

Advanced R

Version 3.0

Licence

This manual is © 2013-15, Simon Andrews.

This manual is distributed under the creative commons Attribution-Non-Commercial-Share Alike 2.0 licence. This means that you are free:

- to copy, distribute, display, and perform the work
- to make derivative works

Under the following conditions:

- Attribution. You must give the original author credit.
- Non-Commercial. You may not use this work for commercial purposes.
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a licence identical to this one.

Please note that:

- For any reuse or distribution, you must make clear to others the licence terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Full details of this licence can be found at

<http://creativecommons.org/licenses/by-nc-sa/2.0/uk/legalcode>

Table of Contents

Licence	2
Introduction	4
Manipulating Data	5
Recap of Filtering	5
Access by index	5
Access by name	5
Access by Boolean	6
Extensions to Lists and Data Frames	7
Deduplicating Data	8
Simple Deduplication	8
Complete Duplicate Removal	9
Building compound variables and other text manipulations	10
Merging Data Sets	12
Adding new rows to an existing dataset	12
Merging columns between datasets	12
Simple Transformations	15
Looping	17
Looping through data frames	17
Data types when using apply	17
Collating data	19
Looping through lists and vectors	20
Functions	21
Passing arguments to functions	21
Using functions with apply	23
More Graphs	Error! Bookmark not defined.
Recap of R "painters model"	Error! Bookmark not defined.
Global graphical parameters	Error! Bookmark not defined.
Margins	Error! Bookmark not defined.
Fonts	Error! Bookmark not defined.
Multi-Panel plots	Error! Bookmark not defined.
Multiple Plot overlays	Error! Bookmark not defined.
Packages	24
Finding packages	24
Installing packages	24
Using packages	25
Documenting your analysis	28
Building up scripts as you work	28
Documenting your analysis	28
Installing and configuring KnitR	28
Creating a KnitR markdown file	29
Editing Markdown	30
Adding R code to a KnitR document	31

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. The course starts by reviewing the core data structures and looks at how to merge and reshape data structures. We then look at how to efficiently loop over data structures to run the same analyses multiple times. We will then cover some more complex graphing, showing how to produce multi-layer graphs of different types, and finally we will look at extending the core R functionality with modules, and will illustrate this using a module which makes it easy to document the work you've been doing in an R session.

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 Boolean

If you generate 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 boolean 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
```

Boolean vectors can be combined with logical operators (& and |) to create more complex filters.

```
> 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
```

Extensions to Lists and Data Frames

All of the methods outlined 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 sets of accessors 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
```

In all recent versions of R you can get away with using single brackets to access list items, but this is discouraged since using double brackets immediately indicate to anyone reading your code in the future that this is a list and not a vector.

The other piece of useful syntax needed for lists and data frames is the `$` notation for extracting columns or list items.

```
> 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"
```

Deduplicating Data

When working with large datasets it is common to find that a set of identifiers you thought were unique actually contained duplicate values. This could be because of genuine coincidental clashes (two unrelated genes being given the same names for example) or from other values which have ended up in that column (blank values or placeholders like “unknown” for example). If the downstream analysis of the data is going to be hindered by the presence of these duplicates then it’s often easier to remove them.

A simple example is shown below where the name column contains the duplicate value GED6 and therefore can’t be used as row names for a 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

> rownames(example.data) <- example.data$Name
Error in `rownames<-.data.frame`(`*tmp*`, value = value) :
  duplicate 'row.names' are not allowed
In addition: Warning message:
non-unique value when setting 'row.names': 'GED6'
```

Simple Deduplication

The simplest method of deduplication involves the use of the `duplicated()` function. This function takes a vector of values and returns a boolean array to say if each item in the list is a duplicate of one which has already been seen further up the list.

