Vectors

R implements vectors as a fundamental object type so get familiar with them. It is useful for storing data of all the same type in a variable / data structure. You can put in variables of type integer, logical, real, and character.

```
1:10

rnorm(10)

y = 5.4     # A single assignment

y = 1:10    # A vector with 10 elements (1 .. 10)

y = c(1,2,3,4,5,6,7,8,9,10) # Same as above yet using the “c” function

y = scan()   # Allows you to enter in elements from the keyboard
1: 10
2:  9
3:  8

..
1:  1
```

Vectors - Indexing

Let’s say we have measured the heights of some people and want to stash it in a vector. Bracket notation is the key to working vectors !

```
height = c(59,70,66,72,62,66,60,60)    # create a vector of 8 heights

Let’s check out the various ways we can index into the vector

height[1:5]           # Get first 5 elements
[1] 59 70 66 72 62

height[5:1]           # Get first 5 elements in reverse
[1] 62 72 66 70 59

height[-1]            # Get all but first element
[1] 70 66 72 62 66 60 60

height[-1:-2]         # Get all but first two elements
[1] 66 72 62 66 60 60

height[c(1,5)]            # Get just first and fifth elements
[1] 59 62
```

Vectors - Logicals

Let's say we have measured the heights of some people and want to stash it in a vector:

```
height = c(59,70,66,72,62,66,60,60)  # create a vector of 8 heights
```

We can apply logical tests to a vector to find elements that satisfy a condition set

```
height
[1] 59 70 66 72 62 66 60 60

height == 72       # Test for values equal to 72
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE

height[height == 72]
[1] 72

# SAME AS

logical.vector = (height == 72)

logical.vector
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE

height[ logical.vector ]
```

Vectors - Logicals

Let’s say we have measured the heights of some people and want to stash it in a vector:

```
> height = c(59,70,66,72,62,66,60,60) # create a vector of 8 heights
```

Comparisons can be combined like a database query:

```
# Note use of the "&" / and operator

height[height > 60 & height < 70]
66      62      66

height[height > 60 & height <= 70]
70      66      62      66
```

Vectors - compared to for loop

It is important to point out that using logical operations within brackets eliminates the need to write a “for loop” every time you want to do some summary information on a vector. Which would you rather do:

This:

```
height[height > 60 & height < 70]
66      62      66
```

OR:

```
for (ii in 1:length(height)) {
    if (height[ii] > 60 & height[ii] < 70) {
       print(height[ii])
   }
}

     66
     62
     66
```

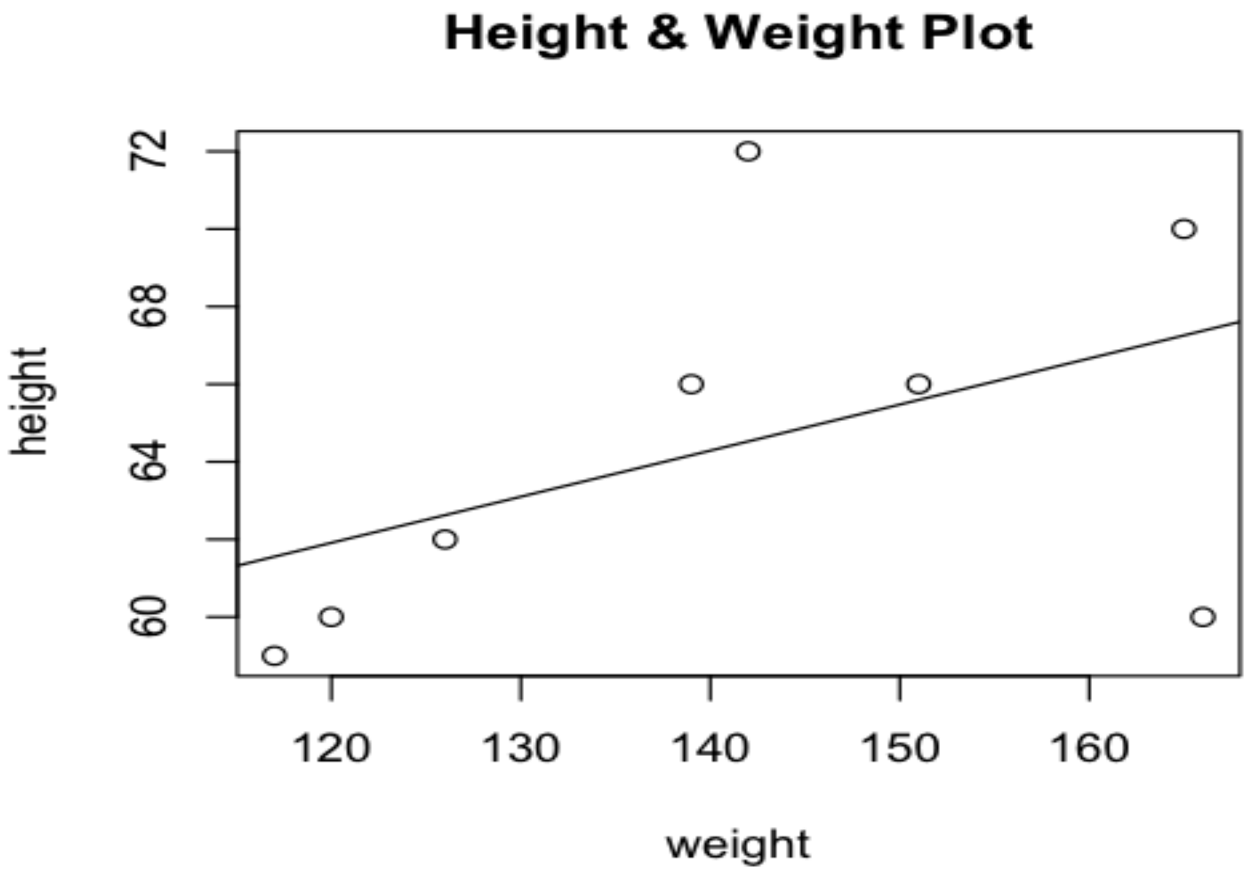
Vectors - Arithmetic

```
weight = c(117,165,139,142,126,151,120,166)   # weight (in lbs)

weight/100
[1] 1.17 1.65 1.39 1.42 1.26 1.51 1.20 1.66

sqrt(weight)
[1] 10.81665 12.84523 11.78983 11.91638 11.22497 12.28821 10.95445
12.88410

weight^2
[1] 13689 27225 19321 20164 15876 22801 14400 27556

sum((weight-mean(weight))^2)/(length(weight)-1)  # The variance formula
[1] 363.9286

var(weight)
[1] 363.9286
```

Supplemental - Vectors - lm

```
height = c(59,70,66,72,62,66,60,60)

weight = c(117,165,139,142,126,151,120,166)
# Get 8 weight measurements

cor(height,weight)            # Are they correlated ?
[1] 0.46295

plot(weight,height,main="Height & Weight Plot")  # Do a X/Y plot

res = lm(height ~ weight)       # Do a linear regression

abline(res)         # Check out the regression line
```

