(Draft!) R primer

Jim Regetz

Last update: 27 Apr 2010

Contents

1 Introduction

R is a powerful software system designed for manipulating, analyzing, and graphing data. It was first created in the early 1990's by Ross Ihaka and Robert Gentleman, who sought a non-commercial statistical computating environment for their students. They begin by implementing a small interpreter in the Scheme language, but increasingly found themselves borrowing syntax and data structures of the S language. S, developed in the mid-1970's by John Chambers and others at Bell Laboratories. Today, R is regarded as an implementation of the S language, although there remain differences that reflect its Scheme origins. R was released under an open source license in 1995. An international community of users and contributors rapidly coalesced, and the rest is history.

Essentially anything you can do using other popular off-the-shelf statistical software can also be done in R. In addition, R is an integrated programming environment, allowing users to script their own functions (or modify existing ones) to do customized tasks. This provides much of the flexibility of languages such as C, but with the advantage of building upon R's high-level statistical routines, data management functions, and graphing tools. While the base installation will be sufficient for many users, dozens of downloadable "packages" have been developed for accomplishing specialized tasks, often using cutting-edge methods. R is supported by a large and active community of developers, including many ecologists and other scientists. It is highly regarded at NCEAS because it is:

Scripted Reproducible, reusable, and shareable

Multi-platform Runs on Windows, OS X, Linux, and UNIX variants

Best in class Well-vetted, robust numerical routines

Open source High transparency of underlying methods

Free Accessible to everyone

Although R certainly requires an initial learning investment, and may seem particularly intimidating if you've only ever used "point-and-click" software, the long-term payoff is significant.

1.1 R is a community

Last but certainly not least, one of the great things about R is its vibrant user and developer community. There is really no reason you should ever have to struggle through something alone in R. At some point you'll find yourself trying to make something work in R, imagining that lots of other folks have gone through the same struggle. Odds are, they have. And odds are, at least one of them asked about it or wrote about it somewhere online. Tap into that online wisdom! Then before you know it, you might even be ready to contribute to it.

Here are a few ways you can begin to tap into that community.

R-help and friends (email lists) https://stat.ethz.ch/mailman/listinfo/

R-forge (developer services) http://r-forge.r-project.org/

Planet R (blog aggregator) http://planetr.stderr.org/

and more...

Another major aspect of the R community is the explosion of contributed add-on packages for doing almost anything you can imagine. We'll talk more about these later.

1.2 R is open source

All R source code is viewable within R. Most of the time, you can see it by simply typing the name of the function (*without* parentheses) at the R console. We'll see later why this works. Other times you have to know some tricks to get the code. It's not that R is being coy, rather there are just some extra things going on under the hood. One major reason for this is that sometimes generic functions actually invoke specialized methods depending on what kind of R object it is operating on, and it can take an extra step or two to figure out which specific method you're interested in, and then access the source. Another example involves use of namespaces, which allow developers to "hide" certain internal functions not intended to be called directly by the user.

For details on how to access the code in these cases, see this archived R newsletter article: Uwe Ligges and Duncan Murdoch. *R help desk: Accessing the sources.* R News, 6(4):43-45, October 2006.

In other cases, R functions actually call compiled code written in C (or other languages). You can find all of this source code online. In fact, you can browse an online repository containing the entire source tree for the base R system here: https://svn.r-project.org/R/trunk/

Another advantage to browsing this repository is that it contains the *original* source code. The code viewed inside R has actually been parsed and reformatted, and any code comments have been removed. If you want to see them, have a look at the online repository.

Finally, all source code for user-contributed packages distributed through CRAN can also be viewed online. For example, you can download the source code (along with the package itself, a manual, and other information) for the `labdsv` package here:
http://cran.cnr.berkeley.edu/web/packages/labdsv

1.3 R is an interface

One of the wonderful things about R is that it lets you access more powerful external applications, languages, and software libraries. To give a few examples, R can be in interface to:

- low-level languages like C/C++, Fortran, Java (via JNI)
- relational databases like PostgreSQL, MySQL, Oracle, SQLite
- spatial libraries and GIS like GDAL/OGR and GRASS
- graphics rendering libraries like OpenGL, GGobi, GTK

These interfaces can provide an effective means of overcoming some of R's limitations. For example, looping constructs in R can be one or two orders of magnitude slower than in a lower level language like C. Although many of R's functions are designed to obviate the need for looping, sometimes it is unavoidable. However, if speed is a concern, one can write the computationally intensive portion of a simulation or analysis in C. R can be used to pre-process data, pass data to the compiled C routine, and retrieve the output for post-processing.

Another example stems from the fact that R runs entirely in memory. This means that the size of datasets is limited by the amount of RAM. Even worse, temporary full copies of objects in memory are often required during invocation of a function; just reading in a dataset can require 4-5x as much memory as the eventual footprint of the dataset! This actually leads to problems far less often than you might imagine, but it can certainly present a major obstacle in some cases. Direct access to relational databases can help: on can store potentially vast amounts of data in a database system, and only load relevant subsets into R for analysis at any given time.

1.4 R is a fancy calculator

At first, you can treat R like a scientific calculator. You type expressions at the command prompt, hit enter, and R tells you the answer.

```
> 5 + 8
[1] 13
> (212 - 32) * 5/9
[1] 100
> cos(pi)
[1] -1
> log(exp(1))
[1] 1
> log(0)
[1] -Inf
```

Standard arithmetic operators are used just as you would expect. For functions like `log` and `cos`, you type the function name followed by the function argument(s) contained in parentheses. We'll talk more about this later. Don't worry about the cryptic `[1]` that appears at the start of each output line - we'll get to that later, too.

Pushing numbers around like this is a useful way to do your taxes, but we won't get too far without learning about *assignment*. Assignment is the act of associating a value with a name, usually because you want to be able to easily refer to it later. By the way, to be more R-like, in general we'll use the term *object* instead of *value*. That might seem like overkill now that our examples just use single numbers, but you'll see later that we can and do create all kinds of complicated structures in R and assign them to names.

In just about every other programming language you're likely to come across, you would use an equals sign (`=`) to indicate assignment: `x=1`. R, however, has a quirky assignment operator: `<-` This is a holdover from the early days of S. Technically, you're now allowed to use `=` instead of `<-` to do assignment in R. Resist the urge. For one thing, `=` is used to pass objects into functions, as we'll discover later; once you get used to it, you'll find that R code is much easier to read when

these two operators are used in their rightful places. For another thing, using `=` for assignment in your code is like wearing a big yellow shirt that says "R NEWBIE". So please, don't do it.

There are some rules about what constitutes a valid name, but we won't go into the details now. One important point is that capitalization matters: `x` does *not* mean the same thing as `X`.

With all that out of the way, here's a very simple assignment statement:

```
> x <- 312
```

We've just told R to take a particular number (`312`) and keep it in memory somewhere associated with a particular symbol (`x`). Now `x` will refer to `312` until either:

- you assign a different object to `x`
- you remove `x` using the expression `rm(x)`
- you quit R (at which time you'll have an opportunity to save all of your named objects to disk)

Of course, once you assign something to a name, you can use that name in a subsequent expression. Indeed, that's the whole point! Here is some simple arithmetic:

```
> jenny <- 867.5309
> jennyUnlisted <- jenny - x
> jennyUnlisted
[1] 555.5309
```

Notice what happened after the last statement: R printed the value of `jennyUnlisted`. When you're working at the console, that's what R does if you just give it the name of an object. To say this more generally, whenever R evaluates a complete expression, it prints the resulting object. The name of a stored object just evaluates to the object itself, and R dutifully prints it. Incidentally, we could also explicitly tell R to print it by entering `print(jennyUnlisted)`, but in most cases it's not necessary when using R interactively. A major exception is one we've already seen: when you do an assignment, R won't bother to print anything. After all, you can always see the object later just by typing the name.

1.5 R is more than just a fancy calculator

If you use R with any regularity, you'll likely find yourself following a somewhat predictable evolutionary pathway.

1. Chatting with the interpreter
2. Typing statements in a text file, and running them one at a time or in batches
3. Writing and using reusable functions
4. Sourcing complete scripts that have a combination of function definitions and procedural sections that use those function
5. Developing packages

2 Administrative details

2.1 Installing and upgrading R

By now we've already played with some commands in R, so presumably you were able to get it up and running. But if you need some help with this step, or need a reminder, or want to know how best to upgrade to a newer version of R, best to start with the official R Installation and Administration manual.

2.2 Getting add-on packages

The main source of add-on packages is CRAN (Comprehensive R Archive Network). This resource is mirrored in dozens of places around the world, but the main location is http://cran.r-project.org.

Installing packages is easy. You can do it from within R itself, right at the console, with just a single function (and most R GUIs make it even simpler).

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

One gotcha for beginners is that installing a package is not the same as loading a package. You only need to install a package once (ignoring upgrades). But you need to *load* a package in any particular R session in order use it. This is done with the `library` function. (The related `require` function is nearly equivalent, but is more useful inside functions.)

```
> library("reshape")
```

At last check, there were more than 2300 packages on CRAN. Yikes! One lightweight form of organization are the volunteer-maintained "Task Views", which present a descriptive overview and listing of packages relevant for particular application areas and/or analytical domains: http://cran.r-project.org/web/views/

2.3 Helping yourself

Before getting too far along, you should learn how to help yourself in R. If you know the name of the function, you can simply ask R for help using the `help` function or the `?` shortcut:

```
> help("dim")    # will open help page for dim function
> ?dim    # same as above
```

Sometimes you need a little extra help to discover the function you want. In that case, `help` itself isn't enough. Try using the `help.search` function or its `??` shortcut:

```
> ?regression
No documentation for 'regression' in specified packages and libraries:
you could try '??regression'


> help.search("regression")    # will list potentially relevant functions
> ??regression    # same as above
```

You can also see a summary page for particular packages using the `help` function. An equivalent message

```
> help(package = "vegan")
```

A final source of on-board information comes in the form of *vignettes* written by package contributors. Unfortunately, they've only been written for a subset of packages, and some of them are incomplete and/or out-of-date. Nevertheless, it's worth taking a peek. You can list all available vignettes, and open a particular vignette (as a PDF) using the `vignette` function:

```
> vignette()
> vignette("vegan")
```

If internal R resources don't satisfy your hunger, there is one more thing to try inside R. The `RSiteSearch` function will magically open a web browser and return a search of the R-help mailing list archives and the R online documentation. (If you're not connected to the internet, well, tough luck.)

```
> RSiteSearch("regression")
```

Of course, sometimes you need a little something more than the quick answer; you need to do a little online research. Here are a few good places to start:

- RSeek: Google search of the R universe (http://www.rseek.org)
- Offical and unofficial R documentation (http://www.r-project.org/other-docs.html)
- R Wiki: The "official" R community wiki (http://wiki.r-project.org)
- Quick-R: A handy reference site (http://www.statmethods.net)

2.4 Get in touch with your workspace

There are a handful of functions for exploring your workspace. Mostly you'll just want to see what you've got, and perhaps sometimes remove some things just to clean up. Here are some examples.

```
> ls()
[1] "jenny"          "jennyUnlisted" "x"
> exists("jenny")
[1] TRUE
> exists("joey")
[1] FALSE
> object.size(jenny)
48 bytes
> rm(jennyUnlisted)
> ls()
[1] "jenny" "x"
```

2.5 Quitting

When the day is over and you're done with an interactive R session, you can quit using the `q()` function. Or you could use `quit()`, if you feel like being verbose. Note that you need to supply the parentheses after the function name, just like with every other function. Otherwise R will print the source of the `quit` function, and that's probably not something you want to get into just before happy hour.

When you quit, R will ask if you want to save your workspace. If you respond affirmatively, the contents of your workspace (i.e., all objects you have created) will be saved in an .RData file in your working directory, along with an .Rhistory file that remembers the expressions you've entered. If you respond negatively, everything will be discarded. This isn't necessarily a bad thing. Saving your workspace gives you the illusion of meaningful preservation, but weeks later you're unlikely to have any idea how you produced the saved objects in the first place, You'll probably have better work habits if you regard everything in the R workspace as ephemeral, and focus on keepng your raw data and scripted commands well-documented and preserved on disk. If you have some object that you specifically want to save for posterity (or that would take hours to reproduce), consider writing it to file in a text format like CSV. For complex objects, this won't do, but in that case perhaps just save the object in question itself in a meaningfully named file, using the `save` function:

```
> save(gnarlyobject, file = "simOutput-YYYYMMDD.RData")
```

3 Fun with data: A teaser

Some might call this premature, but let's jump ahead to playing with some data. That's what R is all about, right? So how do you get data *into* R? Lots of ways, actually. In fact, there is a whole R Data Import/Export manual on the topic, and even that doesn't include everything. But let's start with the basics of reading data from a delimited text file. It turns out that the vast majority of the kinds of data we tend to work with can be conveniently stored in this format, and it makes life much easier than dealing with some proprietary binary format (ahem, Excel).

