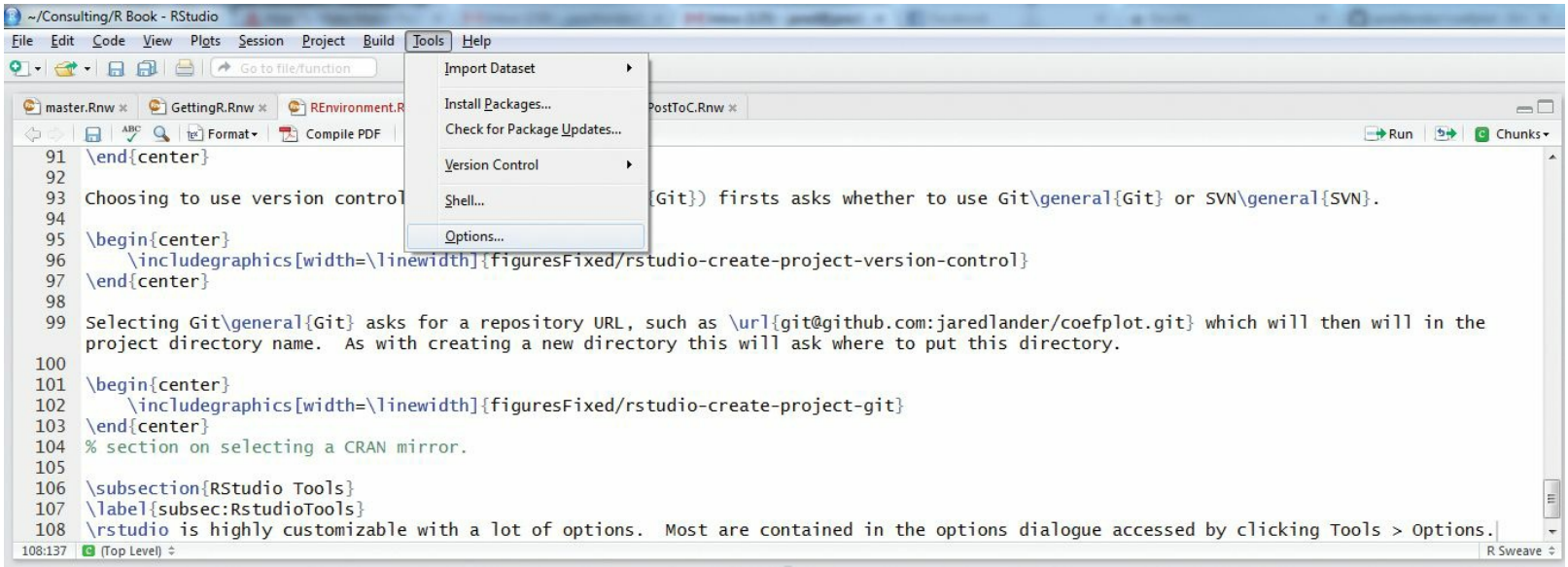

Figure 2.11 Clicking `Tools >> Options` brings up RStudio options.

First are the General options, shown in Figure 2.12. There is a control for selecting which version of `R` to use. This is a powerful tool when a computer has a number of versions of `R`. However, RStudio must be restarted after changing the `R` version. In the future, RStudio is slated to offer the ability to set different versions of `R` for each project. It is also a good idea to not restore or save `.RData` files on startup and exiting.[3]

3. `RData` files are a convenient way of saving and sharing `R` objects and are discussed in Section 6.5.

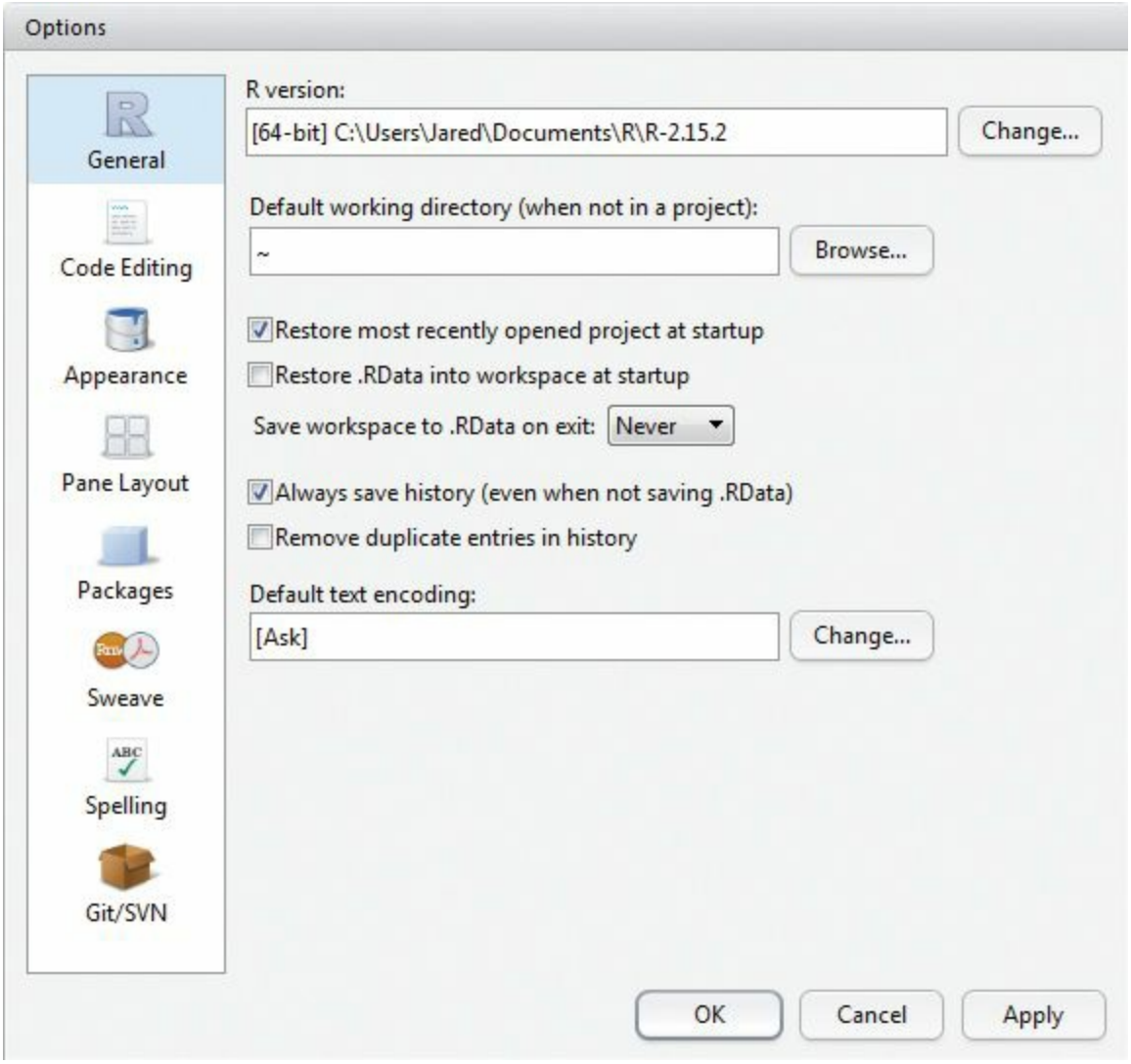

Figure 2.12 General options in RStudio.

The Code Editing options, shown in Figure 2.13, control the way code is entered and displayed in the text editor. It is generally considered good practice to replace tabs with spaces, either two or four. Some hard-core programmers will appreciate vim mode. As of now there is no Emacs mode.

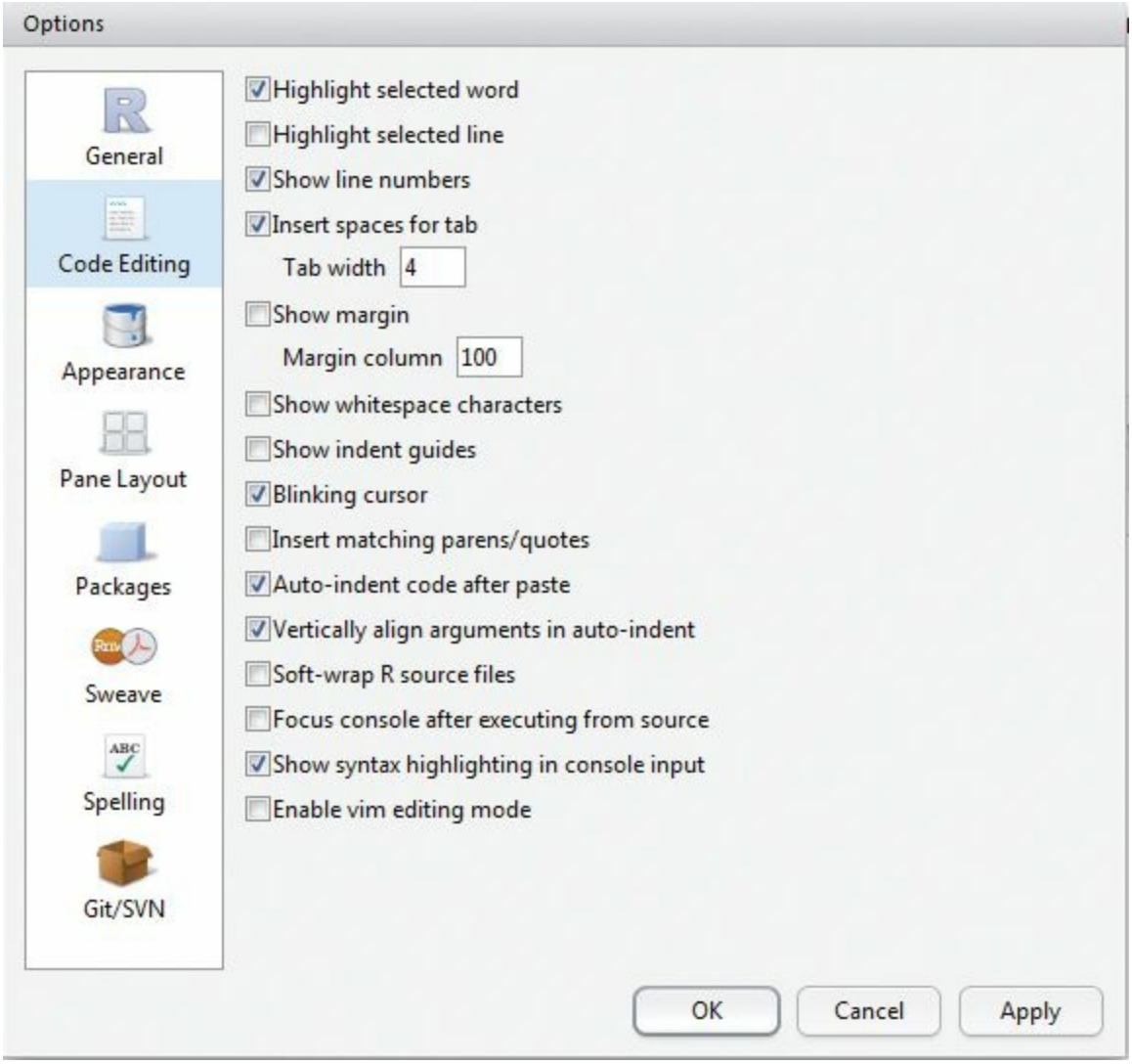

Figure 2.13 Options for customizing the code editing pane.

Appearance options, shown in Figure 2.14, change the way code looks, aesthetically. The font, size and color of the background and text can all be customized here.

