

R Programming Course Notes

Xing Su

Contents

Overview and History of R	3
Coding Standards	4
Workspace and Files	4
R Console and Evaluation	4
R Objects and Data Structures	5
Vectors and Lists	5
Matrices and Data Frames	6
Arrays	7
Factors	8
Missing Values	9
Sequence of Numbers	10
Subsetting	10
Vectors	10
Lists	10
Matrices	11
Partial Matching	11
Logic	12
Understanding Data	12
Split-Apply-Combine Functions	13
split()	13
apply()	13
lapply()	13
sapply()	14
vapply()	14
tapply()	14
mapply()	14
aggregate()	15
Simulation	16
Simulation Examples	16
Generate Numbers for a Linear Model	17
Dates and Times	18

Base Graphics	18
Reading Tabular Data	19
Larger Tables	19
Textual Data Formats	19
Interfaces to the Outside World	20
Control Structures	21
if - else	21
for	21
while	22
repeat and break	22
next and return	22
Functions	23
Scoping	24
Scoping Example	24
Lexical vs Dynamic Scoping Example	25
Optimization	26
Debugging	27
R Profiler	27

Overview and History of R

- **R** = dialect of the **S** language
 - S was developed by John Chambers @ Bell Labs
 - initiated in 1976 as internal tool, originally FORTRAN libraries
 - 1988 rewritten in C (version 3 of language)
 - 1998 version 4 (what we use today)
- **History of S**
 - Bell labs -> insightful -> Lucent -> Alcatel-Lucent
 - in 1998, S won the Association for computing machinery's software system award
- **History of R**
 - 1991 created in New Zealand by Ross Ihaka & Robert Gentleman
 - 1993 first announcement of R to public
 - 1995 Martin Machler convinces founders to use GNU General Public license to make R free
 - 1996 public mailing list created R-help and R-devel
 - 1997 R Core Group formed
 - 2000 R v1.0.0 released
- **R Features**
 - Syntax similar to S, semantics similar to S, runs on any platforms, frequent releases
 - lean software, functionalities in modular packages, sophisticated graphics capabilities
 - useful for interactive work, powerful programming language
 - active user community and **FREE** (4 freedoms)
 - * freedom to run the program
 - * freedom to study how the program works and adapt it
 - * freedom to redistribute copies
 - * freedom to improve the program
- **R Drawbacks**
 - 40 year-old technology
 - little built-in support for dynamic/3D graphics
 - functionality based on consumer demand
 - objects generally stored in physical memory (limited by hardware)
- **Design of the R system**
 - 2 conceptual parts: base R from CRAN vs. everything else
 - functionality divided into different packages
 - * **base R** contains core functionality and fundamental functions
 - * other utility packages included in the base install: `util`, `stats`, `datasets`, ...
 - * Recommended packages: `boot` class, `KernSmooth`, etc
 - 5000+ packages available

Coding Standards

- Always use text files/editor
- Indent code (4 space minimum)
- limit the width of code (80 columns)
- limit the length of individual functions

Workspace and Files

- `getwd()` = return current working directory
- `setwd()` = set current working directory
- `?function` = brings up help for that function
- `dir.create("path/foldername", recursive = TRUE)` = create directories/subdirectories
- `unlink(directory, recursive = TRUE)` = delete directory and subdirectories
- `ls()` = list all objects in the local workspace
- `list.files(recursive = TRUE)` = list all, including subdirectories
- `args(function)` = returns arguments for the function
- `file.create("name")` = create file
 - `.exists("name")` = return true/false exists in working directory
 - `.info("name")` = return file info
 - `.info("name")$property` = returns value for the specific attribute
 - `.rename("name1", "name2")` = rename file
 - `.copy("name1", "name2")` = copy file
 - `.path("name1")` = return path of file

R Console and Evaluation

- `<-` = assignment operator
- `#` = comment
- expression is evaluated after hitting **enter** and result is returned
- autoprining occurs when you call a variable
 - `print(x)` = explicitly printing
- `[1]` at the beginning of the output = which element of the vector is being shown

R Objects and Data Structures

- 5 basic/**atomic** classes of objects:
 1. character
 2. numeric
 3. integer
 4. complex
 5. logical
- **Numbers**
 - numbers generally treated as **numeric** objects (double precision real numbers - decimals)
 - Integer objects can be created by adding **L** to the end of a number(ex. 1L)
 - **Inf** = infinity, can be used in calculations
 - **NaN** = not a number/undefined
 - **sqrt(value)** = square root of value
- **Variables**
 - **variable <- value** = assignment of a value to a variable name

