Babraham Bioinformatics

Advanced R

Version 2018-10

Licence

Contents

Introduction

This course follows on from our introductory R course. In the introductory course we introduced the R and RStudio environments and showed how these can be used to load in datasets, manipulate and filter them and produce simple graphs from the resulting data.

In this course we extend these initial skills by looking at some more complex, but still core, R operations. We also look at how to combine the pieces of functionality you already know to solve more complex problems by constructing more complex compound statements. We start by reviewing the core data structures and looks at how to filter, merge and reshape data structures. We then introduce some more functions which can be used alongside selections to achieve some new functionality. We then look at how to efficiently loop over data structures to run the same analyses multiple times. Finally we will look at extending the core R functionality with modules. Throughout the course we will look at better practices for structuring and recording the work that you’re doing.

At the end of this course you should be able to approach more complex analyses in large data structures and be able to document the work you’ve done.

Manipulating Data

Recap of Filtering

Since the vast majority of operations you'll ever perform in R are really clever ways of mixing together simple selection and filtering rules on the core R data structures it's probably worth recapping the different ways in which we can make selections.

We'll start off looking at a simple vector, since once you can work with these then matrices, lists and data frames are all variations on the same ideas.

```
> a.vector
bob sam sue eve don jon
  4   8   2   3   8   4
```

Access by index

The simplest way to access the elements in a vector is via their indices. Specifically you provide a vector of indices to say which elements from the vector you want to retrieve. Remember that index positions in R start at 1 rather than 0.

```
> a.vector[c(2,3,5)]
sam sue don
  8   2   8

> a.vector[1:4]
bob sam sue eve
  4   8   2   3
```

We can then use this with other functions which generate a list of index positions. The most obvious example would be the order() function which can be used to provide an ordered set of index positions, which can in turn be used to order the original vector. You can do this directly with sort() but order() allows for more complicated uses in the context of lists and data frames.

```
> order(a.vector)
[1] 3 4 1 6 2 5

> a.vector[order(a.vector)]
sue eve bob jon sam don
  2   3   4   4   8   8
```

Access by name

Rather than using index positions, where you have an associated set of names for vector elements you can use these to retrieve the corresponding values.

```
> a.vector[c("jon","don")]
jon don
  4   8
```

You can then use this with any method of generating lists of names which match the names in your vector. A simple example would be ordering the vector by the associated names:

```
> order(names(a.vector))
[1] 1 5 4 6 2 3

> names(a.vector)[order(names(a.vector))]
[1] "bob" "don" "eve" "jon" "sam" "sue"

> a.vector[names(a.vector)[order(names(a.vector))]]
bob don eve jon sam sue
  4   8   3   4   8   2
```

Again, there are more efficient ways to perform this specific task, but it's the principle of the method of access, and stringing together ways of generating and using lists which we need to be sure of.

Access by Logical vector

If you generate a logical (also called a boolean) vector the same size as your actual vector you can use the positions of the true values to pull out certain positions from the full set. You can also use smaller logical vectors and they will be concatenated to match all of the positions in the vector, but this is less common.

```
> a.vector[c(TRUE,TRUE,FALSE,FALSE,FALSE,TRUE)]
bob sam jon
  4   8   4
```

This can be used in conjunction with logical tests which generate this type of list.

```
> a.vector < 5
  bob   sam   sue   eve   don   jon
 TRUE FALSE  TRUE  TRUE FALSE  TRUE

> a.vector[a.vector < 5]
bob sue eve jon
  4   2   3   4
```

Extensions to Lists and Data Frames

All of the methods outlines above transfer to lists and data frames. For lists the extra level required is that the list item can be addressed by any of these methods, but only to return one column. For data frames the principles above apply, but you can use two v to specify which rows and columns you want to return from the full data set.

There are a couple of extra pieces of syntax used for lists and data frames.
For lists you can access the individual list items using double brackets. Inside these you can put either an index number or a column name.

```
> a.list
$names
[1] "bob" "sue"

$heights
 [1] 100 101 102 103 104 105 106 107 108 109 110
```

```
$years
[1] 2013 2013 2013 2013 2013

> a.list[[2]]
 [1] 100 101 102 103 104 105 106 107 108 109 110

> a.list[["years"]]
[1] 2013 2013 2013 2013 2013
```

When accessing a single list or data frame item using square brackets you should always use double brackets. You can use single brackets and a vector of indices to access multiple list items, but this won't work for data frames (since R can't tell whether you're accessing columns or rows).

The other piece of useful syntax needed for lists and data frames is the `$` notation for extracting columns or list items by name. This should be your preferred way of accessing individual vectors in lists or data frames since the use of names makes it much more obvious what you're doing, and the `$` notation is much cleaner and more compact than square brackets.

```
> a.list$names
[1] "bob" "sue"

> a.list$heights
 [1] 100 101 102 103 104 105 106 107 108 109 110

> a.list$years
[1] 2013 2013 2013 2013 2013
```

It's worth noting that for this notation to work there are limits placed on the names which can be used for columns or list slots. When you create a data frame the names in the columns are automatically checked and altered to become syntactically valid. If you want to preserve the original names (and lose the ability to use `$` notation) then you can specify `check.names=FALSE`.

```
> data.frame("Invalid name"=1:10,valid.name=11:20) -> valid.frame
> colnames(valid.frame)
[1] "Invalid.name" "valid.name"


> valid.frame$Invalid.name
 [1]  1  2  3  4  5  6  7  8  9 10

> data.frame("Invalid name"=1:10,valid.name=11:20,check.names=FALSE) ->
invalid.frame
> colnames(invalid.frame)
[1] "Invalid name" "valid.name"

> invalid.frame$Invalid name
Error: unexpected symbol in "invalid.frame$Invalid name"
```

Building an R toolbox

By combining the use of basic selection tools for vectors and data frames with a number of useful functions you can start to build up very complex functionality. Most seemingly complex operations in R are actually the result of combining together much more simple operations.

In this section we are going to recap some existing functions which can be used with selections, and introducing some new ones which can also prove useful. We can then start to think about how we can combine these to achieve more complex operations.

In general, for most operations we're going to cover here there are two ways they can be performed.

1. There will be a function which performs the operation on the data directly, producing the modified data in a single step.

2. There will be an alternate method which can perform the operation in a way which doesn't directly alter the data, but can produce a data structure (index or logical vector) which can be used with a selection to achieve the data modification. Whilst these methods are more cumbersome for simple operations, they are often useful because they provide the flexibility to apply the result to another data structure, or can be combined with other operations to achieve a more complex end result.

Complex conditional statements

One of the easiest ways to build more complex operations is to start combining the logical vectors created by individual logical tests.

Logical vectors can be combined with the logical operators `&` (and) and `|` (or) to create more complex filters. The `!` (not) operator inverts the truth of any logical vector. This allows you to create multi-part conditional statements to answer questions such as "which values are higher than 10 but lower than 20".

```
> a.vector > 5
  bob   sam   sue   eve   don   jon
FALSE  TRUE FALSE FALSE  TRUE FALSE

> names(a.vector) == "sue"
[1] FALSE FALSE  TRUE FALSE FALSE FALSE

> a.vector > 5 | names(a.vector) == "sue"
  bob   sam   sue   eve   don   jon
FALSE  TRUE  TRUE FALSE  TRUE FALSE

> a.vector[a.vector > 5 | names(a.vector) == "sue"]
sam sue don
  8   2   8
```

Duplication

Detecting repeated values in a vector is a simple operation which can be useful on its own. In combination with selection it can also be used to deduplicate more complex data structures. There are a couple of useful functions which let you deal with duplication.

The `duplicated()` function takes a vector of values and returns a logical vector to say if each item in the vector is a duplicate of one which has already been seen earlier in the vector.

```
> data
[1] 1 4 6 2 5 2 5 4 2
> duplicated(data)
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

Since the output is a logical vector it can be used as a selector against the original data to get the duplicated values.

```
> data[duplicated(data)]
[1] 2 5 4 2
```

You can also invert the result using a logical not operation to get the non-duplicated values.

```
> data[!duplicated(data)]
[1] 1 4 6 2 5
```

Because this selection is based on repeated values it will still leave one copy of each value in the selected vector. If you wanted to do a more complete removal then you can use the `fromLast` option to the function which does the search starting at the back.