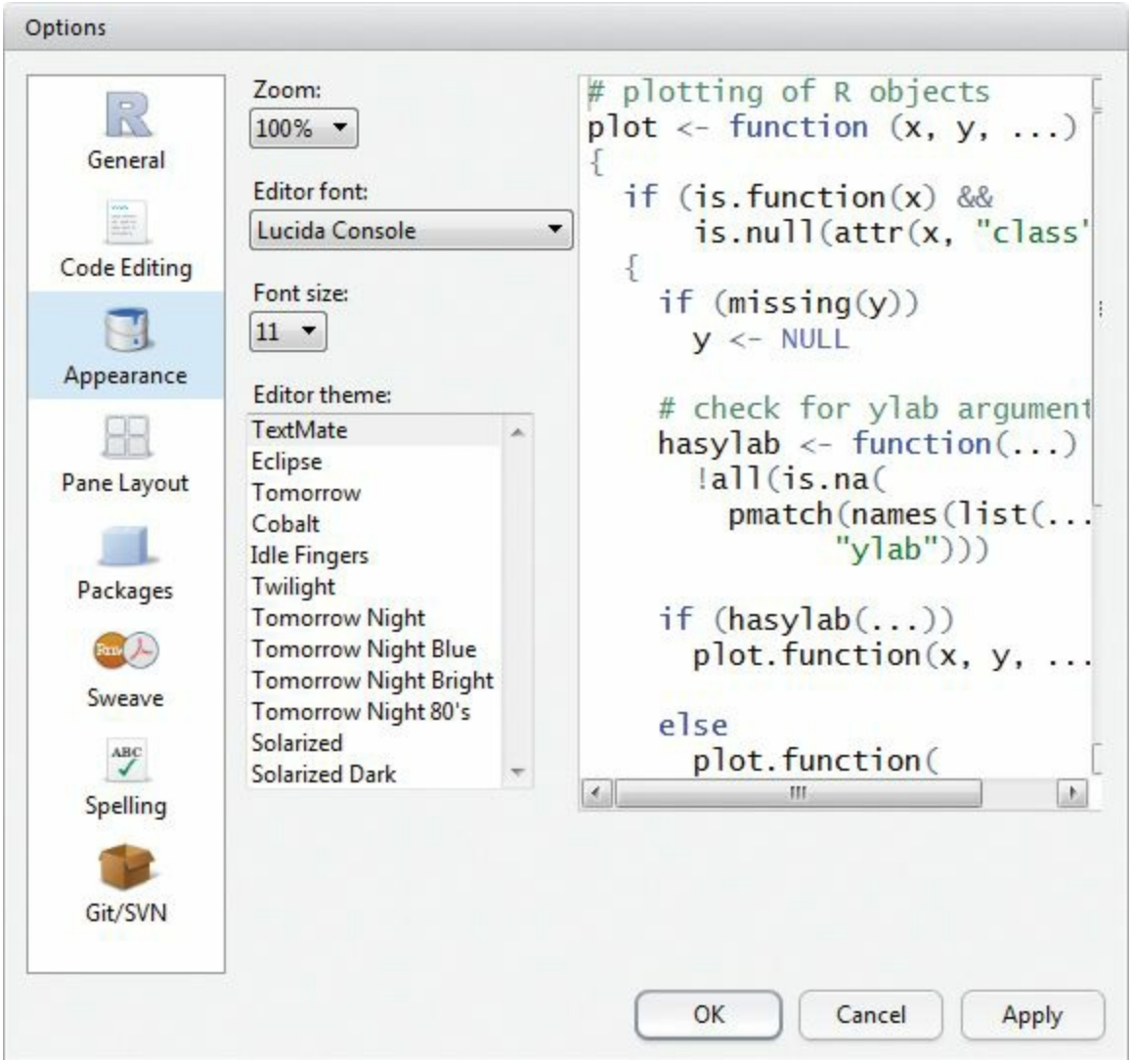

Figure 2.14 Options for code appearance.

The Pane Layout options, shown in Figure 2.15, simply rearrange the panes that make up RStudio.

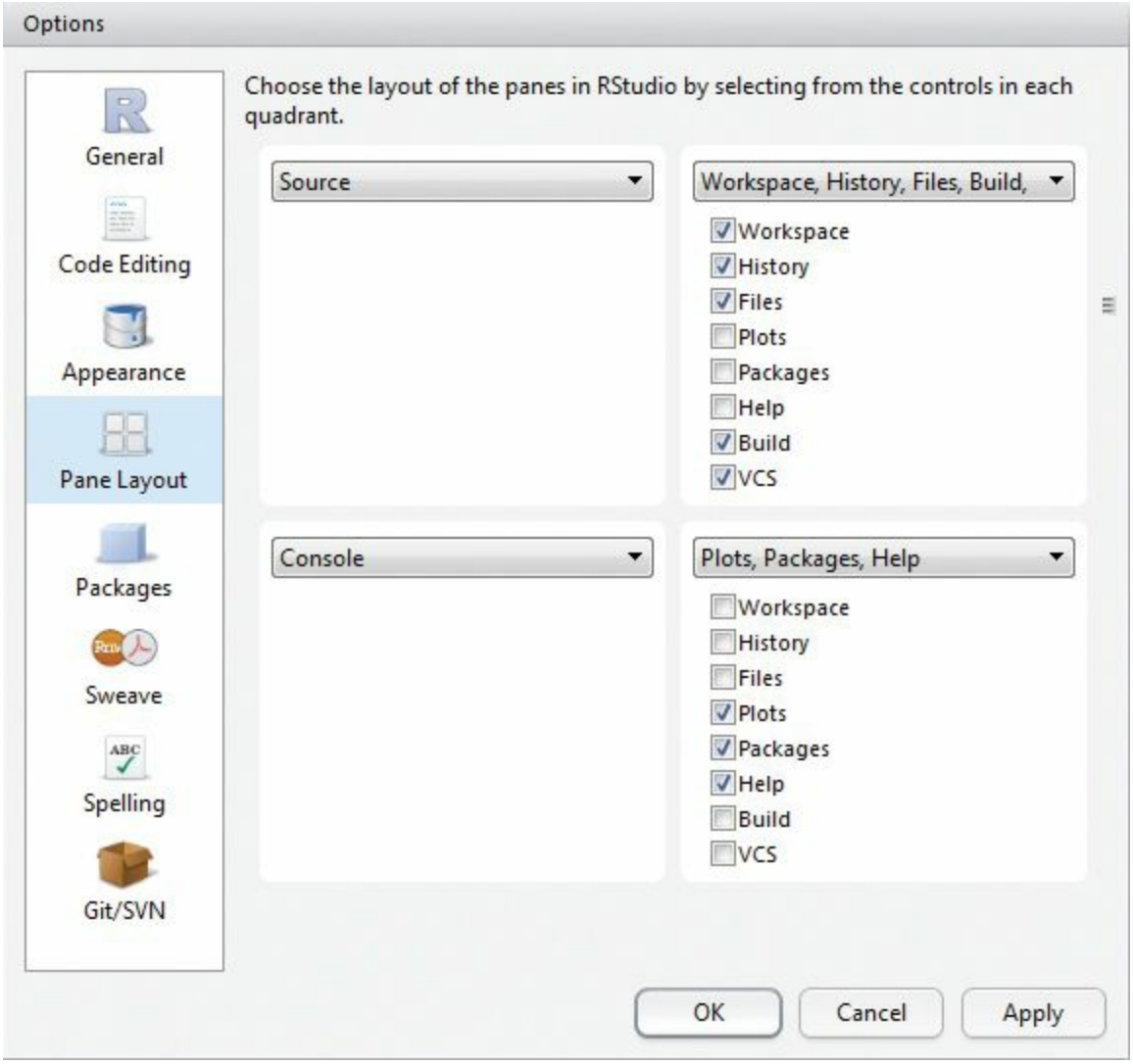

Figure 2.15 These options control the placement of the various panes in RStudio.

The Packages options, shown in Figure 2.16, set options regarding packages, although the most important is the CRAN mirror. While this is changeable from the console, this is the default setting. It is best to pick the mirror that is geographically the closest.

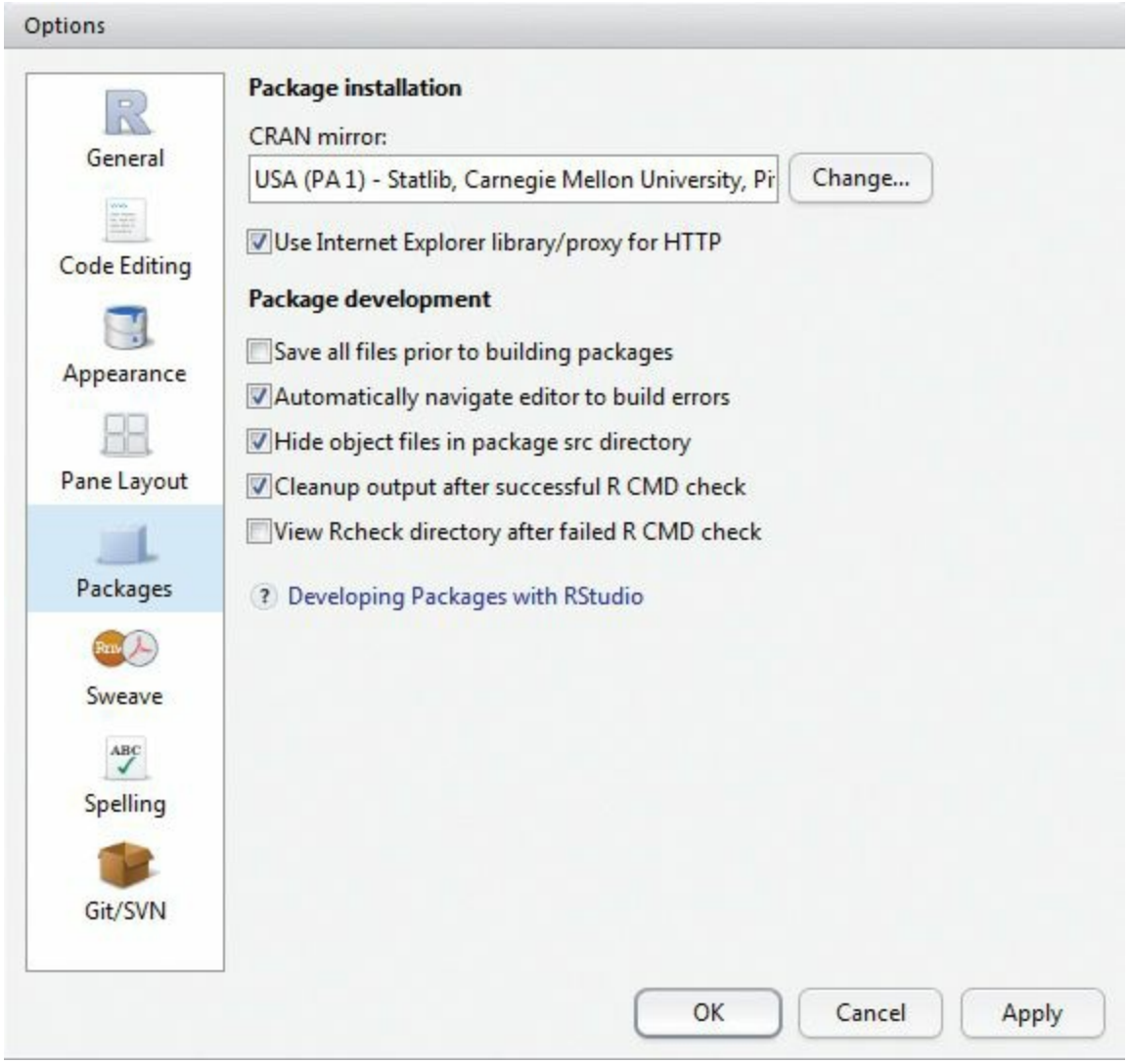

Figure 2.16 Options related to packages. The most important is the CRAN mirror selection.

Sweave, Figure 2.17, may be a bit misnamed, as this is where to choose between using Sweave or `knitr`. Both are used for the generation of PDF documents with `knitr` also enabling the creation of `HTML` documents. `knitr`, detailed in Chapter 23, is by far the better option, although it must be installed first, which is explained in Section 3.1. This is also where the PDF viewer is selected.