```
> duplicated(example.data$Name)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

If this is then applied as a row selector on the data frame then it’s easy to generate a subset of the original data which is guaranteed not to contain duplicates. Since we want to keep the values which

are NOT duplicated we can prefix the logical test with the `!` operator which reverses the boolean values in the original set.

```
> ! duplicated(example.data$Name)
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE

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

> rownames(example.data.simple.dedup) <- example.data.simple.dedup$Name

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

Complete Duplicate Removal

In the above example we have generated a unique set of Name values by removing the duplicate GED6 entry, however the final list still contains the first GED6 entry, and the selection of which of the duplicated entries was kept is based solely on which occurred first in the list. Rather than just removing the duplicated entries you might therefore want to completely remove any entry which occurred more than once so it doesn't appear at all in the final list.

This type of more advanced filtering can also be achieved using the `duplicated()` function, but with some extra logic to the original test.

In the `duplicated()` documentation we can see that there is an extra parameter `fromLast` which starts the search at the end of the vector instead of the start. If we'd added this to the original filter we'd have ended up with a deduplicated list where the last entry was kept instead of the first.

```
> example.data[! duplicated(example.data$Name, fromLast=TRUE),]
      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
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
```

We can therefore use some boolean logic to combine these two variations on `duplicated()` to completely remove any entry which has a duplicate in the set. We combine the forward and reverse

search results with a logical and (using `&`) and any entries which are duplicated will be completely removed.

```
> ! duplicated(example.data$Name) & ! duplicated(example.data$Name,
fromLast=TRUE)
```

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

```
> example.data[! duplicated(example.data$Name) & !
duplicated(example.data$Name, fromLast=TRUE),]
```

	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
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

Building compound variables and other text manipulations

In the previous examples we used an existing column in a data frame to deduplicate the 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"
```

Another common operation would be removing part of a string by doing 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"
```

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
- `grep(pattern, data)` - searches for pattern and provides a vector of indices which match
- `grepl(pattern, data)` - searches for pattern and provides a boolean vector of hits
- `tolower(data)` - converts a string to lower case - useful for case insensitive searches
- `toupper(data)` - converts a string to upper case
- `match(searches, data)` - finds the first index positions of a vector of search terms in a larger data set

Going back to our original problem, we can therefore use `paste` to make up a set of position strings from the various position columns.

```
> paste(example.data$Chr, ":", example.data$Start, "-", example.data$End,
":", example.data$Strand, sep="")
[1] "1:100-200:1" "1:300-400:-1" "1:300-400:-1" "2:300-400:1" "2:500-
600:1" "2:700-800:-1" "3:150-300:1" "3:400-600:1"
[9] "4:125-350:-1" "4:400-550:-1"
```

We could work with these values directly, but it's often easier to include them as a new column in the original data frame.

```
> cbind(example.data, location=paste(example.data$Chr, ":",
example.data$Start, "-", example.data$End, ":", example.data$Strand,
sep="")) -> example.data.with.location
```

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

Now that there is a single column containing a complete merged identifier we can use the methods described previously to deduplicate the data.

```
>
example.data.with.location[!duplicated(example.data.with.location$location)
,]
  Name Chr Start End Strand Value      location
1 ABC1   1   100 200     1   5.32 1:100-200:1
2 MEV1   1   300 400    -1   7.34 1:300-400:-1
4 GED6   2   300 400     1   6.36 2:300-400:1
5 RPL1   2   500 600     1   2.12 2:500-600:1
6 NES2   2   700 800    -1   8.48 2:700-800:-1
7 AKT1   3   150 300     1   3.25 3:150-300:1
```

```
8 GED6 3 400 600 1 3.65 3:400-600:1
9 XYN1 4 125 350 -1 7.23 4:125-350:-1
10 RBP1 4 400 550 -1 4.54 4:400-550:-1
```

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 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
```

Merging columns between datasets

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. If you have multiple columns to add then you can use `cbind` to do this in a single operation.

```
> 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

```
> 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.

