An Short Introduction to R

John Stachurski

May 27, 2014

Contents

Chapter 1

An R Primer

These notes give a quick introduction to the statistical programming language R, with a focus on econometric applications.

1.1 The R Language

1.1.1 Why R?

Perhaps it behooves me to say a few words about why you might want to learn R. R is not trivial to learn. And perhaps you like using Eviews/STATA/whatever because it's nice and simple: point and click and it's done. Who needs to learn a fancy programming language? The short answer is that computers and computing are radically changing econometrics, statistics, and data science, and opening up a world of new possibilities. Real programming skills will set you free.

Here's an example for those who need more convincing (otherwise move on to the next section). Let's say that you are interested in development and microfinance. A new development in this area is the arrival of Kiva, a nonprofit organization that allows people to make small loans over the Internet, mostly to local entrepreneurs in developing countries. Kiva has been good enough to make almost all its data freely available for download. This is done through a web-based API which returns HTTP web queries in XML or JSON (Javascript Object Notation—a text based data exchange format suitable for parsing by computers). If you want to get your hands on Kiva's data and manipulate it, you need to know a bit about computing.

The Kiva example illustrates the need for programming skills when obtaining and parsing data. The other side of the story is what gets done to the data once its stored on our computer. Once again, good programming skills are what give you the freedom to do what you want. Computers have opened up a whole new world of statistical techniques, and not all can or will be conveniently canned and packaged in a point and click interface. You will need to tell your computer what to do with a sequence of written text commands—a program.

R is a good way to jump into more computer intensive statistical techniques. It's programming language is simple and robust enough to learn easily, and descriptive enough to do almost anything you want. Either way, in this course you'll be exposed to R. Whether you're converted or not, R is well worth a look.

(Finally, if you're already a programmer, you might be wondering why I've chosen R over other programming languages popular in econometrics, such as GAUSS or Matlab. The short answer is that R is certainly as good as any other programming language for statistics, and it's also free. Moreover, it's rapidly becoming the default language of statistics within statistics departments around the world. That said, the main aim here is to teach you how to write programs for statistics and econometrics. Almost all the skills you learn in this course are portable to other languages. The important this is that you *learn how to code up your ideas*.)

1.1.2 Introducing R

What is R? The R homepage (`http://www.r-project.org/`) introduces R as a language and environment for statistical computing and graphics, designed as an open source implementation of S (the latter being a statistics language developed at Bell Laboratories. The fact that R is open source means that R is

- free as in "free beer"—it costs nothing to download
- free as in "free speech"—owned by the community, for the community

Despite being free, R is every bit as good as commercial statistical packages.

Of course R is not perfect. Someone once said that the best thing about R is that it was written by statisticians...and the worst thing about R is that it was written by statisticians. That's pretty accurate: It's a great environment to jump into and do serious analysis, but the language is a little quirky relative to some of the more elegant modern programming languages (Python, etc.). It also has a steep learning curve relative to point-and-click style environments such as Eviews and STATA.

On the other hand, R contains a complete, well-structured programming language, which allows users to tackle arbitrarily sophisticated problems. It has excellent graphics and visual presentation, combining sensible defaults with good user control. It implements a vast range of statistical functions, tests and graphical techniques, including linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, and so on. More can be found by perusing the available packages on `http://cran.r-project.org/`.

1.1.3 Getting Started

R can be downloaded from `http://www.r-project.org/`. Typically, the latest version and add on packages will be found on CRAN—the Comprehensive R Archive Network. CRAN can be accessed from the R homepage, where you will be directed to a local mirror of the archive.

A quick Google search will provide lots of information on getting R up and running. (Try searching YouTube too—at the time of writing there are some helpful videos.)

I'll assume that installation has gone fine, and you have just fired up R for the first time. You should now be greeted with information on the version number, copyright, etc., followed by a prompt like so:

```
>
```

The first thing you need to know is how to quit:

```
> q()
```

If you're prompted to save your workspace, say no for now.

The second thing you need to know is how to get help. There's an interactive help system that can be accessed as follows:

```
> help(plot)
```

or

```
> ?plot
```

You'll be presented with a manual page for the `plot` function. On my computer (which is running Linux—might be different for Windows or Mac), pressing "q" exits the manual page, and returns you to the prompt. If `help` doesn't turn anything up, you can try `help.search`:

```
> help.search("plotting")
```

Overall, the help system is fairly good, and I consult it often. However, it can be technical, so general Internet searches may be useful too.

Now we've covered help and quitting, let's learn a bit about the command interface. We are not in the land of point-and-click here, which might make some people nervous. But the command line is highly efficient once you get used to it. Here are a few tips.

First, try typing

```
> plo
```

and press the tab key twice. You'll be presented with a list of possible expansions. This is useful if you're not sure what related commands are available. Also, if the command you've begun is uniquely identified by what you've typed so far, it will be expanded by the tab key to the full command.

Once we start working with our own variables, the same expansion technique will work for them. This is particularly helpful with long names. If you have a variable called

```
interwar.consumption.in.southern.alabama
```

then typing the first few letters and tabbing out will save a lot of time.

Another useful feature of the command line is the *command history*, which is accessed via the up and down arrow keys. For example, to recall a previously entered command from the current session, press the up arrow key until it reappears. (Try it and you'll see.) You can now press enter to re-run the command as is, or edit the command and then run (use the left and right arrow keys).

1.2 Variables and Vectors

Now we have some feel for the command line, let's see how we can put R to good use. The simplest way to use R is as a fancy calculator. For example, to add 12 and 23 we type

```
> 12 + 23
```

and get the response

```
[1] 35
```

(Just ignore the `[1]` for now.) To multiply we type

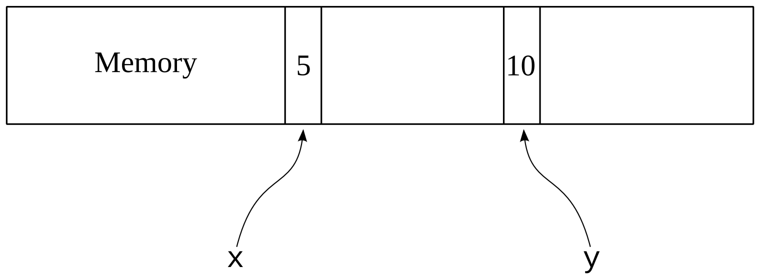

Figure 1.1: Variables stored in memory

```
> 12 * 23
```

To raise 12 to the power of 23 we type

```
> 12^23
```

and so on.

Parentheses are used in a natural way. For example:

```
> 12 * (2 + 10)
[1] 144
```

This indicates to the R interpreter that it should first add 2 and 10, and then multiply the result by 12.

So far so good, but to do anything more interesting we'll need to use *variables*

1.2.1 Variables

At the heart of any programming language is the concept of a variable. A **variable** is a name (often a symbol such as x or y) associated with a value. For example,

```
> x <- 5
```

binds the *name* `x` to the *value* 5. The value 5 is an object stored somewhere in your computers' memory, and the name `x` is associated to that little patch of memory. (The equal sign = can be used as a substitute for the assignment operator `<-`, but the latter is more common.) Figure 1.1 illustrates the idea.

Now, when you use the symbol `x`, R retrieves that value and uses it in place of `x`:

```
> x <- 5
> x
[1] 5
> x + 3
[1] 8
> x
[1] 5
> y <- x + x
> y
[1] 10
```

Notice how typing the variable name by itself causes R to return the value of the variable. Notice also that assignment works from right to left: in the statement `y <- x + x`, the R interpreter first evaluates the expression `x + x` and then binds the name `y` to the result. Understanding this is important for interpreting commands such as

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

First the r.h.s. is evaluated to obtain 10, and then the name `x` is bound to this number. Hence, the value of `x` is now 10.

As before, we are using the `*` symbol to multiply, as in `x * 2`. It cannot be omitted:

```
> x <- x 2
Error: unexpected numeric constant in "x <- x 2"
> x <- x2
Error: object 'x2' not found
```

Exponential and division are as follows:

```
> x^3
[1] 1000
> x / 3
[1] 3.333333
```

We can use `ls` to see what variables we've created so far, and `rm` if we decide that one of them should be deleted:

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

Incidentally, up until now we've used simple names for our variables, such as x and y. More complex names can be used to help us remember what our variables stand for:

```
> number_of_observations <- 200
```

or

```
> number.of.observations <- 200
```

Some rules to remember: `x1` is a legitimate variable name, but `1x` is not (variables can't start with numbers). Also, R is case sensitive (`a` and `A` are distinct names, etc.)

1.2.2 Vectors

So far, the variables we've created have been scalar-valued. Now let's create some vectors. A **vector** is an array of values such as numbers. (Actually, there are other possibilities besides numbers, as we'll see soon enough.) Vectors are important in R. If fact, the "scalar" variables we've created so far are stored internally as vectors of length 1.

```
> a <- c(2, 5, 7.3, 0, -1)
> a
[1]  2.0  5.0  7.3  0.0 -1.0
```

Here we've created a vector `a` using the `c` function, which *concatenates* the numbers 2, 5, 7.3, 0, and -1 into a vector. The resulting vector is of length 5, and we can verify this as follows:

```
> length(a)
[1] 5
```

The `c` function can also concatenate vectors:

```
> a <- c(2, 4)
> b <- c(6, 8)
> a_and_b <- c(a, b)
> a_and_b
[1] 2 4 6 8
```

On thing we often want to do is create vectors of **regular sequences**. Here's one way:

```
> b <- 1:5
> b
[1] 1 2 3 4 5
> b <- 5:1
> b
[1] 5 4 3 2 1
```

If we need to be a bit more flexible, we can use the function `seq`:

```
> b <- seq(-1, 1, length=5)
> b
[1] -1.0 -0.5  0.0  0.5  1.0
```

Try `help(seq)` to learn more. To generate a constant array, try `rep`:

```
> z <- rep(0, 5)
> z
[1] 0 0 0 0 0
```

We can also generate vectors of random variables. For example

```
> x <- rnorm(3)      # 3 draws from N(0,1)
> y <- rlnorm(30)    # 30 lognormals
> z <- runif(300)    # 300 uniforms on [0,1]
```

We'll learn more about this later on.

1.2.3 Indices

Using `[k]` after the name of a vector references the *k*-th element:

```
> a <- c(2, 5, 7.3, 0, -1)
> a[3]
[1] 7.3
> a[3] <- 100
> a
[1]   2   5 100   0  -1
```

Entering a negative index returns all but the indicated value:

```
> a <- c(2, 5, 7.3, 0, -1)
> a[-1]
[1]  5.0  7.3  0.0 -1.0
```

There are other ways to extract several elements at once. The most common is by putting a vector of integers inside the square brackets like so:

```
> a[1:4]
[1] 2.0 5.0 7.3 0.0
```

1.2.4 Vector Operations

Now let's look at some operations that we can perform on vectors. Most of these can be performed on scalar variables as well, but remember that scalar variables are just vectors of length 1 in R, so there's no need to distinguish.

