

Introduction to R language

Khalil MASMOUDI

Based on slides by R. Peng - Johns Hopkins Bloomberg School of Public Health

Historical Notes

- 1991: Created in New Zealand by Ross Ihaka and Robert Gentleman. Their experience developing R is documented in a 1996 JCGS paper.
- 1993: First announcement of R to the public.
- 1995: Martin Machler convinces Ross and Robert to use the GNU General Public License to make R free software.
- 1996: A public mailing list is created (R-help and R-devel)
- 1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.
- 2000: R version 1.0.0 is released.

Features of R

- Syntax is very similar to S, making it easy for S-PLUS users to switch over.
- Semantics are superficially similar to S, but in reality are quite different (more on that later).
- Runs on almost any standard computing platform/OS (even on the PlayStation 3)
- Frequent releases (annual + bugfix releases); active development.
- Quite lean, as far as software goes; functionality is divided into modular packages
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user - > programmer)
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow

Design of the R System

The R system is divided into 2 conceptual parts:

1. The "base" R system that you download from CRAN
2. Everything else.

R functionality is divided into a number of packages.

- The "base" R system contains, among other things, the base package which is required to run R and contains the most fundamental functions.
- The other packages contained in the "base" system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4.
- There are also "Recommend" packages: boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix.

Design of the R System

And there are many other packages available:

- There are about 4000 packages on CRAN that have been developed by users and programmers around the world.
- There are also many packages associated with the Bioconductor project (<http://bioconductor.org>).
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

Objects

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic object is a vector

- A vector can only contain objects of the same class
- BUT: The one exception is a list, which is represented as a vector but can contain objects of different classes (indeed, that's usually why we use them)

Numbers

- Numbers in R are generally treated as numeric objects (i.e. double precision real numbers)
- If you explicitly want an integer, you need to specify the `L` suffix
- Ex: Entering `1` gives you a numeric object; entering `1L` explicitly gives you an integer.
- There is also a special number `Inf` which represents infinity; e.g. `1 / 0`; `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is `0`
- The value `NaN` represents an undefined value ("not a number"); e.g. `0 / 0`; `NaN` can also be thought of as a missing value (more on that later)

Attributes

R objects can have attributes

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes/metadata

Attributes of an object can be accessed using the `attributes()` function.

Entering Input

At the R prompt we type expressions. The `<-` symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
> x <- ## Incomplete expression
```

The `#` character indicates a comment. Anything to the right of the `#` (including the `#` itself) is ignored.

Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
> x <- 5 ## nothing printed
> x      ## auto-printing occurs
[1] 5
> print(x) ## explicit printing
[1] 5
```

The `[1]` indicates that `x` is a vector and `5` is the first element.

Printing

```
> x <- 1:20
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20
```

The `:` operator is used to create integer sequences.

Creating Vectors

The `c()` function can be used to create vectors of objects.

```
> x <- c(0.5, 0.6)      ## numeric
> x <- c(TRUE, FALSE)  ## logical
> x <- c(T, F)         ## logical
> x <- c("a", "b", "c") ## character
> x <- 9:29            ## integer
> x <- c(1+0i, 2+4i)   ## complex
```

Using the `vector()` function

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects

What about the following?

```
> y <- c(1.7, "a")    ## character
> y <- c(TRUE, 2)    ## numeric
> y <- c("a", TRUE)  ## character
```

When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Explicit Coercion

Nonsensical coercion results in **NA**s.

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

Matrices

Matrices are vectors with a dimension attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
      [,1] [,2] [,3]
[1,]  NA  NA  NA
[2,]  NA  NA  NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```


Matrices (cont'd)

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices (cont'd)

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrices

```
> A <- matrix(1:6, nrow = 2, ncol = 3)
> B <- matrix(c(5,-1,4,2), nrow = 2, ncol = 2)
> B
      [,1] [,2]
[1,]    5    4
[2,]   -1    2
> C <- B %*% A
> C
      [,1] [,2] [,3]
[1,]   13   31   49
[2,]    3    5    7

> D = A * C
> D
      [,1] [,2] [,3]
[1,]   13   93  245
[2,]    6   20   42
```

