R Ramblings

Jeff Laake <jeff.laake@noaa.gov>

January 12, 2010

Contents

1 Introduction **1**

2 Some Initial Thoughts **2**

3 Be Interactive with R **3**

4 R Objects **5**

5 Data Structures **8**
5.1 Vectors 8
5.2 Matrices and arrays 14
5.3 Dataframes 22
5.4 Lists 30

6 Scripts and Functions **32**

7 Loops in R **37**
7.1 Apply 38
7.2 Lapply and Sapply 42
7.3 Tapply 47
7.4 Package plyr 50

1 Introduction

Beware! R is addicting and I'll freely admit my addiction.

> "Using R is a bit akin to smoking. The beginning is difficult, one may get headaches and even gag the first few times. But in the long run,it becomes pleasurable and even addictive. Yet, deep down, for those willing to be honest, there is something not fully healthy in it." –Francois Pinard

Healthy or not, R is an amazing work environment for the data analyst. There are volumes of free documentation about R that you can find on the internet. Or if you prefer there are numerous books. Look through the R home page (`http://www.r-project.org/`) for useful links. One of the most useful books that I have encountered is Data Manipulation with R by Phil Spector. It is part of the Use-R series published by Springer(`http://www.springer.com/statistics`). In this document I provide some rambling thoughts about various aspects of R that I think are important. It is not intended to be a complete R tutorial but hopefully it will help you on your journey to addiction. If not, at least this has been a useful learning experience for me to discover the benefits of Sweave which I have used with LyX to generate this document (`http://www.lyx.org/`).

2 Some Initial Thoughts

Bear with me as I try to brainwash you with some of my personal philosophy about data analysis and programming. R was designed as an interactive environment but is unlike your typical computer application with a graphical user interface(gui). With the exceptions of some point and click options with Rgui and gui interfaces in development (`http://www.sciviews.org/_rgui/`), to use R you need to know and use the R language. Being an old FORTRAN programmer that learned programming in the days of card punches and paper tape machines, this does not disappoint me. I'm quite use to reading reference manuals and becoming proficient with a language to accomplish tasks. However, for many, learning the language is a daunting task that may cause real headaches or the metaphorical gagging mentioned in the above quote. Don't worry. Just like a good hangover, the pain will pass even if your agony at the time seems perpetual. I believe that gui applications have many benefits for tasks such as word processing and the like. However, with data analysis you need a computing environment that provides a scripting capability. With scripting (programming) you can document your analysis, ensure reproducibility in your research and easily modify, replicate or simulate your analysis. While point and click interfaces are typically easier to learn, they soon become a drudgery with repetitive tasks and any error in the final result may not be traceable after a multitude of steps. Once upon a time (pre-R), I was a big user of Excel for manipulating data. R cured me of that addiction and provided me with a much less detrimental addiction for data manipulation and analysis. Patrick Burns in a piece titled Spreadsheet Addiction (`http://www.burns-stat.com/`) sums it up quite well:

> The goal of computing is not to get an answer, but to get the correct answer. Often a wrong answer is much worse than no answer at all. There are a number of features of spreadsheets that present a challenge to error-free computing.

I believe his quote could easily be expanded to many other gui applications used for data analysis and computing. While a scripted analysis can easily contain flaws, they are traceable and correctable. Even though applications such as Excel may have a macro or programming capability, they are not integral to the product and it leaves you feeling like you are pounding a square peg into a round hole. That is not the case with R. There are other options for data analysis like MATLAB, SAS or others. I'm not going to use any words

here expounding the advantages and disadvantages of R as you can find reams of that type of discussion on the internet. Personally, I'm sold on R, so let's get on with the possibly useful ramblings.

3 Be Interactive with R

If you own a calculator, do yourself a favor and throw it in the trash or give it to a student to use in grade school. While you are at it, you can also remove any of those calculator apps from your computer. R is your new calculator. R is interactive and that is how you should use it. You can get the printed result for any calculation by typing the equation and return as shown below:

```
> 1 + 1

[1] 2

> sqrt(2)

[1] 1.414214

> exp(1)

[1] 2.718282
```

The result of each calculation is a vector containing a single number. What you see is a printed display of the contents of the vector which is why you see a [1] before the value. After the result is computed, R uses a default print function to display the contents of the vector with a default number of digits displayed. This can be made explicit using the print function:

```
> print(1 + 1)

[1] 2

> print(sqrt(2))

[1] 1.414214

> print(sqrt(2), digits = 10)

[1] 1.414213562

> print(exp(1))

[1] 2.718282

> print(exp(1), digits = 12)
```

```
[1] 2.71828182846
```

While all of this may seem a bit pedantic, it is common to see queries on R-help about why R truncated the precision of numbers. Here is one that just arrived 8 Jan 2010:

> Hi all,
> How can I get R to change the default precision value? For example:
> > x=0.99999999999999999
> > 1-x
> [1] 0
> Is there a way that I can get a non-zero value using some parameter, or some package?
> many thanks.

What many people don't understand is that when you perform a calculation or type the name of an object, a print function specific to the type of object is executed to display the contents of the object. For numeric vectors, the default print function has a default number of digits that are displayed. The default number of digits can be set with the options function (e.g., `options(digits=n)`). For other types of objects, the print function for that object may only print portions of the object or show the contents in a summary fashion (e.g., `lm`). Always remember that the contents of an object are not necessarily the same as what is displayed by the print function for that object. Now in the case of this email, the writer has specified a numeric value with greater precision than can be represented with double precision (at least with a 32 bit machine).

```
> # 11 digits
> x=0.99999999999
> print(1-x,digits=20)

[1] 1.000000082740371e-11

> # 15 digits
> x=0.999999999999999
> print(1-x,digits=20)

[1] 9.99200722162641e-16

> # 16 digits
> x=0.9999999999999999
> print(1-x,digits=20)

[1] 1.110223024625157e-16

> # 17 digits
> x=0.99999999999999999
> print(1-x,digits=20)

[1] 0
```

Precision does have its limits! Although there are some packages available that can precision limits.

4 R Objects

While it is useful to have a powerful calculator, data analysis typically involves more than a single calculation and we will want to save the results of calculations by assigning the results to a named object. The assignment operator in R is <- (e.g., x<-1, 1 is assigned to x or x gets 1); however, you can also use = (e.g., x=1). There are a few places where they are not interchangeable but I have never run into a situation yet where I have been unable to use =. Some choose to use <- to avoid confusion with the use argument=value in function calls or the boolean equality operator ==. You can use either.

Object names in R can include letters, numbers, an underscore (_) or period (.) but they can only begin with a letter or period(.). R is case-sensitive. The names x and X are not the same. If you get an error about an object "not found", the most common errors are due to mis-spelling and incorrect case.

```
> x = 1/2
> X

Error: object 'X' not found
```

Everything in R is an object and there are different kinds of objects in R. Each object has a number of attributes that describe the contents of the object and how the object is used and treated. The mode and class are two of the most important attributes. You can use the mode, class and str functions to examine object attributes. You may see the word "atomic" in an error message and you might wonder why your code "nuked" (ok bad pun) and what atomic is. Atomic structures in R are the simplest structures in which all of the elements are of the same mode. The primary modes for atomic data are numeric, logical and character. You'll probably never encounter the lesser used modes of complex and raw. Logical mode has values TRUE and FALSE. Do not use T and F even though these will work because they are not reserved and you can alias those values. Character mode is specified with quotation marks. Below are examples of the primary data modes and calls to attribute functions:

```
> mode(TRUE)

[1] "logical"

> class(TRUE)

[1] "logical"

> mode("Laake")

[1] "character"

> class("Laake")

[1] "character"
```

```
> mode(1)

[1] "numeric"

> class(1)

[1] "numeric"

> mode(as.integer(1))

[1] "numeric"

> class(as.integer(1))

[1] "integer"
```

For these basic objects the class and mode are the same with the exception of integer. Numeric mode is general and includes both double precision (the default) and integer. If you specify the number 1 as above, it is stored in double precision unless you specify the mode to be integer using as.integer or setting the mode with the mode function.

So far we have only shown vectors with single elements. To specify more than one element in the vector, use the concatenate (i.e., put together) function which is c(...) where ... means an unspecified number of arguments separated by commas. All of the elements in the vector most have the same mode or the mode of non-conforming elements will be coerced to a common mode. In the following example, if we mix numeric and character, the numeric values are converted to character:

```
> c("B", 1, "C")

[1] "B" "1" "C"
```

with as.numeric:

```
> as.numeric(c("2", 1, "3"))

[1] 2 1 3
```

In addition to having a mode and class, vectors also have a length attribute which can be retrieved with the length function.

```
> length(c("2", 1, "3"))

[1] 3
```

Next I'll demonstrate how to create a factor object by creating a character vector and then using the factor function. Factor objects are of numeric mode but with a class attribute that gives it special treatment in which labels (character strings) are displayed even though the storage mode is numeric:

```
> my.factor = c("B", "A", "C")
> my.factor = factor(my.factor)
> mode(my.factor)

[1] "numeric"

> class(my.factor)

[1] "factor"

> my.factor

[1] B A C
Levels: A B C
```

What is stored in my.factor is a numeric vector c(2,1,3) because the levels are alphabetical by default so "B" is the second, "A" is the first, and "C" is the third. We can see that by using the as.numeric function:

```
> as.numeric(my.factor)

[1] 2 1 3
```

You can modify the order of the levels in a factor using the levels argument or with the relevel function:

```
> my.factor = c("B", "A", "C")
> my.factor = factor(my.factor, levels = c("C", "B", "A"))
> my.factor

[1] B A C
Levels: C B A

> as.numeric(my.factor)

[1] 2 3 1

> my.factor = relevel(my.factor, "B")
> my.factor

[1] B A C
Levels: B C A

> as.numeric(my.factor)

[1] 1 3 2
```

It is important to understand factors because they are used in just about every aspect of statistical modelling and data manipulation. I'll cover more on factors later. Now I'll move onto a more general discussion of data structures and subscripting (subsetting) in R.