Supplemental - Vectors - lm



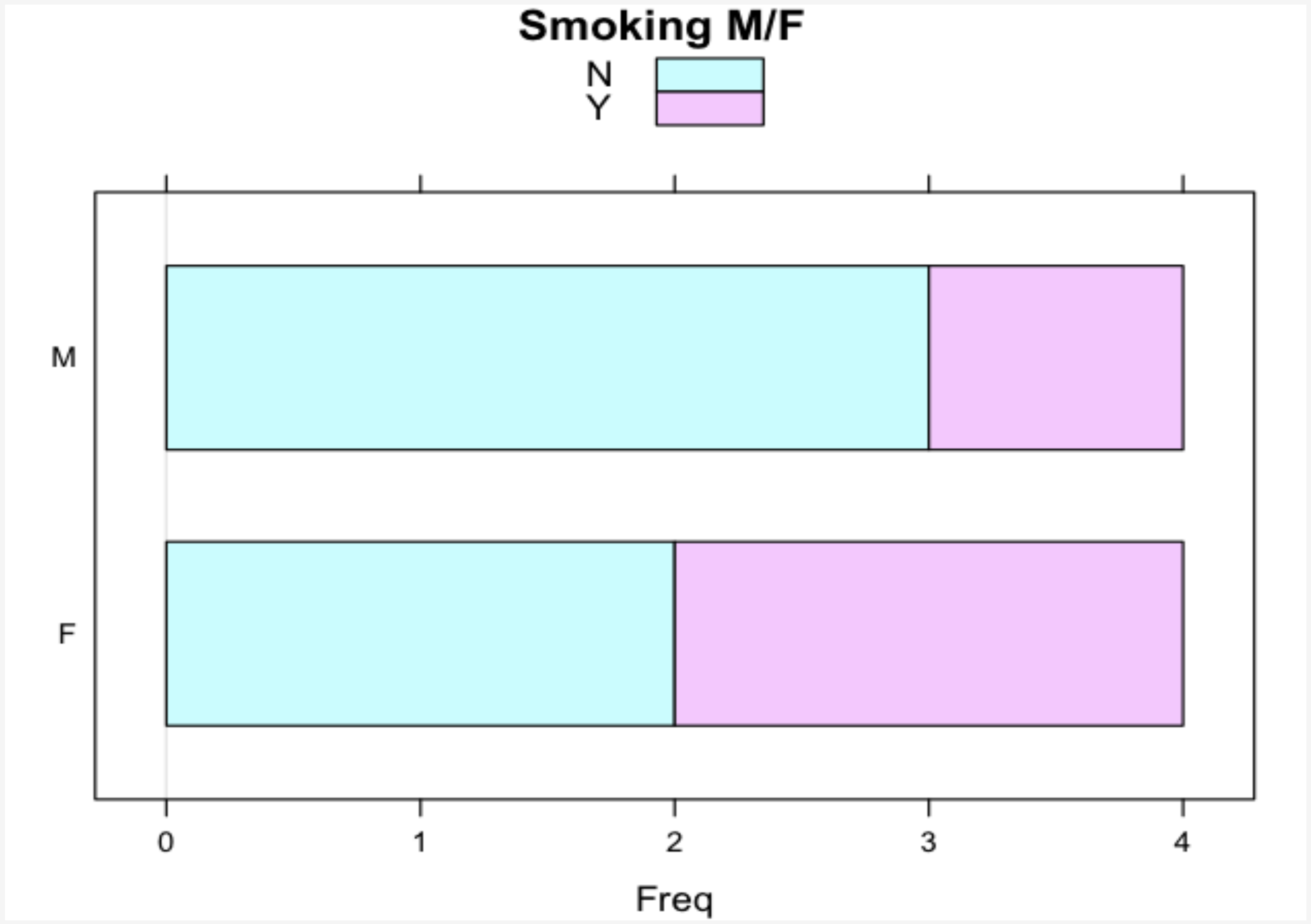
Vectors - Arithmetic

```
weight = c(117,165,139,142,126,151,120,166)   # weight (in lbs)

new.weights = weights + 1        # Vector Addition

new.weights
[1] 118 166 140 143 127 152 121 167

append(weights,new.weights) # Combines the two vectors
[1] 117 165 139 142 126 151 120 166 118 166 140 143 127 152 121 167

c(weight,new.weights)               # Equivalent to the above


weight/new.weights     # Vector division

[1] 0.9915254 0.9939759 0.9928571 0.9930070 0.9921260 0.9934211 0.9917355
0.9940120
```

Vectors - character vectors

```
gender = c("F","M","F","M","F","M","F","M")  # Get their gender

smoker = c("N","N","Y","Y","Y","N","N","N")  # Do they smoke ?

table(gender,smoker)   # Let's count them

      smoker
gender N Y
     F 2 2
     M 3 1

prop.table(table(gender,smoker))
      smoker
gender     N     Y
     F 0.250 0.250
     M 0.375 0.125

library(lattice)

barchart(table(gender,smoker),auto.key=TRUE,main="Smoking M/F")
```

Vectors - character vectors



Vectors - length and recycling

An important attribute of a vector is its length. To determine its length (or set it) one uses the "length" function.

```
y = 1:10

length(y)  # Length of the entire vector
[1] 10
```

When two vectors are combined in some fashion to form a third, the resulting vector takes on the length of the longest vector:

```
vec1 = 1:5

vec2 = c(1,3)

vec1 + vec2        # The shorter vector (vec2) is recycled
[1] 2 5 4 7 6

Warning message:
In vec1 + vec2 :
  longer object length is not a multiple of shorter object length
```

Vectors - naming elements

You can name the elements of the vector. Since we have been using only numeric data this might not add a lot of information. Though it actually does - especially when you work with character vectors. We use the “names” function for this.

Vectors - naming elements

You can name the elements of the vector. In this example, let’s say we have measured some heights of eight people.

```
height = c(59,70,66,72,62,66,60,60)

# Let's also create a character vector that contains the names of people
# whose heights we measured

my.names = c("Jacqueline","Frank","Babette",
             "Mario","Adriana","Esteban","Carole","Louis")

names(height) = my.names
height

Jacqueline    Frank     Babette      Mario     Adriana    Esteban     Carole      Louis
    59          70        66          72         62          66         60          60
```

Vectors - the which command

The which command allows us to determine which element number satisfies a given condition. If the element has a name then we can see it listed.

```
height > 60
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE

which(height > 60)
  Frank Babette   Mario Adriana Esteban 
      2       3       4       5       6 

height[which(height > 60)]
[1] 70 66 72 62 66

Note that the element names do not interfere with numeric evaluations

mean(height)
[1] 64.375
```

Vectors - naming elements

The "paste" function allows us to rapidly generate label names for observations in cases where we don't have original names or id.

```
new.names = paste("ID",1:8,sep="_")

new.names
 [1] "ID_1"  "ID_2"  "ID_3"  "ID_4"  "ID_5"  "ID_6"  "ID_7"  "ID_8"

names(height) = new.names

height
ID_1 ID_2 ID_3 ID_4 ID_5 ID_6 ID_7 ID_8
  59   70   66   72   62   66   60   60
```

Vectors - missing values

```
gender = c("F","M","F","M","F","M","F","M")  # Get their gender

smoker = c("N","N","Y","Y","Y","N","N","N")  # Do they smoke ?

length(gender)      # Gives current length of vector
[1] 8

length(gender) = 10  # Sets length of the vector

gender              # NA represents a missing value

 [1] "F" "M" "F" "M" "F" "M" "F" "M" NA  NA
```

Vectors - missing values

```
is.na(gender)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE

which(is.na(gender))  # Which elements contain missing values
[1]  9 10

which(!is.na(x))      # Which elements don’t have missing values
[1] 1 2 3 4 5 6 7 8

gender[!is.na(gender)]    # Get elements which aren't missing
[1] "F" "M" "F" "M" "F" "M" "F" "M"

gender[9:10] = "-"    # Set all Nas to "-" but probably should leave NAs

[1] "F" "M" "F" "M" "F" "M" "F" "M" "-" "-"
```

Vectors - some common functions

`sum()`	`prod(x)`	Sum/product of vector elements
`cumsum(x)`	`cumprod(x)`	Cumulative sum/prod of elements
`min(x)`	`max(x)`	Returns min / max of vector
`mean(x)`	`median(x)`	Returns mean / median of x
`var(x)`	`sd(x)`	Gives variance / standard deviation
`cov(x,y)`	`cor(x,y)`	Covariance / Correlation of x,y
`range(x)`		Finds range of vector
`quantile(x)`		quantiles of the vector x
`fivenum(x)`		Returns fivenum value

`length(x)`	Returns number of elements in x
`unique(x)`	Returns only unique elements of x
`rev(x)`	Returns x reversed
`sort(x)`	sorts the vector x
`match(x)`	First position of an element in x
`union(x,y)`	Union of x and y
`intersect(x,y)`	Intersection of x and y
`setdiff(x,y)`	Elements of x not in y
`setequal(x,y)`	Test if x and y contain the same elements

Vectors - some common operators

```
Relational Operators

Equal to                        ==        if (myvar == “test”) {print(“EQ”)}
                                ==        if (mnynum == 3)     {print(“EQ”)}
Not equal to                    !=        if (myvar != “test”) {print(“NE”)}
Less than or equal to           <=        if (number <= 5)     {print(“LTE”)}
Less than                       <         if (number < 10)     {print(“LT”)}
Greater than or equal to        >=        if (number >= 10)    {print(“GTE”)}
Greater than                    >         if (number > 12)     {print(“GT”)}

Boolean Operators

And                             &         if ((myvar == “test”) & (num <= 10) ){
                                             print(“Equal and less than”)
                                          }

Not                             !         if (!complete.cases(myvec)) {
                                             print(“Non complete cases”)
                                           }

Or                              |         if ((num > 3) | (num < -3)) {
                                            print(“Only one of these has to be true”)
                                          }
```

Vectors - functions

```
mean(height)        # Get the mean
[1] 64.375

sd(height)          # Get standard deviation
[1] 4.897157

min(height)         # Get the minimum
[1] 59

range(height)       # Get the range
[1] 59 72

fivenum(height)         # Tukey's summary (minimum, lower-hinge, median,
                     upper-hinge, maximum)
[1] 59 60 64 68 72

length(height)      # Vector length
[1] 8

quantile(height)        # Quantiles
  0%  25%  50%  75% 100%
  59   60   64   67   72
```

