A Walk Through Data Handling in R

Using the packages *dplyr* and *reshape2*

Steve Pederson
Bioinformatics Hub,
Level 4, Santos Petroleum Engineering Building,
University of Adelaide,
Adelaide, South Australia
5005

April 27, 2015

Chapter 1

Introduction

1.1 Welcome

Thank you for your attendance & welcome to today's walk through of the packages *dplyr* & *reshape2* Today is presented as part of the Stats Solutions ongoing R stream, by the University of Adelaide, Bioinformatics Hub. The Bioinformatics Hub is a centrally funded initiative from the Department of Vice-Chancellor (Research), with the aim of assisting & enabling researchers in their work. Training workshops & seminars such as this one are an important part of this initiative. The Bioinformatics Hub itself has a web-page at `http://www.adelaide.edu.au/bioinformatics-hub/`, and to be kept up to date on up-coming events and workshops, please join the internal Bioinformatics mailing list on `http://list.adelaide.edu.au/mailman/listinfo/bioinfo`.

Today's workshop has been prepared with generous technical support & advice provided by Dr Jono Tuke (*School Of Mathematical Sciences*), Dr Dan Kortschak (*Adelson Research Group*) & Associate Professor Gary Glonek (*School Of Mathematical Sciences*).

1.2 Workshop Introduction

In today's session we assume a very basic familiarity with R, and with the common data handling methods in Excel. If you're unsure about either, please ask for help as improving your understanding is the entire purpose of the session. Don't be shy!

Hopefully the statistics used in today's session are also intuitive, but again, please ask if you don't understand something. The intended focus is on data manipulation and handling rather that on the statistical methods themselves.

There is no rush to get through the material, and please take your time to explore everything you'd like to. Ask as many questions as you'd like as we are here to help.

Learning R is also just like every other piece of software you use. The more you use it & explore it, the more proficient you become. The scope of applications & uses for R is incredibly diverse and even those with years of experience in the language may have no idea about some functions that you may already be familiar with.

Questions for you to answer throughout today's session are given in separate boxes.

At the end of this workshop you should be able to:

- Open RStudio.
- Load data into R.
- Save your commands for later use.

We hope today's session will be useful in enabling you to continue and to advance your research.

1.3 Software Installation

Today's session will be entirely within the workspace provided by *R Studio*. This type of interface is commonly known as an *Integrated Development Environment* or IDE, and R Studio has become a very powerful tool for those of us working in R. In short, it provides an easy way to access all of the information & objects that you need in an easy & straight-forward manner. The look & layout of R Studio is also virtually identical regardless of whether you are operating within Windows, Linux or Mac.

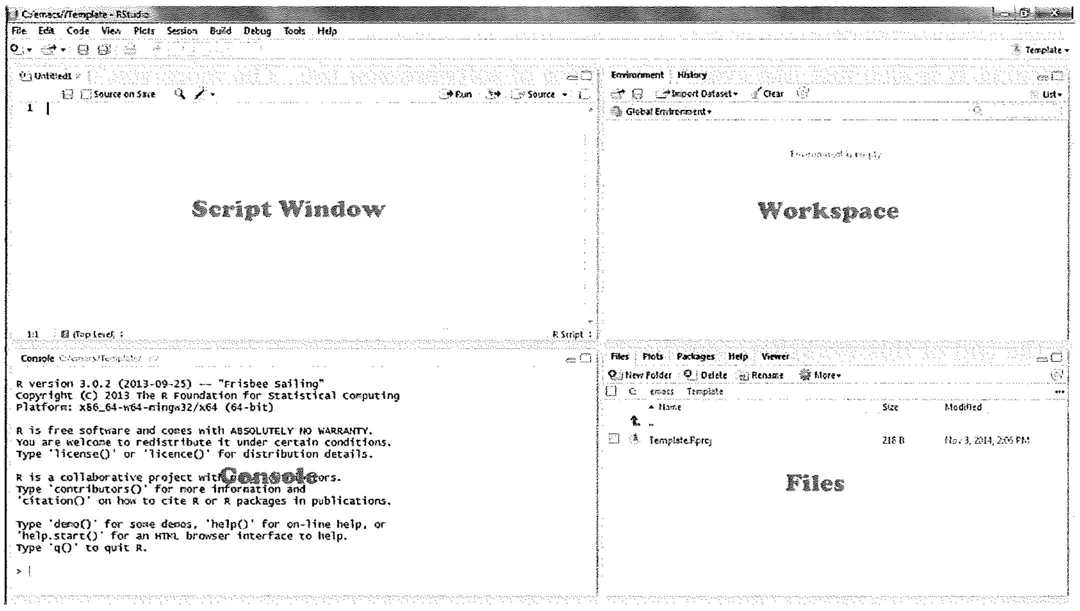
If you do not have either R, or R studio installed, please go to `http://cran.r-project.org/` to download the appropriate version of R for your computer, and `http://www.rstudio.com/` for R studio. These are relatively large installs so they make take a few minutes to download & install. Both should still run smoothly even on older computers too.