5 Data Structures

The data structures in R are vectors, matrices and arrays which are generalization of vectors, dataframes and lists, which is the most general and flexible data structure. I'll describe each of these structures and how to subscript which means extract a subset of the object.

5.1 Vectors

I've already introduced the concept of a vector but next I'll describe ways of creating useful vectors quickly with functions that create sequences as these will be useful for subscripting and in a multitude of other ways. The functions we'll use, from simplest to most complex, are the colon(:) sequence operator (effectively a function) and the sequence and repeat functions. For numeric arguments, a:b generates a regular sequence of +1/-1 from a to b. If a and b are integers the sequence is of integers; otherwise they are of type double.

```
> 1:8

[1] 1 2 3 4 5 6 7 8

> 8:1

[1] 8 7 6 5 4 3 2 1

> -5:5

 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5

> 5:-10

 [1]   5   4   3   2   1   0  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

If the colon operator is used with factors it will generate sequences of combinations similar to the interaction function:

```
> num.fac = factor(1:3)
> alpha.fac = factor(c("A", "B", "C"))
> num.fac:alpha.fac

[1] 1:A 2:B 3:C
Levels: 1:A 1:B 1:C 2:A 2:B 2:C 3:A 3:B 3:C

> alpha.fac:num.fac

[1] A:1 B:2 C:3
Levels: A:1 A:2 A:3 B:1 B:2 B:3 C:1 C:2 C:3

> interaction(alpha.fac, num.fac)
```

```
[1] A.1 B.2 C.3
Levels: A.1 B.1 C.1 A.2 B.2 C.2 A.3 B.3 C.3
```

The colon operator is a short-hand for the more general seq function which can handle numeric, dates and times and the sequences can increment by values other than +1/-1. Here are some illustrations of sequences that can be generated:

```
> seq(from = 1, to = 9, by = 2)

[1] 1 3 5 7 9

> seq(3, 15, 3)

[1]  3  6  9 12 15

> seq(4, -4, -2)

[1]  4  2  0 -2 -4

> seq(from = as.Date("2010-01-01"), to = as.Date("2010-01-30"),
+     by = 5)

[1] "2010-01-01" "2010-01-06" "2010-01-11" "2010-01-16" "2010-01-21"
[6] "2010-01-26"
```

The more general function rep can be used to generate repeated sequences of vectors of any mode and also date, times and factors. The arguments each and times act differently to provide alternative sequences. For example,

```
> rep(1:5, each = 5)

 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5

> rep(c("A", "B", "C"), times = 5)

 [1] "A" "B" "C" "A" "B" "C" "A" "B" "C" "A" "B" "C" "A" "B" "C"

> rep(my.factor, 3)

[1] B A C B A C B A C
Levels: B C A
```

seq, or : return a vector, the functions can be nested as shown above with : and with rep and seq below:

```
> rep(seq(5,9,2),each=5)

 [1] 5 5 5 5 5 7 7 7 7 7 9 9 9 9 9
```

```
> rep(rep(c(1,2),3),each=5)

 [1] 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2 2 2 2 2

> # or alternatively
> rep(c(rep(1,5),rep(2,5)),3)

 [1] 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2 2 2 2 2
```

Although I've not found them very useful, names can be attached to elements of vectors. I introduce them here before launching into subscripts because they will be very useful for subscripts and subsetting, particularly for more complicated data structures. Names can be introduced when the vector is created or using the names function as shown below to set the attributes:

```
> my.vector = c(x = 1, y = 2)
> my.vector

x y
1 2

> str(my.vector)

 Named num [1:2] 1 2
 - attr(*, "names")= chr [1:2] "x" "y"

> names(my.vector) = c("A", "B")
> my.vector

A B
1 2
```

Notice the use of quotes for characters with names but quotes are not needed when assigning the names with the concatenate function. The same will be true with the other data structures.

Now let's consider how to extract a subset of a vector, also known as subscripting. Subscripts can be specified using numeric or logical subscripts or names and the syntax uses single square braces. Numeric subscripting of a vector can be inclusive (which to include) or exclusive (which to exclude). Consider the sequence from 5 to 100 by 5's and wanting to extract the 6th element in the sequence:

```
> my.sequence = seq(5, 100, 5)
> my.sequence[6]

[1] 30
```

Now let's say you wanted everything but the 6th element. Then you use the negative subscript to exclude:

```
> my.sequence[-6]

 [1]   5  10  15  20  25  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

You can extract a subset of more than a single element by providing a numeric vector for the subscripts, but you cannot mix positive and negative subscripts:

```
> my.sequence[c(1, 6, 18)]

[1]  5 30 90

> my.sequence[-(5:8)]

 [1]   5  10  15  20  45  50  55  60  65  70  75  80  85  90  95 100
```

While it may look a little strange, consider the following which has nesting of subscripts shown with and without parentheses to be explicit:

```
> my.sequence[c(1, 6, 18)][2]

[1] 30

> (my.sequence[c(1, 6, 18)])[2]

[1] 30
```

The above selects the second element of the subset containing the first, sixth and 18th elements of the original sequence. While this is not particularly useful here, this nesting is useful with more complicated data structures.

Logical subscripting is a powerful form of subsetting that you'll want to use all the time. Before I introduce, logical subscripts we need to consider logical operations and operators. Many of the common operators like <, >, <=, >= (less than, greater than, less than or equal and greater than or equal) will probably be familiar. Equality uses a double = (==) and the exclusion of equality uses the not operator (!=). A logical operation includes comparison of two objects and an operator and the result is a logical vector (vector of TRUE or FALSE values). Below are some examples:

```
> my.sequence < 25

 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> my.sequence >= 25

 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

> my.sequence == 25
```

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

```
> my.sequence != 25
```

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

Logical operations can also be done with characters, dates, times etc. For example,

```
> c("A", "B", "C") < "B"
```

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

```
> c("A", "B", "C") == "C"
```

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

Factor variables can also be used in equality comparisons but can only be used in order comparisons if the factor is ordered. Attempting to do so with an unordered factor will yield an error:

```
> # The following is okay
> factor(c("A","B","C"))=="B"
```

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

```
> # But the following would give an error
> # factor(c("A","B","C"))<"B"
> # But will work now because the factor is ordered
> factor(c("A","B","C"),ordered=TRUE)<="B"
```

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

Logical operations can also use logical values with the most common operators being & (and), | (or) and ! (not) :

```
> less.than.75 = my.sequence < 75
> greater.than.25 = my.sequence > 25
> less.than.75 & greater.than.25
```

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

```
> !less.than.75 | !greater.than.25
```

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

Subscripting with logicals is as simple as providing a vector with a value of TRUE (include) or FALSE (exclude) for each element in the vector. Using the above example, we can extract the elements of my.sequence with:

```
> my.sequence[less.than.75 & greater.than.25]

[1] 30 35 40 45 50 55 60 65 70

> my.sequence[!less.than.75 | !greater.than.25]

 [1]   5  10  15  20  25  75  80  85  90  95 100
```

The final form of subscripting a vector specifies a set of names of the vector to subset. The names can contain spaces but it is not a wise choice because it will limit your options for subscripting with dataframes and lists, as you'll see later.

```
> my.vector = c(x = 1, y = 14, z = -32)
> my.vector[c("x", "z")]

  x   z
  1 -32

> names(my.vector)[2] = "my y"
> my.vector[c("my y", "x")]

my y    x
  14    1
```

Notice that the subscripts can re-order the values and with numeric subscripts you can specify multiple copies of elements:

```
> # Notice the use of a vector for the times argument
> my.vector[rep(1:3,times=3:1)]

   x    x    x my y my y    z
   1    1    1   14   14  -32
```

One final note before leaving vectors. Without discussing it, we have been using the notion of recycling which means a vector of smaller length will be repeated (recycled) as often as needed to make it equivalent in length to the longer vector. For example, with my.sequence<25 , the vector (number) 25 is recycled twenty times to create a vector of length 20 all with the value 25 to compare to the vector my.sequence which has length 20. Where this gets more interesting is when the smaller vector is not a scalar (length 1). In that case, when the smaller vector is recycled, it must be an even multiple of the longer vector or a warning is issued to warn you that you may not be doing what you want:

```
> my.sequence[my.sequence<c(25,65)]
```

```
[1]  5 10 15 20 30 40 50 60

> # Following works but gives a warning message
> #my.sequence[my.sequence<c(25,65,35)]
> #[1]  5 10 15 20 25 30 40 55
> #Warning message: In my.sequence < c(25, 65, 35) :
> #  longer object length is not a multiple of shorter object length
> # It is equivalent to my.sequence[my.sequence<c(rep(c(25,65,35),7)]
> # which will issue the same warning. But the following will work
> # without a warning because both vectors are of length 20
> my.sequence[my.sequence<c(rep(c(25,65,35),6),25,65)]

[1]  5 10 15 20 25 30 40 55
```

With the exception of recycling a scalar, you are unlikely to use it intentionally with vectors. However, it can be quite useful in manipulating vectors with matrices.