Matrices

```
> B <- matrix(c(5,4,4,2), nrow = 2, ncol = 2)
> B
      [,1] [,2]
[1,]    5    4
[2,]    4    2
> lambda = eigen(B) #eigenvalues
> lambda
eigen() decomposition
$values
[1]  7.7720019 -0.7720019

$vectors
      [,1]      [,2]
[1,] -0.8219256  0.5695948
[2,] -0.5695948 -0.8219256
```

Matrices

```
> B <- matrix(c(5,4,4,2), nrow = 2, ncol = 2)
> B
      [,1] [,2]
[1,]    5    4
[2,]    4    2

> invB = solve(B)
> invB
      [,1]      [,2]
[1,] -0.3333333  0.6666667
[2,]  0.6666667 -0.8333333

> tB = t(B+matrix(c(0,1,0,0),2,2)) # transpose
> tB
      [,1] [,2]
[1,]    5    5
[2,]    4    2
```

cbind-ing and rbind-ing

Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a label.

- Factors are treated specially by modelling functions like `lm()` and `glm()`
- Using factors with labels is better than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Factors

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes no
Levels: no yes
> table(x)
x
no yes
  2  3
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no" "yes"
```


Factors

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
              levels = c("yes", "no"))
> x
[1] yes yes no yes no
Levels: yes no
```

Missing Values

Missing values are denoted by `NA` or `NaN` for undefined mathematical operations.

- `is.na()` is used to test objects if they are `NA`
- `is.nan()` is used to test for `NaN`
- `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
- A `NaN` value is also `NA` but the converse is not true

Missing Values

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

Data frames are used to store tabular data

- They are represented as a special type of list where every element of the list has to have the same length
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
- Data frames also have a special attribute called `row.names`
- Data frames are usually created by calling `read.table()` or `read.csv()`
- Can be converted to a matrix by calling `data.matrix()`

Data Frames

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo  bar
1   1 TRUE
2   2 TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("foo", "bar", "norf")
> x
foo bar norf
  1  2  3
> names(x)
[1] "foo" "bar" "norf"
```

Names

Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
```

```
> x
```

```
$a
```

```
[1] 1
```

```
$b
```

```
[1] 2
```

```
$c
```

```
[1] 3
```

Names

And matrices.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```


Summary

Data Types

- atomic classes: numeric, logical, character, integer, complex \
- vectors, lists
- factors
- missing values
- data frames
- names

Subsetting

There are a number of operators that can be used to extract subsets of R objects.

- `[]` always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
- `$` is used to extract elements of a list or data frame by name; semantics are similar to that of `[[`.

Subsetting

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x[x > "a"]
[1] "b" "c" "c" "d"
> u <- x > "a"
> u
[1] FALSE TRUE TRUE TRUE TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Subsetting a Matrix

Matrices can be subsetted in the usual way with (i,j) type indices.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing.

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

Subsetting a Matrix

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1 x 1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,] 3
```

Subsetting a Matrix

Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
      [,1] [,2] [,3]
[1,]    1    3    5
```

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6)
> x[1]
$foo
[1] 1 2 3 4

> x[[1]]
[1] 1 2 3 4

> x$bar
[1] 0.6
> x[["bar"]]
[1] 0.6
> x["bar"]
$bar
[1] 0.6
```

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```


Subsetting Lists

The `[[` operator can be used with computed indices; `$` can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> x[[name]] ## computed index for 'foo'
[1] 1 2 3 4
> x$name     ## element 'name' doesn't exist!
NULL
> x$foo
[1] 1 2 3 4 ## element 'foo' does exist
```

Subsetting Nested Elements of a List

The `[[` can take an integer sequence.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
> x[[c(1, 3)]]
[1] 14
> x[[1]][[3]]
[1] 14

> x[[c(2, 1)]]
[1] 3.14
```

Partial Matching

Partial matching of names is allowed with `[[` and `$`.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Removing NA Values

A common task is to remove missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

Removing NA Values

What if there are multiple things and you want to take the subset with no missing values?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1] TRUE TRUE FALSE TRUE FALSE TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
```