A useful starting place for this type of operation is being able to find out which identifiers in one dataset can be matched with a corresponding identifier in another dataset. This can be done with a special built-in operator `%in%`.

```
> days
```

```
[1] "Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"
```

```
> to.test
```

```
[1] "Jan" "Feb" "Thur" "Tue" "Oct" "Wed"
```

```
> to.test %in% days
```

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

Here we can see that only the 4th and 6th indices of `to.test` appear in `days`. If we wanted to find out what those values were we could simply do:

```
> to.test[to.test %in% days]
[1] "Tue" "Wed"
```

If we know that one dataset is an incomplete subset of another, even if the rows are not in the same order, we can use this method along with `cbind` to merge the two datasets.

```
> days.1
      Long Value1
Mon   Monday     5
Tue   Tuesday     2
Wed   Wednesday   6
Thu   Thursday     8
Fri   Friday    21
Sat   Saturday     5
Sun   Sunday      2

> days.2
      Value.2
Mon         7
Jan         2
Wed        67
Feb       234
Tue         1
Mar        78
Sat         2

> rownames(days.1)[rownames(days.1) %in% rownames(days.2)] ->
rows.in.common
> rows.in.common
[1] "Mon" "Tue" "Wed" "Sat"

> cbind(days.1[rows.in.common,], days.2[rows.in.common,])
      Long Value1 days.2[rows.in.common, ]
Mon   Monday     5                      7
Tue   Tuesday     2                      1
Wed   Wednesday   6                     67
Sat   Saturday     5                      2
```

This type of retrieval only works if you're merging on row names, and can be very slow with large data sets. A more generic solution, which is also more efficient, is to use the `match` function to look up the indices for the set of names and then extract the corresponding data with a normal index retrieval. This can work on any column and can be applied to the output of `rownames` if you prefer.

```
> frame.1
      letter value.1
1         A        1
2         B        2
```

3	C	3
4	D	4
5	F	5

```
> frame.2
  letter value.2
1      B       1
2      G       2
3      E       3
4      F       4
5      D       5

> letters.in.common = frame.1$letter[frame.1$letter %in% frame.2$letter]
> cbind( frame.1[match(letters.in.common,frame.1$letter),] ,
        frame.2[match(letters.in.common,frame.2$letter),])
  letter value.1 letter value.2
2      B       2      B       1
4      D       4      D       5
5      F       5      F       4
```

This type of simple merging gets very complicated if you want to use a combination of columns to merge and if you want to keep partially filled rows where there is data for only some of the data sets you are merging. In these cases it's easier to use the built-in `merge` function.

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
```

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
      Col11 Col12 Col13 Col14 Col15 Col16 Col17 Col18 Col19 Col110
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)
      Col11 Col12 Col13 Col14 Col15 Col16 Col17 Col18 Col19 Col110
-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, boolean etc). This is most obviously the case when iterating over rows of data with mixed column types, where the vector passed to the function

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))
+ }
```

The `+` lines simply indicate a continuation in the command onto a new line and are not part of the actual command.