Vectors and Lists

- **atomic vector** = contains one data type, most basic object
 - **vector <- c(value1, value2, ...)** = creates a vector with specified values
 - **vector1*vector2** = element by element multiplication (rather than matrix multiplication)
 - * if the vectors are of different lengths, shorter vector will be recycled until the longer runs out
 - * computation on vectors/between vectors (+, -, ==, /, etc.) are done element by element by default
 - * **%*%** = force matrix multiplication between vectors/matrices
 - **vector("class", n)** = creates empty vector of length n and specified class
 - * **vector("numeric", 3)** = creates 0 0 0
- **c()** = concatenate
 - **T, F** = shorthand for TRUE and FALSE
 - **1+0i** = complex numbers
- **explicit coercion**
 - **as.numeric(x), as.logical(x), as.character(x), as.complex(x)** = convert object from one class to another
 - nonsensible coercion will result in NA (ex. **as.numeric(c("a", "b"))**)
 - **as.list(data.frame)** = converts a **data.frame** object into a **list** object
 - **as.characters(list)** = converts list into a character vector
- **implicit coercion**
 - matrix/vector can only contain one data type, so when attempting to create matrix/vector with different classes, forced coercion occurs to make every element to same class
 - * *least common denominator* is the approach used (basically everything is converted to a class that all values can take, numbers -> characters) and *no errors generated*
 - * coercion occurs to make every element to same class (implicit)
 - * **x <- c(NA, 2, "D")** will create a vector of character class
- **list()** = special vector with different classes of elements

- `list` = vector of objects of different classes
- elements of list use `[[]]`, elements of other vectors use `[]`
- **logical vectors** = contain values `TRUE`, `FALSE`, and `NA`, values are generated as result of logical conditions comparing two objects/values
- `paste(characterVector, collapse = " ")` = join together elements of the vector and separating with the `collapse` parameter
- `paste(vec1, vec2, sep = " ")` = join together different vectors and separating with the `sep` parameter
 - ***Note:** vector recycling applies here too*
 - `LETTERS`, `letters` = predefined vectors for all 26 upper and lower letters
- `unique(values)` = returns vector with all duplicates removed

Matrices and Data Frames

- `matrix` can contain **only 1** type of data
- `data.frame` can contain **multiple**
- `matrix(values, nrow = n, ncol = m)` = creates a `n` by `m` matrix
 - constructed **COLUMN WISE** → the elements are placed into the matrix from top to bottom for each column, and by column from left to right
 - matrices can also be created by adding the dimension attribute to vector
 - * `dim(m) <- c(2, 5)`
 - matrices can also be created by binding columns and rows
 - * `rbind(x, y)`, `cbind(x, y)` = combine rows/columns; can be used on vectors or matrices
 - * and / = element by element computation between two matrices
 - * `%*%` = matrix multiplication
- `dim(obj)` = dimensions of an object (returns `NULL` if a vector)
 - `dim(obj) <- c(4, 5)` = assign `dim` attribute to an object
 - * if object is a vector, R converts the vector to a `n` by `m` matrix (i.e. 4 rows by 5 column from the example command)
 - ***Note:** if `n` by `m` is larger than length of vector, then an error is returned*
 - * ***example***

```
# initiate a vector
x <-c(NA, 1, "cx", NA, 2, "dsa")
class(x)
```

```
## [1] "character"
```

```
x
```

```
## [1] NA      "1"      "cx"     NA       "2"      "dsa"
```

```
# convert to matrix
dim(x) <- c(3, 2)
class(x)
```

```
## [1] "matrix"
```

```
##      [,1] [,2]
## [1,] NA   NA
## [2,] "1"  "2"
## [3,] "cx" "dsa"
```

- `data.frame(var = 1:4, var2 = c(...))` = creates a data frame
 - `nrow()`, `ncol()` = returns row and column numbers
 - `data.frame(vector, matrix)` = takes any number of arguments and returns a single object of class “data.frame” composed of original objects
 - `as.data.frame(obj)` = converts object to data frame
 - data frames store tabular data
 - special type of list where every list has the same length (can be of different type)
 - data frames are usually created through `read.table()` and `read.csv()`
 - `data.matrix()` = converts a matrix to data frame
- `colMeans(matrix)` or `rowMeans(matrix)` = returns means of the columns/rows of a matrix/dataframe in a vector
- `as.numeric(rownames(df))` = returns row indices for rows of a data frame with unnamed rows
- **attributes**
 - objects can have attributes: `names`, `dimnames`, `row.names`, `dim` (matrices, arrays), `class`, `length`, or any user-defined ones
 - `attributes(obj)`, `class(obj)` = return attributes/class for an R object
 - `attr(object, "attribute") <- "value"` = creates/assigns a value to a new/existing attribute for the object
 - **names attribute**
 - * all objects can have names
 - * `names(x)` = returns names (NULL if no name exists)
 - `names(x) <- c("a", ...)` = can be used to assign names to vectors
 - * `lists(a = 1, b = 2, ...)` = a, b are names
 - * `dimnames(matrix) <- list(c("a", "b"), c("c", "d"))` = assign names to matrices
 - use list of two vectors: row, column in that order
 - `colnames(data.frame)` = return column names (can be used to set column names as well, similar to `dim()`)
 - `row.names` = names of rows in the data frame (attribute)

