Rtutorial-part1

March 10, 2025

1 R - overview

- Programming language and environment for statistics and data analysis
- Platforms: Linux, MacOS X, Windows
- Published under GNU General Public License (GPL); i.e., freely available (see www.r-project.org)
- Command-line; interpreter (similar to Python)
- Object oriented
- Own programs can easily be integrated
- Extensive statistics library
- Very powerful graphics package

Getting started manual: An Introduction to R https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

by W. N. Venables, D. M. Smith and the R Core Team

1.1 R - basics

[ ]:
```
# expressions: evaluated, displayed (implied print), value lost
1+1
```

[ ]:
```
# assignments: variable <- object
# assignment operator in R is "<-", most of the time "=" can be used instead
# right hand side is evaluated, value assigned to variable
y <- 1+1
(y<- 1+1)                      # print result
z <- cos(pi); cos(pi) -> z     # "->" works in both ways
print(z)
```

- Beginning of a comment is indicated by a hash sign "#"
- Commands are case sensitive
- Commands are separated by a semicolon (";"), or by a newline
- Command history: use up/down arrow keys Previously executed commands can be edited and executed again
- Command editing analogous to Linux shell
- To search in the command history, type ctrl-r

1.1.1 Arithmetic operators

```
^    raise to the power
**   raise to the power
*    multiplication
/    division
+    adding
-    subtracting
```

1.1.2 Integer and modulo division

%/% integer division %% modulo division

- The Integer operator %/% divides the numbers and returns the integer part of the quotient, discarding the remainder.
- The modulo operator %% returns the remainder of the division of two numbers.

Example:

[ ]:
```
### %/%
result <- 10 %/% 3
result
```

[ ]:
```
### %%
result <- 10 %% 3
result
```

1.1.3 Defining vectors in R

In R, vectors are one of the basic data structures and can be defined using the c() function, which combines values into a vector.

Here are some examples:

[ ]:
```
# Define a numeric vector (contains numbers)
numeric_vector <- c(1, 2, 3, 4, 5)
print(numeric_vector)
```

[ ]:
```
# Define a character vector (contains strings of characters)
character_vector <- c("apple", "banana", "cherry")
print(character_vector)
```

[ ]:
```
# Define a logical vector (contains Boolean values (TRUE or FALSE) ).
logical_vector <- c(TRUE, FALSE, TRUE)
print(logical_vector)
```

[ ]:
```
# You can also create a vector by repeating elements using the rep() function.
# e.g. Repeat the number 5, 10 times:
repeated_vector <- rep(5, 10)
print(repeated_vector)
```

You can access elements in a vector using indexing. R uses 1-based indexing.

e.g.

[ ]:
```
# Access the first element of numeric_vector
first_element <- numeric_vector[1]
print(first_element)

# Access the second and third elements of character_vector
subset_vector <- character_vector[2:3]
print(subset_vector)
```

1.1.4 Logical constants, operators, and functions

`TRUE, T`	`true`
`False, F`	`false`
`==, !=`	`equal, not equal`
`<, <=`	`smaller, smaller or equal`
`>, >=`	`larger, larger or equal`
`!`	`not`
`&, %%`	`and`
`|, ||`	`or`
`xor()`	`exclusive or`
`all()`	`tests whether all its arguments are TRUE`
`any()`	`tests whether at least one of its arguments is TRUE`

Examples:

[ ]:
```
### The xor() function performs an exclusive OR operation, which means it␣
↪returns TRUE if one and only one of the arguments is TRUE, and FALSE␣
↪otherwise.
result1 <- xor(TRUE, FALSE)   # TRUE because one is TRUE and the other is FALSE
result2 <- xor(TRUE, TRUE)    # FALSE because both are TRUE
result3 <- xor(FALSE, FALSE)  # FALSE because both are FALSE

print(result1)  # Should print TRUE
print(result2)  # Should print FALSE
print(result3)  # Should print FALSE
```

[ ]:
```
### The all() function tests whether all of its arguments are TRUE. It returns␣
↪TRUE if all arguments are TRUE, and FALSE otherwise.
vec1 <- c(TRUE, TRUE, TRUE)
vec2 <- c(TRUE, FALSE, TRUE)

result4 <- all(vec1)  # TRUE because all elements in vec1 are TRUE
result5 <- all(vec2)  # FALSE because not all elements in vec2 are TRUE

print(result4)  # Should print TRUE
```

```r
print(result5)  # Should print FALSE
```

Using paste() or paste0() allows you to easily include variables and text together in a print statement.

