Introduction to R

Downloading and Installing R

Windows and OS X users can download R from CRAN, the Comprehensive R Archive Network. Linux and Unix users can install R packages using their package management tool:

Windows

1. Open *http://www.r-project.org/* in your browser.
2. Click on "CRAN". You'll see a list of mirror sites, organized by country.
3. Select a site near you.
4. Click on "Windows" under "Download and Install R".
5. Click on "base".
6. Click on the link for downloading the latest version of R (an *.exe* file).
7. When the download completes, double-click on the *.exe* file and answer the usual questions.

OS X

1. Open *http://www.r-project.org/* in your browser.
2. Click on "CRAN". You'll see a list of mirror sites, organized by country.
3. Select a site near you.
4. Click on "MacOS X".
5. Click on the *.pkg* file for the latest version of R, under "Files:", to download it.
6. When the download completes, double-click on the *.pkg* file and answer the usual questions.

Linux or Unix

The major Linux distributions have packages for installing R. Here are some examples:

Distribution	Package name
Ubuntu or Debian	r-base
Red Hat or Fedora	R.i386
Suse	R-base

Use the system's package manager to download and install the package. Normally, you will need the root password or sudo privileges; otherwise, ask a system ad- ministrator to perform the installation.

Installing R on Windows or OS X is straightforward because there are prebuilt binaries for those platforms. You need only follow the preceding instructions. The CRAN Web pages also contain links to installation-related resources, such as frequently asked questions (FAQs) and tips for special situations ("How do I install R when using Win- dows Vista?") that you may find useful.

Theoretically, you can install R on Linux or Unix in one of two ways: by installing a distribution package or by building it from scratch. In practice, installing a package is the preferred route. The distribution packages greatly streamline both the initial in- stallation and subsequent updates.

On Ubuntu or Debian, use apt-getto download and install R. Run under sudoto have the necessary privileges:

```
$ sudo apt-get install r-base
```

On Red Hat or Fedora, use yum:

```
$ sudo yum install R.i386
```

Most platforms also have graphical package managers, which you might find more convenient.

Beyond the base packages, I recommend installing the documentation packages, too. On my Ubuntu machine, for example, I installed r-base-html (because I like browsing the hyperlinked documentation) as well as r-doc-html, which installs the important R manuals locally:

$ **sudo apt-get install r-base-html r-doc-html**

Starting R

Windows

Click on Start → All Programs → R; or double-click on the R icon on your desktop (assuming the installer created an icon for you).

OS X

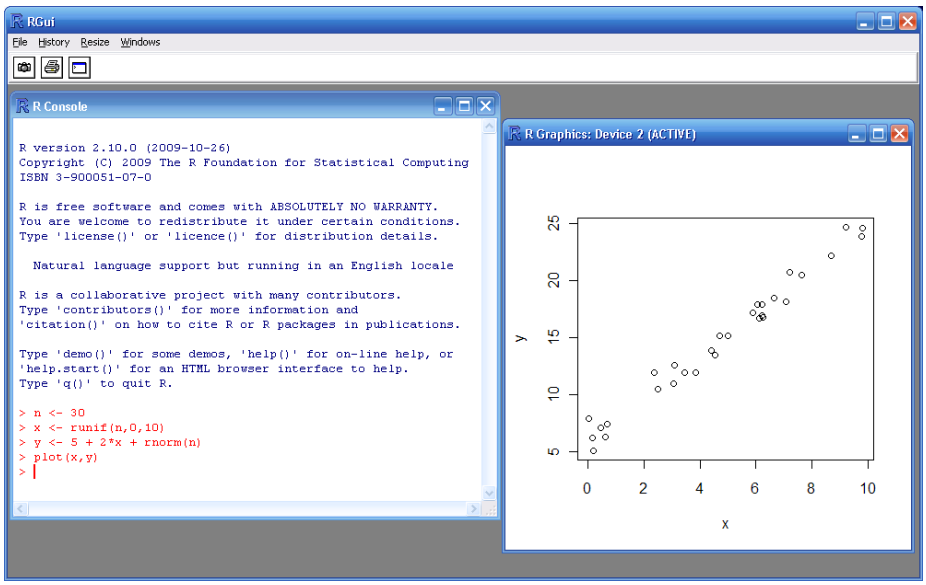
Either click on the icon in the *Applications* directory or put the R icon on the dock and click on the icon there. Alternatively, you can just type R on a Unix command line in a shell.

Linux or Unix

Start the R program from the shell prompt using the R command (uppercase R).

Starting on Windows

When you start R from the Start menu, it opens a new window. The window includes a text pane, called the R Console, where you enter R expressions.



When R starts, what is the working directory? The answer, of course, is that "it depends":

- If you start R from the Start menu, the working directory is normally either *C:\Documents and Settings\<username>\My Documents* (Windows XP)
  or
  C:\Users\<username>\Documents (Windows Vista, Windows 7).
  You can override this de- fault by setting the R_USER environment variable to an alternative directory path.
- If you start R from a desktop shortcut, you can specify an alternative startup directory that becomes the working directory when R is started. To specify the alternative directory, right-click on the shortcut, select Properties, enter the directory path in the box labeled "Start in", and click OK.
- Starting R by double-clicking on your *.RData* file is the most straightforward solution to this little problem. R will automatically change its working directory to be the file's directory, which is usually what you want.

  In any event, you can always use the getwd function to discover your current working directory.

Starting on Linux and Unix

Start the console version of R from the Unix shell prompt simply by typing R, the name of the program. Be careful to type an uppercase R, not a lowercase r. The R program has a bewildering number of command line options. Use the –help option to see the complete list.

R-Studio

Once R is installed you can choose to either work with the basic R console, or with an integrated development environment (IDE). RStudio https://www.rstudio.com/ is by far the most popular IDE for R and supports debugging, workspace management, plotting and much more (make sure to check out the RStudio shortcuts at https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts).

Next to RStudio you also have Architect http://www.openanalytics.eu/architect , and Eclipse-based IDE for R. If you prefer to work with a graphical user interface you can have a look at R-commander http://www.rcommander.com/ (aka as Rcmdr), or Deducer http://www.deducer.org/pmwiki/pmwiki.php?n%3DMain.DeducerManual .

Entering Commands

Simply enter expressions at the command prompt. R will evaluate them and print (dis- play) the result.

R prompts you with ">". To get started, just treat R like a big calculator: enter an expression, and R will evaluate the expression and print the result:

```
> 1+1
[1] 2
```

The computer adds one and one, giving two, and displays the result. The [1]before the 2might be confusing.

To R, the result is a vector, even though it has only one element. R labels the value with [1] to signify that this is the first element of the vector...which is not surprising, since it's the *only* element of the vector.