Arrays

- multi-dimensional collection of data with k dimensions
 - matrix = 2 dimensional array
- `array(data, dim, dimnames)`
 - `data` = data to be stored in array
 - `dim` = dimensions of the array
 - * `dim = c(2, 2, 5)` = 3 dimensional array → creates 5 2x2 array
 - `dimnames` = add names to the dimensions
 - * input must be a list

- * every element of the `list` must correspond in length to the dimensions of the array
- * `dimnames(x) <- list(c("a", "b"), c("c", "d"), c("e", "f", "g", "h", "i"))` =
 set the names for row, column, and third dimension respectively (2 x 2 x 5 in this case)
- `dim()` function can be used to create arrays from vectors or matrices
 - `x <- rnorm(20); dim(x) <- c(2, 2, 5)` = converts a 20 element vector to a 2x2x5 array

Factors

- Factors are used to represent categorical data (integer vector where each value has a label)
- 2 types: **unordered** vs **ordered**
- treated specially by `lm()`, `glm()`
- Factors easier to understand because they self describe (vs. 1 and 2)
- `factor(c("a", "b"), levels = c("1", "2"))` = creates factor
 - `levels()` argument can be used to specify baseline levels vs other levels
 - * ***Note:**without explicit specification, R uses alphabetical order*
 - `table(factorVar)` = how many of each are in the factor

Missing Values

- NaN or NA = missing values
 - NaN = undefined mathematical operations
 - NA = any value not available or missing in the statistical sense
 - * any operations with NA results in NA
 - * NA can have different classes potentially (integer, character, etc)
 - **Note:** *NaN is an NA value, but NA is not NaN*
- `is.na()`, `is.nan()` = use to test if each element of the vector is NA and NaN
 - **Note:** *cannot compare NA (with `==`) as it is not a value but a **placeholder** for a quantity that is not available*
- `sum(my_na)` = sum of a logical vector (`TRUE` = 1 and `FALSE` = 0) is effectively the number of `TRUE`s
- **Removing NA Values**
 - `is.na()` = creates logical vector where T is where value exists, F is NA
 - * subsetting with the above result can return only the non NA elements
 - `complete.cases(obj1, obj2)` = creates logical vector where `TRUE` is where both values exist, and `FALSE` is where any is NA
 - * can be used on data frames as well
 - * `complete.cases(data.frame)` = creates logical vectors indicating which observation/row is good
 - * `data.frame[logicalVector,]` = returns all observations with complete data
- **Imputing Missing Values** = replacing missing values with estimates (can be averages from all other data with the similar conditions)

Sequence of Numbers

- `1:20` = creates a sequence of numbers from first number to second number
 - works in descending order as well
 - increment = 1
- `?'':'` = enclose help for operators
- `seq(1, 20, by=0.5)` = sequence 1 to 20 by increment of .5
 - `length=30` argument can be used to specify number of values generated
- `length(variable)` = length of vector/sequence
- `seq_along(vector)` or `seq(along.with = vector)` = create vector that is same length as another vector
- `rep(0, times = 40)` = creates a vector with 40 zeroes
 - `rep(c(1, 2), times = 10)` = repeats combination of numbers 10 times
 - `rep(c(1, 2), each = 10)` = repeats first value 10 times followed by second value 10 times

Subsetting

- R uses **one based index** -> starts counting at 1
 - `x[0]` returns `numeric(0)`, not error
 - `x[3000]` returns `NA` (not out of bounds/error)
- `[]` = always returns object of same class, can select more than one element of an object `[1;2]`
- `[[]]` = can extract one element from list or data frame, returned object not necessarily list/dataframe
- `$` = can extract elements from list/dataframe that have names associated with it, not necessarily same class

Vectors

- `x[1:10]` = first 10 elements of vector `x`
- `x[is.na(x)]` = returns all NA elements
- `x[!is.na(x)]` = returns all non NA elements
 - `x > 0` = would return logical vector comparing all elements to 0 (`TRUE/FALSE` for all values except for NA and NA for NA elements (NA a placeholder)
- `x[x>"a"]` = selects all elements bigger than a (lexicographical order in place)
- `x[logicalIndex]` = select all elements where logical index = `TRUE`
- `x[-c(2, 10)]` = returns everything **but** the second and tenth element
- `vect <- c(a = 1, b = 2, c = 3)` = names values of a vector with corresponding names
- `names(vect)` = returns element names for object
 - `names(vet) <- c("a", "b", "c")` = assign/change names of vector
- `identical(obj1, obj2)` = returns if the two objects are exactly equal

Lists

- `x <- list(foo = 1:4, bar = 0.6)`
- `x[1]` or `x["foo"]` = returns the list object `foo`

- `x[[2]]` or `x[["bar"]]` or `x$bar` = returns the content of the second element from the list (in this case vector without name attribute)
- `x[c(1, 3)]` = extract multiple elements of list (`[[]]`)
 - ***Note:** `$` can't extract multiple*
- `x[[name]]` = extract using variable, where as `$` must match name of element
- `x[[c(1, 3)]]` or `x[[1]][[3]]` = extracted nested elements of list third element of the first object extracted from the list