```
> duplicated(data)
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE

> duplicated(data, fromLast=TRUE)
[1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE

> duplicated(data) | duplicated(data,fromLast=TRUE)
[1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Using this type of function you can then perform operations such as deduplicating data frames. You simply extract the column you want to deduplicate on, generate the logical vector for the non-duplicated entries and then use that to specify which rows to keep in the data frame.

```
> example.data
   Name Chr Start End Strand Value
1  ABC1   1   100 200      1  5.32
2  MEV1   1   300 400     -1  7.34
3  ARF2   1   300 400     -1  3.45
4  GED6   2   300 400      1  6.36
5  RPL1   2   500 600      1  2.12
6  NES2   2   700 800     -1  8.48
7  AKT1   3   150 300      1  3.25
8  GED6   3   400 600      1  3.65
9  XYN1   4   125 350     -1  7.23
10 RBP1   4   400 550     -1  4.54

> example.data[!duplicated(example.data$Name),] -> example.data.dedup

> example.data.dedup
   Name Chr Start End Strand Value
1  ABC1   1   100 200      1  5.32
2  MEV1   1   300 400     -1  7.34
3  ARF2   1   300 400     -1  3.45
4  GED6   2   300 400      1  6.36
5  RPL1   2   500 600      1  2.12
6  NES2   2   700 800     -1  8.48
7  AKT1   3   150 300      1  3.25
```

```
9   XYN1    4    125  350       -1   7.23
10  RBP1    4    400  550       -1   4.54
```

In some cases a simpler function might be useful for dealing with duplication. The `unique` function gives you back a deduplicated version of a vector. In this case the deduplicated values are returned directly.

```
> data
[1] 1 4 6 2 5 2 5 4 2

> unique(data)
[1] 1 4 6 2 5
```

Because you are getting back values and not a logical vector you have less flexibility for what you can do with the result of `unique` but it can be much quicker for some operations. In general if you are deduplicating a vector then the `unique` function is OK, but if you want to deduplicate something more complex, like a data framt when you'd need to use `duplicated` in combination with a selection.

Ordering and Sorting

In many cases you will want to sort datasets in R. There are two functions which can be useful to do this.

The sort function returns a sorted version of a vector. This will be the quickest way to sort a simple vector.

```
> data
[1] 1 4 6 2 5 2 5 4 2
> sort(data)
[1] 1 2 2 2 4 4 5 5 6
```

The order function takes in a vector and returns a vector of index values which would then order the values in the original vector.

```
> data
[1] 1 4 6 2 5 2 5 4 2
> order(data)
[1] 1 4 6 9 2 8 5 7 3
> data[order(data)]
[1] 1 2 2 2 4 4 5 5 6
```

Because it provides a vector of indices, the order function can therefore be used to sort more complex data structures.

```
> example.data[order(example.data$Value),]
   Name Chr Start End Strand Value
5   RPL1    2    500  600        1   2.12
7   AKT1    3    150  300        1   3.25
3   ARF2    1    300  400       -1   3.45
8   GED6    3    400  600        1   3.65
10  RBP1    4    400  550       -1   4.54
1   ABC1    1    100  200        1   5.32
4   GED6    2    300  400        1   6.36
9   XYN1    4    125  350       -1   7.23
2   MEV1    1    300  400       -1   7.34
6   NES2    2    700  800       -1   8.48
```

Set operations

Being able to combine data in two vectors can be useful (which genes in list A are also in list B – for example). R has a number of functions and operators which relate to combining vectors.

The `%in%` operator is the built in part of the language which deals with combining vectors. It is this operator which underlies the code behind many of the other set based functions.

The `%in%` operator comes between two vectors. It returns a logical vector of the same length as the first vector where TRUE or FALSE depends on whether the value at each position was present anywhere in the second vector.

```
> days <- c("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
> to.test <- c("Jan","Feb","Thur","Tue","Oct","Wed")

> to.test %in% days
[1] FALSE FALSE FALSE  TRUE FALSE  TRUE
```

There are also a number of functions relating to set based operations.

The `intersect` function takes in two vectors and returns a vector of the values which were present in both of them.

```
> intersect(days,to.test)
[1] "Tue" "Wed"
```

The `setdiff` function takes in two vectors and returns the subset of the first vector whose values do not appear in the second.

```
> setdiff(days,to.test)
[1] "Mon" "Thu" "Fri" "Sat" "Sun"
```

Other tests

There are a few other types of basic test, which produce logical vectors in the same way as > == < etc. which can be useful. Examples of these are:

Function	Meaning	Note
`is.na()`	Are the values passed NA values	
`is.infinite()`	Are the values passed infinite	Positive or negative inf values are all caught.
`is.finite()`	Are the values passed finite	
`is.numeric()`	Are the values passed of a numeric type	Any numeric type is OK (integers, doubles etc.)
`is.integer()`	Are the values passed integers	Be careful, floating numbers which are also integers (eg `2.00`) will fail this.

Conditional statements

As well as using the tests we've seen above as selectors for datasets we can also use them to selectively run some parts of our code. A simple conditional statement in R looks like this:

```
score <- 10
my.name <- "Simon"

if (my.name == "Simon") {
  score <- score*10
}

score
```

In the if statement we need a piece of code which produces a single logical value as a result. In this instance that's OK since we're only testing one value, but if we had a vector of values to test then we'd have problems.

```
my.values <- c(1,2,NA,4,5,NA)

if (is.na(my.values)) {
  print("I had some NAs")
}
```

What we're actually (unintentionally) doing here is to test the first value in `my.values` to see if it's `TRUE`. R will complain that this doesn't look like a sensible thing to be doing.

```
Warning message:
In if (is.na(my.values)) { :
  the condition has length > 1 and only the first element will be used
```

The extra bit of logic we need is a way to collapse a logical vector down to a single logical value depending on whether `any` or `all` of the values in the vector are `TRUE`.

```
if (any(is.na(my.values))) {
  print("I had some NAs")
}

if (all(some.numbers == 0)) {
  print("Everything is zero")
}
```

Text Manipulation

Combining text vectors together

In a previous example we used an existing column in a data frame to deduplicate our data, but sometimes the duplication you want to remove is not encoded in a single variable, but would require the combined similarity of several columns. In the example dataset we might want to deduplicate based on a match between Chr, Start, End and Strand.

The easiest way to do this is to build a new compound variable by using string concatenation to join these different columns into a single string which can then be deduplicated.

Text manipulation is not the greatest of R's strengths, but it has some functions which can be useful.
If you need to join two or more strings together then the function to use is `paste()`. You can pass this a set of values which will be concatenated together to return a single string.

```
> paste("Hello","there","number",1)
[1] "Hello there number 1"
```

By default, the different elements are joined with spaces, but you can change this using the `sep` parameter.

```
> paste("Hello","there","number",1,sep=":")
[1] "Hello:there:number:1"
```

Searching and Replacing

Another common operation would be searching for patterns and either selecting the values which match, or replacing the matches with something else.

There are a couple of functions you can use to do flexible text searching.

The `grep` function takes a search pattern and a character vector and by default returns a vector of the indices in the vector which match the pattern. If you add the `value=TRUE` option to grep then it will return the matched values directly. The pattern can be a literal piece of text, but you can also construct more complex degenerate patterns with flexibility in them. You can read up about "regular expressions" to see what sort of flexible patterns you can construct.

```
> sample.names
[1] "WT1" "WT2" "WT3" "KO1" "KO2" "KO3"

> grep("KO",sample.names)
[1] 4 5 6

> grep("KO",sample.names, value=TRUE)
[1] "KO1" "KO2" "KO3"
```

The `grepl` function is very similar except that instead of returning a vector of indices, it returns a logical vector which says which positions matched for not. For simple selections then `grep` along with the value=TRUE option is the easiest solution, but `grepl` can be more flexible since logical vectors allow you to use logical operations such as negation to select values which didn't match for example.

```
> sample.names
[1] "WT1" "WT2" "WT3" "KO1" "KO2" "KO3"

> ! grepl("KO",sample.names)
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE

> sample.names[!grepl("KO",sample.names)]
[1] "WT1" "WT2" "WT3"
```

Rather than just matching a pattern, sometimes you want to do a find and replace. This is often needed to match up slightly different versions of the same strings by removing a prefix or suffix. The function to do this is called `gsub` and requires the pattern to remove, the string to replace it with (which can be blank) and the data in which to do the replacement.

```
> c("A.txt","B.txt","C.txt") -> file.names
> gsub(".txt","",file.names)
[1] "A" "B" "C"
```

The search operations we've seen so far only match a single pattern but can find multiple hits. A simpler problem is finding the index position of a set of values in a larger vector. The matching here is simpler (it requires an exact match), but the scale is larger. This type of problem can be solved with the `match` function, which takes a vector of search strings (not patterns) and searches through a second vector and reports the index positions where each of the search strings is first found. If the string isn't found then `NA` is returned.

```
> sample.names
[1] "WT1" "WT2" "WT3" "KO1" "KO2" "KO3"
> match(c("WT2","KO2","NO2"),sample.names)
[1]  2  5 NA
```

Other String Manipulation Functions

Other common functions used for string manipulation are:

- `substr(data,start index, end index)` - pulls out a substring from a longer string
- `strsplit(data,split character)` - splits an initial delimited string into a vector of strings
- `tolower(data)` - converts a string to lower case - useful for case insensitive searches
- `toupper(data)` - converts a string to upper case

It's worth noting that all matching operations in R are case sensitive so "Hello" is not the same as "hello". If you want to perform case-insensitive matches then using `tolower` or `toupper` during the match will allow you to do this.

```
> sample.names
[1] "Wt1" "WT2" "wt3" "Ko1" "KO2" "ko3"
> sample.names[grepl("WT",sample.names)]
[1] "WT2"
> sample.names[grepl("WT",toupper(sample.names))]
[1] "Wt1" "WT2" "wt3"
```

Extending and Merging Data Sets

Often your data will not come as a single fully processed dataset but may require you to join together two or more other datasets to generate the final dataset you will use for your analysis. There are a few different ways to merge data depending on how the individual datasets you have are structured.

