R crash course - Introduction

Jongbin Jung

Dependencies

- Please download and install
  - Latest version ($\geq$ `3.1.2`) of `R` (*free* – GPL – from `https://www.r-project.org/`)
  - Latest version of `Rstudio` (also *free* – AGPL – from `https://www.rstudio.com/`)

A Question

You are given three sticks, each of a random length between 0 and 1.

What's the probability you can make a triangle?

- The answer is $1/2$
- By the end of this session, we'll confirm this with a simulation in `R`

Rstudio Basics

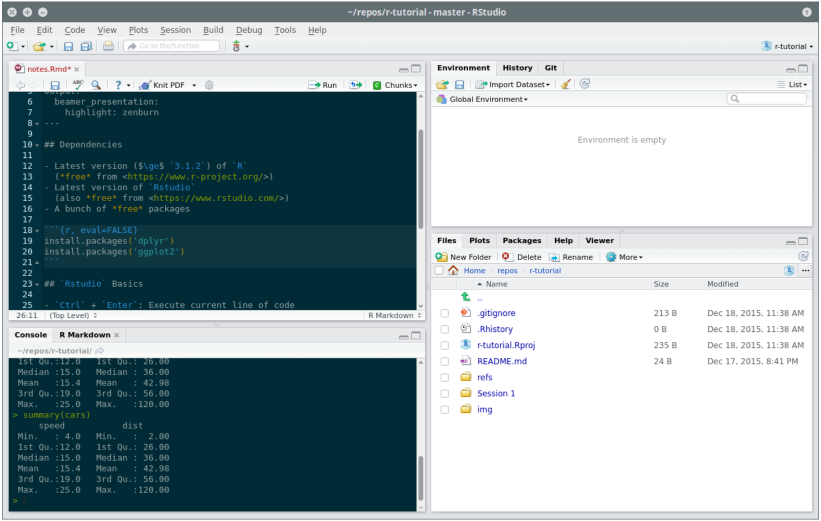

Figure 1: Screenshot of Rstudio

Rstudio Basics

- Save/load scripts as text files named `*.R`
- Save/load environment as `.Rdata` file
- `Ctrl` + `L`: Clear console
- `Ctrl` + `#`: Focus on panel #
- `Ctrl` + `Enter`: Execute selection
- `Ctrl` + `Shift` + `C`: Comment/Uncomment selection
- Many more (if you're willing to explore)

R Basics

R Basics: Working Directory

- Working directory (`wd`) is where your R session will load/save files
- To see where your current working directory is, run

```r
getwd()
```

- To set the working directory to desired `path`, run

```r
setwd("path")
```

- Note that ~ is replaced with your `HOME` directory, e.g. `C:\Users\Username\Documents` in windows
- Use forward slashes (/), even on Windows!

```r
setwd("~/R")
```

R Basics: Math Operations

- Simple math operations

```
3+11  # add stuff
3-11  # subtract stuff
3/11  # divide stuff
3*11  # multiply stuff
2^10  # raise to powers
```

R Basics: Assignments

- Convention for assigning values to variables is <-
- Direction of arrow indicates direction of assignment

```
A <- 12
A  # 12
A + 3 -> B
B  # 15
24 -> A
A  # 24
```

- The equal sign (=) also works, but only for assignment to the left, e.g.

```
A = 12  # good
12 = A  # BAD
```

R Basics: Strings

- A `String` variable can be declared in either double quotes(`""`) or single quotes (`''`)

```
s <- "This is a valid string"
s
```

```
## [1] "This is a valid string"
```

```
s <- 'and so is this'
s
```

```
## [1] "and so is this"
```

R Basics: Re-Assignments

- A variable can be re-assigned to anything

```r
x <- 860306  # first x is assigned a number
x
```

```
## [1] 860306
```

```r
x <- 'This is a variable!'
x  # Now it is a string
```

```
## [1] "This is a variable!"
```

Vectors

Vectors: c()

- Vectors are the building blocks of R — even a single variable is actually an "atomic" vector (vector of size 1)
- Vectors in R are created by *c*oncatenating a series of elements

```r
X <- c(1,2,3)
X  # vector of numbers (1, 2, 3)
```

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

```r
Y <- c('this', 'that', 'those')
Y  # this is a vector of Strings
```

```
## [1] "this"  "that"  "those"
```