Matrices

- `x[1, 2]` = extract the (row, column) element
 - `x[,2]` or `x[1,]` = extract the entire column/row
- `x[, 11:17]` = subset the `x data.frame` with all rows, but only 11 to 17 columns
- when an element from the matrix is retrieved, a vector is returned
 - behavior can be turned off (force return a matrix) by adding `drop = FALSE`
 - * `x[1, 2, drop = F]`

Partial Matching

- works with `[[]]` and `$`
- `$` automatically partial matches the name (`x$a`)
- `[[]]` can partial match by adding `exact = FALSE`
 - `x[["a", exact = false]]`

Logic

- `<`, `>=` -> less than, greater or equal to
- `==` -> exact equality
- `!=` -> inequality
- `A | B` = union
- `A & B` = intersection
- `!` = negation
- `&` or `|` evaluates every instance/element in vector
- `&&` or `||` evaluate only first element
 - All AND operators are evaluated before PR operators
- `isTRUE(condition)` = returns `TRUE` or `FALSE` of the condition
- `xor(arg1, arg2)` = exclusive OR, one argument must equal `TRUE` one must equal `FALSE`
- `which(condition)` = find the indices of elements that satisfy the condition (`TRUE`)
- `any(condition)` = `TRUE` if one or more of the elements in logical vector is `TRUE`
- `all(condition)` = `TRUE` if all of the elements in logical vector is `TRUE`

Understanding Data

- use `class()`, `dim()`, `nrow()`, `ncol()`, `names()` to understand dataset
 - `object.size(data.frame)` = returns how much space the dataset is occupying in memory
- `head(data.frame, 10)`, `tail(data.frame, 10)` = returns first/last 10 rows of data; default = 6
- `summary()` = provides different output for each variable, depending on class,
 - for numerical variables, displays min max, mean median, etc.
 - for categorical (factor) variables, displays number of times each value occurs
- `table(data.frame$variable)` = table of all values of the variable, and how many observations there are for each
 - **Note:** mean for variables that only have values 1 and 0 = proportion of success `str(data.frame)` = structure of data, provides data class, num of observations vs variables, and name of class of each variable and preview of its contents
 - compactly display the internal structure of an R object
 - “What’s in this object”
 - well-suited to compactly display the contents of lists

Split-Apply-Combine Functions

- loop functions = convenient ways of implementing the Split-Apply-Combine strategy for data analysis

split()

- takes a vector/objects and splits it into groups by a factor or list of factors
- `split(x, f, drop = FALSE)`
 - `x` = vector/list/data frame
 - `f` = factor/list of factors
 - `drop` = whether empty factor levels should be dropped
- `interactions(gl(2, 5), gl(5, 2)) = 1.1, 1.2, ... 2.5`
 - `gl(n, m)` = group level function
 - * `n` = number of levels
 - * `m` = number of repetitions
 - `split` function can do this by passing in `list(f1, f2)` in argument
 - * `split(data, list(gl(2, 5), gl(5, 2)))` = splits the data into 1.1, 1.2, ... 2.5 levels

apply()

- evaluate a function (often anonymous) over the margins of an array
- often used to apply a function to the row/columns of a matrix
- can be used to average array of matrices (general arrays)
- `apply(x, margin = 2, FUN, ...)`
 - `x` = array
 - `MARGIN` = 2 (column), 1 (row)
 - `FUN` = function
 - ... = other arguments that need to be passed to other functions
- *examples*
 - `apply(x, 1, sum)` or `apply(x, 1, mean)` = find row sums/means
 - `apply(x, 2, sum)` or `apply(x, 2, mean)` = find column sums/means
 - `apply(x, 1, quantile, probs = c(0.25, 0.75))` = find 25% 75% percentile of each row
 - `a <- array(rnorm(2*2*10), c(2, 2, 10))` = create 10 2x2 matrix
 - `apply(a, c(1, 2), mean)` = returns the means of 10

lapply()

- loops over a **list** and evaluate a function on each element and always returns a **list**
 - ***Note:** since input must be a list, it is possible that conversion may be needed*
- `lapply(x, FUN, ...)` = takes list/vector as input, applies a function to each element of the list, returns a list of the same length
 - `x` = list (if not list, will be coerced into list through “as.list”, if not possible —> error)
 - * `data.frame` are treated as collections of lists and can be used here
 - `FUN` = function (without parentheses)

- * anonymous functions are acceptable here as well - (i.e `function(x) x[,1]`)
- ... = other/additional arguments to be passed for FUN (i.e. `min`, `max` for `runif()`)

- *example*

- `lapply(data.frame, class)` = the `data.frame` is a list of vectors, the `class` value for each vector is returned in a list (name of function, `class`, is without parentheses)
- `lapply(values, function(elem), elem[2])` = example of an anonymous function