Adding new rows to an existing dataset

The simplest kind of merge is where you have two datasets with identical structure but with each containing different subsets of your final dataset. In this instance you want to merge them by concatenating all of the rows together to form a single dataset.

We can do this by using the `rbind` function. This requires that there are the same number of columns in the two datasets. It will try to coerce the data into the same data types as it was in in the original data frames, but will convert data types to be compatible if necessary.

```
> data.set.1
  day value
1 Mon     2
2 Tue     6
3 Wed     3

> data.set.2
  day value
1 Thu     1
2 Fri     8
3 Sat     5

> rbind(data.set.1,data.set.2) -> data.set.1.plus.2

> data.set.1.plus.2
  day value
1 Mon     2
2 Tue     6
3 Wed     3
4 Thu     1
5 Fri     8
6 Sat     5
```

Adding new columns to an existing dataset

There are some simple ways to add new columns to an existing dataset. If your new data is already the same length and in the same order as your existing data then you can either add a single new column manually by assigning to a column name which doesn't exist.

```
> start.data
   cola colb
1     1   11
2     2   12
3     3   13
4     4   14
5     5   15
6     6   16
```

```
7      7   17
8      8   18
9      9   19
10    10   20

> start.data$colc<-21:30
> start.data
   cola colb colc
1     1   11   21
2     2   12   22
3     3   13   23
4     4   14   24
5     5   15   25
6     6   16   26
7     7   17   27
8     8   18   28
9     9   19   29
10   10   20   30
```

If you have multiple columns to add then you can use `cbind` to do this in a single operation, but the rows need to match exactly between the two data frames you're merging. You can `cbind` either two existing data frames, or one data frame with a bunch of new column definitions.

```
> cbind(start.data,cold=31:40,cole=41:50) -> start.data
> start.data
   cola colb colc cold cole
1     1   11   21   31   41
2     2   12   22   32   42
3     3   13   23   33   43
4     4   14   24   34   44
5     5   15   25   35   45
6     6   16   26   36   46
7     7   17   27   37   47
8     8   18   28   38   48
9     9   19   29   39   49
10   10   20   30   40   50
```

Merging columns can be problematic though because you may have a more difficult time describing which rows from the different datasets to match up, and because there may be rows in one dataset which don't have an equivalent in the other.

We could try merging using a combination of `%in%, intersect`, `match` and similar operations, but this will quickly get quite complicated. A more generic solution is to use the built-in `merge` function to bring together related data frames.

The `merge` function takes in a set of data frames to be merged along with a set of column names, all of which need to match between the different datasets to provide a match between the rows in the different data sets. By default only rows which are present in all of the supplied data frames will appear in the merged set, but you can use `all=TRUE` to keep all of the original data and pad any missing values with `NA`s.

In the example below the columns to use for merging have been explicitly set, but we could have omitted these as the function will merge by default on columns with the same names between the two sets.

```
> merge(frame.1,frame.2,by.x="letter",by.y="letter")
  letter value.1 value.2
1      B       2       1
2      D       4       5
3      F       5       4
```

Although you can match rows based on multiple columns, it rarely makes sense to do this. If you need this then it might end up being clearer if you manually make up a compound variable using `paste` and then merge on this.

Simple Transformations

One option which you often need within R is the ability to transform a matrix or data frame so that the rows and columns are swapped. There is a built in function to do this called `t()` (for transpose). This will return a transposed copy of the original data. Since only columns are guaranteed to be all of the same type you can get odd side effects from this transformation if you mix columns of different data types. You should also note that even if you submit a data frame to `t()` you will receive a matrix back and will need to use `as.data.frame()` to convert this back to a data frame.

```
> data.frame(a=c(10,20,30,40),b=c(100,200,300,400),c=c(1000,2000,3000,4000)) ->
original.data
> original.data
   a   b    c
1 10 100 1000
2 20 200 2000
3 30 300 3000
4 40 400 4000

> t(original.data)
  [,1] [,2] [,3] [,4]
a   10   20   30   40
b  100  200  300  400
c 1000 2000 3000 4000

> class(original.data)
[1] "data.frame"

> class(t(original.data))
[1] "matrix"
```

Looping

One of the most unusual features of R is the way that it handles looping constructs to perform an operation over several sets of data. R does have the sorts of loops you might find in other languages (`for` loops, `while` loops etc.) but the use of these is generally discouraged since they tend to be very inefficient compared to more R specific methods designed to iterate through data structures.
The main function used to iterate through data structures in R is `apply`. The basic `apply` function is very powerful and can be configured to perform all kinds of loops. As with many other R functions through there are a series of derivate versions of `apply` which can be used to more quickly generate certain types of loop.

Looping through data frames

Probably the most common type of loop you will perform in R is a loop which performs an operation on each row or each column of a data frame. For this type of loop you generally use the basic `apply` function. The syntax for `apply` is quite minimal and it makes it very easy to set up complex analyses with a very small amount of code.

The basic syntax when looping through a data frame with `apply` is:

```
apply (data.frame, rows(1)/cols(2), function name, function arguments)
```

A simple example is shown below. This generates the set of row and column means from a large dataset. The same effect could have been achieved in this case using the `colMeans` and `rowMeans` functions, but knowing how to do this with `apply` lets you do the same type of analysis for any built in function, or even build your own, as we will see later.

```
> numeric.data
       Col1  Col2  Col3  Col4  Col5  Col6  Col7  Col8  Col9 Col10
Row1   1.17 -0.06 -1.03  0.88  1.69 -1.64 -0.45  2.07  1.60  0.01
Row2   0.69 -1.18  1.08  0.45  2.24 -0.60  0.41 -1.02  1.11 -0.47
Row3  -0.65  1.52  0.98 -1.36 -0.12 -0.42  0.02 -1.66 -1.42 -0.85
Row4   0.11  1.60  0.85  1.15  0.16  0.89  1.06 -0.91 -0.06 -0.76
Row5  -2.49  1.36 -0.35  1.33 -1.43 -0.60  0.36 -1.97 -1.39  0.32
Row6  -0.19 -1.91 -0.03  0.16  0.48 -1.31  0.27 -0.67 -0.25  0.98
Row7  -1.51 -0.23  0.93  0.73 -0.93  0.06  1.17  0.65 -1.11  2.04
Row8  -0.35 -0.01  1.10  1.66 -0.24  1.13 -0.15  0.69  0.18  0.32
Row9  -0.96 -0.27 -1.58  0.73  1.22 -0.36  0.42 -0.48  0.19  0.38
Row10  0.58  0.13  0.52 -2.36  0.49 -1.87  0.66 -0.32  0.14 -0.29

> apply(numeric.data,1,mean)
  Row1   Row2   Row3   Row4   Row5   Row6   Row7   Row8   Row9  Row10
 0.424  0.271 -0.396  0.409 -0.486 -0.247  0.180  0.433 -0.071 -0.232

> apply(numeric.data,2,mean)
  Col1   Col2   Col3   Col4   Col5   Col6   Col7   Col8   Col9  Col10
-0.360  0.095  0.247  0.337  0.356 -0.472  0.377 -0.362 -0.101  0.168
```

Data types when using apply

When running apply one rule which is enforced by R is that the vectors produced during the operation will always have the same type (numerical, character, logical etc). This is most obviously the case when iterating over rows of data with mixed column types, where the vector passed to the function must be coerced into a common type.

When there are mixed types the program will covert these to the lowest common denominator, usually a character vector. This can mean that you can get odd effects when what you assume to be a number is actually provided as a character.

```
> example.data
   Name Chr Start End Strand Value
1  ABC1   1   100 200      1  5.32
2  MEV1   1   300 400     -1  7.34
3  ARF2   1   300 400     -1  3.45
4  GED6   2   300 400      1  6.36
5  RPL1   2   500 600      1  2.12
6  NES2   2   700 800     -1  8.48
7  AKT1   3   150 300      1  3.25
8  GED6   3   400 600      1  3.65
9  XYN1   4   125 350     -1  7.23
10 RBP1   4   400 550     -1  4.54

> apply(example.data,1,class)
 [1] "character" "character" "character" "character" "character" "character"
"character" "character" "character" "character"
```

The same is also true when using apply over columns, even if some columns are purely numerical they can still be passed to the function as characters.

```
> apply(example.data,2,class)
       Name         Chr       Start         End      Strand       Value
"character" "character" "character" "character" "character" "character"
```

There are a couple of potential solutions to this problem.

