INTRODUCTION TO PROGRAMMING

jumping rivers

JUMPINGRIVERS.COM

Contents

Available R courses

See jumpingrivers.com for dates.

All courses can be given onsite

Statistics & analytics

- Statistical modelling with R: a one day course on standard statistical modelling techniques.
- Predictive analytics: cutting edge statistical predictive algorithms.
- Bioconductor: analysing microarray and RNA-seq data.
- Survial analysis with R: an introduction to the key statistical methods.

Bespoke courses also available.

Programming courses

- Introduction to R: a first step.
- Programming with R.
- Advanced programming: OOP and function closures.
- Efficient R programming. Code running slowly?
- Building an R package. A short course on constructing your own R package.

Please ask about our consulting services.

Graphics courses

- Advanced graphics with R. Creating sophisticated graphics using ggplot2.
- Shiny: Turn your analyses into interactive web applications.

Statistical consultancy

If you have a project you would like to discuss, please get in touch. We have expertise in a wide range of topics such as, big data, dashboard and questionnaire design. Previous clients include KPMG, Burberry, and Tesco Bank.

“A BIG COMPUTER, A COMPLEX ALGORITHM AND A LONG TIME DOES NOT EQUAL SCIENCE.”

ROBERT GENTLEMAN.

“IT’S EASY TO LIE WITH STATISTICS; IT IS EASIER TO LIE WITHOUT THEM.”

FREDERICK MOSTELLER.

1 Control statements: for loops

During data analysis, we often want to repeat similar tasks. For example, reading in a large number of CSV files from a directory or repeating the same operation on different rows of a datasets. The easiest option - that will quickly lead to unpleasantness - is to simply *copy and paste*[1] Resit this urge!

[1] First rule of R club: if you copy and paste more than twice, you're doing it wrong.

In this chapter we'll learn about imperative programming, in particular `for` loops[2] Loops provide an easy way of repeating the same operation.

[2] A similar concept is the `while()` loop that we won't cover (due to time) in this course.

Example

Suppose we have the following data set

```
dd = data.frame(x = rnorm(10), y = rnorm(10), z = rnorm(10))
```

We wish to calculate the maximum value in each column. We could accomplish this with some copying and pasting

```
max_x = max(dd[, 1])
max_y = max(dd[, 2])
max_z = max(dd[, 3])
```

This strategy is OKish for three columns, but doesn't scale to large data sets. Also it's easy for bugs to creep in. If we look at the code again, we see that on each line the column number is increasing by 1. This is perfect for a loop:

```
max_cols = numeric(ncol(dd))   # 1. Place to store output
for(i in seq_along(dd)) {      # 2. What to iterate over
    max_cols[i] = max(dd[, i]) # 3. Loop body
}
max_cols

## [1] 1.418885 1.848405 2.579600
```

Loops are a common feature in most programming languages. Once you've grasped the concept, it's easy to run a loop in a different language, e.g. Python.

Every loop has these three components

- Output: Where we store the output of the loop. `ncol(dd)` returns the number of columns in the data frame `dd` - in this case 3. The function `numeric()` then creates a vector of size 3, with each value equal to 0.

- Iterator: A for loop takes a variable and iterates through another option[3]. In the example above, we are iterating through

```
seq_along(dd)

## [1] 1 2 3
```

and the iterator is `i`[4]. At the end of the for loop, we have

```
i

## [1] 3
```

[3] Typically a vector.

[4] The iterator can be called anything. However, it's useful to stick with `i`, `j` and `k`.

- Body: The calculations we want done. This can be one line or multiple lines. In this code, it's calculating the maximum value of column `i` of the data frame `dd`.

Example: Distribution of p-values

In statistics a way of assessing a model is to perform multiple simulations[5]. A standard statistical routine is the one sample t-test. Here we are interested in two competing hypothesis. Is the true underlying mean of your sample equal to zero or is it equal to some other value[6]. This is usually written as

$$H_0 : \mu = 0 \quad \text{and} \quad H_1 : \mu \neq 0\,.$$

[5] For example, bootstrapping or cross-validation

[6] Of course if you're Bayesian, hypothesis testing is a bit silly

To perform a standard t-test in R we use the `t.test()` function.

```
x = rnorm(10) # Simulate 10 numbers from a standard Normal.
t.test(x, mu = 0)$p.value # Extract the p-value

## [1] 0.6052327

p = signif(t.test(x)$p.value, 2)
```

In this case, since $p = 0.61$, we do not have enough evidence to reject[7] H_0.