[ ]:
```r
# Printing text with the result
print(paste("The output for result5 is:", result5))  # Using paste()
```

1.1.5 Vector/matrix operators and functions

`%*%`	inner product
`%o%`	outer product
`dim(), ncol(), nrow()`	number of columns and rows
`diag()`	reading/setting matrix diagonale
`eigen()`	eigenvalues/eigenvectors
`solve()`	inverting a matrix
`tr()`	transpose
`colSums(), colMeans()`	sums, arithmetic means of all matrix columns
`rowSums(), rowMeans()`	sums, arithmetic means of all matrix rows

Example:

[ ]:
```r
### How to perform an inner product operation using R

# Define two vectors
a <- c(1, 2, 3)
b <- c(4, 5, 6)

# Calculate the inner product
inner_product <- a %*% b

# Print the result
print(inner_product)
```

This is because the inner product of vectors **a** and **b** is calculated as:

$$(1 \times 4) + (2 \times 5) + (3 \times 6) = 4 + 10 + 18 = 32$$

The inner product is a single scalar value, but R returns it in a 1x1 matrix form by default.

You can convert this result to a scalar if needed:

[ ]:
```r
scalar_inner_product <- as.numeric(inner_product)
print(scalar_inner_product)
### which will print:
```

1.1.6 Pre-defined functions (many more!)

`max(), min()`	minimum/maximum value

```
abs()                 absolute value
sqrt()                square root
round()               rounding
sum(), prod()         sum, product
log(), log10()        logarithms
exp()                 exponential function
sin(), cos(), tan()   trigonometric functions
mean(), var(), sd()   mean, variance, standard deviation
```

1.1.7 Creating your own functions

```
Syntax:
FunctionName <- function(arguments){
# Commands here
return(value)    # return is optional, otherwise last evaluated expression is returned
}
```

- R differs from Python: function definition is an expression
- R differs from Python: `return()` is a function (not a statement)
- R differs from Python: curly braces used to define command groups, and scopes (not indentation)
- scoping rules for variables similar to Python

[ ]:
```
# example:
TimesTwo <- function(z){
y <- 2.0 * z
}

print(TimesTwo(3))
```

1.1.8 Passing arguments to functions

```
Definition:
f <- function(a,b,c=3){
  # commnds here
}

Envocation:
result <- f(1,2,3)
result <- f(c=3, a=1, b=2)    # equivalent to above
result <- f(1,2)              # assuming the default c=3
```

- assignments to variables in function body are local
- using `<<-` instead of `<-`allows one to change variables in host environment (global variable assignment)

Example:

[ ]:
```
# Define the function
f <- function(a, b, c = 3) {
  # Local variable assignment
  local_result <- a + b + c
  print(paste("Local result:", local_result))

  # Global variable assignment using <<-
  global_result <<- a * b * c
  print(paste("Global result:", global_result))

  return(local_result)
}
```

[ ]:
```
# Invoke the function with named arguments (equivalent to above)
result1 <- f(a = 2, b = 4, c = 6)

# The assignments to variables inside the function body are local by default.
```

[ ]:
```
# This will work because local_result is returned
print(result1)
```

[ ]:
```
# This will give an error because local_result is not defined in the global␣
 ↪environment
print(local_result)
```

[ ]:
```
# The global_result variable is assigned in the global environment using <<- so␣
 ↪this will return a result
# generally it is a bad idea to assign variabes in the global environment from␣
 ↪within a function, but there are cases when you might want to do it
print(global_result)
```

1.1.9 Miscellaneous commands

`getwd()`	get working directory
`setwd("[PATH]")`	set working directory
`system("[COMMAND]")`	issue system command
`source("[FILENAME]")`	execute commands from file
`source(..., echo = TRUE)`	execute and print commands from file
`sink("[FILENAME]")`	divert output to file
`sink()`	end diverting

[ ]:
```
getwd()
```

1.1.10 Recommended Resources

- `?[command]`, `help()`, `help.start()`, `help.search()`, `apropos()`
- Venables et al (20XX), *An introduction to R* (see start of notebook here)

- R reference manual (use table of contents, index, or full text search)
- Website describing tuning R code: Efficient R programming by Gillespie & Lovelace
- Base R cheat sheet

1.2 Plotting in R

Plotting is a good way to get a first impression of a dataset. It is the quickest way to get a feel of what the raw data set looks like. To do this, one needs to have some basic information like the size of a dataset, what kind of variables (continuous, discrete, categorial, binary) are in there, and what range the variables cover. After creating an appropriate plot one can get a first idea about trends, notice outliers, correlations, peaks, and what not. Since plotting is easy in R one can just experiment until one gets something reasonable.