To begin, consider the vector

```
> x <- 1:5    # Same as x <- c(1, 2, 3, 4, 5)
```

We can obtain the sum of all elements via

```
> sum(x)
[1] 15
```

and the minimal and maximal values via

```
> min(x)
[1] 1
> max(x)
[1] 5
```

To get the average of the values we use

```
> mean(x)
[1] 3
```

while the median is obtained by

```
> median(x)
[1] 3
```

The sample variance and standard deviation are obtained by `var(x)` and `sd(x)` respectively.

So far we've looked at functions that take a vector and return a single number. There are many others that transform the vector in question into a new vector of equal length. For example, the `log` function returns the natural log of each element of the vector:

```
> log(x)
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

This is a common pattern: The function acts *elementwise* (i.e., element by element) on its argument. Here are some more examples, returning the exponential and sine of x respectively:

```
> exp(x)
[1]   2.718282   7.389056  20.085537  54.598150 148.413159
> sin(x)
[1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243
```

Naturally, we can perform two or more operations at once:

```
> abs(cos(x))
[1] 0.5403023 0.4161468 0.9899925 0.6536436 0.2836622
> round(sqrt(x), 1)
[1] 1.0 1.4 1.7 2.0 2.2
```

Now let's look at arithmetic operations. In general, standard arithmetic operations like addition and multiplication are performed elementwise. For example

```
> a <- 1:4
> b <- 5:8
> a
[1] 1 2 3 4
> b
[1] 5 6 7 8
> a + b
[1]  6  8 10 12
> a * b
[1]  5 12 21 32
```

How about if we want to multiply each element of a by 2? One way would be to enter

```
> a * rep(2, 4)    # 4 because length(a) = 4
[1] 2 4 6 8
```

or, more generally,

```
> a * rep(2, length(a))
[1] 2 4 6 8
```

However, there's a nicer way to do it:

```
> a * 2
[1] 2 4 6 8
```

The same principle works for addition, division and so on.

1.3 Graphics

R has strong graphics capabilities when it comes to producing statistical figures. There are many different ways to create such figures. A common one is the `plot()` command. Here's an example

```
> x <- seq(-3, 3, length=200)
> y <- cos(x)
> plot(x, y)
```

This produces a *scatter plot* of x and y, as in figure 1.2. If we prefer blue we use the command `plot(x, y, col="blue")` instead, producing figure 1.3. To produce a blue line, try `plot(x, y, col="blue", type="l")`.

If you give only one vector to `plot()` it will be interpreted as a time series. For example, try

```
> x <- rnorm(40)
> plot(x, type="l")
```

1.3.1 High-Level Graphical Commands

In R, graphical commands are either "high-level" or "low-level." The function `plot()` is an example of a high-level command. Low-level commands do things like add points, lines and text to a plotting area. High-level commands are built on top of low-level commands, offering a convenient interface to the creation of common statistical figures.

The important thing to remember about creating figures in R is that the commands for a given figure should contain one and only one high-level command, followed optionally by multiple low-level commands.

Let's discuss some more examples of high-level commands. Histograms are a really common way to investigate a univariate data sample. To produce a histogram we use `hist`:

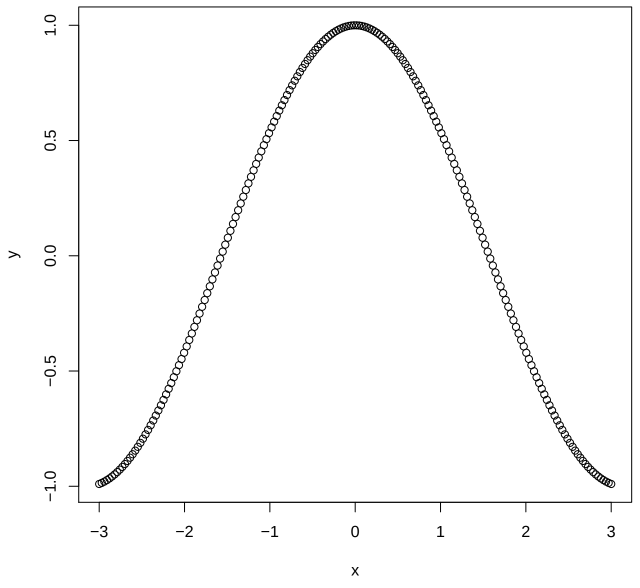

Figure 1.2: Illustration of the `plot` command

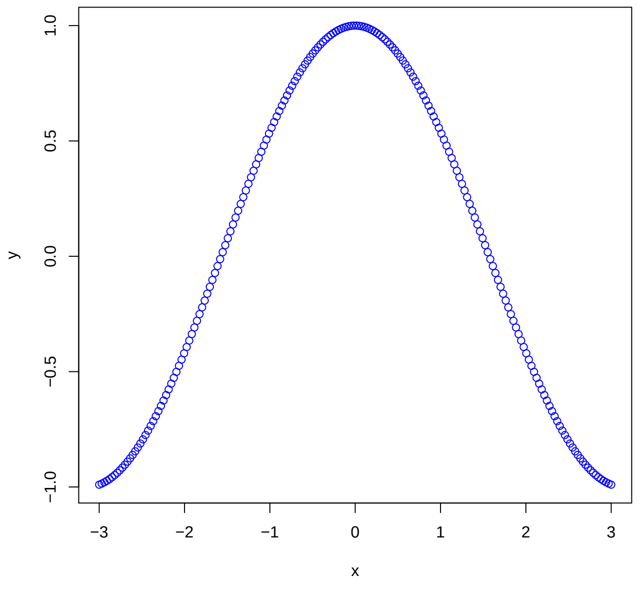

Figure 1.3: Illustration of `plot` again

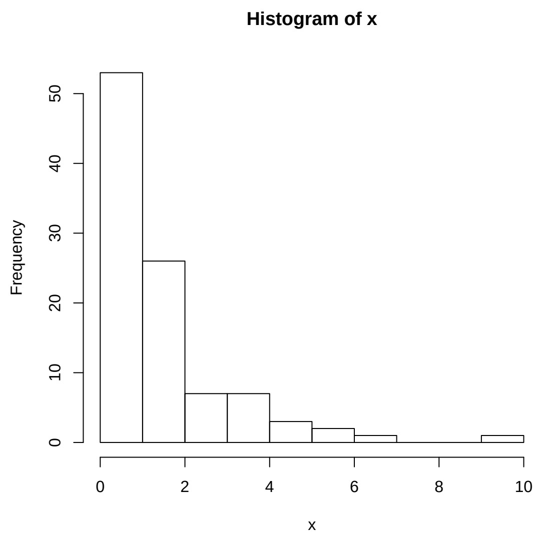

Figure 1.4: Illustration of the `hist` command

```
> x <- rlnorm(100)   # lognormal density
> hist(x)
```

The output is shown in figure 1.4.

Figure 1.5 is a bit fancier. The code for producing it is

```
> x <- rnorm(1000)
> hist(x, breaks=100, col="midnightblue")
```

If you want a background color too, then use `par`, which sets **par**ameters for the figure. See the on-line help for details (type "`?par`" without the quotes).

There are several other high-level plotting functions for presenting statistical data. For example, `barplot` produces bar plots, `boxplot` produces box plots, `pie` produces pie charts, and so on. We will meet some of them as we go along.

Still more high-level graphics functions are available if you want to dig deeper. For example, `contour` produces contour maps of 3D data, while `persp` produces 3D graphs. A 3D graph produced using `persp` is shown in figure 1.6. Details are omitted.

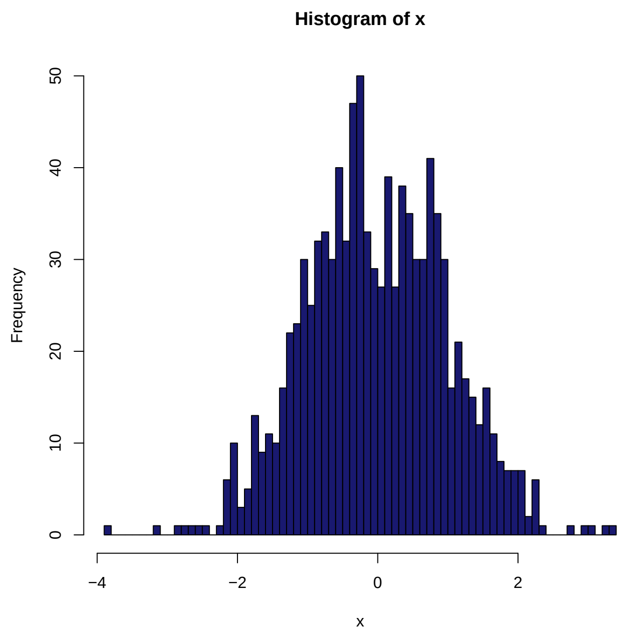

Figure 1.5: Illustration of `hist` again

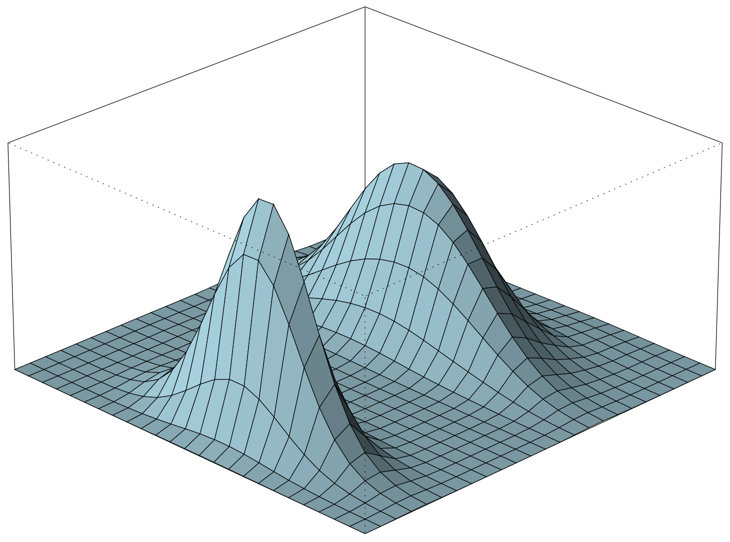

Figure 1.6: Plotting in 3D

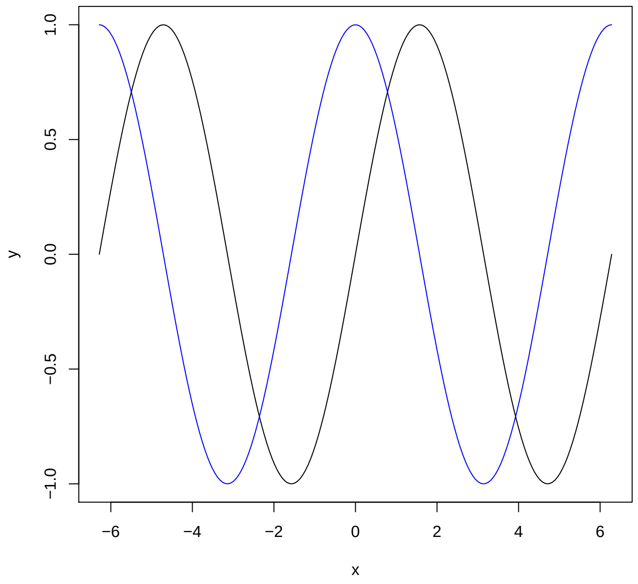

Figure 1.7: Adding points and lines

1.3.2 Low-Level Graphical Commands

High-level graphics commands are built on top of low-level commands. Low-level commands are also available to the user, and can be utilized to *add additional components to figures that were initially produced by a high-level command.* Examples include `points`, which adds points, `lines`, which adds lines described by x and y coordinates, `text`, which adds text, `abline`, which adds straight lines, `polygon`, which adds polygons (useful for filling regions), `arrows`, which adds arrows, and `legend`, which adds a legend.

The following code gives an example.