The main function of interest is `read.table`. It actually accepts many possible arguments controlling the details of how it works, but conveniently R provides two special versions of this function:

```
> read.delim("a_tab_delimited_file.txt")
> read.csv("a_comma_separated_file.csv")
```

One handy feature is that R can read tabular data files directly from the web. Using a completely fabricated URL, the function call in R might look something like this:

```
> sal <- read.csv("http://sofia.usgs.gov/pubt/outgoing/salinity/dec00track.txt")
```

But for now, let's just read a local file:

```
> labdata <- read.csv("labdata.csv")
```

We just read in a CSV file, creating a special kind of R object called a data frame (more on this later). We gave it the name `labdata`. As we learned earlier, if you want R to display this object for you, just can just type the name. However, if your data frame has many rows, you might prefer just to see the first handful of them. The `head` function will show you the first 6 rows by default, but R being the accommodating program it is, you can ask for a different number of rows instead. Let's just look at the first three rows:

```
> head(labdata, 3)
  init.size treatment mean    height   weight
1     Small   Control   16 0.5658715 39.23252
2     Small   Control   16 2.5970133 72.45949
3     Small   Control   16 6.5067285 93.52200
```

Naturally, the `tail` function displays the bottom rows.

```
> tail(labdata, 3)
   init.size treatment mean    height    weight
88     Large      High    8  4.859559  95.30480
89     Large      High    8 11.412196 114.55008
90     Large      High    8  7.113136  84.27726
```

Or how about a quick summary of the data?

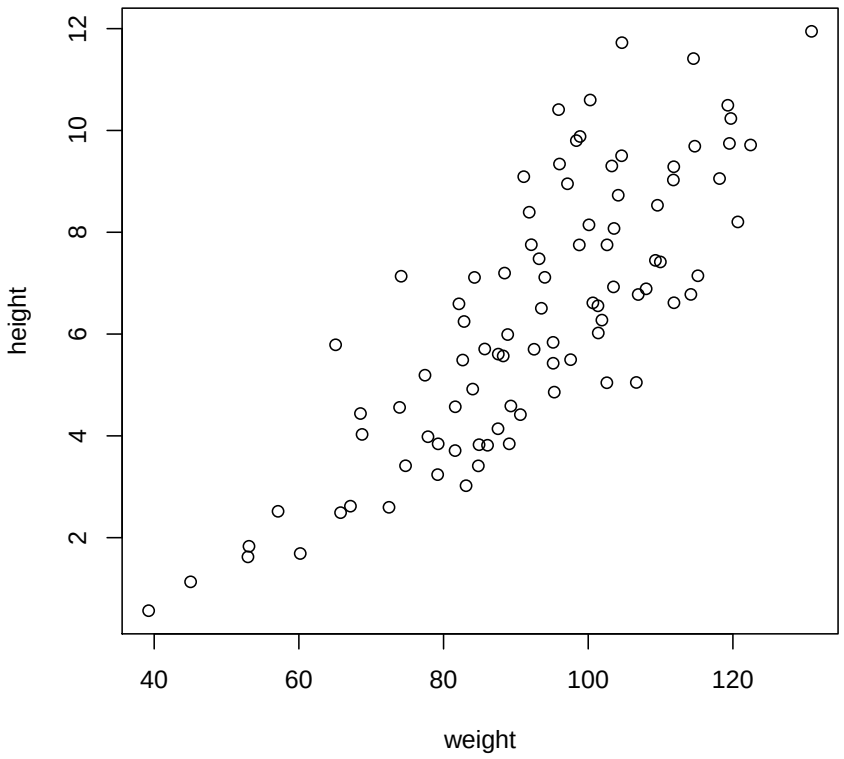
```
> summary(labdata)
  init.size    treatment       mean           height           weight
 Large :30   Control:30   Min.   : 4.00   Min.   : 0.5659   Min.   : 39.23
 Medium:30   High   :30   1st Qu.: 8.00   1st Qu.: 4.4693   1st Qu.: 82.70
 Small :30   Low    :30   Median :12.00   Median : 6.5310   Median : 93.76
                          Mean   :12.67   Mean   : 6.3891   Mean   : 92.35
                          3rd Qu.:16.00   3rd Qu.: 8.3463   3rd Qu.:104.00
                          Max.   :24.00   Max.   :11.9474   Max.   :130.87
```

If your dataset contains many *columns*, all of the above functions might produce output that is a little overwhelming. So at least for starters, you might just want some reassurance that `labdata` has the expected number of rows and columns. You can ask R to report each of those bits of information separately, or together in one function:

```
> nrow(labdata)
[1] 90
> ncol(labdata)
[1] 5
> dim(labdata)
[1] 90  5
```

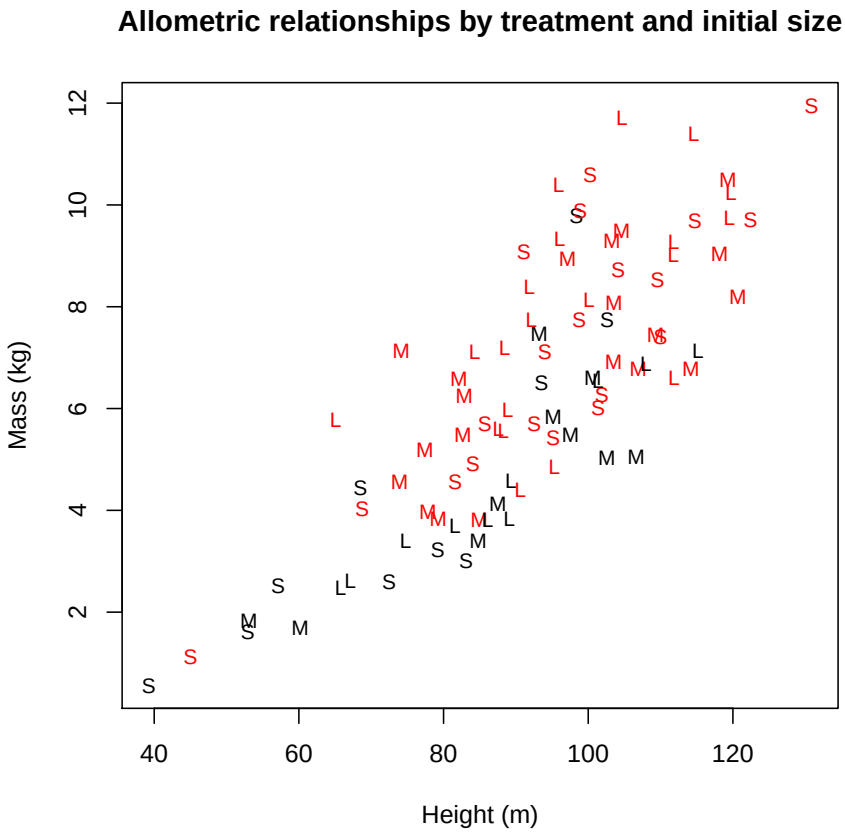
A plot would be nice, eh?

```
> plot(height ~ weight, data = labdata)
```



Plot functions are very flexible, and have a lot of features. We won't get into details now, but have a look at this souped-up version of the previous figure:

```
> plot(height ~ weight, data = labdata, pch = substr(init.size,
    1, 2), main = "Allometric relationships by treatment and initial size",
    xlab = "Height (m)", ylab = "Mass (kg)", cex = 0.8, col = ifelse(treatment ==
        "Control", "black", "red"))
```



How is this `labdata` data actually represented in R? Let's take a look at the guts. The `str` function presents a compact display of the internal structure of an R object. At first it will look very strange, but eventually it will start to become familiar.

```
> str(labdata)
'data.frame':        90 obs. of  5 variables:
 $ init.size: Factor w/ 3 levels "Large","Medium",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ treatment: Factor w/ 3 levels "Control","High",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ mean     : int  16 16 16 16 16 16 16 16 16 16 ...
 $ height   : num  0.566 2.597 6.507 3.022 2.518 ...
 $ weight   : num  39.2 72.5 93.5 83.1 57.1 ...
```

This tells us that `labdata` is a `data.frame`, which is a specialized and important type of R object for representing tabular data in R. This data frame has 90 observations (rows) and 5 variables (columns). The other lines summarize the columns, for each one displaying its name, its data type, and a sampling of its contents.

Time to learn more about lists and vectors...

4 Understanding vectors and lists

By now, most R tutorials would have taught you everything you need to know (and more) about vectors and lists. These are, in fact, two of the most important basic types of R objects. But we like to do things a little different around here. We've started with an entire table of data, because that's probably something you're quite familiar with. However, to really understand how to work with data frames, you really *do* need to know something about vectors and lists. So here goes...

4.1 Vectors

Recall that the `labdata` data frame has 4 columns. As it happens, each column is a vector. A vector is simply a collection of one or more values, all of the same type. The most important kinds of vectors are *logical*, *numeric*, and *character*. Technically, there is also an *integer* type, but in general you don't need to worry about the distinction between integer and numeric. There are also *complex* and *raw* vectors, but you probably won't ever encounter them. Finally, there are objects called *factors* that act a lot like vectors, but have some key differences. We'll talk more about factors later.

Before we get too far into vectors, there are a small number of special values that you should be aware of:

`NA` Vectors of any type may contain `NA` values, which are generally intended to represent missing or unknown values. The tricky thing about them is that most functions applied to an `NA` value will return an `NA`. In fact, many summarization functions applied to a vector will return `NA` if *any* of its values are `NA`. As a convenience some of these functions include a `na.rm` argument for removing `NA` values before performing the calculation, though in any particular application you think about whether this is a sensible thing to do. In any event, one should test for `NA` values using the `is.na` function, *not* using a standard comparison operator. In addition, the `na.omit` function can be used to remove `NA` values, but a more common approach is to use `is.na` in an index, as we'll discuss momentarily.

```
> foo <- c(1, NA, 3)
> foo == NA
[1] NA NA NA
> is.na(foo)
[1] FALSE  TRUE FALSE
> c(na.omit(foo))
[1] 1 3
```

`Inf, -Inf, NaN` These are special values that may occur specifically in numeric (and complex) vectors, usually as the result of some computation.

```
> log(0)
[1] -Inf
> 0/0
[1] NaN
```

The function `is.infinite` tests for `Inf`/`-Inf` values, whereas the function `is.finite` does the opposite. `NaN` ("not a number") elements can be detected using `is.nan`. Note that `is.na` will detect both `NA` and `NaN`, but `is.nan` will only detect `NaN`.

```
> is.infinite(log(c(0, 1, 2)))
[1]  TRUE FALSE FALSE
```

4.1.1 Creating vectors

There are several ways to create them from scratch.

Combining things together An extremely common way to create vectors is to use the `c()` function to *combine* values into a vector. Inside this function, simply separate the values with commas:

```
> vecLog <- c(TRUE, FALSE, TRUE)
> vecNum <- c(1.9, 0.4, 10)
> vecChar <- c("Moe", "Larry", "Curly")
> c(vecChar, "Shep")
[1] "Moe"   "Larry" "Curly" "Shep"
```

Notice the last statement above. Here, we supplied an existing vector as one of the inputs. R will happily build smaller vectors into bigger vectors. But remember, a vector can only contain one data type. You can try to combine different types in one vector, but R will just turn them all into the same type (there are specific rules for which type gets priority when you mix them together; see `?c`).

```
> bigVec <- c(vecNum, vecLog, vecChar, "Shep")
> bigVec
 [1] "1.9"   "0.4"   "10"    "TRUE"  "FALSE" "TRUE"  "Moe"   "Larry" "Curly"
[10] "Shep"
> class(bigVec)
[1] "character"
```

A few things here. First, notice that in this case R just converted everything into a character vector. That's probably not what you wanted. Second, see what happens when the printed vector wraps onto more than one line? To help orient you, R always indicates the index (i.e., position number) of the first element on each line of output when displaying a vector. The first line always starts with `[1]`, for obvious reasons. Here, we can immediately see that "Shep" is the 10th element of `bigVec`.