Removing NA Values

```
> airquality[1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41    190  7.4   67     5    1
2    36    118  8.0   72     5    2
3    12    149 12.6   74     5    3
4    18    313 11.5   62     5    4
5    NA     NA  14.3   56     5    5
6    28     NA  14.9   66     5    6

> good <- complete.cases(airquality)
> airquality[good, ][1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41    190  7.4   67     5    1
2    36    118  8.0   72     5    2
3    12    149 12.6   74     5    3
4    18    313 11.5   62     5    4
7    23    299  8.6   65     5    7
```

Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if, else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop while a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an iteration of a loop
- `return`: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

Control Structures: if

```
if(<condition>) {  
    ## do something  
} else {  
    ## do something else  
}  
  
if(<condition1>) {  
    ## do something  
} else if(<condition2>) {  
    ## do something different  
} else {  
    ## do something different  
}
```


if

This is a valid if/else structure.

```
if(x > 3) {  
    y <- 10  
} else {  
    y <- 0  
}
```

So is this one.

```
y <- if(x > 3) {  
    10  
} else {  
    0  
}
```

if

Of course, the else clause is not necessary.

```
if(<condition1>) {  
  
}  
  
if(<condition2>) {  
  
}
```

for

`for` loops take an iterator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
Un = 5
for(i in 1:20) {
  oldUn = Un
  Un = sqrt(1+Un)
  print(cbind(i,Un))
  if(abs(Un - oldUn) < 1e-6) {
    break;
  }
}
```

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

for

These three loops have the same behavior.

```
x <- c("a", "b", "c", "d")

for(i in 1:4) {
  print(x[i])
}

for(i in seq_along(x)) {
  print(x[i])
}

for(letter in x) {
  print(letter)
}

for(i in 1:4) print(x[i])
```

while

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0
while(count < 10) {
  print(count)
  count <- count + 1
}
```

While loops can potentially result in infinite loops if not written properly. Use with care!

while

Sometimes there will be more than one condition in the test.

```
z <- 5

while(z >= 3 && z <= 10) {
  print(z)
  coin <- rbinom(1, 1, 0.5)

  if(coin == 1) { ## random walk
    z <- z + 1
  } else {
    z <- z - 1
  }
}
```

Conditions are always evaluated from left to right.

repeat

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a `repeat` loop is to call `break`.

```
x0 <- 1
tol <- 1e-8

repeat {
  x1 <- computeEstimate()

  if(abs(x1 - x0) < tol) {
    break
  } else {
    x0 <- x1
  }
}
```

repeat

The loop in the previous slide is a bit dangerous because there's no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

next, return

`next` is used to skip an iteration of a loop

```
for(i in 1:100) {  
  if(i <= 20) {  
    ## Skip the first 20 iterations  
    next  
  }  
  ## Do something here  
}
```

`return` signals that a function should exit and return a given value

Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {  
  ## Do something interesting  
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have named arguments which potentially have default values.

- The formal arguments are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be missing or might have default values

Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

Argument Matching

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
         method = "qr", model = TRUE, x = FALSE,
         y = FALSE, qr = TRUE, singular.ok = TRUE,
         contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Argument Matching

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Argument Matching

Function arguments can also be partially matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Defining a Function

```
f <- function(a, b = 1, c = 2, d = NULL) {  
  
}
```

In addition to not specifying a default value, you can also set an argument value to `NULL`.

Lazy Evaluation

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

```
f <- function(a, b) {  
  a^2  
}  
f(2)
```

```
## [1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the `2` gets positionally matched to `a`.

Lazy Evaluation

```
f <- function(a, b) {  
  print(a)  
  print(b)  
}  
f(45)
```

```
## [1] 45
```

```
## Error in print(b): l'argument "b" est manquant, avec aucune valeur par défaut
```

Notice that "45" got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

Looping on the Command Line

Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of `lapply`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

lapply

`lapply` takes three arguments: (1) a list `X`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list`.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x000000000725dfd8>
## <environment: namespace:base>
```

The actual looping is done internally in C code.