```
> x <- seq(-2 * pi, 2 * pi, length=200)
> y <- sin(x)
> z <- cos(x)
> plot(x, y, type="l")
> lines(x, z, col="blue")
```

First we call the high-level graphics command `plot`, and then the low-level command `lines`. The resulting output is shown in figure 1.7. You can experiment to see what happens if you issues these commands in the reverse order, or replace the word `lines` with `plot`.

1.3.3 Getting Hardcopy

R provides range of *graphics drivers* to produce hardcopy. For example, to produce a histogram as a PDF, try

```
> x <- rlnorm(100)
> pdf("foo.pdf")   # Write to a file called foo.pdf
> hist(x)
> dev.off()
```

The command `dev.off` signals to the graphics driver that you have finished adding components to foo.pdf. R now writes the PDF output to a file called foo.pdf in your current working directory. To see what your current working directory is, type `getwd`.

1.4 Data Types

[roadmap]

1.4.1 Basic Data Types

Like most programming languages, R can work with and perform operations on various kinds of data, such as numbers, text and Boolean values (see below). We can investigate the type of a variable by using either the `mode` or `class` function:

```
> b <- c(3, 4)
> class(b)
[1] "numeric"
> mode(b)
[1] "numeric"
```

Here both `mode` and `class` return `"numeric"`, which indicates that the elements of the vector are stored as **floating point numbers**. Floating point numbers (or floats) are a computer's approximation to real numbers.

Although `mode` and `class` returned the same answer in the previous example, we will see that this is not always the case. In general, `mode` refers to the primitive data type, whereas `class` is more specific. We'll talk more about classes below.

Another common data type is **strings**. Strings are pieces of text—any of the alphanumeric characters and other symbols on your keyboard. In R, strings have mode "character."

```
> x <- "foobar"  # Bind name x to string "foobar"
> mode(x)
[1] "character"
```

We can concatenate strings using the `paste` function:

```
> paste("foo", "bar")
[1] "foo bar"
```

By default, the two strings are separated by a space, but we can eliminate the space as follows:

```
> paste("foo", "bar", sep="")  # Separate by empty string
[1] "foobar"
```

Here's a more useful example of `paste`:

```
> paste("Today is ", date())
[1] "Today is  Mon Feb 28 11:22:11 2011"
```

In the code above, you will have noticed that, when we work with strings, we usually write them between quote marks. Why do we do this? The reason is that if we don't add quote marks, then R interprets the sequence of letters as the name of a variable. For example,

```
> x <- foobar
Error: object 'foobar' not found
```

Here the interpreter looks for the variable `foobar`, and, not finding it, issues an error message.

Here's another example. In §1.3, the command

```
> hist(x, breaks=100, col="midnightblue")
```

was used to produce figure 1.5. Here `"midnightblue"` is a string, that's passed as an argument to the function `hist`. Why can't we just type

```
> hist(x, breaks=100, col=midnightblue)  # Wrong!
```

instead? Because then R would think that `midnightblue` is a variable, and look for it in the current environment.

There are a few other basic data types besides numeric and character. One is **Boolean**, which holds the logical values TRUE and FALSE. As with other types, Boolean values can be stored in vectors. For example:

```
> x <- TRUE
> mode(x)
[1] "logical"
> x <- c(TRUE, TRUE, FALSE)
> mode(x)
[1] "logical"
```

As we'll soon see, Boolean vectors are very important in R. To save a bit of typing, you can use the abbreviations `T` and `F`:

```
> x <- c(T, T, F)
> x
[1]  TRUE  TRUE FALSE
```

One thing to remember about vectors is that in any one vector, all data must be of the same type. For example, suppose that we try to make a vector with two different modes:

```
> x <- c(1.1, "foobar")
> mode(x)
[1] "character"
> x
[1] "1.1"    "foobar"
```

We see that the numeric value `1.1` has been converted to a character string, in order to ensure all elements have the same type.

At this point, an obvious question is: What if we want to store several different data types as a single object? For example, let's say we have data on employees at a given firm, and we want to store their surname and salary together. How should we accomplish this?

In R, one way to do it is to use a **list**. A list is like a vector, except that it's elements have "names" that can be used to access them, and, in addition, there's no restriction on the data types of the elements. Here's an example, where we create a list using the `list` function:

```
> employee1 <- list(surname="Smith", salary=50000)
```

Here we've given the names `surname` and `salary` to the two elements of the list. The elements can now be accessed via these names using the "$" symbol:

```
> employee1$surname
[1] "Smith"
> employee1$salary
[1] 50000
```

If you're dealing with a list and you can't remember the names of the elements, you can extract them with `names`:

```
> names(employee1)
[1] "surname" "salary"
```

Although we won't have much cause to use the `list` function during this course, we will still be creating quite a lot of lists. The reason is that when R functions need to return a whole lot of different information, they do so using a list. For example, whenever we run a regression using the standard linear regression function `lm`, this function will return a list that contains information about the estimated coefficients, the residuals, and so on.

One final point on basic data types is that sometimes we need to test the type of a variable to make sure it will work in a given operation. If not, we may need to change its type. For this purpose, R provides a collection of `is.` and `as.` functions. Here's an example:

```
> x <- c("100", "200")
> is.numeric(x)
[1] FALSE
> sum(x)
Error in sum(x) : invalid type
> x <- as.numeric(x)  # Convert x to numeric
> is.numeric(x)
[1] TRUE
> sum(x)
[1] 300
```

1.4.2 Data Frames

In statistical programming we work a lot with data sets, reading in data, storing it, selecting subsets, making changes, and so on. In R, data is usually stored in **data frames**. Data frames are special kinds of lists that are used to store "columns" of related data. Typically, each column corresponds to observations on a particular variable. Data frames are a bit like matrices (which we'll meet later), but the columns can hold different data types (numeric, character, etc).

Let's start with a simple and light-hearted example. On the financial blog "Alphaville," Tracy Alloway noted the correlation between speaking fees paid by major financial firms to economist Lawrence Summers (Director of the White House National Economic Council) and the relative stability of their share price during the

GFC. The firms in question were Goldman Sachs, Lehman Bros, Citigroup and JP Morgan, who paid Summers speaking fees of (roughly) $140,000, $70,000, $50,000 and $70,000 respectively. Over the year to April 2009, their share prices fell by 35%, 100%, 89% and 34% respectively.

Let's record this in a data frame. We can do this in different ways. Perhaps the easiest is to first put the data in vectors

```
> fee <- c(140000, 70000, 50000, 70000)
> price <- c(-35, -100, -89, -34)
```

and then build a data frame:

```
> summers <- data.frame(fee, price)
```

As discussed above, a data frame is a list:

```
> mode(summers)
[1] "list"
```

But it's not just a list, it's a special kind of list called a data frame:

```
> class(summers)
[1] "data.frame"
```

We'll talk more about the class function in §1.4.3 below. For now, let's have a look at our data frame:

```
> summers
     fee price
1 140000   -35
2  70000  -100
3  50000   -89
4  70000   -34
```

We see that R has numbered the rows for us, and used the variable names as names for the columns. We can produce more descriptive column names as follows:

```
> summers <- data.frame(Speak.Fee=fee, Price.Change=price)
> summers
        Speak.Fee     Price.Change
1       140000         -35
2        70000        -100
3        50000         -89
4        70000         -34
```

Since summers is a kind of list, with columns of the data frame corresponding to elements of the list, the columns can be accessed using their names:

```
> summers$Speak.Fee
[1] 140000  70000  50000  70000
> summers$Price.Change
[1]   -35 -100   -89   -34
```

(Are you remembering to use the TAB key to expand these long names?) On the other hand, the column names by themselves won't work:

```
> Speak.Fee
Error: object "Speak.Fee" not found
```

This is because these variables are not part of the current workspace, but rather they are "hidden" inside the data frame. This *data encapsulation* is deliberate, and helps us things organized. It's the same idea with directories (folders) on computers: Files are grouped in different directories to keep the organized by topic. Just as we can have the same file name in different directories, we can have the same column name in different data frames.

Returning to our data frame, we can also access the data in the data frame using "matrix style" index notation. For example,

```
> summers[1, 2]  # First row, second column
[1] -35
> summers[2,]    # All of second row
    Speak.Fee Price.Change
2      70000         -100
> summers[,2]    # All of second column
[1]   -35 -100   -89   -34
```

One thing we could do to make the data frame more descriptive is to replace the row numbers with the names of the banks. This is done through the row.names function, which acts on data frames. Let's see how this works:

```
> row.names(summers)
[1] "1" "2" "3" "4"
> firm <- c("Goldman", "Lehman", "Citi", "JP Morgan")
> row.names(summers) <- firm
> row.names(summers)
[1] "Goldman"   "Lehman"    "Citi"      "JP Morgan"
```

Now the summers data frame looks as follows:

```
> summers
             Speak.Fee      Price.Change
Goldman         140000               -35
Lehman           70000              -100
Citi             50000               -89
JP Morgan        70000               -34
```

One of the nice things about data frames is that many R functions know how to interact with them directly. For example, if we enter

```
> plot(summers)
```

we immediately get a scatter plot. If we use the function `summary` we get a summary of the data:

```
> summary(summers)
    Speak.Fee     Price.Change
Min.   : 50000   Min.   :-100.00
1st Qu.: 65000   1st Qu.: -91.75
Median : 70000   Median : -62.00
Mean   : 82500   Mean   : -64.50
3rd Qu.: 87500   3rd Qu.: -34.75
Max.   :140000   Max.   : -34.00
```

Here, we plot, run a linear regression and then add the line of best fit to the plot:

```
> plot(summers)
> reg <- lm(Price.Change ~ Speak.Fee, data=summers)
> abline(reg)
```

The resulting plot in figure 1.8 shows positive correlation. (Again, this is light-hearted, don't read too much into this.) More discussion of univariate linear regression is given in §1.5.

1.4.3 A Note on Classes

In the proceeding example, we created a data frame called `summers` and ran a regression with the code