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

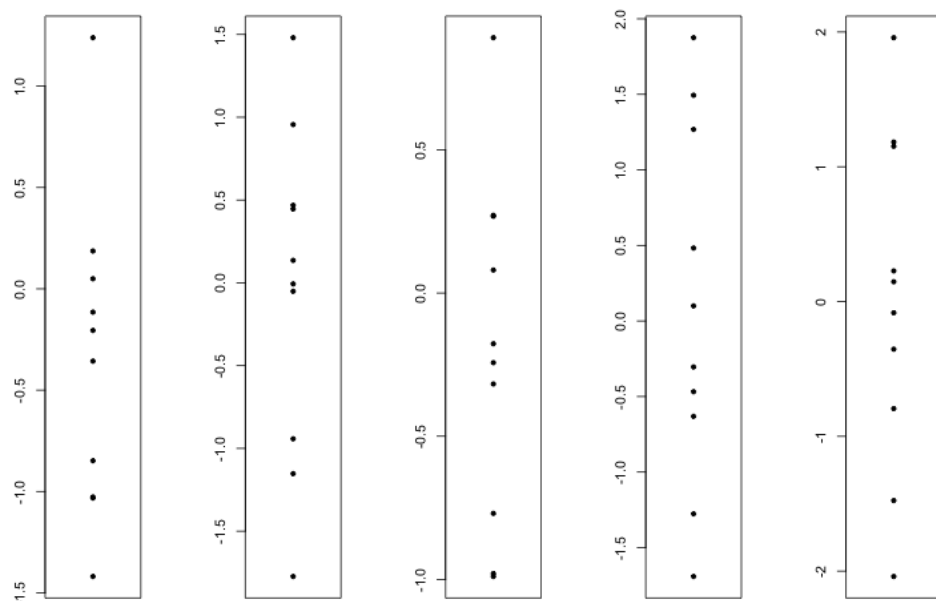
Now that you know how to construct your own functions you will suddenly find that apply statements are a lot more powerful than they used to be. You can use them either to perform more complex transformations than you could do with the core functions, or you can use them as a simple way to automate operations.

```
> 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 through 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:

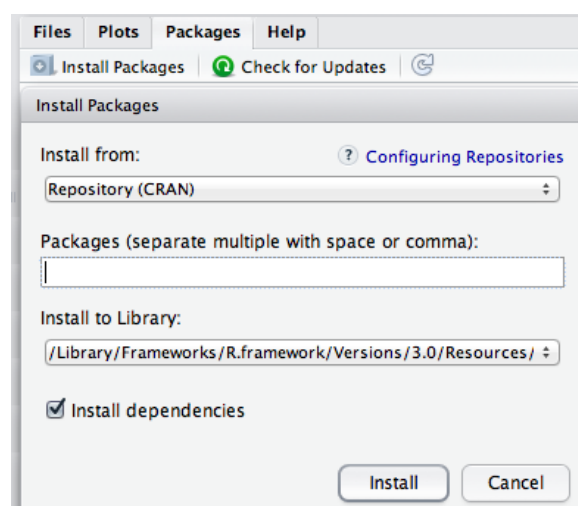
- <http://www.rseek.org/>
- <http://crantastic.org/>
- <http://www.inside-r.org/packages>

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.



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("http://www.bioconductor.org/biocLite.R")
trying URL
'http://www.bioconductor.org/packages/2.13/bioc/bin/macosx/contrib/3.0/Bioc
Installer_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', 'genefilter', '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'.
```