5.2 Matrices and arrays

A matrix is a 2 dimensional collection of vectors, viewed as either row vectors or column vectors, in which all vectors are of the same mode and length. Matrices are typically used for numerical variables although logical matrices will be useful for subscripting. I've not yet encountered a need for character matrices. A matrix has 2 dimensions: the number of rows and the number of columns. A matrix can be constructed easily with the `matrix` function or `diag` can be used to create a diagonal matrix or modify the diagonal values:

```
> # create a matrix with all values being 1
> my.matrix=matrix(1,nrow=3,ncol=2)
> my.matrix

     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    1    1

> # you can use dim to get the row, column dimensions
> dim(my.matrix)

[1] 3 2

> # or use nrow and ncol
> nrow(my.matrix)

[1] 3

> ncol(my.matrix)
```

```
[1] 2

> # you can modify the dimensions as long as the overall size is the same
> dim(my.matrix)=c(2,3)
> my.matrix

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1

> dim(my.matrix)=c(1,6)
> my.matrix

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    1    1    1    1

> # use diag to create an identity matrix
> my.matrix=diag(1,nrow=3,ncol=3)
> my.matrix

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1

> diag(my.matrix)=1:3
> my.matrix

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

Matrices can also be created with the row-binding function (rbind) and the column binding function (cbind) where the vectors being bound into a matrix are either of the same length or the vector can be recycled to the same length:

```
> cbind(1:3, 4:6, 7:9)

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> rbind(1:3, 4:6, 7:9)
```

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

```
> rbind(1:3, 4, 5)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    4    4
[3,]    5    5    5
```

To illustrate another argument of the matrix function, I'll show how the first and second examples could also be constructed:

```
> matrix(1:9, nrow = 3)
```

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

```
> matrix(1:9, nrow = 3, byrow = TRUE)
```

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

Essential linear algebra operations for matrices are available including: matrix/vector multiplication (`%*%`), transpose with `t(x)`and inverse with `solve(x)`, etc. However, here I'll focus on subscripting and traditional arithmetic operators, because you'll more likely use those with matrices and there are parallel operations with dataframes.

Subscripting matrices is similar to vectors with the added notion of a null dimension. If we want the element in the ith row and jth column in a matrix `x`, use `x[i,j]` and the result is a vector of length 1. However, if we want the entire ith row or the jth column, use `x[i,]`and `x[,j]` and leave the other dimension as unspecified (null). Here the result is a vector of length `ncol(x)` and `nrow(x)` respectively. If we want the result to be a matrix, add `drop=FALSE` as in `x[i, ,drop=FALSE]`, `x[,j,drop=FALSE]`or `x[i,j,drop=FALSE]` to get a 1 by `ncol(x)`, `nrow(x)` by 1, and 1 by 1 matrix respectively. Note the need to maintain the commas so the arguments are in their proper positions:

```
> my.matrix[2, 3]
```

```
[1] 0
```

```
> my.matrix[2, ]
```

```
[1] 0 2 0
```

```
> my.matrix[2, , drop = FALSE]
```

```
     [,1] [,2] [,3]
[1,]    0    2    0
```

```
> my.matrix[, 3]
```

```
[1] 0 0 3
```

```
> my.matrix[, 3, drop = FALSE]
```

```
     [,1]
[1,]    0
[2,]    0
[3,]    3
```

Logical subscripting works similar to vectors except that the result of a logical comparison with a matrix is a matrix of logical values which can then be used to extract the values with the result being a vector. For example,

```
> my.matrix > 0
```

```
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE FALSE
[2,] FALSE  TRUE FALSE
[3,] FALSE FALSE  TRUE
```

```
> my.matrix[my.matrix > 0]
```

```
[1] 1 2 3
```

This can be useful to assign new values at the locations in the matrix which meet the logical condition:

```
> my.matrix[my.matrix > 0] = -1
> my.matrix
```

```
     [,1] [,2] [,3]
[1,]   -1    0    0
[2,]    0   -1    0
[3,]    0    0   -1
```

```
> my.matrix[my.matrix == -1] = 3:1
> my.matrix
```

```
     [,1] [,2] [,3]
[1,]    3    0    0
[2,]    0    2    0
[3,]    0    0    1
```

Matrices have the additional form of subscripting in which you can construct a numerical subscript matrix with 2 columns which specify row-column pairs. For example, if I wanted to extract the following i,jth elements: (1,3), (2,1), (3,3) I could do that as follows:

```
> my.matrix = matrix(1:9, nrow = 3, ncol = 3)
> indices = cbind(1:3, c(3, 1, 3))
> indices
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    1
[3,]    3    3
```

```
> my.matrix[indices]
```

```
[1] 7 2 9
```

Arithmetic operations with matrices and vectors are quite simple as long as you understand recycling and how matrices are constructed. An old FORTRAN programmer might do the following to add 10 to a 3 by 3 matrix which as an inefficient equivalent in R:

```
> # do 2 i=1,3
> #   do 1 j=1,3
> #     x(i,j)=x(i,j)+10
> #   1 continue
> # 2 continue
> x=diag(1:3,nrow=3,ncol=3)
> for(i in 1:nrow(x))
+ for(j in 1:ncol(x))
+  x[i,j]=x[i,j]+10
```

However, there is no need to approach matrix and vector operations in such a crude manner as the following will work much more efficiently:

```
> # addition of a scalar
> x=diag(1:3,nrow=3,ncol=3)
> x=x+10
> x
```

```
     [,1] [,2] [,3]
[1,]   11   10   10
[2,]   10   12   10
[3,]   10   10   13
```

```
> # or multiplication by a scalar
> x=x*1.5
> x

     [,1] [,2] [,3]
[1,] 16.5   15 15.0
[2,] 15.0   18 15.0
[3,] 15.0   15 19.5

> # or addition of 2 matrices of the same size
> x+x

     [,1] [,2] [,3]
[1,]   33   30   30
[2,]   30   36   30
[3,]   30   30   39

> # or elementwise multiplication of 2 matrices of the same size
> x*x

       [,1] [,2]   [,3]
[1,] 272.25  225 225.00
[2,] 225.00  324 225.00
[3,] 225.00  225 380.25
```

When elementwise arithmetic operations are conducted with a vector and matrix, you need to understand that a matrix is actually treated as a vector in which the columns are stacked one after the other. This is consistent with the manner in which matrices are created from vectors by default (`byrow=FALSE`) where they are entered into the matrix down the columns. Thus when a vector is arithmetically combined elementwise, the vector is recycled to match the total size of the matrix (# rows times #columns) and the elements are matched down the columns. If the vector length is not a multiple of the matrix size, a warning will be issued. The following shows some examples with addition and multiplication:

```
> # addition of a vector and matrix
> xmat=matrix(1:6,nrow=2,ncol=3)
> xvec=1:3
> xmat+xvec

     [,1] [,2] [,3]
[1,]    2    6    7
[2,]    4    5    9

> # multiplication
> xmat*xvec
```

```
     [,1] [,2] [,3]
[1,]    1    9   10
[2,]    4    4   18
```

The elementwise order of the operation can be modified by to row-order by using the transpose function which switches the rows and columns of the matrix:

```
> # addition of a vector and matrix
> t(t(xmat)+xvec)
```

```
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

```
> # multiplication
> t(t(xmat)*xvec)
```

```
     [,1] [,2] [,3]
[1,]    1    6   15
[2,]    2    8   18
```

The inner transpose forces the row-order element-wise computations and the outer transpose then switches the orientation back.

Arrays provide extensions to allow for 3 or more dimensions but they like matrices are simply vectors with a dimension (`dim`). For example, we can create a 3-d array as follows:

```
> my.array=array(1:(3*2*3),dim=c(3,2,3))
> my.array
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2

     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12

, , 3

     [,1] [,2]
[1,]   13   16
[2,]   14   17
[3,]   15   18
```

Just like with matrices, the first index changes first, then the second index,...,final index changes last as apparent in the ordering of the sequential vector. As with matrices names can be assigned to the dimensions:

```
> dimnames(my.array)=list(paste("x",1:3,sep=""),paste("y",1:2,sep=""),
+                         paste("z",1:3,sep=""))
> my.array

, , z1

   y1 y2
x1  1  4
x2  2  5
x3  3  6

, , z2

   y1 y2
x1  7 10
x2  8 11
x3  9 12

, , z3

   y1 y2
x1 13 16
x2 14 17
x3 15 18
```

You can use numeric, logical and name subscripts with arrays as done with matrices:

```
> my.array[1,2,2:3]

z2 z3
10 16

> my.array["x2","y1","z3"]

[1] 14

> my.array[my.array<5]=-1
> my.array

, , z1

   y1 y2
```

```
x1 -1 -1
x2 -1  5
x3 -1  6

, , z2

   y1 y2
x1  7 10
x2  8 11
x3  9 12

, , z3

   y1 y2
x1 13 16
x2 14 17
x3 15 18

> my.array[cbind(1:2,1:2,2:3)]

[1]  7 17
```

5.3 Dataframes

A dataframe is the typical structure used to store data for analysis. It is similar to a matrix in that it is composed of column vectors which all have the same length (the number of rows in the dataframe). However, the primary difference is that column vectors do not all have to be the same mode. One column can be character, another numeric and another a factor. Dataframes can be treated and converted to a matrix but the conversion coerces all of the vectors to a single common mode. I'll not discuss entering data into a dataframe and instead will focus on subscripting and manipulating dataframes. I'll use the iris data that accompanies R.

First let's look at the mode, class and structure of the iris dataframe.

```
> data(iris)
> mode(iris)

[1] "list"

> class(iris)

[1] "data.frame"

> str(iris)
```

```
'data.frame':        150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

We put the iris dataframe into our workspace with the `data` function. The mode of iris is a `list` structure but I'll defer discussion of lists until later. The `class` is a `data.frame` and with the `str` function, we see that it has 150 rows (number of observations) and it has 5 columns (variables). We also see the first 4 variables (columns) are are numeric mode and their names are Sepal.Length, Sepal.Width, Petal.Length, and Petal.Width. The final column is a `factor` variable with 3 levels. If you only wanted to see the names of the variables, you could use `names(iris)`. Another useful function with dataframes is the `summary` function. Prior to using it, without explanation, I'll add a character variable which is the character label of the species:

```
> iris$Species.Label = as.character(iris$Species)
> summary(iris)

  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species   Species.Label
 setosa    :50   Length:150
 versicolor:50   Class :character
 virginica :50   Mode  :character
```

We now have a summary of each variable in the dataframe. For numeric variables, we get a range, quartiles, and mean. Note that each of these printed values have limited precision for pretty printing so the true min and max could be slightly different. For factor variables, it prints a table of counts of each factor level and for character variables, all we get is a length. We can construct a tabular count of a character variable (or for any variable) using the `table` function:

```
> table(iris$Species.Label)

    setosa versicolor  virginica
        50         50         50

> table(iris$Species)
```

```
    setosa versicolor  virginica
        50         50         50

> table(as.numeric(iris$Species))

 1  2  3
50 50 50
```

The first two look identical in this case even though the variables are of different modes because the levels of the factor variable are printed. The third makes it clear that the underlying mode of `Species` is numeric.

Without any explanation, I have been using a standard form of data subscripting using the $ to extract a single variable from the dataframe. As with matrices and vectors, dataframes can also be subscripted with numeric, names and logicals. Even though column modes can vary, a dataframe is rectangular like a matrix with rows and columns. Thus, numerical subscripting uses the exact same structure. We can select subsets of rows, columns or rows and columns. To demonstrate without taking up a lot of space, I'll use the `head` function which shows the first `n` values with `n=6` as the default. A `tail` function works similarly to show the last `n` values.