1.4 Working in R & R Studio

Once installed, open R Studio then select

```
File > New > R script
```

from the drop down menus. This should give you a view which looks like Figure 1.1. Whilst there may be a slight discrepancy here or there depending on which version of RStudio you



```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with [illegible]ors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 1.1: The basic R Studio layout

have installed, the core ideas will remain the same.

1.4.1 The Console

All the exciting action happens in the section on the bottom left, which would have taken the majority of the LHS of the screen, until you opened that new script a minute ago. This region (or frame) is known as the *Console* and commands are executed here. To make sure this region is active either click in the region, or enter `Ctrl + 2`. Now the console will be where any typed text is entered, and this will all appear next to the > symbol.

(Hint: If you find yourself typing and you find unexpectedly that nothing is appearing in the Console, just hit Ctrl+2 and this frame will become active again.)

As a simple example of using R as a calculator, enter

```
1 + 1
```

then hit the <`enter`> key. Like a simple calculator, the answer will appear below your typed entries.

We can choose to simply print out the results of our calculations in this manner, and all the basic mathematical functions are here, such as `+, -, *, /`. Note that we don't need to use the equals sign, we just hit the <`enter`> key instead. We can also perform some common mathematical functions such as using exponents. As another simple example, find 3^2 by entering:

```
3^2
```

1.4.2 Inbuilt Functions

Natural logarithms can also be found using the function `log()`, with the common bases of 2 & 10 also having the in-built commands `log2()` and `log10()`. Interestingly, to perform these logarithmic calculations we will need to put the number inside those brackets '()' which immediately follow the command. For example, try entering:

```
log10(100)
```

and you will see the expected answer 2 (remember $10^2 = 100$). If you feel excited by this new piece of knowledge, see what numbers you get using natural logarithms and base 2.

```
log(100)
log2(100)
```

These are all a simple example of what we refer to in the R-Universe as a `function`. Essentially, someone at some point in time decided these were likely to be common computations and wrote some fancy code to perform all the calculations. This means that to find the logarithm of a number x, we can simply enter `log(`x`)` instead of having to figure out the code to perform the lengthy calculation hidden beneath this command. (See Equation 1.1 to give yourself a small heart attack, and remind yourself what maths is actually behind finding the logarithm of a number. Aren't you glad someone wrote this function so you don't have to worry about numerical integration?!)

$$\log(x) = \int_1^x \frac{1}{y} dy \tag{1.1}$$

(OK, now you've recovered from the shock of seeing an equation, back to the important stuff...)

Note that in the R functions given above, the brackets came immediately after the commands. This is the universal convention for R users, although in reality you could have placed one or more spaces there. Spaces are not as important in R as some languages and we generally only use them to separate numbers, words and symbols, or to make our code easier to read. We could have entered

```
log10                              (100)
```

if we'd chosen, but any R programmer would've thought you were strange, and any reviewer inspecting your manuscript would probably begin to doubt your skills quite seriously.

Another useful mathematical function we regularly use is finding the square root of numbers, which we do using the command `sqrt()`, e.g.

```
sqrt(49)
```

although the mathematically inclined amongst you would have also realised you can also enter

```
49^(0.5)
```

Personally, I find the first much easier to read and this is an important concept when using R. When we write R scripts to perform an analysis, it is highly likely that we will have to go back & re-read our code a year or two later when we have completely forgotten what we did. Making your code easy to read to someone with a minimal idea of what you were thinking at the time is VERY important! (Figure 1.2)

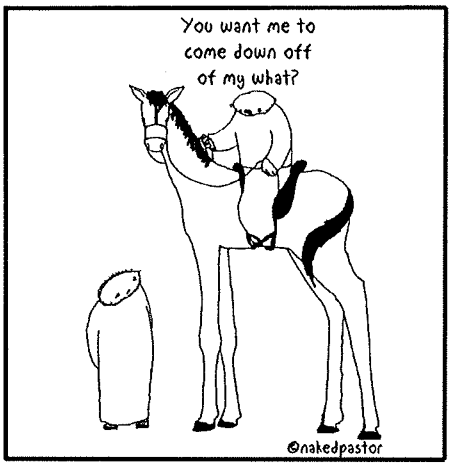


Figure 1.2: Steve gets excited about R

1.4.3 Saving Objects Into The Workspace

Although everything we've done in the above was essentially quite trivial, the important point was that we can simply print the output of a function to the screen, as we did when calculating logarithms and square roots. An alternative to this, especially if it's a result we'll need to refer to later, would be to save the results to an object within R.

R is commonly referred to as an `environment` and essentially it's a little world where can place objects of our own making. Every time we save some results, it's like writing them on a piece of paper and saving that piece of paper somewhere in our R workspace (or environment). We then save our complete workspace, which can contain hundreds or thousands of smaller objects which are specific to an analysis. This is very useful for keeping our data well organised.

Look at the set of tabs in the top right of the RStudio window & click on the one named `Environment`. If you've used RStudio before this may or may not be empty, but for today if it's not empty (and you don't have important objects in there) click the `Clear` button next to the small broom icon. This ensures we are starting with an empty workspace.

Let's say we are going to need the value from the calculation $5 * 5$ numerous times in our work. Instead of having to calculate this every time, we can simply save this as an object in our workspace. We'll need to give it a name, so now enter:

```
fiveByFive <- 5 * 5
```

Notice that the results didn't appear on the screen, but in the Environment Tab we now have an entry with the name we have just given our object. To see the actual value, we can either have a look in this window (for simple objects like this), or we could enter the command in the console.