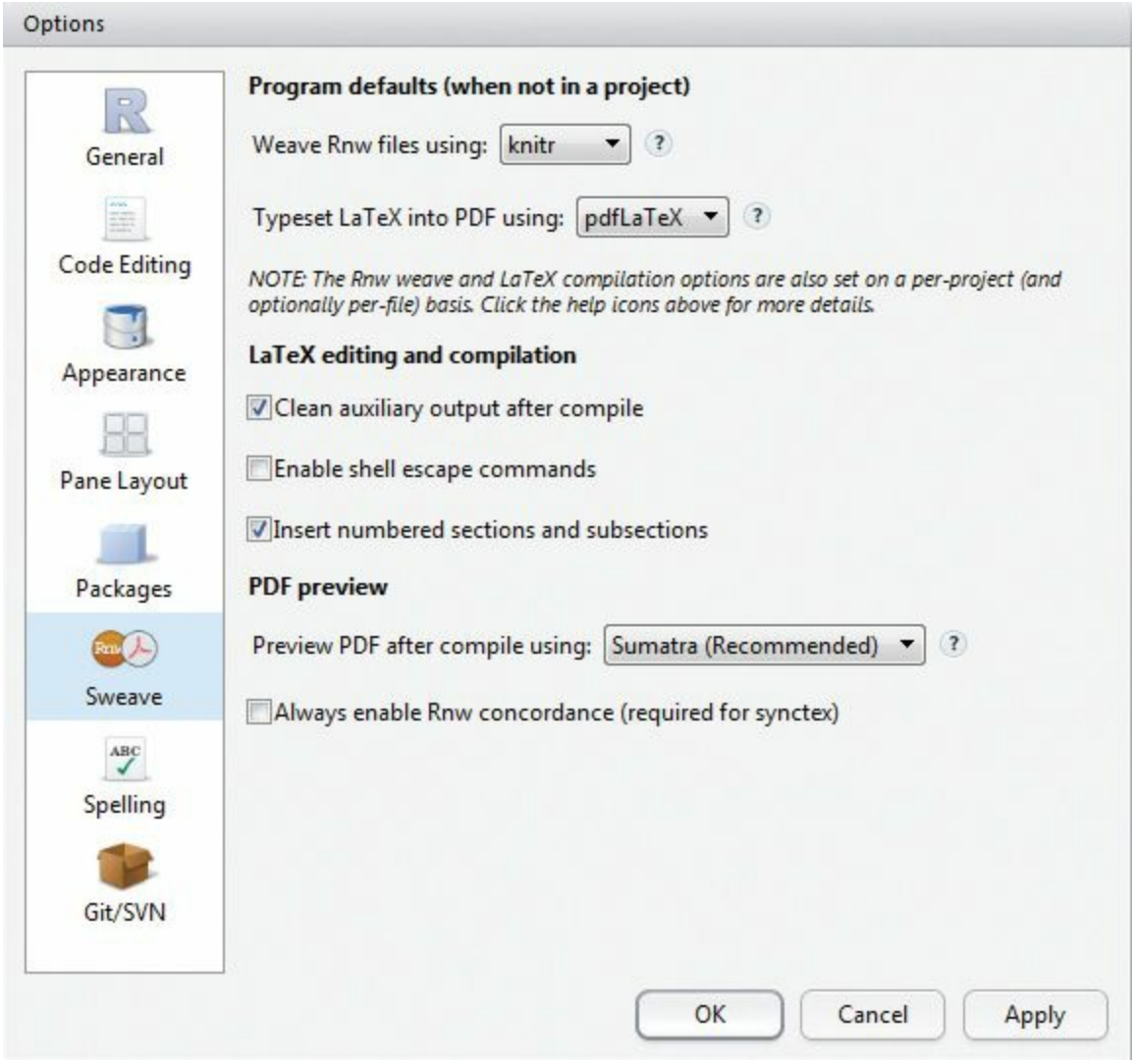

Figure 2.17 This is where to choose whether to use Sweave or `knitr` and select the PDF viewer.

RStudio contains a spelling checker for writing LATEX and Markdown documents (using `knitr`, preferably), which is controlled from the Spelling options, Figure 2.18. Not much needs to be set here.

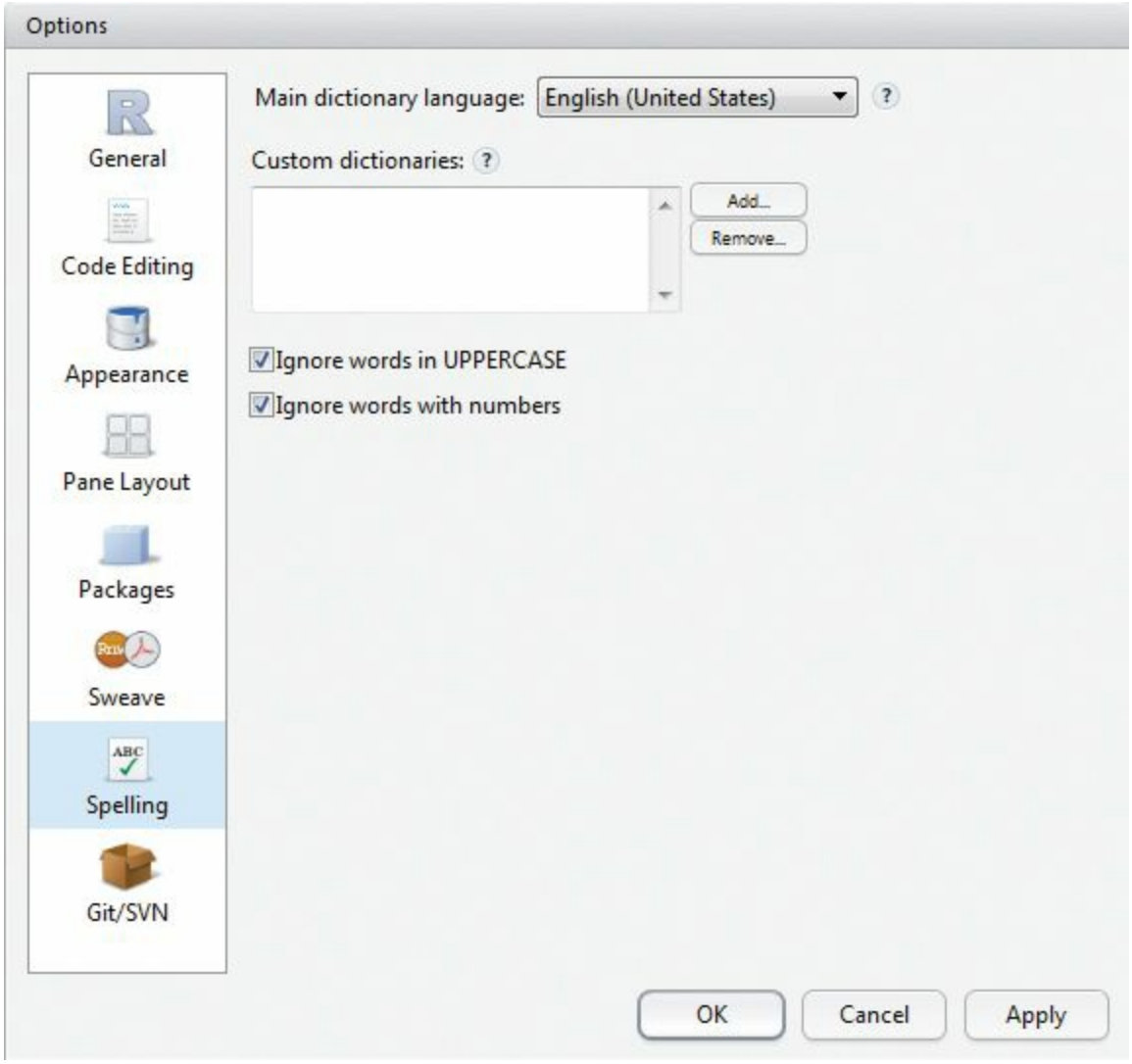

Figure 2.18 These are the options for the spelling check dictionary, which allows language selection and the custom dictionaries.

The last option, Git/SVN, Figure 2.19, indicates where the executables for Git and SVN exist. This needs to be set only once but is necessary for version control.

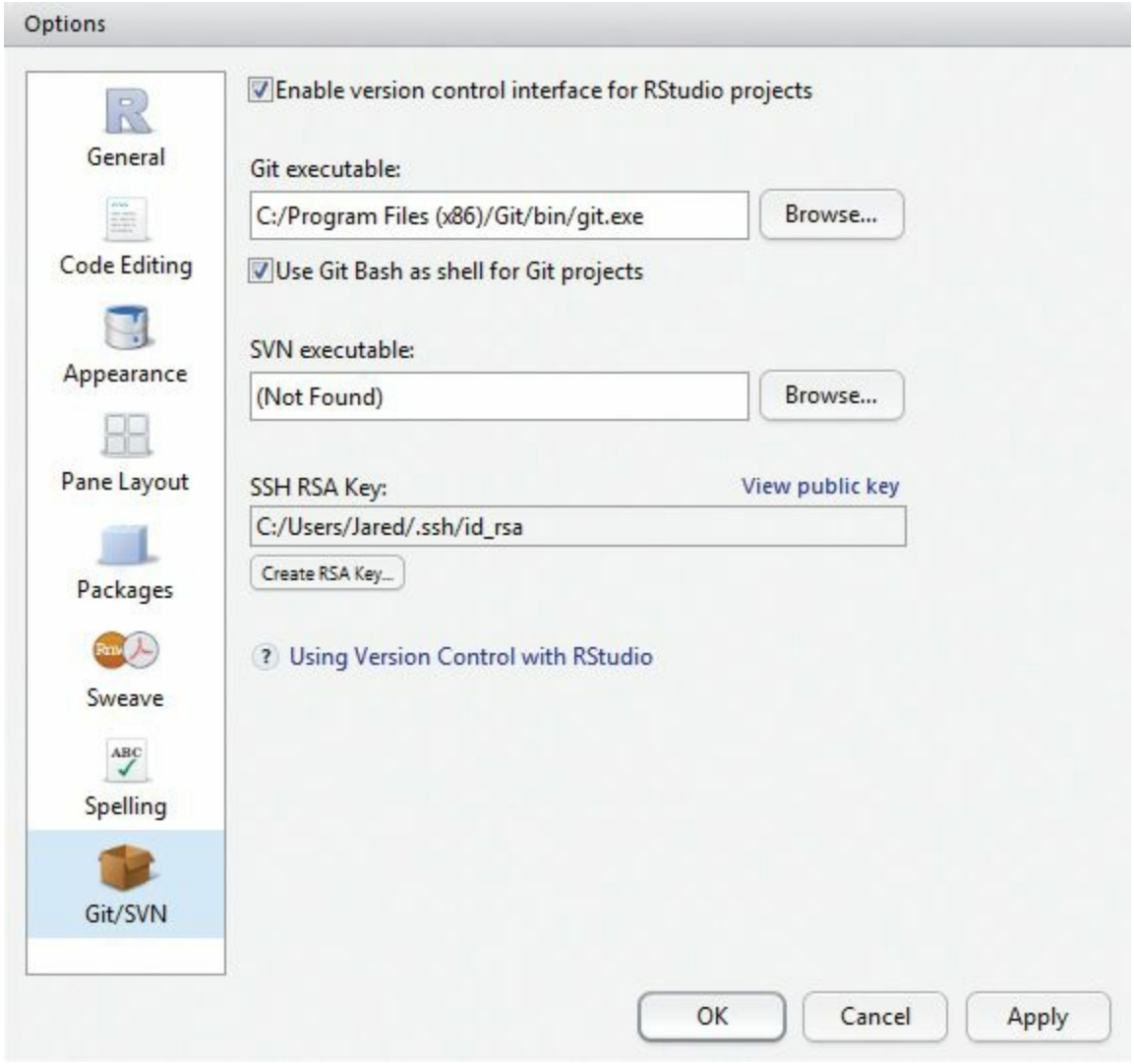

Figure 2.19 This is where to set the location of Git and SVN executables so they can be used by RStudio.