`sapply()`

- performs same function as `lapply()` except it simplifies the result
 - if result is of length 1 in every element, `sapply` returns vector
 - if result is vectors of the same length (>1) for each element, `sapply` returns matrix
 - if not possible to simplify, `sapply` returns a `list` (same as `lapply()`)

`vapply()`

- safer version of `sapply` in that it allows to you specify the format for the result
 - `vapply(flags, class, character(1))` = returns the `class` of values in the `flags` variable in the form of character of length 1 (1 value)

`tapply()`

- split data into groups, and apply the function to data within each subgroup
- `tapply(data, INDEX, FUN, ..., simplify = FALSE)` = apply a function over subsets of a vector
 - `data` = vector
 - `INDEX` = factor/list of factors
 - `FUN` = function
 - ... = arguments to be passed to function
 - `simplify` = whether to simplify the result
- *example*
 - `x <- c(rnorm(10), runif(10), rnorm(10, 1))`
 - `f <- gl(3, 10); tapply(x, f, mean)` = returns the mean of each group (f level) of x data

`mapply()`

- multivariate apply, applies a function in parallel over a set of arguments
- `mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE)`
 - `FUN` = function
 - ... = arguments to apply over
 - `MoreArgs` = list of other arguments to FUN
 - `SIMPLIFY` = whether the result should be simplified
- *example*

```
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

`aggregate()`

- `aggregate` computes summary statistics of data subsets (similar to multiple `tapply` at the same time)
- `aggregate(list(name = dataToCompute), list(name = factorVar1, name = factorVar2), function, na.rm = TRUE)`
 - `dataToCompute` = this is what the function will be applied on
 - `factorVar1`, `factorVar2` = factor variables to split the data by
 - ***Note:* order matters here in terms of how to break down the data*
 - `function` = what is applied to the subsets of data, can be sum/mean/median/etc
 - `na.rm = TRUE` -> removes NA values

Simulation

- `sample(values, n, replace = FALSE)` = generate random samples
 - `values` = values to sample from
 - `n` = number of values generated
 - `replace` = with or without replacement
 - `sample(1:6, 4, replace = TRUE, prob=c(.2, .2...))` = choose four values from the range specified with replacing (same numbers can show up twice), with probabilities specified
 - `sample(vector)` = can be used to permute/rearrange elements of a vector
 - `sample(c(y, z), 100)` = select 100 random elements from combination of values y and z
 - `sample(10)` = select positive integer sample of size 10 without repeat
- Each probability distribution functions usually have 4 functions associated with them:
 - `r***` function (for “random”) -> random number generation
 - `d***` function (for “density”) -> calculate density
 - `p***` function (for “probability”) -> cumulative distribution
 - `q***` function (for “quantile”) -> quantile function
- If Φ is the cumulative distribution function for a standard Normal distribution, then `pnorm(q) = $\Phi(q)$` and `qnorm(p) = $\Phi^{-1}(p)$` .
- `set.seed()` = reproduce same data

Simulation Examples

- `rbinom(1, size = 100, prob = 0.7)` = returns a binomial random variable that represents the number of successes in a give number of independent trials
 - `1` = corresponds number of observations
 - `size = 100` = corresponds with the number of independent trials that culminate to each resultant observation
 - `prob = 0.7` = probability of success
- `rnorm(n, mean = m, sd = s)` = generate n random samples from the standard normal distribution (mean = 0, std deviation = 1 by default)
 - `rnorm(1000)` = 1000 draws from the standard normal distribution
 - `n` = number of observation generated
 - `mean = m` = specified mean of distribution
 - `sd = s` = specified standard deviation of distribution
- `dnorm(x, mean = 0, sd = 1, log = FALSE)`
 - `log` = evaluate on log scale
- `pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)`
 - `lower.tail` = left side, `FALSE` = right
- `qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)`
 - `lower.tail` = left side, `FALSE` = right
- `rpois(n, lambda)` = generate random samples from the poisson distribution
 - `n` = number of observations generated
 - `lambda` = λ parameter for the poisson distribution or rate
- `rpois(n, r)` = generating Poisson Data

- `n` = number of values
- `r` = rate
- `ppois(n, r)` = cumulative distribution
 - `ppois(2, 2) = Pr(x ≤ 2)`
- `replicate(n, rpois())` = repeat operation `n` times

Generate Numbers for a Linear Model

- Linear model

$$y = \beta_0 + \beta_1 x + \epsilon \text{ where } \epsilon \sim N(0, 2^2), x \sim N(0, 1^2), \beta_0 = 0.5, \beta_1 = 2$$

```
set.seed(20)
x <- rnorm(100)           # normal
x <- rbinom(100, 1, 0.5)  # binomial
e <- rnorm(100, 0, 2)
y <- 0.5 + 2 * x + e
```

- Poisson model

$$Y \sim \text{Poisson}(\mu) \log(\mu) = \beta_0 + \beta_1 x \text{ where } \beta_0 = 0.5, \beta_1 = 2$$