[7] Technical point: we can never *accept* H_0.

Let's perform a simulation study to determine the distribution of p-values. We will simulate 10,000 data sets of size $n = 10$ and calculate the corresponding p-value

```
p_values = numeric(10000)
for(i in seq_along(p_values)) {
    x = rnorm(10) # Generate new data set
    p_values[i] = t.test(x)$p.value # Store value
}
hist(p_values)
```

Figure 1.1 shows that the distribution of p-values under the null hypothesis follows a uniform distribution.

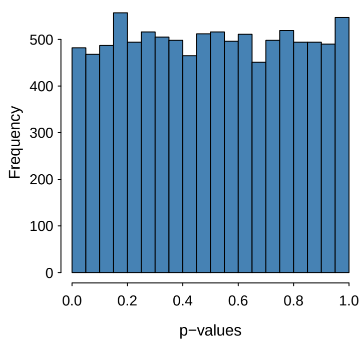


Figure 1.1: Distribution of p-values under the H_0 hypothesis. The values should be Uniformly distributed.

Example: looping through a directory

Suppose we have a large number of files and we want to read them into R. We want to read in the `.txt` files, but ignore the `README`. Here's a traditional computing solution. First we create a variable `d`

```
d = NULL
```

We are going to use `d` to store the data frames. Next, we obtain a list of files in the current working directory

We'll cover this in chapter 5, but to get your current working directory, type `getwd()`

```
(filenames = list.files(pattern = "*.txt"))

## [1] "A.txt" "B.txt" "C.txt"
```

Finally we loop over the files and make sure they contain data:

```
for(filename in filenames) {
    message("Reading file: ", filename)
    d = rbind(d, read.csv(filename))
}

## Reading file:  A.txt
## Reading file:  B.txt
## Reading file:  C.txt
```

A few comments:

- The `message()` function prints the current `filename` to the screen; helpful for debugging!
- `rbind` is used to combine data frames. A much more efficient way of implementing this function is to create a large empty data frame, and fill up the entries.
- We initially set `d = NULL`. When we combine a data frame with `NULL` using `rbind`, `NULL` disappears.
- The `pattern` argument in the `list.files` function:

  ```
  list.files(pattern="*.txt")
  ```

  avoids the `README` file.

Warning example: Summing numbers

Suppose we want calculate[8]

$$1^2 + 2^2 + 3^3 + \ldots + 10^2$$

Obviously the correct way of doing this is to make use of R's vector operations

[8] In mathematics we write this as $\sum_{i=1}^{10} i^2$.

```r
sum((1:10)^2)

## [1] 385
```

However occasionally you'll stumble across R code like[9]

```r
total = 0
for(i in 1:10) {
  total = total + i^2
}
total

## [1] 385
```

[9] If the programmer comes from a language such as C or Fortran, then this is the *obvious* way of doing this.

This `for` loop is slower and longer than the standard base R solution.

Looping through vectors

Occasionally we want to loop through values in a vector. The natural way to do this is[10]

[10] We could use `1:length(x)` instead of `seq_along(x)`, but what happens in if `x` is empty?

```r
total = 0; x = 4:6
for(i in seq_along(x)) {
  total = total + x[i]
}
total

## [1] 15
```

Of course, in R we could do this more easily using the `sum` function, i.e.

```r
sum(x)

## [1] 15
```

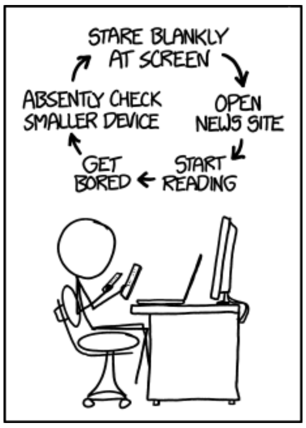


Figure 1.2: http://xkcd.com/1411/

2
Functions

A very powerful aspect of R is that it is relatively easy to write your own functions. Functions can take inputs (or arguments) and return a *single* object. We have already come across a number of functions. For example, `mean()`, `median()` and `plot()` are functions. In this chapter we look at how to construct our own functions.