```
> reg <- lm(Price.Change ~ Speak.Fee, data=summers)
```

If you now type

```
> summary(reg)
```

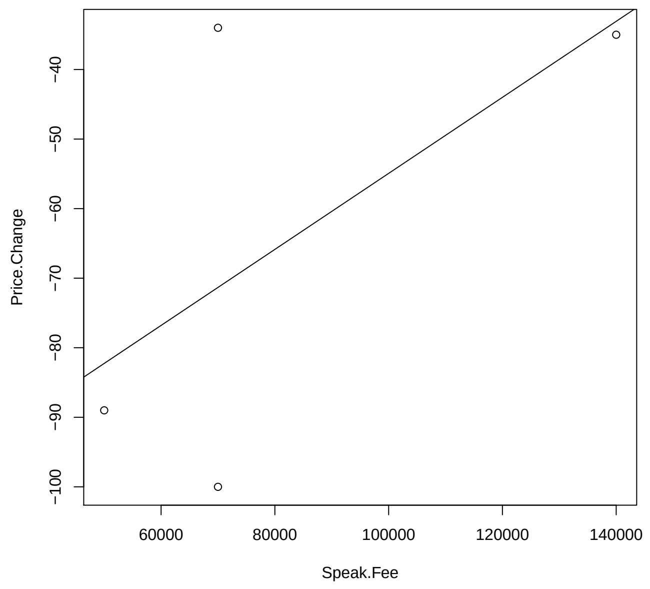

Figure 1.8: Scatter plot and line of best fit

you'll be presented with a nice table giving you estimated coefficients and other summary statistics. We'll talk more about this output in §1.5, but for now I'd like you to notice that the function `summary` was used previously in the code

```
> summary(summers)
```

which gave a basic description of the data in the data frame `summers`. What's interesting here is that we are using the same function `summary` with two very different arguments, and each time R gives an appropriate result. It's useful to know how this works.

On one hand, we can check that both the data frame `summers` and the object `reg` returned by the regression are lists:

```
> mode(reg)
[1] "list"
> mode(summers)
[1] "list"
```

However, the `summary` function needs to distinguish between these objects, so that it knows what kind of information to return. The way this is accomplished is that the two objects are given addition type information beyond their mode. This second, more specific, type information is called the objects **class**:

```
> class(reg)
[1] "lm"
> class(summers)
[1] "data.frame"
```

The function `summary`, when passed an object such as `reg`, first investigates its class. Once it knows the class of the object, it knows what action to perform. Functions like `summary`, that act differently on different objects according to their class, are called **generic functions**.

1.5 Simple Regressions in R

Let's have a quick look at the basic technique for running regressions in R.

1.5.1 The `lm` Function

To set up an example, let's generate some data:

```
> N <- 25
> x <- seq(0, 1, length=N)
> y <- 5 + 10 * x + rnorm(N)
```

The data is plotted in figure 1.9, along with the function $y = 5 + 10x$ (in blue). We can regress y on x as follows:

```
> results <- lm(y ~ x)
> class(results)
[1] "lm"
```

The function `lm` is the standard function for linear, least squares regression. It returns an object of class `lm`, which, as discussed in §1.4.3, is a kind of list. In this example, we have bound the name `results` to this list.

The list object returned by a called to `lm` includes as its elements various various other vectors and lists containing the information produced by the regression. For example, the coefficients are an element of the list:

```
> results$coefficients
(Intercept)           x
   4.491593   11.456543
```

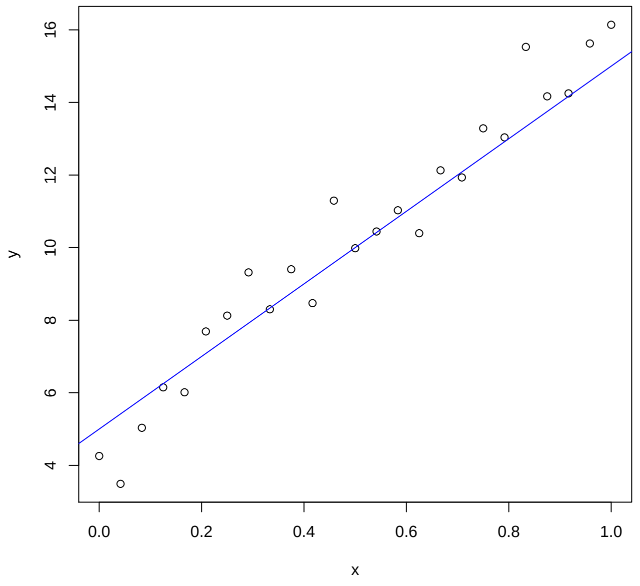

Figure 1.9: The data

To see the full list, use `names(results)`.

There are in fact various "extractor" functions for obtaining coefficients, fitted values and so on from `results`. Try, for example,

```
> coef(results)
> fitted.values(results)
> residuals(results)
```

On top of these extractor functions that obtain low level information about the output of the regression, several generic functions perform further calculations based on the results to provide information about the regression. We learned about `summary` in §1.4.3. Another such function is `abline`, which adds the regression line to a plot. In figure 1.10 we've added a regression line via the call

```
> abline(results, lty=2, lw=2)  # Dashed, double thickness
```

1.5.2 Formulas

Now let's have another look at the syntax of our call to `lm`. The argument we used was `y ~ x`. This argument is called a **formula**, which is a special syntax used for

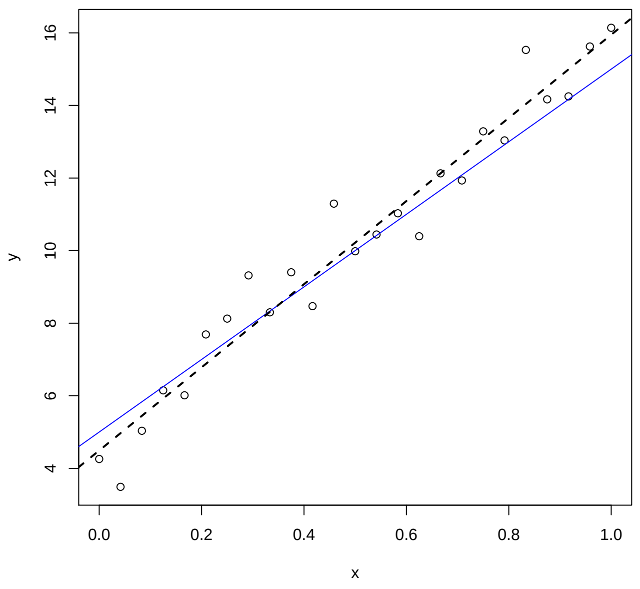

Figure 1.10: The regression line

specifying statistical models in R. Let's look at some further examples. First, if you happen to have another predictor z, then you can regress y on x and z via the call

```
> lm(y ~ x + z)
```

What's important to remember is that the "+" symbol does *not* represent addition per se—it is part of the formula, and indicates that y should be regressed on x *and* z.

In regressions such as the one above, the intercept is included by default. We can also include it explicitly via the call

```
> lm(y ~ 1 + x + z)
```

but this is exactly equivalent to the last call. To remove the intercept we can use the formula

```
> lm(y ~ 0 + x + z)  # Or lm(y ~ x + z - 1)
```

Finally, we sometimes want R to evaluate the expressions in our formulas as ordinary arithmetic operations. For example, suppose that we want to regress y on the square of x, rather than x itself. This can be achieved by the call

```
> lm(y ~ I(x^2))
```

The function `I` indicates that the operation `x^2` should be treated as an arithmetic operation, rather than be regarded as part of the formula.

Chapter 2

More R Techniques

Now let's dig a bit deeper into R. In this chapter, we'll learn about working with files, and also how to code up simple programs.

2.1 Input and Output

Writing programs and getting data in and out of R involves sending information to the screen and working with files. Let's run through the basics of how this is done.

2.1.1 The Current Working Directory

Often, when you are working with R, you will have a directory (folder) that contains files related to the project you are working on—data files, script files containing R programs, figures you have created, and so on. On the other hand, when you start R, an internal variable is initialized that stores the **current working directory**. This is the directory where R looks for any files you reference at the command line, and writes files that you create. If you are using something other than a Linux machine, these two directories will not be automatically matched. This can cause confusion.

On part of this confusion is that some operating systems use the backslash symbol \ to separate directories in a path, while others use the forward slash `/`. Regardless of your operating system, however, R uses `/`. For example, if you are using a Windows machine and have a directory called `"C:\Foobar"` on your computer, then R will refer to this directory as `"C:/Foobar"`. Please be aware of this in what follows, and convert forward slashes to backslashes as necessary.

Let's look at an example. I've created a file on my computer called "test.txt". It is a text file that contains the single line

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

The full path to the file on my Linux machine is

```
/home/john/emet_project/test.txt
```

If I start R in "/home/john" then this will be my current working directory. This can be checked with the command getwd:

```
> getwd()
[1] "/home/john"
```

Now I try to read the data in "test.txt" into a vector with the scan function:

```
> x <- scan("test.txt")
```

but receive an error message telling me the file cannot be found. The problem is that "test.txt" is not in the current working directory.

There are a few ways I can rectify this problem, including shifting the file into the current working directory, but the best solution is to change the current working directory to where the file is. This can be accomplished as follows:

```
> setwd("/home/john/emet_project/")
```

and the call to scan now succeeds:

```
> x <- scan("test.txt")
Read 10 items
> x
 [1] 10  9  8  7  6  5  4  3  2  1
```

If you want to see the contents of your current working directory, you can do so with the dir function:

```
> dir()
[1] "test.txt"
```

If I next create a figure and save it, the file will be saved in the current working directory:

```
> pdf("foo.pdf")
> plot(x)
> dev.off()
null device
          1
> dir()
[1] "foo.pdf"  "test.txt"
```

Two points to finish this section: First, it's tedious to have to manually set the current working directory every time you start R. There are work-arounds for all operating systems. Googling will tell you what you need to know. Second, there may be times when you want to read data from a file that is located somewhere on your computer, and it's easiest to locate that file by point and click. In that case, try the `file.choose` function, as in the following example:

```
x <- scan(file=file.choose())
```

On Windows machines, this should open up a dialog box which will allow you to select the file.

2.1.2 Reading Data in

We have already met the `scan` function, which is a low-level I/O function for reading data from files. The `scan` function returns either a vector or a list, depending on the contents of the file. When working with data sets, a much more common function to use is the `read.table` function, which attempts to read the data from a file into a data frame.

Let's look at an example. I've created a text file on my computer called `"testdf.txt"` with two columns of data that looks as follows:

```
X  Y
1 10
2 20
3 30
```

This file is in my current working directory, and I can read it into a data frame as follows:

```
> df <- read.table("testdf.txt", header=TRUE)
> df
  X  Y
```

```
1 1 10
2 2 20
3 3 30
> class(df)
[1] "data.frame"
```

Here I've set `header=TRUE` because the first row contains column names.

The `read.table` function has many options, which you can investigate `?read.table` . You can skip lines of crud at the start of your file using `skip`, work with comma separated values via `sep` and so on. (R can also handle many foreign data file formats. For example, R can read and write data files in the formats used by Excel, STATA, SPSS and SAS. Please look up the documentation as required.)

Another thing you can do with `read.table` and other input functions is read data directly from the Internet by giving the URL instead of a filename:

```
> read.table("http://johnstachurski.net/emet/testdf.txt",
   header=TRUE)
```

On your home installation this command should work fine. At work or at university it may not, because many office and university computers are behind firewalls, where HTTP traffic is routed through a proxy server. If R doesn't know about the proxy, then you can set it as follows:

```
Sys.setenv(http_proxy="http://user:pass@proxy:8080")
```

Here `user` is your username, `pass` is your password, and `proxy` is your proxy server. (One place to find those details is in your browser, which must be aware of the proxy server if it's working. However, you will not be able to see your password.)

If the above command still doesn't help, you can always save the file in question to your local hard disk with your browser, and then proceed as before.

2.1.3 Other I/O Functions

There are several low level functions for input and output. One is `scan`, which we met in §2.1.1. This function is quite flexible, but the most common use is the one we mentioned: Reading a sequence of numbers from a text file and converting it to a vector. The vector can then be converted into a matrix if desired—we'll talk about this process later.

If no file name is given, then `scan` reads from the screen:

```
> x <- scan()
1: 1 12 3 55 128    # Me typing numbers in
6:                  # And hitting return a second time
Read 5 items
> x
[1]   1  12   3  55 128
```

Another way to get information in is via the `readline` function. For example, if other people are going to use your program, you can get information from them along the following lines:

```
> x <- readline("Enter the value of x: ")
```

Note that the result will be a string, so you may need to call a function such as `as.numeric` if you want to convert `x` to a numeric value.

The last few functions we have discussed pertain to input. Let's talk briefly about output. To write a whole data frame, try something along the lines of

```
> write.table(summers, "summers.txt")
```

There are many options to this command, and I leave you to investigate.

A lower level function for data output is `cat`. One important use for this function is as a substitute for the `print` function, with more detailed control. For example,

```
> x <- 3
> cat("The value of x is", x, "\n")
The value of x is 3
```

The final `"\n"` tells R to end with a new line. (Try without if you're not sure what I mean.) By specifying a file name, we can also write this information to the hard disk:

```
> cat("The value of x is", x, "\n", file="test.txt")
```

2.1.4 Scripting

Another kind of data you will want to read into R is sequences of commands, or a *programs*. Suppose for example that we type the next two commands at the prompt:

```
> plot(x <- sort(rnorm(50)), type = "s")
> points(x, cex = .5, col = "dark red")
```

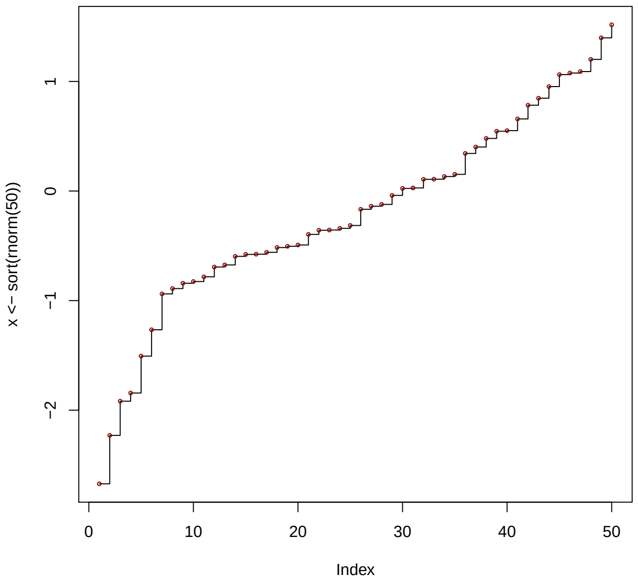

Figure 2.1: Step function

On my computer, these commands produce the graph in figure 2.1. We might be rather happy with this figure, and decide to save these commands in a file. In interpreted languages such as R, a sequence of commands saved in a file is often called a **script**. A more generic term is **program**.

There are many reasons why you'll need to write and work with scripts. As the tasks you implement become longer and more complicated, typing the commands one-by-one at the prompt becomes impractical. When you have a long sequence of commands that need to be executed in a certain order, writing a script allows to you run the whole sequence of commands easily, isolate problems, make incremental improvements and so on. The resulting program can then be shared with colleagues, etc.

When creating a script for the first time, the most important thing to remember is that scripts are saved in **text files**. Students often confuse text files with word processor documents, but they are quite different. R will not be able to read a program written in a word processor such as MS Word unless you specifically save it as a text file.[1]

[1]Yes, if you save as text then you can actually use a word processor to write scripts. But please don't. For starters, no-one will take you seriously if you tell them you code in MS Word. It's just not cool. More importantly, word processors aren't designed for the job, and they don't do it well.

Usually, text files are manipulated with **text editors**. Pretty much all computing environments supply a basic text editor. On Windows, a simple one is Notepad, while on Mac OS you can use TextEdit. On Linux there are many options. Alternatively, depending on the version of R you are running, you might find that above the R command line there are several menus. In the "File" menu, you should have the option to create a script. Selecting this option opens up a simple text editor.

Good text editors include features such as syntax highlighting, automatic indentation, code completion, and so on. These features may sound minor, but in fact they make coding much faster and more enjoyable. There are plenty of good open source text editors. By all means, google around to find out about text editors that play well with R, and install a nice one on your home computer.

Now let's say that we've written a script in some text editor and we want to run it. For a short script, the most elementary way to run it is to open up the text file with the text editor, and then copy and paste into R's interpreter (i.e., into the command line, where the > prompt is). While this is not a bad way to start off, a much faster way is to use the R function `source`, which reads in commands from a specified text file:

```
> source("foobar.R")
```

For serious scripting this second method is the only viable one. However, note that the script must be in the current working directory. See §2.1.1 for information on how to set the current working directory.

Once you are familiar with R, you will find that your work is a mix of reading in commands from programs (source files), and entering commands at the prompt (i.e., interacting directly with the interpreter). The latter is always useful for testing, getting help, and rapid prototyping of programs.

A final comment: Generally speaking, running commands via a script produces the same result as typing the commands at the prompt one by one. One exception is as follows: If I have a variable `x` and I type `x` at the prompt and hit return, I get the value of the variable. In a script run via `source`, you will need an explicit call to `print` or `cat` to get the information to the screen.

2.2 Conditions

[roadmap]

2.2.1 Comparisons

We have already met the logical values `TRUE` and `FALSE`:

```
> x <- TRUE
> x
[1] TRUE
```

(In many situations, `TRUE` and `FALSE` can be shortened to `T` and `F`.) Some expressions in R *evaluate* to either `TRUE` or `FALSE`. For example:

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

Here we're testing for strict and weak inequality respectively, using the **relational operators** > and >=. Testing for equality and inequality is as follows:

```
> 2 == 3   # Note double equal sign!!
[1] FALSE
> 2 != 3
[1] TRUE
```

The exclamation mark means "not," and reverses truth values. For example:

```
> is.numeric("foo")
[1] FALSE
> !is.numeric("foo")
[1] TRUE
```

Note the double equal sign when *testing* for equality. A single equal sign means *assignment* (i.e., is equivalent to `<-`). For example, consider:

```
> x <- 1
> x == 2   # Testing equality
[1] FALSE
> x = 2    # Assignment, equivalent to x <- 2
> x
[1] 2
```

While on the topic of greater than and less than, one interesting numerical object in R is `Inf`. `Inf` behaves much as the symbol ∞ does in comparisons, as well as arithmetic operations:

```
> 1 > Inf
[1] FALSE
> 10^100 > Inf
[1] FALSE
> Inf + Inf
[1] Inf
> Inf - Inf   #  Result is NaN (Not a Number)
[1] NaN
> 1 + Inf
[1] Inf
> 0 > -Inf
[1] TRUE
> 10 / Inf
[1] 0
```

Let's continue our discussion of the relational operators. When applied to vectors, these operators produce element by element comparisons. For example:

```
> x <- c(1, 2)
> y <- c(0, 10)
> x > y
[1]  TRUE FALSE
```

Here `x[1]` is compared against `y[1]` and `x[2]` is compared against `y[2]`.

Often, we want to compare a whole vector against a single value. For example, let's create a vector `x` and then ask which elements are greater than 1:

```
> x <- c(1, 4, 5, 9, 0)
> x > 1
[1] FALSE  TRUE  TRUE  TRUE FALSE
```

2.2.2 Boolean Operators

Relational operators can be combined with the **Boolean operators** `AND` and `OR`. To understand these operators, consider two statements `P` and `Q`, such as "2 is greater than 3," or "I live on Mars". Given statements `P` and `Q`, we can also consider the statements `P AND Q` and `P OR Q`. The statement `P AND Q` is true if both `P` and `Q` are true, and false otherwise. The statement `P OR Q` is false if both `P` and `Q` are false, and true otherwise.

In R, the operators `AND` and `OR` are represented by the symbols & and `|`respectively:

```
> 1 < 2 & 2 < 3  # AND: Both true, so true
[1] TRUE
> 1 < 2 & 2 < 1  # AND: One false, so false
[1] FALSE
> 1 < 2 | 2 < 1  # OR: One true, so true
[1] TRUE
> 1 < 2 | 2 < 3  # OR: Both true, so true
[1] TRUE
```

Try experimenting with different combinations.

The operators AND and OR can also be applied to vectors. As usual, the action is performed element by element:

```
> x <- c(1, 4, 5, 9, 0)
> x >= 5 & x <= 7  #  All x in the interval [5, 7]
[1] FALSE FALSE  TRUE FALSE FALSE
> x <= 1 | x > 5
[1]  TRUE FALSE FALSE  TRUE  TRUE
```

2.2.3 Boolean Arithmetic

As we saw earlier, the values TRUE and FALSE are primitive data types in R, of class **logical**:

```
> mode(TRUE)
[1] "logical"
```

One important property of logical values is that they *can be used in algebraic expressions*, where TRUE evaluates to one and FALSE evaluates to zero:

```
> FALSE + TRUE
[1] 1
> FALSE * TRUE
[1] 0
> sum(c(TRUE, TRUE, FALSE))
[1] 2
```

This is very handy. For example, if we want to know how many elements of a numerical vector y exceed 3, we can use the command

```
> sum(y > 3)
```

If we want to know the *fraction* of elements of y that exceed 3, we can use

```
> mean(y > 3)
```

Can you see how this works? Make sure that it's clear in your mind.

2.2.4 Conditional Extraction

One important fact regarding logical values is that vectors can be indexed by logical vectors. For example,

```
> y <- seq(2, 4, length=5)
> y
[1] 2.0 2.5 3.0 3.5 4.0
> index <- c(TRUE, FALSE, FALSE, FALSE, TRUE)
> y[index]     # Extract first and last element of y
[1] 2 4
```

This feature of vector indexing allows us to perform *conditional extraction* on vectors with very simple syntax. For example, if y is any vector, then

```
> y[y > 3]
```

returns all elements of y that exceed 3. This works because the expression inside the square brackets produces a logical vector, and only the elements of y corresponding to TRUE are extracted.

Here's an example of what we can achieve with conditional extraction. Let's suppose we have a data frame called wages, the first column of which records the sex of the individual, and the second of which records his or her salary. The first few lines of wages are as follows:

```
    sex salary
1   F   11.21
2   F   10.79
3   M    8.92
4   F   10.42
5   M    9.75
6   F    9.90
```

How can we compute the average wage for females? We can do it in one line, like so:

```
> mean(wages$salary[wages$sex=="F"])
[1] 10.03096
```

Take your time to think through how this works.

2.2.5 If-Then-Else

Next let's discuss the if-then-else construct. In the simplest case, we can combine logical expressions with the `if` statement to determine whether a piece of code should be executed or not. For example,

```
> if (2 > 3) print("foo")
```

prints nothing, while

```
> if (3 > 2) print("foo")
```

prints "foo". Conditionals are mainly used in programs, rather than at the command line. Listing 1 gives a (completely artificial) example of the full if-then-else construct, contained in a small program written in a text file. Try reproducing and then running it, and see what it does.

Listing 1 If-then-else

```
password <- readline("Enter your password: ")
if (password == "foobar") {
    print("Welcome")
    # Do something
} else {
    print("Access denied.")
    # Do something else
}
```

Often, the if-then-else syntax can be replaced by the convenient function `ifelse`. To illustrate the latter, consider the following:

```
> ifelse(1 > -1, "foo", "bar")   # Returns "foo"
[1] "foo"
> ifelse(-1 > 1, "foo", "bar")   # Returns "bar"
[1] "bar"
```

The first statement inside the brackets is evaluated. If true, the second value is returned. If false, the third value is returned.

The function `ifelse` is vectorized. For example, suppose we have a vector of data on years of schooling, including university:

```
> ys
[1] 10 12 15 12 16 17 11
```

We want to create a dummy (i.e., binary) variable in a new vector `tertiary` that has value 1 if more than 12 years of schooling has been attained (i.e., tertiary educated) and zero if not. This can be accomplished as follows:

```
> tertiary <- ifelse(ys > 12, 1, 0)
> tertiary
[1] 0 0 1 0 1 1 0
```

2.3 Repetition

The beauty of computers is that they can perform lots of small caculations quickly—much faster than a human. If a human needs to intervene and give the command for each calculation explicitly, this kind of misses the point. What we want to do is provide a set of instructions at the start, detailing all the calculations in a parsimonious way. This is done using *loops*.

2.3.1 For Loops

The most common kind of loop is a `for` loop. Suppose for example that we want to sum the integers from 1 to 100. The next piece of code performs this task:

```
> x <- 0
> for (i in 1:100) {x <- x + i}
> x
[1] 5050
```

How does this work? First `x` is set to zero. Next, `i` is stepped through each element of the vector `1:100`, and the calculation `x <- x + i` is performed at each step. Another way to get R to do this would be to write it all out in full:

```
> x <- 0
> i <- 1
> x <- x + i
> i <- 2
> x <- x + i
. . .          # Many lines omitted
> i <- 100
> x <- x + i
```

You can see that would be more than a little tedious.

The previous example loop is just for illustration. As a matter of fact, you've already learned a simpler way of performing the same calculation:

```
> sum(1:100)
[1] 5050
```

In R, the latter is more efficient than the `for` loop. We'll talk about why that's the case in §2.5.1.

Let's look at another example of a `for` loop. Suppose we want to simulate flipping a coin 1,000 times, and count the number of heads we get in the process. Listing 2 shows how we can do this using a loop. Since the program is a bit longer, it's been written in a text file. This is why we need the explicit `print` call, to get the value of `num.heads` sent to the screen.

How does it work? The variable `i` is stepped through each element of the vector `1:1000`, and the commands inside the curly brackets are performed at each step. The coin flip is simulated by drawing a uniform random number between zero and one. If the number is less than 1/2, the outcome is regarded as heads.

Listing 2 A `for` loop

```
num.heads <- 0
for (i in 1:1000) {
    b <- runif(1)
    if (b < 0.5) num.heads <- num.heads + 1
}
print(num.heads)
```

Once again, there's an easier way to do this in R. For example,

```
> sum(runif(1000) < 0.5)
```

will also do the job. (Can you see how it works?) So will

```
> rbinom(1, size=1000, prob=0.5)
```

If you're not sure about this last one, you need to read up on the binomial distribution.

For loops are very often used to step through the indices of a vector. For example, let's say that we have a numeric vector with data on firm sizes (in number of employees) called `fsize`. We want to create a new vector `ftype` that replaces these numbers with the labels S, M, and L (small, medium and large), depending on

Listing 3 Another `for` loop

```
ftype <- character(0)  # Empty character vector

for (i in 1:length(fsize)) {
    if (fsize[i] < 500) ftype[i] <- "S"
    if (fsize[i] >= 500 & fsize[i] <= 1000) ftype[i] <- "M"
    if (fsize[i] > 1000) ftype[i] <- "L"
}
```

whether the number of employees is in $[0, 500)$, $[500, 1000]$ or $[1000, \infty)$. This can be accomplished as in listing 3.

Once again, there are special functions in R that can be used to avoid this loop (see the function `cut`, for example).[2]

From the previous examples, it might seem that `for` loops are almost unnecessary in R, because there are many convenient functions that avoid the need for explicit looping. Often this is true, especially with short programs performing standard operations. However, for longer programs, explicit loops are pretty much essential. Suppose for example that we want to regress a variable such as inflation on all possible subsets of ten different regressors, and see which has the highest adjusted R squared. There are over 1,000 different possible subsets, and a `for` loop would be a natural choice to step through these possibilities.

2.3.2 While Loops

Now let's briefly look at `while` loops. Suppose we want to model flipping a coin until the time the first head appears. In other words, we want to simulate a random variable that returns the number of the flip resulting in the first head.[3] An implementation is given in listing 4. This loop continues to execute the statements inside the curly brackets until the condition `coin.face == 0` evaluates to `FALSE`. The logic of the program is illustrated in figure 2.2.

Now let's repeat this simulation 1,000 times, and calculate the average value of the random variable over these repetitions.[4] To do this, we put our `while` loop inside a

[2] Personally, I often favor explicit loops over specialized R functions, because I program in several languages, and my brain is more comfortable with generic—rather than R-specific—coding styles.

[3] This random variable has the so-called *negative binomial* distribution, which can be simulated using `rnbinom`. We'll produce our own implementation for the sake of the exercise.

[4] This gives us an estimate of the expected value—more on this later.

Listing 4 A `while` loop

```
flip.num <- 0
coin.face <- 0
while (coin.face == 0) {
    flip.num <- flip.num + 1
    b <- runif(1)
    if (b < 0.5) coin.face <- 1
}
print(flip.num)
```

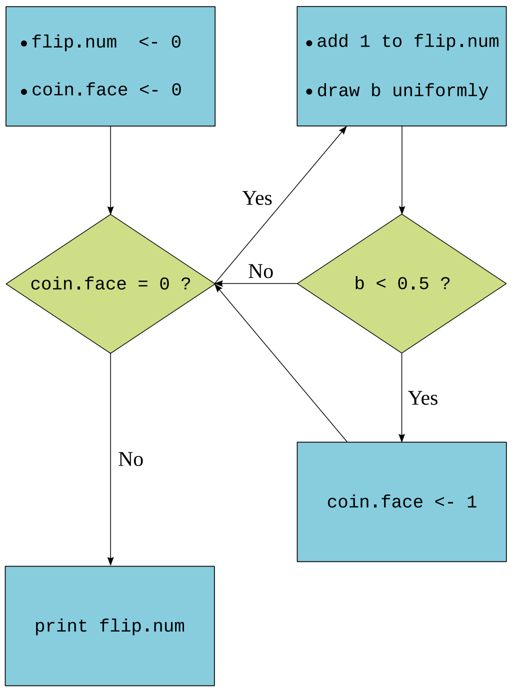

Figure 2.2: Flow chart

`for` loop, as seen in listing 5. In the listing, the second line creates an empty vector `outcomes`. As the `for` loop progresses, this vector is populated with the result of each individual simulation. At the end we take the mean and print it.

Listing 5 Loop inside a loop

```
num.repetitions <- 1000
outcomes <- rep(0, num.repetitions)  # Set up empty vector
for (i in 1:num.repetitions) {
    flip.num <- 0
    coin.face <- 0
    while (coin.face == 0) {
        flip.num <- flip.num + 1
        b <- runif(1)
        if (b < 0.5) coin.face <- 1
    }
    outcomes[i] <- flip.num  # Record result of i-th flip
}
print(mean(outcomes))
```

2.4 Functions

One aspect of programming that separates good and bad programmers is their use of functions. As the next step along our journey to becoming good programmers, let's spend some time learning the mechanics of building our own functions.

2.4.1 Built-In Functions

R provides numerous **functions** as part of the R environment, many of which we've already met. For example, `sum` is a function. It takes a vector as its **argument** (also called **parameter**), and **returns** the sum of that vector's elements:

```
> y <- sum(x)  # x is argument, y gets return value
```

Another built-in function is `integrate`, which performs numerical integration. For example,

```
> integrate(cos, -1, 5)
```

computes the integral $\int_{-1}^{5} \cos(x)dx$ numerically by evaluating the function cos at a number of points between -1 and 5, fitting a polynomial to those points, and returning the integral of this approximating polynomial. On my computer, this function call returns the output

```
-0.1174533 with absolute error < 4.3e-14
```

(You can verify that this is approximately correct by computing $-\sin(5) + \sin(-1)$.) The function integrate takes three arguments: a function representing the integrand, a number representing the lower bound of integration, and a number representing the upper bound of integration. This example illustrates the fact that R functions can take any object as an argument—a vector, a string, a data frame, or even another function.

Let's look at a third example: the plot function. Typically, this function receives at least two arguments, and often many more. Consider, for example, the function call

```
> plot(a, b, col="blue")  # a and b are vectors
```

The first two arguments are called **positional arguments**. Their meaning is determined from their order in the function call. In particular, since a appears before b, R knows that the elements of a correspond to x-axis values, while those of b correspond to y-axis values. The ordering is clearly important here, because if a and b differ, then so will the output of the call

```
> plot(b, a, col="blue")  # Order of a and b reversed
```

The argument "blue" is a **named argument**, with col being the name of the argument. Named arguments serve two purposes. First, if there are many arguments, then distinguishing by names rather than position makes it easier to remember the roles of the different arguments. Second, named arguments have a default value attached, which means that if such an argument is not supplied, the function can revert to a sensible default.

Note that we write col="blue" rather that col<-"blue". When assigning values to positional arguments, one must use = rather than <-.

2.4.2 User-Defined Functions

The functions sum and plot are examples of **built-in functions**, which are part of the R environment. It's often convenient to define our own **user-defined functions**. In fact, when we start writing longer programs, user-defined functions become almost indispensable. Further discussion of "why" is left to §2.4.4. For now let's concentrate on "how."

We begin with an example. In §2.4.1 we used the built-in function `integrate` to calculate the approximate integral of the cosine function over $[-1, 5]$ via the call