1. If you only need to process the numeric rows from a data frame which contains both numeric and character rows then subset the data frame before running apply to extract only the numeric rows so the type conversion will not happen.
2. If you need to use both character and numeric elements within the apply function then make sure you use `as.numeric` on the individual elements before using them in calculations.

```
> apply(example.data[,3:6],1,class)
 [1] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
"numeric" "numeric" "numeric"
```

Collating data

One variant of the basic apply function can be used to subset your data based on the categories defined in one or more columns, and to provide summarised data for the different category groups identified. This variant is called `tapply` and the basic usage is shown below:

```
tapply(vector of values, vector of categories, function to apply)
```

What `tapply` will do is to break the vector of values up according to the categories defined in the category vector. Each of the sets of values will then be passed to the function.
In our example data we could use this method to calculate the mean value for each chromosome using the code below.

```
> example.data
   Name Chr Start End Strand Value
1  ABC1   1   100 200      1  5.32
2  MEV1   1   300 400     -1  7.34
3  ARF2   1   300 400     -1  3.45
4  GED6   2   300 400      1  6.36
5  RPL1   2   500 600      1  2.12
6  NES2   2   700 800     -1  8.48
7  AKT1   3   150 300      1  3.25
8  GED6   3   400 600      1  3.65
9  XYN1   4   125 350     -1  7.23
10 RBP1   4   400 550     -1  4.54

> tapply(example.data$Value,example.data$Chr,mean)
       1        2        3        4 
5.370000 5.653333 3.450000 5.885000 
```

You can also use `tapply` in conjunction with more than one category vector. You can supply a list of category vectors (selecting columns from a data frame also works since data frames are also lists) and the grouping will take place over the combination of factors.

In the code below we count how many genes of each name are present on each chromosome.

```
> tapply(example.data$Value,example.data[,c("Name","Chr")],length)
      Chr
Name    1  2  3  4
  ABC1  1 NA NA NA
  AKT1 NA NA  1 NA
  ARF2  1 NA NA NA
  GED6 NA  1  1 NA
  MEV1  1 NA NA NA
  NES2 NA  1 NA NA
  RBP1 NA NA NA  1
  RPL1 NA  1 NA NA
  XYN1 NA NA NA  1
```

We can also combine `tapply` with apply to allow us to calculate clustered values from several different columns of a data frame. If we wanted to calculate per-chromosome means for all of the numeric columns then we could do this by wrapping our `tapply` code in a standard apply to loop over the columns.

```
> apply(example.data[,3:6],2,tapply,example.data$Chr,mean)
     Start      End     Strand    Value
1 233.3333 333.3333 -0.3333333 5.370000
2 500.0000 600.0000  0.3333333 5.653333
3 275.0000 450.0000  1.0000000 3.450000
4 262.5000 450.0000 -1.0000000 5.885000
```

Looping through lists and vectors

Looping through rows or columns of a data frame are probably the most common application of apply statements, but you can also use these on the simpler data structures of lists or even vectors. To do this there are two variants of apply which can be used, `lapply` and `sapply`. In essence these are the same basic idea – a way to run a function over all of the elements of a list or vector. The only difference between the two is the nature of the result they return. An `lapply` statement returns a list of values so if you run something simple, like getting the absolute value from a vector of values you'll get something like this:

```
> lapply(rnorm(5),abs)
[[1]]
[1] 0.1300789

[[2]]
[1] 1.442206

[[3]]
[1] 1.239424

[[4]]
[1] 0.5090184

[[5]]
[1] 0.659651
```

Since there is only one element in each of the list vectors it doesn't really make sense to have this as a list. It would be much more useful if we could get back a simple vector rather than a list. If you therefore use `sapply` instead then R will try to simplify the result into a vector which should make the data returned more easily able to be incorporated into downstream analysis.

```
> sapply(rnorm(5),abs)
[1] 1.7945917 1.6110064 1.1298580 0.1672146 0.5205267
```

Functions

In the apply examples in the previous section we used built in R functions in the apply statements. Whilst this is perfectly valid the examples shown are somewhat artificial since they could generally have been performed with normal vectorised operations using these same functions rather than requiring the complexity of an apply statement. The real power of loops really only starts to assert itself when you start to construct custom functions which offer functionality which isn't generally present in the core functions.

Writing your own function is very simple. A function is created the same way as any data structure using the `<-` or `->` assignment operators. The only difference is how you specify the block of code you wish to be associated with the name you supply.

Function definitions start with the special `function` keyword, followed by a list of arguments in round brackets. After this you can provide a block of code which uses the variables you collected in the function definition.

Data is passed back out of a function either by the use of an explicit `return` statement, or if one of these is not supplied, then the result of the last function processed within the function is returned.
A typical function might look like this:

```
my.function <- function (x) {
      print(paste("You passed in",x))
}
```

Once you've constructed a function you can use it the same as any other core function in R.

```
> my.function(5)
[1] "You passed in 5"
```

Functions are vectorised by default so if you pass it a vector then the function will be run across each element of the vector and you will get back a vector with the results of the function.

```
> my.function(1:5)
[1] "You passed in 1" "You passed in 2" "You passed in 3" "You passed in 4" "You
passed in 5"
```

Passing arguments to functions

When you construct a function you set up the set of required and optional parameters which have to be passed to it. The syntax for setting these up is the same as you see in the function documentation in R. Default parameters can just be assigned a name which is then valid within the function body. Optional parameters are specified with name=value pairs where the value specified is used unless the user overrides this when calling the function.

R functions do not have any mechanisms for type or sanity checking the data which is provided so you should perform whatever validation you require within the function body. Fatal errors, where the function should exit immediately should be reported using the `stop()` function. Warnings can be issues using the `warning()` function which will report a problem but allow processing to continue.
The example below constructs a function which calculates the median value from a list of values after removing outliers.

```
> clipped.median <- function (x,min=0,max=100) {
  if (! (typeof(x) == "integer" | typeof(x) == "numeric" | typeof(x) ==
"double")) {
    stop(paste("X must be numeric not",typeof(x)))
  }
  if (max<=min) {
    warning(paste(max,"is less than or equal to ",min))
  }
  return (median(x[x>=min & x<=max]))
}
```

In normal usage this function will operate as shown below:

```
> clipped.median(-100:1000)
[1] 50

> clipped.median(-100:1000, min=-1000)
[1] 0

> clipped.median(-100:1000, max=500)
[1] 250

> clipped.median(-100:1000, min=-100, max=500)
[1] 200
```

This function also performs two checks. It sees if the vector x passed in is a valid numeric type and will stop if it isn't.

```
> clipped.median(c("A","B","C"))
Error in clipped.median(c("A", "B", "C")) : X must be numeric
```

It will also check to see if the max value passed in is higher than the min value. If this isn't the case the function will not stop, but will write out a warning which the user will see to indicate that the parameters used are not sensible.

```
> clipped.median(1:10,max=10,min=20)
[1] NA
Warning message:
In clipped.median(1:10, max = 10, min = 20) :
  10 is less than or equal to  20
```

Because of the way we've constructed this function we can optionally add max and min values, but we have to supply the x value since there is not default for this. If we try to run the function with no arguments then we'll get an error.

```
> clipped.median()
Error in typeof(x) : argument "x" is missing, with no default
```