For convenience and easier reference, vector elements are allowed to have names. These can be specified when creating the vector, or added/modified afterwards.

```
> c(crosby = 67, stills = 64, nash = 67)
crosby stills   nash
    67     64     67
> ages <- c(67, 64, 67)
> names(ages) <- c("crosby", "stills", "nash")
> ages
crosby stills   nash
    67     64     67
```

As a brief aside, a vector of one item is still a vector. In math terms you might call that a scalar, but to R there is no difference. As a convenience, you don't have to use the `c` function to create a vector of length one (but you could if you wanted to). So actually, we've already made some vectors in R. Remember `jenny`?

```
> jenny
[1] 867.5309
> is.vector(jenny)
[1] TRUE
> length(jenny)
[1] 1
```

This, by the way, explains the presence of the `[1]` in the output display of single numbers. R is printing `jenny` just like it prints any other vector; it just happens to contain only one element. But let's get back to creating bigger and better vectors.

Repeats and sequences There are also some convenient functions for creating either repeated or sequential values. These functions are called `rep()` and `seq()`. These functions can be used in a variety of ways, but here is a small sampling.

```
> rep(10, 3)
[1] 10 10 10
> rep(c("a", "b"), each = 3)
[1] "a" "a" "a" "b" "b" "b"
> seq(1, 10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1, 10, by = 3)
[1]  1  4  7 10
```

The `seq` function actually takes a handful of useful arguments for flexibly creating sequences according to different rules. In addition, there are some built-in variants of `seq` that are optimized for the more common usages. See `?seq` for more information.

In addition to `seq`, a convenient shortcut for creating a sequence of integers is to use the colon (`:`) operator. The sequence will be increasing or decreasing depending on whether the smaller number comes first or second.

```
> 1:5
[1] 1 2 3 4 5
> 3:-3
[1]  3  2  1  0 -1 -2 -3
```

Pre-initialized vectors Lastly, you can create vectors of a specified type and length, but with pre-defined placeholder starting values (empty strings for characters, zeroes for numerics and integers, `FALSE`s for logical). This can be useful if you plan to fill it in with other values later. Conveniently, the functions that do that are exactly the vector types mentioned previously:

```
> character(10)
 [1] "" "" "" "" "" "" "" "" "" ""
> numeric(10)
 [1] 0 0 0 0 0 0 0 0 0 0
> logical(10)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

4.1.2 Indexing vectors

Once you've got a vector, often you'll want to extract particular elements from it based on their position. This is where indexing comes in. The general syntax is `vector_of_data[index_vector]`. How does it work? Well, it all depends on what kind of vector you supply inside the square brackets. Let's look at the options:

Numeric indexing Perhaps the most straightforward approach is to index by element position number. If you place a numeric (or integer) vector inside the square brackets, R assumes you want the elements located at the associated position numbers. (Note that if you use a numeric index with something after the decimal point, R will simply truncate it.)

```
> vec <- c("b", "a", "n", "a", "n", "a")
> vec[c(1, 3, 5)]
[1] "b" "n" "n"
```

Notice that the desired indices must themselves supplied as a vector. The following produces an error:

```
> vec[1, 3, 5]


Error in vec[1, 3, 5] : incorrect number of dimensions
```

It is legal to use an index containing all negative numbers, indicating which elements you *don't* want. As you might imagine, it's an error to mix negative and positive indices, so don't do it.

```
> vec[-c(1, 3, 5)]
[1] "a" "a" "a"
```

Logical indexing You can also use a logical vector for indexing. Vector elements corresponding to `TRUE` will be returned, and those corresponding to `FALSE` will not. This might strike you as esoteric at first, but you will probably use this method of indexing more than any other!

```
> is.not.a <- (vec != "a")
> is.not.a
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
> vec[is.not.a]
[1] "b" "n" "n"
```

A common example of logical indexing involves removal of `NA` values, like this:

```
> vec2 <- c(1, 2, NA, 4)
> vec2[!is.na(vec2)]
[1] 1 2 4
```

Usually, the vector used for logical indexing is of the same length as the vector you want to index. but technically it can be shorter. In that case, R will do something called *recycling*. This means it will repeat the index vector until it is long enough. That's why this works:

```
> vec[c(TRUE, FALSE)]
[1] "b" "n" "n"
```

R uses *recycling* in a number of places, and we'll encounter a few more in this document. It can be a handy shortcut, but it can also be a pitfall. Watch out.

Character indexing Can the index vector be a character vector? Yes! In that case, it will match against the element *names* of the vector.

```
> ages[c("crosby", "young")]
crosby   <NA>
    67     NA
```

Whoops! The `ages` vector has no element named `"young"`. Consequently, R returns `NA` for that particular index. Note that this is not specific to character indexing. Any time you attempt to index a vector element that doesn't exist, R will return `NA`. This issue does, nevertheless, tend to be more common when using character indexing. For what it's worth, you can return only the elements that actually *exist* in your character vector by using a construct like this:

```
> ages[which(names(ages) %in% c("crosby", "young"))]
crosby
    67
```

One other gotcha with character indexing involves duplicate names. Only the first match will be returned. And no, it won't help to supply the name multiple times in the index vector. For these reason, it may not make much sense to index this way unless you first ensure uniqueness of names.

```
> c(a = 1, b = 2, a = 3)[c("a", "a")]
a a
1 1
```

4.1.3 Modifying vectors "in place"

R provides a mechanism for modifying particular elements of a vector, while leaving the others unchanged. This is known as *replacement*, and it is simply a combination of two things we've already done: indexing and assignment.

```
> vec[c(1, 3, 5)] <- "f"
> vec
[1] "f" "a" "f" "a" "f" "a"
> vec[is.not.a] <- c("e", "i", "o")
> vec
[1] "e" "a" "i" "a" "o" "a"
```

What if you attempt replacement on a vector element that doesn't exist? R will simply extend the vector as necessary, and fill in any empty elements with `NA` values:

```
> vec[10] <- "Wow!"
> vec
 [1] "e"    "a"    "i"    "a"    "o"    "a"    NA     NA     NA     "Wow!"
```

4.1.4 Matrices

MATLAB users in particular may be shocked to see that matrices don't get their own section in this document. In R, matrices are nothing special. They are simply vectors with a `dim` (dimension) attribute giving the number of rows and columns. The `matrix` function is typically used to create a matrix from scratch, but the following code shows how a matrix is related to a vector:

```
> x1 <- letters[1:12]
> class(x1)
[1] "character"
> dim(x1) <- c(4, 3)
> class(x1)
[1] "matrix"
> x1
     [,1] [,2] [,3]
[1,] "a"  "e"  "i"
[2,] "b"  "f"  "j"
[3,] "c"  "g"  "k"
[4,] "d"  "h"  "l"
> identical(x1, matrix(letters[1:12], nrow = 4))
[1] TRUE
```

Notice that R uses *column-major* order when creating matrices, which means each column is filled out in order. If you want to create a matrix using row-major ordering, you can add a `byrow=TRUE` argument to the `matrix` function.

Just like with vectors, you can extract pieces of a matrix using positional, logical, or name-based indices. In fact, as illustrated below, you can use a single index as though the matrix were one-dimensional (again, this is a column-major index). But it's almost always more convenient to use use two-dimensional indexing with matrices. Inside the indexing brackets, the first expression indexes rows, and the second expression indexes columns. An empty index will return all values along that dimension.

```
> x1[c(4, 8, 12)]
[1] "d" "h" "l"
> x1[4, ]
[1] "d" "h" "l"
> x1[1:3, c(TRUE, FALSE, TRUE)]
     [,1] [,2]
[1,] "a"  "i"
[2,] "b"  "j"
[3,] "c"  "k"
```

Look again at the results above. There is one subtle gotcha with matrix subsetting (you'll see later that this applies to data frames too). If the result of your subset is a single dimension, R will automatically "demote" it to a vector. If that's not what you want, you'll need to supply a `drop=FALSE` argument to the index:

```
> x1[4, , drop = FALSE]
     [,1] [,2] [,3]
[1,] "d"  "h"  "l"
```

A final method for indexing matrices is to supply a 2-column numeric matrix, in which each row specifies the row index and column index of a desired element.

```
> ind <- cbind(c(2, 3, 4), c(1, 2, 3))
> ind
     [,1] [,2]
[1,]    2    1
[2,]    3    2
[3,]    4    3
```

```
> x1[ind]
[1] "b" "g" "l"
```

As a final comment, R also supports an `array` class of objects. An array is basically a multi-dimensional generalization of a matrix. Arrays can be created using the `array` function, but otherwise the concepts presented above for matrices all apply equally well to array objects.

4.2 Lists

Wouldn't it be nice to have something like a vector that can hold different kinds of objects all in one container? There is! It's called a list. You can think of a list as a generic kind of vector. A single list might have some elements that are vectors (of differing types and lengths), some elements that are lists themselves, and some elements that are other objects we haven't even talked about yet. Any valid R object can be put in a list.

Be aware that this generality has a cost. For one thing, there aren't nearly as many functions for doing specific things with vectors, since functions can't know in general what the list will contain. Moreover, operations on large lists are generally slower than on vectors. If you can put something in a vector instead, you probably should.

On the other hand, it is worth understanding lists, for the following reasons:

- Lists are useful for "ragged" data (vectors of different lengths)
- Many specialized objects in R are really just lists, *including data frames*. Usually you will use specialized functions designed to work with these objects, but sometimes you'll just need to get under the hood and do something yourself!
- Looping can be emulated more efficiently by applying some custom function to every element of a list, and in turn returning the results in a list

4.2.1 Creating lists

You can create lists from scratch using the `list()` function. Just like with the `c()` function, separate the elements using commas. Unlike with vectors, when creating lists you'll almost always want to supply names for each component. This simply uses a `name=object` syntax within the `list` function call:

```
> tree <- list(genus = "Liriodendron", species = c("chinense",
     "tulipifera"), isNative = c(FALSE, TRUE))
> tree
$genus
[1] "Liriodendron"

$species
[1] "chinense"   "tulipifera"

$isNative
[1] FALSE  TRUE
```

Notice the more complicated display. Each element of the list is printed beneath its name (if no names were provided, the positional index would be shown instead, inside double square brackets).

4.2.2 Indexing lists

List indexing is similar to vector indexing, but it is a little more nuanced. Recall that `vecLog[ind]` returns a vector containing only the subset of `vecLog` specified by `ind`. In an analogous manner, `tree[ind]` returns a *list* containing only the subset of `tree` specified by `ind`.

```
> tree["species"]
$species
[1] "chinense"   "tulipifera"
```

Here we asked for one element of the list, and so we got back a list with one element in it. But what if you want the actual vector of species names itself, rather than a one-item list containing that vector? There are actually two methods: one way is to use *double* square brackets (`[[]]`), and other is to use dollar sign (`$`) notation:

```
> tree[["species"]]
[1] "chinense"   "tulipifera"
> tree$species
[1] "chinense"   "tulipifera"
```

The latter form is more common, probably just because it is easier to type. However, it differs in two ways that are more disadvantageous than not. First, it will do *partial matching* on the element names, which ends up being not so much a convenience as a way to introduce hard-to-catch bugs in your code.

```
> tree[["sp"]]
NULL
> tree$sp
[1] "chinense"   "tulipifera"
```

Secondly, sometimes it can be handy to access a list element not by supplying the name directly, but by supplying a variable that holds the name. That can only be done using the `[[]]` indexing method:

```
> myvar <- "species"
> tree[[myvar]]
[1] "chinense"   "tulipifera"
> tree$myvar
NULL
```

As a final point, note that both of these indexing methods can only be used to select a *single* list element.

```
> tree[[c(1, 2)]]

Error in tree[[c(1, 2)]] : subscript out of bounds
```

4.2.3 Modifying lists "in place"

As with vectors, one can combine indexing with assignment to replace elements of a list. When doing replacement in lists, it really doesn't matter which form of indexing you use.

```
> tree$species <- 2
> tree
$genus
[1] "Liriodendron"

$species
[1] 2

$isNative
[1] FALSE  TRUE
```

What if you attempt replacement on a list element that doesn't exist? Again, just as with vectors, R will simply extend the list. With lists, however, it is more common to create a new element by name rather than by number. In this case, the new element is simply added at the end of the list. (This is actually true for vectors too, though we didn't mention it.)