Basic functions

Lets begin with an example of creating a simple function. The following function takes in a single argument x and returns x^2:

```
Fun1 = function(x) {
  return (x*x)
}
```

The key elements of R's function construction are:

- the word `function`;
- the brackets ( ) which enclose the **argument** list - this list may be empty;
- a sequence of statements in curly braces { };
- a `return` statement.

We call `Fun1()` in the following manner:

```
Fun1(5)

## [1] 25

(y = Fun1(10))

## [1] 100

## We can also pass a vector
z = c(1, 2, 3, 4); Fun1(z)

## [1]  1  4  9 16
```

Multiple arguments

Functions can have multiple arguments. For example,

```
Fun2 = function(x, y) {
  return (x*y)
}
Fun2(3, 4)

## [1] 12
```

Functions can also have no arguments

```
Fun3 = function() {
  x = 5
  return(x)
}
Fun3()

## [1] 5
```

Default arguments

We often create functions that have default arguments. For example, when we call the plot function

```
plot(x, y)
```

R uses *default* arguments for the colour of the points, labels, and axis ranges. Adding default arguments to the above functions is straightforward:

```
## Default argument
Fun4 = function(x = 1) {
  return (x*x)
}
```

When we call `Fun4()`, but omit the argument, R uses `x = 1` as a default. However, if we pass a value for `x`, R uses the new value instead. For example,

```
## Use the default argument
Fun4()

## [1] 1

## Pass a new value
Fun4(10)

## [1] 100
```

We can create functions that have a mixture of arguments:

```
Fun5 = function(x, y = 10) {
  return(x*y)
}
```

Figure 2.1: Rage: a poor (but satisfying) debugging technique.

Syntax matters

All computing languages are fussy with syntax. If you use the wrong syntax, at best the program will crash. At worst, you have introduced a hard to find bug.

Some obviously incorrect function calls are:

```
Fun1()

## Error in Fun1():  argument "x" is missing, with no default

Fun1("5")

## Error in x * x:  non-numeric argument to binary operator
```

The error messages (sometimes) give you an idea of what is going wrong.

A more useful function

The function below takes in a vector, plots a histogram and returns a vector containing the mean and standard deviation:

```
investigate = function(values) {
  hist(values)
  m_std = c(mean(values), sd(values))
  return(m_std)
}
```

Once we have created our function, we can put it to good use:

```
data(movies, package = "ggplot2movies")
investigate(movies$rating)
```

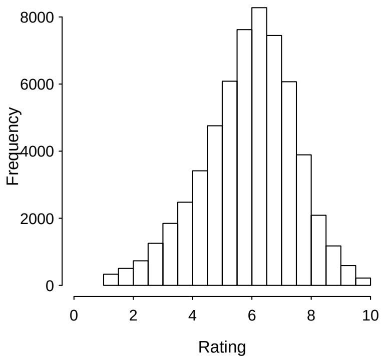


Figure 2.2: Histogram of movie ratings. Output from the `investigate` function.

and generate figure 2.2.

If you find yourself copying and pasting R code, then you should consider creating a function, as functions have a number of benefits:

- Functions are reusable - you only have to create it once;
- Functions reduce code length;
- Functions help with debugging.

Example: maximum values

Recall in Chapter 1 we used a loop to calculate the column maximums. This can be easily wrapped into a function

```
get_max_cols = function(dd) {
    max_cols = numeric(ncol(dd))
    for(i in seq_along(dd)) {
        max_cols[i] = max(dd[, i])
    }
    return(max_cols)
}
```

which we can use

```
get_max_cols(dd)

## [1] 1.3369898 0.1524523 1.1727799
```

Variable scope

When we call a function, R first looks for *local* variables, then *global* variables. For example, the following function uses a *global* variable:

```
blob = 5
Fun6 = function() {
    return(blob)
}
## Function uses the global blob
Fun6()

## [1] 5
```

While `Fun7` uses a local variable:

```
## Here we use a local variable
Fun7 = function() {
  blob = 6
  return(blob)
}
Fun7()

## [1] 6
```

Remember, local doesn't affect global

```
blob

## [1] 5
```

As a general rule, functions should only use **local** variables. This makes your code more portable and less likely to have bugs[1]

[1] We've glossed over a couple of points regarind functions, such as lexical scope and lazy evaluation. We cover these details in our Advanced R course.