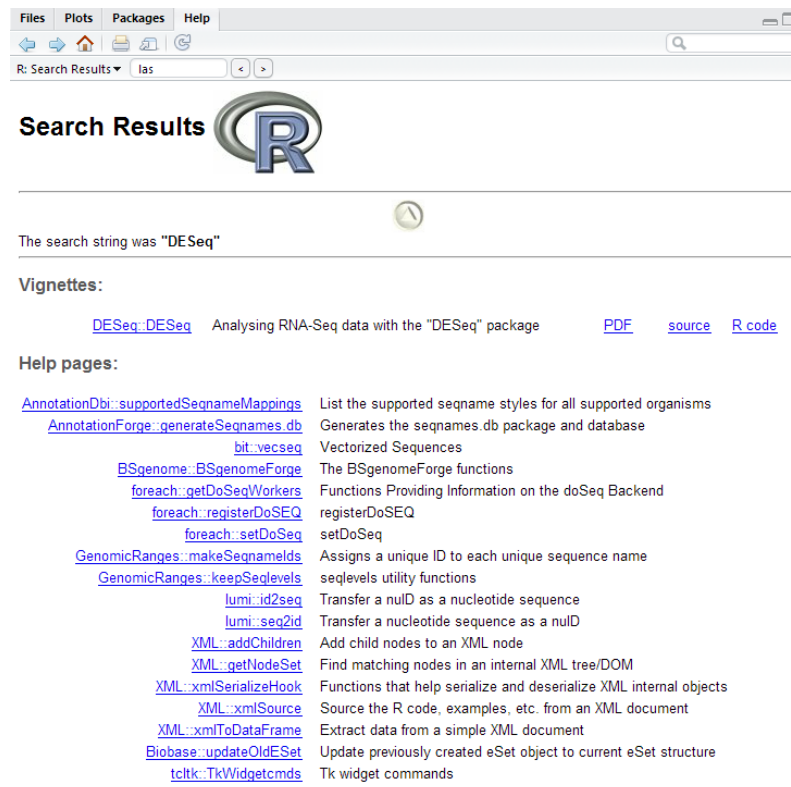
Since packages can define functions of any name then you will get a warning if a package you have loaded includes a function name which is already in use in your session. Sometimes these kinds of clash are intentional, but sometimes they aren't and you should check whether newly loaded libraries might 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
```



The screenshot shows the RStudio interface with the 'Search Results' window open. The search string is 'DESeq'. The results page displays the R logo and the search string. Under the 'Vignettes' section, there is a link to 'DESeq: DESeq' with a description 'Analysing RNA-Seq data with the "DESeq" package' and links for 'PDF', 'source', and 'R code'. The 'Help pages' section lists various functions and their descriptions, including 'AnnotationDbi::supportedSeqnameMappings', 'AnnotationForge::generateSeqnames.db', 'bit::vecseq', 'BSgenome::BSgenomeForge', 'foreach::getDoSeqWorkers', 'foreach::registerDoSEQ', 'foreach::setDoSeq', 'GenomicRanges::makeSeqnames', 'GenomicRanges::keepSeqlevels', 'lumi::id2seq', 'lumi::seq2id', 'XML::addChildNodes', 'XML::getNodeSet', 'XML::xmlSerializeHook', 'XML::xmlSource', 'XML::xmlToDataFrame', 'Biobase::updateOldESet', and 'tcltk::TkWidgetcmds'.

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

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. This 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.

Installing and configuring KnitR

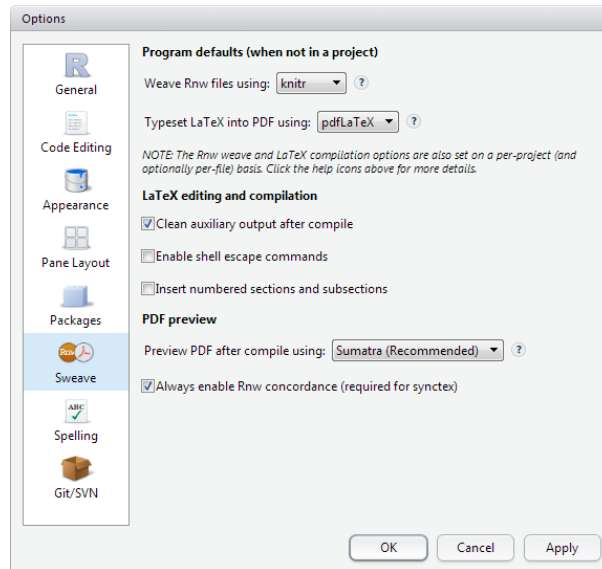
KnitR is a standard R package and as such it can be installed either from the command line, using:

```
install.packages("knitr")
```

...or you can install it from the package manager within R studio by simply searching for KnitR.

Once Knitr is installed you also need to configure RStudio to use it. By default RStudio uses a related package called Sweave which is similar to Knitr but uses a more complex syntax to format its reports.

To configure RStudio to use Knitr simply go to the RStudio preferences (Tools > Options), then select the “Sweave” icon on the left. In the main panel you need to change the option which says “Weave Rnw files using” so that it says “knitr”.



Creating a Knitr markdown file

Knitr works with a couple of different standard formatting languages called LaTeX and Markdown. For this example we’re going to use Markdown as it’s a simpler syntax, but if you’re familiar with LaTeX already you can use that instead.

To create a new Knitr markdown file you simply go to File > New > R Markdown. You should see a new document open up in your text editor which looks like this:

```
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).
```

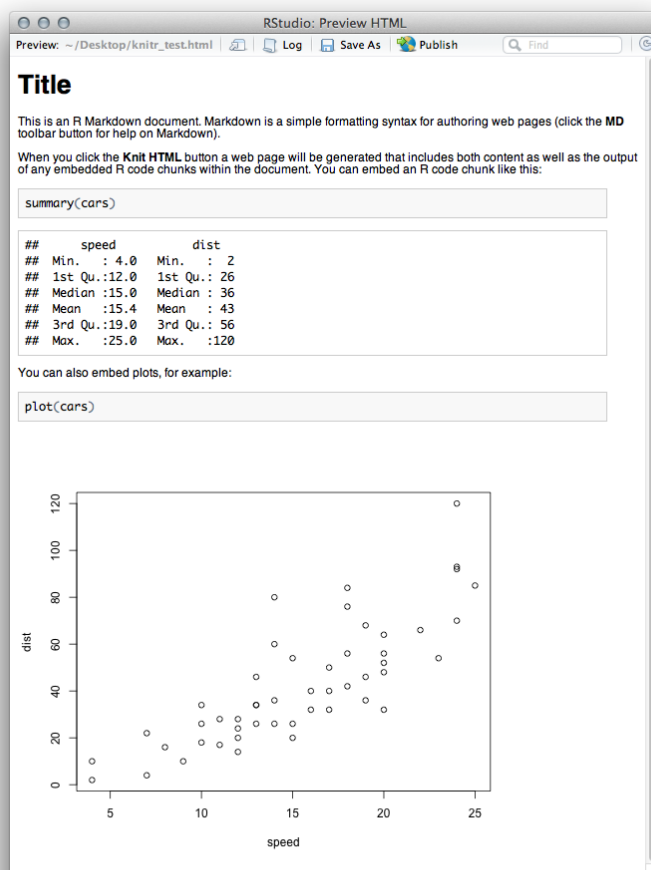
```
When you click the **Knit HTML** button a web page will be generated that
includes both content as well as the output of any embedded R code chunks
within the document. You can embed an R code chunk like this:
```

```
```{r}
summary(cars)
```
```

You can also embed plots, for example:

```
```{r fig.width=7, fig.height=6}
plot(cars)
```
```

This is a simple template file which actually shows most of the code you need to create reports with KnitR. To see what this module can do you can simply press the Knit HTML button at the top of the editor. The program will ask you to save the file. 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 should see a preview of the KnitR report appear in a new window.



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.

Editing Markdown

The Rmd 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

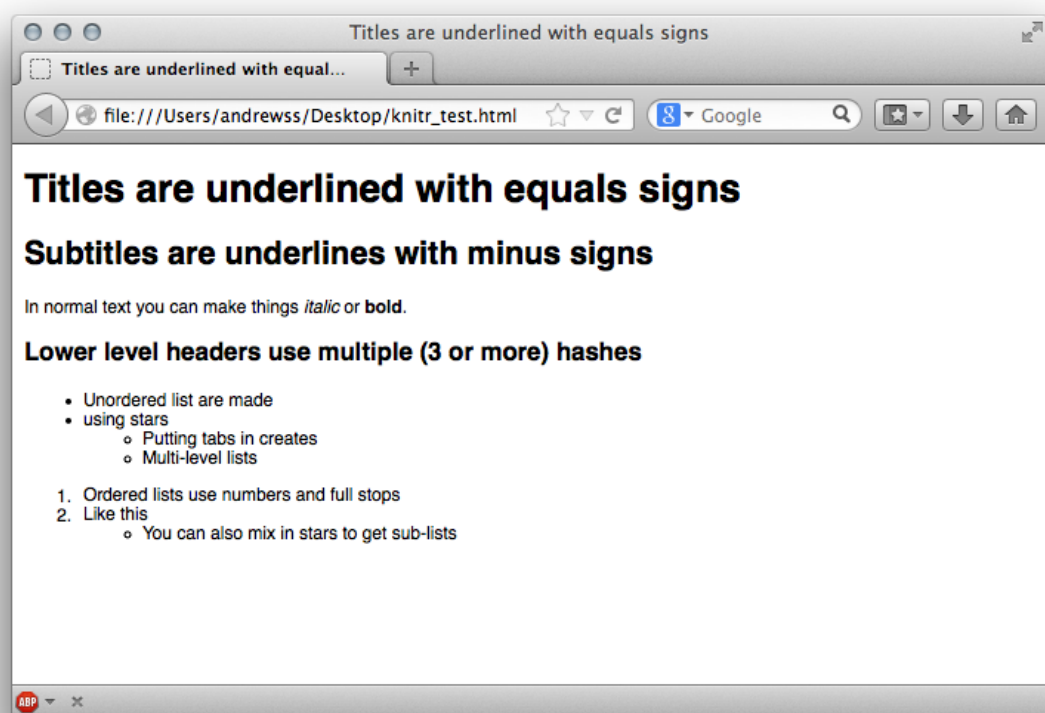
In normal text you can make things *italic* or **bold**.