```
x <- rnorm(100)
log.mu <- 0.5 + 0.3 * x
y <- rpois(100, exp(log.mu))
```

Dates and Times

- `Date` = date class, stored as number of days since 1970-01-01
- `POSIXct` = time class, stored as number of seconds since 1970-01-01
- `POSIXlt` = time class, stored as list of sec min hours
- `Sys.Date()` = today's date
- `unclass(obj)` = returns what obj looks like internally
- `Sys.time()` = current time in `POSIXct` class
- `t2 <- as.POSIXlt(Sys.time())` = time in `POSIXlt` class
 - `t2$min` = return min of time (only works for `POSIXlt` class)
- `weekdays(date)`, `months(date)`, `quarters(date)` = returns weekdays, months, and quarters of time/date inputed
- `strptime(string, "%B %d, %Y %H:%M")` = convert string into time format using the format specified
- `difftime(time1, time2, units = 'days')` = difference in times by the specified unit

Base Graphics

- `data(set)` = load data
- `plot(data)` = R plots the data as best as it can
 - `x` = variable, x axis
 - `y` = variable
 - `xlab`, `ylab` = corresponding labels
 - `main`, `sub` = title, subtitle
 - `col` = 2 or `col` = "red" = color
 - `pch` = 2 = different symbols for points
 - `xlim,ylim(v1, v2)` = restrict range of plot
- `boxplot(x ~ y, data = d)` = creates boxplot for x vs y variables using the data.frame provided
- `hist(x, breaks)` = plots histogram of the data
 - `break` = 100 = split data into 100 bins

Reading Tabular Data

- `read.table()`, `read.csv()` = most common, read text files (rows, col) return data frame
- `readLines()` = read lines of text, returns character vector
- `source(file)` = read R code
- `dget()` = read R code files (R objects that have been reparsed)
- `load()`, `unserialize()` = read binary objects
- writing data
 - `write.table()`, `writeLines()`, `dump()`, `put()`, `save()`, `serialize()`
- `read.table()` arguments:
 - `file` = name of file/connection
 - `header` = indicator if file contains header
 - `sep` = string indicating how columns are separated
 - `colClasses` = character vector indicating what each column is in terms of class
 - `nrows` = number of rows in dataset
 - `comment.char` = char indicating beginning of comment
 - `skip` = number of lines to skip in the beginning
 - `stringsAsFactors` = defaults to TRUE, should characters be coded as Factor
- `read.table` can be used without any other argument to create data.frame
 - telling R what type of variables are in each column is helpful for larger datasets (efficiency)
 - `read.csv()` = `read.table` except default sep is comma (read.table is " ") and header = TRUE

Larger Tables

- **Note:** *help page for read.table important*
- need to know how much RAM is required → calculating memory requirements
 - $\text{numRow} \times \text{numCol} \times 8 \text{ bytes/numeric value} = \text{size required in bites}$
 - double the above results and convert into GB = amount of memory recommended
- `set comment.char = ""` to save time if there are no comments in the file
- specifying `colClasses` can make reading data much faster
- `nrow = n`, number of rows to read in (can help with memory usage)
 - `initial <- read.table("file", rows = 100)` = read first 100 lines
 - `classes <- sapply(initial, class)` = determine what classes the columns are
 - `tabAll <- read.table("file", colClasses = classes)` = load in the entire file with determined classes

Textual Data Formats

- `dump` and `dput` preserve metadata
- text formats are editable, not space efficient, and work better with version control system (they can only track changes in text files)
- `dput(obj, file = "file.R")` = creates R code to store all data and meta data in "file.R" (ex. data, class, names, row.names)
- `dget("file.R")` = loads the file/R code and reconstructs the R object
- `dput` can only be used on one object, where as `dump` can be used on multiple objects
- `dump(c("obj1", "obj2"), file= "file2.R")` = stores two objects
- `source("file2.R")` = loads the objects

Interfaces to the Outside World

- `url()` = function can read from webpages
- `file()` = read uncompressed files
- `gzfile()`, `bzfile()` = read compressed files (gzip, bzip2)
- `file(description = "", open = "")` = file syntax, creates connection
 - `description` = description of file
 - `open` = r - readonly, w - writing, a - appending, rb/wb/ab - reading/writing/appending binary
 - `close()` = closes connection
 - `readLines()` = can be used to read lines after connection has been established
- `download.file(fileURL, destfile = "fileName", method = "curl")`
 - `fileURL` = url of the file that needs to be downloaded
 - `destfile = "fileName"` = specifies where the file is to be saved
 - * “dir/fileName” = directories can be referenced here
 - `method = "curl"` = necessary for downloading files from “<https://>” links on Macs
 - * `method = "auto"` = should work on all other machines

Control Structures

- 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
- ***Note:** Control structures are primarily useful for writing programs; for command-line interactive work, the `apply` functions are more useful*

`if - else`