R will prompt you for input until you type a complete expression. The expression max(1,3,5) is a complete expression, so R stops reading input and evaluates what it's got:

```
> max(1,3,5)
[1] 5
```

In contrast, "max(1,3," is an incomplete expression, so R prompts you for more input. The prompt changes from greater-than (>) to plus (+), letting you know that R expects more:

```
> max(1,3,
+ 5)
[1] 5
```

It's easy to mistype commands, and retyping them is tedious and frustrating. So R includes command-line editing to make life easier. It defines single keystrokes that let you easily recall, correct, and re-execute your commands. A typical command- line interaction goes like this:

1. enter an R expression with a typo.
2. R complains about the mistake.

3. press the up-arrow key to recall the mistaken line.
4. use the left and right arrow keys to move the cursor back to the error.
5. use the Delete key to delete the offending characters.
6. type the corrected characters, which inserts them into the command line.
7. press Enter to re-execute the corrected command.

R supports the usual keystrokes for recalling and editing command lines, as listed in the table below.

Keystrokes for command-line editing

Lab	Ctrl-key	Effect
Up	Ctrl-P	Recall previous command by moving backward through the history of
Do	Ctrl-N	Move forward through the history of commands.
Bac	Ctrl-H	Delete the character to the left of cursor.
Dele	Ctrl-D	Delete the character to the right of cursor.
Ho	Ctrl-A	Move cursor to the start of the line.
End	Ctrl-E	Move cursor to the end of the line.
Righ	Ctrl-F	Move cursor right (forward) one character.
Left	Ctrl-B	Move cursor left (back) one character.
	Ctrl-K	Delete everything from the cursor position to the end of the line.
	Ctrl-U	Clear the whole darn line and start over.
Tab		Name completion (on some platforms).

On Windows and OS X, you can also use the mouse to highlight commands and then use the usual copy and paste commands to paste text into a new command line.

Viewing the Supplied Documentation

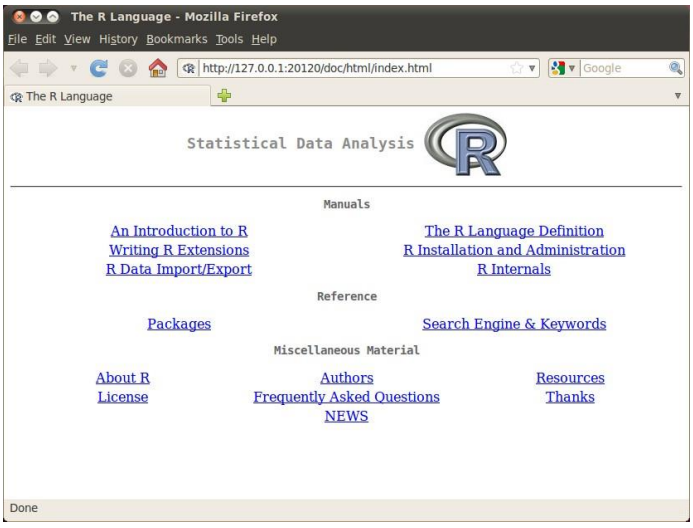
Use the help.start function to see the documentation's table of contents:

> **help.start()**

From there, links are available to all the installed documentation.

The base distribution of R includes a wealth of documentation—literally thousands of pages. When you install additional packages, those packages contain documentation that is also installed on your machine.

It is easy to browse this documentation via the help.start function, which opens a window on the top-level table of contents;



The two links in the Reference section are especially useful:

Packages
Click here to see a list of all the installed packages, both in the base packages and the additional, installed packages. Click on a package name to see a list of its func- tions and datasets.

Search Engine & Keywords
Click here to access a simple search engine, which allows you to search the documentation by keyword or phrase. There is also a list of common keywords, organized by topic; click one to see the associated pages.

Getting Help on a Function

You want to know more about a function that is installed on your machine.

Use help to display the documentation for the function:
> help(*functionname*)

Use args for a quick reminder of the function arguments:
> args(*functionname*)

Use example to see examples of using the function:
> example(*functionname*)

If a function catches your interest, I strongly suggest reading the help page for that function.

Suppose you want to know more about the mean function. Use the help function like this:
> **help(mean)**

This will either open a window with function documentation or display the documentation on your console, depending upon your platform. A shortcut for the help command is to simply type ?followed by the function name:
> **?mean**

Sometimes you just want a quick reminder of the arguments to a function: What are they, and in what order do they occur? Use the args function:

> **args(mean)** function (x, ...) NULL
> **args(sd)**
function (x, na.rm = FALSE) NULL

The first line of output from args is a synopsis of the function call. For mean, the synopsis shows one argument, x, which is a vector of numbers. For sd, the synopsis shows the same vector, x, and an optional argument called na.rm. (You can ignore the second line of output, which is often just NULL.)

Most documentation for functions includes examples near the end. A feature of R is that you can request that it execute the examples, giving you a little demonstration of the function's capabilities. The documentation for the mean function, for instance, contains examples, but you don't need to type them yourself. Just use the example function to watch them run:
> **example(mean)**

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.1)) [1] 8.75 5.50

mean> mean(USArrests, trim = 0.2) Murder Assault
UrbanPop Rape
7.42 167.60 66.20 20.16

The user typed example(mean). Everything else was produced by R, which executed the examples from the help page and displayed the results.

Searching the Supplied Documentation

You want to know more about a function that is installed on your machine, but the help function reports that it cannot find documentation for any such function.

Alternatively, you want to search the installed documentation for a keyword.

Use help.search to search the R documentation on your computer:

> **help.search("*pattern*")**

A typical *pattern* is a function name or keyword. Notice that it must be enclosed in quotation marks.

You can also invoke a search by using two question marks (in which case the quotes are not required):

> **??*pattern***

You may occasionally request help on a function only to be told R knows nothing about it:

```
> help(adf.test)
No documentation for 'adf.test' in specified packages and libraries:
you could try 'help.search("adf.test")'
```

This can be frustrating if you *know* the function is installed on your machine. Here the problem is that the function's package is not currently loaded, and you don't know which package contains the function. It's a kind of catch-22 (the error message indicates the package is not currently in your search path, so R cannot find the help file.

The solution is to search all your installed packages for the function. Just use the help.search function, as suggested in the error message:

```
> help.search("adf.test")
```

The search will produce a listing of all packages that contain the function:

```
Help files with alias or concept or title matching 'adf.test' using regular expression matching:

tseries::adf.test           Augmented Dickey-Fuller Test

Type '?PKG::FOO' to inspect entry 'PKG::FOO TITLE'.
```

The following output, for example, indicates that the tseries package contains the adf.test function. You can see its documentation by explicitly telling help which pack- age contains the function:

```
> help(adf.test, package="tseries")
```

Alternatively, you can insert the tseries package into your search list and repeat the original help command, which will then find the function and display the documentation.