2.2.3. Git Integration

Using version control is a great idea for many reasons. First and foremost it provides snapshots of code at different points in time and can easily revert to those snapshots. Ancillary benefits include having a backup of the code and the ability to easily transfer the code between computers with little effort.

While SVN used to be the gold standard in version control it has since been superseded by Git, so that will be our focus. After associating a project with a Git repository[4] RStudio has a pane for Git like the one shown in Figure 2.20.

4. A Git account should be set up with either GitHub (https://github.com/) or Bitbucket (https://bitbucket.org/) beforehand.

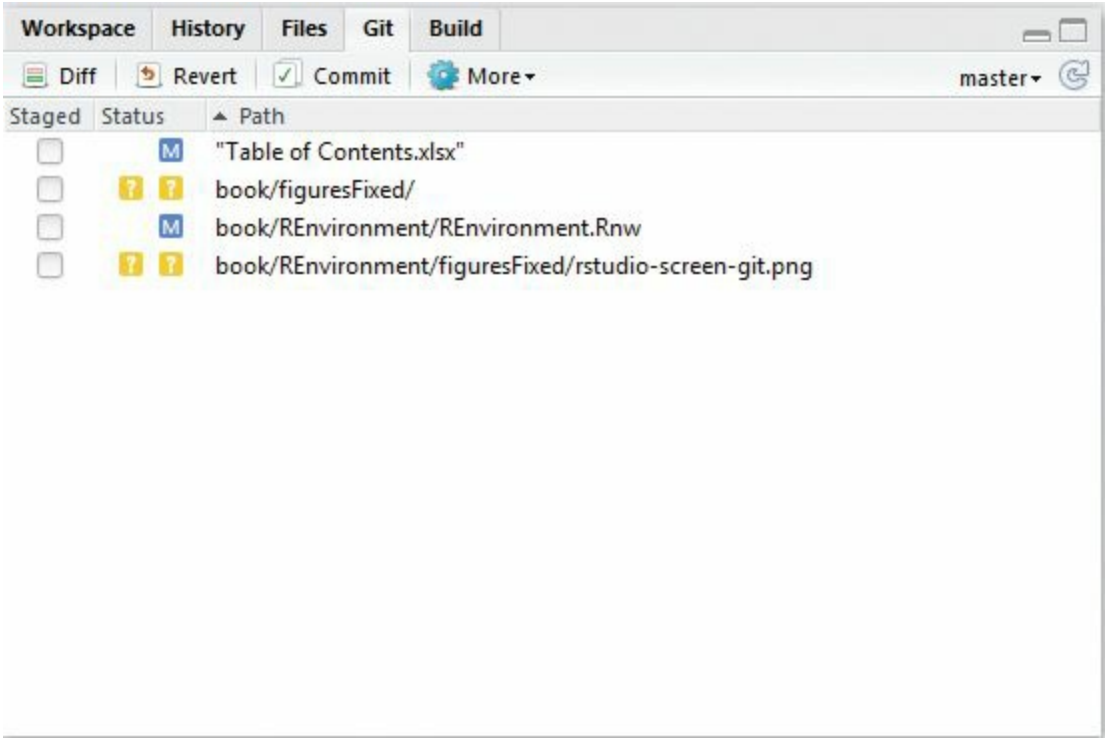

Figure 2.20 The Git pane shows the Git status of files under version control. A blue square with a white M indicates a file has been changed and needs to be committed. A yellow square with a white question mark indicates a new file that is not being tracked by Git.

The main functionality is committing changes, pushing them to the server and pulling changes made by other users. Clicking the Commit button brings up a dialog, Figure 2.21, which displays files that have been modified, or new files. Clicking on one of these files displays the changes; deletions are colored pink and additions are colored green. There is also a space to write a message describing the commit.

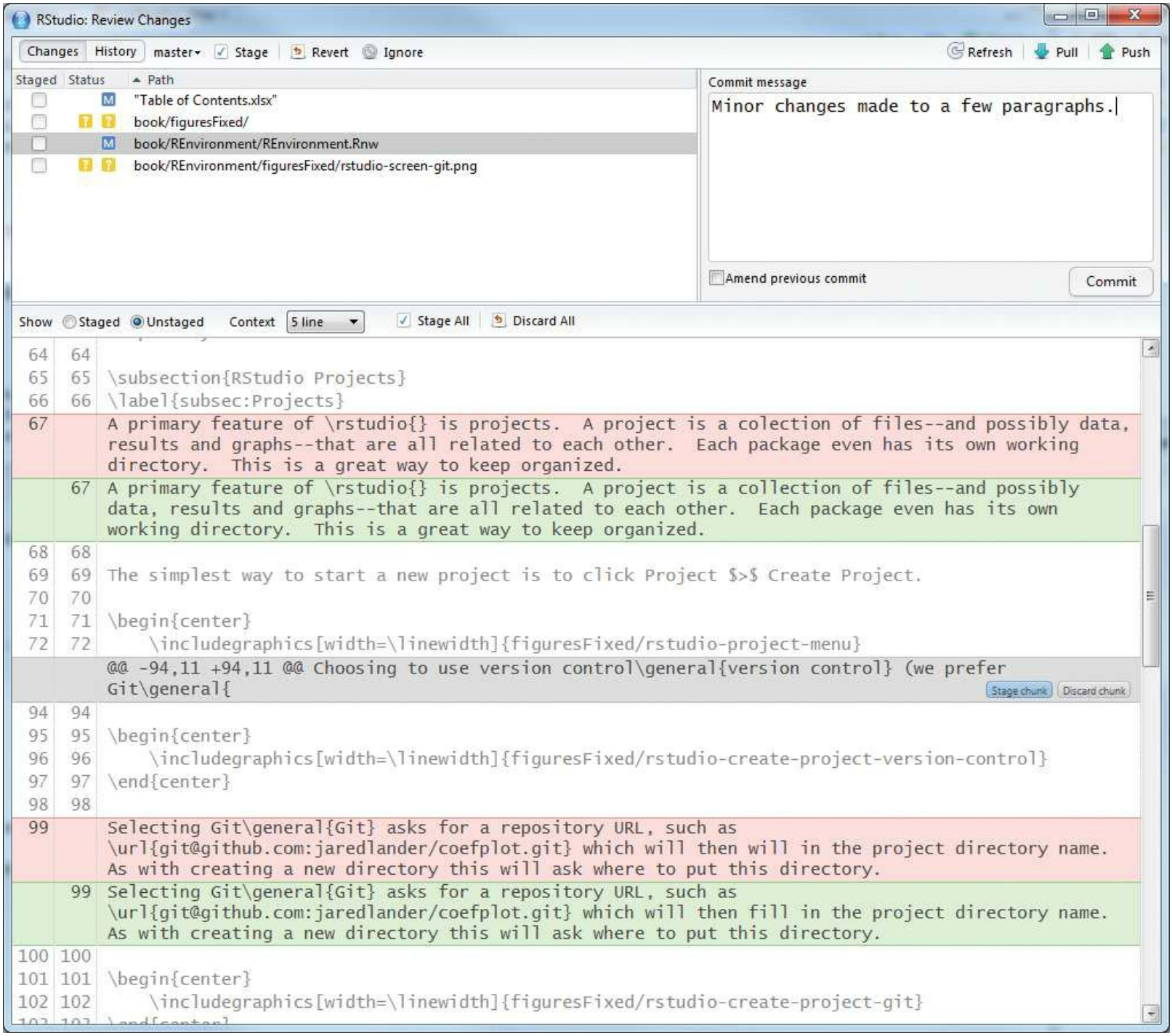

Figure 2.21 This displays files and the changes made to the files, with green being additions and pink being deletions. The upper right contains a space for writing commit messages.

Clicking Commit will stage the changes and clicking Push will send them to the server.

2.3. Revolution Analytics RPE

Revolution Analytics provides an IDE based on Visual Studio called the R Productivity Environment (RPE). The greatest benefit of the RPE is the visual debugger. If this feature is not needed,[5] we recommend using Revolution with RStudio as the front-end, which can be set in the General options detailed in Section 2.2.2.

5. The latest version of RStudio now also offers a visual debugger.

2.4. Conclusion

R's usability has greatly improved over the past few years, mainly thanks to Revolution Analytics'

RPE and RStudio. Using an IDE can greatly improve proficiency, and change working with R from merely tolerable to actually enjoyable.[6] RStudio's code completion, text editor, Git integration and projects are indispensable for a good programming work flow.

6. One of our students relayed that he preferred Matlab to R until he used RStudio.

Chapter 3. R Packages

Perhaps the biggest reason for R's phenomenally ascendant popularity is its collection of user-contributed packages. As of mid-September 2013, there were 4,845 packages available on CRAN[1], written by an estimated 2,000 different people. Odds are good that if a statistical technique exists, it has been written in R and contributed to CRAN. Not only are there an incredibly large number of packages, many are written by the authorities in the field such as Andrew Gelman, Trevor Hastie, Dirk Eddelbuettel and Hadley Wickham.

1. http://cran.r-project.org/web/packages/

A package is essentially a library of prewritten code designed to accomplish some task or a collection of tasks. The `survival` package is used for survival analysis, `ggplot2` is used for plotting and `sp` is for dealing with spatial data.

It is important to remember that not all packages are of the same quality. Some are built to be very robust and are well-maintained, while others are built with good intentions but can fail with unforeseen errors and others still are just plain poor. Even with the best packages, it is important to remember that most were written by statisticians for statisticians, so they may differ from what a computer engineer would expect.

This book will not attempt to provide an exhaustive list of good packages to use because that is constantly changing. However, there are some packages that are so pervasive that they will be used in this book as if they were part of base R. Some of these are `ggplot2`, `reshape2` and `plyr` by Hadley Wickham; `glmnet` by Trevor Hastie, Robert Tibshirani and Jerome Friedman; `Rcpp` by Dirk Eddelbuettel; and `knitr` by Yihui Xie. We have written a package on CRAN, `coefplot`, with more to follow.

3.1. Installing Packages

As with many tasks in R, there are multiple ways to install packages. The simplest is to install them using the GUI provided by RStudio and shown in Figure 3.1. Access the Packages pane shown in this figure either by clicking its tab or by pressing `Ctrl+7` on the keyboard.