```
# basic structure
if(<condition>) {
  ## do something
} else {
  ## do something else
}

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

- `y <- if(x>3){10} else {0}` = slightly different implementation than normal, focus on assigning value

`for`

```
# basic structure
for(i in 1:10) {
  # print(i)
}

# nested for loops
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))) {
    # print(x[i, j])
  }
}
```

- `seq_along()` = to length of the vector

- `for(letter in x)` = loop through letter in character vector
- `seq_len()` = 1:length of vector

while

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

- conditions can be combined with logical operators

repeat and break

- Repeat initiates an infinite loop
- 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 # requires algorithm to converge
  }
}
```

- **Note:** The above loop 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 and return

- `next` = (no parentheses) skips an element, to continue to the next iteration
- `return` = signals that a function should exit and return a given value

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

Functions

- `name <- function(arg1, arg2, ...){ }`
 - inputs can be specified with default values by `arg1 = 10`
 - it is possible to define an argument to `NULL`
 - returns **last expression** of function
 - many functions have `na.rm`, can be set to `TRUE` to remove NA values from calculation
- structure

```
f <- function(<arguments>) {  
## Do something interesting  
}
```
- functions are first class objects and can be **treated like other objects** (pass into other functions)
 - functions can be nested, so that you can define a function inside of another function
- functions have named arguments (i.e. `x = mydata`) which can be used to specify **default values**
 - `sd(x = mydata)` (matching by name)
- formal arguments = arguments included in the functional definition
 - `formals()` = returns all formal arguments
 - not all functional call specifies all arguments, some can be missing and may have default values
 - `args()` = return all arguments you can specify
 - multiple arguments inputted in random orders (R performs positional matching) -> not recommended
 - argument matching order: exact -> partial -> positional
 - * *partial* = instead of typing `data = x`, use `d = x`
- Lazy Evaluation
 - R will evaluate as needed, so everything executes until an error occurs
 - * `f <- function (a, b) {a^2}`
 - * if `b` is not used in the function, calling `f(5)` will not produce an error
- “...” argument
 - used to extend other functions by representing the rest of the arguments
 - generic functions use ... to pass extra arguments (i.e. `mean = 1, sd = 2`)
 - necessary when the number of arguments passed can not be known in advance
 - * functions that use “...” = `paste()`, `cat()`
 - **Note:** *arguments coming after ... must be explicitly matched and cannot be partially matched*

Scoping

- scoping rules determine how a value is associated with a free variable in a function
- **free variables** = variables not explicitly defined in the function (not arguments, or local variables - variable defined in the function)
- R uses **lexical/static scoping**
 - common alternative = **dynamic scoping**
 - **lexical scoping** = values of free vars are searched in the environment in which the function is defined
 - * environment = collection of symbol/value pairs ($x = 3.14$)
 - each package has its own environment
 - only environment **without** parent environment is the *empty environment*
 - **closure/function closure** = function + associated environment
- search order for free variable
 1. environment where the function is defined
 2. parent environment
 3. ... (repeat if multiple parent environments)
 4. top level environment: global environment (workspace) or namespace package
 5. empty environment → produce error
- when a function/variable is called, R searches through the following list to match the first result
 1. `.GlobalEnv`
 2. `package:stats`
 3. `package:graphics`
 4. `package:grDevices`
 5. `package:utils`
 6. `package:datasets`
 7. `package:methods`
 8. `Autoloads`
 9. `package:base`
- **order matters**
 - `.GlobalEnv` = everything defined in the current workspace
 - any package that gets loaded with `library()` gets put in position 2 of the above search list
 - namespaces are separate for functions and non-functions
 - * possible for object `c` and function `c` to coexist

Scoping Example

```
make.power <- function(n) {  
  pow <- function(x) {  
    x^n  
  }  
  pow  
}  
cube <- make.power(3) # defines a function with only n defined (x^3)  
square <- make.power(2) # defines a function with only n defined (x^2)  
cube(3) # defines x = 3
```



```
## [1] 27
```

```
square(3)           # defines x = 3
```

```
## [1] 9
```

```
# returns the free variables in the function  
ls(environment(cube))
```

```
## [1] "n"    "pow"
```

```
# retrieves the value of n in the cube function  
get("n", environment(cube))
```