You can broaden your search by using keywords. R will then find any installed documentation that contains the keywords. Suppose you want to find all functions that mention the Augmented Dickey–Fuller (ADF) test. You could search on a likely pattern:

```
> help.search("dickey-fuller")
```

On my machine, the result looks like this because I've installed two additional packages (fUnitRoots and urca) that implement the ADF test:

```
Help files with alias or concept or title matching 'dickey-fuller' using fuzzy matching:

fUnitRoots::DickeyFullerPValues
                            Dickey-Fuller p Values tseries::adf.test      Augmented Dickey-
Fuller Test urca::ur.df     Augmented-Dickey-Fuller Unit Root Test

Type '?PKG::FOO' to inspect entry 'PKG::FOO TITLE'.
```

Getting Help on a Package

You want to learn more about a package installed on your computer.

Use the help function and specify a package name (without a function name):

> **help(package="*packagename*")**

Sometimes you want to know the contents of a package (the functions and datasets). This is especially true after you download and install a new package, for example. The help function can provide the contents plus other information once you specify the package name.

This call to help will display the information for the tseries package, a standard pack- age in the base distribution:

> **help(package="tseries")**

The information begins with a description and continues with an index of functions and datasets. On my machine, the first few lines look like this:

Information on package 'tseries' Description:

Package:	tseries
Version:	0.10-22
Date:	2009-11-22
Title:	Time series analysis and computational finance
Author:	Compiled by Adrian Trapletti <a.trapletti@swissonline.ch>
Maintainer:	Kurt Hornik <Kurt.Hornik@R-project.org> Description: Package for time series analysis and computational finance
Depends:	R (>= 2.4.0), quadprog, stats, zoo
Suggests:	its
Imports:	graphics, stats, utils
License:	GPL-2
Packaged:	2009-11-22 19:03:45 UTC; hornik
Repository:	CRAN Date/Publication: 2009-11-22 19:06:50
Built:	R 2.10.0; i386-pc-mingw32; 2009-12-01 19:32:47 UTC; windows

Index:

NelPlo	Nelson-Plosser Macroeconomic Time Series
USeconomic	U.S. Economic Variables adf.test Augmented Dickey-Fuller Test arma Fit ARMA Models to Time Series

.
. *(etc.)*
.

Some packages also include vignettes, which are additional documents such as introductions, tutorials, or reference cards. They are installed on your computer as part of the package documentation when you install the package. The help page for a package includes a list of its vignettes near the bottom.

You can see a list of all vignettes on your computer by using the vignette function:

> **vignette()**

You can see the vignettes for a particular package by including its name:

> **vignette(package="*packagename*")**

Each vignette has a name, which you use to view the vignette:

> **vignette("*vignettename*")**

Searching the Web for Help

You want to search the Web for information and answers regarding R.

Inside R, use the RSiteSearch function to search by keyword or phrase:

> **RSiteSearch("*key phrase*")**

Inside your browser, try using these sites for searching:

http://rseek.org
This is a Google custom search that is focused on R-specific websites.

http://stackoverflow.com/
Stack Overflow is a searchable Q&A site oriented toward programming issues such as data structures, coding, and graphics.

http://stats.stackexchange.com/
The Statistical Analysis area on Stack Exchange is also a searchable Q&A site, but it is oriented more toward statistics than programming.

The RSiteSearch function will open a browser window and direct it to the search engine on the R Project website. There you will see an initial search that you can refine. For example, this call would start a search for "canonical correlation":

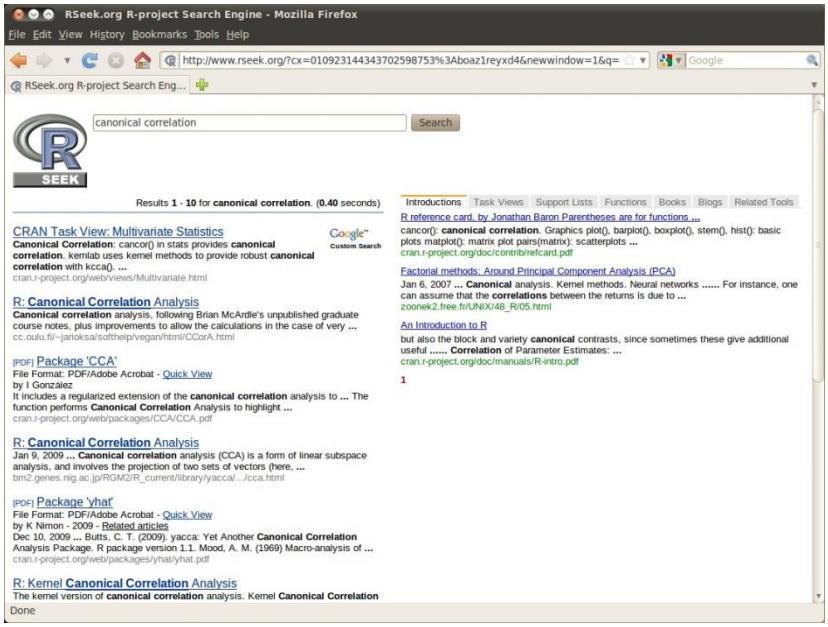
```
> RSiteSearch("canonical correlation")
```

This is quite handy for doing quick web searches without leaving R. However, the search scope is limited to R documentation and the mailing-list archives.

The *rseek.org* site provides a wider search. Its virtue is that it harnesses the power of the Google search engine while focusing on sites relevant to R. That eliminates the extraneous results of a generic Google search. The beauty of *rseek.org* is that it organizes the results in a useful way.

The figure below shows the results of visiting *rseek.org* and searching for "canonical correlation". The left side of the page shows general results for search R sites. The right side is a tabbed display that organizes the search results into several categories:

- Introductions
- Task Views
- Support Lists
- Functions
- Books
- Blogs
- Related Tools



If you click on the Introductions tab, for example, you'll find tutorial material. The Task Views tab will show any Task View that mentions your search term. Likewise, clicking on

Functions will show links to relevant R functions. This is a good way to zero in on search results.

Finding Relevant Functions and Packages

There are more than 2,000 packages available for free download from CRAN. Each package has a summary page with a short description and links to the package documentation. Once you've located a potentially interesting package, you would typically click on the "Reference manual" link to view the PDF documentation with full details.

- Visit the list of task views at *http://cran.r-project.org/web/views/*. Find and read the task view for your area, which will give you links to and descriptions of relevant packages. Or visit *http://rseek.org*, search by keyword, click on the Task Views tab, and select an applicable task view.
  - Visit crantastic and search for packages by keyword.
- To find relevant functions, visit *http://rseek.org*, search by name or keyword, and click on the Functions tab.