Vectors - functions

```
my.vals = rnorm(10000,20,2)

max(my.vals)
[1] 28.94032

which.max(my.vals)
[1] 2570

my.vals[ which.max(my.vals) ]
[1] 28.94032

min(my.vals)
[1] 12.49251

my.vals[ which.min(my.vals) ]
[1] 12.49251

x = 1:16
x[x %% 2 == 0]                # Find all the odd numbers from 1 to
16
[1]  2  4  6  8 10 12 14 16
```

Vectors - functions

Suppose we have x defined as follows. We want to find the sum of all the elements that are less than 5.

```
x = 0:10

x[ x < 5 ]
[1] 0 1 2 3 4

sum( x[x<5] )
[1] 10
```

Vectors - functions

Here is another vector. What if we wanted to compute the sum of the three largest values ? This would be easy by visual inspection but let's do it using some functions. This way we could use it on a vector that is possibly millions of elements long.

```
x = c(20,22,4,27,9,7,5,19,9,12)

sort(x)
 [1]  4  5  7  9  9 12 19 20 22 27

rev(sort(x))
 [1] 27 22 20 19 12  9  9  7  5  4

rev(sort(x))[1:3]
[1] 27 22 20

sum(rev(sort(x))[1:3])
[1] 69
```

Vectors - sample

The sample function takes a sample of a specified size from the elements of a given vector using either with or without replacement.

```
LETTERS      # A built-in character vector with the alphabet

[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
"R" "S" "T" "U" "V" "W" "X" "Y" "Z"

sample(LETTERS, 26, replace=F)
[1] "Q" "J" "V" "I" "H" "A" "K" "W" "U" "E" "M" "D" "G" "O" "S" "Y" "L"
"C" "Z" "B" "N" "F" "X" "T" "P" "R"

sample(LETTERS, 26, replace=TRUE)
[1] "G" "V" "C" "M" "J" "B" "K" "Q" "M" "D" "V" "H" "D" "E" "C" "O" "B"
"K" "V" "Y" "S" "C" "S" "C" "N" "J"

sample(LETTERS,8,replace=FALSE)
[1] "S" "G" "U" "M" "F" "V" "O" "B"
```

Vectors - sample

```
my.coins = c("Heads","Tails")      # Create a coin vector

sample(my.coins,5,replace=TRUE)        # 5 coin tosses
[1] "Tails" "Tails" "Heads" "Tails" "Heads"


my.vec = sample(my.coins,100,replace=TRUE)

my.vec
[1] "Heads" "Tails" "Heads" "Heads" "Tails" "Heads" "Tails" "Tails"
"Heads"
..
[100] "Tails"

table(my.vec)
my.vec
Heads Tails
   55    45

my.heads = my.vec[my.vec == "Heads"] # Gives us all the Heads

length(my.heads) / length(my.vec) * 100  # gives percentage of Heads
```

Vectors - sample

```
my.coins = c("Heads","Tails")        # Create a coin vector


# LET'S SIMULATE 1,000,000 TOSSES AND TABULATE

faircoin = table(sample(my.coins,1000000,replace=TRUE))

 Heads  Tails
500072 499928

# NOW LET'S CHEAT AND RIG THE COIN

unfaircoin = table(sample(my.coins,1000000,
                          replace=TRUE,prob=c(.75,.25)))

unfaircoin

 Heads  Tails
749811 250189
```

http://www.sigmafield.org/comment/21

Vectors - sample

```
# Does faircoin represent a fair coin ? Yes

chisq.test(faircoin, p=c(.5,.5))

    Chi-squared test for given probabilities

data:  faircoin
X-squared = 0.3069, df = 1, p-value = 0.5796


# Is unfaircoin "fair" ? Of course not

chisq.test(unfaircoin, p=c(.5,.5))

    Chi-squared test for given probabilities

data:  unfaircoin
X-squared = 249622.1, df = 1, p-value < 2.2e-16
```

Supplemental - Vectors - sample

```
# LET'S DO A SIMPLE BOOTSTRAP EXAMPLE


# Generate 1,000 values from a normal dist, mu=10

my.norm = rnorm(1000,10)

# Sample with replacement, collect means

mean(sample(my.norm,replace=TRUE))
[1] 10.01396

mean(sample(my.norm,replace=TRUE))
[1] 9.963395

..
..

mean(sample(my.norm,replace=TRUE))

Do this 1,000 times then do quantile of all the means according
to .95 confidence
```

Supplemental - Vectors - sample

```
# LET'S DO A SIMPLE BOOTSTRAP EXAMPLE

my.norm = rnorm(1000,10)  # Generate 1,000 values from a normal dist,
mu=10

# NOW USE THE REPLICATE FUNCTION TO GENERATE 1,000 MEANS

quantile(replicate(1000, mean(sample(my.norm, replace = TRUE))),
+           probs = c(0.025, 0.975))
     2.5%     97.5%
 9.927472 10.044173

# COMPARE TO T.TEST

t.test(my.norm)$conf.int
[1]  9.923378 10.044916
```

http://www.sigmafield.org/comment/21

Vectors - characters

Let's look back at the character vectors:

```
char.vec = c("here","we","are","now","in","winter")

nchar(char.vec)
[1] 4 2 3 3 2 6

rev(char.vec)   # Reverses the vector
[1] "winter" "in"     "now"    "are"    "we"     "here"


char.vec[-1]   # Omit the first element
[1] "we"     "are"    "now"    "in"     "winter"

char.vec = c(char.vec,"Its cold")   # Append the vector

[1] "here"     "we"       "are"      "now"      "in"       "winter"   "Its
cold"
```

Vectors - characters

R has support for string searching and manipulation. This is important for managing sequencing data. Let's start with some basics.

```
char.vec = c("here","we","are","now","in","winter")

grep("ar",char.vec)
[1] 3

char.vec[3]
[1] "are"

grep("ar",char.vec,value=T)
[1] "are"

grep("^w",char.vec,value=TRUE) # Words that begin with "w"
[1] "we"     "winter"

grep("w",char.vec, value=TRUE)                 # Any words that contain w
[1] "we"     "now"    "winter"

grep("w$",char.vec, value=TRUE)              # words that end with "w"
[1] "now"
```

Vectors - characters

R has support for string searching and manipulation. This is important for managing sequencing data. Let's start with some basics.

```
char.vec = c("here","we","are","now","in","winter")

char.vec[ -grep("ar",char.vec)]  # All words NOT containing "ar"
[1] "here"   "we"      "now"     "in"      "winter"


-grep("ar",char.vec)
[1] -3

char.vec[-3]

gsub("here","there",char.vec)  # We can change words too !
[1] "there"  "we"      "are"     "now"     "in"      "winter"

gsub("^w","Z",char.vec) # Replace any "w" at the beginning of a word to Z

[1] "here"   "Ze"      "are"     "now"     "in"      "Zinter"
```

Vectors - DNA character strings

We can search within a character vector for some specific characters. Let’s find all the occurrences of the “G” string:

```
dna = c("A","A","C","G","A","C","C","C","G","G","A","T","G","A","C","T","G",
"A","A","C")

dna
"A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G" "A" "A"
"C"

grep("G",dna)          # Extracts the elements numbers
[1]  4  9 10 13 17

dna[ grep("G",dna) ]
[1] "G" "G" "G" "G" "G"

OR MORE SIMPLY

grep("G",dna, value = TRUE)
[1] "G" "G" "G" "G" "G"

length(grep("G",dna, value = TRUE))  # 5 occurrences of G
[1] 5
```

Vectors - DNA character strings

We can search within a character vector for some specific characters. Let’s find all the occurrences of the “C” string:

```
set.seed(188)

dna = sample(c("A","C","G","T"),20,T)

dna
 [1] "A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G"
"A" "A" "C"

grep("C",dna, value = TRUE)
[1] "C" "C" "C" "C" "C" "C"


length(grep("C",dna, value=T))
[1] 6
```

Vectors - DNA character strings

Let's look at some special cases.

```
dna = c("A","A","C","G","A","C","C","C","G","G","A","T","G","A","C","T","G",
"A","A","C")

dna
 [1] "A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G" "A"
"A" "C"

my.str = paste(dna,collapse="")
[1] "AACGACCCGGATGACTGAAC"


length(my.str)
[1] 1                    # Not what you expected ?

my.str
[1] "AACGACCCGGATGACTGAAC"

rev(my.str)                   # What's going on ?
[1] "AACGACCCGGATGACTGAAC"

str(my.str)                      # Its now just a character string not a vector
 chr "AACGACCCGGATGACTGAAC"
```

Vectors - character strings

Let's look at some special cases.

```
my.str = paste(dna,collapse="")
[1] "AACGACCCGGATGACTGAAC"


substr(my.str,1,1)
[1] "A"

substr(my.str,1,2)
[1] "AA"

substr(my.str,1,3)
[1] "AAC"

substr(my.str,1,4)
[1] "AACG"

gsub("TG","G",my.str)
[1] "AACGACCCGGAGACGAAC"
```

Vectors - character strings

Let's look at some special cases.