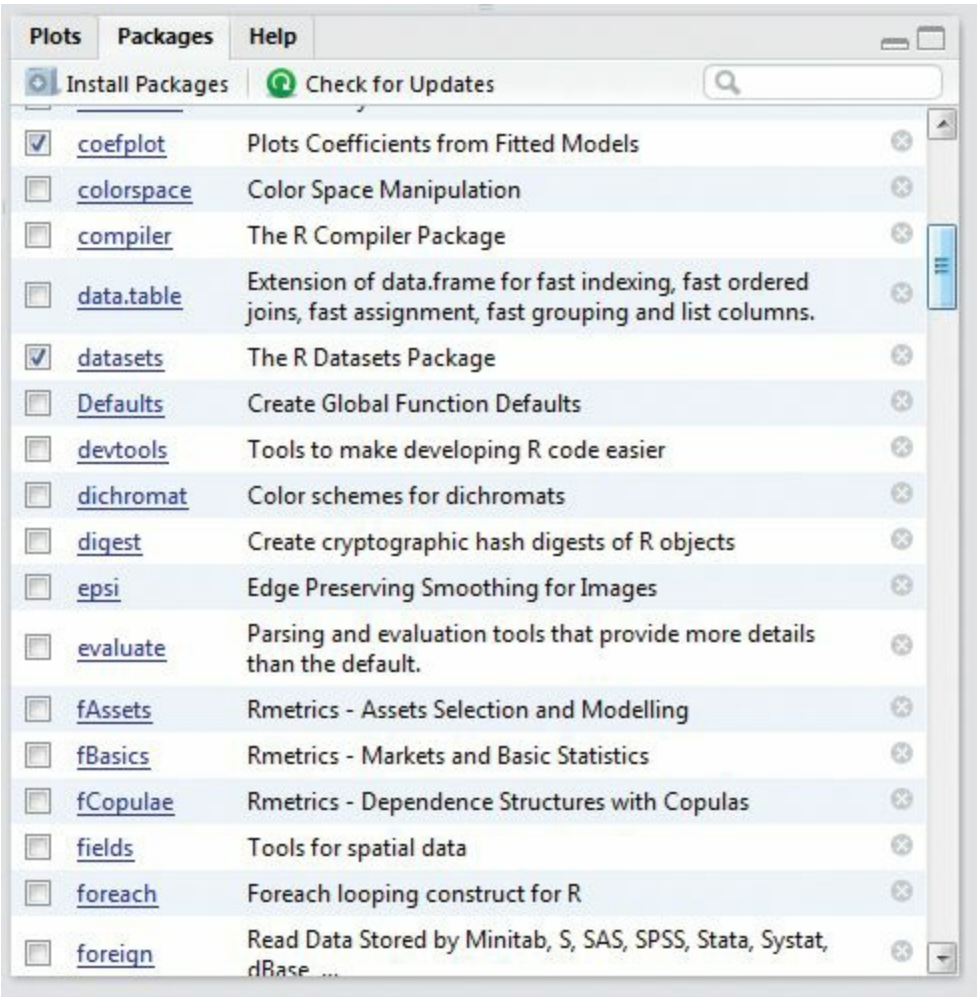

Figure 3.1 RStudio's Packages pane.

In the upper-left corner, click the Install Packages button to bring up the dialog in Figure 3.2.

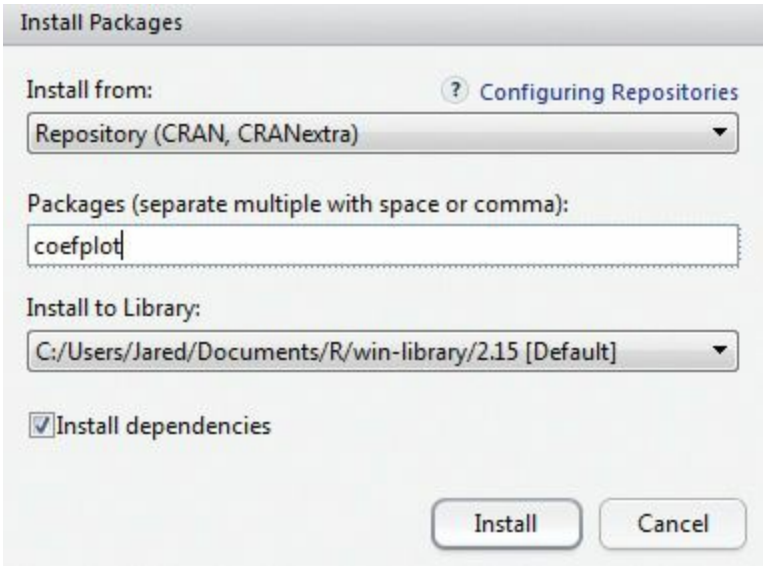

Figure 3.2 RStudio's package installation dialog.

From here simply type the name of a package (RStudio has a nice autocomplete feature for this) and click Install. Multiple packages can be specified, separated by commas. This downloads and installs the desired package, which is then available for use. Selecting the Install dependencies checkbox will automatically download and install all packages that the desired package requires to work. For example, our `coefplot` package depends on `ggplot2`, `plyr`, `useful`, `stringr` and `reshape2`, and each of those may have further dependencies.

An alternative is to type a very simple command into the console:

Click here to view code image

```
> install.packages("coefplot")
```

This will accomplish the same thing as working in the GUI.

There has been a movement recently to install packages directly from GitHub or BitBucket repositories, especially to get the development versions of packages. This can be accomplished using `devtools`.

Click here to view code image

```
> require(devtools)
> install_github(repo = "coefplot", username = "jaredlander")
```

If the package being installed from a repository contains source code for a compiled language—generally C++ or FORTRAN—then the proper compilers must be installed. More information is in Section 24.6.

Sometimes there is a need to install a package from a local file, either a zip of a prebuilt package or a tar.gz of package code. This can be done using the installation dialog mentioned before but switching the Install from: option to Package Archive File as shown in Figure 3.3. Then browse to the file and install. Note that this will not install dependencies, and if they are not present the installation will fail. Be sure to install dependencies first.

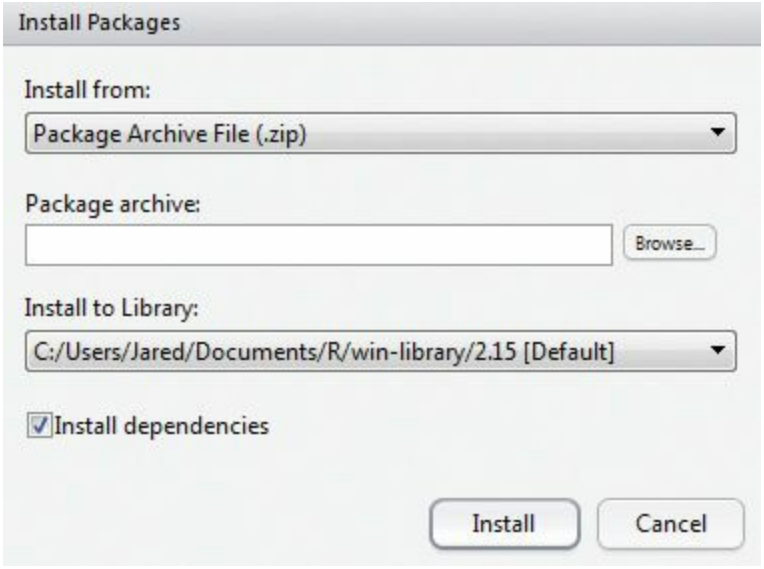

Figure 3.3 RStudio's package installation dialog to install from an archive file.

Similarly to before, this can be accomplished using `install.packages`.

Click here to view code image

```
> install.packages("coefplot_1.1.7.zip")
```

3.1.1. Uninstalling Packages

In the rare instance when a package needs to be uninstalled, it is easiest to click the white X inside a grey circle on the right of the package description in RStudio's Packages pane shown in Figure 3.1. Alternatively, this can be done with `remove.packages` where the first argument is a `character vector` naming the packages to be removed.

3.2. Loading Packages

Now that packages are installed they are almost ready to use and just need to be loaded first. There are two commands that can be used, either `library` or `require`. They both accomplish the same thing—loading the package—but `require` will return `TRUE` if it succeeds and `FALSE` with a warning if it cannot find the package. This returned value is useful when loading a package from within a function, a practice considered acceptable to some, improper to others. In general usage there is not much of a difference, so it comes down to personal preference. The argument to either function is the name of the desired package, with or without quotes. So loading the `coefplot` package would look like:

Click here to view code image

```
> require(coefplot)

Loading required package: coefplot
Loading required package: ggplot2
```

It prints out the dependent packages that get loaded as well. This can be suppressed by setting the argument `quietly` to `TRUE`.

Click here to view code image

```
> require(coefplot, quietly = TRUE)
```

A package only needs to be loaded when starting a new `R` session. Once loaded, it remains available until either `R` is restarted or the package is unloaded, as described in Section 3.2.1.

An alternative to loading a package through code is to select the checkbox next to the package name in RStudio's Packages pane, seen on the left of Figure 3.1. This will load the package by running the code just shown.

3.2.1. Unloading Packages

Sometimes a package needs to be unloaded. This is simple enough either by clearing the checkbox in RStudio's Packages pane or by using the `detach` function. The function takes the package name preceded by `package:` all in quotes.

Click here to view code image

```
> detach("package:coefplot")
```

It is not uncommon for functions in different packages to have the same name. For example, `coefplot` is in both `arm` (by Andrew Gelman) and `coefplot`.[2] If both packages are loaded, the function in the package loaded last will be invoked when calling that function. A way around this is to precede the function with the name of the package, separated by two colons (::).

2. This particular instance is because we built `coefplot` as an improvement on the one available in `arm`. There are other instances where the names have nothing in common.

Click here to view code image

```
> arm::coefplot(object)
> coefplot::coefplot(object)
```

Not only does this call the appropriate function, it also allows the function to be called without even loading the package beforehand.

3.3. Building a Package

3.3. Building a Package

Building a package is one of the more rewarding parts of working with R, especially sharing that package with the community through CRAN. Chapter 24 discusses this process in detail.

3.4. Conclusion

Packages make up the backbone of the R community and experience. They are often considered what makes working with R so desirable. This is how the community makes its work, and so many of the statistical techniques, available to the world. With such a large number of packages, finding the right one can be overwhelming. CRAN Task Views (http://cran.r-project.org/web/views/) offers a curated listing of packages for different needs. However, the best way to find a new package might just be to ask the community. Appendix A gives some resources for doing just that.

Chapter 4. Basics of R

R is a powerful tool for all manner of calculations, data manipulation and scientific computations. Before getting to the complex operations possible in R we must start with the basics. Like most languages R has its share of mathematical capability, variables, functions and data types.

4.1. Basic Math

Being a statistical programming language, R can certainly be used to do basic math and that is where we will start.

We begin with the "Hello, World!" of basic math: 1 + 1. In the console there is a right angle bracket (>) where code should be entered. Simply test R by running

```
> 1 + 1

[1] 2
```

If this returns 2, then everything is great; if not, then something is very, very wrong. Assuming it worked, let's look at some slightly more complicated expressions:

```
> 1 + 2 + 3

[1] 6

> 3 * 7 * 2

[1] 42

> 4/2

[1] 2

> 4/3

[1] 1.333
```

These follow the basic order of operations: Parenthesis, Exponents, Multiplication, Division, Addition and Subtraction (PEMDAS). This means operations inside parentheses take priority over other operations. Next on the priority list is exponentiation. After that multiplication and division are performed, followed by addition and subtraction.

This is why the first two lines in the following code have the same result while the third is different.

```
> 4 * 6 + 5

[1] 29

> (4 * 6) + 5

[1] 29

> 4 * (6 + 5)

[1] 44
```

So far we have put white space in between each operator such as * and /. This is not necessary but is encouraged as good coding practice.

4.2. Variables

Variables are an integral part of any programming language and R offers a great deal of flexibility. Unlike statically typed languages such as C++, R does not require variable types to be declared. A variable can take on any available data type as described in Section 4.3. It can also hold any R object such as a function, the result of an analysis or a plot. A single variable can at one point hold a number, then later hold a character and then later a number again.

4.2.1. Variable Assignment

There are a number of ways to assign a value to a variable, and again, this does not depend on the type of value being assigned.

The valid assignment operators are <- and = with the first being preferred.

For example, let's save 2 to the variable *x* and 5 to the variable *y*.

```
> x <- 2
> x

[1] 2

> y = 5
> y

[1] 5
```

The arrow operator can also point in the other direction.

```
> 3 <- z
> z

[1] 3
```

The assignment operation can be used successively to assign a value to multiple variables simultaneously.

```
> a <- b <- 7
> a

[1] 7

> b

[1] 7
```

A more laborious, though sometimes necessary, way to assign variables is to use the `assign` function.

```
> assign("j", 4)
> j

[1] 4
```

Variable names can contain any combination of alphanumeric characters along with periods (.) and

underscores (_). However, they cannot *start* with a number or an underscore.