Sometimes you simply have a generic interest—such as Bayesian analysis, econometrics, optimization, or graphics. CRAN contains a set of *task view* pages describing packages that may be useful. A task view is a great place to start since you get an overview of what's available. You can see the list of task view pages at *http://cran.r-project.org/web/views/* or search for them as described in the Solution.

Suppose you happen to know the name of a useful package—say, by seeing it mentioned online. A complete, alphabetical list of packages is available at *http://cran.r-project.org/web/packages/* with links to the package summary pages.

Printing Something

If you simply enter the variable name or expression at the command prompt, R will print its value. Use the print function for generic printing of any object. Use the cat function for producing custom formatted output.

It's very easy to ask R to print something: just enter it at the command prompt:

```
> pi
[1] 3.141593
> sqrt(2)
[1] 1.414214
```

When you enter expressions like that, R evaluates the expression and then implicitly calls the print function. So the previous example is identical to this:

```
> print(pi)
[1] 3.141593
> print(sqrt(2))
[1] 1.414214
```

The beauty of print is that it knows how to format any R value for printing, including structured values such as matrices and lists:

```
> print(matrix(c(1,2,3,4), 2, 2))
     [,1] [,2] [1,]    1    3 [2,]    2    4
> print(list("a","b","c"))
[[1]] [1] "a"

[[2]] [1] "b"

[[3]] [1] "c"
```

This is useful because you can always view your data: just print it. You needn't write special printing logic, even for complicated data structures.

The print function has a significant limitation, however: it prints only one object at a time.

Trying to print multiple items gives this mind-numbing error message:

```
> print("The zero occurs at", 2*pi, "radians.")
Error in print.default("The zero occurs at", 2 * pi, "radians.") :
                unimplemented type 'character' in 'asLogical'
```

The only way to print multiple items is to print them one at a time, which probably isn't what you want:

```
> print("The zero occurs at"); print(2*pi); print("radians")
[1] "The zero occurs at" [1] 6.283185
[1] "radians"
```

The cat function is an alternative to print that lets you combine multiple items into a continuous output:

```
> cat("The zero occurs at", 2*pi, "radians.", "/n")
The zero occurs at 6.283185 radians.
```

Notice that cat puts a space between each item by default. You must provide a newline character (\n) to terminate the line.

The cat function can print simple vectors, too:

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> cat("The first few Fibonacci numbers are:", fib, ".../n")
The first few Fibonacci numbers are: 0 1 1 2 3 5 8 13 21 34 ...
```

Using cat gives you more control over your output, which makes it especially useful in R scripts. A serious limitation, however, is that it cannot print compound data struc- tures such as matrices and lists. Trying to cat them only produces another mind- numbing message:

```
> cat(list("a","b","c"))
Error in cat(list(...), file, sep, fill, labels, append) :
argument 1 (type 'list') cannot be handled by 'cat'
```

Setting Variables

Use the assignment operator (<-). There is no need to declare your variable first:

```
> x <- 3
```

Using R in "calculator mode" gets old pretty fast. Soon you will want to define variables and save values in them. This reduces typing, saves time, and clarifies your work.

There is no need to declare or explicitly create variables in R. Just assign a value to the name and R will create the variable:

```
> x <- 3
> y <- 4
> z <- sqrt(x^2 + y^2)
> print(z)
[1] 5
```

Notice that the assignment operator is formed from a less-than character (<) and a hyphen (-) with no space between them.

When you define a variable at the command prompt like this, the variable is held in your *workspace*. The workspace is held in the computer's main memory but can be saved to disk when you exit from R. The variable definition remains in the workspace until you remove it.

R is a *dynamically typed language*, which means that we can change a variable's data type at will. We could set x to be numeric, as just shown, and then turn around and immediately overwrite that with (say) a vector of character strings. R will not complain:

```
> x <- 3
> print(x)
[1] 3
> x <- c("fee", "fie", "foe", "fum")
> print(x)
[1] "fee" "fie" "foe" "fum"
```

In some R functions you will see assignment statements that use the strange-looking assignment operator <<-:

```
x <<- 3
```

That forces the assignment to a global variable rather than a local variable.

R also supports two other forms of assignment statements. A single equal sign (=) can be used as an assignment operator at the command prompt. A rightward assignment operator (->) can be used anywhere the leftward assignment operator (<-) can be used:

```
> foo = 3
> print(foo)
[1] 3
> 5 -> fum
> print(fum)
[1] 5
```

These forms should be avoided. The equals-sign assignment is easily confused with the test for equality. The rightward assignment is just too unconventional and, worse, becomes difficult to read when the expression is long.

Listing Variables

Use the ls function. Use ls.str for more details about each variable.

The ls function displays the names of objects in your workspace:

```
> x <- 10
> y <- 50
> z <- c("three", "blind", "mice")
> f <- function(n,p) sqrt(p*(1-p)/n)
> ls()
[1] "f" "x" "y" "z"
```

Notice that ls returns a vector of character strings in which each string is the name of one variable or function. When your workspace is empty, ls returns an empty vector, which produces this puzzling output:

```
> ls()
character(0)
```

That is R's quaint way of saying that ls returned a zero-length vector of strings; that is, it returned an empty vector because nothing is defined in your workspace.

If you want more than just a list of names, try ls.str; this will also tell you something about each variable:

```
> ls.str()
f : function (n, p)
x :  num 10 y :  num 50
z :  chr [1:3] "three" "blind" "mice"
```

The function is called ls.str because it is both listing your variables and applying the str function to them, showing their structure.

Ordinarily, ls does not return any name that begins with a dot (.). Such names are considered hidden and are not normally of interest to users. (This mirrors the Unix convention of not listing files whose names begin with dot.) You can force ls to list everything by setting the all.names argument to TRUE:

```
> .hidvar <- 10
> ls()
[1] "f" "x" "y" "z"
> ls(all.names=TRUE)
[1] ".hidvar" "f"       "x"       "y"       "z"
```

Deleting Variables

You want to remove unneeded variables or functions from your workspace or to erase its contents completely, use the rm function.

Your workspace can get cluttered quickly. The rm function removes, permanently, one or more objects from the workspace:

```
> x <- 2*pi
> x
```

```
[1] 6.283185
> rm(x)
> x
Error: object "x" not found
```

There is no “undo”; once the variable is gone, it’s gone. You

can remove several variables at once:

```
> rm(x,y,z)
```

You can even erase your entire workspace at once. The rmfunction has a listargument consisting of a vector of names of variables to remove. Recall that the lsfunction returns a vector of variables names; hence you can combine rmand lsto erase everything:

```
> ls()
[1] "f" "x" "y" "z"
> rm(list=ls())
> ls()
character(0)
```

Creating a Vector

Use the c(...)operator to construct a vector from given values.

Vectors are a central component of R, not just another data structure. A vector can contain either numbers, strings, or logical values but not a mixture.

The c(...)operator can construct a vector from simple elements:

```
> c(1,1,2,3,5,8,13,21)
[1]  1  1  2  3  5  8 13 21
> c(1*pi, 2*pi, 3*pi, 4*pi)
[1]  3.141593  6.283185  9.424778 12.566371
> c("Everyone", "loves", "stats.")
[1] "Everyone" "loves"    "stats."
> c(TRUE,TRUE,FALSE,TRUE)
[1]  TRUE  TRUE FALSE  TRUE
```

If the arguments to c(...)are themselves vectors, it flattens them and combines them into one single vector:

```
> v1 <- c(1,2,3)
> v2 <- c(4,5,6)
> c(v1,v2)
[1] 1 2 3 4 5 6
```

Vectors cannot contain a mix of data types, such as numbers and strings. If you create a vector from mixed elements, R will try to accommodate you by converting one of them:

```
> v1 <- c(1,2,3)
> v3 <- c("A","B","C")
> c(v1,v3)
[1] "1" "2" "3" "A" "B" "C"
```

Here, the user tried to create a vector from both numbers and strings. R converted all the numbers to strings before creating the vector, thereby making the data elements compatible.

Technically speaking, two data elements can coexist in a vector only if they have the same *mode*. The modes of 3.1415and "foo"are numeric and character, respectively:

```
> mode(3.1415)
[1] "numeric"
> mode("foo")
[1] "character"
```

Those modes are incompatible. To make a vector from them, R converts 3.1415 to character mode so it will be compatible with "foo":

```
> c(3.1415, "foo")
[1] "3.1415" "foo"
> mode(c(3.1415, "foo"))
[1] "character"
```

Computing Basic Statistics

You want to calculate basic statistics: mean, median, standard deviation, variance, correlation, or covariance.

Use one of these functions as applies, assuming that x and y are vectors:

- mean(x)
- median(x)
- sd(x)
- var(x)
- cor(x, y)
- cov(x, y)

Standard deviation and other basic statistics are calculated by simple functions. Ordinarily, the function argument is a vector of numbers and the function returns the calculated statistic:

```
> x <- c(0,1,1,2,3,5,8,13,21,34)
> mean(x)
[1] 8.8
> median(x)
[1] 4
> sd(x)
[1] 11.03328
> var(x)
[1] 121.7333
```

The sd function calculates the sample standard deviation, and var calculates the sample variance.

The cor and cov functions can calculate the correlation and covariance, respectively, between two vectors:

```
> x <- c(0,1,1,2,3,5,8,13,21,34)
> y <- log(x+1)
> cor(x,y)
[1] 0.9068053
> cov(x,y)
[1] 11.49988
```

All these functions are picky about values that are not available (NA). Even one NA value in the vector argument causes any of these functions to return NA or even halt altogether with a cryptic error:

```
> x <- c(0,1,1,2,3,NA)
> mean(x)
[1] NA
> sd(x)
[1] NA
```

It's annoying when R is that cautious, but it is the right thing to do. You must think carefully about your situation. Does an NA in your data invalidate the statistic? If yes, then R is doing the right thing. If not, you can override this behavior by setting na.rm=TRUE, which tells R to ignore the NA values:

```
> x <- c(0,1,1,2,3,NA)
> mean(x, na.rm=TRUE)
[1] 1.4
> sd(x, na.rm=TRUE)
[1] 1.140175
```

A beautiful aspect of mean and sd is that they are smart about data frames. They understand that each column of the data frame is a different variable, so they calculate their statistic for each column individually. This example calculates those basic statistics for a data frame with three columns:

```
> print(dframe)
      small     medium      big
1 0.6739635 10.526448  99.83624
2 1.5524619  9.205156 100.70852
3 0.3250562 11.427756  99.73202
```

```
4  1.2143595  8.533180  98.53608
5  1.3107692  9.763317 100.74444
6  2.1739663  9.806662  98.58961
7  1.6187899  9.150245 100.46707
8  0.8872657 10.058465  99.88068
9  1.9170283  9.182330 100.46724
10 0.7767406  7.949692 100.49814
> mean(dframe)
     small     medium        big
1.245040  9.560325 99.946003
> sd(dframe)
small     medium        big
0.5844025 0.9920281 0.8135498
```

Notice that `mean` and `sd` both return three values, one for each column defined by the data frame. (Technically, they return a three-element vector whose `names` attribute is taken from the columns of the data frame.)

The `var` function understands data frames, too, but it behaves quite differently than do `mean` and `sd`. It calculates the covariance between the columns of the data frame and returns the covariance matrix:

```
> var(dframe)
              small      medium         big small    0.34152627 -
0.21516416 -0.04005275 medium -0.21516416    0.98411974 -
0.09253855 big   -0.04005275 -0.09253855 0.66186326
```

Likewise, if `x` is either a data frame or a matrix, then `cor(x)` returns the correlation matrix and `cov(x)` returns the covariance matrix:

```
> cor(dframe)
              small      medium         big small    1.00000000 -
0.3711367 -0.08424345 medium -0.37113670    1.0000000 -
0.11466070 big   -0.08424345 -0.1146607 1.00000000
> cov(dframe)
              small      medium         big small    0.34152627 -
0.21516416 -0.04005275 medium -0.21516416    0.98411974 -
0.09253855 big   -0.04005275 -0.09253855 0.66186326
```

Creating Sequences

If you want to create a sequence of numbers. Use an `n:m` expression to create the simple sequence *n*, *n*+1, *n*+2, ..., *m*:

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

Use the `seq` function for sequences with an increment other than 1:

```
> seq(from=1, to=5, by=2)
[1] 1 3 5
```

Use the `rep` function to create a series of repeated values:

```
> rep(1, times=5)
[1] 1 1 1 1 1
```

The colon operator (`n:m`) creates a vector containing the sequence *n*, *n*+1, *n*+2, ..., *m*:

```
> 0:9
[1] 0 1 2 3 4 5 6 7 8 9
> 10:19
[1] 10 11 12 13 14 15 16 17 18 19
> 9:0
[1] 9 8 7 6 5 4 3 2 1 0
```

Observe that R was clever with the last expression (`9:0`). Because 9 is larger than 0, it counts backward from the starting to ending value.

The colon operator works for sequences that grow by 1 only. The `seq` function also builds sequences but supports an optional third argument, which is the increment:

```
> seq(from=0, to=20)
[1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> seq(from=0, to=20, by=2)
[1]  0  2  4  6  8 10 12 14 16 18 20
> seq(from=0, to=20, by=5)
```