```
> # Show data records 3 through 5
> iris[3:5,]

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species Species.Label
3          4.7         3.2          1.3         0.2  setosa        setosa
4          4.6         3.1          1.5         0.2  setosa        setosa
5          5.0         3.6          1.4         0.2  setosa        setosa

> # Show columns 3 through 5
> head(iris[,3:5])

  Petal.Length Petal.Width Species
1          1.4         0.2  setosa
2          1.4         0.2  setosa
3          1.3         0.2  setosa
4          1.5         0.2  setosa
5          1.4         0.2  setosa
6          1.7         0.4  setosa

> # Show rows 1 through 3 of columns 3 through 5
> iris[1:3,3:5]

  Petal.Length Petal.Width Species
1          1.4         0.2  setosa
2          1.4         0.2  setosa
3          1.3         0.2  setosa
```

Dataframes can also be subscripted using names of either rows or columns. For the iris data the row names are character representations of the row number so they are not particularly informative but I'll show how both row names and column names can be used to replicate the above example:

```
> # Show data records 3 through 5
> iris[as.character(3:5),]
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species Species.Label
3          4.7         3.2          1.3         0.2  setosa        setosa
4          4.6         3.1          1.5         0.2  setosa        setosa
5          5.0         3.6          1.4         0.2  setosa        setosa
```

```
> # Show columns 3 through 5
> head(iris[,c("Petal.Length", "Petal.Width", "Species")])
```

```
  Petal.Length Petal.Width Species
1          1.4         0.2  setosa
2          1.4         0.2  setosa
3          1.3         0.2  setosa
4          1.5         0.2  setosa
5          1.4         0.2  setosa
6          1.7         0.4  setosa
```

```
> # or get the third to fifth names of iris
> head(iris[,names(iris)[3:5]])
```

```
  Petal.Length Petal.Width Species
1          1.4         0.2  setosa
2          1.4         0.2  setosa
3          1.3         0.2  setosa
4          1.5         0.2  setosa
5          1.4         0.2  setosa
6          1.7         0.4  setosa
```

```
> # Show rows 1 through 3 of columns 3 through 5
> iris[as.character(3:5),names(iris)[3:5]]
```

```
  Petal.Length Petal.Width Species
3          1.3         0.2  setosa
4          1.5         0.2  setosa
5          1.4         0.2  setosa
```

Okay, so that may have been a bit much. I'm sure you are wondering why you need to know so many forms for subscripting and when should you use which. If you only want at single column, use $ as in `iris$Species`. However, if the name of the variable includes spaces (bit of a pain), then you can use names to extract the variable(s) or a special form of the $ operator:

```
> # Change last name of iris to Species Label
> names(iris)[6]="Species Label"
> names(iris)

[1] "Sepal.Length"  "Sepal.Width"   "Petal.Length"  "Petal.Width"
[5] "Species"       "Species Label"

> # iris$Species Label will not work but
> # iris$"Species Label" will work as well as
> head(iris[,c("Petal.Width","Species Label")])

  Petal.Width Species Label
1         0.2        setosa
2         0.2        setosa
3         0.2        setosa
4         0.2        setosa
5         0.2        setosa
6         0.4        setosa
```

Extracting columns based on names is most useful when you compute the name of the variables to extract. For example, if you have many column names that have a similar structure and you can compose the names with `paste` or a different manner as shown in the dummy example below:

```
> # Create dummy data with 200 columns and 5 rows
> my.data.frame=as.data.frame(matrix(1:1000,ncol=200))
> names(my.data.frame)=paste("Measurement",1:200,sep="")
> # Extract measurements 17-20
> my.names=paste("Measurement",17:20,sep="")
> my.data.frame[,my.names]

  Measurement17 Measurement18 Measurement19 Measurement20
1            81            86            91            96
2            82            87            92            97
3            83            88            93            98
4            84            89            94            99
5            85            90            95           100
```

Subscripting based on row names will only be useful if you have assigned a meaningful name to each row.

Logical subscripting is certainly one of the most useful for dataframes because it provides a mechanism for subsetting the data based on conditions that you specify. Once you allow columns to be of different modes, as in a dataframe, logical subscripting of column subsets does not make much sense. Instead, what we want to subset or extract are rows (observations) based on values in the column(s). For example, if we only wanted to use the iris data for the versicolor species we could extract those rows as follows:

```
> # iris$Species=="versicolor" creates a vector of logical values
> # which is either TRUE (include the row) or FALSE (exclude the row)
> versicolor=iris[iris$Species=="versicolor",]
> head(versicolor)
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species Species Label
51          7.0         3.2          4.7         1.4 versicolor    versicolor
52          6.4         3.2          4.5         1.5 versicolor    versicolor
53          6.9         3.1          4.9         1.5 versicolor    versicolor
54          5.5         2.3          4.0         1.3 versicolor    versicolor
55          6.5         2.8          4.6         1.5 versicolor    versicolor
56          5.7         2.8          4.5         1.3 versicolor    versicolor
```

If we wanted to select versicolor with petals longer than 4.5 we would use:

```
> versicolor.longpetal=iris[iris$Species=="versicolor"&iris$Petal.Length>4.5,]
> head(versicolor.longpetal)
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species Species Label
51          7.0         3.2          4.7         1.4 versicolor    versicolor
53          6.9         3.1          4.9         1.5 versicolor    versicolor
55          6.5         2.8          4.6         1.5 versicolor    versicolor
57          6.3         3.3          4.7         1.6 versicolor    versicolor
59          6.6         2.9          4.6         1.3 versicolor    versicolor
64          6.1         2.9          4.7         1.4 versicolor    versicolor
```

Notice that in both cases the row numbers and names are maintained from the original iris dataframe. Now let's make it slightly more complicated and create the subset of versicolor with long petals and setosa with short petals:

```
> # To make it more readable create logical subsetting vectors
> versicolor.longpetal=iris$Species=="versicolor"&iris$Petal.Length>4.5
> setosa.shortpetal=iris$Species=="setosa"&iris$Petal.Length<4.5
> mixed=iris[versicolor.longpetal | setosa.shortpetal,]
> table(mixed$Species,mixed$Petal.Length)
```

```
            1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.9 4.6 4.7 4.8 4.9  5 5.1
  setosa    1   1   2   7  13  13   7   4   2   0   0   0   0  0   0
  versicolor 0  0   0   0   0   0   0   0   0   3   5   2   2  1   1
  virginica  0  0   0   0   0   0   0   0   0   0   0   0   0  0   0
```

Even with breaking up the statements, they can be difficult to read. I'll describe 2 alternatives with the `subset` function and then a more general approach using `with`. Using the subset function the above example could be done as follows:

```
> mixed=subset(iris,subset=(Species=="versicolor"& Petal.Length>4.5)|
+                         (Species=="setosa"    & Petal.Length<4.5) )
> table(mixed$Species,mixed$Petal.Length)
```

```
            1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.9 4.6 4.7 4.8 4.9  5 5.1
  setosa    1   1   2   7  13  13   7   4   2   0   0   0   0  0   0
  versicolor 0  0   0   0   0   0   0   0   0   3   5   2   2  1   1
  virginica  0  0   0   0   0   0   0   0   0   0   0   0   0  0   0
```

You can also use the select argument to specify which columns to select:

```
> mixed=subset(iris,subset=Species=="versicolor"& Petal.Length>4.5,
+                   select=c("Species","Petal.Length") )
> head(mixed)

      Species Petal.Length
51 versicolor          4.7
53 versicolor          4.9
55 versicolor          4.6
57 versicolor          4.7
59 versicolor          4.6
64 versicolor          4.7
```

Notice that I selected only 2 columns and I rearranged them as `Species, Petal.Length`. You can also use the `select` without `subset` to select specific columns and possibly rearrange them.

A more general solution to reduce the redundancy and typing is to use `with`, which is like a temporary `attach` which I have not introduced because I think it should be avoided. To demonstrate with I'll replicate the last example:

```
> versicolor.longpetal = with(iris, Species == "versicolor" & Petal.Length >
+     4.5)
> mixed = iris[versicolor.longpetal, c("Species", "Petal.Length")]
> head(mixed)

      Species Petal.Length
51 versicolor          4.7
53 versicolor          4.9
55 versicolor          4.6
57 versicolor          4.7
59 versicolor          4.6
64 versicolor          4.7
```

In this example, `with` doesn't help much with the redundancy but it can be much more useful in more complicated calculations with multiple steps using braces {} with multiple expressions.

Before leaving dataframes, I'll also mention the `transform` function which like `subset` provides a more readable approach that allows you to modify or add columns in a dataframe. For example, if I wanted to add a column which is the log of Petal.Width and scale Petal.Length by multiplying by 100 I could use the $ notation or `transform` as shown below:

```
> # Using transform
> new.iris=transform(iris,Petal.Length=Petal.Length*100,
+                          Log.Petal.Width=log(Petal.Width))
> summary(new.iris)