```
> tree$height.class <- "very tall"
> tree
$genus
[1] "Liriodendron"

$species
[1] 2

$isNative
[1] FALSE  TRUE

$height.class
[1] "very tall"
```

Note how replacement versus extension hinges on whether the supplied name already exists (exactly) in the list. If you tend to be inconsistent with your capitalization or spelling, or for that matter a bad typist, watch out.

4.2.4 Data frames

Hopefully you still remember our `labdata` data frame. We are now equipped to talk about what it's made up of. A data frame is simply a `list` of `vectors` all of the same length; each vector represents a column of data. Data frames also contain a separate character vector of row names, and these are guaranteed to be unique (they are initially based on the row number, if no other names are provided).

Although data frames are often produced by loading external data, one can be created from scratch using the `data.frame` function.

```
> dat <- data.frame(id = letters[1:5], days = c(1, 1, 1, 2, 2),
    count = rpois(5, 20))
> dat
  id days count
1  a    1    11
2  b    1    22
3  c    1    17
```

```
4  d    2    25
5  e    2    13
```

Because a data frame is a list, one can extract columns using the same indexing methods described for lists above. Behavior of `[ind]`-style indexing is consistent with what we've seen before; when applied to data frames, it will return a data frame contains the columns specified by the index. As usual, the index vector can be character (as in the examples below), numeric, or logical.

```
> dat["id"]
  id
1  a
2  b
3  c
4  d
5  e
> dat[c("days", "count")]
  days count
1    1    11
2    1    22
3    1    17
4    2    25
5    2    13
```

Analogous to what we saw with lists more generally, a particular underlying column vector can be extracted using either `$` or `[[]]` notation. As an added convenience when using data frames, one can also use matrix-like indexing based on rows and columns. All three statement below return the same vector:

```
> dat$count
> dat[["count"]]
> dat[, "count"]


[1] 11 22 17 25 13
```

Unlike lists - but similar to matrices - data frames also support two-dimensional indexing for simultaneously extracting certain rows and certain columns. Just like with matrices, when using the `[r, c]`-style indexing, you will need to supply the `drop=FALSE` argument if you want to keep R from converting one-dimensional reults to a vector rather than returning a data frame.

```
> dat[3:5, c("id", "count")]
  id count
3  c    17
4  d    25
5  e    13
> dat[3:5, "count"]
[1] 17 25 13
> dat[3:5, "count", drop = FALSE]
  count
3    17
4    25
5    13
```

Finally, column replacement and addition of new columns is done just like with lists. Frequently, you'll find yourself indexing a particular column *and* replacing particular elements of a column at the same time. Below, we'll modify some elements of one column, then compute a new column based on existing ones. Also notice how the recycling applies below.

```
> dat$days[3:5] <- 3
> dat$freq <- dat$count/dat$days
> dat
  id days count      freq
1  a    1    11 11.000000
2  b    1    22 22.000000
3  c    3    17  5.666667
4  d    3    25  8.333333
5  e    3    13  4.333333
```

5 A few other important kinds of objects

Don't get confused by the fact that these all start with the letter *f*. These are three very different beasts in R.

5.1 Factors

Factors are used for representing a discrete set of values (called levels). Let's illustrate via example:

```
> fac <- factor(c("high", "high", "medium", "low", "low", "low"))
> fac
[1] high   high   medium low    low    low
Levels: high low medium
> levels(fac)
[1] "high"   "low"    "medium"
```

Notice how the printed display differs from that of a character vector. The internal representation is also different. R assigns each unique element to an integer. The benefit of factors is that a variety of high level statistical and plotting functions know to do something special with factors. However, the problem is that although factors *seem* to be just like character vectors, they aren't. Many functions applied to factors will do something with the integer representation, rather that with the text values themselves. For example, if you use a factor to index a vector, it will use the underlying integers, not the actual factor levels.

Another problem that you can't just change a factor element to something not already among the factor levels. Instead, you would either have to add the new level first, or coerce the factor to character, modify the character vector, then coerce back to factor.

```
> fac[1] <- "veryhigh"
> fac
[1] <NA>   high   medium low    low    low
Levels: high low medium
> levels(fac) <- c(levels(fac), "veryhigh")
> fac[1] <- "veryhigh"
> fac
[1] veryhigh high     medium   low      low      low
Levels: high low medium veryhigh
```

By default, any column of characters will be converted to a factor column by the `data.frame` function. This also happens when using `read.table` and related functions. In cases where factors might get you into trouble, you can supply an extra argument `stringsAsFactors=FALSE` to keep these columns as character vectors.

A word of caution: On occasion you may find yourself with a factor that you really want to convert to numeric. Although you may be tempted to use `as.numeric`, don't do it! This will simply return the integer codes, rather than the values themselves.

```
> y <- factor(c(45, 20, 12))
> as.numeric(y)
[1] 3 2 1
```

Use one of the following instead (the second statement is more cryptic, but also more efficient):

```
> as.numeric(as.character(y))
[1] 45 20 12
> as.numeric(levels(y))[y]
[1] 45 20 12
```

One final comment about factors. By default the levels will be in alphabetical order. Using the `factor` function, custom ordering (or for that matter, additional levels not found in the data) can be supplied using the `levels` argument. Moreover, a factor can be declared an `ordered=TRUE` argument, or by using the `ordered` function in place of the `factor` function:

```
> ordered(c("high", "high", "medium", "low", "low", "low"), levels = c("low",
    "medium", "high"))
[1] high   high   medium low    low    low
Levels: low < medium < high
```

5.2 Formulas

A formula in R is a symbolic representation of a model. Formulas are the most common way to specify a model form, which can then supplied as an argument to most functions that perform model fitting.

The most basic formula takes the form y $\sim$ x, which indicates that y is modeled as a function of x. Note that statistical functions assume an implied intercept, but you can explicitly exclude the intercept as documented below. Of course, you may want to model `y` as more than just a function of `x`. Here are some examples of several simple formulas:

```
y ~ x                 # One response, one predictor, implied intercept
y ~ 0 + x             # Exclude the intercept term
y ~ -1 + x            # Same as above
y ~ x1 + x2           # Two predictors
y ~ x1 + x2 + x1:x2   # Two predictors plus their interaction
y ~ x1*x2             # Same as above
```

Incidentally, formulas are also used in a variety of other functions. In fact, we already saw an example of this.

```
> plot(weight ~ height, data = labdata)
```

This basically says, "using columns from the `labdata` data frame, plot `weight` as a function of `height`". Although the basic `plot` function in R doesn't really do anything useful with more complicated formulas, plotting functions in the `lattice` package can produce more complicated figures that involve nesting and crossing of variables.

5.3 Functions

Time to learn about one more basic building block before moving on to more advanced usage. We've already looked at vectors, lists, and some compound objects that build upon them (like data frames). Now we need to *do* something with them. This is what functions are for. Obviously we've seen a lot of functions already, but now we'll take a look at the main components of functions, how to use them, and how to make your own!

Let's start out with this gem from the R-help list:

```
> library(fortunes)
> fortune(188)
Corinna Schmitt: How can I divide the number 0.285 with 2. I need a function.
  Result: 0.285 / 2 = 0.1425
Gabor Csardi: Well, i think
  half.of.0.285 <- function() {
    0.1425
  }
would do the trick.
   -- Corinna Schmitt and Gabor Csardi
      R-help (April 2007)
```

Well, the function proposed above has limited utility, but it's a function nevertheless. Let's add one tiny enhancement, and then explain how it all works.

```
> half.of <- function(x) {
     x/2
 }
> half.of(0.285)
[1] 0.1425
```

As usual, on the left side of the `<-` is the name we're assigning to this function. On the right side is an expression that creates and returns a function. Immediately following the assignment operator is the word `function`, which is the function that creates functions. (Feel free to read the last sentence if it didn't make sense the first time.) Inside the parentheses following `function` appear any *arguments* of the function, in this case just `x`. Finally, we write the function *body*. Typically this is enclosed in curly braces (`{}`), but technically you don't need them for functions involving only a single expression.

Now let's make a slightly more interesting function.

```
> half.of.a.difference <- function(x, y) {
     difference <- x - y
     answer <- difference/2
     return(answer)
 }
> half.of.a.difference(11, 3)
[1] 4
```

In R, a function is just another kind of object. This means, for example, that if you just type the object name, R will print it. This also means you can pass a function to other functions, just like you can pass vectors and lists and any other object.

```
> half.of.some.operation <- function(x, y, FUN) {
     intermediate <- FUN(x, y)
     answer <- intermediate/2
     return(answer)
 }
> half.of.some.operation(11, 3, max)
```

```
[1] 5.5
> half.of.some.operation(11, 3, min)
[1] 1.5
```

6 More fun with data

Let's do some simple statistical modeling. The `lm` function is used to fit linear models, including regression, ANOVA, and ANCOVA. Here we'll just do a simple regression, The first argument of `lm` is a *formula* of the form `y ~ x`, where `y` is the response and `x` is the predictor; the other argument you'll usually pass to `lm` is the name of the `data.frame` that contains your variables.

```
> fm1 <- lm(height ~ init.size, data = labdata)
> fm1
Call:
lm(formula = height ~ init.size, data = labdata)

Coefficients:
    (Intercept)  init.sizeMedium   init.sizeSmall
         6.7904          -0.6242          -0.5799
```

The `lm` function returns an object that we've saved as `fm1`. When we ask R to show us the object, we get a rather spartan presentation of the original function call, followed by parameter estimates (intercept and slope, in this case). However, this object is actually chock full of information that we can subsequently view, extract, and pass to other functions. Let's take a peek under the hood:

```
> str(fm1)
List of 13
 $ coefficients : Named num [1:3] 6.79 -0.624 -0.58
  ..- attr(*, "names")= chr [1:3] "(Intercept)" "init.sizeMedium" "init.sizeSmall"
 $ residuals    : Named num [1:90] -5.645 -3.614 0.296 -3.189 -3.692 ...
  ..- attr(*, "names")= chr [1:90] "1" "2" "3" "4" ...
 $ effects      : Named num [1:90] -60.61 -1.49 2.25 -2.56 -3.06 ...
  ..- attr(*, "names")= chr [1:90] "(Intercept)" "init.sizeMedium" "init.sizeSmall" "" ...
 $ rank         : int 3
 $ fitted.values: Named num [1:90] 6.21 6.21 6.21 6.21 6.21 ...
  ..- attr(*, "names")= chr [1:90] "1" "2" "3" "4" ...
 $ assign       : int [1:3] 0 1 1
 $ qr           :List of 5
  ..$ qr   : num [1:90, 1:3] -9.487 0.105 0.105 0.105 0.105 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:90] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:3] "(Intercept)" "init.sizeMedium" "init.sizeSmall"
  .. ..- attr(*, "assign")= int [1:3] 0 1 1
  .. ..- attr(*, "contrasts")=List of 1
  .. .. ..$ init.size: chr "contr.treatment"
  ..$ qraux: num [1:3] 1.11 1.07 1.11
  ..$ pivot: int [1:3] 1 2 3
  ..$ tol  : num 1e-07
  ..$ rank : int 3
  ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 87
```

```
 $ contrasts    :List of 1
  ..$ init.size: chr "contr.treatment"
 $ xlevels      :List of 1
  ..$ init.size: chr [1:3] "Large" "Medium" "Small"
 $ call         : language lm(formula = height ~ init.size, data = labdata)
 $ terms        :Classes 'terms', 'formula' length 3 height ~ init.size
  .. ..- attr(*, "variables")= language list(height, init.size)
  .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "height" "init.size"
  .. .. .. ..$ : chr "init.size"
  .. ..- attr(*, "term.labels")= chr "init.size"
  .. ..- attr(*, "order")= int 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(height, init.size)
  .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "factor"
  .. .. ..- attr(*, "names")= chr [1:2] "height" "init.size"
 $ model        :'data.frame':        90 obs. of  2 variables:
  ..$ height   : num [1:90] 0.566 2.597 6.507 3.022 2.518 ...
  ..$ init.size: Factor w/ 3 levels "Large","Medium",..: 3 3 3 3 3 3 3 3 3 3 ...
  ..- attr(*, "terms")=Classes 'terms', 'formula' length 3 height ~ init.size
  .. .. ..- attr(*, "variables")= language list(height, init.size)
  .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. .. ..$ : chr [1:2] "height" "init.size"
  .. .. .. .. ..$ : chr "init.size"
  .. .. ..- attr(*, "term.labels")= chr "init.size"
  .. .. ..- attr(*, "order")= int 1
  .. .. ..- attr(*, "intercept")= int 1
  .. .. ..- attr(*, "response")= int 1
  .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. .. ..- attr(*, "predvars")= language list(height, init.size)
  .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "factor"
  .. .. .. ..- attr(*, "names")= chr [1:2] "height" "init.size"
 - attr(*, "class")= chr "lm"
```

Yikes! Fortunately, you almost never have to wade through the object to get what you need (although you certainly *can*, if you feel like it). For starters, the `summary` function prints a collection of useful information, including quartiles of the residuals, a complete coefficients table (with estimates, standard errors, t-statistics, and p-values), residual standard error, R-squared values, and results of an F-test of model significance:

```
> summary(fm1)
Call:
lm(formula = height ~ init.size, data = labdata)

Residuals:
     Min       1Q   Median       3Q      Max
-5.64465 -2.00214  0.07246  2.00498  5.73691

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)       6.7904     0.4822  14.084   <2e-16 ***
init.sizeMedium  -0.6242     0.6819  -0.915    0.363
init.sizeSmall   -0.5799     0.6819  -0.850    0.397
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.641 on 87 degrees of freedom
Multiple R-squared: 0.01185,        Adjusted R-squared: -0.01086
F-statistic: 0.5218 on 2 and 87 DF,  p-value: 0.5953
```

In addition, a handful of special purpose *extractor* functions are available for pulling out specific bits of information. Here are a couple of examples:

```
> coef(fm1)
    (Intercept) init.sizeMedium  init.sizeSmall
      6.7904052      -0.6241736      -0.5798828
> logLik(fm1)
'log Lik.' -213.5778 (df=4)
> AIC(fm1)
[1] 435.1557
```

For more information, consult the help documentation for `?lm` and `?summary.lm`.

7 Basic data manipulation

Let's continue to work with our `labdata` data frame.

7.1 Vectorization

You will frequently want to perform some operation on each element in a vector, or compute some sort of summary over all elements. In lower level programming languages like C, this would require a loop. Although one can create explicit loops in R, they tend to be much slower. More importantly, they are usually unnecessary! Many built-in functions in R are *vectorized*, which means they act on entire vectors automatically.

Let's illustrate using some data extracted from our `labdata` data frame. To save space, we'll just take a subset of values.

```
> wgt <- labdata$weight[labdata$init.size == "Small" & labdata$treatment ==
    "Control"]
> hgt <- with(labdata, height[init.size == "Small" & treatment ==
    "Control"])
```

(Note the use of the `with` function in the second statement above. This saves us from having to type out the name of the data frame every time we reference one of the column.)

```
> length(wgt)  # Report number of elements in a vector
[1] 10
> wgt/hgt   # Element-wise division of one vector by another
 [1] 69.33115 27.90109 14.37312 27.50648 22.68472 13.23389 32.62598 10.03600
 [9] 15.43396 24.44982
> 100 * wgt    # 100 gets "recycled" here!
```

```
 [1]  3923.252  7245.949  9352.200  8311.182  5712.095 10259.748  5295.962
 [8]  9835.914  6851.491  7918.459
> sqrt(wgt)    # Take the square root of all elements
 [1]  6.263587  8.512314  9.670677  9.116569  7.557840 10.129042  7.277336
 [8]  9.917618  8.277373  8.898573
> range(wgt)   # Report the min and max
[1]  39.23252 102.59748
> mean(wgt)    # Calculate the mean
[1] 74.70625
> sd(wgt)      # Calculate the standard deviation
[1] 20.72585
```

7.2 Matching elements

Return the indices of object elements satisfying some condition:

```
> x <- c(1, 0, 0, 1, 1, 0, 0, 1, 0, 0)
> which(x == 1)
[1] 1 4 5 8
```

As above, but return indices along each margin (e.g. row, column):

```
> x <- matrix(c(1, 0, 0, 1, 1, 0, 0, 1, 0, 0), nrow = 2)
> which(x == 1, arr.ind = TRUE)
     row col
[1,]   1   1
[2,]   2   2
[3,]   1   3
[4,]   2   4
```

7.3 Set operations

```
> a <- c("c", "h", "a", "r", "l", "e", "s")
> b <- c("d", "a", "r", "w", "i", "n")
> setequal(a, b)
[1] FALSE
> setdiff(a, b)
[1] "c" "h" "l" "e" "s"
> setdiff(b, a)
[1] "d" "w" "i" "n"
> intersect(a, b)
[1] "a" "r"
> union(a, b)
 [1] "c" "h" "a" "r" "l" "e" "s" "d" "w" "i" "n"
```

7.4 Sorting, ordering, and ranking

```
> x <- c(12, 15, 11, 13, 14)
```

Return the *indices* sorted by value

```
> order(x)
[1] 3 1 4 5 2
```

Return the *ranks* of each element. See `?rank` for options regarding ties.

```
> rank(x)
[1] 2 5 1 3 4
```

Return a sorted vector

```
> sort(x)
[1] 11 12 13 14 15
> x[order(x)]
[1] 11 12 13 14 15
> x[order(c(1, 1, 1, 2, 2), x)]
[1] 11 12 15 13 14
```

7.5 Tabulation and cross-tabulation

table The `table` function is extremely handy for creating a frequency table from a vector. In other words, it tells you how many of each unique value you have in your vector.

Generate a frequency table based on one or more columns (usually where the columns are discrete groupings). Syntax: specify factor columns of interest.

```
> table(labdata$init.size)
 Large Medium  Small
    30     30     30
> table(labdata$treatment)
Control    High     Low
     30      30      30
> table(labdata$init.size, labdata$treatment)
         Control High Low
  Large       10   10  10
  Medium      10   10  10
  Small       10   10  10
```

xtabs This makes a contingency table of selected factors, summing some variable. Similar to running tapply with `sum` function, but the ouput is a contingency table with associated chi-square test results.

Compare:

```
> tapply(labdata$height, labdata[1:2], sum)
         treatment
init.size  Control     High      Low
   Large  45.07217 70.39736 88.24263
   Medium 46.59428 65.24584 73.14683
   Small  42.06358 65.16819 79.08390
```

```
> xtabs(height ~ init.size + treatment, labdata)
          treatment
init.size  Control     High      Low
   Large  45.07217 70.39736 88.24263
   Medium 46.59428 65.24584 73.14683
   Small  42.06358 65.16819 79.08390
```

...but note:

```
> summary(xtabs(height ~ init.size + treatment, labdata))
Call: xtabs(formula = height ~ init.size + treatment, data = labdata)
Number of cases in table: 575.0148
Number of factors: 2
Test for independence of all factors:
        Chisq = 0.8165, df = 4, p-value = 0.9362
```

(Obviously the chi-square test is not really useful in this case, but is nice in cases where you have, e.g., several columns of factors and one column containing frequency data.)

7.6 Binning

The `cut` function creates a factor by binning a numeric variable. This is a bit off topic, but since we've been talking about summarizing data based on factors, here is a function that takes a vector of numeric data, and creates a factor variable based on N equal interval groupings

```
> cut(wgt, 3)
 [1] (39.2,60.3] (60.3,81.5] (81.5,103]  (81.5,103]  (39.2,60.3] (81.5,103]
 [7] (39.2,60.3] (81.5,103]  (60.3,81.5] (60.3,81.5]
Levels: (39.2,60.3] (60.3,81.5] (81.5,103]
```

Usually you'll want to be more specific with `cut`. The example below generates create two `wgt` bins, one for values smaller than the median, and one for values larger than the median. We can also supply friendlier labels:

```
> cut(wgt, breaks = c(min(wgt), median(wgt), max(wgt)), labels = c("small",
    "big"), include.lowest = TRUE)
 [1] small small big   big   small big   small big   small big
Levels: small big
```

Later we'll see how to do useful things with these bins. For now, let's add a new column to our data frame to store this binning.

```
> labdata$weight.bin <- with(labdata, cut(weight, breaks = c(min(weight),
    median(weight), max(weight)), labels = c("small", "big"),
    include.lowest = TRUE))
> head(labdata)
  init.size treatment mean    height    weight weight.bin
1     Small   Control   16 0.5658715  39.23252      small
2     Small   Control   16 2.5970133  72.45949      small
3     Small   Control   16 6.5067285  93.52200      small
4     Small   Control   16 3.0215359  83.11182      small
5     Small   Control   16 2.5180364  57.12095      small
6     Small   Control   16 7.7526340 102.59748        big
```

8 Higher level data manipulation

In this section we'll look at some important general functions for working on multiple rows or columns of data. New R users often attempt to write their own loop constructs to accomplish these tasks. Why work hard to write (and debug!) a custom loop when you can just use a built-in function instead, especially when this function will almost certainly run *faster* than your hand-written code? In short, it is definitely worth learning to use these functions. For quick reference, here is a list of the functions covered in this section:

lapply Repeat some function for each element of a list (e.g., data frame columns)

sapply Similar to `lapply`, but puts results in a (nicer looking) matrix if possible

apply Repeat some function for each row or column (e.g., take mean value across rows)

mapply Repeat some function of >1 columns, for each row

tapply Repeat some function for groups of rows identified by common factor(s) [uses vectors]

by Similar to tapply, but works on data frames

aggregate Similar to tapply, but only with functions that return a scaler (mean, max, min, etc)

subset Return a subset of columns and/or rows according to some user-supplied criteria

cbind Concatenate two data frames side-by-side (column-wise)

rbind Concatenate two data frames atop one another (row-wise)

merge Combine (a.k.a. join) two data frames based on matching values in particular columns, or possibly by common row names

---

8.1 Applying functions across rows or columns

Before we get to the general functions below, commit to memory four functions for calculating sums or means of rows or columns of a matrix, array, or data frame. Not only are these easier to write, but they run faster, too:

```
> colSums(x)
> rowSums(x)
> colMeans(x)
> rowMeans(x)
```

It should be obvious what these do, based on their names. You'll get an error if you try to run these on character vectors or factors, but otherwise they are quite straightforward to use. Be sure to add an `na.rm=TRUE` argument if you want to drop any `NA` values before computing the result.

Now on to the more general problem. Imagine you have data from an experiment. Each row represents an independent experimental unit, and you have measured variables stored in different columns. You have some analytical procedure that you want to apply to each variable. How do you do it?

lapply Repeat some function for each element of a list (e.g. data frame columns!)

First, note that we can't do numerical processing of all columns because they're not all numeric. This line returns an error:

```
> lapply(labdata, max)


Error in Summary.factor(structure(c(3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L,  :
  max not meaningful for factors
```

So let's just work with the 3rd and 4th columns, which we know are numbers. Certain functions, like `mean`, will automatically do columnwise operations when used on a `data.frame` object:

```
> mean(labdata[3:4])
     mean    height
12.666667  6.389053
```

But most of the time, this isn't the case, and you'll either get an error or a result that you didn't expect:

```
> max(labdata[3:4])
[1] 24
```

Here, `max` returned the maximum value over all the numbers in those columns. This is because it is designed (and documented) to work on *vectors*, not to do columnwise operations on `data.frames`. This is one place where `lapply` can come in handy. Here's how it works:

```
> lapply(labdata[3:4], max)
$mean
[1] 24

$height
[1] 11.94743
```

The general translation of `lapply` is, "for each element in the first argument, apply the function given by the second argument, and return the results in a list." Because a `data.frame` is just a `list` whose elements are `vectors`, this equates to applying the function `max` to each column of `labdata[3:4]`.

Note that the "long form" of this expression is:

```
> lapply(labdata[3:4], function(x) max(x))
```

This allows us to apply our own function, either by defining the function within the `lapply` statement itself:

```
> lapply(labdata[3:4], function(x) mean(log(x + 1)))
```

or by using a function previously defined elsewhere:

```
> meanlog <- function(x) {
      mean(log(x + 1))
  }
> lapply(labdata[3:4], meanlog)
$mean
[1] 2.498812

$height
[1] 1.924030
```

What if we use a more complicated function? For example, `quantile` returns a named vector rather than just a scalar. No problem!