```
[1]  0  5 10 15 20
```

Alternatively, you can specify a length for the output sequence and then R will calculate the necessary increment:

```
> seq(from=0, to=20, length.out=5)
[1]  0  5 10 15 20
> seq(from=0, to=100, length.out=5)
[1]  0  25  50  75 100
```

The increment need not be an integer. R can create sequences with fractional increments, too:

```
> seq(from=1.0, to=2.0, length.out=5)
[1] 1.00 1.25 1.50 1.75 2.00
```

For the special case of a "sequence" that is simply a repeated value you should use the rep function, which repeats its first argument:

```
> rep(pi, times=5)
[1] 3.141593 3.141593 3.141593 3.141593 3.141593
```

Comparing Vectors

You want to compare two vectors or you want to compare an entire vector against a scalar.

The comparison operators (==, !=, <, >, <=, >=) can perform an element-by-element comparison of two vectors. They can also compare a vector's element against a scalar. The result is a vector of logical values in which each value is the result of one element- wise comparison.

R has two *logical values*, TRUE and FALSE. These are often called *Boolean values* in other programming languages.

The comparison operators compare two values and return TRUE or FALSE, depending upon the result of the comparison:

```
> a <- 3
> a == pi     # Test for equality
[1] FALSE
> a != pi     # Test for inequality
[1] TRUE
> a < pi
[1] TRUE
> a > pi
[1] FALSE
> a <= pi
[1] TRUE
> a >= pi
[1] FALSE
```

You can experience the power of R by comparing entire vectors at once. R will perform an element-by-element comparison and return a vector of logical values, one for each comparison:

```
> v <- c( 3, pi,  4)
> w <- c(pi, pi, pi)
> v == w                    # Compare two 3-element vectors
[1] FALSE  TRUE FALSE # Result is a 3-element vector
> v != w
[1]  TRUE FALSE  TRUE
> v < w
[1]  TRUE FALSE FALSE
> v <= w
[1]  TRUE  TRUE FALSE
> v > w
[1] FALSE FALSE  TRUE
> v >= w
[1] FALSE  TRUE  TRUE
```

You can also compare a vector against a single scalar, in which case R will expand the scalar to the vector's length and then perform the element-wise comparison. The pre- vious example can be simplified in this way:

```
> v <- c(3, pi, 4)
> v == pi                    # Compare a 3-element vector against one number
[1] FALSE  TRUE FALSE
> v != pi
[1]  TRUE FALSE  TRUE
```

After comparing two vectors, you often want to know whether *any* of the comparisons were true or whether *all* the comparisons were true. The anyand allfunctions handle those tests. They both test a logical vector. The anyfunction returns TRUEif any element of the vector is TRUE. The allfunction returns TRUEif all elements of the vector are TRUE:

```
> v <- c(3, pi, 4)
> any(v == pi)          # Return TRUE if any element of v equals pi
[1] TRUE
> all(v == 0)           # Return TRUE if all elements of v are zero
[1] FALSE
```

Selecting Vector Elements

You want to extract one or more elements from a vector.

Select the indexing technique appropriate for your problem:

- Use square brackets to select vector elements by their position, such as v[3]for the third element of v.
  - Use negative indexes to exclude elements.
  - Use a vector of indexes to select multiple values.
  - Use a logical vector to select elements based on a condition.
  - Use names to access named elements.

Selecting elements from vectors is another powerful feature of R. Basic selection is handled just as in many other programming languages—use square brackets and a simple index:

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> fib
[1]  0  1  1  2  3  5  8 13 21 34
> fib[1]
[1] 0
> fib[2]
[1] 1
> fib[3]
[1] 1
> fib[4]
[1] 2
> fib[5]
[1] 3
```

Notice that the first element has an index of 1, not 0 as in some other programming languages.

A cool feature of vector indexing is that you can select multiple elements at once. The index itself can be a vector, and each element of that indexing vector selects an element from the data vector:

```
> fib[1:3]        # Select elements 1 through 3 [1] 0 1 1
> fib[4:9]        # Select elements 4 through 9 [1]  2  3  5  8 13 21
```

An index of 1:3 means select elements 1, 2, and 3, as just shown. The indexing vector needn't be a simple sequence, however. You can select elements anywhere within the data vector—as in this example, which selects elements 1, 2, 4, and 8:

```
> fib[c(1,2,4,8)]
[1]  0  1  2 13
```

R interprets negative indexes to mean *exclude* a value. An index of −1, for instance, means exclude the first value and return all other values:

```
> fib[-1]          # Ignore first element
[1]  1  1  2  3  5  8 13 21 34
```

This method can be extended to exclude whole slices by using an indexing vector of negative indexes:

```
> fib[1:3]         # As before
[1] 0 1 1
> fib[-(1:3)]      # Invert sign of index to exclude instead of select
[1]  2  3  5  8 13 21 34
```

Another indexing technique uses a logical vector to select elements from the data vector. Everywhere that the logical vector is TRUE, an element is selected:

```
> fib < 10                # This vector is TRUE wherever fib is less than 10 [1]  TRUE  TRUE  TRUE
TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
> fib[fib < 10]           # Use that vector to select elements less than 10 [1] 0 1 1 2 3 5 8
```

```
> fib %% 2 == 0          # This vector is TRUE wherever fib is even
[1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
> fib[fib %% 2 == 0]     # Use that vector to select the even elements
[1]  0  2  8 34
```

Ordinarily, the logical vector should be the same length as the data vector so you are clearly either including or excluding each element.

By combining vector comparisons, logical operators, and vector indexing, you can per- form powerful selections with very little R code:

Select all elements greater than the median

```
v[ v > median(v) ]
```

Select all elements in the lower and upper 5%

```
v[ (v < quantile(v,0.05)) | (v > quantile(v,0.95)) ]
```

Select all elements that exceed ±2 standard deviations from the mean

```
v[ abs(v-mean(v)) > 2*sd(v) ]
```

Select all elements that are neither NA *nor* NULL

```
v[ !is.na(v) & !is.null(v) ]
```

One final indexing feature lets you select elements by name. It assumes that the vector has a namesattribute, defining a name for each element. This can be done by assigning a vector of character strings to the attribute:

```
> years <- c(1960, 1964, 1976, 1994)
> names(years) <- c("Kennedy", "Johnson", "Carter", "Clinton")
> years
Kennedy Johnson  Carter Clinton
     1960     1964     1976     1994
```

Once the names are defined, you can refer to individual elements by name:

```
> years["Carter"]
Carter
1976
> years["Clinton"]
Clinton
1994
```

This generalizes to allow indexing by vectors of names: R returns every element named in the index:

```
> years[c("Carter","Clinton")]
Carter Clinton
1976     1994
```

Performing Vector Arithmetic

The usual arithmetic operators can perform element-wise operations on entire vectors. Many functions operate on entire vectors, too, and return a vector result.