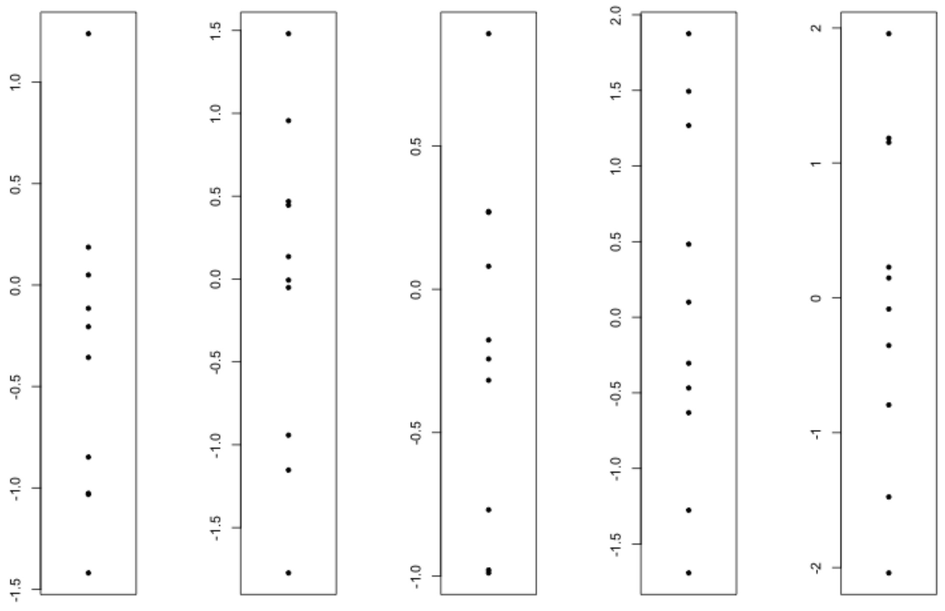
Using functions with apply

Now that you know how to construct your own functions you will suddenly find that apply statements are a lot more powerful than they used to be. You can use them either to perform more complex transformations than you could do with the core functions, or you can use them as a simple way to automate operations.

```
> lots.of.data
            V1           V2          V3          V4          V5
1   0.04967188 -0.051192183  0.89172871 -0.30428023  1.95786789
2   1.23928194  0.135691329 -0.97940217 -1.69096872 -0.35344006
3  -0.20528145  0.468364022 -0.24318453  0.09980036 -0.08327642
4  -0.35676245 -0.942082095  0.27140134  1.26787384 -2.03970202
5  -1.03212361 -1.773402735 -0.98974102  0.48265880  0.14707311
6   0.18666712  0.445848673  0.08022584  1.49388520  1.15202492
7  -0.84804009 -1.152551436 -0.31755238 -0.46805887  0.22729159
8  -1.41925226  1.480603315  0.26814839 -0.63131064  1.18353603
9  -1.02707818 -0.006430327 -0.76944039  1.87510061 -1.47620999
10 -0.11489543  0.955823903 -0.17702384 -1.27632832 -0.79418689

> apply(lots.of.data,1,clipped.median,min=-100)
 [1]  0.04967188 -0.35344006 -0.08327642 -0.35676245 -0.98974102  0.44584867 -
0.46805887  0.26814839 -0.76944039 -0.17702384
```

You can also construct functions on the fly within the apply statement rather than having to create a separate named function.

```
> par(mfrow=c(1,5))
> apply(lots.of.data,2,function(x)stripchart(x,vertical=TRUE,pch=19))
NULL
```



Packages

All of the work we've done to this point has used only the core functionality of R. Whilst the R core provides some very powerful functionality there are a wide range of extensions which can be used to supplement this, and which allow you to reduce the amount of work you need to do by reusing code which other people have written previously. The mechanism for extending the core R functionality is though R packages. In this course we won't look at how to construct your own modules, but will instead focus on importing additional functionality from modules to use within your R scripts.

Finding packages

There are a couple of common locations where you might find R packages which you want to install. The generic location for R packages is a system called CRAN (the comprehensive R archive network), which is linked in to every R installation and from which new packages can be installed. The main CRAN site is at http://cran.r-project.org/, but it's not a very user friendly site and although it lists all of the packages hosted there it doesn't have good search functionality. Rather than searching for new CRAN packages on the main CRAN site it's probably better to use one of the dedicated R search sites such as:

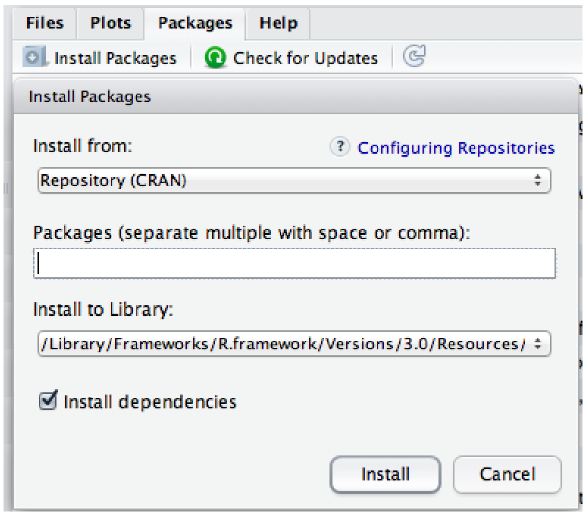
- http://www.rseek.org/
- http://crantastic.org/
- http://www.inside-r.org/packages

The other main R archive of packages for biological analyses is bioConductor. This is a coordinated set of packages which share a common infrastructure and are highly interconnected. BioConductor is effectively the standard for reference implementations of new methods in bioinformatics. It is a separately maintained repository from CRAN, but it is easy to get code from this repository as well as from CRAN.

Installing packages

New packages from CRAN can be installed from either the R command line or from the RStudio GUI. From the command line you can use the `install.packages()` function to install a named package and all of its dependencies in a single command. R will check what permissions you have on your system and will either install the package for all users if you have permission to do that or just for you if you are not an administrator.

If you'd prefer to use a graphical front end then there is a package installation tool in RStudio into which you can simply enter the name of a package and have it installed.



Install from: Repository (CRAN)

Packages (separate multiple with space or comma):

Install to Library: /Library/Frameworks/R.framework/Versions/3.0/Resources/

☑ Install dependencies

Bioconductor packages are generally installed from the command line, and Bioconductor have created their own installer which performs a similar job to `install.packages()`. Installing a Bioconductor package is simple a case of loading the new installer and then specify the packages you want to install.

```
> source("https://www.bioconductor.org/biocLite.R")
trying URL
'https://www.bioconductor.org/packages/2.13/bioc/bin/macosx/contrib/3.0/BiocInsta
ller_1.12.0.tgz'
Content type 'application/x-gzip' length 46403 bytes (45 Kb)
opened URL
==================================================
downloaded 45 Kb

Bioconductor version 2.13 (BiocInstaller 1.12.0), ?biocLite for help

> biocLite("DESeq")
BioC_mirror: http://bioconductor.org
Using Bioconductor version 2.13 (BiocInstaller 1.12.0), R version 3.0.2.
Installing package(s) 'DESeq'
also installing the dependencies ‘DBI’, ‘RSQLite’, ‘IRanges’, ‘xtable’, ‘XML’,
‘AnnotationDbi’, ‘annotate’, ‘BiocGenerics’, ‘Biobase’, ‘locfit’, ‘genefilter’,
‘geneplotter’, ‘RColorBrewer’
```

Using packages

Using a package within your R script is as simple as loading it and then using the functionality it provides. Ultimately what most packages do is to add new functions into your R session and using them is done in the same way as for any other core R function.

To load a package into an R session you use the `library()` function. You pass in the name of the package you want to use in this session (it must already have been installed) and R will load not only the library you specified, but also any other libraries on which it depends.

```
> library("DESeq")
Loading required package: BiocGenerics
Loading required package: parallel

Attaching package: ‘BiocGenerics’

The following objects are masked from ‘package:parallel’:

    clusterApply, clusterApplyLB, clusterCall, clusterEvalQ, clusterExport,
clusterMap, parApply, parCapply, parLapply,
    parLapplyLB, parRapply, parSapply, parSapplyLB

The following object is masked from ‘package:stats’:

    xtabs

The following objects are masked from ‘package:base’:
    anyDuplicated, append, as.data.frame, as.vector, cbind, colnames, duplicated,
eval, evalq, Filter, Find, get, intersect,
    is.unsorted, lapply, Map, mapply, match, mget, order, paste, pmax, pmax.int,
pmin, pmin.int, Position, rank, rbind,
    Reduce, rep.int, rownames, sapply, setdiff, sort, table, tapply, union,
unique, unlist

Loading required package: Biobase
Welcome to Bioconductor

    Vignettes contain introductory material; view with 'browseVignettes()'. To
cite Bioconductor, see 'citation("Biobase")',
    and for packages 'citation("pkgname")'.

Loading required package: locfit
locfit 1.5-9.1     2013-03-22
Loading required package: lattice
    Welcome to 'DESeq'. For improved performance, usability and functionality,
please consider migrating to 'DESeq2'.
```

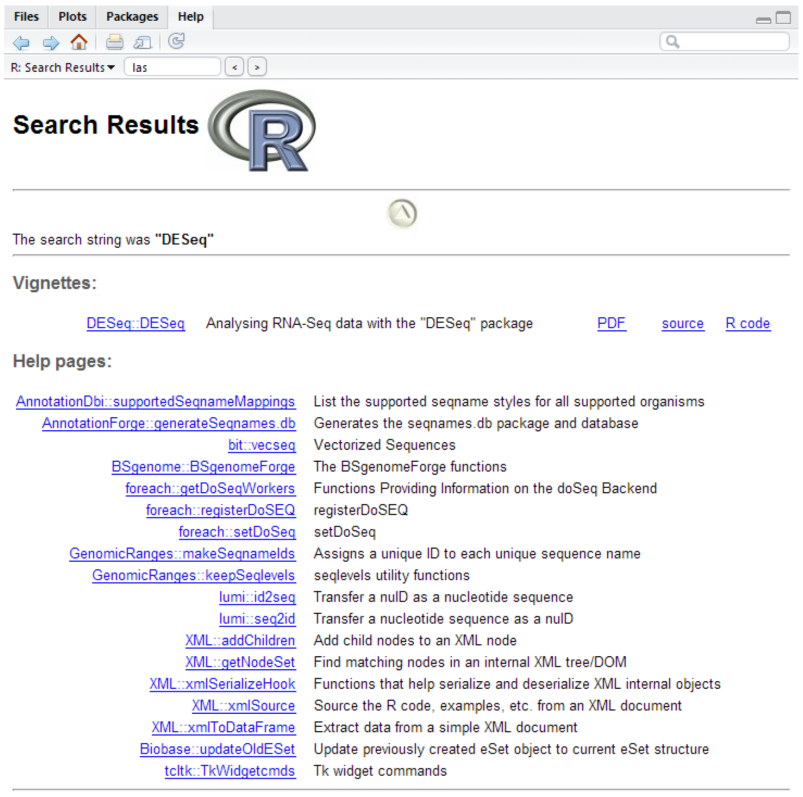
Since packages can define functions of any name then you will get a warning if a package you have loaded includes a function name which is already in use in your session. Sometimes these kinds of clash are intentional, but sometimes they aren't and you should check whether newly loaded libraries might have interfered with your existing code by hiding functions you were using. For this reason it's not a good idea to load in more libraries than you actually need for your analysis.