```
fiveByFive
```

```
## [1] 25
```

1.4.4 The Assignment Operator

In the step above where we created the object `fiveByFive`, you may have noticed the use of <-, which is known as the assignment operator. Although you may not have seen this before it acts exactly like an arrow, which is what it is trying to represent. In the above it is sending the results from $5 * 5$ into the object named `fiveByFive`. This is the convention in R, and we avoid using the equals symbol (=) as this is used primarily for logical tests. Keeping these two symbols separate in this way, will also help keep your code clean for any other parties.

On the positive side, this operator can also be used at the end of a calculation.

```
5 * 5 -> fiveByFiveB
```

This is not historically very common in R, but with the advent of the package *dplyr* which we will explore today, this is becoming more common-place. This advantage of directionality is another important reason to use the assignment operator instead of '=' in our R code.

1.4.5 Accessing Help

Before we finally move on to look at the packages for today, it's helpful to know how to access the help pages within R. In the Console enter

```
?log
```

and the help page for the function `log` will appear. This page also lists a few related functions, which don't have to worry about too much for now. The important thing is to note that the function `log` contains two arguments inside the brackets. The first argument is `x`, and this is the number we are finding the logarithm of. The second argument is `base = exp(1)`. This is where we would set the base of the logarithm, and note that the argument has a sensible name so that we can easily understand it.

You can immediately see that instead of using the function `log10()` as we did above to find the logarithm of the value 100, we could have used this command in the form `log(100, 10)`. The order of placement for these values is very important, and note that we didn't need to specify them by name. For the sticklers amongst us, the most explicit use of the `log` function would be to enter

```
log(x = 100, base = 10)
```

As you look down the page, you will see a longer description of each of these arguments. In case you're not familiar with the term `vector`, in R this refers to a type of object which is just a bunch of numbers (or maybe even letters) stuck together in a one-dimensional object. A simple example would be to create the vector `x`, which contains the numbers 1 to 10.

```
x <- 1:10
x
```

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

A very brief word about vectors

We'll assume that you are comfortable with the idea of a vector, but if you are unsure what is meant, you can imagine that a vector is like a column of data in an Excel spreadsheet. However, all values in a vector will be the same type of value. Either they will be all numbers as in the *numeric vector* `x` we created above, or they could be all letters. There are two inbuilt vectors hiding in every R workspace which are made up of all letters, which we more formally refer to as *character vectors.*

```
letters
```

See if you can guess the name of the upper-case vector? These are two simple examples of character vectors, but in our work we will probably have numeric vectors when we measure something, and character vectors when we describe something, like a collection point or a cell type. Again, & for easy reference, these are just like columns in an Excel spreadsheet.

1.4.6 Logical tests

As well as performing calculations, R is also very well equipped to perform logical tests, which give a `TRUE/FALSE` response. Most of these are quite intuitive, such as:

```
2 > 1

## [1] TRUE

2 > 5

## [1] FALSE
```

However, there are a few expressions that are common, but that you may not have seen before. To check for equality between two values, we use a double equals sign (==), whilst to check for non-equality, we use an exclamation point in the place of the first equals sign (! =).

```
1 == 1

## [1] TRUE

1 != 1

## [1] FALSE
```

The exclamation point is also commonly used to reverse a `TRUE/FALSE` expression by placing it at the beginning of a test.

```
!1 == 1

## [1] FALSE
```

The `%in%` special operator

One additional, but extremely useful logical test is to use the special operator `%in%`, which asks if a given value is in a set of test values, for example

```
1 %in% 1:10

## [1] TRUE

11 %in% 1:10

## [1] FALSE
```

This can be thought of as asking the question *'is in'* of your data, and becomes very helpful when dealing with vectors.

```
1:5 %in% 1:10

## [1] TRUE TRUE TRUE TRUE TRUE

1:10 %in% 1:5

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

Notice how a TRUE/FALSE value was given for each number in the first vector, which corresponded to whether this value was in the second vector. We can also reverse this to test *not in* by placing an exclamation point at the beginning of the expression.

```
!1:10 %in% 1:5
```

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

There are wide variety of logical tests which can be applied in R which are beyond the scope of today, but a couple that you may find helpful in your own data are `is.numeric` and `is.na` which are less mathematical, but can help you detect common problems in your data. Some common issues arise when you have missing data, which may be read into R as the value NA, or if you read in a column from a spreadsheet which contains a mix of characters and numbers, in which case all of the numbers will be treated as characters by R.

```
is.numeric(1)

## [1] TRUE

is.numeric("1")

## [1] FALSE

is.na(1)

## [1] FALSE

is.na(NA)

## [1] TRUE
```

Table 1.1: Some important commands for logical tests in R

Command	Meaning
==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
%in%	is in the set of values

Chapter 2

The *dplyr* package

2.1 Packages in R

In the R way of doing things, packages are simply a collection of useful functions and methods which are bundled together for convenience by the package author. Even the functions we saw in the Introductory section are actually contained within a package, which in this case was called `base` and is loaded by default when R is opened. There are a handful of packages like this which are considered part of the general set-up which occurs when you install R, which are loaded by default when you start R. Everything else we use we have to load for ourselves, and this is usually the first part of any script or analysis we are performing.