Vector operations are one of R's great strengths. All the basic arithmetic operators can be applied to pairs of vectors. They operate in an element-wise manner; that is, the operator is applied to corresponding elements from both vectors:

```
> v <- c(11,12,13,14,15)
> w <- c(1,2,3,4,5)
> v + w
[1] 12 14 16 18 20
> v - w
[1] 10 10 10 10 10
> v * w
[1] 11 24 39 56 75
> v / w
[1] 11.000000  6.000000  4.333333  3.500000  3.000000
> w ^ v
[1]           1           4096        1594323     268435456 30517578125
```

Observe that the length of the result here is equal to the length of the original vectors. The reason is that each element comes from a pair of corresponding values in the input vectors.

If one operand is a vector and the other is a scalar, then the operation is performed between every vector element and the scalar:

```
> w
[1] 1 2 3 4 5
> w + 2
[1] 3 4 5 6 7
> w - 2
[1] -1  0  1  2  3
> w * 2
[1]  2  4  6  8 10
> w / 2
[1] 0.5 1.0 1.5 2.0 2.5
> w ^ 2
[1]  1  4  9 16 25
> 2 ^ w
[1]  2  4  8 16 32
```

For example, you can recenter an entire vector in one expression simply by subtracting the mean of its contents:

```
> w
[1] 1 2 3 4 5
> mean(w)
[1] 3
> w - mean(w)
[1] -2 -1  0  1  2
```

Likewise, you can calculate the *z*-score of a vector in one expression: subtract the mean and divide by the standard deviation:

```
> w
[1] 1 2 3 4 5
> sd(w)
[1] 1.581139
> (w - mean(w)) / sd(w)
[1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
```

Yet the implementation of vector-level operations goes far beyond elementary arith- metic. It pervades the language, and many functions operate on entire vectors. The functions sqrt and log, for example, apply themselves to every element of a vector and return a vector of results:

```
> w
[1] 1 2 3 4 5
> sqrt(w)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
> log(w)
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
> sin(w)
[1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243
```

There are two great advantages to vector operations. The first and most obvious is convenience. Operations that require looping in other languages are one-liners in R. The second is speed. Most vectorized operations are implemented directly in C code, so they are substantially faster than the equivalent R code you could write.

Getting Operator Precedence Right

Your R expression is producing a curious result, and you wonder if operator precedence is causing problems.

The full list of operators is shown in the table below, listed in order of precedence from highest to lowest. Operators of equal precedence are evaluated from left to right except where indicated.

Operator	Meaning

[[[	Indexing
:: :::	Access variables in a name space
$ @	Component extraction, slot extraction
^	Exponentiation (right to left)
- +	Unary minus and plus
:	Sequence creation
%*any*%	Special operators
* /	Multiplication, division
+ -	Addition, subtraction
== != < >	Comparison
!	Logical negation
& &&	Logical "and", short-circuit "and"
| ||	Logical "or", short-circuit "or"
~	Formula
-> ->>	Rightward assignment
=	Assignment (right to left)
<- <<-	Assignment (right to left)
?	Help

Getting your operator precedence wrong in R is a common problem. You may think that the expression 0:n−1will create a sequence of integers from 0 to $n - 1$ but it does not:

```
> n <− 10
> 0:n−1
[1] −1 0 1 2 3 4 5 6 7 8 9
```

It creates the sequence from −1 to $n - 1$ because R interprets it as (0:n)−1.

You might not recognize the notation %*any*% in the table. R interprets any text between percent signs (%...%) as a binary operator. Several such operators have predefined meanings:

%% Modulo operator
%/% Integer division
%*% Matrix multiplication
%in% Returns TRUE if the left operand occurs in its right operand; FALSE otherwise

You can also define new binary operators using the %...% notation; all such operators have the same precedence.

Defining a Function

Create a function by using the function keyword followed by a list of parameters and the function body. A one-liner looks like this:

function($param_1$, ..., $param_N$) *expr*

The function body can be a series of expressions, in which case curly braces should be used around the function body:

function($param_1$, ..., $param_N$) {
$expr_1$
.
.
.
$expr_M$
}

Function definitions are how you tell R, "Here's how to calculate *this*." For example, R does not have a built-in function for calculating the coefficient of variation. You can define the calculation in this way:

```
> cv <- function(x) sd(x)/mean(x)
```

```
> cv(1:10)
[1] 0.5504819
```

The first line creates a function and assigns it to cv. The second line invokes the function, using 1:10 for the value of parameter x. The function returns the value of its single- expression body, sd(x)/mean(x).

After defining a function we can use it anywhere a function is expected, such as the second argument of lapply (Recipe 6.2):

```
> cv <- function(x) sd(x)/mean(x)
> lapply(lst, cv)
```

A multiline function uses curly braces to delimit the start and end of the function body. Here is a function that implements Euclid's algorithm for computing the greatest com- mon divisor of two integers:

```
> gcd <- function(a,b) {
+   if (b == 0) return(a)
+   else return(gcd(b, a %% b))
+ }
```

R also allows *anonymous functions*; these are functions with no name that are useful for one-liners. The preceding example using cv and lapply can be shrunk to one line by using an anonymous function that is passed directly into lapply:

```
> lapply(lst, function(x) sd(x)/mean(x))
```

Return value
All functions return a value. Normally, a function returns the value of the last expression in its body. You can also use return(*expr*).

Call by value
Function parameters are "call by value"—in other words, if you change a parameter then the change is local and does not affect the caller's value.

Local variables
You create a local variable simply by assigning a value to it. When the function exits, local variables are lost.

Conditional execution
The R syntax includes an if statement. See help(Control) for details.

Loops
The R syntax also includes for loops, while loops, and repeat loops. For details, see help(Control).

Global variables
Within a function you can change a global variable by using the <<- assignment operator, but this is not encouraged.

Typing Less and Accomplishing More

You are getting tired of typing long sequences of commands and especially tired of typing the same ones over and over.

Open an editor window and accumulate your reusable blocks of R commands there. Then, execute those blocks directly from that window. Reserve the command line for typing brief or one-off commands.

When you are done, you can save the accumulated code blocks in a script file for later use.

The typical beginner to R types an expression and sees what happens. As he gets more comfortable, he types increasingly complicated expressions. Then he begins typing multiline expressions. Soon, he is typing the same multiline expressions over and over, perhaps with small variations, in order to perform his increasingly complicated calculations.

The experienced user does not often retype a complex expression. She may type the same expression once or twice, but when she realizes it is useful and reusable she will cut-

and-paste it into an editor window of the GUI. To execute the snippet thereafter, she selects the snippet in the editor window and tells R to execute it, rather than retyping it. This technique is especially powerful as her snippets evolve into long blocks of code.