All packages in CRAN or Bioconductor come with some documentation which describe the functionality of the package and provide some examples of its use so you can clearly see how the package works.

There are two types of documentation which may come with a package. The first would be standard R documentation which would be similar to the system you'd use for viewing the help for any core R function. The second is a system called 'vignettes' which are a more guided set of instructions and guidance for the use of the package. Any given package could have either or both (but hopefully at least one) of these types of documentation.

You can search for the documentation for a package by using the standard R search query, ie:

```
??DESeq
```



Search Results

The search string was "DESeq"

Vignettes:

DESeq::DESeq	Analysing RNA-Seq data with the "DESeq" package	PDF	source	R code

Help pages:

AnnotationDbi::supportedSeqnameMappings	List the supported seqname styles for all supported organisms
AnnotationForge::generateSeqnames.db	Generates the seqnames.db package and database
bit::vecseq	Vectorized Sequences
BSgenome::BSgenomeForge	The BSgenomeForge functions
foreach::getDoSeqWorkers	Functions Providing Information on the doSeq Backend
foreach::registerDoSEQ	registerDoSEQ
foreach::setDoSeq	setDoSeq
GenomicRanges::makeSeqnameIds	Assigns a unique ID to each unique sequence name
GenomicRanges::keepSeqlevels	seqlevels utility functions
lumi::id2seq	Transfer a nuID as a nucleotide sequence
lumi::seq2id	Transfer a nucleotide sequence as a nuID
XML::addChildren	Add child nodes to an XML node
XML::getNodeSet	Find matching nodes in an internal XML tree/DOM
XML::xmlSerializeHook	Functions that help serialize and deserialize XML internal objects
XML::xmlSource	Source the R code, examples, etc. from an XML document
XML::xmlToDataFrame	Extract data from a simple XML document
Biobase::updateOldESet	Update previously created eSet object to current eSet structure
tcltk::TkWidgetcmds	Tk widget commands

This searches for all documentation relating to the `DESeq` package. In this case you'd see that there was no core documentation for this package, but that it did contain a vignette. You can view the PDF of the vignette or even run the source of the worked example by clicking on the links in the help.

Generally running code in a package is simply a case of following the instructions or vignettes. You will need to use your code R knowledge in order to prepare the correct input data for the start of the pipeline but there is no standard rule about how code distributed in packages should be run or structured.

Documenting your analysis

The interactive nature of an R session means that you will often perform more analyses than you might end up using. Running your analysis in a program such as R-studio will keep a complete record of every command you ran in the session, but this isn't hugely useful when reviewing the work you've done.

Building up scripts as you work

A better approach is to create an R script file in the text editor into which you copy the path of commands which actually produce the end result from your analysis. You can easily do this by moving code between the console, history and editor windows either using standard copy/paste or via the special buttons in the history window which move selected blocks to different windows within RStudio. The advantages of working this way are that you can keep a clean record of the analysis you've done and can easily modify this to try out new variations.

At the end of your analysis it is always a good idea to re-run the full analysis script you've built up to ensure that you really can generate the output that you've ended up with using the script you've recorded. It's surprisingly easy to miss out one or two steps which you ran in the console and then never passed back to your script. You should note that just because your script works when you run it at the end of your analysis session doesn't necessarily mean that it will also work when you next start up an R session. You should be careful to ensure that you aren't using variables or functions which are defined in your current session but aren't created in your script. If you want to be really sure you can wipe out all of your stored variables using `rm(list=ls())` and then try to re-run your script. If this works then you can be sure that the script contains everything you actually need to replicate your analysis.

Documenting your analysis with Notebooks

As you are creating your analysis script you can insert comments between the actual lines of code to provide some explanation of what your intention was with each statement. Standard R comments can be created by placing a `#` character on a line and then everything after that will be treated as comment text and will not be executed when the script is run. R has no support for multi-line comments.

Whilst standard R comments are useful they do not completely document your analysis as they have a few limitations:

1. Writing larger blocks of text is difficult as you have to comment each line
2. There is no formatting support for the comment text
3. It is difficult to annotate the output of analyses (graphs etc.) since these won't be associated with the corresponding line of code.

One solution to this is to use a package called KnitR and the associated concept of an R workbook. KnitR is a package which allows you to annotate your code with more complete comments and can interleave the results of your analysis with the code which produced it, allowing you to comment on the results as well as the code easily. It also supports a number of different formatting styles which allow you to produce complete reports which as well as documenting the analysis also provide the code in a format which would allow it to be re-run directly.

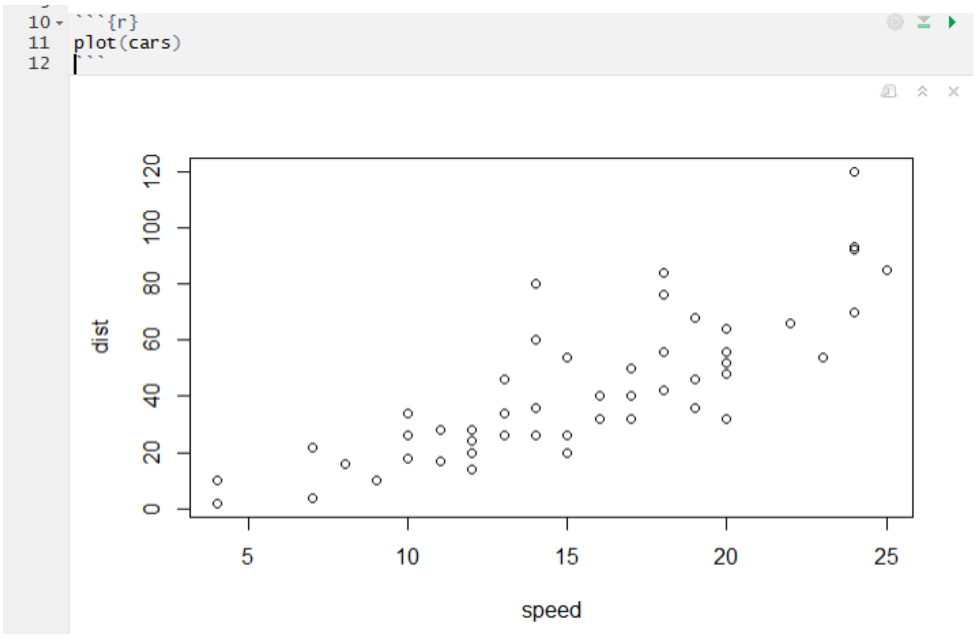
Using R Notebooks

To create a new Notebook markdown file you simply go to File > New File > R Notebook. The first time you do this you may be prompted to install some R packages which support notebooks, but this should only happen once. You should then see a new document open up in your text editor which looks like this:

~~~
---
title: "R Notebook"
output: html_notebook
---

This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When you execute
code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or by
placing your cursor inside it and pressing *Ctrl+Shift+Enter*.

```{r}
plot(cars)
```

Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by
pressing *Ctrl+Alt+I*.

When you save the notebook, an HTML file containing the code and output will be
saved alongside it (click the *Preview* button or press *Ctrl+Shift+K* to preview
the HTML file).
~~~

This is a simple template file which actually shows most of the code you need to create a notebook report.

There are two main ways of working with this type of report. Firstly it can be used interactively. To test this put the cursor to the list which says "plot(cars)" and press Control+Shift+Return. You will see that the notebook expands so that the results of this command are now seen inline in the document below the R code block which generated them.