Further information on functions

In all my examples of functions, I have { } and `return`. However, these are (in some cases) optional. If you omit the `return` from your function, then if the last expression is a single expression, that value will be returned instead. For example,

```
f = function(x) {
  ## Do some stuff
  x
}
f(100)

## [1] 100
```

Also, if your function is a single line, then you can construct a function without brackets, i.e.

```
Ratio = function(x) mean(x)/median(x)
Ratio(movies$length)

## [1] 0.9148653
```

This can be helpful in `apply` functions

Putting it all together

Reusing the code from the last chapter, we create a function with two arguments. The first argument `path` enables us to easily change directory. The second argument allows to specify file type:

```
read_txt_files = function(path = ".", # Directory to look
                          pattern = "*") { #Type of files to list
  filenames = list.files(path=path, pattern=pattern)

  d = NULL
  for(filename in filenames)
    d = rbind(d, read.csv(filename))
  return(d)
}
```

We can now call the function

```
read_txt_files(path = "data/")
read_txt_files(path = "data/", pattern = "*.txt")
```

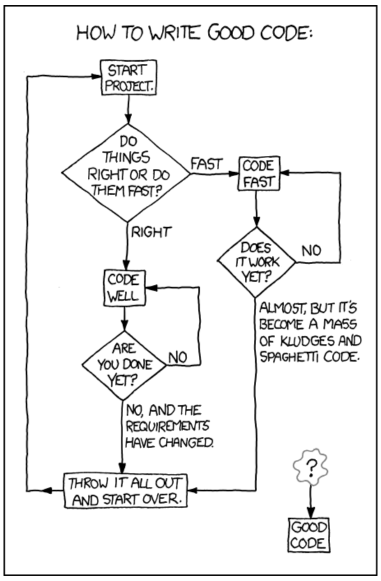

HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
CODE WELL
DOES IT WORK YET?
NO
ARE YOU DONE YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Figure 2.3: http://xkcd.com/844/

3
Conditionals

Conditional statements are features of a programming language which perform different computations or actions depending on whether a condition evaluates to `TRUE` or `FALSE`. They are used in almost all computer programs.

`If` statements

The basic structure of an `if` statement is:

```
if(expr) {
  ## do something
}
```

where `expr` is evaluated to be either `TRUE` or `FALSE`. The following example illustrates `if` statements in R:

```
x = 5; y = 5
## If x is less than 5, set
## y equal to 0
if(x < 5) {
  y = 0
}
y

## [1] 5
```

In previous code chunk `x < 5` evaluates to be `FALSE` so the following brackets are not evaluated. We test for greater than in a similar manner:

```
x = 5; y = 5
if(x > 0) {
  y = 0
}
y

## [1] 0
```

Here $x > 0$ evaluates to be `TRUE` so `y` is set equal to 0. If we wanted to determine if `x` was greater than or equal to zero, we would use the $>=$ operator.

`If else` statements

We can link together a number of `if` statements using the `if else` and `else` constructs. For example,

We use the `message()` function for printing statements to the screen. It's very helpful when trying to determine what's going on in a computer program. For example, suppose you program takes a long time to run. The `message` function can be used to give an idea of progress.

```
x = 5;
if(x > 10) {
  message("x is greater than 10")
} else if (x > 0) {
  message("x is greater than 0")
} else if(x > -5) {
  message("x is greater than -5")
} else {
  message("x must be less than -5")
}

## x is greater than 0
```

A few things to note:

- The final `else` is optional.
- When one of the conditions is evaluated to be `TRUE`, R ignores all remaining `if` statements. In the example above, `x` $>$ `0` and `x` $>$ `-5`, but only a single `message` statement is executed.

Example: The `investigate` function

Recall the investigate function in the previous chapter. Suppose we wished to add an option of only plotting the histogram depending on user input. This is straightforward with an `if` statement

```
## Default: produce a histogram
investigate = function(values, plot = TRUE) {
  if(plot) {
   hist(values)
  }
  m_std = c(mean(values), sd(values))
  return(m_std)
}
```

Example: Reading in files

Another common argument many functions have is `verbose`. This controls whether the function should produce output. For example,

```
read_txt_files = function(path = ".", pattern = "*",
                          verbose = TRUE) {

  filenames = list.files(path = path, pattern = pattern)

  d = NULL
  for(filename in filenames) {
      if(verbose) {
          message("Reading in: ", filname, " - ", nrow(filename), " rows")
      }
    d = rbind(d, read.csv(filename))
  }
  return(d)
}
```