  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :100.0   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:160.0   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :435.0   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :375.8   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:510.0   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :690.0   Max.   :2.500
       Species   Species.Label      Log.Petal.Width
 setosa    :50   Length:150         Min.   :-2.3026
 versicolor:50   Class :character   1st Qu.:-1.2040
 virginica :50   Mode  :character   Median : 0.2624
                                    Mean   :-0.1723
                                    3rd Qu.: 0.5878
                                    Max.   : 0.9163

> # Using standard notation
> iris$Petal.Length=iris$Petal.Length*100
> iris$Log.Petal.Width=log(iris$Petal.Width)
> summary(iris)

  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :100.0   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:160.0   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :435.0   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :375.8   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:510.0   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :690.0   Max.   :2.500
       Species   Species Label      Log.Petal.Width
 setosa    :50   Length:150         Min.   :-2.3026
 versicolor:50   Class :character   1st Qu.:-1.2040
 virginica :50   Mode  :character   Median : 0.2624
                                    Mean   :-0.1723
                                    3rd Qu.: 0.5878
                                    Max.   : 0.9163
```

One final note about removing a column in a dataframe which can be done by assigning NULL to the column:

```
> names(iris)

[1] "Sepal.Length"    "Sepal.Width"     "Petal.Length"    "Petal.Width"
[5] "Species"         "Species Label"   "Log.Petal.Width"
```

```
> iris$Log.Petal.Width=NULL
> names(iris)

[1] "Sepal.Length"  "Sepal.Width"   "Petal.Length"  "Petal.Width"
[5] "Species"       "Species Label"
```

5.4 Lists

A list is the most general data structure and it is often used as the value returned by a function when more than one value needs to be returned. Lists have elements that can contain any other R object including another list. A list is a vector with a mode of list. As mentioned previously, dataframes are lists where list elements are columns(variables) in the dataframe and all of the columns have the same length. Typically the columns in a dataframe are one-dimensional atomic vectors but they can be more general (e.g., list with same length as rows in dataframe). Below I create an example list with the `list` function and show how to use subscripts with a list.

```
> my.list = list(a.vector = my.vector, a.data.frame = iris, a.grocery.list = c("bread"
+     "milk"))
> str(my.list)

List of 3
 $ a.vector      : Named num [1:3] 1 14 -32
  ..- attr(*, "names")= chr [1:3] "x" "my y" "z"
 $ a.data.frame  :'data.frame':        150 obs. of  6 variables:
  ..$ Sepal.Length : num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
  ..$ Sepal.Width  : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
  ..$ Petal.Length : num [1:150] 140 140 130 150 140 170 140 150 140 150 ...
  ..$ Petal.Width  : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
  ..$ Species      : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ..
  ..$ Species Label: chr [1:150] "setosa" "setosa" "setosa" "setosa" ...
 $ a.grocery.list: chr [1:2] "bread" "milk"
```

So the first element in the list is a numeric vector of length 3, the second is a dataframe with 150 rows and 6 columns and the third is a character vector of length 2. If I wanted to extract a sub-list of the list I use single square braces and the result is a list with 2 elements in this case:

```
> my.sub.list=my.list[c(1,3)]
> # or I could have used the names as follows
> my.sub.list=my.list[c("a.vector","a.grocery.list")]
> str(my.sub.list)

List of 2
 $ a.vector      : Named num [1:3] 1 14 -32
  ..- attr(*, "names")= chr [1:3] "x" "my y" "z"
 $ a.grocery.list: chr [1:2] "bread" "milk"
```

```
> is.list(my.sub.list)

[1] TRUE
```

Because a dataframe is a list then you can now realize that I can extract columns from a dataframe with:

```
> # a matrix approach or
> head(iris[,2:4])

  Sepal.Width Petal.Length Petal.Width
1         3.5          140         0.2
2         3.0          140         0.2
3         3.2          130         0.2
4         3.1          150         0.2
5         3.6          140         0.2
6         3.9          170         0.4

> # or with list extraction
> head(iris[2:4])

  Sepal.Width Petal.Length Petal.Width
1         3.5          140         0.2
2         3.0          140         0.2
3         3.2          130         0.2
4         3.1          150         0.2
5         3.6          140         0.2
6         3.9          170         0.4
```

If instead I want to extract a particular element of the list then I use the double square braces which returns a result with the mode of the element and not a list:

```
> a.vector = my.list[[1]]
> str(a.vector)

 Named num [1:3] 1 14 -32
 - attr(*, "names")= chr [1:3] "x" "my y" "z"

> is.list(a.vector)

[1] FALSE
```

Alternatively, if the list elements are named then I can use the $ operator as shown with dataframes which is equivalent to the square double braces:

```
> a.data.frame = my.list$a.data.frame
> str(a.data.frame)
```

```
'data.frame':        150 obs. of  6 variables:
 $ Sepal.Length : num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width  : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length : num  140 140 130 150 140 170 140 150 140 150 ...
 $ Petal.Width  : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species      : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Species Label: chr  "setosa" "setosa" "setosa" "setosa" ...
```

```
> is.list(a.data.frame)
```

```
[1] TRUE
```

Note that a dataframe is a list so the call to `is.list` returns `TRUE` because `a.data.frame` is a list within the list `my.list`.

6 Scripts and Functions

Even though I've suggested that you work with R interactively that is only a starting point and eventually you'll want to save your commands as a script, which is simply a text file containing R code. If you have been interactively working in R, you can use `history(n)` to recall the last n commands into a text file that you can edit and save. I recommend using Tinn-R (`http://www.sciviews.org/Tinn-R/`) to create and edit scripts because it knows the R language and for any file with a .r extension it will highlight pairs of braces and parentheses to help you align and check for missing ones. It also provides context sensitive help. For example, as you enter an R function it will show the arguments and their order. I work out a problem by typing commands into R and trying various solutions that I think will work and as I get close to my solution, I copy my commands to Tinn-R (`http://www.sciviews.org/Tinn-R/`) and then refine and document the code. Unless the analysis is one-off and quite simple, after composing the various pieces of the analysis I use the package capability to create a package with code, data and documentation.

Most of what you do with R involves calling functions. And without describing how functions work, I've been using many functions so far. However, it is useful to have a more fundamental understanding of functions. A function accepts values for its arguments, assigns defaults (if any) for unspecified arguments, manipulates values of the arguments to accomplish a particular task, and then will usually return an object which you can use or assign to an object for later usage. Let's consider the function `mean` which is part of the R base package. If you want to see a description of the function, type `?mean` or `help(mean)` and you will get a description and details about the function, its arguments and its values (what the function returns), some examples and other material. If you type `mean` followed by return, the function is not executed and instead the contents of the function will be displayed:

```
> mean
```

```
function (x, ...)
UseMethod("mean")
<environment: namespace:base>
```

I think it is important to look at functions because one of the best ways to learn R programming is to see how R developers write code. To go very far with that, you need to understand the concept of generic functions like `mean`. Most likely the code does not look like what you would expect. You might have expected that the code for this function to sum the values and divide by the sample size. However, generic functions like mean that contain the function call `UseMethod()` work for different classes of objects. Generic functions may also have a line in the help description showing "## Default S3 method: " in the usage section which means that they work with S3 classes of objects. S3 classes are a simple structure for a type of object-oriented programming. You only need to know that when you use a generic function, it evaluates the `class` of the first argument (usually `x`), and calls a function that was designed for that `class`, if it exists. The name of the class-specific function will be `generic.class` where "generic" is the name of the generic function and "class" is the object `class`. If a function does not exist for the `class`, then it will call `generic.default`, the default for that generic function. You can see what classes have specific functions for a generic by using the `methods` function. Below I show the results for `mean`:

```
> methods(mean)

[1] mean.data.frame mean.Date       mean.default    mean.difftime
[5] mean.POSIXct    mean.POSIXlt
```

It shows there are specific functions for dataframes, times and dates, but for most objects `mean.default` will be called. The code for `mean.default` is:

```
function (x, trim = 0, na.rm = FALSE, ...)
{
    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
        warning("argument is not numeric or logical: returning NA")
        return(NA_real_)
    }
    if (na.rm)
        x <- x[!is.na(x)]
    if (!is.numeric(trim) || length(trim) != 1L)
        stop("'trim' must be numeric of length one")
    n <- length(x)
    if (trim > 0 && n) {
        if (is.complex(x))
            stop("trimmed means are not defined for complex data")
        if (trim >= 0.5)
            return(stats::median(x, na.rm = FALSE))
        lo <- floor(n * trim) + 1
        hi <- n + 1 - lo
        x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
    }
    .Internal(mean(x))
```

```
}
<environment: namespace:base>
```

The code does some tests for the validity of the arguments, handles missing values (NA), handles any trimming and as long as trimming is less than 50%, it calls `.Internal(mean(x))` which means that it is calling an internal function which will be written in either C or FORTRAN. While you can view the source for internal functions, that is probably further than you want to go into the bowels of R. While basic functions are internal, most functions in R are written with R and you'll be able to examine all of the code.

Many users may always be able to write simple scripts to analyze their data by calling functions that are already written as part of the base package or in one of the 1000s of contributed packages for R. However, if you ever find yourself replicating code in your script in which only change a few things, then you probably should be creating your own function that you can call from your script. However, before you set out on writing a function for a particular application, do yourself a favor and poke around the 1000s of packages using a search engine such as at (`http://finzi.psych.upenn.edu/`) to see if a function has already been written for your application. With a little time spent searching you may save yourself some time. Additionally, it is useful to know how to write simple functions for use with the family of `apply` functions which I'll describe later. Also, for other functions like `optim` or `integrate`, the primary argument that you pass is the name of a function that you write which is minimized or integrated, respectively.

Once you know the R language, learning to create a function is fairly simple. All you need to do is to create a function name (if saved), define its arguments and it's return value(s) and write the code that performs whatever task you need. The general format is:

```
my.function=function(arguments separated by commas)
{
your R code
return(your value)
}
```

Instead of using the return function, you can make the last line of the function be the object to be returned. Also, the braces {} are only needed if there is more than one statement in the function. So it is possible to write a function as simple as `function(x)sum(x>0)`. Unnamed simple functions like these are used frequently with the `apply` family of functions. That simple but useful function creates a logical vector with the values `TRUE` if the element in `x` is `> 0` and `FALSE` otherwise. Then `sum(x>0)`provides the count of number of positive values in `x` because logical values are automatically coerced to `1/0` for `TRUE/FALSE` in any arithmetic operation.

Function arguments are passed by value which means a copy of the object is passed. Thus, any changes you make to the copy are not made in the original object if it was an object in your workspace. If you wanted to modify an object in the workspace, you would need to return the new object as the return value of the function and then assign the result to the original object. Below, I give an example with a silly little function called `add2`.