Vectors: `seq()`

- Create a vector from a sequence with `seq(from, to, by=1)`

```
seq(1, 10)
```

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

```
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

- Use short-hand `from:to` if you're incrementing by one

```
1:10
```

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

Vectors: `rep()`

- Use `rep()` to repeat values

```r
rep(13, 4)
```

```
## [1] 13 13 13 13
```

```r
rep('Yes!', 3)
```

```
## [1] "Yes!" "Yes!" "Yes!"
```

```r
rep(c('Sat.', 'Sun.'), 2)
```

```
## [1] "Sat." "Sun." "Sat." "Sun."
```

Vectors: *rdist* ()

- Generate vector of n samples from a specified distribution

```r
runif(n = 10)  # 10 samples from Unif(0, 1)
rnorm(n = 10)  # 10 samples from Norm(0, 1)
rpois(n = 10)  # 10 samples from Poisson(1)
rexp(n = 10)  # 10 samples from Exp(1)
```

- Distribution parameters can be specified as arguments, e.g.

```r
# 100 samples from a Norm(20, 5) distribution
rnorm(n = 100, mean = 20, sd = 5)
```

- Read documentation for available distributions

```r
?Distributions
```

Vectors: Indexing

- Use square braces (`[]`) to index a vector (base 1)

```
X <- c(10, 11, 12, 13)
X[1]
```

```
## [1] 10
```

```
X[4]
```

```
## [1] 13
```

```
X[5]
```

```
## [1] NA
```

Vectors: Indexing (cont'd)

- Negative indexing is used to exclude elements

```
X[-1]
```

```
## [1] 11 12 13
```

- Index multiple objects by indexing with a vector

```
ind <- c(2, 4)
X[ind]
```

```
## [1] 11 13
```

Vectors: Re-assignment with Indices

- Replace elements by re-assigning with index

```r
X[1] <- 101
X
```

```
## [1] 101  11  12  13
```

- Replace multiple elements as well

```r
X[2:3] <- c(22, 33)
X
```

```
## [1] 101  22  33  13
```

Vectors: Add Elements by Index

- Add new elements to a vector by assigning

```
X[5]
```

```
## [1] NA
```

```
X[5] <- 555
X
```

```
## [1] 101  22  33  13 555
```

Vectors: Advanced Indexing

- Vectors can be indexed by a binary vector (`TRUE/FALSE`) of equal length
- i.e., you can index vectors by a specified condition, e.g.,

```
X <- 1:100
# create a binary vector with the same length of X
# where the element is TRUE if the element of X
# in the corresponding position satisfies condition
ind <- X > 95
tail(ind)  # take a peek at the last few entries
```

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

```
X[ind]
```

```
## [1]  96  97  98  99 100
```

Matrices

Creating Matrices

- A matrix is created from a vector, using `matrix()`, e.g.

```r
X <- c(1:12)
# syntax: matrix(vector, # of rows, # of columns)
A <- matrix(X, 3, 4)
A
```

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

- Notice that the matrix is created column-first

Matrix Indexing

- Similar to vectors, a matrix can be indexed with square braces, with the syntax `[row #, col #]`, e.g.
- Leaving an entry empty will result in the full row/column

```
A[3,2]  # third row, second column
```

```
## [1] 6
```

```
A[2,]   # entire second row
```

```
## [1]  2  5  8 11
```

```
A[,4]   # entire fourth column
```

```
## [1] 10 11 12
```

Vector/Matrix Operations

Vector Operations

```r
X = c(1:4)
t(X)  # transpose (column) vector X to row vector
```

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

```r
X + X  # element-wise summation
```

```
## [1] 2 4 6 8
```

```r
X - X  # element-wise subtraction
```

```
## [1] 0 0 0 0
```

Vector Operations (cont'd)

```
X^3   # element-wise exponentiation
```

```
## [1]  1  8 27 64
```

```
X * X  # element-wise multiplication
```

```
## [1]  1  4  9 16
```

```
X %*% X  # dot (inner) product
```

```
##      [,1]
## [1,]   30
```

Matrix Operations

```
A = matrix(1:4, 2, 2)  # create 2x2 matrix
t(A)  # transpose (column) vector A to row vector
```

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

```
A + A  # element-wise summation
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

```
A - A  # element-wise subtraction
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
```

Matrix Operations (cont'd)

```
A^3   # element-wise exponentiation
```

```
##      [,1] [,2]
## [1,]    1   27
## [2,]    8   64
```

```
A * A  # element-wise multiplication
```

```
##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

```
A %*% A  # dot (inner) product
```

```
##      [,1] [,2]
## [1,]    7   15
## [2,]   10   22
```

Matrix Operations: Warning

- Dimensions must make sense!