1.2.1 Plotting commands

Conceptually, there is a hierarchy of plotting commands in R:

- **High-level plotting** functions create a new plot on the graphics device; usually with axes, labels, titles, etc.
- **Low-level plotting** functions add more information to an existing plot, such as extra points, lines, and labels
- **Interactive** graphics functions allow you to interactively add information to, or extract information from, a graphics window, using the mouse
- In **jupyter notebooks**, interactive graphics is typically not working with inline graphics; moreover, all commands for creating a plot should go into the same cell

[ ]:
```
# perhaps most important: plot() for x-y-plots
x <- seq(1.0, 5.0, 0.5)
y <- x^2
plot(x,y)
```

[ ]:
```
# for a line plot, specify the "type"
# one may also add color information
plot(x, y, type='l', col='red')
lines(x, 25-y, col='green')          # low level command for adding a line
                                     # try also points()
```

[ ]:
```
# barplots are popular in statistics
fruit  <- c('apple', 'banana', 'cherry', 'pear')
amount <- c(20, 10, 70, 30)
barplot(amount, names.arg=fruit, ylab='Amount')
```

[ ]:
```
# sometimes one wants this transposed
barplot(amount, names.arg=fruit, xlab='Amount', horiz=TRUE)
```

[ ]:
```
# histograms are often used in statistics
x <- rnorm(5000)  # create 5000 random values distributed according to a Normal␣
 ↪(Gaussian) distribution
```

```
# Don't panic (we will cover probability distributions later)
hist(x)
```

[ ]:
```
# often one wants not frequencies but the probability density
hist(x, prob=TRUE)
```

[ ]:
```
# 2D plots
y <- x <- seq(-2*pi,2*pi,length.out=100)
xy <- expand.grid(x,x)                            # create 2D (x,y)coordinates
z  <- matrix(sin(xy[,1])*cos(xy[,2]),100,100)    # chess board patters
image(x, y, z, main='False color image')
contour(x, y, z,main='Contour plot')
persp(x, y, z, theta=30, phi=45, main='3D surface plot')
```

2 Reading Data Files in R

Let us quickly see how to read data from different types of files in R, for example CSV and text files.

2.1 Reading CSV Files

CSV (Comma-Separated Values) files are a common format for storing tabular data. You can read a CSV file in R using the `read.csv()` function.

2.1.1 Example

"'r # Reading a CSV file data <- read.csv("path/to/your/file.csv")

3 Display the first few rows of the data

head(data)

3.1 Reading Text Files

Text files can be read using the `read.table()` function. This function can handle various delimiters, such as spaces, tabs, or commas.

3.1.1 Example

"'r # Reading a text file with space-separated values data <- read.table("path/to/your/file.txt", header = TRUE)

4 Display the first few rows of the data

head(data)

4.1 R - keeping track of things

R (like Python) is not stictly typed. Moreover, the return type of functions in R is not always what one would expect. Hence, sometimes one is loosing track what kind of variable one is working with. Some functions in R help to keep track:

```
class()                    object class
mode()                     type of object, e.g., "numeric", "function", "list"
is.vector(), ismatrix()    vector, matrix?
length()                   length of a vector
dim()                      dimensions of multi-D arrays
names(), dimnames()        provides/sets names in objects
summary()                  prints summary of an object
```

[ ]:
```
summary(fruit)
```

4.2 General remarks on programming exercises

- For each exercise, write all commands that you want to execute into one or more files, and load it/them with source(). In a notebook, sort things in cells.
- Name your files or notebooks in a sensible way. Examples:
  - good: verify_central_limit_theorem.R
  - bad: test (note the file extensions)
- Save and keep all the data and program files that you produce during the exercises since you may need them again in later exercises.
- Write your programs step by step. Start with the core tasks, then increase the functionality and complexity stepwise.
- During each programming step, carefully test your program. Use the print() and cat() functions to produce screen output.
- Include comments in your programs, so that after 10 years you will still be able to understand how your program works.
- I will not *immediately* help you debugging your programs; you must learn how to do this yourself. However, I will answer general questions about R, and assist with debugging in desperate cases.

[ ]:
```

```