2.1.1 Loading *dplyr*

Hopefully you have the package *dplyr* installed already, and this is a package which Excel users can easily adapt to as the layout of processes can be almost intuitive once you play with it for a while. To load the package, enter the command

```
library("dplyr")
```

Here we've called the function `library()` and given it the package name `"dplyr"`, which loads all of the functions and methods associated with the *dplyr* package, and makes them available to us in the Console. This package is designed to work with a type of R object called a `data.frame`, which is very similar to an Excel spreadsheet. Many R objects require all the values to be of the same type (e.g. all numeric values, or all characters), whilst data.frames can contain values of mixed type. Each column will be of a single data type, but the overall object will contain multiple data types. Most dataset we analyse are of this form, and in Excel we usually don't think about this but it's good to know this type of information in R.

2.1.2 Keeping track of things

Before we go any further, click on the *History* tab in the top-right frame in RStudio, where you will see everything you've entered in the Console so far in this session. Select the line where you loaded *dplyr* and click the *'To Source'* button. This will copy this command to the Script Window we opened at the start of the session. It's my personal preference (& a common practice) to load all required packages as the first few lines of any script I write, so this is a good habit to develop.

If there are key lines of any exploration you can copy from the history at any time during an analysis, although this frame will only show the most recent ~200 or so lines of code. Alternatively, you can enter every line of code in the Script Window, and send it to the Console by placing the cursor on a line, and hitting `Ctrl + Enter`, or by clicking on the *Run* icon in the top right of this frame.

This is the place to write & keep track of your code in your actual analyses too!

2.1.3 Two sample datasets

To start our explorations today we'll use two objects `mtcars` & `iris` which come pre-installed in R. These are hidden from view, but we can find them by entering their names.

```
mtcars
iris
```

Although this is not specifically biological data, they'll be helpful for use to grasp a few key concepts which are useful features of *dplyr*. When we entered the names above, the full data object was printed to the screen, but an alternative to this is to wrap the object name in the command `tbl_df()`, which will just give you a glimpse at the first few entries and columns. This is similar to the function `head()`, with the added bonus that it describes the object for us, and if we have a huge number of columns, only the first few will be printed.

```
tbl_df(mtcars)
tbl_df(iris)
```

As you can see, `mtcars` is an old dataset about the performance and specifications of some car models, whilst `iris` contains some measurements for 3 different species of iris. To find the interpretation of each of the columns, check the help page for the objects using `?mtcars` or `?iris`.

2.2 Beginning with *dplyr*

filter

The first simple thing we might like to do with the mtcars data is to have a look at the 8 cylinder cars in the data object. The command filter works exactly like you might expect, and the traditional way of working with R would be to call the function filter() by placing the data.frame first, then applying the filter with a logical test which gives a TRUE/FALSE response.

```
filter(mtcars, cyl == 8)
```

Clearly this just gives the entries in the mtcars object which have the value 8 in the column titled cyl.

The %>% special operator

Now we've covered the basics, one feature of *dplyr* which is unique but makes the package quite intuitive is the special operator %>%. This is uniquely defined as part of the package and can be simply interpreted as the word *then.* (Bash shell users might also see the parallels to the pipe command). This gives us the ability to work in a linear fashion with data.frame objects, instead of the inevitable nesting that would occur as we drill into our data, and also helps to avoid the creation of multiple copies of each object with a single transformation applied at each step.

To demonstrate this, an alternative way of writing the above would be to take the data.frame and using the command *'then'*, send it to the filter.

```
mtcars %>% filter(cyl==8)
```

This is the identical command to above but can be easier to read back in long strings of commands. We can also place then on separate lines for much easier comprehension. Once this operator has been placed at the end of a line, the R Console will wait for more input before completing any instructions.

arrange

For example, now that we've learned how to filter the dataset based on cylinders, we might want to arrange it in order of horsepower (i.e. the hp column). We can chain multiple commands together using this approach.

```
mtcars %>%
  filter(cyl==8) %>%
  arrange(hp)
```

This orders the results in ascending order, like the *Sort* function in Excel, and we can also place them in descending order by changing the final line to:

```
mtcars %>%
  filter(cyl==8) %>%
  arrange(desc(hp))
```

mutate

One noteworthy but less intuitive feature of the *dplyr* package, is that the rownames of the `data.frame` are not returned in the results. This can be disconcerting at first, but is easy to get around, by placing the values within the object itself. We can add rows to a `data.frame` in *dplyr* by using the command `mutate`.

```
mtcars %>%
  mutate(car = rownames(mtcars))
```

select

The above column addition has put the new column on the right of the object, but we could change the order of the columns by using the command `select` after we've added this.

```
mtcars %>%
  mutate(car = rownames(mtcars)) %>%
  select(car, mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb)
```

Now we've added data, it might be a good idea to save this as a new object

```
mtcars %>%
  mutate(car = rownames(mtcars)) %>%
  select(car, mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb) ->
  new_mtcars
```