```
### 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:



Adding R code to a KnitR document

The big advantage to using KnitR to document your R code is that you can mix code and comments together into a single report. R code is incorporated by placing it between placeholders which flag the text as R code. The structure for an R code block looks like this:

```
```{r}
Some R code goes here
```
```

The quotes won't show up in the final report but the R code will, and will be nicely formatted and highlighted. Any output generated by the code will be placed inline in the final report so it will immediately follow the code which generated it.

This is some comment text

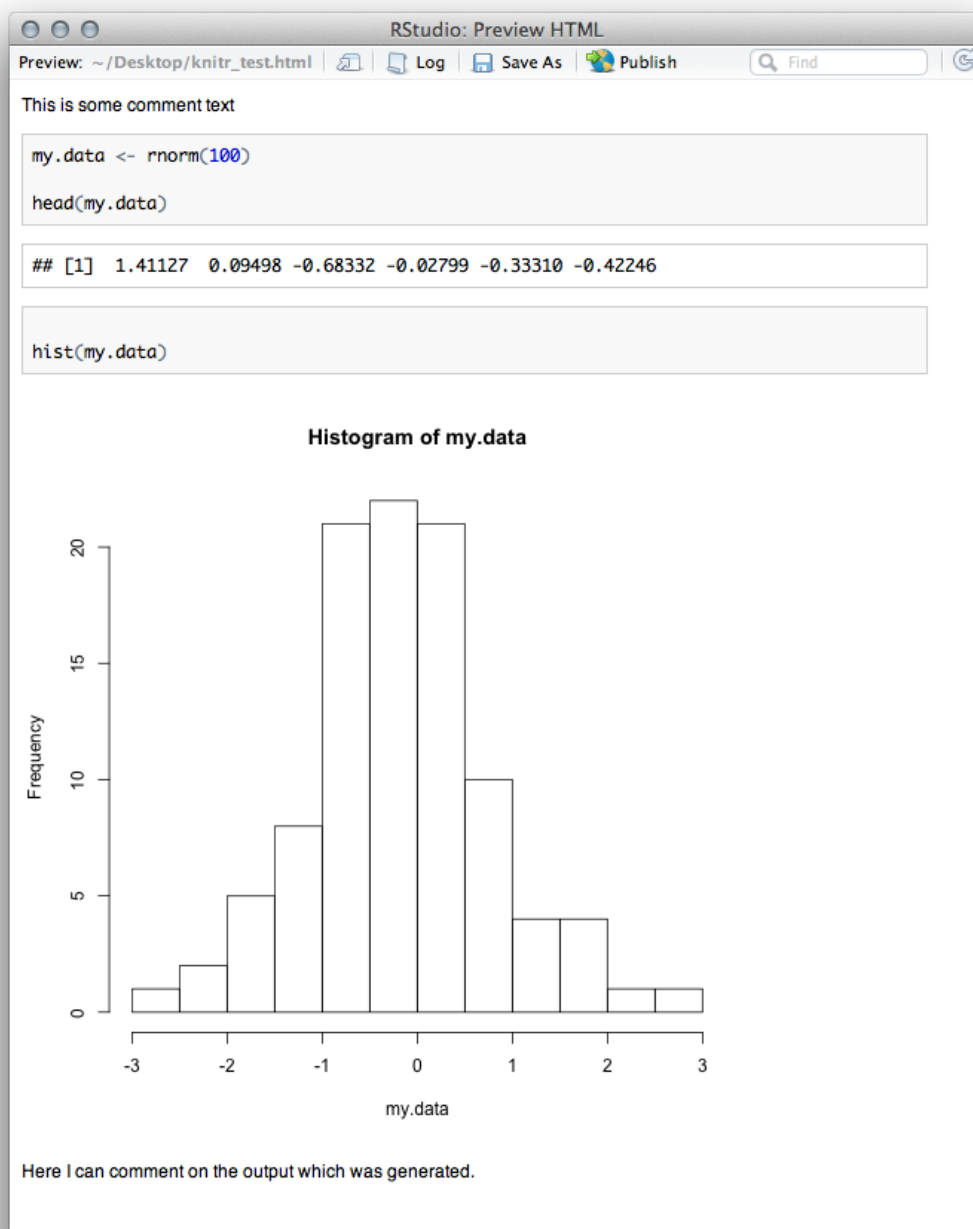
```
```{r}
my.data <- rnorm(100)
```

```
my.data
```

```
hist(my.data)
```

```
```
```

Here I can comment on the output which was generated.



You can see that the R output immediately follows the statement which generated it, so that if you have a block containing several statements this will be split up in the final report and interleaved with the output from each stage.

Whilst this is normally fairly routine for most bits of R code there are some things you will want to change to alter how the R output is reported and change some of the display results.

