the R environment

The R language is an integrated suite of software facilities for:

- Data Handling and storage
- Matrix Math: Manipulating matrices, vectors, and arrays
- Statistics: A large, integrated set of tools for data analysis
- Graphics: Graphical facilities for data analysis and display
- Programming: Powerful programming language ('S')

Basic Syntax

R Prompt:	>
Continuation prompt:	+
Assignment Operator (->)	>value <- 2*4
Last Expression(.Last.value)	> 2 * 4
	[1] 8
	> value <- .Last.value
Removing objects	rm(value)

Legal R names: Any combination of letters, numbers, periods(.) but MUST NOT start with a number. CASE SENSITIVE

Spaces: R will ignore extra spaces between object names and operators, but spaces count when they are inside character strings. e.g. "Hello world" != "Hello world"

Finding objects: R looks for objects in a sequence of places known as the search path. The search path is a sequence of environments beginning with the Global Environment. (more on this later)

R works on Objects

R object types

Vector: a one-dimensional array of arbitrary **length**.

Matrix: a two-dimensional array with an arbitrary **number of rows and columns**.

Array: as a matrix, but of **arbitrary dimension** (i.e., more than 2).

Data frame: a set of data organized similarly to a matrix. However each column of the data frame may contain its own type of data. Columns typically correspond to variables in a statistical study, while rows correspond to observations of these variables.

Function: a set of commands that are packaged into a unit with defined input and output (I/O is not necessary, though).

List: an arbitrary collection of other R objects (which may include other lists).

```
> x <- cbind(a=1:3, pi=pi) # simple
                           # matrix w/ dimnames
> x
     a       pi
[1,] 1 3.141593
[2,] 2 3.141593
[3,] 3 3.141593

> class(x)
[1] "matrix"

> attributes(x)
$dim
[1] 3 2

$dimnames
$dimnames[[1]]
NULL

$dimnames[[2]]
[1] "a"  "pi"

> attributes(x) <- NULL
> x # now just a vector of length 6

[1] 1.000000 2.000000 3.000000 3.141593
   3.141593 3.141593

> class(x)      # vector is default mode
[1] "numeric"
```

Four Atomic Data Types in R:

- Numeric

```
> value <- 605
> value
[1] 605
```

- Character

```
> string <- "Hello World"
> string
[1] "Hello World"
```

- Logical

```
> 2 < 4
[1] TRUE
```

- Complex number

```
> cn <- 2 + 3i
> cn
[1] 2+3i
```

The attribute of an object becomes important when manipulating objects.

All objects have two attributes, the **mode** and their **length** (number of vector elements).

```
> mode(value)
[1] "numeric"
> length(value)
[1] 1

> mode(string)
[1] "character"
> length(string)
[1] 1

> mode(2<4)
[1] "logical"

> mode(cn)
[1] "complex"
> length(cn)
[1] 1

> mode(mean)
[1] “function”
```

What is it?

So "Atomic" datatypes refer to the single element. (not able to break it down any further).

When working with a new package, you want to know what kind of object you are dealing with:

mode : the atomic data type (**class** = **mode** if object is atomic)

class : refers to the classes that the object inherits (which methods will be used depends on the class of the object)

All objects have **mode** and **length** (everything is a vector in R)

Derived objects have **class** and **attributes**

Missing, Indefinite, and Infinite Values

NA = missing value

is.na() function checks for missing values

```
> value <- c(3,6,23,NA)

> is.na(value)

[1] FALSE FALSE FALSE TRUE

> any(is.na(value))

[1] TRUE

> na.omit(value)

[1] 3 6 23
```

Indefinite and Infinite values (Inf, -Inf and NaN) can also be tested using the is.finite, is.infinite, is.nan and is.number functions in a similar way as shown above.

These values come about usually from a division by zero or taking the log of zero.

```
> value1 <- 5/0

> value2 <- log(0)

> value3 <- 0/0

> cat("value1 = ",value1," value2 = ",value2,

" value3 = ",value3,"\n")

value1 = Inf value2 = -Inf value3 = NaN
```