lapply

`lapply` always returns a list, regardless of the class of the input.

```
x <- list(a = 1:5, b = rnorm(10))  
lapply(x, mean)
```

```
## $a  
## [1] 3  
##  
## $b  
## [1] 0.08014972
```

lapply

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))  
lapply(x, mean)
```

```
## $a  
## [1] 2.5  
##  
## $b  
## [1] -0.2304558  
##  
## $c  
## [1] 1.263421  
##  
## $d  
## [1] 4.890913
```

lapply

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

lapply

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 3.302142

[[2]]
[1] 6.848960 7.195282

[[3]]
[1] 3.5031416 0.8465707 9.7421014

[[4]]
[1] 1.195114 3.594027 2.930794 2.766946
```


lapply

`lapply` and friends make heavy use of anonymous functions.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
      [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
      [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

lapply

An anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

sapply

`sapply` will try to simplify the result of `lapply` if possible.

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

sapply

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

sapply

```
> sapply(x, mean)
```

```
      a      b      c      d  
2.5000000 0.06082667 1.46708277 5.07474950
```

```
> mean(x)
```

```
[1] NA
```

Warning message:

```
In mean.default(x) : argument is not numeric or logical: returning NA
```

Read and write data

```
> thedata <- read.csv("data.csv", sep = ";", header = T)
> head(thedata)

> library(dplyr)
> newData <- mutate(thedata, portfolio = apple + google+sp500)
> write.csv(newData, "newdata.csv", sep=";")
```

Generating Random Numbers

Functions for probability distributions in R

- `rnorm`: generate random Normal variates with a given mean and standard deviation
- `dnorm`: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- `pnorm`: evaluate the cumulative distribution function for a Normal distribution
- `rpois`: generate random Poisson variates with a given rate

Generating Random Numbers

Probability distribution functions usually have four functions associated with them. The functions are prefixed with a

- d for density
- r for random number generation
- p for cumulative distribution
- q for quantile function

Generating Random Numbers

Working with the Normal distributions requires using these four functions

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

If Φ is the cumulative distribution function for a standard Normal distribution, then $\text{pnorm}(q) = \Phi(q)$ and $\text{qnorm}(p) = \Phi^{-1}(p)$.

Generating Random Numbers

```
> x <- rnorm(10)
> x
[1] 1.38380206 0.48772671 0.53403109 0.66721944
[5] 0.01585029 0.37945986 1.31096736 0.55330472
[9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
[1] 23.38812 20.16846 21.87999 20.73813 19.59020
[6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 18.32   19.73   20.55   20.67   21.67   23.39
```

Generating Random Numbers

Setting the random number seed with `set.seed` ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
```

Always set the random number seed when conducting a simulation!

Generating Random Numbers

Generating Poisson data

```
> rpois(10, 1)
[1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
[1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
[1] 20 11 21 20 20 21 17 15 24 20

> ppois(2, 2) ## Cumulative distribution
[1] 0.6766764 ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347 ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662 ## Pr(x <= 6)
```

Generating Random Numbers From a Linear Model

Suppose we want to simulate from the following linear model $y = \beta_0 + \beta_1 x + \varepsilon$ where $\varepsilon \sim \mathcal{N}(0, 2^2)$. Assume $x \sim \mathcal{N}(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median 
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
> plot(x, y)
```

Generating Random Numbers From a Linear Model

What if x is binary?

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median 
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
> plot(x, y)
```

Generating Random Numbers From a Generalized Linear Model

Suppose we want to simulate from a Poisson model where

$Y \sim \text{Poisson}(\lambda)$; $\log \lambda = \beta_0 + \beta_1 x$

and $\beta_0 = 0.5$ and $\beta_1 = 0.3$. We need to use the `rpois` function for this

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.00   1.00   1.00   1.55   2.00   6.00
> plot(x, y)
```

Random Sampling

The `sample` function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
> sample(1:10) ## permutation
[1] 4 7 10 6 9 2 8 3 1 5
> sample(1:10)
[1] 2 3 4 1 9 5 10 8 6 7
> sample(1:10, replace = TRUE) ## Sample w/replacement
[1] 2 9 7 8 2 8 5 9 7 8
```


Useful packages :

rootSolve : Nonlinear Root Finding, Equilibrium and Steady-State Analysis of Ordinary Differential Equations.