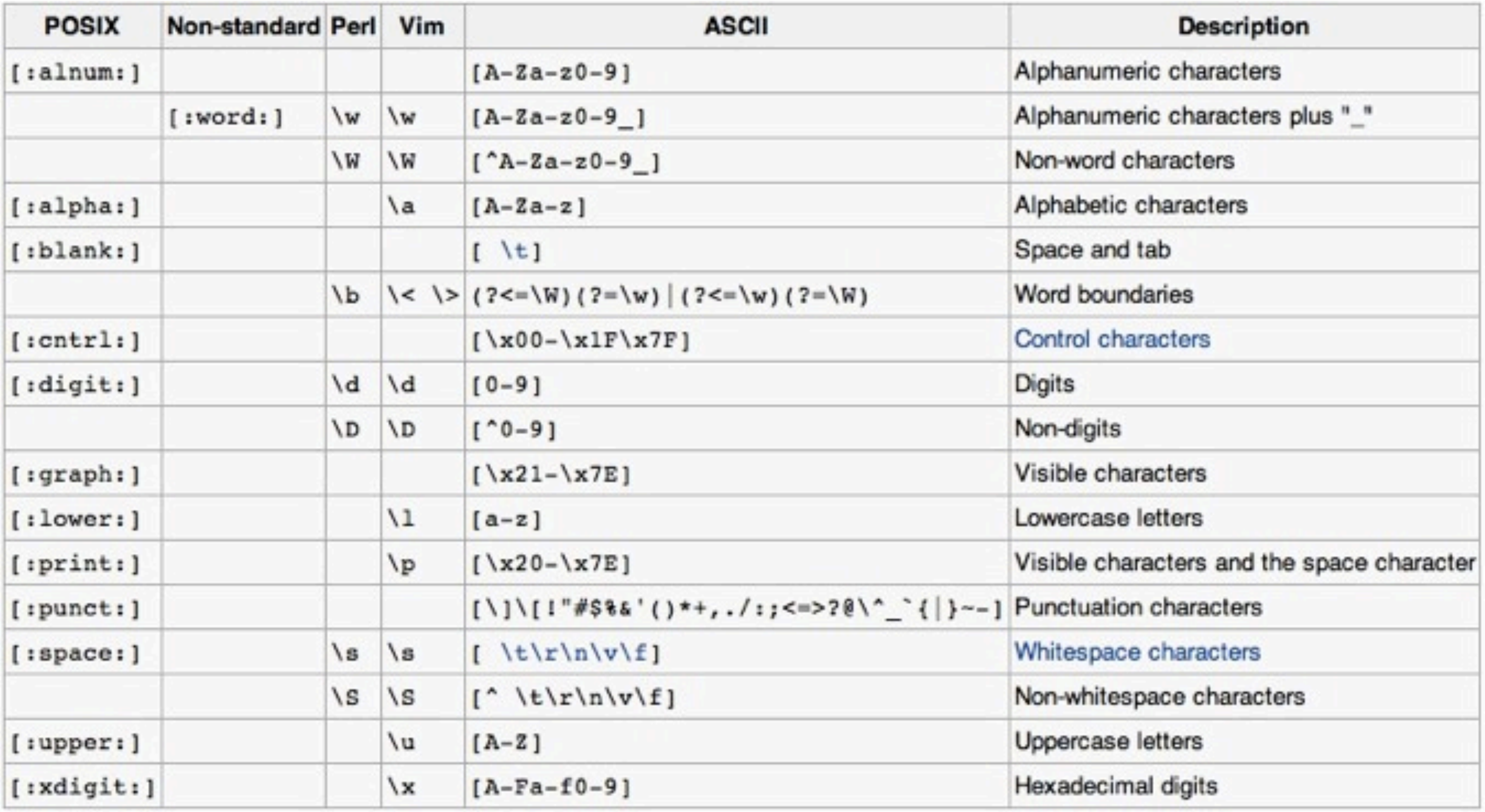
```
my.str
[1] "AACGACCCGGATGACTGAAC"

substr(my.str,2,8)
[1] "ACGACCC"

substr(my.str,2,8) = "TTTTTTT"

my.str
[1] "ATTTTTTTGGATGACTGAAC"
```

Supplemental - Vectors - character strings

Let's look at some special cases.

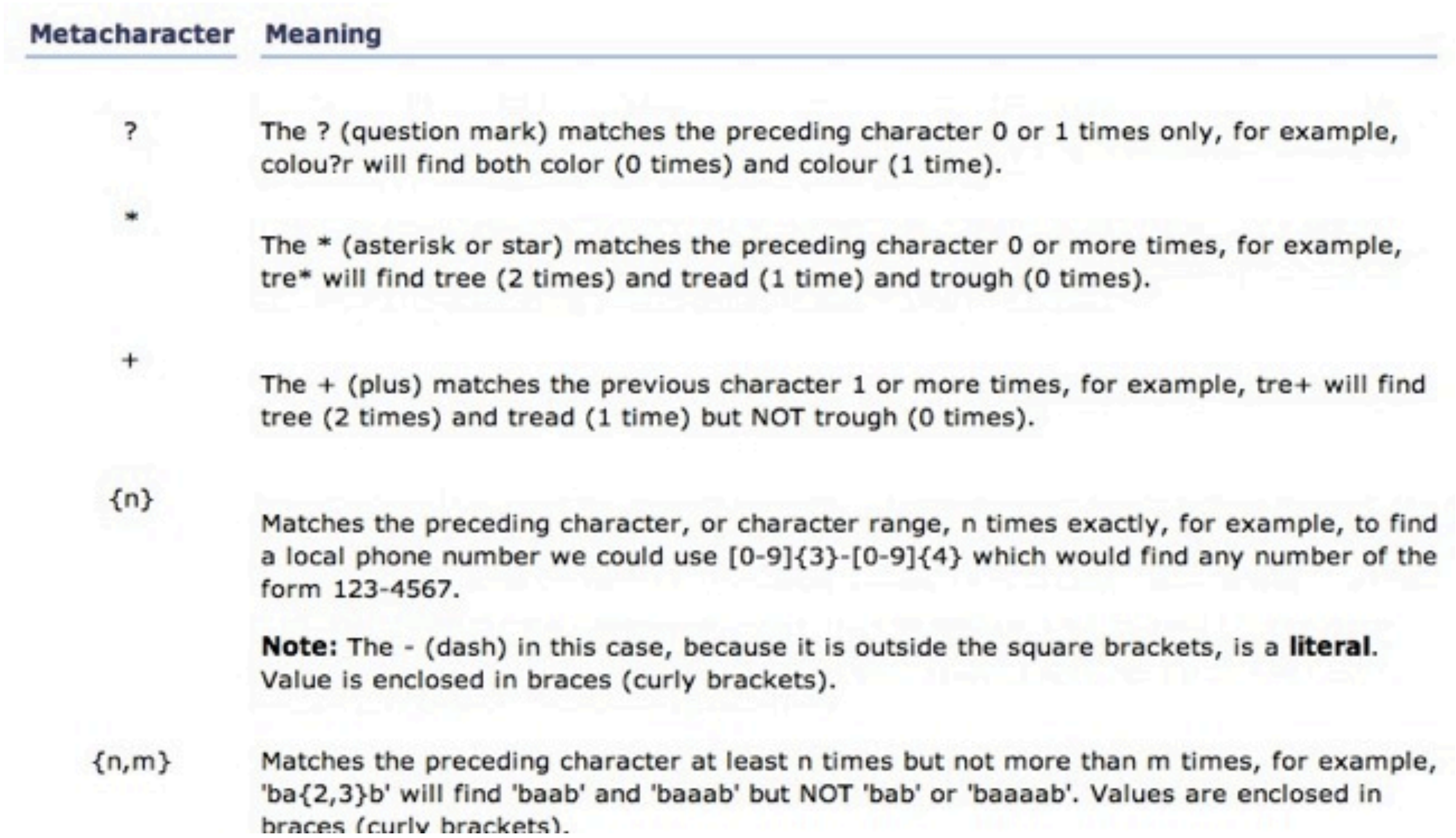
```
nchar(my.str)
[1] 20

for (ii in 1:nchar(my.str)) {
    cat(substr(my.str,ii,ii))
}
AACGACCCGGATGACTGAAC


for (ii in nchar(my.str):1) {
    cat(substr(my.str,ii,ii))
}
CAAGTCAGTAGGCCCAGCAA

# Recipe to get the "collapsed" string back into a vector with separate
elements for each letter

unlist(strsplit(my.str,""))
 [1] "A" "A" "C" "G" "A" "C" "C" "C" "G" "G" "A" "T" "G" "A" "C" "T" "G" "A"
"A" "C"
```