```
> integrate(cos, -1, 5)
```

Now let's suppose that, for whatever reason, we want to compute $\int_{-1}^{5} x^2 \cos(x) dx$. We can do this by creating our own *user-defined* function that represents $y = x^2 \cos(x)$ and then passing it to `integrate`. The first step is to create the function:

```
> f <- function(x) return(x * x * cos(x))
```

Here `f` is just a name that we've chosen arbitrarily, while `function(x)` indicates that we are creating a function with one argument, called `x`. The built in function `return` determines what value the function will return when we call it.

Let's test out our function to check that it works as expected:

```
> f(3)
[1] -8.909932
> 3 * 3 * cos(3)
[1] -8.909932
```

In the first line we are calling our function, using the function name followed by the argument in brackets. The return value is the right one, as we confirmed in the third and fourth line.

Simple R functions like the one we have just defined are much like the mathematical notion of a function. For example, just as $y = x^2 \cos(x)$ and $y = z^2 \cos(z)$ describe exactly the same functional relationship, the variable name `x` can be any name here. For example,

```
> f <- function(z) return(z * z * cos(z))
```

creates the same function as did the previous function definition.

Anyway, we are now ready to perform the integration, passing `f` as the first argument to `integrate`:

```
> integrate(f, -1, 5)
-18.97950 with absolute error < 2.5e-13
```

Now let's create some more complicated functions. In doing so, we can have as many arguments as we like. For example, the code

```
> f <- function() print("foo")
> f()
[1] "foo"
```

creates and calls a function with no arguments, and no specified return value.[5] For an example with two arguments, consider

```
> g <- function(x, y) return(x^2 + 2 * x * y + y^2)
> g(2, 3)
[1] 25
```

We can also create functions with named arguments:

```
> h <- function(x, intercept=0) return(intercept + 2 * x)
```

Here the statement `intercept=0` in the definition of the function `h` indicates that `intercept` is a named argument with default value 0. If the function is called without specifying a value for `intercept`, the default value will be used:

```
> h(1)
[1] 2
> h(1, intercept=3)
[1] 5
```

Note that, although functions can have many arguments, they always have just one return value (i.e., they always return a single object). If you want to send back multiple pieces of data from your user-defined function, then bundle that data into a vector or a list.

Many functions are larger than the ones we've described, with each call involving a sequence of commands. To create such a function, we enclose the commands in curly brackets. For example, the function in listing 6 packages the simulation in listing 4 in a more convenient form. Notice the curly brackets in the first and last line, which indicate the start and end of the function body respectively. Commands inside these brackets are executed at each function call.

The function in listing 6 has an argument `q`, that represents the probability of heads for our (biased) coin. The function returns the number of flips it took to obtain the first heads. Here's how the function is called:

```
> f(.01)
[1] 408
> f(.9)
[1] 1
```

Why did calling `f` with a small number return a big number?

[5] Actually, `f` returns the string `"foo"`, even though we did not specify a return value.

Listing 6 A longer function

```
f <- function(q) {    #  q = the probability of heads
    flip.num <- 0
    coin.face <- 0
    while (coin.face == 0) {
        flip.num <- flip.num + 1
        b <- runif(1)
        if (b < q) coin.face <- 1  # with prob q
    }
    return(flip.num)
}
```

2.4.3 Variable Scope

One technical issue about functions needs to be mentioned: Variables defined inside a function are **local variables**. For example, consider the following code:

```
> x <- 1
> f <- function() {x <- 2; print(x)}
> f()
[1] 2
> x
[1] 1
```

You might find it surprising to see that even after the function call `f`, which involved binding `x` to 2, when we query the value of `x` at the end we find it is still bound to 1.

Here's how it works: The first assignment `x <- 1` binds the name `x` to 1. This variable has what is called **global scope**, because it is created outside of any function. The next assignment `x <- 2` occurs inside the body of the function `f`. This variable has **local scope**. When we execute the function call `f`, the local variable `x` is created in a separate environment specific to that function call, and bound to the value 2. Any use of the name `x` inside that environment is resolved by first looking for the variable name inside this environment, and, if it is not found, then looking in the global environment. In this case, the name `x` is found in the local environment, and the local value 2 is printed.

Once execution of the function call finishes, the local environment created for that function call is destroyed, and the local `x` is lost. Execution returns to the global environment. Now, when we query the value of `x`, R looks for this variable name in the global environment. In this case, the value returned is 1.

Almost all programming languages differentiate between local and global variables. The reason is data encapsulation: If you call some function in R that implements a complex operation, that function will likely declare lots of variables that you have no prior knowledge of. It would not be a happy situation in those variable names conflicted with the variable names that you are using in your global environment.

2.4.4 Why Functions?

The single most important reason to use functions is that they break programs down into smaller logical components. Each of these logical components can be designed individually, considering only the task it must perform. This process of reducing programs to functions fits our brains well because it corresponds to the way that solve complex problems in our heads: By breaking them down into smaller pieces.

Related to this point is the fact that, because they are used to solve specific tasks, functions encourage code reuse. For example, suppose for some reason that R had no `integrate` function for numerical integration, and everyone had to write their own. This would involve an enormous duplication of effort, since numerical integration is common to many statistical problems. Moreover, if there's just one implementation that everyone uses, more time can be spent making that one implementation as good as possible.

2.5 General Programming Tips

Let's finish up our introduction to the R language by covering some general pointers for writing programs in R (and other related languages).

2.5.1 Efficiency

If you persist with statistics and econometrics, sooner or later you will bump up against the limits of what your computer can do. Computers may be getting faster, but data sets are also getting larger, and the programming problems tackled by econometricians are becoming increasingly complex. Throwing money at these problems by buying new hardware often makes little difference. If you are faced with such a problem, then you will almost always need to look at the efficiency of your code.

In this course we won't be dealing with huge data sets or enormous computational problems. However, it's worth understanding the basics of how interpreted languages like R work, in order that code can be structured appropriately.

All standard computer programs must be converted into **machine code** before they can be executed by the CPU. In a compiled language such as C, Fortran or Java, this is done in one pass of the entire program, prior to execution by the user. On the other hand, in an interpreted language such as MATLAB or R, individual commands are converted to machine code on the fly.

Once off compilation prior to execution is efficient for two reasons. First, the compiler is able to see the program as a whole, and optimize the machine code accordingly. Second, interpreted languages must pay the overhead of continually calling the machine code compiler, whereas compiled languages like C need do this only once. As a result, a language like C can be hundreds of times faster than R in certain operations.

Why don't we all just program in C then, if it's so much faster? Go ahead and try, and you will soon find out: Programming statistics in C is a painful experience. R may not be optimized for computers, but it is optimized for humans, and human time is far more valuable than computer time.

In fact, R is written mainly in C. You can think of R as a friendly interface to C, suitable for statistical calculations. The benefit is that most of the necessary C routines have been coded up for you, and all you have to do to use them is type in intuitive commands at the prompt. The cost is that you lose low-level control relative to the alternative option of hand-coding C yourself.

Even if, like most people, you decide to write statistical procedures in R rather than hand-code them in C, you can still learn from the preceding discussion. In particular, we can learn that to program R efficiently, we need to pack big batches of operations into individual commands. This allows R to pass the whole operation out to optimized machine code, pre-compiled from purpose-built C routines.

Listing 7 A slow loop

```
f <- function(x) {
    y <- numeric(length(x))
    for (i in 1:length(x)) {
        y[i] <- x[i]^2
    }
    return(y)
}
```

To illustrate these ideas, suppose we write our own naive function to obtain the square of all elements in a vector x, as in listing 7. Let's compare it against the natural, vectorized operation in R:

```
> n <- 10^6
> x <- rnorm(n)
> system.time(x^2)
   user  system elapsed
  0.024   0.004   0.046
> system.time(f(x))
   user  system elapsed
  3.412   0.012   3.423
```

We see that our function is over 100 times slower, because R blindly steps through the instructions in the `for` loop, translating into machine code as it goes. On the other hand, the vectorized method allows R to see the problem as a whole, and pass it to the compiler in an optimal way.

As a rule of thumb, vectorized calculations are far more efficient than explicit loops. When R calculations can be vectorized, the performance of R is often quite similar to that of C, Fortran or Java. When operations are not vectorized, it can be far slower.

2.5.2 Clarity and Debugging

Having talked about efficiency, it's now very important to note that *very little of your code needs to be optimized.* Often, 99% of your CPU time will be used by a tiny subset of the code that you write. Only this code needs to be optimized, and only if excessive run-time of the program justifies the extra work. Once you've spent a few days debugging your programs, it will become very clear to you that, for the vast majority of your code, clarity is the priority.

Clarity is crucial, since writing and reading programs is not an easy business for human beings. That said, there are some things you can do to make it easier on yourself and others who read your programs. One is to add comments to your programs. (A comment is a `#` symbol, followed by some text.) The text can be anything, but the idea is to make a useful comment on your code. Adding comments is helpful to others who read your code, and to you when you come back to your code after several weeks, months or years.

Another useful technique to improve the clarity of your code is to use indentation (i.e., whitespace at the start of a line of code). For example, the indentation in listings 5 and 6 helps to separate different logical parts of the program for the reader. (Like comments, this is irrelevant to the R interpreter: Whitespace is ignored.)

Despite your best efforts at clarity, however, you will still find that a lot of your programming time is spend hunting down bugs. Errors come in two main classes: **syntax errors,** which are flaws in the syntax of your instructions, and cause R to issue an error message. The other kinds of errors are **semantic errors,** which are logical mistakes that cause your program to operate incorrectly. These can be very difficult to track down, because you don't have the benefit of an error message.

Debugging is a bit of an art, and I won't say much about it, apart from suggesting that, should you have a bug in one of your programs, a good first step is to fill the program with calls to `cat` such as

```
cat("x =", x, "and y =", y, "\n")
```

so that you can keep track of your variables during execution. Although this technique for tracking your variables is not very sophisticated, it can be extremely helpful.

2.6 More Statistics

[Roadmap]

2.6.1 Distributions in R

R has handy functions for accessing all the common distributions. These functions have the form

`lettername`

where "name" is one of the named distributions in R, such as

`norm` (normal), `lnorm` (log normal), `unif` (uniform), etc.

and "letter" is one of `p` , `d`, `q` or `r`. The meanings of the letters are

`p`	cumulative distribution function
`d`	density function
`q`	quantile function
`r`	generates random variables

Here are a couple of examples:

```
> pnorm(2, mean=.1, sd=2)  # F(2), cdf of N(.1, 4)
> qcauchy(1/2)  # median of cauchy distribution
> runif(100, 2, 4)  # 100 uniform r.v.s on [2, 4]
```

See the documentation for further details on these functions.

With respect to random number generation, you should be aware that "random" numbers generated by a computer are not truly random. In fact they are not random at all—they are generated in a purely deterministic way according to specified rules. By clever design of these rules, it is possible to generate deterministic sequences the statistical properties of which resemble independent draws from common distributions. These sequences are called **pseudo random numbers.**

Pseudo random numbers follow well defined patterns determined by initial conditions. By default, these initial conditions come from the system clock and are different each time you call on the random number generator. This is a good thing if you want new draws each time. However, sometimes it's helpful to set the initial conditions, such as when you run a simulation experiment, and you want others to be able to reproduce your results.

The way to do this is via the function `set.seed`, which sets the initial conditions (seed) for the random number generator. Here's an example of usage:

```
> set.seed(123)
> runif(5)   # First draw
[1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
> runif(5)   # Draw again, without resetting seed
[1] 0.0455565 0.5281055 0.8924190 0.5514350 0.4566147
> set.seed(123)
> runif(5)   # Back to start again
[1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

2.6.2 Working with Vectors

Vectors in R are rather similar to the abstract mathematical notion of a vector we've just discussed. In particular, they have no notion of being either row vectors or column vectors. This can be observed by displaying the `dim` attribute associated with a given vector via the function `dim`. Here's an example:

```
> x <- rnorm(20)  # Create an arbitrary vector x in R
> dim(x)          # By default, x is a flat vector
NULL
```

If we wish to, we can alter the `dim` attribute to make `x` a column or row vector.

```
> dim(x) <- c(20, 1)  # Make x a column vector
> dim(x)
[1] 20  1
```

This is useful for performing matrix multiplication, an operation that distinguishes between row and column vectors.