```
> quantile(labdata[[3]])
  0%  25%  50%  75% 100%
   4    8   12   16   24
> lapply(labdata[3:4], quantile)
$mean
  0%  25%  50%  75% 100%
   4    8   12   16   24

$height
        0%        25%        50%        75%       100%
 0.5658715  4.4693091  6.5309893  8.3463277 11.9474274
```

Want output in a data frame?

```
> as.data.frame(lapply(labdata[3:4], quantile))
     mean     height
0%      4  0.5658715
25%     8  4.4693091
50%    12  6.5309893
75%    16  8.3463277
100%   24 11.9474274
```

sapply Similar to `lapply`, but puts results in a (nicer looking) matrix if possible. In particular, `sapply` tries to produce nice tabular (matrix) output by default: Note that the returned object really is a matrix, *not* a data frame. (If you really want a data frame instead, just use `as.data.frame` to convert the matrix)

```
> sapply(labdata[3:4], mean)
     mean    height
12.666667  6.389053
> sapply(labdata[3:4], sum)
     mean    height
1140.0000  575.0148
> sapply(labdata[3:4], quantile)
     mean     height
0%      4  0.5658715
25%     8  4.4693091
50%    12  6.5309893
75%    16  8.3463277
100%   24 11.9474274
> sapply(labdata[3:4], meanlog)
    mean   height
2.498812 1.924030
```

apply Repeat some function for each column or row of tabular data (e.g., take mean value across rows). The `lapply` and `sapply` are usually preferable for doing

With `apply`, the second argument is used to specify margin to use: 1 for row, 2 for column:

```
> apply(labdata[3:4], 2, mean)
     mean    height
12.666667  6.389053
> apply(labdata[3:4], 2, meanlog)
    mean   height
2.498812 1.924030
> head(apply(labdata[3:4], 1, sum))
[1] 16.56587 18.59701 22.50673 19.02154 18.51804 23.75263
```

mapply Repeat some function of >1 columns, for each row.

```
> head(mapply("*", labdata$height, labdata$weight))
[1]  22.20056 188.17827 608.52228 251.12535 143.83264 795.40073
> head(mapply("*", labdata["height"], labdata["weight"]))
        height
[1,]  22.20056
[2,] 188.17827
[3,] 608.52228
[4,] 251.12535
[5,] 143.83264
[6,] 795.40073
> head(sum(mapply("*", labdata["height"], labdata["weight"])))
[1] 56535.98
> myfunc <- function(x, y) {
     return(2.6 + 0.5 * x + 1.3 * y)
 }
> head(mapply(myfunc, labdata["height"], labdata["weight"]))
        height
[1,]  53.88521
[2,]  98.09585
[3,] 127.43197
[4,] 112.15614
[5,]  78.11626
[6,] 139.85304
> intrcpt <- 2.6
> coef.h <- 0.5
> coef.w <- 1.3
> myfunc2 <- function(height, weight, a, b1, b2) {
     return(a + b1 * height + b2 * weight)
 }
> head(mapply(myfunc2, labdata["height"], labdata["weight"], a = intrcpt,
     b1 = coef.h, b2 = coef.w))
```

```
         height
[1,]  53.88521
[2,]  98.09585
[3,] 127.43197
[4,] 112.15614
[5,]  78.11626
[6,] 139.85304
```

Note: this method of passing "extra parameters" can by used for all of the other functions in this 'apply' family!

Summary

- If you need to do something for every element of a vector, every column of a data frame, or (more generally) every element of a list? If so, use `lapply` or `sapply`. The `lapply` will return a *list* of results (one per input element). The `sapply` function internally calls `lapply`, but then tries to *simplify* the result; if the output of each function call is a scalar value, the result will be a vector, whereas if the output of each function call is a vector of more than two elements, the result will be a matrix (with one column per input element).
- If you need to do something for every *row* of a `data.frame`, or every row or column of a `matrix`, use `apply`. It's unlikely to *run* any faster than a loop you write yourself, but it will sure be faster to program.
- Lastly, if you need to do something for every row (or vector element), where that "something" is a function of more than one column (or vector), use `mapply`. But remember that many basic computations are already vectorized. So don't use `mapply` to do elementwise addition of vectors `x` and `y` when you can simply do `x + y`!

8.2 Summarizing data by groups

Example: Recall our `labdata` dataset from before. Each experimental unit is categorized by its initial size (small, medium, large) and by its treatment type (control, experiment). With a dataset like this, often you will want to do some statistical summaries (and maybe apply analyses) to blocks of data corresponding to one or more unique factor levels. Here's how...

tapply The `tapply` function splits the elements of a vector into groups according to a factor (or combination of multiple factors), applies a specified function separately to each group, and returns the result. The first argument of `tapply` is the vector you want to operate on. The second argument is either a factor or a list of one or more factors; if you supply a vector instead of a factor, it will be coerced to factor. The third argument is the function you want to apply.

In practice, frequently the vectors you want to use are columns in a data frame, so we'll use columns from `labdata` in our examples. Remember that there are multiple ways to refer to a column vector! Consequently, the statements below all do the same thing – they just illustrate some different ways of extracting the desired columns from our data frame to pass to `tapply`:

```
> tapply(labdata$height, labdata$treatment, mean)
> tapply(labdata[["height"]], labdata[["treatment"]], mean)
> tapply(labdata[[3]], labdata[[2]], mean)
> tapply(labdata[, 3], labdata[[2]], mean)
> tapply(labdata[, "height"], labdata[, "treatment"], mean)
```

```
 Control     High      Low
4.457667 6.693713 8.015779
```

Notice the slightly different output if the second argument is a list of factors (including one or more factor columns of a data frame), rather than a second factor:

```
> tapply(labdata$height, labdata[2], mean)
> tapply(labdata[[3]], labdata["treatment"], mean)
```

```
treatment
  Control      High       Low
 9.666667 15.666667 12.666667
```

Now let's get a little fancier by supplying a list of two factors in the second argument:

```
> tapply(labdata$height, labdata[1:2], mean)
```

```
         treatment
init.size  Control     High      Low
   Large  4.507217 7.039736 8.824263
   Medium 4.659428 6.524584 7.314683
   Small  4.206358 6.516819 7.908390
```

Remember, `tapply` operates on a vector. Thus, the following statement will return an error because the first argument is a (one-column) data frame, and `tapply` cannot work directly on a data frame:

```
> tapply(labdata[3], labdata[2], mean)
```

```
Error in tapply(labdata[3], labdata[2], mean) :
  arguments must have same length
```

What if you want to summarize multiple columns of data at the same time?? Read on...

by This function is similar to tapply, but it works on data frames, and it produces rather different looking output. The form of the function call should look familiar. However, a key difference is that the first argument is a data frame. In reality, you *can* supply a vector instead, but `by` will internally convert it to a data frame before it does its thing. The following lines all do essentially the same thing, although note that the grouping labels will be slighly different in the output:

```
> by(labdata["height"], labdata$treatment, mean)
```

```
labdata$treatment: Control
[1] 4.457667
------------------------------------------------------------
labdata$treatment: High
[1] 6.693713
------------------------------------------------------------
labdata$treatment: Low
[1] 8.015779
```

If the second argument is given in the form of a named list rather than a single factor, the group labels will use the names. This tends to look nicer. You can do this either by supplying a list in the function call, or simply by passing factor(s) in their data frame column form. This works because a data frame *is* a named list! Notice how this differs from the previous output:

```
> by(labdata["height"], list(trt = labdata$treatment), mean)
```

```
trt: Control
[1] 4.457667
------------------------------------------------------------
trt: High
[1] 6.693713
------------------------------------------------------------
trt: Low
[1] 8.015779
```

```
> by(labdata["height"], labdata["treatment"], mean)
```

```
treatment: Control
[1] 4.457667
------------------------------------------------------------
treatment: High
[1] 6.693713
------------------------------------------------------------
treatment: Low
[1] 8.015779
```

Now for some real excitement. now let's supply a list of 2 factors in the second argument:

```
> by(labdata["height"], labdata[1:2], mean)
```

```
init.size: Large
treatment: Control
[1] 4.507217
------------------------------------------------------------
init.size: Medium
treatment: Control
[1] 4.659428
------------------------------------------------------------
init.size: Small
treatment: Control
[1] 4.206358
------------------------------------------------------------
init.size: Large
treatment: High
[1] 7.039736
------------------------------------------------------------
init.size: Medium
treatment: High
[1] 6.524584
------------------------------------------------------------
init.size: Small
treatment: High
[1] 6.51682
------------------------------------------------------------
init.size: Large
treatment: Low
[1] 8.824263
------------------------------------------------------------
init.size: Medium
treatment: Low
[1] 7.314683
------------------------------------------------------------
init.size: Small
```

```
treatment: Low
[1] 7.90839
```

How about 2 columns in the first argument?

```
> by(labdata[3:4], labdata[2], mean)
treatment: Control
    mean   height
9.666667 4.457667
------------------------------------------------------------
treatment: High
     mean    height
15.666667  6.693713
------------------------------------------------------------
treatment: Low
     mean    height
12.666667  8.015779
```

But watch out!!!

```
> by(labdata[3:4], labdata[2], sum)
treatment: Control
[1] 423.73
------------------------------------------------------------
treatment: High
[1] 670.8114
------------------------------------------------------------
treatment: Low
[1] 620.4734
```

Why is this happening? Note how these two functions work:

```
> mean(labdata[3:4])
     mean    height
12.666667  6.389053
> sum(labdata[3:4])
[1] 1715.015
```

If you want to be sure the columns are treated separately, simply use an appropriate columnwise function or expression:

```
> by(labdata[3:4], labdata[2], colSums)
> by(labdata[3:4], labdata[2], function(x) sapply(x, sum))
```

aggregate The `aggregate` function is similar to `by`, but only works with functions that return a scalar (`mean`, `max`, `min`, etc).

One form is `aggregate(vector, list, function)`. All these do the same thing:

```
> aggregate(labdata[["height"]], labdata[2], mean)
> aggregate(labdata[, "height"], labdata[2], mean)
> aggregate(labdata[, 3], labdata[2], mean)
```

```
  treatment         x
1   Control  9.666667
2      High 15.666667
3       Low 12.666667
```

Another form is `aggregate( list, list, function)`. The two statements below are identical to each other, but slightly different from the previous ones:

```
> aggregate(labdata["height"], labdata["treatment"], mean)

> aggregate(labdata[3], labdata[2], mean)
  treatment      mean
1   Control  9.666667
2      High 15.666667
3       Low 12.666667
```

...and now using list of 2 factors in the second argument:

```
> aggregate(labdata$height, labdata[1:2], mean)
  init.size treatment        x
1     Large   Control 4.507217
2    Medium   Control 4.659428
3     Small   Control 4.206358
4     Large      High 7.039736
5    Medium      High 6.524584
6     Small      High 6.516819
7     Large       Low 8.824263
8    Medium       Low 7.314683
9     Small       Low 7.908390
```

...and now using list of 2 variables in the first argument:

```
> aggregate(labdata[3:4], labdata[2], mean)
  treatment      mean   height
1   Control  9.666667 4.457667
2      High 15.666667 6.693713
3       Low 12.666667 8.015779
```

Note that 'sum' works like you would expect:

```
> aggregate(labdata[3:4], labdata[2], sum)
  treatment mean   height
1   Control  290 133.7300
2      High  470 200.8114
3       Low  380 240.4734
```

Finally, using 2 variables AND 2 factors:

```
> aggregate(labdata[3:4], labdata[1:2], mean)
  init.size treatment mean   height
1     Large   Control    4 4.507217
2    Medium   Control    9 4.659428
3     Small   Control   16 4.206358
4     Large      High    8 7.039736
5    Medium      High   15 6.524584
6     Small      High   24 6.516819
7     Large       Low    6 8.824263
8    Medium       Low   12 7.314683
9     Small       Low   20 7.908390
```

BUT NOTE: Whereas 'tapply' and 'by' can use functions that return non-scalar values:

```
> tapply(labdata[, 3], labdata[2], quantile)
$Control
  0%  25%  50%  75% 100%
   4    4    9   16   16

$High
  0%  25%  50%  75% 100%
   8    8   15   24   24

$Low
  0%  25%  50%  75% 100%
   6    6   12   20   20
> by(labdata[, 3], labdata[2], quantile)
treatment: Control
  0%  25%  50%  75% 100%
   4    4    9   16   16
------------------------------------------------------------
treatment: High
  0%  25%  50%  75% 100%
   8    8   15   24   24
------------------------------------------------------------
treatment: Low
  0%  25%  50%  75% 100%
   6    6   12   20   20
```

...'aggregate' cannot! The following line would produce an error:

```
> aggregate(labdata[, 3], labdata[2], quantile)

Error in aggregate.data.frame(as.data.frame(x), ...) :
  'FUN' must always return a scalar
```

Performance comparisons First let's make a big ol' data frame with 101 factor levels and a column of normal random numbers, then time the three different summarization functions:

```
> nreps <- 500
> test <- data.frame(a = rep(c(-50:50), each = nreps))
> test$x <- rnorm(nrow(test), test$a, 0.1)
> system.time(tapply(test$x, test$a, sum))
   user  system elapsed
  0.030   0.000   0.034
> system.time(by(test["x"], test["a"], sum))
   user  system elapsed
  0.090   0.000   0.093
> system.time(aggregate(test["x"], test["a"], sum))
   user  system elapsed
  0.050   0.000   0.053
```

For the special case of taking sums by groups, it's nice to have the speedy `rowsum` function in your back pocket. Here we'll use the `reorder=FALSE` argument, because sorting can be slow and we don't really care about it in this case.