The most common form of assignment in the R community is the left arrow (<-), which may seem awkward to use at first but eventually becomes second nature. It even seems to make sense, as the variable is sort of pointing to its value. There is also a particularly nice benefit for people coming from languages like SQL, where a single equal sign (=) tests for equality.

It is generally considered best practice to use actual names, usually nouns, for variables instead of single letters. This provides more information to the person reading the code. This is seen throughout this book.

4.2.2. Removing Variables

For various reasons a variable may need to be removed. This is easily done using `remove` or its shortcut `rm`.

Click here to view code image

```
> j

[1] 4

> rm(j)
> # now it is gone
> j

Error: object 'j' not found
```

This frees up memory so that R can store more objects, although it does not necessarily free up memory for the operating system. To guarantee that, use `gc`, which performs garbage collection, releasing unused memory to the operating system. R automatically does garbage collection periodically, so this function is not essential.

Variable names are case sensitive, which can trip up people coming from a language like SQL or Visual Basic.

Click here to view code image

```
> theVariable <- 17
> theVariable

[1] 17

> THEVARIABLE

Error: object 'THEVARIABLE' not found
```

4.3. Data Types

There are numerous data types in R that store various kinds of data. The four main types of data most likely to be used are `numeric`, `character` (string), `Date/POSIXct` (time-based) and `logical` (TRUE/FALSE).

The type of data contained in a variable is checked with the `class` function.

```
> class(x)

[1] "numeric"
```

4.3.1. Numeric Data

As expected, R excels at running numbers, so numeric data is the most common type in R. The most commonly used numeric data is `numeric`. This is similar to a `float` or `double` in other languages. It handles integers and decimals, both positive and negative, and, of course, zero. A numeric value stored in a variable is automatically assumed to be `numeric`. Testing whether a variable is `numeric` is done with the function `is.numeric`.

```
> is.numeric(x)

[1] TRUE
```

Another important, if less frequently used, type is `integer`. As the name implies this is for whole numbers only, no decimals. To set an integer to a variable it is necessary to append the value with an `L`. As with checking for a `numeric`, the `is.integer` function is used.

```
> i <- 5L
> i

[1] 5

> is.integer(i)

[1] TRUE
```

Do note that, even though `i` is an `integer`, it will also pass a `numeric` check.

```
> is.numeric(i)

[1] TRUE
```

R nicely promotes `integers` to `numeric` when needed. This is obvious when multiplying an `integer` by a `numeric`, but importantly it works when dividing an `integer` by another `integer`, resulting in a decimal number.

```
> class(4L)

[1] "integer"

> class(2.8)

[1] "numeric"

> 4L * 2.8

[1] 11.2

> class(4L * 2.8)

[1] "numeric"

>
> class(5L)

[1] "integer"

> class(2L)
```

```
[1] "integer"

> 5L/2L

[1] 2.5

> class(5L/2L)

[1] "numeric"
```

4.3.2. Character Data

Even though it is not explicitly mathematical, the character (string) data type is very common in statistical analysis and must be handled with care. `R` has two primary ways of handling character data: `character` and `factor`. While they may seem similar on the surface, they are treated quite differently.

```
> x <- "data"
> x

[1] "data"

> y <- factor("data")
> y

[1] data
Levels: data
```

Notice that `x` contains the word "data" encapsulated in quotes, while `y` has the word "data" without quotes and a second line of information about the `levels` of `y`. That is explained further in Section 4.4.2 about `vectors`.

`Characters` are case sensitive, so "Data" is different from "data" or "DATA."

To find the length of a `character` (or `numeric`) use the `nchar` function.

```
> nchar(x)

[1] 4

> nchar("hello")

[1] 5

> nchar(3)

[1] 1

> nchar(452)

[1] 3
```

This will not work for `factor` data.

Click here to view code image

```
> nchar(y)

Error: 'nchar()' requires a character vector
```

4.3.3. Dates

Dealing with dates and times can be difficult in any language, and to further complicate matters R has numerous different types of dates. The most useful are Date and POSIXct. Date stores just a date while POSIXct stores a date and time. Both objects are actually represented as the number of days (Date) or seconds (POSIXct) since January 1, 1970.

Click here to view code image

```
> date1 <- as.Date("2012-06-28")
> date1

[1] "2012-06-28"

> class(date1)

[1] "Date"

> as.numeric(date1)

[1] 15519

>
> date2 <- as.POSIXct("2012-06-28 17:42")
> date2

[1] "2012-06-28 17:42:00 EDT"

> class(date2)

[1] "POSIXct" "POSIXt"

> as.numeric(date2)

[1] 1340919720
```

Easier manipulation of date and time objects can be accomplished using the lubridate and chron packages.

Using functions such as as.numeric or as.Date does not merely change the formatting of an object but actually changes the underlying type.

Click here to view code image

```
> class(date1)

[1] "Date"

> class(as.numeric(date1))

[1] "numeric"
```

4.3.4. Logical

logicals are a way of representing data that can be either TRUE or FALSE. Numerically, TRUE is the same as 1 and FALSE is the same as 0. So TRUE * 5 equals 5 while FALSE * 5 equals 0.

```
> TRUE * 5

[1] 5
```

```
> FALSE * 5

[1] 0
```

Similar to other types, `logicals` have their own test, using the `is.logical` function.

```
> k <- TRUE
> class(k)

[1] "logical"

> is.logical(k)

[1] TRUE
```

`R` provides `T` and `F` as shortcuts for `TRUE` and `FALSE`, respectively, but it is best practice not to use them, as they are simply variables storing the values `TRUE` and `FALSE` and can be overwritten, which can cause a great deal of frustration as seen in the following example.

```
> TRUE

[1] TRUE

> T

[1] TRUE

> class(T)

[1] "logical"

> T <- 7
> T

[1] 7

> class(T)

[1] "numeric"
```

`logicals` can result from comparing two numbers, or characters.

Click here to view code image

```
> # does 2 equal 3?
> 2 == 3

[1] FALSE

> # does 2 not equal three?
> 2 != 3

[1] TRUE

> # is two less than three?
> 2 < 3

[1] TRUE

> # is two less than or equal to three?
```

```
> 2 <= 3

[1] TRUE

> # is two greater than three?
> 2 > 3

[1] FALSE

> # is two greater than or equal to three?
> 2 >= 3

[1] FALSE

> # is 'data' equal to 'stats'?
> "data" == "stats"

[1] FALSE

> # is 'data' less than 'stats'?
> "data" < "stats"

[1] TRUE
```

4.4. Vectors

A `vector` is a collection of elements, all of the same type. For instance, `c(1, 3, 2, 1, 5)` is a `vector` consisting of the numbers 1, 3, 2, 1, 5, in that order. Similarly, `c("R", "Excel", "SAS", "Excel")` is a `vector` of the `character` elements "R," "Excel," "SAS" and "Excel." A `vector` cannot be of mixed type.

`vectors` play a crucial, and helpful, role in `R`. More than being simple containers, `vectors` in `R` are special in that `R` is a vectorized language. That means operations are applied to each element of the `vector` automatically, without the need to loop through the `vector`. This is a powerful concept that may seem foreign to people coming from other languages, but it is one of the greatest things about `R`.

`vectors` do not have a dimension, meaning there is no such thing as a column `vector` or row `vector`. These `vectors` are not like the mathematical `vector` where there is a difference between row and column orientation.[1]

1. Column or row `vectors` can be represented as one-dimensional `matrices`, which are discussed in Section 5.3.

The most common way to create a `vector` is with `c`. The "c" stands for combine because multiple elements are being combined into a `vector`.

Click here to view code image

```
> x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> x

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

4.4.1. Vector Operations

Now that we have a `vector` of the first ten numbers, we might want to multiply each element by 3. In `R` this is a simple operation using just the multiplication operator ($\star$).

Click here to view code image

```
> x * 3

 [1]  3  6  9 12 15 18 21 24 27 30
```

No loops are necessary. Addition, subtraction and division are just as easy. This also works for any number of operations.

Click here to view code image

```
> x + 2

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

> x - 3

 [1] -2 -1  0  1  2  3  4  5  6  7

> x/4

 [1] 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50

> x^2

 [1]   1   4   9  16  25  36  49  64  81 100

> sqrt(x)

 [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
```

Earlier we created a `vector` of the first ten numbers using the `c` function, which creates a `vector`. A shortcut is the `:` operator, which generates a sequence of consecutive numbers, in either direction.

Click here to view code image

```
> 1:10

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

> 10:1

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

> -2:3

[1] -2 -1  0  1  2  3

> 5:-7

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

Vector operations can be extended even further. Let's say we have two vectors of equal length. Each of the corresponding elements can be operated on together.

Click here to view code image

```
> # create two vectors of equal length
> x <- 1:10
> y <- -5:4
> # add them
> x + y
```

```
 [1] -4 -2  0  2  4  6  8 10 12 14

> # subtract them
> x - y

 [1] 6 6 6 6 6 6 6 6 6 6

> # multiply them
> x * y

 [1] -5 -8 -9 -8 -5 0 7 16 27 40

> # divide them--notice division by 0 results in Inf
> x/y

 [1] -0.2 -0.5 -1.0 -2.0 -5.0 Inf 7.0 4.0 3.0 2.5

> # raise one to the power of the other
> x^y

 [1] 1.000e+00 6.250e-02 3.704e-02 6.250e-02 2.000e-01 1.000e+00
 [7] 7.000e+00 6.400e+01 7.290e+02 1.000e+04

> # check the length of each
> length(x)

[1] 10

> length(y)

[1] 10

> # the length of them added together should be the same
> length(x + y)

[1] 10
```

In the preceding code block, notice the hash # symbol. This is used for comments. Anything following the hash, on the same line, will be commented out and not run.

Things get a little more complicated when operating on two `vectors` of unequal length. The shorter `vector` gets recycled, that is, its elements are repeated, in order, until they have been matched up with every element of the longer `vector`. If the longer one is not a multiple of the shorter one, a warning is given.

Click here to view code image

```
> x + c(1, 2)

 [1]  2  4  4  6  6  8  8 10 10 12

> x + c(1, 2, 3)

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

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

Comparisons also work on `vectors`. Here the result is a vector of the same length containing

TRUE or FALSE for each element.

Click here to view code image

```
> x <= 5

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

> x > y

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

> x < y

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

To test whether all the resulting elements are TRUE, use the all function. Similarly, the any function checks whether any element is TRUE.

```
> x <- 10:1
> y <- -4:5
> any(x < y)

[1] TRUE

> all(x < y)

[1] FALSE
```

The nchar function also acts on each element of a vector.

Click here to view code image

```
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+        "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
> nchar(q)

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

> nchar(y)

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

Accessing individual elements of a vector is done using square brackets ([ ]). The first element of x is retrieved by typing x[1], the first two elements by x[1:2] and nonconsecutive elements by x[c(1, 4)].

```
> x[1]

[1] 10

> x[1:2]

[1] 10 9

> x[c(1, 4)]

[1] 10  7
```

This works for all types of vectors whether they are numeric, logical, character and

so forth.

It is possible to give names to a `vector` either during creation or after the fact.

Click here to view code image

```
> # provide a name for each element of an array using a name-value pair
> c(One = "a", Two = "y", Last = "r")

 One Two Last
 "a" "y" "r"

>
> # create a vector
> w <- 1:3
> # name the elements
> names(w) <- c("a", "b", "c")
> w

a b c
1 2 3
```

4.4.2. Factor Vectors

`factors` are an important concept in `R`, especially when building models. Let's create a simple `vector` of text data that has a few repeats. We will start with the `vector q` we created earlier and add some elements to it.

Click here to view code image

```
> q2 <- c(q, "Hockey", "Lacrosse", "Hockey", "Water Polo",
+         "Hockey", "Lacrosse")
```

Converting this to a `factor` is easy with `as.factor`.

Click here to view code image

```
> q2Factor <- as.factor(q2)
> q2Factor

 [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
 [7] Basketball Tennis     Cricket    Soccer     Hockey     Lacrosse
[13] Hockey     Water Polo Hockey     Lacrosse
11 Levels: Baseball Basketball Cricket Curling Football ... Water Polo
```

Notice that after printing out every element of `q2Factor`, `R` also prints the `levels` of `q2Factor`. The `levels` of a `factor` are the unique values of that `factor` variable. Technically, `R` is giving each unique value of a `factor` a unique `integer` tying it back to the `character` representation. This can be seen with `as.numeric`.

Click here to view code image

```
> as.numeric(q2Factor)

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

In ordinary `factors` the order of the `levels` does not matter and one `level` is no different from another. Sometimes, however, it is important to understand the order of a `factor`, such as when coding education levels. Setting the `ordered` argument to `TRUE` creates an `ordered factor` with the order given in the `levels` argument.