We can also use `select` to only include specific columns The `iris` object is great for showing some of the flexibility of the `select` command. Instead of just naming everything we need, we could include or exclude columns which have a certain phrase amongst their names. To just include the columns with the phrase "Petal", we can use

```
iris %>%
  select(contains("Petal"))
```

To exclude these

```
iris %>%
  select(-contains("Petal"))
```

We can also match based on the start or end of the column name.

```
iris %>%
  select(ends_with("Width"))

iris %>%
  select(-starts_with("Sepal"))
```

Question The options for finding data are quite many & varied so here is one more simple example. What are we finding in the following set of commands? Could we have performed these filter operations in the same line?

```
new_mtcars %>%
  filter(cyl == 4) %>%
  filter(hp == max(hp))
```

distinct

One final operation which is useful before we move on to the next section, is `distinct` which chooses the first of each value in the specified column.

```
new_mtcars %>%
  distinct(cyl)
```

This can be a useful operation when you have repeated measurements for something which are of no interest for your current analysis. For example, you may have data for every transcript within a gene but may only require gene-level information, like the chromosomal band or something similar.

Question Using the command `distinct`, how could we find the car with the most horsepower for each number of cylinders, starting with 4, then 6, then 8 cylinders? Here's what you should expect to see once you figure it out.

```
##             car  mpg cyl  disp  hp drat    wt qsec vs am gear carb
## 1  Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## 2  Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## 3 Maserati Bora 15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
```

The operations we have covered above are found in Table 2.2

Table 2.1: Some handy operations in the package *dplyr*

Operations	Meaning
arrange	sorts the data based the specified column
distinct	chooses the first entry matching each different level in the specified column
filter	only show the entries which match the given criteria
mutate	add a column to the `data.frame`
select	select only the specified columns

2.3 Summarising across groups

The ability to summarise information across groups is also very useful in *dplyr*, and relies on the `group_by` command. This adds a hidden layer to the data, and gives the results in the same style as the `tbl_df` command we came across earlier. This first operation won't seem particularly exciting:

```
iris %>% group_by(Species)
```

However, this becomes extremely powerful when looking for summary values, such as the mean of a variable:

```
iris %>%
  group_by(Species) %>%
  summarise(Mean = mean(Petal.Length))
```

It's only a quick step from there to also incorporating the standard deviation & sample numbers.

```
iris %>%
  group_by(Species) %>%
  summarise(Mean = mean(Petal.Length),
            Sd = sd(Petal.Length),
            Count = n())
```

Clearly this is extremely useful when we have categorical data fields, and there are a handful of useful summary-type functions we can use. See if you can determine what each of the grouping commands in Table 2.2 do.

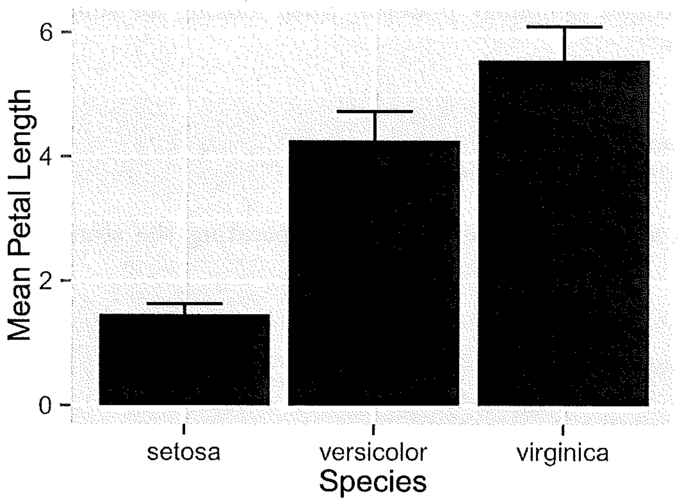
Table 2.2: Some group summary operations in the package *dplyr*

Operations	Meaning
min()	
max()	
mean()	Calculate the mean of a measurement for each group
sd()	Calculate the standard deviation of a measurement for each group
median()	
sum()	
n()	Count the number of measurements for each group
n_distinct()	

From here, it would be a simple step to plot these values using the package *ggplot2*. First we'll save the summary as an object

```
iris %>%
  group_by(Species) %>%
  summarise(Mean = mean(Petal.Length),
            Sd = sd(Petal.Length),
          Count = n()) ->
  iris_summary

library(ggplot2)
ggplot(iris_summary, aes(x = Species, y = Mean)) +
  geom_bar(stat = "identity") +
  geom_errorbar(aes(ymin = Mean - Sd, ymax = Mean + Sd),
                width = 0.4) +
  labs(y = "Mean Petal Length")
```



We're not really going to spend time explaining the package *ggplot2* today, so if you haven't seen the above methods for plotting, ask Steve for some explanations.

2.4 Some Biological Data

Let's apply this to some larger data which is at least connected with biological research. First find out what your current working directory is

```
getwd()
```

Now using your Internet browser, head to the NCBI ftp site `ftp://ftp.ncbi.nlm.nih.gov/` then click your way through to the *genomes* > *Bacteria* > *Lactobacillus_acidophilus_30SC_-uid63605* page and download the file *NC_015214.gff* to your current working directory.