```
> add2 = function(x) x + 2
> my.vector
```

```
  x my y    z
  1   14  -32
```

```
> my.vector = add2(my.vector)
> my.vector
```

```
  x my y    z
  3   16  -30
```

Arguments to functions can be passed by order or by `name=value` or by both methods in the same call. It is quite common to specify the first argument or two in order and then the remaining arguments by value. Also, some arguments have default values and need not be specified. For example, consider the function `rnorm` which generates random values from a normal distribution with a particular mean and standard deviation. It's arguments are n,mean=0,sd=1 so if we are happy with generating variables with a standard normal distribution then we only need to specify a value for the first argument n. Below we show some examples of different methods of calling this function:

```
> # set random seed so it always shows same values
> set.seed(1)
> # only set n
> rnorm(5)
```

```
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

```
> # set n and sd by order
> rnorm(5,,3)
```

```
[1] -2.4614052  1.4622872  2.2149741  1.7273441 -0.9161652
```

```
> # set n by order and sd by name=value
> rnorm(5,sd=3)
```

```
[1]  4.535344  1.169530 -1.863722 -6.644100  3.374793
```

A special argument used with functions is ... which allows an arbitrary set of arguments passed as argument=value to a nested function call. This allows passing a set of arguments without having to specify all of the possible arguments. The family of apply functions (described below) use this approach to be able to pass unspecified arguments to a user-defined function. I'll show an example (not particularly useful) below in which I write a simple function that calls plot and to be able to pass any of the plot arguments I'll use the ... argument.

```
> my.plot.function=function(x,y,...)
+ {
+   log.x=log(x)
+   log.y=log(y)
+   plot(log.x.log.y,...)
+   invisible()
+ }
```

The above function used `invisible()`in place of returning a value which you might want to do if the function was written solely for its side effect like producing a plot. However, typically you'll want to return one or more values. When you want to return more than one value use list with named elements. Here is another silly function that extends add2 above:

```
> add2.and.add3 = function(x) return(list(add2 = x + 2, add3 = x +
+     3))
> my.vector

   x my y    z
   3   16  -30

> my.vector.list = add2.and.add3(my.vector)
> my.vector.list

$add2
   x my y    z
   5   18  -28

$add3
   x my y    z
   6   19  -27
```

Before finishing with functions, I'll discuss what objects are available to a function and which are not. This is called scoping. R uses lexical scoping which means that the scope of the variable depends on where a function was defined. Other programming languages use dynamic scoping which means that the scope of the variable depends on where the function was called. The easiest way to understand this is with an example. Below I define a function myf1 which contains the definition of another function myf2. Each of these functions simply prints out the value of a variable y. The variable y =1 is defined in the same environment that myf1 is defined. In myf1, myf2 is called and the value y=1 is printed and then y is assigned 2 and myf2 is called again and it prints y=2. Then upon completing the call to myf1, y is printed again and its value is still 1. The same would have been true if y was an argument to myf1 because arguments are passed by value and cannot be changed in the function. The only exception is the use of the <<- assignment operator which will change the value globally. The use of that assignment operator is not recommended. Thus, without being specified as an argument functions can use values of variables from the environment in which they are called. This can be a benefit when using functions like myf2 defined within myf1 but it can be dangerous in cases where say you intended to type y2 in a function, but type y and it uses a value from the environment instead of one you have as a defined argument.

```
> y = 1
> myf1 = function() {
+     myf2 = function() {
```

```
+         cat("y=", y, "\n")
+         invisible()
+     }
+     cat("\ny=", y, "\n")
+     myf2()
+     y = 2
+     myf2()
+     invisible()
+ }
> myf1()

y= 1
y= 1
y= 2

> cat("\ny=", y)

y= 1
```

I'll show one more example, in which I define another function myf2 which sets y=3 and then calls myf1. Note that this myf2 does not interfere with the other myf2 defined inside myf1 because it is local to myf1. If R used dynamic scoping, then you would expect to see 3,3,2 printed but in fact you get the same result as above because myf1 was defined in the environment with y=1 and being called from myf2 where y was set to 3 does not affect the environment from which myf1 was defined. Scoping is lexical and not dynamic.

```
> myf2 = function() {
+     y = 3
+     myf1()
+     invisible()
+ }
> myf2()

y= 1
y= 1
y= 2
```

7 Loops in R

R was designed as an interactive environment and the R language is interpreted rather than compiled as in FORTRAN or C; although many of the primitive functions like `mean` use compiled code. You may hear complaints from various people that they tried R and thought it was much too slow for their particular application. Sometimes this occurs because programmers approach R and attempt to use it like they would with FORTRAN or C. Often this involves looping which means repeatedly conducting some set of calculations

while possibly changing some index to work on different values of the data. You can use looping constructs in R programming but they can be slower in some circumstances. In earlier versions of R looping was extremely slow in comparison to alternatives like the apply family of functions but that is no longer the case. Thus, you should do what works for you. If a loop construct is not too slow and you can't work out how to use the apply functions then use loops. In some cases, no matter what you do it is possible that R may simply be too slow for what you want to accomplish.. However, that does not mean that you need to abandon R for a compiled language. If you can program in C or FORTRAN then you can write compiled code in one of those languages and use it from R for your specific circumstance for the parts of your code that are slow. I'm not going to discuss that aspect here but I will describe the `apply` family of functions and how you can think of them in terms of looping. The `apply` family of functions usually provides an improvement in speed but more importantly they are a concise approach to specifying loops. Once you understand them you are unlikely to go back to loops.

The `apply` family of functions applies a user-specified function over the elements of the primary argument. The various apply functions differ based on the mode of the primary argument and the values they return. The family includes: `apply, sapply, lapply, tapply, mapply, rapply` and others. If you want to loop over a matrix or array, you'll use `apply`. If you want to loop over a vector (remember that lists are vectors), use `lapply` if you want the result to be a list or use `sapply` if you want the result to be converted to a vector or matrix, as appropriate. If you want to loop over an atomic vector (e.g., numeric vector), use `tapply` to return either an array or a list depending on the function. The function `mapply` is a multivariate version of `sapply` in which any number of vectors can be supplied as arguments and `rapply` is a recursive version of `lapply`.

7.1 Apply

I'll start with `apply` because I think it is the easiest to demonstrate and to understand. I'll start with a matrix which has 2 dimensions with the rows being dimension 1 and the columns dimension 2. These are referred to as margins. For example, if I want to know the sum of the elements in each row of the matrix, I could write that with the following loop:

```
> # define a matrix to use
> my.matrix=matrix(1:12,nrow=3,ncol=4)
> # pre-allocate a vector to contain the row sums; using nrow(my.matrix)
> # as the length of the vector is good practice because it makes it
> # easy to generalize the code for a function
> my.row.sums=vector("numeric",nrow(my.matrix))
> # Loop over the rows and sum the elements in the row
> for(i in 1:nrow(my.matrix))
+   my.row.sums[i]=sum(my.matrix[i,])
> my.row.sums

[1] 22 26 30
```

```
> # The above will be faster than the following because it uses the
> # vectorized sum function which sums all the elements in a vector;
> # An old FORTRAN programmer might have the tendency
> # to do the following.  This is not advised.
> for(i in 1:nrow(my.matrix))
+ {
+   my.row.sums[i]=0
+   for (j in 1:ncol(my.matrix))
+     my.row.sums[i]=my.row.sums[i]+my.matrix[i,j]
+ }
> my.row.sums

[1] 22 26 30
```

With `apply` we can accomplish the same task with:

```
> # The first agument is the object to be used
> # The second argument is the margin
> # The third argument is the function to be applied
> apply(my.matrix,1,sum)

[1] 22 26 30
```

To demonstrate the use of ... discussed in 6, I'll add a missing value (`NA`) to the matrix and demonstrate how the argument `na.rm` is passed to `sum`.

```
> # Replace element 2,3 with an NA
> my.matrix[2,3]=NA
> # See how sum of second row is now NA
> apply(my.matrix,1,sum)

[1] 22 NA 30

> # Use na.rm=TRUE to remove(ignore) NA values in computing the sum
> apply(my.matrix,1,sum,na.rm=TRUE)

[1] 22 18 30
```

Any argument=value pairings given at the end of the function call are passed to the function you specified (`sum` in this example) using the ... notation in `apply`. If I wanted to compute the means of each column, then the looping and equivalent apply code would be written as:

```
> # pre-allocate a vector to contain the coumn means sums
> my.col.means=vector("numeric",ncol(my.matrix))
> # Loop over the columns and compute mean of values in the column and ignore NA
> for(i in 1:ncol(my.matrix))
+   my.col.means[i]=mean(my.matrix[,i],na.rm=TRUE)
> my.col.means
```

```
[1]  2  5  8 11

> # now use apply
> apply(my.matrix,2,mean,na.rm=TRUE)

[1]  2  5  8 11
```

I've used fairly simple examples to keep the focus on the equivalence between loops and `apply`, but you should know that for these simple common cases, there are primitive functions `rowSums`, `colSums`, `rowMeans` and `colMeans` which you can apply directly to a matrix. Also, remember that most operations in R are vectorized so `sum(my.matrix)` and `mean(my.matrix)` will produce the sum and mean of all elements in the matrix.