```r
A <- matrix(1:6, 2, 3)
B <- matrix(1:6, 3, 2)
A %*% B  # 2x3 times 3x2: OK
```

```
##      [,1] [,2]
## [1,]   22   49
## [2,]   28   64
```

```r
A %*% A  # 2x3 times 2x3: Nope
```

```
## Error in A %*% A: non-conformable arguments
```

Vector/Matrix Comparisons

- Comparisons are all done element-wise

```r
c(1, 2, 3) == c(1, 2, 4)
```

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

```r
c(1, 2, 3) < c(1, 2, 4)
```

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

```r
c(1, 2, 3) >= c(1, 2, 4)
```

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

- Note the double equal sign for comparing equality (one would be assignment!)

Helpful Vector Functions

- If possible, avoid `loops` by operating over the Vector/Matrix as a whole

```
mean(X)                     # mean
sd(X)                       # standard deviation
var(X)                      # variance
max(X)                      # maximum
min(X)                      # minimum
median(X)                   # median
sum(X)                      # sum
prod(X)                     # product
quantile(X,probs=0.5)       # quantile for specified probs
length(X)                   # length of the vector
range(X)                    # range
```

Helpful Matrix Functions

- If possible, avoid `loops` by operating over the Vector/Matrix as a whole

```
rowSums(A)    # Row sums
colSums(A)    # Column sums
rowMeans(A)   # Row means
colMeans(A)   # Columns means
diag(A)       # Diagonal of a matrix
solve(A)      # Inverse of a matrix
cov(A)        # Variance covariance matrix
cor(A)        # Correlation matrix
```

Functions

Some more built-in functions

- We've already seen many built-in functions, but here are some more!

```
log(X)   # element-wise log
exp(X)   # element-wise exponential
sqrt(X)  # element-wise square root
```

Functions for Strings

```r
# concatenate two (or more) strings
paste('one plus one equals', 1+1, '!')
```

```
## [1] "one plus one equals 2 !"
```

```r
# specify a separator
paste('one plus one', 1+1, sep='=')
```

```
## [1] "one plus one=2"
```

```r
# if you're into C-style formatting ...
sprintf('one plus one = %d', 1+1)
```

```
## [1] "one plus one = 2"
```

Functions for Strings (cont'd)

- Often, we want to concatenate strings with no spaces (e.g., when constructing filenames/paths in run-time)

```
# short-hand for concatenation w/o spaces
filename = 'some_file_name.csv'
paste0('path/to/', filename)
```

```
## [1] "path/to/some_file_name.csv"
```

```
# function specifically for constructing file paths
file.path('path', 'to', filename)
```

```
## [1] "path/to/some_file_name.csv"
```

Functions for Strings (cont'd)

- To enforce upper/lower cases

```
s <- 'SoMe CraZY STRING'
tolower(s)
```

```
## [1] "some crazy string"
```

```
toupper(s)
```

```
## [1] "SOME CRAZY STRING"
```

Generic Functions

- Some functions for exploring objects

```r
obj <- 1:100
head(obj, n=5)  # display first n rows of obj
```

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

```r
tail(obj, n=5)  # display last n rows of obj
```

```
## [1]  96  97  98  99 100
```

Generic Functions (cont'd)

```r
str(obj)  # display structure of obj
```

```
##  int [1:100] 1 2 3 4 5 6 7 8 9 10 ...
```

```r
summary(obj)  # display summary of obj
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0    25.8    50.5    50.5    75.2   100.0
```

Control Flow

- `if` statements

```
if (condition) {
  # stuff to do when condition is TRUE
} else if(other_condition) {  # (OPTIONAL)
  # stuff to do if other_condition is TRUE
} else {  # (OPTIONAL)
  # stuff to do if all other conditions are FALSE
}
```

- Some consider the catch-all `else` clause to be dangerous
  - Some style guides even recommend avoiding the use of `else` for better readability
- Best practice is to make conditions explicit whenever possible

Loops: Note!

- Loops in `R` are inefficient
- For most cases, there will be a much faster, *vectorized* alternative to looping
- We cover loops because there are some *rare* cases in which a loop might make more sense, but in general, loops should be avoided when writing `R`

Loops

- `for` statements