Vectors - characters



POSIX	Non-standard	Perl	Vim	ASCII	Description
[:alnum:]				[A-Za-z0-9]	Alphanumeric characters
	[:word:]	\w	\w	[A-Za-z0-9_]	Alphanumeric characters plus "_"
		\W	\W	[^A-Za-z0-9_]	Non-word characters
[:alpha:]			\a	[A-Za-z]	Alphabetic characters
[:blank:]				[\t]	Space and tab
		\b	\< \>	(?<=\W)(?=\w)|(?<=\w)(?=\W)	Word boundaries
[:cntrl:]				[\x00-\x1F\x7F]	Control characters
[:digit:]		\d	\d	[0-9]	Digits
		\D	\D	[^0-9]	Non-digits
[:graph:]				[\x21-\x7E]	Visible characters
[:lower:]			\l	[a-z]	Lowercase letters
[:print:]			\p	[\x20-\x7E]	Visible characters and the space character
[:punct:]				[\]\[!"#$%&'()*+,./:;<=>?@\^_`{|}~-]	Punctuation characters
[:space:]		\s	\s	[\t\r\n\v\f]	Whitespace characters
		\S	\S	[^ \t\r\n\v\f]	Non-whitespace characters
[:upper:]			\u	[A-Z]	Uppercase letters
[:xdigit:]			\x	[A-Fa-f0-9]	Hexadecimal digits

Vectors - characters



Metacharacter	Meaning
?	The ? (question mark) matches the preceding character 0 or 1 times only, for example, colou?r will find both color (0 times) and colour (1 time).
*	The * (asterisk or star) matches the preceding character 0 or more times, for example, tre* will find tree (2 times) and tread (1 time) and trough (0 times).
+	The + (plus) matches the previous character 1 or more times, for example, tre+ will find tree (2 times) and tread (1 time) but NOT trough (0 times).
{n}	Matches the preceding character, or character range, n times exactly, for example, to find a local phone number we could use [0-9]{3}-[0-9]{4} which would find any number of the form 123-4567. **Note:** The - (dash) in this case, because it is outside the square brackets, is a **literal**. Value is enclosed in braces (curly brackets).
{n,m}	Matches the preceding character at least n times but not more than m times, for example, 'ba{2,3}b' will find 'baab' and 'baaab' but NOT 'bab' or 'baaaab'. Values are enclosed in braces (curly brackets).

Matrices - Intro

R also supports **matrices**, which are objects that typically refer to a numeric array of rows and columns.

Matrices are ideal for storing information on gene expression and metabolomic data as well as many other types of scientific information.

Arrays, (matrices with dimensions greater than 2), can easily handle multi-dimensional research types.

There are two common ways to create matrices in R:

Matrices - Creating

There are two common ways to create matrices in R:

1) The "dim" command turns the vector into a matrix

```
myvec = c(1:12)

dim(myvec) = c(3,4)

myvec
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Note that columns are "filled" before rows. Note also that the requested dimension must make sense with the available number of elements.

```
dim(myvec = c(5,4))
Error in dim(myvec = c(5, 4)) :
supplied argument name 'myvec' does not match 'x'
```

Matrices - Creating

There are three common ways to create matrices in R:

2) Using the matrix command

```
mymat = matrix( c(7, 4, 2, 4, 7, 2), nrow=3, ncol=2)
mymat
     [,1] [,2]
[1,]    7    4
[2,]    4    7
[3,]    2    2
```

You can specify explicitly the nrow and ncol arguments. Note also that you can request that the rows get filled first as opposed to the columns:

```
mymat = matrix( c(7, 4, 2, 4, 7, 2), nrow=3, ncol=2, byrow=TRUE)
mymat
     [,1] [,2]
[1,]    7    4
[2,]    2    4
[3,]    7    2
```

Matrices - Naming Rows and Columns

It is useful to name the rows and columns of a matrix.

```
set.seed(123)
X = matrix(rpois(20,1.5),nrow=4)
X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    1    2    1
[2,]    2    0    1    2    0
[3,]    1    1    4    0    1
[4,]    3    3    1    3    4
```

```
Let's say that these refer to four trials and we want to label the rows
"Trial.1", "Trial.2", etc.
```

```
rownames(X) = paste("Trial",1:nrow(X),sep=".")
X
        [,1] [,2] [,3] [,4] [,5]
Trial.1    1    4    1    2    1
Trial.2    2    0    1    2    0
Trial.3    1    1    4    0    1
Trial.4    3    3    1    3    4
```

The R Book – Michael J. Crawley

Matrices - Naming Rows and Columns

And we can do something similar with the columns:

```
colnames(X) = paste("P",1:ncol(X),sep=".")

X
        P.1 P.2 P.3 P.4 P.5
Trial.1   1   4   1   2   1
Trial.2   2   0   1   2   0
Trial.3   1   1   4   0   1
Trial.4   3   3   1   3   4
```

The R Book – Michael J. Crawley

Matrices - Naming Rows and Columns

You aren't restricted to naming things with a pattern (though it is usually preferable.

```
drug.names = c("aspirin","paracetamol","nurofen","hedex","placebo")
colnames(X) = drug.names

X
        aspirin paracetamol nurofen hedex placebo
Trial.1       1           4       1     2       1
Trial.2       2           0       1     2       0
Trial.3       1           1       4     0       1
Trial.4       3           3       1     3       4
```

The R Book – Michael J. Crawley

Matrices - Indexing by Name

Names provide a convenient way to index into a matrix

```
X
        aspirin paracetamol nurofen hedex placebo
Trial.1       1           4       1     2       1
Trial.2       2           0       1     2       0
Trial.3       1           1       4     0       1
Trial.4       3           3       1     3       4

X['Trial.1',]   # Gets all columns for Trial #1

    aspirin paracetamol     nurofen       hedex     placebo
          1           4           1           2           1

# Get's the nurofen column for Trial.1

X['Trial.1','nurofen']
[1] 1
```

The R Book – Michael J. Crawley

Matrices - Indexing By Name

```
X
        aspirin paracetamol nurofen hedex placebo
Trial.1       1           4       1     2       1
Trial.2       2           0       1     2       0
Trial.3       1           1       4     0       1
Trial.4       3           3       1     3       4


X[,'nurofen']                    # Get all Trials for nurofen
Trial.1 Trial.2 Trial.3 Trial.4
      1       1       4       1

X[,'nurofen',drop=FALSE]   # Preserves matrix structure if desired
  nurofen
Trial.1       1
Trial.2       1
Trial.3       4
Trial.4       1
```

The R Book – Michael J. Crawley

Matrices - Numeric Indexing

It is more common to use numeric indexing.

```
set.seed(123)
X = matrix(rpois(9,1.5),nrow=3)
X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
[3,]    1    0    1

X[1,1]     # First row, First Column
[1] 1

X[2,2]     # Second row, Second Column
[1] 4

X[3,3]     # Third row, Third column
1] 1

diag(X)     # Ah, there is a function that gets the diagonal.
[1] 1 4 1   # Always check to see if there is already a function
            # to do what you want
```

Matrices - Indexing

You need to know how to extract information from a matrix. This can be confusing at first but becomes much easier with practice:

```
X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
[3,]    1    0    1


X[1:2,1]     # Gets First and second rows and the first column
[1] 1 2

X[1:2,2]     # Gets First and second rows and the second column
[1] 3 4

X[1:2,]      # Gets First and second rows and ALL columns
    [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
```

Matrices - Indexing

You need to know how to extract information from a matrix. This can be confusing at first but becomes much easier with practice:

```
X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
[3,]    1    0    1


X[,c(1,3)]     # Get all rows but only columns 1 and 3
      [,1] [,2]
[1,]    1    1
[2,]    2    3
[3,]    1    1


X[,-2]   # Same effect as above. Get all rows and columns except 2
     [,1] [,2]
[1,]    1    1
[2,]    2    3
[3,]    1    1
```

Matrices - Indexing Like a Vector

Keep in mind that a matrix is basically a vector with dimensions so you can index into it as if it were a vector. This might not be so intuitive at first:

```
X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
[3,]    1    0    1

X[1:4]
[1] 1 2 1 3

X >= 2
      [,1]  [,2]  [,3]
[1,] FALSE  TRUE FALSE
[2,]  TRUE  TRUE  TRUE
[3,] FALSE FALSE FALSE


X[X >= 2]   # Returns which values are greater or equal to 2
[1] 2 3 4 3

which(X >= 2)   # Returns which elements are greater or equal to 2
[1] 2 4 5 8
```

Matrices - Indexing Like a Vector

Keep in mind that a matrix is basically a vector with dimensions so you can index into it as if it were a vector. This might not be so intuitive at first:

```
X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
[3,]    1    0    1

X %% 2 == 0          # Returns a logical matrix
      [,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,]  TRUE  TRUE FALSE
[3,] FALSE  TRUE FALSE


> X[ X %% 2 == 0 ]   # Finds the actual element values
[1] 2 4 0
```

Matrices - Indexing Like a Vector

Keep in mind that a matrix is basically a vector with dimensions so you can index into it as if it were a vector. This might not be so intuitive at first:

```
X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
[3,]    1    0    1


X[X %%  2 == 0] = 99

X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    99   99   3
[3,]    1    99   1
```

Matrices - Indexing Like a Vector

There are two functions called row and col that return the numeric row and column, respectively of the matrix. Kinda weird but useful. An example is in order.