Now let's consider a more complicated situation with margins using a 3 dimensional array. We'll use `my.array` from 5.2and we'll compute the mean of subsets across the margins. Let's say that we want to compute the mean across the values in the 3rd dimension for each of the first 2 dimensions.

```
> apply(my.array,c(1,2),mean)

         y1        y2
x1 6.333333  8.333333
x2 7.000000 11.000000
x3 7.666667 12.000000

> # if we switch the order of the margins then the result is transposed
> # and is equivalent to t(apply(my.array,c(1,2),mean))
> #
> apply(my.array,c(2,1),mean)

         x1 x2        x3
y1 6.333333  7  7.666667
y2 8.333333 11 12.000000
```

Now if we wanted the same quantity across elements in the first and third dimension, we would use the following to return a vector:

```
> sqrt(apply(my.array,c(2),mean))

      y1       y2
2.645751 3.231787
```

The margins that are excluded are used in the computation for the dimensions that are specified. It is important to understand what the function does with the object that you give you are passing as part of the apply process. Instead of the mean, let's consider computing the standard deviation using the square root of the variance (`var`) function.

```
> # Note that sqrt is working on a matrix and computing elementwise
> # square roots; in this case.
> sqrt(apply(my.array,c(1,2),var))

         y1       y2
x1 7.023769 8.621678
x2 7.549834 6.000000
x3 8.082904 6.000000

> # Now if we want to compute the variance of all the elements in
> # dimensions 1 and 3 for each
> # entry in the second dimenstion then we might try the following:
> #
> sqrt(apply(my.array,c(2),var))

      y1       y2
 [1,]  0 3.785939
 [2,]  0 1.870829
 [3,]  0 1.870829
 [4,]  0 1.870829
 [5,]  1 1.000000
 [6,]  1 1.000000
 [7,]  0 1.870829
 [8,]  1 1.000000
 [9,]  1 1.000000
```

You were probably expecting a vector with 2 values for the second dimension. However, what you are passing to `var` is a matrix and if you pass a single matrix to `var`, it will call `cov` and compute the covariance matrix between the rows and columns of the matrix and the result above is the result of turning each of the 3 by 3 covariance matrices into a vector (9 rows) and then displaying the matrix with 2 columns. To see that is the case, try `sqrt(var(my.array[,2,]))` to see that it corresponds to the second column when the matrix is converted to a vector (columnwise). That is certainly not we had in mind and the result using the `sd` (standard deviation) function also will do what we want. So this provides an opportunity to show how you can write your own function to be applied. Below I create a simple function that will compute the standard deviation of a vector or matrix after converting the argument to a vector. Then I apply it to various sets of margins.

```
> # create the function and use ... so any arguments to sd can be passed along
> my.sd=function(x,...)
+  sd(as.vector(x),...)
> apply(my.array,2,my.sd)

      y1       y2
6.576473 6.267199
```

```
> # replace a value with NA and show how to use na.rm=TRUE
> my.array[2,2,2]=NA
> apply(my.array,2,my.sd)

      y1       y2
6.576473       NA

> apply(my.array,2,my.sd,na.rm=TRUE)

      y1       y2
6.576473 6.696214

> # if I was only going to use the function once, it could be defined
> # inside the body of the function.  Here I purposefully make this
> # one line function into 2 lines to show that this is not limited
> # to one line functions. Note that the function is not named.
> #
> apply(my.array,c(1,2),
+      function(x,...)
+        {
+          x=as.vector(x)
+          sd(x,...)
+        },
+      na.rm=TRUE)

         y1       y2
x1 7.023769 8.621678
x2 7.549834 8.485281
x3 8.082904 6.000000
```

7.2 Lapply and Sapply

If you want to loop over a vector either atomic or a list, then you'll want to use `lapply` if you want the result to be a list or use `sapply` if the result of the function can be simplified to a vector or a matrix. I'll carry-on using the `mean` example with the `iris` dataframe (also a list) to see the parallel with `apply`. I'll use the first 4 numeric fields in iris and show the differences in results for `lapply` and `sapply` on this list.

```
> # get iris data and create new.iris with first 4 columns
> #
> data(iris)
> new.iris=iris[,1:4]
> # sapply loops over the list elements which are the columns in
> # a dataframe. Because mean returns a single value, sapply returns a vector
> #
> sapply(new.iris,mean)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333

> # It is equivalent to the following loop; note use of the [[]]
> # to extract a vector rather than using [] to get a list with one element.
> #
> my.iris.means=vector("numeric",ncol(new.iris))
> for (i in 1:ncol(new.iris))
+   my.iris.means[i]=mean(new.iris[[i]])
> # If I want to add the names of each column I could do that with
> #
> names(my.iris.means)=names(new.iris)
> my.iris.means

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

Now if I do the same thing with `lapply`, I get a list instead of a vector because `lapply` always returns a list without trying to simplify the list:

```
> lapply(new.iris,mean)

$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333

> # lapply is equivalent to the following loop
> #
> my.iris.means=vector("list",ncol(new.iris))
> for (i in 1:ncol(new.iris))
+   my.iris.means[[i]]=mean(new.iris[[i]])
> my.iris.means

[[1]]
[1] 5.843333

[[2]]
[1] 3.057333
```

```
[[3]]
[1] 3.758

[[4]]
[1] 1.199333

> # If I want to add the names of each column I could do that with
> #
> names(my.iris.means)=names(new.iris)
> my.iris.means

$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333

> # Note that both of these examples would not be used in practice
> # because mean is a generic function with a useful version for dataframes
> mean(new.iris)

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

So let's move away from means and consider an example where you want to compute a correlation matrix of all the numeric measurements (first 4 columns) separately for each species. The result will be a matrix so we'll probably want to use `lapply` because `sapply` would combine all the matrices and we would have to split them up by species:

```
> # create a list with 3 dataframes for each of the 3 species
> #
> my.split.df=split(iris,iris$Species)
> # Now use lapply to construct a list of correlation matrices
> # but only using the first 4 columns
> #
> species.cor=lapply(my.split.df, function(x) cor(x[1:4]))
> species.cor
```

```
$setosa
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.7425467    0.2671758   0.2780984
Sepal.Width     0.7425467   1.0000000    0.1777000   0.2327520
Petal.Length    0.2671758   0.1777000    1.0000000   0.3316300
Petal.Width     0.2780984   0.2327520    0.3316300   1.0000000

$versicolor
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.5259107    0.7540490   0.5464611
Sepal.Width     0.5259107   1.0000000    0.5605221   0.6639987
Petal.Length    0.7540490   0.5605221    1.0000000   0.7866681
Petal.Width     0.5464611   0.6639987    0.7866681   1.0000000

$virginica
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.4572278    0.8642247   0.2811077
Sepal.Width     0.4572278   1.0000000    0.4010446   0.5377280
Petal.Length    0.8642247   0.4010446    1.0000000   0.3221082
Petal.Width     0.2811077   0.5377280    0.3221082   1.0000000
```

Now what if we wanted to get for each species the maximum positive correlation while excluding the diagonal using the lower.tri function to provide acceptable indices within the matrix. We can use `sapply` on the list of matrices:

```
> sapply(species.cor, function(x) max(x[lower.tri(x)]) )

    setosa versicolor  virginica
 0.7425467  0.7866681  0.8642247
```

maximum correlation. I can expand the above code and extract the row and column names of the variables with the maximum correlation. I use the `which` function to get the indices (`arr.ind=TRUE`) of the matrix elements (possibly more than one) with the maximum correlation. With the row and column index, I can extract the row and column names and return a dataframe and `lapply` will return a list of dataframes.

```
> # This function is rather involved so define it and then use it
> # This is useful if you get an error and then want to use
> # debug(my.function) to step through the function to locate the error
> #
> max.cor=function(x)
+ {
+    max.corr=max(x[lower.tri(x)])
+    max.indices=which(x==max.corr&lower.tri(x),arr.ind=TRUE)
+    var1=row.names(x)[max.indices[,1]]
+    var2=colnames(x)[max.indices[,2]]
```

```
+    data.frame(Var1=var1,Var2=var2,max.corr=max.corr)
+ }
> max.cor.df=lapply(species.cor,max.cor)
> max.cor.df

$setosa
         Var1         Var2  max.corr
1 Sepal.Width Sepal.Length 0.7425467

$versicolor
         Var1         Var2  max.corr
1 Petal.Width Petal.Length 0.7866681

$virginica
          Var1         Var2  max.corr
1 Petal.Length Sepal.Length 0.8642247
```

I could have used `sapply` but with a mix of characters and numeric it would have coerced a return vector to a character matrix. If I want to end up with a single dataframe rather than a list of dataframes, I can use the `do.call` function which uses a named function (`rbind` in this case) with all the elements in the list:

```
> do.call("rbind",max.cor.df)

                  Var1         Var2  max.corr
setosa      Sepal.Width Sepal.Length 0.7425467
versicolor  Petal.Width Petal.Length 0.7866681
virginica  Petal.Length Sepal.Length 0.8642247
```