The two most common things you will need to change are the types of output which are shown, and the size of graphical figures.

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")
```
```



```
RStudio: Preview HTML
Preview: ~/Desktop/knitr_test.html | Log | Save As | Publish | Find

library("DESeq")

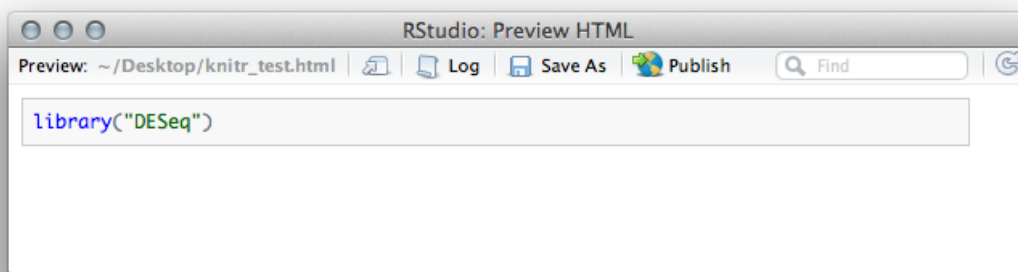
## Loading required package: BiocGenerics
## Loading required package: parallel
## Attaching package: 'BiocGenerics'
## The following objects are masked from 'package:parallel':
##   clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##   clusterExport, clusterMap, parApply, parCapply, parLapply,
##   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:

| | |
|---------------------------------|-----------------------|
| <code>message=TRUE/FALSE</code> | Show messages |
| <code>warning=TRUE/FALSE</code> | Show warning messages |
| <code>error=TRUE/FALSE</code> | 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")
```
```

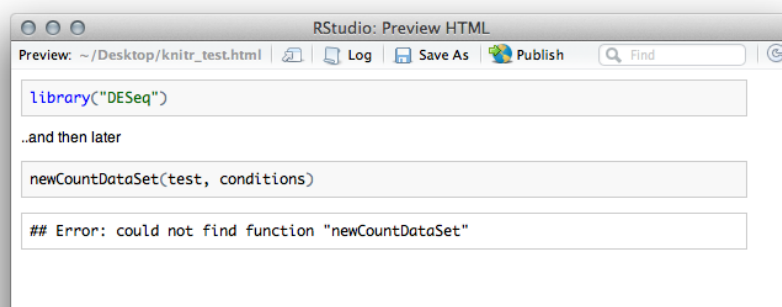


If you want to go one step further you can also set `eval=FALSE` which will not even run the R code in this block but it will still show up in the report.

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

..and then later

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



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")
```
```