```
X
     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    3
[3,]    1    0    1

row(X)
     [,1] [,2] [,3]
[1,]    1    1    1    # The values correspond to the actual row number
[2,]    2    2    2
[3,]    3    3    3

col(X)                  # The values correspond to the actual col number
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```

Matrices - Indexing Like a Vector

There are two functions called row and col that return the numeric row and column, respectively of the matrix. Kinda weird but useful. An example is in order.

```
row(X) == col(X)

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


X[row(X) == col(X)]
[1] 1 4 1

X[row(X) == col(X)] = 0  # Put zeroes in the diagonal

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

Matrices - Adding Rows and Columns

Sometimes we need to add rows and columns to a matrix. There are two commands to do this: rbind and cbind.

```
set.seed(123)
X = matrix(rpois(9,1.5),nrow=3)
colnames(X) = c("aspirin","paracetamol","nurofen")
rownames(X) = paste("Trial",1:3,sep=".")
```

```
rbind(X,Trial.4=c(4,7,5))
        aspirin paracetamol nurofen
Trial.1       1           3       1
Trial.2       2           4       3
Trial.3       1           0       1
Trial.4       4           7       5
```

Matrices - Adding Rows and Columns

Binding columns works pretty much the same way:

```
X
        aspirin paracetamol nurofen
Trial.1       1           3       1
Trial.2       2           4       3
Trial.3       1           0       1


rowSums(X)
Trial.1 Trial.2 Trial.3
      5       9       2

cbind(X,rowsums = rowSums(X))
        aspirin paracetamol nurofen rowsums
Trial.1       1           3       1       5
Trial.2       2           4       3       9
Trial.3       1           0       1       2
```

Matrices - Doing Calculations

Let’s look at some examples involving calculations on matrices:

```
set.seed(123)

X = matrix(rpois(9,1.5),nrow=3)

colnames(X) = c("aspirin","paracetamol","nurofen")

rownames(X) = paste("Trial",1:3,sep=".")

X
        aspirin paracetamol nurofen
Trial.1       1           3       1
Trial.2       2           4       3
Trial.3       1           0       1

mean(X[,3])  # Mean of the 3rd column
[1] 1.666667

var(X[3,]) # Variance of the 3rd row
[1] 0.3333333
```

Matrices - Doing Calculations

Let’s look at some examples involving calculations on matrices. But there are some general functions to help with this kind of thing:

```
X
        aspirin paracetamol nurofen
Trial.1       1           3       1
Trial.2       2           4       3
Trial.3       1           0       1

rowSums(X)
Trial.1 Trial.2 Trial.3
      5       9       2

colSums(X)
aspirin paracetamol     nurofen
   4           7           5
```

Maybe columns represent protein expression and you are trying to determine if there are differences between the mean expression levels.

The R Book – Michael J. Crawley

Matrices - Doing Calculations

But there are some general functions to help with this kind of thing:

```
rowMeans(X)
  Trial.1   Trial.2   Trial.3
1.6666667 3.0000000 0.6666667

colMeans(X)
    aspirin paracetamol     nurofen
   1.333333    2.333333    1.666667

colMeans(X)[3]
 nurofen
1.666667
```

These are fast and can work on very large matrices. Though be careful if you have missing values in your data.

The R Book - Michael J. Crawley

Matrices - Doing Calculations

These are fast though be careful if you have missing values in your data.

```
X[1,2] = NA
X
        aspirin paracetamol nurofen
Trial.1       1          NA       1
Trial.2       2           4       3
Trial.3       1           0       1



colMeans(X)
    aspirin paracetamol     nurofen
   1.333333          NA    1.666667

colMeans(X, na.rm=TRUE)
    aspirin paracetamol     nurofen
   1.333333    2.000000    1.666667
```

The R Book – Michael J. Crawley

Matrices - Doing Calculations - apply

Its worth pointing out that you can do similar things with the apply function. It allows you to plug in any function - not just the mean function.

```
X
        aspirin paracetamol nurofen
Trial.1       3           1       3
Trial.2       2           2       2
Trial.3       2           0       5

apply(X,1,summary)                 # 1 is for rows
        Trial.1 Trial.2 Trial.3
Min.      1.000     2.0  0.0000
1st Qu.   1.000     2.5  0.5000
Median    1.000     3.0  1.0000
Mean      1.667     3.0  0.6667
3rd Qu.   2.000     3.5  1.0000
Max.      3.000     4.0  1.0000

apply(X,2,summary)                  # 2 is for columns
        aspirin paracetamol nurofen
Min.      1.000       0.000   1.000
1st Qu.   1.000       1.500   1.000
Median    1.000       3.000   1.000
Mean      1.333       2.333   1.667
3rd Qu.   1.500       3.500   2.000
Max.      2.000       4.000   3.000
```

The R Book – Michael J. Crawley

Matrices - Linear Algebra

R supports common linear algebra operations also.

```
A = matrix(c(1,3,2,2,8,9),3,2)
A
     [,1] [,2]
[1,]    1    2
[2,]    3    8
[3,]    2    9
```

```
t(A)
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    8    9
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix}^{T} = \begin{bmatrix} 1 & 3 & 2 \\ 2 & 8 & 9 \end{bmatrix}$$

http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf

Matrices - Linear Algebra

```
A
     [,1] [,2]
[1,]    1    2
[2,]    3    8
[3,]    2    9

B = matrix(c(5,8,4,2),2,2)

A %*% B
     [,1] [,2]
[1,]   21    8
[2,]   79   28
[3,]   82   26
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} 5 & 4 \\ 8 & 2 \end{bmatrix} = \left[\begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} 5 \\ 8 \end{bmatrix} : \begin{bmatrix} 1 & 2 \\ 3 & 8 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix} \right]$$

$$= \begin{bmatrix} 1 \cdot 5 + 2 \cdot 8 & 1 \cdot 4 + 2 \cdot 2 \\ 3 \cdot 5 + 8 \cdot 8 & 3 \cdot 4 + 8 \cdot 2 \\ 2 \cdot 5 + 9 \cdot 8 & 2 \cdot 4 + 9 \cdot 2 \end{bmatrix} = \begin{bmatrix} 21 & 8 \\ 79 & 28 \\ 82 & 26 \end{bmatrix}$$

http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf

Matrices - Linear Algebra

The inverse of a n x n matrix A is the matrix B (which is also n x n) that when multiplied by A gives the identity matrix.

```
A = matrix(1:4,2,2)
A
     [,1] [,2]
[1,]    1    3
[2,]    2    4

B = solve(A)

B
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5

A %*% B            # We get the identity matrix
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf

Matrices - Linear Algebra

Suppose you have the following system of equations. This can be represented as:

$$\begin{aligned} x_1 + 3x_2 &= 7 \\ 2x_1 + 4x_2 &= 10 \end{aligned}$$

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 7 \\ 10 \end{bmatrix} \quad \text{i.e. } Ax = b$$

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

b = c(7,10)
x = solve(A) %*% b
x
     [,1]
[1,]    1
[2,]    2
```

Since $A^{-1}A = I$ and since $Ix = x$ we have

$$x = A^{-1}b = \begin{bmatrix} -2 & 1.5 \\ 1 & -0.5 \end{bmatrix} \begin{bmatrix} 7 \\ 10 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf

Supplemental - Matrices - Linear Algebra

```
B
     [,1] [,2]
[1,]    5    4
[2,]    8    2

diag(B)                # Fetches the diagonal
[1] 5 2

diag(c(1,2,3))         # Creates a matrix with 1,2,3 on the diagonal
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3

diag(1,4)              # Creates a 4 x 4 Indentity matrix
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

http://bendixcarstensen.com/APC/linalg-notes-BxC.pdf

Supplemental - Matrices - eigen values

Eigen values and vectors show up a lot in statistics - like with Principal Components Analysis.

```
my.wines = read.csv("http://www.bimcore.emory.edu/wine.csv",header=T)

my.scaled.wines = scale(my.wines)   # Scale the data

my.cov = cov(my.scaled.wines)       # Get the covariance matrix

my.eigen = eigen(my.cov)            # Now find the eigen values/vectors

options(digits=3)

my.eigen                            # Check out the Eigen values and vectors
$values
[1]  4.76e+00  1.81e+00  3.53e-01  7.44e-02  3.73e-16 -2.61e-16 -2.99e-16

$vectors
        [,1]    [,2]     [,3]    [,4]      [,5]      [,6]      [,7]
[1,] -0.3965  0.1149  0.80247  0.0519 -1.46e-01  0.00e+00 -4.02e-01
[2,] -0.4454 -0.1090 -0.28106 -0.2745  4.84e-01 -5.18e-01 -3.64e-01
[3,] -0.2646 -0.5854 -0.09607  0.7603  5.41e-16  3.75e-16 -1.16e-15
[4,]  0.4160 -0.3111  0.00734 -0.0939  3.24e-01  4.88e-01 -6.15e-01
[5,] -0.0485 -0.7245  0.21611 -0.5474 -2.16e-01 -3.23e-02  2.80e-01
[6,] -0.4385  0.0555 -0.46576 -0.1687 -5.67e-01  3.86e-01 -2.97e-01
[7,] -0.4547  0.0865  0.06430 -0.0835  5.20e-01  5.85e-01  4.01e-01

$loadings = my.eigen$vectors
```

Supplemental - Matrices - eigen values

Eigen values and vectors show up a lot in statistics - like with Principal Components Analysis.