Click here to view code image

```
> factor(x=c("High School", "College", "Masters", "Doctorate"),
+        levels=c("High School", "College", "Masters", "Doctorate"),
+        ordered=TRUE)

[1] High School College     Masters     Doctorate
Levels: High School < College < Masters < Doctorate
```

`factors` can drastically reduce the size of the variable because they are storing only the unique values, but they can cause headaches if not used properly. This will be discussed further throughout the book.

4.5. Calling Functions

Earlier we briefly used a few basic functions like `nchar`, `length` and `as.Date` to illustrate some concepts. Functions are very important and helpful in any language because they make code easily repeatable. Almost every step taken in `R` involves using functions, so it is best to learn the proper way to call them. `R` function calling is filled with a good deal of nuance, so we are going to focus on the gist of what is needed to know. Of course, throughout the book there will be many examples of calling functions.

Let's start with the simple `mean` function, which computes the average of a set of numbers. In its simplest form it takes a `vector` as an argument.

```
> mean(x)

[1] 5.5
```

More complicated functions have multiple arguments that can be either specified by the order they are entered or by using their name with an equal sign. We will see further use of this throughout the book.

`R` provides an easy way for users to build their own functions, which we will cover in more detail in Chapter 8.

4.6. Function Documentation

Any function provided in `R` has accompanying documentation, of varying quality of course. The easiest way to access that documentation is to place a question mark in front of the function name, like this: `?mean`.

To get help on binary operators like +, * or == surround them with back ticks (`).

```
> ?`+`
> ?`*`
> ?`==`
```

There are occasions when we have only a sense of the function we want to use. In that case we can look up the function by using part of the name with `apropos`.

Click here to view code image

```
> apropos("mea")

 [1] ".cache/mean-simple_ce29515dafe58a90a771568646d73aae"
 [2] ".colMeans"
```

```
 [3] ".rowMeans"
 [4] "colMeans"
 [5] "influence.measures"
 [6] "kmeans"
 [7] "mean"
 [8] "mean.Date"
 [9] "mean.default"
[10] "mean.difftime"
[11] "mean.POSIXct"
[12] "mean.POSIXlt"
[13] "mean_cl_boot"
[14] "mean_cl_normal"
[15] "mean_sdl"
[16] "mean_se"
[17] "rowMeans"
[18] "weighted.mean"
```

4.7. Missing Data

Missing data plays a critical role in both statistics and computing, and R has two types of missing data, NA and NULL. While they are similar, they behave differently and that difference needs attention.

4.7.1. NA

Often we will have data that has missing values for any number of reasons. Statistical programs use varying techniques to represent missing data such as a dash, a period or even the number 99. R uses NA. NA will often be seen as just another element of a vector. is.na tests each element of a vector for missingness.

Click here to view code image

```
> z <- c(1, 2, NA, 8, 3, NA, 3)
> z

[1] 1 2 NA 8 3 NA 3

> is.na(z)

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

NA is entered simply by typing the letters “N” and “A” as if they were normal text. This works for any kind of vector.

Click here to view code image

```
> zChar <- c("Hockey", NA, "Lacrosse")
> zChar

[1] "Hockey" NA           "Lacrosse"

> is.na(zChar)

[1] FALSE TRUE FALSE
```

Handling missing data is an important part of statistical analysis. There are many techniques depending on field and preference. One popular technique is multiple imputation, which is discussed in detail in Chapter 25 of Andrew Gelman and Jennifer Hill’s book *Data Analysis Using Regression*

and Multilevel/Hierarchical Models, and is implemented in the `mi`, `mice` and `Amelia` packages.

4.7.2. NULL

`NULL` is the absence of anything. It is not exactly missingness, it is nothingness. Functions can sometimes return `NULL` and their arguments can be `NULL`. An important difference between `NA` and `NULL` is that `NULL` is atomical and cannot exist within a `vector`. If used inside a `vector` it simply disappears.

```
> z <- c(1, NULL, 3)
> z

[1] 1 3
```

Even though it was entered into the `vector z`, it did not get stored in `z`. In fact, `z` is only two elements long.

The test for a `NULL` value is `is.null`.

```
> d <- NULL
> is.null(d)

[1] TRUE

> is.null(7)

[1] FALSE
```

Since `NULL` cannot be a part of a `vector`, `is.null` is appropriately not vectorized.

4.8. Conclusion

Data come in many types, and `R` is well equipped to handle them. In addition to basic calculations, `R` can handle numeric, character and time-based data. One of the nicer parts of working with `R`, although one that requires a different way of thinking about programming, is vectorization. This allows operating on multiple elements in a `vector` simultaneously, which leads to faster and more mathematical code.

Chapter 5. Advanced Data Structures

Sometimes data requires more complex storage than simple `vectors` and thankfully `R` provides a host of data structures. The most common are the `data.frame`, `matrix` and `list` followed by the `array`. Of these, the `data.frame` will be most familiar to anyone who has used a spreadsheet, the `matrix` to people familiar with matrix math and the `list` to programmers.

5.1. `data.frames`

Perhaps one of the most useful features of `R` is the `data.frame`. It is one of the most often cited reasons for `R`'s ease of use.

On the surface a `data.frame` is just like an Excel spreadsheet in that it has columns and rows. In statistical terms, each column is a variable and each row is an observation.

In terms of how `R` organizes `data.frames`, each column is actually a `vector`, each of which has the same length. That is very important because it lets each column hold a different type of data (see Section 4.3). This also implies that within a column each element must be of the same type, just like with `vectors`.

There are numerous ways to construct a `data.frame`, the simplest being to use the `data.frame` function. Let's create a basic `data.frame` using some of the `vectors` we have already introduced, namely `x`, `y` and `q`.

Click here to view code image

```
> x <- 10:1
> y <- -4:5
> q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
+        "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")
> theDF <- data.frame(x, y, q)
> theDF

   x   y          q
1 10  -4     Hockey
2  9  -3   Football
3  8  -2   Baseball
4  7  -1    Curling
5  6   0      Rugby
6  5   1   Lacrosse
7  4   2 Basketball
8  3   3     Tennis
9  2   4    Cricket
10 1   5     Soccer
```

This creates a `10x3 data.frame` consisting of those three `vectors`. Notice the names of `theDF` are simply the variables. We could have assigned names during the creation process, which is generally a good idea.

Click here to view code image

```
> theDF <- data.frame(First = x, Second = y, Sport = q)
> theDF

   First Second      Sport
1     10     -4     Hockey
```

```
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer
```

`data.frames` are complex objects with many attributes. The most frequently checked attributes are the number of rows and columns. Of course there are functions to do this for us: `nrow` and `ncol`. And in case both are wanted at the same time there is the `dim` function.

```
> nrow(theDF)

[1] 10

> ncol(theDF)

[1] 3

> dim(theDF)

[1] 10  3
```

Checking the column names of a `data.frame` is as simple as using the `names` function. This returns a `character vector` listing the columns. Since it is a `vector` we can access individual elements of it just like any other `vector`.

Click here to view code image

```
> names(theDF)

[1] "First" "Second" "Sport"

> names(theDF)[3]

[1] "Sport"
```

We can also check and assign the row names of a `data.frame`.

Click here to view code image

```
> rownames(theDF)

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

> rownames(theDF) <- c("One", "Two", "Three", "Four", "Five", "Six",
+                      "Seven", "Eight", "Nine", "Ten")
> rownames(theDF)

 [1] "One"   "Two"   "Three" "Four"  "Five"  "Six"   "Seven"  "Eight"
 [9] "Nine"  "Ten"

> # set them back to the generic index
> rownames(theDF) <- NULL
> rownames(theDF)

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

Usually a `data.frame` has far too many rows to print them all to the screen, so thankfully the `head` function prints out only the first few rows.

Click here to view code image

```
> head(theDF)

  First Second    Sport
1    10     -4   Hockey
2     9     -3 Football
3     8     -2 Baseball
4     7     -1  Curling
5     6      0    Rugby
6     5      1 Lacrosse

> head(theDF, n = 7)

  First Second      Sport
1    10     -4     Hockey
2     9     -3   Football
3     8     -2   Baseball
4     7     -1    Curling
5     6      0      Rugby
6     5      1   Lacrosse
7     4      2 Basketball

> tail(theDF)

  First Second      Sport
5     6      0      Rugby
6     5      1   Lacrosse
7     4      2 Basketball
8     3      3     Tennis
9     2      4    Cricket
10    1      5     Soccer
```

As we can with other variables, we can check the `class` of a `data.frame` using the `class` function.

```
> class(theDF)

[1] "data.frame"
```

Since each column of the `data.frame` is an individual `vector`, it can be accessed individually and each has its own `class`. Like many other aspects of `R`, there are multiple ways to access an individual column. There is the `$` operator and also the square brackets. Running `theDF$Sport` will give the third column in `theDF`. That allows us to specify one particular column by name.

Click here to view code image

```
> theDF$Sport

 [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
 [7] Basketball Tennis     Cricket    Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis
```

Similar to `vectors`, `data.frames` allow us to access individual elements by their position using square brackets, but instead of having one position two are specified. The first is the row number and the second is the column number. So to get the third row from the second column we use

`theDF[3, 2]`.

```
> theDF[3, 2]

[1] -2
```

To specify more than one row or column use a `vector` of indices.

Click here to view code image

```
> # row 3, columns 2 through 3
> theDF[3, 2:3]

  Second    Sport
3     -2 Baseball

>
> # rows 3 and 5, column 2
> # since only one column was selected it was returned as a vector

> # hence the column names will not be printed
> theDF[c(3, 5), 2]

[1] -2 0

>
> # rows 3 and 5, columns 2 through 3
> theDF[c(3, 5), 2:3]

  Second    Sport
3     -2 Baseball
5      0    Rugby
```

To access an entire row, specify that row while not specifying any column. Likewise, to access an entire column, specify that column while not specifying any row.

Click here to view code image

```
> # all of column 3
> # since it is only one column a vector is returned
> theDF[, 3]

 [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
 [7] Basketball Tennis     Cricket    Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis
>
> # all of columns 2 through 3
> theDF[, 2:3]

  Second      Sport
1     -4     Hockey
2     -3   Football
3     -2   Baseball
4     -1    Curling
5      0      Rugby
6      1   Lacrosse
7      2 Basketball
8      3     Tennis
9      4    Cricket
10     5     Soccer

>
```

```
> # all of row 2
> theDF[2, ]

  First Second    Sport
2     9     -3 Football

>
> # all of rows 2 through 4
> theDF[2:4, ]

  First Second    Sport
2     9     -3 Football
3     8     -2 Baseball
4     7     -1  Curling
```

To access multiple columns by name, make the column argument a `character vector` of the names.

Click here to view code image

```
> theDF[, c("First", "Sport")]

   First      Sport
1     10     Hockey
2      9   Football
3      8   Baseball
4      7    Curling
5      6      Rugby
6      5   Lacrosse
7      4 Basketball
8      3     Tennis
9      2    Cricket
10     1     Soccer
```

Yet another way to access a specific column is to use its column name (or its number) either as second argument to the square brackets or as the only argument to either single or double square brackets.