Unfortunately this file has 5 header rows describing the important file information, so we'll need to skip reading those lines in. This is what the argument `skip = 5` means. As specified by `header = FALSE`, the file also has no column names, so R will automatically call them as V1 to V9.

```
gffFile <- file.path(getwd(), "NC_015214.gff")
file.exists(gffFile)
gffData <- read.delim(gffFile, skip = 5, header = FALSE)
tbl_df(gffData)
```

The columns that make up a *.gff* file are

1. seqname - name of the chromosome or scaffold;
2. source - the data source
3. feature - feature type name, e.g. Gene, Variation, Similarity
4. start - Start position of the feature, with sequence numbering starting at 1.
5. end - End position of the feature, with sequence numbering starting at 1.
6. score - A floating point value.
7. strand - defined as + (forward) or - (reverse).
8. frame - One of '0', '1' or '2'. '0' indicates that the first base of the feature is the first base of a codon, '1' that the second base is the first base of a codon, and so on.
9. attribute - A semicolon-separated list of tag-value pairs, providing additional information about each feature

Let's tidy this up into a form we can find a bit more useful. We can start by seeing what different entries the in the first column. As this is a bacterial genome, they should all be from the same circular DNA sequence and this column should contain only one value.

```
gffData %>%
  distinct(V1)
```

Question Is there only the one entry in our results? If not, how could we remove any problematic entries using *dplyr*?

Question How many sources is this data file collated from?

Let's now remove these rows, and while we're doing this, we can remove any irrelevant columns & rename the ones of interest. The entries in the `rename` command are just split over multiple lines to make it easier to read, so you don't have to type that part exactly.

```
gffData %>%
  filter(V1 != "###") %>%
  select(one_of(paste("V", c(3:5, 7, 9), sep=""))) %>%
  rename(feature = V3, start = V4, end  = V5,
         strand = V7, attribute = V9) ->
  gffData
tbl_df(gffData)
```

Note how easy this is to read back! It still takes a little bit of effort, but there's no real need to leave comments in our code. My early code was filled with lines like the following, but thanks to *dplyr* these are much less frequent in my code.

```
# Remove those weird lines with hashes in them
# Now only keep the interesting columns
```

Of course you can do that if you'd like, but the more experience you get, the more sense your code will actually make.

Question How do we find how many entries are in the file for each type of feature?

```
## Source: local data frame [6 x 2]
##
##   feature Count
## 1     CDS  2037
## 2    exon    75
## 3    gene  2112
## 4  region    50
## 5    rRNA    12
## 6    tRNA    63
```

Question How do we find how many genes are in the first region?

```
## Source: local data frame [1 x 2]
##
##   feature Count
## 1    gene  2112
```

Question What are the attributes for the longest CDS in this set of features? (*Hint: You may need to create an extra column to do this*)

```
## ID=cds1344
## Name=YP_004292478.1
## Parent=gene1408
## Dbxref=Genbank:YP_004292478.1,GeneID:10275716
## gbkey=CDS
## product=hypothetical protein
## protein_id=YP_004292478.1
## transl_table=11
```

NB: I had to be a bit fancy to make those line breaks. Your answer will appear in a single line of text.

Chapter 3

The *reshape2* package

3.1 Converting To Long Format

Now that we have a good feel for handling `data.frame` objects we can look at manipulating them even further. The objects we've seen so far can be considered as being in the *'wide'* format, where the measurements go across the page for a single sample. An alternative format is what can be considered as the *'long'* format, where the measurements for the various categories run down the page. The package *reshape2* is designed especially to manage this very common task. Load the package, then call up the help page for the function `melt.data.frame`.

```
library("reshape2")
## ?melt.data.frame
```

As we're no longer in the happy land of *dplyr* we'll have to move back to the regular syntax of R. The first 3 arguments to this function are where the action happens.