```
for (ind in sequence/set) {
  # iterate over sequence or elements of a set
  # do stuff
}
```

- `while` statements

```
while (condition) {
  # stuff to do while the condition is TRUE
  # the condition must become FALSE at some point!
}
```

Loops: Example

```r
for (i in 1:3) {
  print(paste('iteration', i))
}
```

```
## [1] "iteration 1"
## [1] "iteration 2"
## [1] "iteration 3"
```

```r
while (i >= 0) {
  print(paste('de-iteration', i))
  i <- i - 1  # beware of infinite loops!
}
```

```
## [1] "de-iteration 3"
## [1] "de-iteration 2"
## [1] "de-iteration 1"
## [1] "de-iteration 0"
```

User Defined Functions

- Write your own functions in the form

```
name_of_function <- function(arguments) {
  # do some stuff with arguments
  return(result)
}
```

- You can use your functions like any other function, e.g.,

```
name_of_function(arguments)  # gives you the 'result'
```

User Defined Functions: Example

- Write a function that will take a vector in $\mathbb{R}^3$ and tell you if you can make a triangle or not

```
is_good <- function(vec) {
  if (length(vec) != 3) {
      # it's always a good idea to make sure your
      # function gets what it expects to get
      stop('is_good requires a vector of length 3')
  }
  for (i in 1:3) {
    if (vec[i] > sum(vec[-i])) {
      return(FALSE)
    }
  }
  return(TRUE)
}
```

apply

- Loops in R are inefficient, and best avoided if possible
- Vectorize operations whenever possible.
- Using apply to *loop* over rows/columns of matrix/table is considerd best practice (in terms of clarity)

```
apply(DATA, MARGIN, FUNCTION)
# MARGIN = 1 applies FUNCTION to each row of DATA
# MARGIN = 2 applies FUNCTION to each column of DATA
# MARGIN = c(1,2) applies FUNCTION to both
#    rows and columns of DATA
```

- e.g., to get the variance of each row/column of a matrix X,

```
apply(X, 1, var)  # variance of rows
apply(X, 2, var)  # variance of columns
```

Exercise

The Question

You are given three sticks, each of a random length between 0 and 1.

What's the probability you can make a triangle?

- The answer is $1/2$
- Use `R` to simulate 100,000 times and estimate the answer by
  1. generate 100,000 triplets of uniform (0, 1) random variables
  2. find the portion that can be made into a triangle (hint: use the `is_good` function)

Answer 1: A (not too good) Way

```
system.time({  # measure execution time
  N <- 1e5;
  m <- 0
  for (i in 1:N) {
    X <- runif(3)
    if (is_good(X)) {
        m <- m + 1
    }
  }
  cat(sprintf('%.2f%%\n', m / N * 100))
})
```

```
## 49.91%
```

```
##    user  system elapsed
##   0.714   0.005   0.737
```

Answer 2: Preferred R Way

```
system.time({  # measure execution time
  N <- 1e5;
  X <- matrix(runif(N*3), nrow=3, ncol=N)
  m <- apply(X, 2, is_good)
  cat(sprintf('%.2f%%\n',  mean(m) * 100 ))
})
```

```
## 50.12%
```

```
##    user  system elapsed
##   0.575   0.005   0.583
```

Packages

Installing R Packages

- R has many (*MANY*) packages created by other users that implement state-of-the-art tools (e.g., data manipulation, statistical models)
- These packages can be downloaded from the Comprehensive R Archive Network (CRAN)
- This is as simple as running a single line of code:

```
install.packages("package name")
```

- You will have to select one of many CRAN mirrors (copies across different servers) from which to download the package from
- For example, to install the `tidyverse` package, run

```
install.packages("tidyverse")
```

- You only need to do this *once* for each machine

Loading Packages

- Once you've installed a package on a machine, you can load the package into your current workspace with the `library()` command
- For example, to use the `tidyverse` package, first load it with

```
library("tidyverse")
```

- Note: If you ever need to use the two packages `plyr` and `tidyverse` within the same workspace (which you will), ALWAYS (!!!) load `plyr` first!