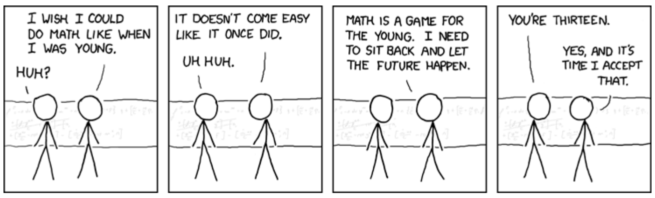

I WISH I COULD DO MATH LIKE WHEN I WAS YOUNG.
HUH?
IT DOESN'T COME EASY LIKE IT ONCE DID.
UH HUH.
MATH IS A GAME FOR THE YOUNG. I NEED TO SIT BACK AND LET THE FUTURE HAPPEN.
YOU'RE THIRTEEN.
YES, AND IT'S TIME I ACCEPT THAT.

Figure 3.1: http://xkcd.com/447/

4
The Apply Family

R has been designed with manipulating data in mind. Typically, we want to *apply* a function to a set of data. The set of data could be a row, column or separated according to a particular type.

The `apply` function

We use the `apply()` function when we want to *apply* the same function to every row or column of a data frame or a matrix. For example, suppose we have movie ratings[1]

[1] Each row should sum to 100, but this doesn't always occur due to rounding.

```
ratings = movies[, 7:16]
head(ratings, 2)

##    r1   r2  r3   r4   r5   r6   r7   r8  r9  r10
## 1 4.5  4.5 4.5  4.5 14.5 24.5 24.5 14.5 4.5  4.5
## 2 0.0 14.5 4.5 24.5 14.5 14.5 14.5  4.5 4.5 14.5
```

This data frame has 58788 rows and 10 columns

```
dim(ratings)

## [1] 58788    10
```

Suppose we wanted to calculate row median, then we have[2]

[2] The second argument indicates if we are working with rows (1) or columns (2).

```
## Row medians
apply(ratings, 1, median)
```

Or perhaps we want the column sum

```
## Column sums
apply(ratings, 2, sum)

##       r1       r2       r3       r4       r5       r6       r7       r8
## 412361.5 236467.5 277547.5 374764.5 575928.0 766546.0 914041.0 815742.0
##       r9      r10
## 526400.0 990814.0
```

which we can plot

```
plot(apply(ratings, 2, sum), type="l")
```

to get figure 4.1.

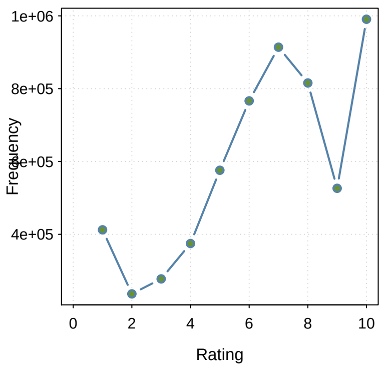


Figure 4.1: Plot movie rating frequency. Notice that the dip in ratings 2 & 9. Also the peak at 7.

WE CAN ALSO construct bespoke functions and apply that function to a row or column. For example, to calculate the average ratings

```
average_rating = function(x) {
  return(sum(x*(1:10))/100)
}

head(apply(ratings, 1, average_rating))

## [1] 6.375 6.230 9.685 7.800 4.360 5.875
```

Or using an anonymous function

```
r = apply(ratings, 1, function(x) sum(x*(1:10))/100)
```

This variable can then be used by another function. For example, if we sort the overall movie rating and plot

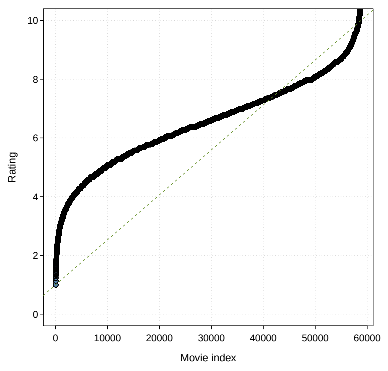


Figure 4.2: Plot of sorted movie ratings. If movie ratings were uniformly distributed

```
plot(sort(r))
abline(1, 9/length(r), col=2, lty=2)
```

we can clearly observe that movie ratings are not uniform - if they were uniform they would fall on the red line.