```
> head(rowsum(test$x, test$a, reorder = FALSE))
         [,1]
-50 -24996.42
-49 -24497.76
-48 -24001.28
-47 -23502.61
-46 -22999.23
-45 -22499.55
> system.time(rowsum(test$x, test$a, reorder = FALSE))
   user  system elapsed
  0.000   0.000   0.001
```

8.3 Subsetting data frames

In the next couple of sections, we will work with two new sample tables:

```
> farms <- read.csv("farms.csv")
> flowers <- read.csv("flowers.csv")
> head(farms)
          site elevation
1 Happy Valley       130
2   Blue Ridge        85
3        Lot B         5
> head(flowers)
          site transect             species numFlowers
1 Happy Valley       27     Stellaria media          2
2 Happy Valley       26    Senecio vulgaris          2
3 Happy Valley       25     Stellaria media          1
4 Happy Valley       24     Stellaria media          6
5 Happy Valley       24 Lamium amplexicaule          3
6      Margate       10      Brassica nigra        133
```

Subsetting data is a common task in R. Often, you will load an entire dataset, but particular analyses and other tasks will only be based on part of the data.

`$`, `[]`, `[[]]` You've already been introduced to these basic indexing operators, so this section is really just a reminder that they can be used to accomplish data subsetting.

```
> farms["site"]
          site
1 Happy Valley
2   Blue Ridge
3        Lot B
> farms[farms$elevation>100, "site"]  # dimension info is discarded!
[1] Happy Valley
Levels: Blue Ridge Happy Valley Lot B
> farms[farms$elevation>100, "site", drop=FALSE]
          site
1 Happy Valley
```

subset The `subset` function essentially provides a more readable way to accomplish tasks that can achieved using the basic indexing operators. One usage is to subset a data frame to include only certain *rows*, based on one or more criteria. These criteria are given by the `subset` argument, which is the second function argument. If you have multiple criteria, these combined using `&` (logical AND) or `|` (logical OR) operators.

```
> subset(flowers, species == "Lamium amplexicaule")
          site transect             species numFlowers
5 Happy Valley       24 Lamium amplexicaule          3
> subset(flowers, !is.na(numFlowers) & numFlowers > 10)
         site transect        species numFlowers
6     Margate       10 Brassica nigra        133
8     Margate        7 Brassica nigra         42
10    Margate        6 Brassica nigra         15
12 Blue Ridge       44 Galium aparine         16
```

Alternatively, by using the `select` argument, one can restrict the result to include only certain *columns*. Note that in database terminology, this emulates a SQL `SELECT` statement.

```
> subset(farms, select = site)
          site
1 Happy Valley
2   Blue Ridge
3        Lot B
> subset(farms, select = c("site", "elevation"))
          site elevation
1 Happy Valley       130
2   Blue Ridge        85
3        Lot B         5
> subset(farms, select = -elevation)
          site
1 Happy Valley
2   Blue Ridge
3        Lot B
```

And last but not least, one can combine these usages in order to subset both *rows and columns* at the same time.

```
> subset(farms, site != "Lot B", select = -site)
  elevation
1       130
2        85

> merge(farms, flowers)
           site elevation transect             species numFlowers
1    Blue Ridge        85       44      Galium aparine         16
2    Blue Ridge        85       44     Stellaria media          5
3    Blue Ridge        85       44   Medicago lupulina          4
4    Blue Ridge        85       43   Medicago lupulina          3
5    Blue Ridge        85       43     Stellaria media          1
6  Happy Valley       130       27     Stellaria media          2
7  Happy Valley       130       24     Stellaria media          6
8  Happy Valley       130       24 Lamium amplexicaule          3
9  Happy Valley       130       26    Senecio vulgaris          2
10 Happy Valley       130       25     Stellaria media          1
```

```
> subset(merge(farms, flowers), select = c("species", "elevation"),
    species != "none")
```

```
               species elevation
1       Galium aparine        85
2      Stellaria media        85
3    Medicago lupulina        85
4    Medicago lupulina        85
5      Stellaria media        85
6      Stellaria media       130
7      Stellaria media       130
8  Lamium amplexicaule       130
9     Senecio vulgaris       130
10     Stellaria media       130
```

```
> unique(subset(merge(farms, flowers), select = c("species", "elevation"),
    species != "none"))
```

```
              species elevation
1      Galium aparine        85
2     Stellaria media        85
3   Medicago lupulina        85
6     Stellaria media       130
8 Lamium amplexicaule       130
9    Senecio vulgaris       130
```

8.4 Combining data frames

rbind/cbind The `rbind` and `cbind` are commonly used to combine data by rows or columns, respectively.

```
> extrafarm <- data.frame(site = "Honeybrook", elevation = 100)
> extrafarm
```

```
       site elevation
1 Honeybrook       100
```

```
> farms2 <- rbind(farms, extrafarm)
> farms2
```

```
         site elevation
1 Happy Valley       130
2   Blue Ridge        85
3        Lot B         5
4   Honeybrook       100
```

```
> extravar <- data.frame(area = c(140, 350, 130, 55))
> cbind(farms2, extravar)
```

```
         site elevation area
1 Happy Valley       130  140
2   Blue Ridge        85  350
3        Lot B         5  130
4   Honeybrook       100   55
```

If you `cbind` or `rbind` objects that don't have matching dimension lengths, sometimes R will report an error, sometimes R will "recycle" the shorter object and report a warning, and sometimes R will apply recycling without reporting an error. The rules differ depending both on what you are combining with what (matrices with matrices, matrices with data frames, data frames with

vectors, etc.), and (in some cases) whether the smaller object would be recycled a whole number of times.

As an example, when `cbind`ing a data frame to shorter data frame or vector, recycling will be used (silently) if the shorter object can be recycled a whole number of times, but return an error otherwise. But `cbind`ing a data frame to a matrix will only work if they have the same exact number of rows. In the examples below, it's a good bet that the recycled result is nonsense, in which case you'd probably prefer to get an error message!

```
> # cbind and rbind
> try(cbind(farms2, c(140,350,50))) # error!
> try(cbind(farms2, data.frame(area=c(140,350,50)))) # error!
> cbind(farms2, data.frame(area=c(140,350))) # recyling
          site elevation area
1 Happy Valley       130  140
2   Blue Ridge        85  350
3        Lot B         5  140
4   Honeybrook       100  350
> cbind(farms2, c(140,350)) # recycling
          site elevation c(140, 350)
1 Happy Valley       130         140
2   Blue Ridge        85         350
3        Lot B         5         140
4   Honeybrook       100         350
> try(cbind(farms2, matrix(c(140,350,50), ncol=1))) # error!
```

merge The `merge` function is used to join two data frames by one or more columns. The default behavior is to match on all common columns, based on column name:

```
> head(merge(farms, flowers, sort = FALSE))
          site elevation transect             species numFlowers
1 Happy Valley       130       27     Stellaria media          2
2 Happy Valley       130       24     Stellaria media          6
3 Happy Valley       130       24 Lamium amplexicaule          3
4 Happy Valley       130       26    Senecio vulgaris          2
5 Happy Valley       130       25     Stellaria media          1
6   Blue Ridge        85       44      Galium aparine         16
```

Note that by default, this is an *inner join* in database lingo, meaning that only matched records are returned.

You can also tell the `merge` function to join the data frames using particular columns. If the column names are the same in both data frames, simply specify them using the `by` argument. If the names differ, use the `by.x` and `by.y` arguments:

```
> merge(farms, flowers, by = "site")
> merge(farms, flowers, by.x = "farmsite", by.y = "location")
```

Sometimes it can be useful to indicate the join columns by number instead, or even to join on the basis of row names (here denoted as column number 0). Both of these would produce non-sensical results using our sample data, but the expressions below would do the trick:

```
> merge(farms, flowers, by = 2)
> merge(farms, flowers, by = 0)
```

Using database lingo, here is a summary of how to achieve different kinds of joins using particular arguments of `merge`:

Inner join Return only records that are matched in both tables. As shown above, this is the default behavior of `merge`.

Right outer join Force return of ALL records of second table

```
> merge(farms, flowers, all.y = TRUE)
           site elevation transect                species numFlowers
1    Blue Ridge        85       44         Galium aparine         16
2    Blue Ridge        85       44        Stellaria media          5
3    Blue Ridge        85       44      Medicago lupulina          4
4    Blue Ridge        85       43      Medicago lupulina          3
5    Blue Ridge        85       43        Stellaria media          1
6  Happy Valley       130       27        Stellaria media          2
7  Happy Valley       130       24        Stellaria media          6
8  Happy Valley       130       24    Lamium amplexicaule          3
9  Happy Valley       130       26       Senecio vulgaris          2
10 Happy Valley       130       25        Stellaria media          1
11      Margate        NA        7         Erodium botrys          7
12      Margate        NA       10         Brassica nigra        133
13      Margate        NA       10 Centaurea solstitialis          1
14      Margate        NA        7         Brassica nigra         42
15      Margate        NA        6         Brassica nigra         15
16      Margate        NA        6  Carduus pycnocephalus          3
```

Left outer join Force return of ALL records of first table

```
> merge(farms, flowers, all.x = TRUE)
           site elevation transect             species numFlowers
1    Blue Ridge        85       44      Galium aparine         16
2    Blue Ridge        85       44     Stellaria media          5
3    Blue Ridge        85       44   Medicago lupulina          4
4    Blue Ridge        85       43   Medicago lupulina          3
5    Blue Ridge        85       43     Stellaria media          1
6  Happy Valley       130       27     Stellaria media          2
7  Happy Valley       130       24     Stellaria media          6
8  Happy Valley       130       24 Lamium amplexicaule          3
9  Happy Valley       130       26    Senecio vulgaris          2
10 Happy Valley       130       25     Stellaria media          1
11       Lot B          5       NA               <NA>         NA
```

Right outer join Force return of ALL records of second table

```
> merge(farms, flowers, all.y = TRUE)
           site elevation transect                species numFlowers
1    Blue Ridge        85       44         Galium aparine         16
2    Blue Ridge        85       44        Stellaria media          5
3    Blue Ridge        85       44      Medicago lupulina          4
4    Blue Ridge        85       43      Medicago lupulina          3
5    Blue Ridge        85       43        Stellaria media          1
6  Happy Valley       130       27        Stellaria media          2
7  Happy Valley       130       24        Stellaria media          6
8  Happy Valley       130       24    Lamium amplexicaule          3
```

```
9  Happy Valley       130       26       Senecio vulgaris          2
10 Happy Valley       130       25        Stellaria media          1
11      Margate        NA        7         Erodium botrys          7
12      Margate        NA       10         Brassica nigra        133
13      Margate        NA       10 Centaurea solstitialis          1
14      Margate        NA        7         Brassica nigra         42
15      Margate        NA        6         Brassica nigra         15
16      Margate        NA        6  Carduus pycnocephalus          3
```

Full outer join Force return of ALL records of BOTH tables

```
> merge(farms, flowers, all = TRUE)
           site elevation transect                species numFlowers
1    Blue Ridge        85       44         Galium aparine         16
2    Blue Ridge        85       44        Stellaria media          5
3    Blue Ridge        85       44      Medicago lupulina          4
4    Blue Ridge        85       43      Medicago lupulina          3
5    Blue Ridge        85       43        Stellaria media          1
6  Happy Valley       130       27        Stellaria media          2
7  Happy Valley       130       24        Stellaria media          6
8  Happy Valley       130       24    Lamium amplexicaule          3
9  Happy Valley       130       26       Senecio vulgaris          2
10 Happy Valley       130       25        Stellaria media          1
11       Lot B          5       NA                   <NA>         NA
12      Margate        NA        7         Erodium botrys          7
13      Margate        NA       10         Brassica nigra        133
14      Margate        NA       10 Centaurea solstitialis          1
15      Margate        NA        7         Brassica nigra         42
16      Margate        NA        6         Brassica nigra         15
17      Margate        NA        6  Carduus pycnocephalus          3
```

Cross-product What if the two data frames do not share any common names? In that case, all possibly pairwise combinations of rows are returned! In general this is likely to produce non-sensical results.