```
> library(rootSolve)
> fun <- function (x) cos(1+x^2)
> curve(fun(x), -5, 5, main = "f(x)")
> All <- uniroot.all(fun, c(-5, 5))
> points(All, rep(0, length(All)), col = "red", pch = 16)
> fun(All)
> grad
```

Image Converting

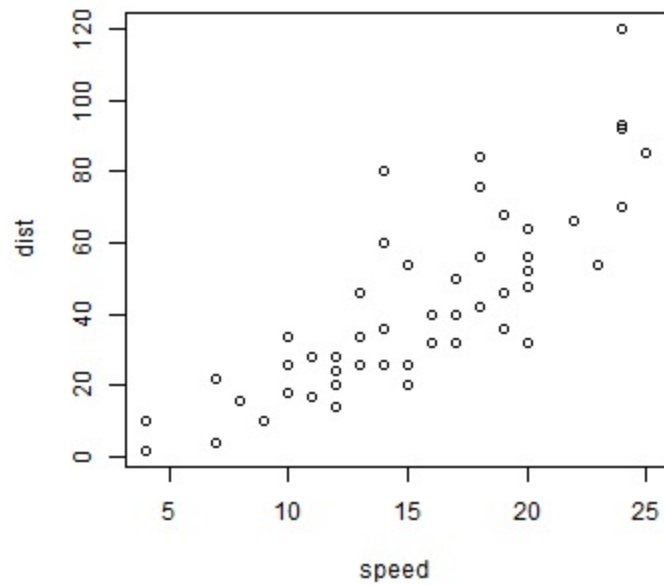
```
> library("png")  
> library("colorspace")  
> x <- readPNG("twitter.png")  
> library(jpeg)  
> img <- readJPEG(system.file("img", "Rlogo.jpg", package="jpeg"))
```

The Base Plotting System

- Convenient, mirrors how we think of building plots and analyzing data
- Can't go back once plot has started (i.e. to adjust margins); need to plan in advance
- Difficult to "translate" to others once a new plot has been created (no graphical "language")
- Plot is just a series of R commands

Base Plot

```
library(datasets)
data(cars)
with(cars, plot(speed, dist))
```



The Lattice System

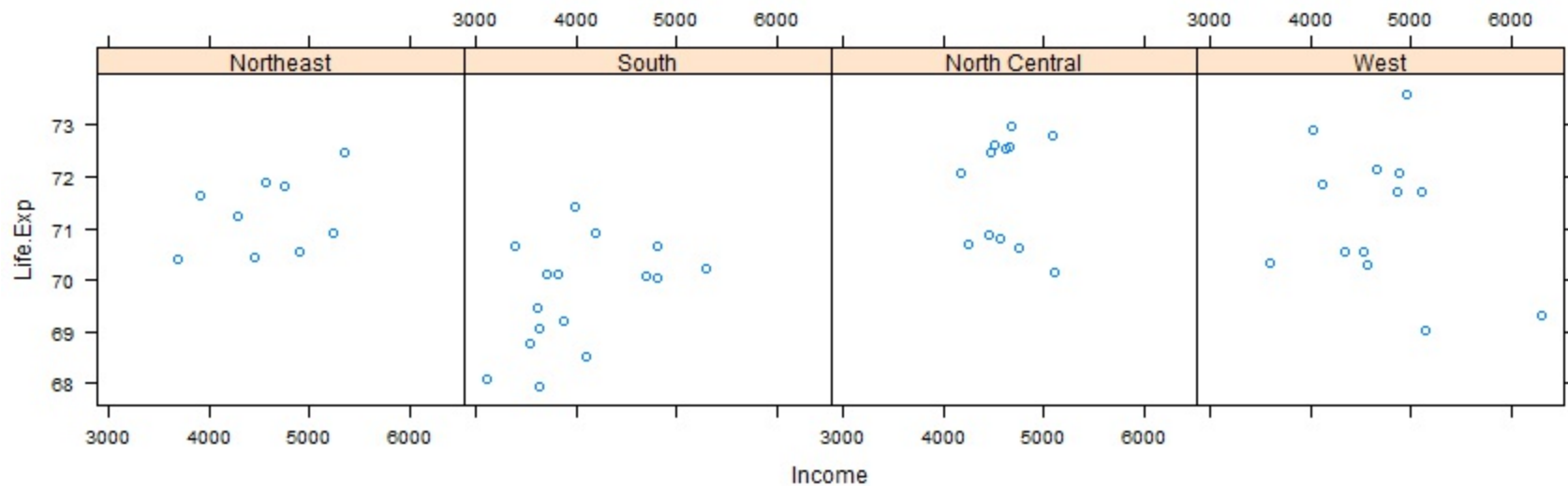
- Plots are created with a single function call (`xypplot`, `bwplot`, etc.)
- Most useful for conditioning types of plots: Looking at how y changes with x across levels of z
- Things like margins/spacing set automatically because entire plot is specified at once
- Good for putting many many plots on a screen

The Lattice System

- Sometimes awkward to specify an entire plot in a single function call
- Annotation in plot is not especially intuitive
- Use of panel functions and subscripts difficult to wield and requires intense preparation
- Cannot "add" to the plot once it is created

Lattice Plot