The `tapply` function

The `tapply()` function is very useful, but at first glance can be tricky to understand. It's best described using an example:

```
## The first two arguments are vectors
tapply(movies$length, movies$mpaa, mean)

##                NC-17        PG     PG-13         R
##  80.64271 110.18750  97.38258 104.97009 100.16997
```

In the above code, we have calculated the average movie length conditional on its MPAA rating. So the average length of a PG movie is 97 minutes and the average NC-17 movie length is 110 minutes. To extract the names and values from the `tapply()` command, we do the following

```
names(tapply(movies$length, movies$mpaa, mean))

## [1] ""      "NC-17" "PG"    "PG-13" "R"

as.vector(tapply(movies$length, movies$mpaa, mean))

## [1]  80.64271 110.18750  97.38258 104.97009 100.16997
```

Similar to `apply()`, we can construct bespoke functions to `tapply`:

```
tapply(movies$length, movies$mpaa,
       function(x) mean(x)/median(x))

##                 NC-17        PG     PG-13         R
## 0.8960301 1.1359536 1.0250797 1.0393078 1.0434372
```

With `tapply()` we can do we very interesting things. For example, in the next piece of code, we calculate the average movie length conditional on its rating

```
rating_by_len = tapply(movies$length, movies$rating, mean)
```

We can use `head()` to view the first few values:

```
head(rating_by_len)

##        1      1.1      1.2      1.3      1.4      1.5
## 64.91509 70.65909 66.47222 73.67568 65.87755 74.47458
```

and plot it in the standard way

```
plot(names(rating_by_len), rating_by_len)
```

Imagine trying to produce figure 4.2 in Excel!

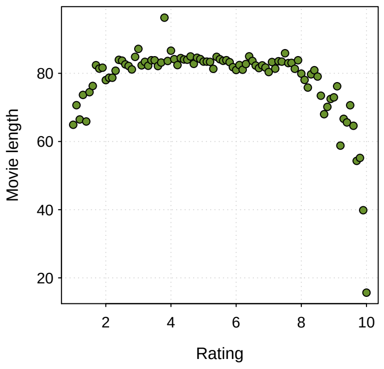


Figure 4.3: Plot of movie length conditional on its rating.

Other `apply` commands

There are a few other commands in the `apply` family, that can be useful:

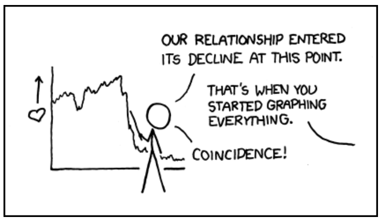
- `lapply()` and `sapply()`: apply a function to each element of a vector or a list.
- `eapply()`: applies a function to the named values from an 'environment' and returns the results as a list. We haven't covered environments in this course.
- `mapply()`: a multivariate version of `sapply()`.

Two sites that give an excellent overview of the `apply()` family are

http://stackoverflow.com/q/3505701/203420

and

http://goo.gl/VfFBhE


OUR RELATIONSHIP ENTERED ITS DECLINE AT THIS POINT.
THAT'S WHEN YOU STARTED GRAPHING EVERYTHING.
COINCIDENCE!
Figure 4.4: http://xkcd.com/523/

5
Using R Help

This is just a brief chapter to give you a few tips when looking for help. All functions have help files. For example, to see the help file for `plot`, just type:

```
?plot
```

When I look at help files, I tend to go straight to the example section at the bottom of the help file. You can either copy and paste the code, or use the `example` command, viz.

For a package to be accepted into CRAN, all examples must run.

```
example(plot)
```

ANOTHER USEFUL section in the help file is *See Also:*. In the `plot` help file, it gives pointers to 3d plotting.

Sometimes a package will contain vignettes. To browse any vignettes, we can use the handy function

```
browseVignettes(package = "jrProgramming")
```

Other sources of help

My favourite place for asking for help is

`http://www.stackoverflow.com`

You can find all R questions under the R tag:

`http://stackoverflow.com/questions/tagged/r`

There are a few useful questions & answers that are worth looking at:

- How to search for R materials.

  `http://stackoverflow.com/q/102056/203420`

- How to make a great R reproducible example?

  `http://stackoverflow.com/q/5963269/203420`

When asking a question, here are a few pointers:

1. Make your example reproducible.
2. Clearly state your problem. Don't confuse a statistical problem with an R problem.
3. Read a few other questions to learn the format of the site.
4. People aren't under **any** obligation to answer your question!