1. data - this is where we specify the `data.frame` we wish to transform

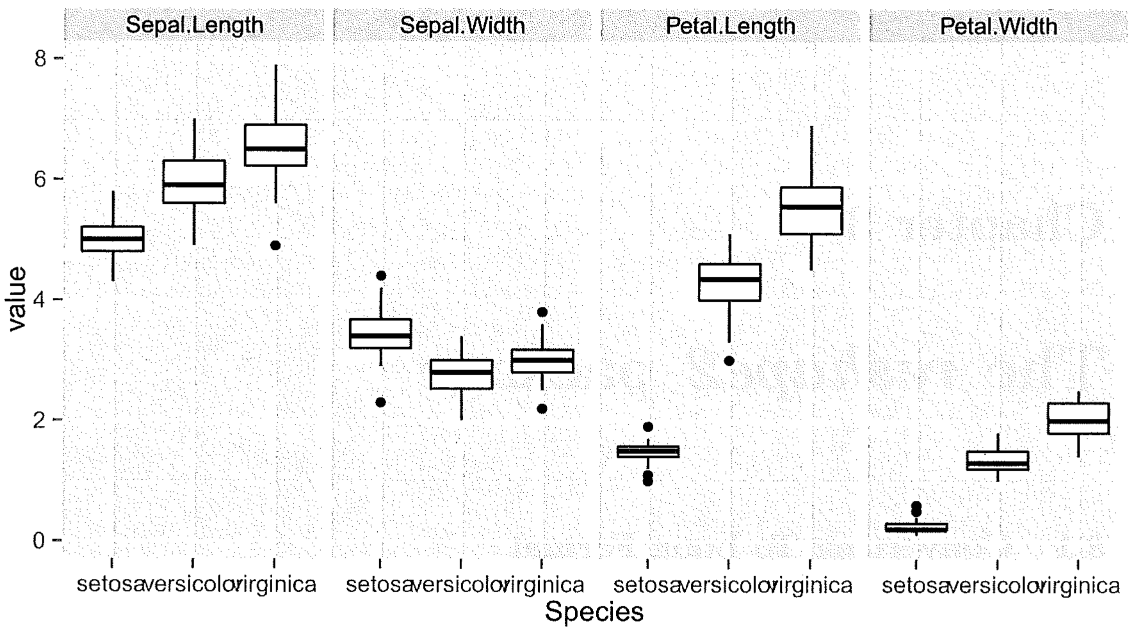
2. id.vars - here we specify the main identifying variable. In the first code fragment below, we'll use the Species variable from the `iris` object.

3. measure.vars - these are the measurements we wish to turn into a single column.

```
iris_melt <- tbl_df(melt(iris, "Species", 1:4, "measurement"))
iris_melt
```

Note that now we have a `data.frame` which is 600 rows long and 3 columns wide, instead of the original 150 x 5 object. This makes plotting the values very simple, and again, we can use the *ggplot2* package. The ability to easily break a plot into facets based on the different values of a categorical variable like the different types of measurements in our new variable called *measurement* is very useful for inspecting your data.

```
ggplot(iris_melt, aes(x=Species, y=value)) +
  geom_boxplot() +
  facet_grid(~measurement)
```

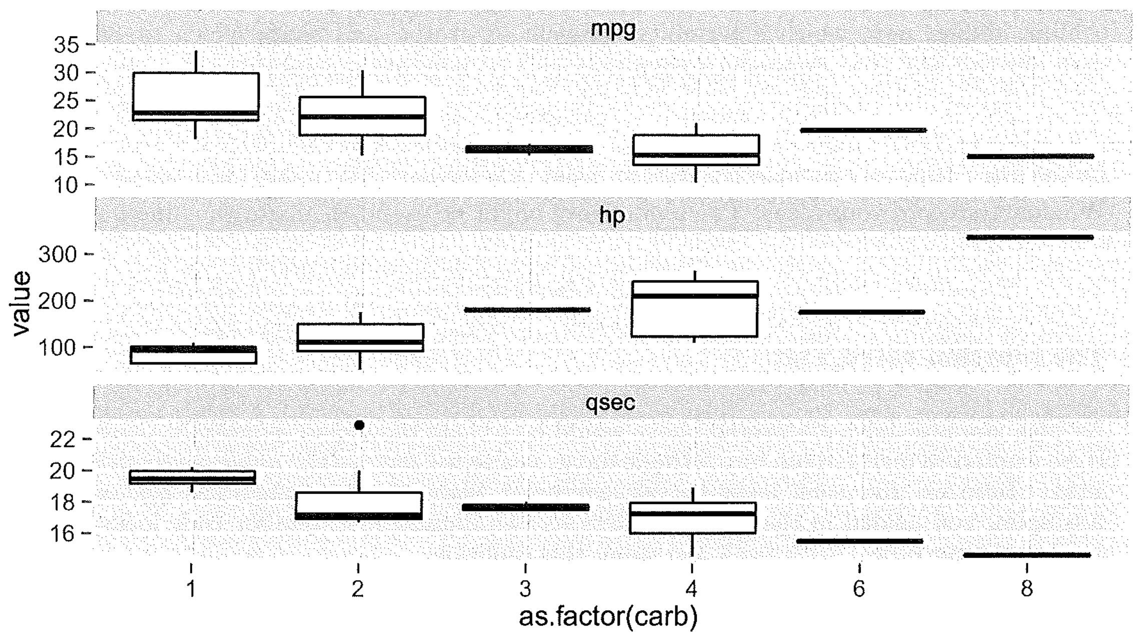


In this example we only had one identification variable, and four measurements for each sample. In the mtcars example, we essentially would have multiple variables which are determined by the production of the car (e.g. the number of cylinders) and we would be investigating any effect they might have on multiple performance-related measurements (e.g.. miles per gallon). Thus to explore these variables at the data inspection stage, we could first melt the `data.frame` using the production-related variables as the `id.vars` and the observed variables as the measurements.

```
mtcars_melt <- melt(new_mtcars,
                    c("car", "cyl", "drat", "wt", "am", "gear", "carb"),
                    c("mpg", "hp", "qsec"))
```

Now we can view the measured variables by any of the factory defined values. Let's try viewing them plotted against the number of carburettors. Note also that as this is a number, R needs to know that we're treating it as a categorical variable, which we do by the command `as.factor(carb)`.

```
ggplot(mtcars_melt, aes(x = as.factor(carb), y = value)) +
  geom_boxplot() +
  facet_wrap(~variable, ncol=1, scale="free_y")
```



3.2 Working With Long Format Data

Sometimes the data we are working with is already in the long format. In this case it can be easy to plot, but we may need to convert to a table to perform a χ^2 test or something similar.