Click here to view code image

```
> # just the "Sport" column
> # since it is one column it returns as a (factor) vector
> theDF[, "Sport"]

  [1] Hockey     Football  Baseball   Curling     Rugby     Lacrosse
  [7] Basketball Tennis    Cricket    Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis

> class(theDF[, "Sport"])

[1] "factor"

>
> # just the "Sport" column
> # this returns a one column data.frame
> theDF["Sport"]

        Sport
1      Hockey
2    Football
3    Baseball
```

```
4     Curling
5       Rugby
6    Lacrosse
7  Basketball
8      Tennis
9     Cricket
10     Soccer

> class(theDF["Sport"])

[1] "data.frame"

>
> # just the "Sport" column
> # this also returns a (factor) vector
> theDF["Sport"]

 [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
 [7] Basketball Tennis     Cricket    Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis

> class(theDF["Sport"])

[1] "factor"
```

All of these methods have differing outputs. Some return a `vector`, some return a single-column `data.frame`. To ensure a single-column `data.frame` while using single-square brackets, there is a third argument: `drop=FALSE`. This also works when specifying a single column by number.

Click here to view code image

```
> theDF[, "Sport", drop = FALSE]

        Sport
1      Hockey
2    Football
3    Baseball
4     Curling
5       Rugby
6    Lacrosse
7  Basketball
8      Tennis
9     Cricket
10     Soccer

> class(theDF[, "Sport", drop = FALSE])

[1] "data.frame"

>
> theDF[, 3, drop = FALSE]

        Sport
1      Hockey
2    Football
3    Baseball
4     Curling
5       Rugby
6    Lacrosse
7  Basketball
8      Tennis
```

```
9     Cricket
10     Soccer

> class(theDF[, 3, drop = FALSE])

[1] "data.frame"
```

In Section 4.4.2 we see that `factors` are stored specially. To see how they would be represented in `data.frame` form, use `model.matrix` to create a set of indicator (or dummy) variables. That is one column for each `level` of a `factor`, with a 1 if a row contains that `level` or a 0 otherwise.

Click here to view code image

```
> newFactor <- factor(c("Pennsylvania", "New York", "New Jersey", "New York",
+     "Tennessee", "Massachusetts", "Pennsylvania", "New York"))
> model.matrix(~newFactor - 1)

  newFactorMassachusetts newFactorNew Jersey newFactorNew York
1                      0                   0                  0
2                      0                   0                  1
3                      0                   1                  0
4                      0                   0                  1
5                      0                   0                  0
6                      1                   0                  0
7                      0                   0                  0
8                      0                   0                  1
  newFactorPennsylvania newFactorTennessee
1                     1                  0
2                     0                  0
3                     0                  0
4                     0                  0
5                     0                  1
6                     0                  0
7                     1                  0
8                     0                  0
attr(,"assign")
[1] 1 1 1 1 1
attr(,"contrasts")
attr(,"contrasts")$newFactor
[1] "contr.treatment"
```

We learn more about formulas (the argument to `model.matrix`) in Sections 11.2 and 12.3.2 and Chapters 15 and 16.

5.2. Lists

Often a container is needed to hold arbitrary objects of either the same type or varying types. `R` accomplishes this through `lists`. They store any number of items of any type. A `list` can contain all `numerics` or `characters` or a mix of the two or `data.frames` or, recursively, other `lists`.

`Lists` are created with the `list` function where each argument to the function becomes an element of the `list`.

Click here to view code image

```
> # creates a three element list
> list(1, 2, 3)
```

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

[[2]]
[1] 2

[[3]]
[1] 3

>
> # creates a single element list where the only element is a vector
> # that has three elements
> list(c(1, 2, 3))

[[1]]
[1] 1 2 3

>
> # creates a two element list
> # the first element is a three element vector
> # the second element is a five element vector
> (list3 <- list(c(1, 2, 3), 3:7))

[[1]]
[1] 1 2 3

[[2]]
[1] 3 4 5 6 7
>
> # two element list
> # first element is a data.frame
> # second element is a 10 element vector
> list(theDF, 1:10)

[[1]]
   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

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

>
> # three element list
> # first is a data.frame
> # second is a vector
> # third is list3, which holds two vectors
> list5 <- list(theDF, 1:10, list3)
> list5

[[1]]
   First Second      Sport
```

```
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

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

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

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

Notice in the previous block of code (where `list3` was created) that enclosing an expression in parentheses displays the results after execution.

Like `data.frames`, `lists` can have names. Each element has a unique name that can be either viewed or assigned using `names`.

Click here to view code image

```
> names(list5)

NULL

> names(list5) <- c("data.frame", "vector", "list")
> names(list5)

[1] "data.frame" "vector"     "list"

> list5

$data.frame
   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

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

$list
$list[[1]]
[1] 1 2 3
```

```
$list[[2]]
[1] 3 4 5 6 7
```

Names can also be assigned to `list` elements during creation using name-value pairs.

Click here to view code image

```
> list6 <- list(TheDataFrame = theDF, TheVector = 1:10, TheList = list3)
> names(list6)

[1] "TheDataFrame" "TheVector"      "TheList"

> list6

$TheDataFrame
   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

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

$TheList
$TheList[[1]]
[1] 1 2 3

$TheList[[2]]
[1] 3 4 5 6 7
```

Creating an empty `list` of a certain size is, perhaps confusingly, done with `vector`.

Click here to view code image

```
> (emptyList <- vector(mode = "list", length = 4))

[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

To access an individual element of a `list`, use double square brackets, specifying either the element number or name. Note that this allows access to only one element at a time.

Click here to view code image

```
> list5[[1]]
```

```
   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

> list5[["data.frame"]]

   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer
```

Once an element is accessed it can be treated as if that actual element is being used, allowing nested indexing of elements.

Click here to view code image

```
> list5[[1]]$Sport

  [1] Hockey     Football   Baseball  Curling    Rugby      Lacrosse
  [7] Basketball Tennis     Cricket   Soccer
10 Levels: Baseball Basketball Cricket Curling Football ... Tennis

> list5[[1]][, "Second"]

 [1] -4 -3 -2 -1  0  1  2  3  4  5

> list5[[1]][, "Second", drop = FALSE]

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

It is possible to append elements to a `list` simply by using an index (either numeric or named) that does not exist.

Click here to view code image

```
> # see how long it currently is
```

```
> length(list5)

[1] 3

>
> # add a fourth element, unnamed
> list5[[4]] <- 2
> length(list5)

[1] 4

>
> # add a fifth element, named
> list5[["NewElement"]] <- 3:6
> length(list5)

[1] 5

>
> names(list5)

[1] "data.frame" "vector"     "list"       ""           "NewElement"

> list5

$data.frame
   First Second      Sport
1     10     -4     Hockey
2      9     -3   Football
3      8     -2   Baseball
4      7     -1    Curling
5      6      0      Rugby
6      5      1   Lacrosse
7      4      2 Basketball
8      3      3     Tennis
9      2      4    Cricket
10     1      5     Soccer

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

$list
$list[[1]]
[1] 1 2 3

$list[[2]]
[1] 3 4 5 6 7

[[4]]
[1] 2

$NewElement
[1] 3 4 5 6
```

Occasionally appending to a `list`—or `vector` or `data.frame` for that matter—is fine, but doing so repeatedly is computationally expensive. So it is best to create a `list` as long as its final desired size and then fill it in using the appropriate indices.

5.3. Matrices

A very common mathematical structure that is essential to statistics is a `matrix`. This is similar to a `data.frame` in that it is rectangular with rows and columns except that every single element, regardless of column, must be the same type, most commonly all `numerics`. They also act similarly to `vectors` with element-by-element addition, multiplication, subtraction, division and equality. The `nrow`, `ncol` and `dim` functions work just like they do for `data.frames`.

Click here to view code image

```
> # create a 5x2 matrix
> A <- matrix(1:10, nrow = 5)
> # create another 5x2 matrix
> B <- matrix(21:30, nrow = 5)
> # create another 5x2 matrix
> C <- matrix(21:40, nrow = 2)
> A

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

> B

     [,1] [,2]
[1,]   21   26
[2,]   22   27
[3,]   23   28
[4,]   24   29
[5,]   25   30

> C

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   21   23   25   27   29   31   33   35   37    39
[2,]   22   24   26   28   30   32   34   36   38    40

> nrow(A)

[1] 5

> ncol(A)

[1] 2

> dim(A)

[1] 5 2

> # add them
> A + B

     [,1] [,2]
[1,]   22   32
[2,]   24   34
[3,]   26   36
[4,]   28   38
[5,]   30   40
```

```
> # multiply them
> A * B

     [,1] [,2]
[1,]   21  156
[2,]   44  189
[3,]   69  224
[4,]   96  261
[5,]  125  300

> # see if the elements are equal
> A == B

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

`Matrix` multiplication is a commonly used operation in mathematics, requiring the number of columns of the left-hand `matrix` to be the same as the number of rows of the right-hand `matrix`. Both `A` and `B` are 5*X*2 so we will transpose `B` so it can be used on the right-hand side.

Click here to view code image

```
> A %*% t(B)

     [,1] [,2] [,3] [,4] [,5]
[1,]  177  184  191  198  205
[2,]  224  233  242  251  260
[3,]  271  282  293  304  315
[4,]  318  331  344  357  370
[5,]  365  380  395  410  425
```

Another similarity with `data.frames` is that `matrices` can also have row and column names.

Click here to view code image

```
> colnames(A)

NULL

> rownames(A)

NULL

> colnames(A) <- c("Left", "Right")
> rownames(A) <- c("1st", "2nd", "3rd", "4th", "5th")
>
> colnames(B)

NULL

> rownames(B)

NULL

> colnames(B) <- c("First", "Second")
```

```
> rownames(B) <- c("One", "Two", "Three", "Four", "Five")
>
> colnames(C)

NULL

> rownames(C)

NULL

> colnames(C) <- LETTERS[1:10]
> rownames(C) <- c("Top", "Bottom")
```

There are two special `vectors`, `letters` and `LETTERS`, that contain the lower-case and upper-case letters, respectively.

Notice the effect when transposing a `matrix` and multiplying `matrices`. Transposing naturally flips the row and column names. `Matrix` multiplication keeps the row names from the left `matrix` and the column names from the right `matrix`.

Click here to view code image

```
> t(A)
      1st 2nd 3rd 4th 5th
Left    1   2   3   4   5
Right   6   7   8   9  10

> A %*% C

      A   B   C   D   E   F   G   H   I   J
1st 153 167 181 195 209 223 237 251 265 279
2nd 196 214 232 250 268 286 304 322 340 358
3rd 239 261 283 305 327 349 371 393 415 437
4th 282 308 334 360 386 412 438 464 490 516
5th 325 355 385 415 445 475 505 535 565 595
```

5.4. Arrays

An `array` is essentially a multidimensional `vector`. It must all be of the same type and individual elements are accessed in a similar fashion using square brackets. The first element is the row index, the second is the column index and the remaining elements are for outer dimensions.

Click here to view code image

```
> theArray <- array(1:12, dim = c(2, 3, 2))
> theArray

, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

> theArray[1, , ]
```

```
     [,1] [,2]
[1,]    1    7
[2,]    3    9
[3,]    5   11

> theArray[1, , 1]

[1] 1 3 5

> theArray[, , 1]

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

The main difference between an `array` and a `matrix` is that `matrices` are restricted to two dimensions while `arrays` can have an arbitrary number.

5.5. Conclusion

Data come in many types and structures, which can pose a problem for some analysis environments but `R` handles them with aplomb. The most common data structure is the one-dimensional `vector`, which forms the basis of everything in `R`. The most powerful structure is the `data.frame`—something special in `R` that most other languages do not have—which handles mixed data types in a spreadsheet-like format. `Lists` are useful for storing collections of items like a hash in Perl.