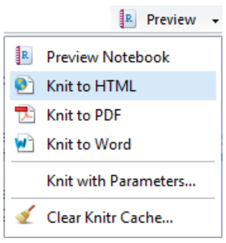


~~~
```{r}
plot(cars)
```
~~~

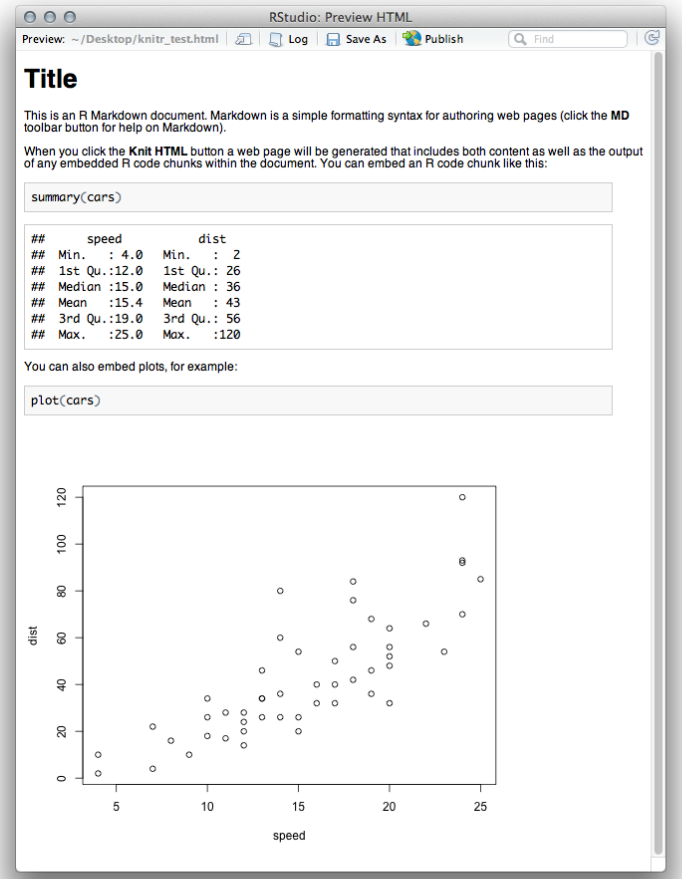
You can therefore use an R notebook in a very similar manner to developing scripts in the text editor except that you now separate the code from the comments by using code blocks (the sections starting with `` ```{r} `` and ending with `` ``` ``), and the results appear inline with the code.

You can add new code blocks either by typing the appropriate block headers manually, or by using the Insert button at the top of the Notebook window to create one automatically at the current edit point.

You can run (or re-run) a code block by pressing Control+Shift+Return and this will update the output below that block. Outputs are only updated when the block is specifically refreshed so changes you make to your data will not be automatically reflected in plots you have already drawn.



When you have finished your analysis in a notebook you can finally generate a fully rendered report. To do this you must first save the notebook. You can put the file wherever you like, but you MUST save it with a .Rmd extension otherwise later steps in the process won't work. Once you've done that you can open the drop down box under Preview at the top of the notebook and select "Knit to HTML".



Title

This is an R Markdown document. Markdown is a simple formatting syntax for authoring web pages (click the **MD** toolbar button for help on Markdown).

When you click the **Knit HTML** button a web page will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2
##  1st Qu.:12.0   1st Qu.: 26
##  Median :15.0   Median : 36
##  Mean   :15.4   Mean   : 43
##  3rd Qu.:19.0   3rd Qu.: 56
##  Max.   :25.0   Max.   :120
```

You can also embed plots, for example:

```
plot(cars)
```

You will also find that an HTML file has been created in the same directory as your Rmd file which also contains this report as a single file HTML document with embedded graphics.

R code differences in Notebooks

In general, the R code you write in a notebook is exactly the same as what you would write in any other type of R script. There are only a couple of ways in which you need to slightly adapt the way you write or run code.

The first is the use of data files, and the setting of the working directory. Inside a notebook the working directory is automatically set to be the directory in which the notebooks Rmd file is saved. Notebooks are designed with the intention that you will keep the notebook plus all of your data in the same folder. If you do this then life is easy.

If, for some reason, you need to access data which is in another folder, then you can do this by using the `setwd()` function, as you would in any R script – however – inside a notebook the altered working directory will only persist for the duration of the code block in which it was changed (and you will get a warning about this). If you want to read files outside the save directory of the Rmd file then you will need to reset the working directory in every block where you want to read a file.

The other slight difference in a notebook is the way that you construct multi-layer plots. Multi-layer plots are constructed by sequentially calling functions to construct a base layer followed by additional layers, so you can do something like:

```
barplot(1:3)
abline(h=2)
```

To draw a barplot with a horizontal line running across it. In a normal R session you run these as two distinct functions.

Within an R notebook the output graphical device is reset after each plotting operation, so if you run the `barplot` function followed by the `abline` function then you will get an error:

```
Error in int_abline(a = a, b = b, h = h, v = v, untf = untf, ...) :
  plot.new has not been called yet
```

To make this work properly you need to run both the `barplot` and the `abline` in a single operation. This means either selecting both lines and pressing Control+Return, or just doing shift+Control+Return to run all of the code in the current block (which is easier).

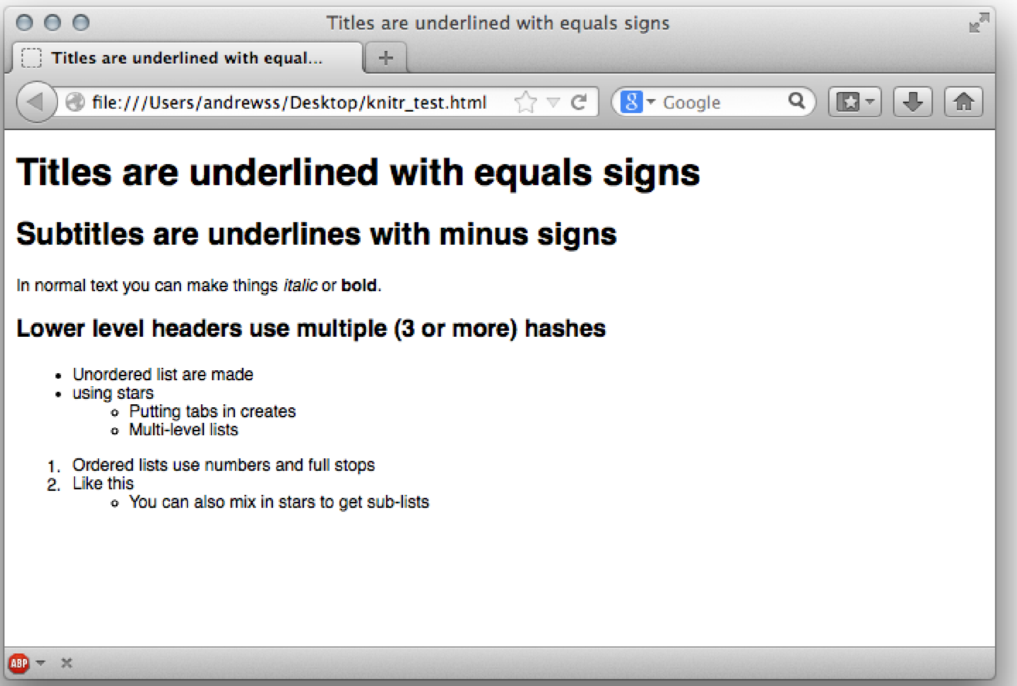
Along similar lines, in a notebook each code block gets its own graphical output device (the area in the book immediately below the block). This means that changes to the plotting area using `par` will also only affect the current graphical block, and not the rest of the document.

Editing Markdown

The notebook files are split into two types of section, one contains the markdown formatted comments around your code, the other is the actual code blocks which will appear in the report you created but will also be executed and have their output inserted into the document immediately after the code.
The markdown parts of the document can have various styles of formatting applied to them to make your document look nicer and easier to read. You can see a full list of markdown formatting options by pressing the small MD icon immediately above the text editor. A simple example of some formatting is shown below:

```
Titles are underlined with equals signs
=======================================

Subtitles are underlines with minus signs
-----------------------------------------

In normal text you can make things *italic* or **bold**.

