



Introduction To R

Lesson 01 – R Language

Lesson Objectives

- Introduction to R & Installation on Windows
- Understanding R Concepts and Syntax
 - Data types, arithmetic operations
 - Subset R objects using the "[", "[[", and "\$" operators and logical vectors
 - Explicit coercion feature of R
 - Remove missing (NA) values from a vector



Introduction to R : What is R ?

- R is a system for statistical computation and graphics
- It consists of ...
 - A language
 - Run-time environment with graphics
 - Debugger
 - Access to certain system functions
 - Ability to run programs stored in script files

Introduction to R : Installing R on Windows

- Download R from <https://cran.r-project.org/>
- And execute it to install.
- The bin/windows directory of a CRAN site contains binaries for a base distribution and add-on packages from CRAN to run on Windows 7 and later (including 64-bit versions of Windows)
- During Installation **choose between MDI or SDI interface. (SDI is selected)**
- Set your working directory

Introduction to R : Bit of Hands-On with R using Console

- Set your working directory.
- Bit of Hands-On with R
 - We will discuss go aheadit in-depth as we

Understanding R : R Input and Evaluation

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.

Understanding R : R Input and 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.

Understanding R : R Input, Evaluation and 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.

Understanding R : Data Types - R Objects and Attributes

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)

Empty vectors can be created with the `vector()` function.

Understanding R : Data Types - R Objects and Attributes

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)

Empty vectors can be created with the `vector()` function.

Understanding R : Data Types - 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)

Understanding R : Data Types - 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.

Understanding R : Data Types - Vectors and Lists

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
```

Understanding R : Data Types - Vectors and Lists

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.

Understanding R : Data Types - Vectors and Lists

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 TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Understanding R : Data Types - Vectors and Lists

Nonsensical coercion results in NAs.

```
> 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] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```


Understanding R : Data Types - Vectors and 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
```

Understanding R : Data Types - 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
```

Understanding R : Data Types - Matrices

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

Understanding R : Data Types - Matrices

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
```

Understanding R : Data Types - Matrices

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
      x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
      [,1] [,2] [,3]
x         1     2     3
y        10    11    12
```

Understanding R : Data Types - 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.

Understanding R : Data Types - 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"
```

Understanding R : Data Types - 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
```


Understanding R : Data Types - 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

Understanding R : Data Types - 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
```

Understanding R : Data Types - 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()`

Understanding R : Data Types - 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
```

Understanding R : Data Types - Names Attribute

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"
```

Understanding R : Data Types - Names Attribute

Lists can also have names.

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

$b
[1] 2

$c
[1] 3
```

Understanding R : Data Types - Names Attribute

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
```

Understanding R : Data Types - Summary

Data Types

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

Understanding R : Reading Tabular Data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (inverse of `dump`)
- `dget`, for reading in R code files (inverse of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

Understanding R : Writing Data

There are analogous functions for writing data to files

- `write.table`
- `writeLines`
- `dump`
- `dput`
- `save`
- `serialize`

Understanding R : Reading Tabular Data

The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are separated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset
- `comment.char`, a character string indicating the comment character
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, should character variables be coded as factors?

Understanding R : Reading Tabular Data

For small to moderately sized datasets, you can usually call `read.table` without specifying any other arguments

```
data <- read.table("foo.txt")
```

R will automatically

- skip lines that begin with a `#`
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table Telling R all these things directly makes R run faster and more efficiently.
- `read.csv` is identical to `read.table` except that the default separator is a comma.

Understanding R : Reading Large Tables

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for `read.table`, which contains many hints
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.

Understanding R : Reading Large Tables

- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick and dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                    colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

Understanding R : Reading Large Tables

In general, when using R with larger datasets, it's useful to know a few things about your system.

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?

Understanding R : Reading Large Tables

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?

$$1,500,000 \times 120 \times 8 \text{ bytes/numeric}$$

$$= 1440000000 \text{ bytes}$$

$$= 1440000000 / 2^{20} \text{ bytes/MB}$$

$$= 1,373.29 \text{ MB}$$

$$= 1.34 \text{ GB}$$

Understanding R : Textual Data Formats

- `dumping` and `dputing` are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- Unlike writing out a table or csv file, `dump` and `dput` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- `Textual` formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the "Unix philosophy"
- Downside: The format is not very space-efficient

Understanding R : Textual Data Formats

Another way to pass data around is by deparsing the R object with `dput` and reading it back in using `dget`.

```
> y <- data.frame(a = 1, b = "a")
> dput(y)
structure(list(a = 1,
               b = structure(1L, .Label = "a",
                             class = "factor")),
          .Names = c("a", "b"), row.names = c(NA, -1L),
          class = "data.frame")
> dput(y, file = "y.R")
> new.y <- dget("y.R")
> new.y
  a  b
1 1 a
```

Understanding R : Textual Data Formats

Multiple objects can be deparsed using the dump function and read back in using `source`.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
  a b
1 1 a
> x
[1] "foo"
```

Understanding R : Connections: Interfaces to the World

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

Understanding R : Connections: Interfaces to the World

```
> str(file)
function (description = "", open = "", blocking = TRUE,
          encoding = getOption("encoding"))
```

- `description` is the name of the file
- `open` is a code indicating
 - “r” read only
 - “w” writing (and initializing a new file)
 - “a” appending
 - “rb”, “wb”, “ab” reading, writing, or appending in binary mode (Windows)

Understanding R : Connections: Interfaces to the World

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
con <- file("foo.txt", "r")  
data <- read.csv(con)  
close(con)
```

is the same as

```
data <- read.csv("foo.txt")
```

Understanding R : Connections: Interfaces to the World

```
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
[1] "1080"      "10-point" "10th"      "11-point"
[5] "12-point"  "16-point" "18-point"  "1st"
[9] "2"         "20-point"
```

`writeLines` takes a character vector and writes each element one line at a time to a text file.

Understanding R : Connections: Interfaces to the World

`readLines` can be useful for reading in lines of webpages

```
## This might take time
con <- url ('http://www.capgemini.com','r')
x <- readLines(con)
> head(x)
[1] "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">"
[2] ""
[3] "<html>"
[4] "<head>"
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8"
```


Understanding R : Subsetting - Basics

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 `[[`.

Understanding R : Subsetting - Basics

```
> 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"
```

Understanding R : Subsetting - Lists

6/14

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
```

Understanding R : 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"
```

Understanding R : 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
```

Understanding R : Subsetting - Lists

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
```

Understanding R : Subsetting - Matrix

Matrices can be subsetting 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
```

Understanding R : Subsetting - Matrix

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 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
```


Understanding R : Subsetting - 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
```

Understanding R : Subsetting - 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
```

Understanding R : Subsetting - Removing Missing 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
```

Understanding R : Subsetting - Removing Missing 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"
```

Understanding R : Subsetting - Removing Missing 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

Understanding R : Vectorized Operations

Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x + y
[1] 7 9 11 13
> x > 2
[1] FALSE FALSE TRUE TRUE
> x >= 2
[1] FALSE TRUE TRUE TRUE
> y == 8
[1] FALSE FALSE TRUE FALSE
> x * y
[1] 6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Lesson Summary

- Introduction to R & Installation on Windows
- Understanding R Concepts and Syntax
 - Data types, arithmetic operations
 - Subset R objects using the "[", "[[", and "\$" operators and logical vectors
 - Explicit coercion feature of R
 - Remove missing (NA) values from a vector