```
## [1] 3
```

Lexical vs Dynamic Scoping Example

```
y <- 10  
f <- function(x) {  
  y <- 2  
  y^2 + g(x)  
}  
g <- function(x) {  
  x*y  
}
```

- **Lexical Scoping**

1. $f(3) \rightarrow$ calls $g(x)$
2. y isn't defined locally in $g(x) \rightarrow$ searches in parent environment (working environment/global workspace)
3. finds $y \rightarrow y = 10$

- **Dynamic Scoping**

1. $f(3) \rightarrow$ calls $g(x)$
2. y isn't defined locally in $g(x) \rightarrow$ searches in calling environment (f function)
3. find $y \rightarrow 2$
 - **parent frame** = refers to calling environment in R, environment from which the function was called

- **Note:** when the defining environment and calling environment is the same, lexical and dynamic scoping produces the same result

- **Consequences of Lexical Scoping**

- all objects must be carried in memory
- all functions carry pointer to their defining environment (memory address)

Optimization

- Optimization routines in R (`optim`, `nlm`, `optimize`) require you to pass a function whose argument is a vector of parameters
 - *Note: these functions **minimize**, so use the negative constructs to maximize a normal likelihood*
- **Constructor functions** = functions to be fed into the optimization routines
- *example*

```
# write constructor function
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
  }
}
# initialize seed and print function
set.seed(1); normals <- rnorm(100, 1, 2)
nLL <- make.NegLogLik(normals); nLL
```

```
## function(p) {
##     params[!fixed] <- p
##     mu <- params[1]
##     sigma <- params[2]
##     a <- -0.5*length(data)*log(2*pi*sigma^2)
##     b <- -0.5*sum((data-mu)^2) / (sigma^2)
##     -(a + b)
## }
## <environment: 0x7f8c4295a990>
```

```
# Estimating Parameters
optim(c(mu = 0, sigma = 1), nLL)$par
```

```
##      mu      sigma
## 1.218239 1.787343
```

```
# Fixing sigma = 2
nLL <- make.NegLogLik(normals, c(FALSE, 2))
optimize(nLL, c(-1, 3))$minimum
```

```
## [1] 1.217775
```

```
# Fixing mu = 1
nLL <- make.NegLogLik(normals, c(1, FALSE))
optimize(nLL, c(1e-6, 10))$minimum
```

```
## [1] 1.800596
```

Debugging

- **message**: generic notification/diagnostic message, execution continues
 - `message()` = generate message
- **warning**: something's wrong but not fatal, execution continues
 - `warning()` = generate warning
- **error**: fatal problem occurred, execution stops
 - `stop()` = generate error
- **condition**: generic concept for indicating something unexpected can occur
- **invisible()** = suppresses auto printing
- **Note**: *random number generator must be controlled to reproduce problems (`set.seed` to pinpoint problem)*
- **traceback**: prints out function call stack after error occurs
 - must be called right after error
- **debug**: flags function for debug mode, allows to step through function one line at a time
 - `debug(function)` = enter debug mode
- **browser**: suspends the execution of function wherever its placed
 - embedded in code and when the code is run, the browser comes up
- **trace**: allows inserting debugging code into a function at specific places
- **recover**: error handler, freezes at point of error
 - `options(error = recover)` = instead of console, brings up menu (simi)

R Profiler

- optimizing code cannot be done without performance analysis and profiling

```
# system.time example
system.time({
  n <- 1000
  r <- numeric(n)
  for (i in 1:n) {
    x <- rnorm(n)
    r[i] <- mean(x)
  }
})
```

```
##      user  system elapsed
## 0.145   0.004   0.198
```

- `system.time(expression)`
 - takes R expression, returns amount of time needed to execute (assuming you know where)
 - computes time (in sec) - gives time until error if error occurs
 - can wrap multiple lines of code with `{}`
 - returns object of class `proc_time`
 - * **user time** = time computer experience

- * **elapsed time** = time user experience
- * usually close for standard computation
 - *elapse* > *user* = CPU wait around other processes in the background (read webpage)
 - *elapsed* < *user* = multiple processor/core (use multi-threaded libraries)
- **Note:** *R doesn't multi-thread (performing multiple calculations at the same time) with basic package*
 - * Basic Linear Algebra Standard [BLAS] libraries do, prediction, regression routines, matrix
 - * i.e. vecLib/Accelerate, ATLAS, ACML, MKL
- **Rprof()** - useful for complex code only
 - keeps track of functional call stack at regular intervals and tabulates how much time is spent in each function
 - default sampling interval = 0.02 second
 - calling **Rprof()** generates **Rprof.out** file by default
 - * **Rprof("output.out")** = specify the output file
- **summaryRprof()** = summarizes **Rprof()** output, 2 methods for normalizing data
 - loads the **Rprof.out** file by default, can specify output file **summaryRprof("output.out")**
 - **by.total** = divide time spent in each function by total run time
 - **by.self** = first subtracts out time spent in functions above in call stack, and calculates ratio to total
 - **\$sample.interval** = 0.02 - interval
 - **\$sampling.time** = 7.41 - seconds, elapsed time
 - * **Note:** *generally user spends all time at top level function (i.e. **lm()**), but the function simply calls helper functions to do work so it is not useful to know about the top level function times*
 - **Note:** *by.self* = more useful as it focuses on each individual call/function
- Good to break code into functions so profilers can give useful information about where time is spent
- C/FORTRAN code is not profiled
- **Note:** *R must be compiled with profiles support (generally the case)*
- **Note:** *should NOT be used with **system.time()***