To use a function with `lapply` or `sapply`, it is the first argument of the named function that is varied for each call. What if we wanted construct a linear regression of petal width versus petal length separately for each species. For lm, the first argument is the formula and it is the data that we want to vary for each regression. To accomplish a task like that we can use either sapply or lapply across an integer sequence which indexes the subsets. In this case, the parallel with a loop construct is very obvious. Both `sapply` and `lapply` work with lists and a useful function called `split` can create a list of dataframes based on one or more factors. For our example, we want to split the iris data by species and with each of the dataframes conduct a linear regression of petal width versus length. The following shows how that can be done with lapply where the model is returned and with sapply where only the coefficients of the regression are returned and I want a matrix of results. In each case, I show how the same thing can be done with a loop. In this case, a loop construct is very clear because of the way the lm function is structured. :

```
> # First I'll create a user-defined function which calls lm
> # and pass it an index (.element) and a formula
> #
```

```
> my.lm=function(.element,formula)
+         lm(formula=formula,data=my.split.df[[.element]])
> # lapply and sapply:
> # Next I'll call lapply to create a list of models (one for each subset)
> #
> my.species.models=lapply(1:length(my.split.df),my.lm,
+                                 formula=Petal.Width~Petal.Length)
> # Next I'll use sapply to extract the coefficients
> # and add the species names to the rows after transposing
> #
> my.species.coefficients=sapply(my.species.models,coef)
> my.species.coefficients=t(my.species.coefficients)
> row.names(my.species.coefficients)=names(my.split.df)
> my.species.coefficients

           (Intercept) Petal.Length
setosa     -0.04822033    0.2012451
versicolor -0.08428835    0.3310536
virginica   1.13603130    0.1602970

> # Loop equivalent:
> # I could do the same thing with the following looping code
> #
> my.species.models=vector("list",length(my.split.df))
> for (.element in 1:length(my.split.df))
+   my.species.models[[.element]]=
+           lm(Petal.Width~Petal.Length,data=my.split.df[[.element]])
> my.species.coefficients=matrix(NA,nrow=length(my.split.df),ncol=2)
> for (.element in 1:length(my.split.df))
+   my.species.coefficients[.element,]=
+           coef(my.species.models[[.element]])
> row.names(my.species.coefficients)=names(my.split.df)
> colnames(my.species.coefficients)=names(coef(my.species.models[[1]]))
> my.species.coefficients

           (Intercept) Petal.Length
setosa     -0.04822033    0.2012451
versicolor -0.08428835    0.3310536
virginica   1.13603130    0.1602970
```

7.3 Tapply

If you only need to operate on a single atomic object (e.g., numeric vector), that is split into groups by one or more grouping variables (typically factors) then `tapply` is what you should use. Below are some examples using the iris data:

```
> # Compute the mean Sepal.Width for each species
> #
> result1=with(iris,tapply(Sepal.Width,Species,mean))
> result1

    setosa versicolor  virginica
     3.428      2.770      2.974

> str(result1)

 num [1:3(1d)] 3.43 2.77 2.97
 - attr(*, "dimnames")=List of 1
  ..$ : chr [1:3] "setosa" "versicolor" "virginica"

> # Compute the mean ratio Sepal.Width/Sepal.Length for each species
> # and breaks in Petal.Length
> #
> iris=transform(iris,Petal.Length.categories=cut(Petal.Length,breaks=0:6+0.5))
> result2=with(iris,
+   tapply(Sepal.Width/Sepal.Length,list(Species,Petal.Length.categories),mean))
> result2

           (0.5,1.5] (1.5,2.5] (2.5,3.5] (3.5,4.5] (4.5,5.5] (5.5,6.5]
setosa     0.6846836 0.6830096        NA        NA        NA        NA
versicolor        NA        NA 0.4592265 0.4725759 0.4598594        NA
virginica         NA        NA        NA 0.5102041 0.4624291 0.4510351

> str(result2)

 num [1:3, 1:6] 0.685 NA NA 0.683 NA ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:3] "setosa" "versicolor" "virginica"
  ..$ : chr [1:6] "(0.5,1.5]" "(1.5,2.5]" "(2.5,3.5]" "(3.5,4.5]" ...

> # Compute the median Sepal.Width for each species and breaks in
> # Sepal.Length and Petal.Length
> #
> iris=transform(iris,Sepal.Length.categories=cut(Sepal.Length,breaks=4:8))
> result3=with(iris,
+   tapply(Sepal.Width,
+     list(Species,Petal.Length.categories,Sepal.Length.categories),median))
> result3

, , (4,5]

           (0.5,1.5] (1.5,2.5] (2.5,3.5] (3.5,4.5] (4.5,5.5] (5.5,6.5]
```

```
setosa           3.2       3.4        NA        NA        NA        NA
versicolor        NA        NA       2.3        NA        NA        NA
virginica         NA        NA        NA       2.5        NA        NA

, , (5,6]

           (0.5,1.5] (1.5,2.5] (2.5,3.5] (3.5,4.5] (4.5,5.5] (5.5,6.5]
setosa           3.7       3.8        NA        NA        NA        NA
versicolor        NA        NA      2.55       2.7      2.95        NA
virginica         NA        NA        NA        NA      2.75        NA

, , (6,7]

           (0.5,1.5] (1.5,2.5] (2.5,3.5] (3.5,4.5] (4.5,5.5] (5.5,6.5]
setosa            NA        NA        NA        NA        NA        NA
versicolor        NA        NA        NA       2.9      2.95        NA
virginica         NA        NA        NA        NA      3.00       3.1

, , (7,8]

           (0.5,1.5] (1.5,2.5] (2.5,3.5] (3.5,4.5] (4.5,5.5] (5.5,6.5]
setosa            NA        NA        NA        NA        NA        NA
versicolor        NA        NA        NA        NA        NA        NA
virginica         NA        NA        NA        NA        NA         3
```

```
> str(result3)

 num [1:3, 1:6, 1:4] 3.2 NA NA 3.4 NA NA NA 2.3 NA NA ...
 - attr(*, "dimnames")=List of 3
  ..$ : chr [1:3] "setosa" "versicolor" "virginica"
  ..$ : chr [1:6] "(0.5,1.5]" "(1.5,2.5]" "(2.5,3.5]" "(3.5,4.5]" ...
  ..$ : chr [1:4] "(4,5]" "(5,6]" "(6,7]" "(7,8]"
```

In each case, the function returned a single value and the result was a vector for a single factor variable, a matrix for two factor variables and a 3 dimensional array for 3 categorical variables. If the function does not return a single value then tapply will typically return a list.

```
> result4= with(iris,tapply(Sepal.Width,Species,cut,breaks=c(-Inf,.5,1.5,3.5,Inf)))
> result4

$setosa
 [1] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (3.5, Inf] (3.5, Inf]
 [7] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (3.5, Inf] (1.5,3.5]
[13] (1.5,3.5]  (1.5,3.5]  (3.5, Inf] (3.5, Inf] (3.5, Inf] (1.5,3.5]
[19] (3.5, Inf] (3.5, Inf] (1.5,3.5]  (3.5, Inf] (3.5, Inf] (1.5,3.5]
```

```
[25] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
[31] (1.5,3.5]  (1.5,3.5]  (3.5, Inf] (3.5, Inf] (1.5,3.5]  (1.5,3.5]
[37] (1.5,3.5]  (3.5, Inf] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
[43] (1.5,3.5]  (1.5,3.5]  (3.5, Inf] (1.5,3.5]  (3.5, Inf] (1.5,3.5]
[49] (3.5, Inf] (1.5,3.5]
Levels: (-Inf,0.5] (0.5,1.5] (1.5,3.5] (3.5, Inf]

$versicolor
 [1] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5]
 [8] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5]
[15] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5]
[22] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5]
[29] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5]
[36] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5]
[43] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5] (1.5,3.5]
[50] (1.5,3.5]
Levels: (-Inf,0.5] (0.5,1.5] (1.5,3.5] (3.5, Inf]

$virginica
 [1] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
 [7] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (3.5, Inf] (1.5,3.5]  (1.5,3.5]
[13] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (3.5, Inf]
[19] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
[25] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
[31] (1.5,3.5]  (3.5, Inf] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
[37] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
[43] (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]  (1.5,3.5]
[49] (1.5,3.5]  (1.5,3.5]
Levels: (-Inf,0.5] (0.5,1.5] (1.5,3.5] (3.5, Inf]
```

```
> mode(result4)

[1] "list"
```

7.4 Package plyr

It isn't particularly easy to keep all of those apply functions in your head and remember the types of arguments and the variety of results that can be returned. If you are confused and feeling overwhelmed by all of the above, you are probably not alone. The solution is to install and use the package plyr written by Hadley Wickham. He wrote a package with a plethora of apply type functions but they are aptly named and there is no guessing as to what the arguments and returned values will be. For example, the `ddply` function accepts a dataframe and returns a dataframe; whereas, `dlply` returns a list, `daply` returns an array and `d_ply` returns nothing and the function is only used for it's side effects like plotting. Likewise there are an equivalent set of functions that start with "`a`" for arrays as

the input argument, "`l`" for list as the argument, "`m`" where arguments for replicate calls to a function are in a dataframe or array, and "`r`" for expression arguments.

I'll use a quote from his package documentation to describe this set of functions:

> All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply .fun to each piece, and then combine the pieces into a single data structure.

Even though you can specify the structure of the input and output arguments, that does not mean you can use any function and expect it to do the conversion. If you specify a function to apply that returns a result that is not consistent with the output data structure, it will issue an error. So you still need to think about what you are doing. As an example, I'll repeat the maximum correlation example in 7.2in which I needed a call to `lapply` followed by `do.call`. This can be done with a single call to `ldply` after creating the list and function:

```
> library(plyr)
> ldply(species.cor,max.cor)

         .id         Var1         Var2  max.corr
1     setosa  Sepal.Width Sepal.Length 0.7425467
2 versicolor  Petal.Width Petal.Length 0.7866681
3  virginica Petal.Length Sepal.Length 0.8642247
```

If I then preferred a list result, I could use `llply`:

```
> llply(species.cor,max.cor)

$setosa
         Var1         Var2  max.corr
1 Sepal.Width Sepal.Length 0.7425467

$versicolor
         Var1         Var2  max.corr
1 Petal.Width Petal.Length 0.7866681

$virginica
          Var1         Var2  max.corr
1 Petal.Length Sepal.Length 0.8642247
```

If the function you are applying takes awhile or you are applying it to a large number of subsets, then you have the option to show a variety of different progress bars that show the amount completed.