On Windows and OS X, a few features of the GUI facilitate this workstyle:

To open an editor window
From the main menu, select File → New script.

To execute one line of the editor window
Position the cursor on the line and then press Ctrl-R to execute it.

To execute several lines of the editor window
Highlight the lines using your mouse; then press Ctrl-R to execute them.

To execute the entire contents of the editor window
Press Ctrl-A to select the entire window contents and then Ctrl-R to execute them; or, from the main menu, select Edit → Run all.

Copying lines from the console window to the editor window is simply a matter of copy and paste. When you exit R, it will ask if you want to save the new script. You can either save it for future reuse or discard it.

Avoiding Some Common Mistakes

Here are some easy ways to make trouble for yourself:

Forgetting the parentheses after a function invocation
You call an R function by putting parentheses after the name. For instance, this line invokes the ls function:

```
> ls()
[1] "x" "y" "z"
```

However, if you omit the parentheses then R does not execute the function. Instead, it shows the function definition, which is almost never what you want:

```
> ls
function (name, pos = -1, envir = as.environment(pos), all.names = FALSE, pattern)
{
   if (!missing(name)) {
            nameValue <- try(name)
            if (identical(class(nameValue), "try-error")) {
                name <- substitute(name)
            .
            . (etc.)
            .
```

Forgetting to double up backslashes in Windows file paths
This function call appears to read a Windows file called *F:\research\bio\assay.csv*, but it does not:

```
> tbl <- read.csv("F: \research\bio\assay.csv")
```

Backslashes (\) inside character strings have a special meaning and therefore need to be doubled up. R will interpret this file name as *F:researchbioassay.csv*, for ex- ample, which is not what the user wanted.

Mistyping "<-" as "<(blank) -"
The assignment operator is <-, with no space between the <and the -:

```
> x <- pi          # Set x to 3.1415926...
```

If you accidentally insert a space between <and -, the meaning changes completely:

```
> x < - pi         # Oops! We are comparing x instead of setting it!
```

This is now a comparison (<) between *x* and negative π (- pi). It does not change x. If you are lucky, x is undefined and R will complain, alerting you that something is fishy:

```
> x < - pi
Error: object "x" not found
```

If x is defined, R will perform the comparison and print a logical value, TRUE or FALSE.

That should alert you that something is wrong: an assignment does not normally print anything:

```
> x <- 0          # Initialize x to zero
> x < - pi        # Oops! [1] FALSE
```

Incorrectly continuing an expression across lines

R reads your typing until you finish a complete expression, no matter how many lines of input that requires. It prompts you for additional input using the +prompt until it is satisfied. This example splits an expression across two lines:

```
> total <- 1 + 2 + 3 +          # Continued on the next line
+       4 + 5
> print(total)
[1] 15
```

Problems begin when you accidentally finish the expression prematurely, which can easily happen:

```
> total <- 1 + 2 + 3            # Oops! R sees a complete expression
>       + 4 + 5                 # This is a new expression; R prints its value
[1] 9
> print(total)
[1] 6
```

There are two clues that something is amiss: R prompted you with a normal prompt (>), not the continuation prompt (+); and it printed the value of 4 + 5.

This common mistake is a headache for the casual user. It is a nightmare for pro- grammers, however, because it can introduce hard-to-find bugs into R scripts.

Using =instead of ==

Use the double-equal operator (==) for comparisons. If you accidentally use the single-equal operator (=), you will irreversibly overwrite your variable:

```
> v == 0        # Compare v against zero
> v = 0         # Assign 0 to v, overwriting previous contents
```

Writing 1:n+1 *when you mean* 1:(n+1)

You might think that 1:n+1 is the sequence of numbers 1, 2, ..., *n*, *n* + 1. It's not. It is the sequence 1, 2, ..., *n* with 1 added to every element, giving 2, 3, ..., *n*, *n* +
1. This happens because R interprets 1:n+1 as (1:n)+1. Use parentheses to get ex- actly what you want:

```
> n <- 5
> 1:n+1
[1] 2 3 4 5 6
> 1:(n+1)
[1] 1 2 3 4 5 6
```

Getting bitten by the Recycling Rule

Vector arithmetic and vector comparisons work well when both vectors have the same length. However, the results can be baffling when the operands are vectors of differing lengths.

Installing a package but not loading it with library() *or* require()

Installing a package is the first step toward using it, but one more step is required. Use library or require to load the package into your search path. Until you do so, R will not recognize the functions or datasets in the package.

```
> truehist(x,n)
Error: could not find function "truehist"
> library(MASS)         # Load the MASS package into R
> truehist(x,n)
>
```

Writing aList[i] *when you mean* aList[[i]]*, or vice versa*

If the variable lst contains a list, it can be indexed in two ways: lst[[n]] is the *n*th element of the list; whereas lst[n] is a list whose only element is the *n*th element of lst. That's a big difference.

Using &instead of &&, or vice versa; same for |and ||

Use &and |in logical expressions involving the logical values TRUE and FALSE.

Use &&and ||for the flow-of-control expressions inside ifand whilestatements. Programmers accustomed to other programming languages may reflexively use &&and ||everywhere because “they are faster.” But those operators give peculiar results when applied to vectors of logical values, so avoid them unless that’s really what you want.

Passing multiple arguments to a single-argument function

What do you think is the value of mean(9,10,11)? No, it’s not 10. It’s 9. The mean function computes the mean of the first argument. The second and third arguments are being interpreted as other, positional arguments.

Some functions, such as mean, take one argument. Other arguments, such as max and min, take multiple arguments and apply themselves across all arguments. Be sure you know which is which.

Thinking that max*behaves like* pmax*, or that* min*behaves like* pmin

The max and min functions have multiple arguments and return one value: the maximum or minimum of all their arguments.

The pmax and pmin functions have multiple arguments but return a vector with values taken element-wise from the arguments.

Misusing a function that does not understand data frames

Some functions are quite clever regarding data frames. They apply themselves to the individual columns of the data frame, computing their result for each individual column. The meanand sdfunctions are good examples. These functions can com- pute the mean or standard deviation of each column because they understand that each column is a separate variable and that mixing their data is not sensible.

Sadly, not all functions are that clever. This includes the median, max, and minfunc- tions. They will lump together every value from every column and compute their result from the lump, which might not be what you want. Be aware of which func- tions are savvy to data frames and which are not.

Posting a question to the mailing list before searching for the answer

Don’t waste your time. Don’t waste other people’s time. Before you post a question to a mailing list or to Stack Overflow, do your homework and search the archives. Odds are, someone has already answered your question. If so, you’ll see the answer in the discussion thread for the question.