```
> head(merge(farms, subset(flowers, select = -site)), 10)
           site elevation transect          species numFlowers
1  Happy Valley       130       27  Stellaria media          2
2    Blue Ridge        85       27  Stellaria media          2
3         Lot B         5       27  Stellaria media          2
4  Happy Valley       130       26 Senecio vulgaris          2
5    Blue Ridge        85       26 Senecio vulgaris          2
6         Lot B         5       26 Senecio vulgaris          2
7  Happy Valley       130       25  Stellaria media          1
8    Blue Ridge        85       25  Stellaria media          1
9         Lot B         5       25  Stellaria media          1
10 Happy Valley       130       24  Stellaria media          6
```

9 Advanced goodies

9.1 Manipulating text

To apply partial text pattern matching on an object, use the `grep` function. This function uses a user-supplied pattern (called a *regular expression*) to match text in a vector, returning (by default) the index number of all matching elements. Typically you use this result to index the vector, but in some cases you might want to tell grep to return the actual matched text values instead.

```
> grep("Brassica", flowers$species)
[1]  6  8 10
> flowers[grep("Brassica", flowers$species), ]
      site transect        species numFlowers
6  Margate       10 Brassica nigra        133
8  Margate        7 Brassica nigra         42
10 Margate        6 Brassica nigra         15
> grep("Brassica", flowers$species, value = TRUE)
[1] "Brassica nigra" "Brassica nigra" "Brassica nigra"
```

A related pair of functions, `sub` and `gsub` are used not just for finding patterns in text, but replacing those patterns with different text. Note that `sub` will only replace the first occurrence of the pattern encountered in any given character element, whereas `gsub` will replace all occurrences.

```
> sometext <- c("hey_there", "hey_there_you!")
> sub("_", "-", sometext)
[1] "hey-there"      "hey-there_you!"
> gsub("_", "-", sometext)
[1] "hey-there"      "hey-there-you!"
```

Note that just like with almost all other R functions, `sub` and `gsub` don't actually modify the vector you give it. If you want to "keep" the changes, you'll need to assign the result to something (possibly the same name as the input vector, if you want to replace it).

Incidentally, the patterns we used in the examples above are extremely simple. You can do all kinds of powerful things with these patterns, which are called *regular expressions*. For more information, you might want to start with the R help page at `?regexp`.

9.2 Accessing external databases

R can access data stored in an external database application, including:

- PostgreSQL
- MySQL
- Oracle
- SQLlite
- MS Access

If you know SQL, you can send statements to the database application, have the application do heavy lifting like joins and unions and arbitrary queries, and then simply return the result to R as a data frame.

9.3 Working with files and directories

R provides a handful of functions for interacting with files in your working directory. This can save you the trouble of having to open a separate file browser to list or delete files, or having to open a separate text editor to read text files. Some of these functions appear below.

Get and set your current working directory:

```
> getwd()
[1] "/home/regetz"
> setwd("..")
> getwd()
[1] "/home"
```

Note that `setwd` does not give confirmation, though it will complain if it fails. You can either specific absolute path names, or choose a path relative to your current working directory (as in the example above). If you're constructing a path in a script that you want to use on different operating systems, it is safer to use `file.path` to build up the path from the component directories. This function always uses the default file seperator on your OS (this is stored in .Platform$file.sep):

```
> file.path("c:", "foo", "bar", "somefolder")
[1] "c:/foo/bar/somefolder"
```

Now let's list all *.csv files in the current working directory, and get some basic information about one of them:

```
> list.files(pattern = ".csv")
[1] "farms.csv"   "flowers.csv" "labdata.csv"
> file.info("labdata.csv")
            size isdir mode               mtime               ctime
labdata.csv 4381 FALSE  644 2010-04-09 11:21:53 2010-04-09 11:21:53
                         atime  uid  gid  uname grname
labdata.csv 2010-04-27 16:36:24 1000 1000 regetz regetz
```

You can directly open and even edit any text file (e.g. "mynotes.txt") using the following statements. Note that the specific behavior of these functions is OS-dependent; e.g. in Linux the file may be displayed by `file.show` in the current window using the `less` program, while in Windows XP the file may be displayed in Notepad. With the `file.edit` function, you can supply a specific `editor` argument if you wish

```
> file.show("mynotes.txt")
> file.edit("data.csv")
> file.edit("data.csv", editor = "notepad")
> file.edit("data.csv", editor = "vi")
```

If you wish, you can even delete a file in the working directory. This might be useful if you have a script that generates some temporary file output when it runs. But use this carefully!

```
> unlink("filename")    # File is deleted silently
> file.remove("filename")    # As above, but success is confirmed
```

Finally, note that you can also send commands directly to the operating system. Linux and Mac OS X users in particular may prefer to issue shell commands directly, rather than using the above wrapper functions. For example:

```
> system("firefox http://www.nceas.ucsb.edu")
```

9.4 R tricks

Reading data from the clipboard Often it can be really useful to "paste" data into an R `data.frame`. For example, maybe you have a table embedded in a text file or email message, or perhaps someone sent you an Excel worksheet that contains a useful table amidst lots of other junk, and you just want to grab the relevant data rather than having to clean up the file to read into R. You can! Just copy the data to the clipboard, then do one of the following in R:

```
> mydat <- read.table("clipboard")  # on Windows or Linux
> mydat <- read.table(pipe("pbPaste"))  # on OS X
```

Note that R reads from the clipboard just like it would from a file, so all the usual rules apply about whether to use `read.table` versus `read.csv`, whether you need additional arguments, etc. The only limitation I'm aware of is that there may be size limitations to what fits on the clipboard. So do be sure to check the resulting `data.frame` to make sure it has all the rows and columns you expect.

9.5 Constants and stuff

A small handful of constants come with R. These can be handy, if for no other reason to save you some typing from time to time.

- `pi`
- `letters`, `LETTERS`
- `month.name`, `month.abb`
- `state.name`, `state.abb`, `state.area`, `state.division`, `state.region`

10 Programming

10.1 If you must loop...

If you do use an explicit `for` loop, it is faster to use "Perl-style" looping rather than "C-style" looping, in the following way. This means looping over the object directly, rather looping over an index:

```
> # SLOWER (C-style):
> for(i in 1:length(my_vector)) {
    some_function(my_vector[i])
  }
> # FASTER (Perl-style):
> for(element in my_vector) {
    some_function(element)
  }
```

10.2 Object systems in R

R features two main object systems, called S3 and S4. Both support the notion of *generic functions*, with *method dispatch* determined by the argument class.

S3 This system is widespread in R, and is easy to implement. If `foo` has class `lm`, then the expression `summary(foo)` actually invokes the method `summary.lm(foo)`. However, for better or worse, an object's class is just a attribute (character string) that can be modified arbitrarily – there is no validation to ensure, for example, that an object of class `lm` really is an `lm` object! Another limiting aspect of S3 is that method dispatch is based only on the class of the first function argument.

S4 This system is newer and more powerful, but more complicated to understand and implement. More formal class and method definitions. Objects are explicitly created via new(), and checked for validity against the class definition. Method dispatch is more sophisticated, mostly insofar as it can be based on the classes of multiple function arguments.

A Menagerie of (base) R functions

This is not a complete listing! First of all, it only focuses on functions that come in the *base* part of a standard R installation. Secondly, it primarily focuses on functions for working with vectors (remember that this includes columns of data frames) and matrices (remember that these are just vectors with two dimensions); no plotting functions, stats functions, programming tools, etc. And last but not least, this only includes a subset of these functions! Nevertheless, this should give you a flavor for some very different *kinds* of operations you can perform on data. If you want to do something that can't be done with these functions (or some combination thereof), odds are good that such a function exists!

For documentation on any of the functions below, simply ask R (e.g., `?is.numeric`).

A.1 Vector operations

Some of the functions below can be applied to *any* type of vectors, whereas others only work on certain vector types. For the latter functions, note that R will "coerce" vectors to the right type when possible. For example, if a function that does numeric computation is applied to a logical vector, it will simply convert each `TRUE` to 1 and each `FALSE` to 0, then do the computation.

A.1.1 Summary information

You can think of these functions as being responsible for finding something out about a vector, and (usually) returning a single value.

What kind of vector is it?

- `class`, `mode`, `storage.mode`, `typeof`
- `is.numeric`, `is.logical`, `is.integer`, `is.character`, `is.factor`

Summarize any kind of vector

- `length`

Summarize numeric vector

- `sum`
- `mean`, `median`, `var`, `sd`
- `min`, `max`, `range`

Summarize logical vector

- `any`, `all`

A.1.2 Element-wise information

Here, the purpose is not summary information about the vector as a whole, but information about *each element*. The returned object is another vector of the same length as the input vector, although it may be of a different type. The returned vector tells you something about every element in the input vector. This might be used subsequently as new data (e.g., assigned to a new column in the data frame), or used to identify a *subset* of data you subsequently want to extract from or modify in the original vector or data.frame column.

Return TRUE or FALSE for every element of a vector

- `duplicated`
- `%in%`
- `is.na`, `is.na`, `is.infinite`, `is.finite`

Return factor for every element of a vector

- `cut`

Operate on every logical, return indices

- `which`

Operate on every character, return numeric

- `nchar`

Operate on every character, return logical

- `nzchar`, `grepl`

Return indices of all vector elements

- `order`, `rank`

Return indices of certain vector elements

- `match`
- `grep` (with value=FALSE)
- `which.max`, `which.min`

A.1.3 Transformation

Another common action is to do some operation elementwise to a column of data. This might be a numeric transformation or a search-and-replace on character strings, just to give two examples. The point is that the result of this action is itself a vector of the same length and type (usually) as the input vector. You might then assign this back to the original column, particularly in cases where your objective was to *fix* or *update* the data. Or, if your objective was to create new data, you would instead assign this to a new column in your data frame.

Operate on every numeric element, return numeric vector

- `abs`, `sqrt`
- `log`, `logb`, `log10`, `log2`, `log1p`, `exp`, `expm1`
- `cos`, `acos`, `sin`, `asin`, `tan`, `atan`
- `cosh`, `acosh`, `sinh`, `asinh`, `tanh`, `atanh`
- `besselI`, `besselK`, `besselJ`, `besselY`
- `gamma`, `lgamma`, `psigamma`, `digamma`, `trigamma`

- ceiling, floor, trunc, round, signif, zapsmall
- cumsum, cumprod, cummin, cummax
- scale
- jitter

Operate on every character element, return character vector

- substr
- sub, gsub
- tolower, toupper, chartr, casefold

Operate on every complex element, return complex vector

- Re, Im, Mod, Arg, Conj

Rearrange vector elements

- rev
- sort
- sample

A.1.4 Miscellany

Comparing two vectors

- setequal, setdiff, intersect, union
- pmin, pmax, pmin.int, pmax.int

Other

- diff
- unique

A.2 Factor-specific operations

[definitely not comprehensive!]

Factor-specific operations

- gl, levels, nlevels

A.3 Matrix-specific operations

Get information about a matrix-like object

- col, row
- nrow, ncol, dim

How is your linear algebra?

- t, aperm
- diag
- crossprod, tcrossprod
- outer, %o%
- kronecker, %x%
- lower.tri, upper.tri
- det, determinant
- backsolve, forwardsolve
- eigen, svd
- qr, chol, chol2inv
- kappa, kappa.tri, rcond

Higher level tabular functions

- colSums, rowSums, colMeans, rowMeans
- col, row
- max.col

A.4 Other

Probability stuff

- beta, lbeta, choose, lchoose, factorial, lfactorial