As we saw in §1.2.4, scalar multiplication and vector addition are performed in a natural way:

```
> x <- 1:4              # Vector
> y <- 5:8              # Vector
> a <- 1                # Scalar
> b <- .5               # Scalar
> a * x                 # Scalar multiplication
> x + y                 # Vector addition
> a * x + b * y         # Both together
```

2.6.3 Working with Matrices

Let's look at how to work with matrices in R. Most often matrices are read in from data files, or created from other computations. We can also create them from scratch if we need to:

```
> A <- matrix(c(10, 20, 30, 40, 50, 60), nrow=3, ncol=2)
> A
     [,1] [,2]
[1,]   10   40
[2,]   20   50
[3,]   30   60
> B <- matrix(c(1, 2, 3, 4, 5, 6), nrow=2, ncol=3)
> B
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Alternatively, we can create B in two steps:

```
> B <- c(1, 2, 3, 4, 5, 6)
> class(B)  #  B is a flat vector with no dimension
[1] "numeric"
> dim(B) <- c(2, 3)   # Set the dimension attribute
> B
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> class(B)       # Now B is a matrix
[1] "matrix"
```

Finally, here's a third way that I use often:

```
> B <- matrix(nrow=2, ncol=3)
> B
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> B[1,] <- c(1, 3, 5)
> B[2,] <- c(2, 4, 6)
> B
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Internally, matrices in R are closely related to data frames. In particular, we can access elements of the matrices using square brackets followed by row and column numbers:

```
> A[3, 2]   #  Third row, second column
[1] 60
> A[3,]     #  Third row, all columns
[1] 30 60
> A[, 2]    #  All rows, second column
[1] 40 50 60
```

A and **B** are not conformable for addition, but $\mathbf{A}'$ and **B** are. To take the transpose of **A** we use the transpose function `t`:

```
> t(A)
     [,1] [,2] [,3]
[1,]   10   20   30
[2,]   40   50   60
```

Addition is now straightforward:

```
> t(A) + B
     [,1] [,2] [,3]
[1,]   11   23   35
[2,]   42   54   66
```

Notice that the multiplication symbol `*` gives element by element multiplication, as follows

```
> t(A) * B
     [,1] [,2] [,3]
[1,]   10   60  150
[2,]   80  200  360
```

This is natural, because it's consistent with the algebraic operations on vectors we saw earlier. Matrix multiplication is different, and uses the symbol combination `%*%`:

```
> A %*% B
     [,1] [,2] [,3]
[1,]   90  190  290
[2,]  120  260  400
[3,]  150  330  510
```

As we learned above, the product **AB** is formed by taking as it's i, j-th element the inner product of the i-th row of **A** and the j-th column of **B**. For example:

```
> A[2,] %*% B[,3]
     [,1]
[1,]  400
```

We can also check the fact that $(\mathbf{AB})' = \mathbf{B}'\mathbf{A}'$:

```
> t(A %*% B)
     [,1] [,2] [,3]
[1,]   90  120  150
[2,]  190  260  330
[3,]  290  400  510
> t(B) %*% t(A)
     [,1] [,2] [,3]
[1,]   90  120  150
[2,]  190  260  330
[3,]  290  400  510
```

Let's have a look at solving linear systems of equations. Suppose now that **A** is the matrix

```
> A
     [,1] [,2] [,3]
[1,]   58   25    3
[2,]   43   97   90
[3,]   18   32   80
```

and **b** is the column vector

```
> b
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

We are interesting in solving the system of equations $\mathbf{Ax} = \mathbf{b}$. We know that a unique solution will exist provided that $\mathbf{A}$ has nonzero determinant (and is therefore of full rank). Let's check this:

```
> det(A)
[1] 236430
```

The inverse of $\mathbf{A}$ can be calculated as follows:

```
> Ainv <- solve(A)
> Ainv
             [,1]         [,2]        [,3]
[1,]  0.020640359 -0.008053124  0.00828575
[2,] -0.007697839  0.019396862 -0.02153280
[3,] -0.001564945 -0.005946792  0.01924883
```

We can solve for $\mathbf{x}$ as $\mathbf{A}^{-1}\mathbf{b}$:

```
> x <- Ainv %*% b
> x
            [,1]
[1,]  0.02939136
[2,] -0.03350252
[3,]  0.04428795
```

While this is valid in theory, it turns out that there are more efficient ways to do this numerically. As a rule, computing inverses of matrices directly is relatively unstable numerically, in the sense that round off errors have large effects. Other techniques are available that mitigate this problem. In R, the preferred method is to use the formula `solve(A, b)`:

```
> solve(A, b)
            [,1]
[1,]  0.02939136
[2,] -0.03350252
[3,]  0.04428795
```

In this case we can see that there was no difference in the results produced by the two techniques, but for large matrices the latter method is significantly more accurate and computationally efficient.

Let's conclude this section with a few tips for working with matrices and vectors. First, suppose that we want to use a vector in a matrix operation such as matrix multiplication. Since a vector in R has no dimension attribute, how will R know whether to treat it as a column vector or a row vector? (Clearly the result will depend on which choice is made.) The answer is that R will make an educated guess, depending on which of the two choices is conformable. The next piece of code illustrates:

```
> A <- matrix(c(1, 2, 3, 4), nrow=2)
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> x <- c(5, 6)
> dim(x)   #  x is a flat vector, with no dimension
NULL
> A %*% x
     [,1]
[1,]   23
[2,]   34
> x %*% A
     [,1] [,2]
[1,]   17   39
```

In the first case x is treated as a column vector, while in the second it is treated as a row vector.

The case x %*% x is ambiguous, and R always gives the inner product.

Sometimes we need to combine matrices or vectors. Here's an example that forms a matrix by stacking two vectors row-wise and then column-wise.

```
> a <- c(1, 2, 3)
> b <- c(4, 5, 6)
> cbind(a, b)
     a b
[1,] 1 4
[2,] 2 5
[3,] 3 6
> rbind(a, b)
```

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

The function diag provides simple way to extract the diagonal elements of a matrix:

```
> A
  [,1] [,2]
[1,]    1    3
[2,]    2    4
> diag(A)
[1] 1 4
```

(The trace can now be obtained by summing.) Somewhat confusingly, the same function is also used to create diagonal matrices:

```
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Last but not least, we often need to create matrices of zeros or ones. Here's how:

```
> matrix(1, 3, 2)
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    1    1
> matrix(0, 4, 4)
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
```

[internally, matrices are just flat arrays, so some vectorized operations can be performed one-off. example, want to center each row around its mean. write function, use apply.]

2.6.4 Multiple Regression in R

Let's look briefly at running multiple regressions in R. The standard method is to use the function lm, which we discussed in §1.5. Let's now recall the method, and compare it with the result of direct calculations according to the theoretical results we have derived. As a starting point, we generate some synthetic data:

```
> set.seed(1234)
> N <- 500
> beta <- c(1, 1, 1)
> X <- cbind(rep(1, N), runif(N), runif(N))
> y <- X %*% beta + rnorm(N)
```

The seed for the random number generator has been set so that the same pseudo-random numbers are produced each time the program is run. The matrix $\mathbf{X}$ consists of a vector of ones as the first column, and then two columns of observations on non-constant regressors. Hence, $K = 3$. The coefficient vector $\boldsymbol{\beta}$ has been set to $(1, 1, 1)$ for simplicity.

We can run an OLS regression in R as follows.

```
> results <- lm(y ~ 0 + X)
```

In the call to `lm`, the term `0` indicates that a constant term should not be added—in this case, because we already set up $\mathbf{X}$ to have a constant vector as the first column.

As a first step, let's compare the estimate of the coefficient vector produced by `lm` to the one we get when using the theoretical formula directly:

```
> results$coefficients
       X1        X2        X3
0.9662670 0.9532119 1.0045879
> solve(t(X) %*% X) %*% t(X) %*% y
          [,1]
[1,] 0.9662670
[2,] 0.9532119
[3,] 1.0045879
```

Here, exactly the same results are produced. However, you should be aware that the direct method is less stable numerically. Numerical error will be zero or negligible in mosts settings, but may be significant when data sets are large and multicollinearity is present in the data.

Let's also check our theoretical methods for computing $\hat{\mathbf{y}}$ and $\hat{\mathbf{u}}$. Since these vectors are too large to print out and compare, we'll look instead at the squared norms of the vectors, which correspond to the ESS and SSR respectively:

```
> yhat <- results$fitted.values
> uhat <- results$residuals
> P <- X %*% solve(t(X) %*% X) %*% t(X)
> M <- diag(N) - P
> sum(yhat * yhat)  # Fitted values calculated by lm()
[1] 2002.222
> sum((P %*% y) * (P %*% y))  # Fitted vals, direct method
[1] 2002.222
> sum(uhat * uhat)  # Residuals calculated by lm()
[1] 458.4417
> sum((M %*% y) * (M %*% y))  # Residuals, direct method
[1] 458.4417
```

Again, the results of `lm` and the results of our direct calculations from our theoretical results are in exact agreement.

Regarding the coefficient of determination, the calculation by `lm` can be viewed using the `summary` function:

```
> summary(results)
# Some lines omitted
Multiple R-squared: 0.8137
```

According to the theory, this should agree with $\|\hat{\mathbf{y}}\|^2 / \|\mathbf{y}\|^2$. Let's check this:

```
> sum(yhat * yhat) / sum(y * y)
[1] 0.8136919
```

The results are in agreement.

2.7 Exercises

Ex. 2.7.1. How could you use `sum` to determine whether a numeric vector `x` contains the value 10?

Ex. 2.7.2. Suppose we are processing vector of zeros and ones. The vectors corresponds to the employment histories of individuals. In a given vector, 1 means that the individual was employed at the associated point in time, while 0 means unemployed. Write a function that takes such a vector (of arbitrary length) and compute the longest (consecutive) period of employment.