The loadings are the principal components

```
loadings = my.eigen$vectors
```

The scores are the product of the matrix multiplication between the scaled.wines and the loadings. This takes the original wine data and re-expresses it in terms of the “principal components”.

```
scores = my.scaled.wines %*% loadings
```

Supplemental - Matrices - Cluster Analysis

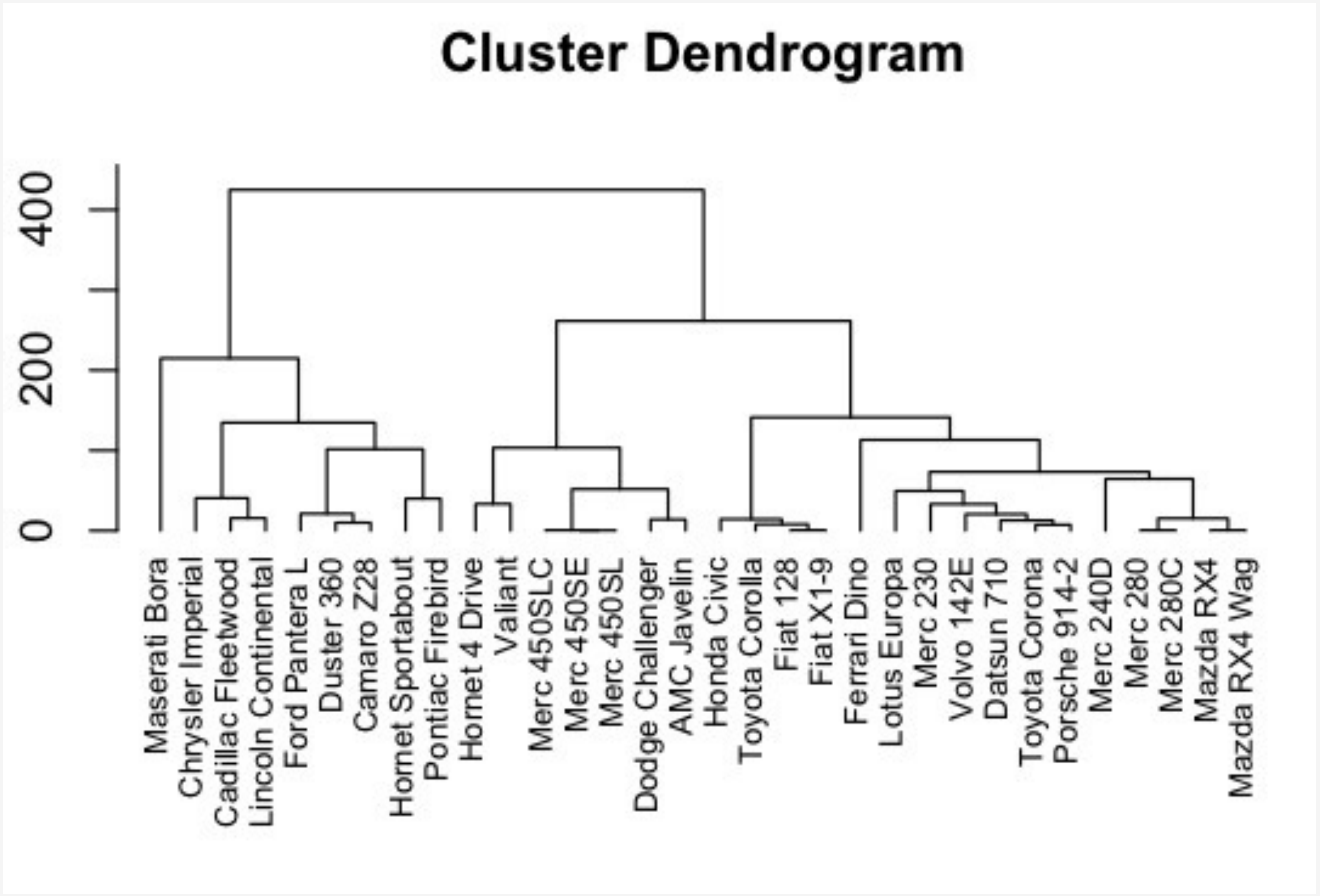
Matrices are also used a lot in cluster analysis. Let's look at a matrix of 32 cars and attempt to cluster them according to their various attributes such as MPG, Number of Cylinders, Gears, Weight, etc. This data set (mtcars) is internal to R so you can refer to it easily.

```
mtcars
                    mpg cyl  disp  hp drat   wt qsec vs  am gear carb
Mazda RX4          21.0   6 160.0 110 3.90 2.62 16.5  0   1    4    4
Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.88 17.0  0   1    4    4
Datsun 710         22.8   4 108.0  93 3.85 2.32 18.6  1   1    4    1
Hornet 4 Drive     21.4   6 258.0 110 3.08 3.21 19.4  1   0    3    1
Hornet Sportabout  18.7   8 360.0 175 3.15 3.44 17.0  0   0    3    2
..

..
```

We first compute a distance between the rows and then cluster them.

```
hc <- hclust(dist(mtcars[,2:11]))  # The first column is a label.
plot(hc, hang = -1,cex=0.7)
```

Supplemental - Matrices - Cluster Analysis



Supplemental - Matrices - Alternative Ways to

We can create matrices using the replicate command. This approach is useful if you are trying to capture the results of repeated sampling activity like when bootstrapping. In the simplest case here is an example. This generates a 4 column matrix with 5 rows. Each time we generate a new column we are effectively getting a new sample of data from a normal distribution.

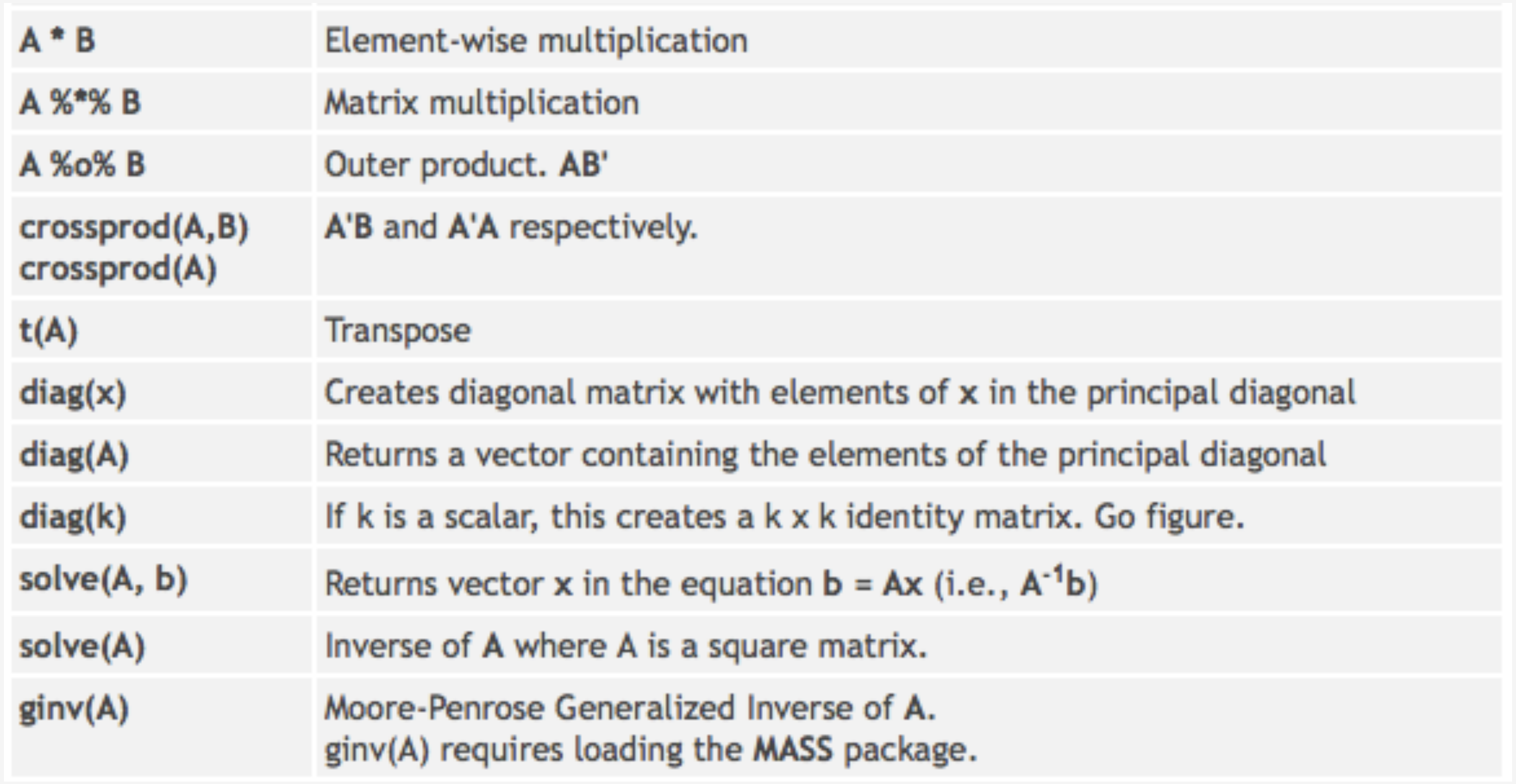
```
replicate(4,rnorm(5))
             [,1]       [,2]       [,3]        [,4]
[1,] -1.181720384  0.2717525 -1.4716542  2.26654104
[2,]  0.268970133  0.3423814  0.9553185  0.07749788
[3,]  0.007413904 -1.2102476  0.2273662 -0.46742459
[4,]  1.726961040  0.9977138  2.0491924  0.77174367
[5,]  0.950821481 -1.8599874 -0.8587209  0.95906263

some.population = rnorm(1000)

replicate(4,sample(some.population, 5, replace=TRUE))
           [,1]        [,2]       [,3]       [,4]
[1,] -0.4680211  0.27727612 -0.5346220 0.94161600
[2,]  0.3138391  0.36105532  0.1108916 0.35186402
[3,] -1.8416441 -0.05812402  1.3535505 0.05288187
[4,] -0.9483933 -0.24572418  1.6950778 1.30636068
[5,]  1.0369327 -0.66983941  0.3055545 1.57318148
```