Probability Distributions in R

Table 3: Probability Distributions in R

R Function	Distribution	Parameters	Package
`beta`	Beta	shape1,shape2	stats
`binom`	Binomial	size,prob	stats
`cauchy`	Cauchy	location,scale	stats
`chisq`	(non-central) Chi-squared	df,ncp	stats
`dirichlet`	Dirichlet	alpha	MCMCpack
`exp`	Exponential	rate	stats
`f`	F	df1,df2	stats
`gamma`	Gamma	shape,rate	stats
`geom`	Geometric	prob	stats
`gev`	Generalized Extreme Value	xi,mu,sigma	evir
`gpd`	Generalized Pareto	xi,mu,beta	evir
`hyper`	Hypergeometric	m,n,k	stats
`invgamma`	Inverse Gamma	shape,rate	MCMCpack
`iwish`	Inverse Wishart	v,S	MCMCpack
`logis`	Logistic	location,scale	stats
`lnorm`	Log Normal	meanlog,sdlog	stats
`multinom`	Multinomial	size,prob	stats
`mvnorm`	Multivariate Normal	mean,sigma	mvtnorm
`mvt`	Multivariate-t	sigma,df	mvtnorm
`nbinom`	Negative Binomial	size,prob	stats
`norm`	Normal	mean,sd	stats
`pois`	Poisson	lambda	stats
`signrank`	Wilcoxon Signed Rank Statistic	n	stats
`t`	Student-t	df	stats
`unif`	Uniform	min,max	stats
`weibull`	Weibull	shape,scale	stats
`wilcox`	Wilcoxon Rank Sum Statistic	m,n	stats
`wish`	Wishart	v,S	MCMCpack

In R, each distribution has a name prefix indicating:

- p: probabilities (distribution functions)
- q: quantiles (percentage points)
- d: density functions (probability for discrete RVs)
- r: random (or simulated) values

```
> norm.vals1 <- rnorm(n=10)
> norm.vals2 <- rnorm(n=100)
> norm.vals3 <- rnorm(n=1000)
> norm.vals4 <- rnorm(n=10000)
# set up plotting region
> par(mfrow=c(2,2))
# produce plots
> hist(norm.vals1,main="10 RVs")
> hist(norm.vals2,main="100 RVs")
> hist(norm.vals3,main="1000 RVs")
> hist(norm.vals4,main="10000 RVs")
```

Basic Operators in R

Table 1: Arithmetic Operators

Operator	Description	Example
+	Addition	`> 2+5` `[1] 7`
−	Subtraction	`> 2-5` `[1] -3`
×	Multiplication	`>2*5` `[1] 10`
/	Division	`> 2/5` `[1] 0.4`
∧	Exponentiation	`> 2∧5` `[1] 32`
%/%	Integer Divide	`> 5%/%2` `[1] 2`
%%	Modulo	`> 5%%2` `[1] 1`

Table 2: Logical Operators

Operator	Description	Example

==	Equals	`> value1`
		`[1] 3 6 23`
		`> value1==23`
		`[1] FALSE FALSE TRUE`

!=	Not Equals	`> value1 != 23` `[1] TRUE TRUE FALSE`
<	Less Than	`> value1 < 6` `[1] TRUE FALSE FALSE`
>	Greater Than	`> value1 > 6` `[1] FALSE FALSE TRUE`
<=	Less Than or Equal To	`> value1 <= 6` `[1] TRUE TRUE FALSE`
>=	Greater Than or Equal To	`> value1 >= 6` `[1] FALSE FALSE TRUE`

&	Elementwise And	`> value2`
		`[1] 1 2 3`
		`> value1==6 & value2 <= 2`
		`[1] FALSE TRUE FALSE`

|	Elementwise Or	`> value1==6 | value2 <= 2` `[1] TRUE TRUE FALSE`

&&	Control And	`> value1[1] <- NA`
		`> is.na(value1) && value2 == 1`
		`[1] TRUE`

||	Control Or	`> is.na(value1) || value2 == 4` `[1] TRUE`
xor	Elementwise Exclusive Or	`> xor(is.na(value1), value2 == 2)` `[1] TRUE TRUE FALSE`
!	Logical Negation	`> !is.na(value1)` `[1] FALSE TRUE TRUE`

When applied to vectors or matrices, these operators work on an element-by-element basis (except control logical operators)

Lists

Like a data frame, but more irregular. Lists can be composed of vectors, each of different type and length.

Often used for returning output.

Created using **list()**

```
> L1 <- list(x = sample(1:5, 20, rep=T),  y = rep(letters[1:5], 4), z = rpois(20, 1))
> L1
$x
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
$y
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b"
[13] "c" "d" "e" "a" "b" "c" "d" "e"
$z
[1] 1 3 0 0 3 1 3 1 0 1 2 2 0 3 1 1 0 1 2 0
```

Accessing List Elements

By **name** of that component (if names are assigned) or by **number**

- using subsetting ([[ ]]) NOTE: Double brackets
- the extraction operator ($).

```
> L1[["x"]]
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1$x
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1[[1]]   # extracts the first list element (here, x)
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

To extract a sublist (i.e., part of a list as opposed to a vector which you can think of as a single component of a list), we use single brackets. The following example extracts the first component only.

```
> L1[1]
$x
[1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

Working with Lists

Length:

```
> length(L1)  #  is number of components

[1] 3
```

Names of lists can also be altered in a similar way to that for data frames.

```
> names(L1) <- c("Item1","Item2","Item3")
```

Above assigns names to elements of L1.