```
library(lattice)
state <- data.frame(state.x77, region = state.region)
xyplot(Life.Exp ~ Income | region, data = state, layout = c(4, 1))
```

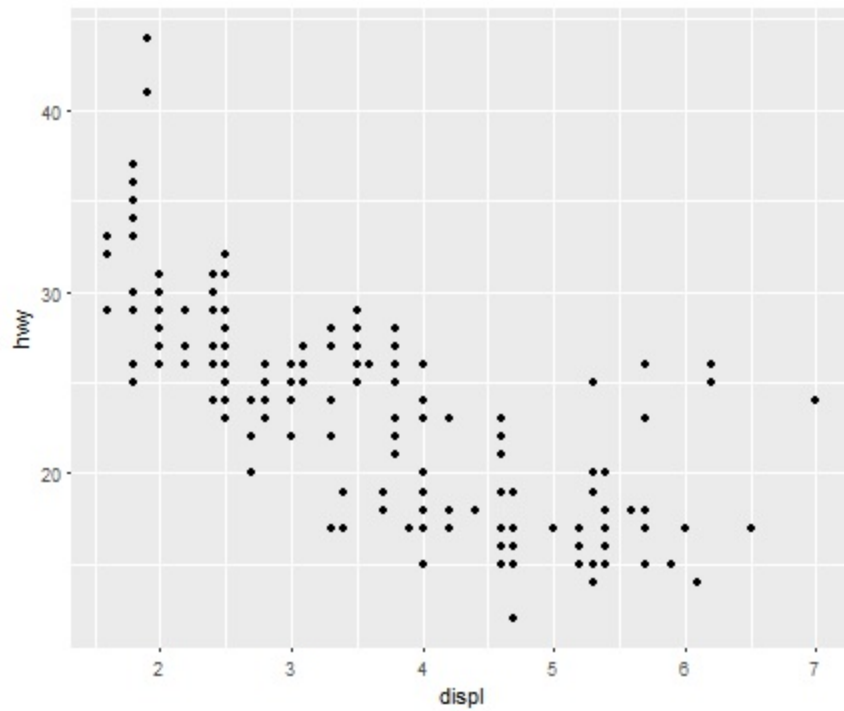


The ggplot2 System

- Splits the difference between base and lattice in a number of ways
- Automatically deals with spacings, text, titles but also allows you to annotate by "adding" to a plot
- Superficial similarity to lattice but generally easier/more intuitive to use
- Default mode makes many choices for you (but you can still customize to your heart's desire)

ggplot2 Plot

```
library(ggplot2)  
data(mpg)  
qplot(displ, hwy, data = mpg)
```



Summary

- Base: "artist's palette" model
- Lattice: Entire plot specified by one function; conditioning
- ggplot2: Mixes elements of Base and Lattice