Let's have a look at the genotype data which is 2 SNP loci that have been genotyped across 4 populations. The data can be downloaded from `http://tinyurl.com/krfwzjb`, then save it to your current working directory. Unfortunately, whoever created this file wasn't thinking of being nice to R users and the 2 loci are in separate locations in the spreadsheet. Open the file in Notepad, Excel or whatever you prefer to have a look at the file.

```
##    system..head.genotypes.csv...intern...TRUE.
## 1                                       Locus1
## 2                   Genotype,ID,State,Location
## 3                       AB,Sample1,SA,Coastal
## 4                       AB,Sample2,SA,Coastal
## 5                       AB,Sample3,SA,Coastal
## 6                       BB,Sample4,SA,Coastal
## 7                       AB,Sample5,SA,Coastal
## 8                       BB,Sample6,SA,Coastal
## 9                       AA,Sample7,SA,Coastal
## 10                      AB,Sample8,SA,Coastal
```

The first few rows are printed in the box above, and note how the Loci are saved separately in the file. Go to the help page for `?read.table` and have a look at the potentially overwhelming list of arguments that we can set. This is the function we'll need to use to load the data. Some useful arguments we might need to consider are `header`, which determines whether we have column names, `sep`, which determines the field separator, and `skip`, which we saw earlier with our *.gff* file. The argument `nrows` may also be helpful in answering the next question.

Question How can we load the data in from this format? Try a few ideas using the Script Window to record your ideas. The hints above might be useful, as might the function `mutate`.

The function `rbind_list`

An idea you may have discovered above would be to load the loci in separately as two objects. If we wanted to merge them into a single `data.frame` we can use the *dplyr* command `rbind_-list`. Here we give each locus to the command where the help page has the triple dots. Assuming you loaded in the genotype data as two data.frames, one for each locus, you can create a larger object with both loci using this function.

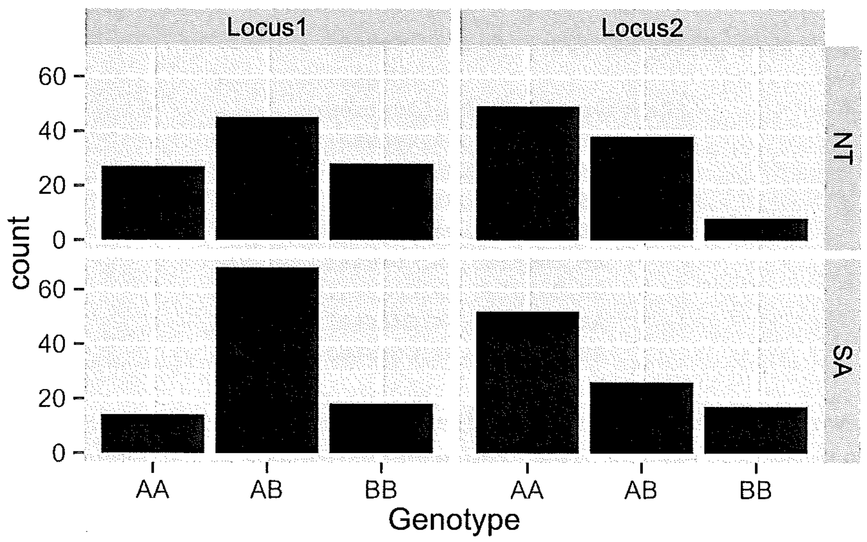
```
snpData <- rbind_list(loc1, loc2)
```

Now we can have a quick look using our familiar methods from *dplyr*, and also using the `str` function which is short for *structure*. This latter function is pretty much what is displayed in the *Environment* tab in the top right.

```
tbl_df(snpData)
str(snpData)
```

We could also plot this using `ggplot` and using both the Locus and the State as facets?

```
ggplot(snpData, aes(x = Genotype)) +
  geom_bar(stat="bin", position = "dodge") +
  facet_grid(State ~ Locus)
```



3.2.1 Summary Tables

Now we have the data loaded, we can get easily extract summary table using the command `dcast`. Check the help page first using `?dcast`.

If we wanted to summarise both Loci by genotype & the State they were collected in, we could use the following command. Note that this uses the R `formula` format, which is defining `Genotype` as the response variable, and the `Locus` and `State` as the predictor variables.

```
dcast(snpData, Genotype ~ Locus + State)
```

Question How could we also use `dplyr` to just summarise the genotypes for Locus1, using both State and Location as the predictor variables? Save this as an object named `loc1_table`. The results should look like

```
loc1_table

##   Genotype NT_Coastal NT_Inland SA_Coastal SA_Inland
## 1       AA         10        17          5         9
## 2       AB         21        24         34        34
## 3       BB         14        14          6        12
```

We could now perform a χ^2 test on the data.

```
chisq.test(loc1_table %>% select(-Genotype))
```

Advanced Question Could you write a function to apply this to both loci and output the locus alongside the *p*-value for the test?