Indexing: also similar to data frames:

```
> L1$Item1[L1$Item1>2]  # L1$item1's > 2

[1] 4 3 4 5 3 3 3 5 3 3 5
```

Joining two lists: c() or append():

```
> L2 <- list(x=c(1,5,6,7),
+  y=c("apple","orange","melon","grapes"))
> c(L1,L2)

$Item1
[1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1

$Item2
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b"
[13]"c" "d" "e" "a" "b" "c" "d" "e"

$Item3
[1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2

$x
[1] 1 5 6 7

$y
[1] "apple" "orange" "melon" "grapes"
```

Working with Lists

Append Function:

```
> append(L1,L2,after=2)

$Item1
[1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1

$Item2
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a"
[12]"b" "c" "d" "e" "a" "b" "c" "d" "e"

$x
[1] 1 5 6 7

$y
[1] "apple" "orange" "melon" "grapes"

$Item3
[1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2
```

Adding elements to a list can be achieved by

- adding a new component name:

```
> L1$Item4 <-
    c("apple","orange","melon","grapes")

# alternative way

> L1[["Item4"]] <-
    c("apple","orange","melon","grapes")
```

- adding a new component element, whose index is greater than the length of the list

```
L1[[4]] <-
    c("apple","orange","melon","grapes")

> names(L1)[4] <- c("Item4")
```

Vectorized Calculations are *fast* in R (but what is that?)

Operate on the entire vector, matrix, array, dataframe, or list, *all at once!*

Vectorised Calculations

Simple matrix math:

```
x <- matrix( c(1:4), nrow=2)
x + 1
```

Apply functions:

```
apply(X, MARGIN, FUN, ...)
```

if you want to apply more than one function, then you have to write your own function and embed it

Vectorised Calculations

apply operates on successive sections of an array

apply(array or vector, MARGIN(which dimension?), function, optional arguments to function)

```
iris[1:4,]                          # the data
apply(iris[,-5],2,mean)# iris[,-5] takes the iris dataset, but drops
                                    #      the 5th column
                                # the "2" is the row index and function= "mean"
                                # so we're taking the mean over rows
```

flavors

tapply operates on a "ragged" array (i.e., groups of different sizes, for instance, one vector subscripted by a factor)

tapply(array or vector, indexing vector, function)

```
spnames <- iris[,5]
tapply(iris[,4], spnames, mean) # take 4th column of data
...                             # iris[,5] contains species names
                                # so we're taking the mean by species
for(i in 1:4) {print(tapply(iris[,i], iris[,5],
mean)) }                                    # why the print statement?
```

Vectorised Calculations

```
for(i in 1:4) {print(tapply(iris[,i], iris[,5],
mean)) }          # the print statement is needed because we're inside
...               # of a loop -- any calculations will go on, but nothing will
                  # display on the screen unless we "print" to screen
```

these all operate on components of a list or vector

apply # apply functions over a margin of an array (row, column, etc.)

lapply # returns a list

sapply # returns friendly vector or array (will return a matrix or vector if
appropriate

mapply # works on multiple arguments

aggregate # compute summary statistics over subsets of the data

Why bother with the apply family when I know/understand looping???

Vectorized versus loops

Example: a silly program that subtracts one if less than 5, and divides by itself if greater than 5.

traditional program:

```
x <- 1:10; z <- NULL     # set up
for (i in 1:length(x)) # looping over the length of your vector is
{                                  # very convenient -- you don't have to manually
                                   # figure out how long your loop should be
  if (x[i]<5) {z <- c(z,x[i]-1)} # conditionally modifies elements of x
  else { z <- c(z,x[i]/x[i])}
}
```

Same thing, but in two-steps: "test" function to decide which method, then apply to all elements of x vector

```
x <- 1:10; z <- NULL
test <- function(x) {if (x<5) { x-1 } else { x/x }}
apply(as.matrix(x), 1, test)
```

simplest apply version:

```
apply(as.matrix(x), 1, test <- function(x) { if (x<5) {x-1}
```

Vectorized versus loops

Example: a silly program that subtracts one if less than 5, and divides by itself if greater than 5.

Finally, shortest apply version: define the function inside the call to apply

```
apply(as.matrix(x), 1, test <- function(x) { if (x<5) {x-1}
else {x/x} })
```

function is embedded, so it doesn't need a name because we won't ever need to refer to it by name. just used once and then it "disappears"

Timing: which is more efficient?

system.time returns a numeric vector of length 5: user cpu, system cpu, elapsed, subproc1, subproc2 times (latter are usually zero).

```
> system.time(for (i in 1:length(x)) {if (x[i]<5) {z <-
c(z,x[i]-1)} else { z <- c(z,x[i]/x[i])}})[[3]] #elapsed

          [1] 0.195
```

```
>
system.time(apply(as.matrix(x)
, 1, test <- function(x) { if
(x<5) { x-1 } else { x/x } }))
[[3]]

          [1] 0.001
```

nearly 200x faster!

is this consistently true? ->