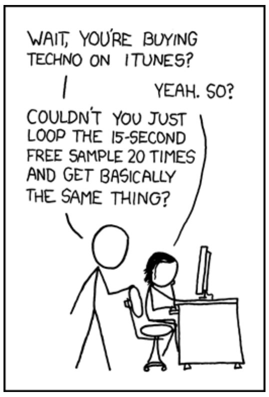

WAIT, YOU'RE BUYING TECHNO ON ITUNES?
YEAH. SO?
COULDN'T YOU JUST LOOP THE 15-SECOND FREE SAMPLE 20 TIMES AND GET BASICALLY THE SAME THING?

Figure 5.1: http://xkcd.com/411/

6
Possible R Workflows

For any serious data analysis problem, we can quickly generate a large number of R scripts and files. If we are not careful, this can lead to a very messy file structure that causes problems when we return to the analysis a few weeks later. This is particularly problematic when we need to redo analysis for a publication!

Directory structure

Each project that I work on has its own directory. In this directory, I tend to have the following sub-directories:

- `input`: this directory contains the *original* data files. If these files are `.xls`, it will also contain their `csv` counterparts. I avoid editing these files at all costs.

  When working I tend to go to extreme lengths to avoid altering the original data file. For example, I rename columns and do search/replace operations in R. This makes my analysis reproducible.

- `R`: this directory contains all my R scripts.
- `graphics`: I tend to label all figures as `figureX.pdf`. In practice, I can delete all the graphics in this directory and easily generate them again.
- `output`: a place to dump any output data.

The key principle is *reproducible research*. It should be straightforward to generate the `output` and `graphics` directories, given the `input` and `R` directories.

The `R` directory

For simple projects, I break my work flow into a few simple pieces.[1]

- `load.R`: this takes care of loading in all the data required.[2]
- `clean.R`: This is where all the ugly stuff lives. String manipulation, taking care of missing values, merging data frames, handling outliers, making column names consistent. At the end of this file,

[1] I got this idea from a stackoverflow answer: http://stackoverflow.com/a/1434424/203420

[2] Usually from the `input` directory.

I use the `rm` command to clean up my workspace. Occasionally, I merge `clean.R` and `load.R`.

- `func.R`: Contains all of the functions needed to perform the actual analysis. This file only loads in functions, it doesn't actually alter the data or generate output. If this file is altered, we don't need to reload `load.R` and `clean.R`.
- `do.R`: Calls the functions defined in `func.R` to perform the analysis. The first few lines source the other files:

```
source("clean.R")
source("func.R")
```

- `graphics.R`: Generates any publication plots. Each piece of R code generates a particular figure. The figures are saved in the graphics directory.

I find that this structure makes it easier to return to a project after a few months. It also keeps the code compartmentalised. For many analyses, `do.R` quickly grows. In this case, I just split the file into smaller, sensible sub files.

SORRY IT'S SUCH A DISASTER IN HERE.
WHOA— WHAT'S WRONG WITH ME?
MY ROOM NEVER LOOKS AS NICE AS THE ROOMS OTHER PEOPLE APOLOGIZE FOR.

Figure 6.1: http://xkcd.com/1267/

Working with directories

When separating out the different components to other directories, it is helpful to know a few R directory commands. Table 6.1 gives a few standard commands.

When you ask R to load a file, say

```
read.csv("movie.csv")
```

R checks its working directory for that file. To determine the current working directory, we use the following command:

```
getwd()
```

we can set the working directory using

Command	Summary
`list.files()`	Lists the files in the current directory. It has some useful arguments, such as `include.dirs` and `pattern`
`getwd()`	Prints the current working directory.
`setwd()`	Sets the current working directory.
`dir.create()`	Create a directory. Useful for programmatically creating a large number of output directories.

Table 6.1: Useful commands when working with directories.

```r
setwd("path_to_directory")
```

Other useful commands are `list.files()` for listing files and `dir.create()` for creating a directory.

Appendix

Course R package

This course has an associated R package. Installing this package is straightforward. First install the `drat` package

```r
install.packages("drat")
```

Then run the command[3]

[3] This adds a new repo URL to you list of repositories.

```r
drat::addRepo("jr-packages")
```

Then install the package as usual

```r
install.packages("jrProgramming")
```

To load the package, use

```r
library("jrProgramming")
```