Supplemental - Matrices - Alternative Ways to

So if we have done a repeated sample from a population we could then process each column to see if a hypothesized mean feel into a confidence interval. So in this case we kind of know what the true mean is but let's pretend we don't.

```
set.seed(177)

some.pop = rnorm(1000)

mean(some.pop)
[1] -0.01982588

my.boot = replicate(4, sample(some.pop, 5, replace=T))

            [,1]       [,2]       [,3]       [,4]
[1,] -0.06820582 -0.1160524  0.3222247  1.1403365
[2,] -0.96508173  0.4916324 -0.9598382 -1.9968074
[3,]  0.26232581 -0.7655528  0.9046675 -0.3021725
[4,] -0.02585224 -1.1985142 -1.0101444 -0.3309738
[5,]  0.92775589 -1.1999768 -1.2637646  0.5954983
```

Supplemental - Matrices - Some Functions

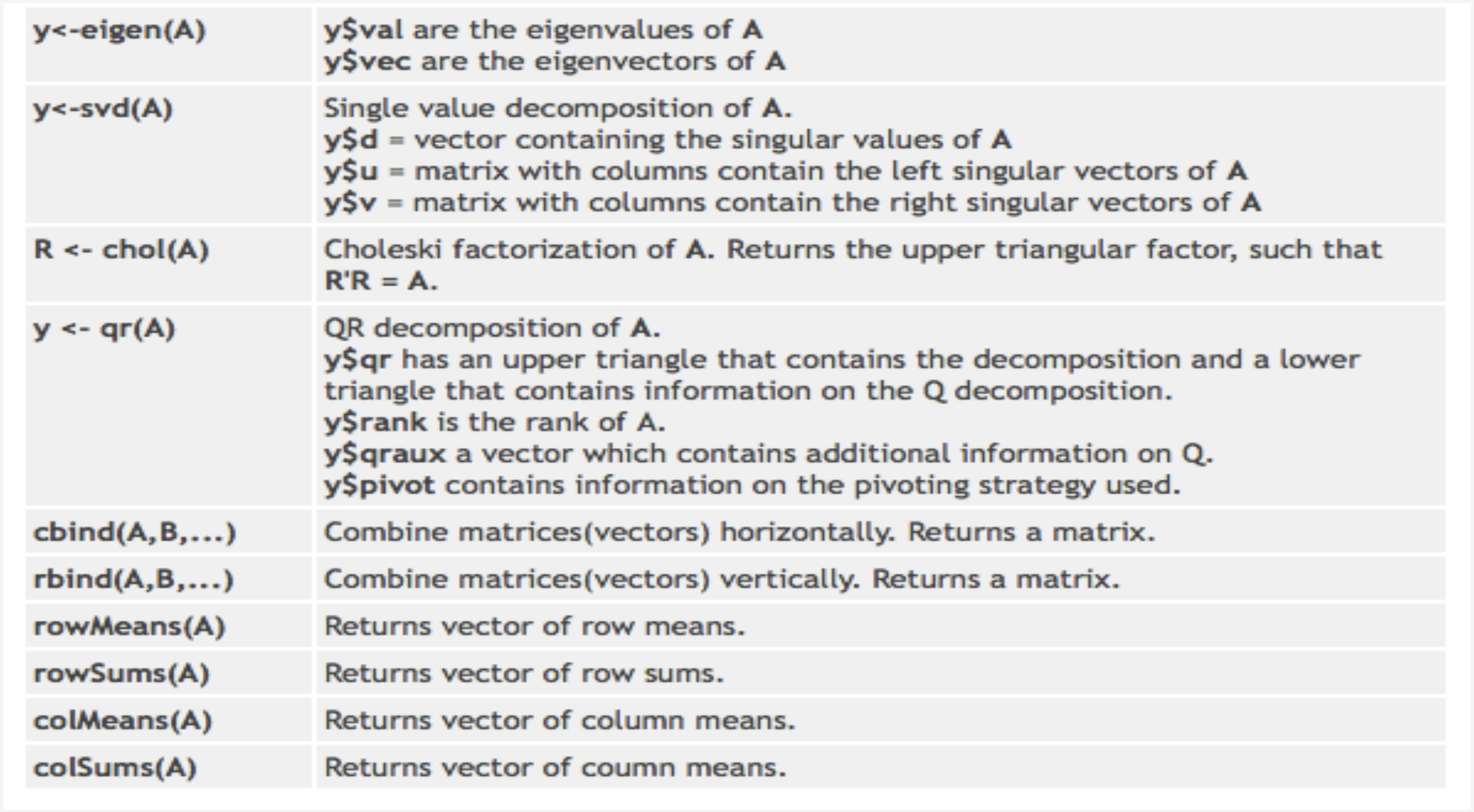


A * B	Element-wise multiplication
A %*% B	Matrix multiplication
A %o% B	Outer product. AB'

crossprod(A,B)
crossprod(A)
	A'B and A'A respectively.

t(A)	Transpose
diag(x)	Creates diagonal matrix with elements of **x** in the principal diagonal
diag(A)	Returns a vector containing the elements of the principal diagonal
diag(k)	If k is a scalar, this creates a k x k identity matrix. Go figure.
solve(A, b)	Returns vector **x** in the equation **b** = **Ax** (i.e., $A^{-1}b$)
solve(A)	Inverse of **A** where A is a square matrix.
ginv(A)	Moore-Penrose Generalized Inverse of **A**. ginv(A) requires loading the **MASS** package.

Supplemental - Matrices - Some Functions



y<-eigen(A)	**y$val** are the eigenvalues of **A** **y$vec** are the eigenvectors of **A**
y<-svd(A)	Single value decomposition of **A**. **y$d** = vector containing the singular values of **A** **y$u** = matrix with columns contain the left singular vectors of **A** **y$v** = matrix with columns contain the right singular vectors of **A**
R <- chol(A)	Choleski factorization of **A**. Returns the upper triangular factor, such that **R'R = A**.
y <- qr(A)	QR decomposition of **A**. **y$qr** has an upper triangle that contains the decomposition and a lower triangle that contains information on the Q decomposition. **y$rank** is the rank of A. **y$qraux** a vector which contains additional information on Q. **y$pivot** contains information on the pivoting strategy used.
cbind(A,B,...)	Combine matrices(vectors) horizontally. Returns a matrix.
rbind(A,B,...)	Combine matrices(vectors) vertically. Returns a matrix.
rowMeans(A)	Returns vector of row means.
rowSums(A)	Returns vector of row sums.
colMeans(A)	Returns vector of column means.
colSums(A)	Returns vector of coumn means.