### Lower level headers use multiple (3 or more) hashes

* Unordered list are made
* using stars
  * Putting tabs in creates
    * Multi-level lists

1. Ordered lists use numbers and full stops
2. Like this
  * You can also mix in stars to get sub-lists
```

This code would produce the following document:


Titles are underlined with equals signs
Titles are underlined with equal...
file:///Users/andrewss/Desktop/knitr_test.html
Google

Titles are underlined with equals signs

Subtitles are underlines with minus signs

In normal text you can make things *italic* or **bold**.

Lower level headers use multiple (3 or more) hashes

- Unordered list are made
- using stars
  - Putting tabs in creates
  - Multi-level lists

1. Ordered lists use numbers and full stops
2. Like this
   - You can also mix in stars to get sub-lists

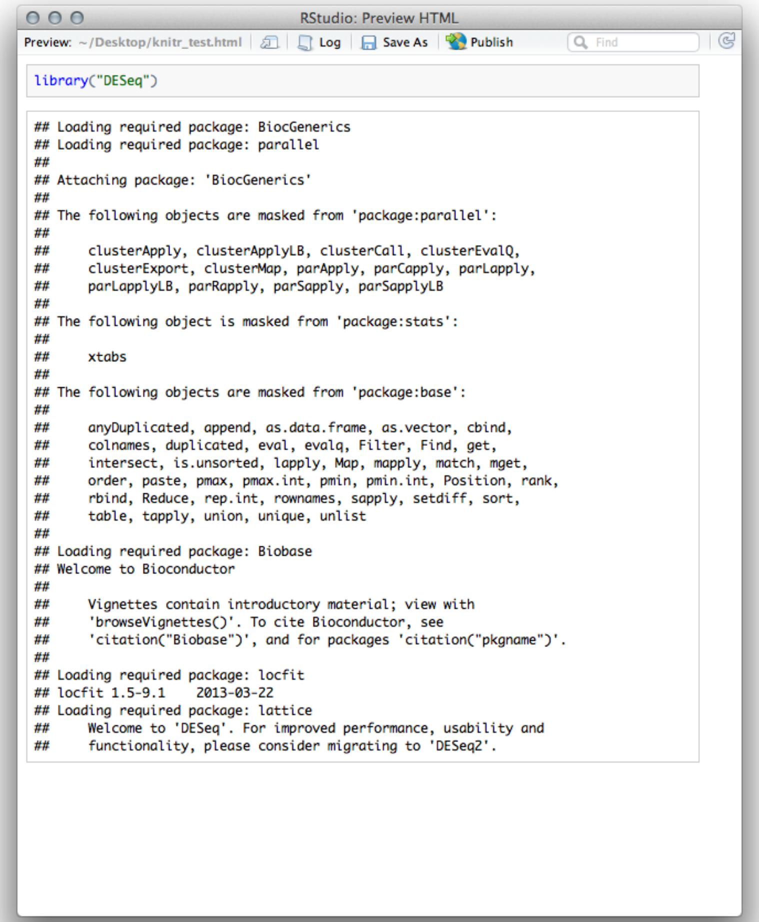
R code block options

For some R code you will want to change the default output options. The two most common things you will need to change are the types of output which are shown, and the size of graphical figures. Both of these are done by editing the R block header line.

To control the level of output you can choose to either completely silence a block of code so that none of its output is included in the report or you can selectively remove parts of the output. The options for removing parts of the output are based around the different types of message which R can generate, specifically you can remove errors, warnings or messages. This is particularly useful if part of your code loads packages or performs other operations where progress messages are displayed which you don't want to show up in the output.

As an example if we have a simple script which loads the DESeq package the default output shows a lot of progress messages which we don't really need to see:

~~~
```{r}
library("DESeq")
```
~~~



```
library("DESeq")
```

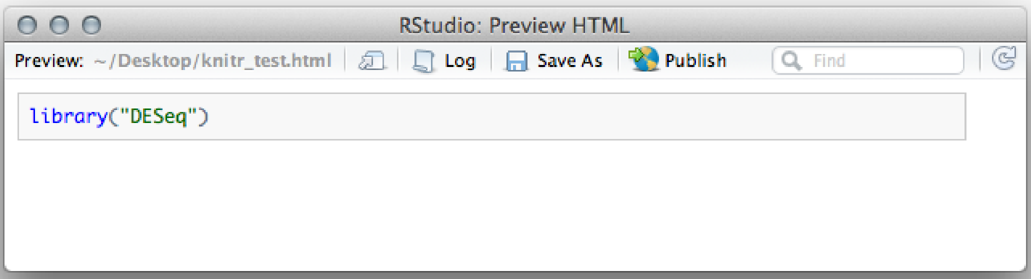
```
## Loading required package: BiocGenerics
## Loading required package: parallel
##
## Attaching package: 'BiocGenerics'
##
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
##
## The following object is masked from 'package:stats':
##
##     xtabs
##
## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, as.vector, cbind,
##     colnames, duplicated, eval, evalq, Filter, Find, get,
##     intersect, is.unsorted, lapply, Map, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,
##     rbind, Reduce, rep.int, rownames, sapply, setdiff, sort,
##     table, tapply, union, unique, unlist
##
## Loading required package: Biobase
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
##
## Loading required package: locfit
## locfit 1.5-9.1    2013-03-22
## Loading required package: lattice
##     Welcome to 'DESeq'. For improved performance, usability and
##     functionality, please consider migrating to 'DESeq2'.
```

To alter the output we can put extra parameters into the brackets which define the R code block in the Rmd file. The additional options come as KEY=VALUE pairs and multiple pairs can be separated by commas. In this case the options that are relevant are:

`message=TRUE/FALSE`	Show messages
`warning=TRUE/FALSE`	Show warning messages
`error=TRUE/FALSE`	Show error messages

So if we turn all of these off we don't see any output, but the library is still loaded and can be used by later statements in the report.

~~~
```{r warning=FALSE, message=FALSE, error=FALSE}
library("DESeq")
```
~~~



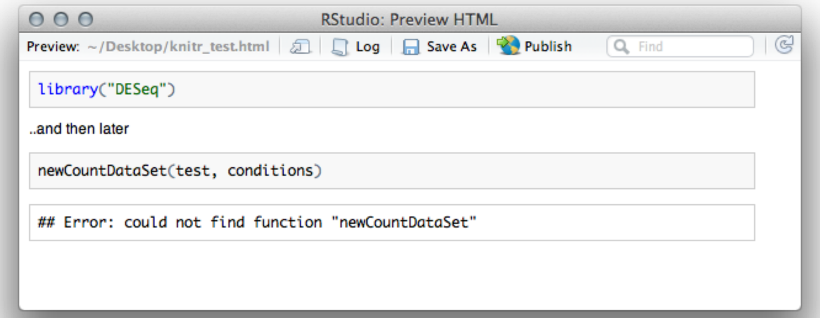
```
library("DESeq")
```

If you want to go one step further you can also set `eval=FALSE` which will not even run the R code in this block but it will still show up in the report.

~~~
```{r eval=FALSE}
library("DESeq")
```
~~~

..and then later

~~~
```{r}
newCountDataSet(test,conditions)
```
~~~



```
library("DESeq")
```

..and then later

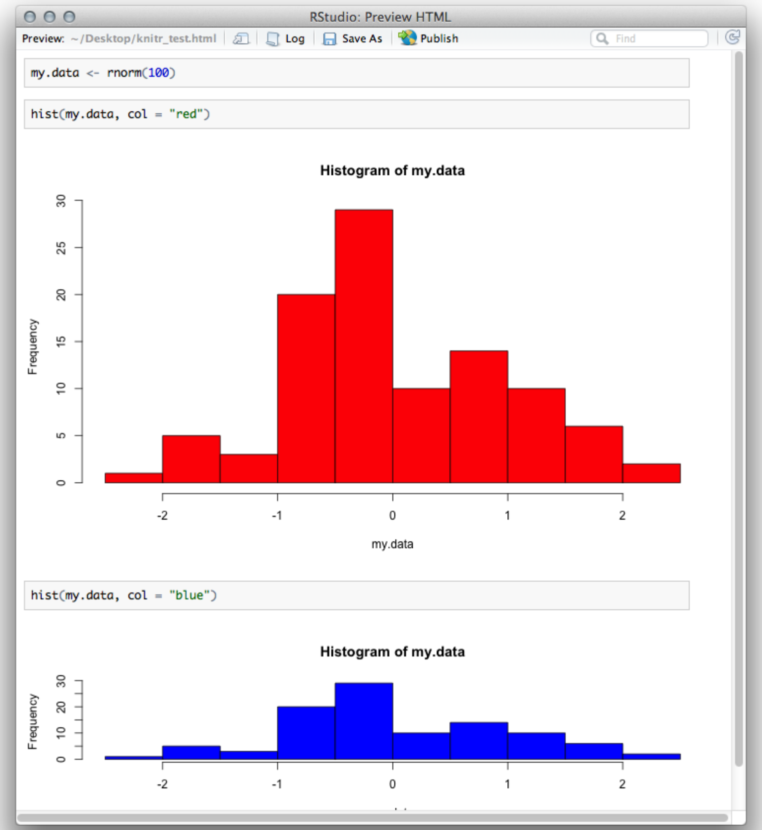
```
newCountDataSet(test, conditions)
```

```
## Error: could not find function "newCountDataSet"
```

The other thing you will often want to change will be the size of the graphics files created. Each graphic in a KnitR document gets a new graphics device and these are standard R graphics devices, so you can still use functions such as par to set up the layout of the graphics. What you can change in the Rmd file are the sizes of the output in the report.

Changing the size of the output graphics is simply a case of setting `fig.height` and `fig.width` values in the R block options.

~~~
```{r}
my.data <- rnorm(100)
```

```{r fig.height=6, fig.width=10}
hist(my.data,col="red")
```

```{r fig.height=3, fig.width=10}
hist(my.data,col="blue")
```
~~~



```
my.data <- rnorm(100)
```

```
hist(my.data, col = "red")
```

```
hist(my.data, col = "blue")
```