R Syntax Examples

Nathan Young

R Nuts and Bolts

Entering Input

We type expressions into the R prompt. The <- symbol is the assignment operator

```r
x <- 5 # Assignment of value to variable
print(x) # Explicit printing
```

```
## [1] 5
```

```r
x # auto-printing
```

```
## [1] 5
```

```r
y <- 10:20
```

When [1] is printed it indicates that x is a vector and that 5 is the first element The : operator is used to create integer sequences

R Objects

R has five basic classes of objects - Character - Numeric (real numbers) - Integer - Complex - Logical (True/False)

The vector is the most basic object type, initialized with vector() function.
A vector can only contain objects of the same class A *list* is represented as a vector but can contain objects of different classes.

Numbers

Numbers are generally treated as numeric objects (such as double precision real numbers). Meaning, even if you see a '1' it is represented behind the scene as a numeric object like '1.00'. If you explicitly want an integer, use the L suffix. Entering '1' gives you a numeric object, while entering '1L' gives an integer object. Inf represents infinity. NaN represents an undefined value such as 0/0

Attributes

R Objects can have many attributes (like metadata) to help describe the object. * Names, dimnames * dimensions (matrices, arrays) * class (integer, numeric) * length * other user-defined attributes/metadata

Attributes can be accessed using attributes() function.

Creating Vectors

The c() function can be used to create vectors of objects by concatenating things together.

```
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
x <- vector("numeric", length = 10) # initialize vector
```

Mixing Objects

Sometimes we mix objects (on accident or on purpose).
When different objects are mixed in a vector, they are *coerced* to be the same class. Sometimes this works the way you expect others, not.

Explicit Coercion

You can explicitly coerce between classes using the as.* function

```
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"
```

Sometimes, R doesn't know what to do and will return NA.

Matrices

Vectors with a *dimension* attribute. The dimension attribute is an integer vector of length 2

```
m <- matrix (nrow = 2, ncol = 3)
dim(m)
```

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

```r
attributes(m)
```

```
## $dim
## [1] 2 3
```

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

```r
m <- matrix(1:6,nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

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

```r
m <- 1:10
m
```

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

```r
dim(m) <- c(2,5)
m
```

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

Can be created with *column-binding* or *row-binding*

```r
x <- 1:3
y <- 10:13
cbind(x,y)
```

```
## Warning in cbind(x, y): number of rows of result is not a multiple of vector
## length (arg 1)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
## [4,] 1 13
```

```r
rbind(x,y)
```

```
## Warning in rbind(x, y): number of columns of result is not a multiple of vector
## length (arg 1)
```

```
##   [,1] [,2] [,3] [,4]
## x    1    2    3    1
## y   10   11   12   13
```

Lists

Special vectors that contain different elements of different classes.
Can be explicitly created with the list() function.

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

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Can also create an empty list of length with the vector() function.

```
x <- vector("list", length = 5)
```

Factors

Used to represent categorical data that is unordered or ordered. Akin to integer vector where each integer has a label. Important for statical modeling and treated special in functions like lm() and glm().
Using factors with labels is better than using integers because factors are self-describing. Factor with values "Male" and "Female" is better than variable with values 1 and 2. Create with factor() function

```
x <- factor(c("yes", "yes", "no", "yes", "yes"))
x
```

```
## [1] yes yes no  yes yes
## Levels: no yes
```

```
table(x)
```

```
## x
##  no yes
##   1   4
```

```
unclass(x)
```

```
## [1] 2 2 1 2 2
## attr(,"levels")
## [1] "no"  "yes"
```

Often, factors are automatically created when reading data with function like data.table().
Order of levels can be set using levels argument to factor(). Often important in linear modeling with first level as baseline.

Missing Values

Denoted by NA or NaN for undefined mathematical operations - is.na() is used to test for NA - is.nan() is used to test for NaN - NA values have a class also (integer NA, character NA, etc) - A NaN value is also NA

```
# create a vector with NAs in it
x <- c(1,2,NA,10,NaN,3)
# Return logical vector indicating which elements are NA
is.na(x)
```

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

```
# Return logical vector indicating which elements are NaN
is.nan(x)
```

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

Data Frames

Used to store tabular data in R. Important in many statistical modeling. dplyr has optimized set of functions to work with data frames.
Represented as special type of list where everyelement of the list has to have the same length. Each element can be thought of as a column and the length of each element of the list is the number of rows.
Unlike matrices, can store different classes of objects in ach column.
In addition to column names, DF have attribute called row.names to indicate informration about each row.
Usually created by reading a dataset using read.table() or read.csv(). Can be created with data.frame() or converted from other objects.
Can be converted to matrix with data.matrix().

```
x <- data.frame(foo = 1:4, bar = c(TRUE, TRUE, FALSE, FALSE))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```
nrow(x)
```

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

```
ncol(x)
```

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

Names

R objects can have names, helpful for readable code.

```
x <- 1:3
names(x)
```

```
## NULL
```

```
names(x) <- c("New York", "Seattle", "Los Angeles")
x
```

```
##    New York     Seattle Los Angeles
##           1           2           3
```

```
names(x)
```

```
## [1] "New York"    "Seattle"     "Los Angeles"
```

```
# Lists can also have names
x <- list("Los Angeles" = 1, Boston = 2, London = 3)
x
```

```
## $`Los Angeles`
## [1] 1
##
## $Boston
## [1] 2
##
## $London
## [1] 3
```

```
names(x)
```

```
## [1] "Los Angeles" "Boston"      "London"
```

```
# Matrices can have both row and column names
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
```

```
# Column names and row names can be specified separately
colnames(m) <- c("h","f")
rownames(m) <- c("x","z")
m
```

```
##   h f
## x 1 3
## z 2 4
```

Note that for dataframes, there is a separate function for setting row names, the row.names(). DF do not have column names, just names. So you use the names() function.

Getting Data In and Out of R

Reading and Writing Data

There are a few principal functions - read.table, read.csv for reading tabluar 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 simple R objects in binary form.

Many packages exist for other datasets

Analogous functions for writing to files - write.table for writing tabular to text files or connections - writeLines, for writing character data line-by-line to file or conn - dump, for dumping a textual representation of multiple R objects - dput, for outputting a textual representation of an R object - save, for saving an arbitrary number of R objects in binary format to a file - serialize, for converting an R object into a binary format for outputting to a connection.

Reading Data Files with read.table()

Commonly used. Help file is worth reading. Arguments: - file, name of file, or connection - header, logical indicating if file has a header - sep, a string indicating how columns are separated - colClasses, character vector indicating the class of each column - nrows, number of rows. Default reads whole file - comment.char, chacter string indicating comment character. - skip, number of lines to skip from beginning - stringsAsFactors, should character variables be coded as factors? defaults to True.

For small to moderate size, you can usually call read.table

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

- In this case, R will automatically skip lines that begin with #
- figure out how many rows there are and how much mem is needed.
- figure out type of variable in each column.

Telling R these things directly makes R faster and more efficient.

Reading in larger data sets with read.table

With larger data sets, make life easier and prevent R from choking by: - Read the help page for read.table - Make a rough calculation of the memory required to store your dataset. If dataset is larger than RAM, stop. - set comment.char = “ ” if no commented lines in file - Use colClasses argument. Specifing this can make it run **much** faster, often twice as fast.

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

- set nrows. Doesn’t make faster but helps with memory.

Also useful to know about your system. - How much mem is available? - Can you close other applications? - Are there other users logged in? - What OS are you using? Some limit amount of memory.

Calculating Memory Requirements for R Objects

Because R stores all objects in physical memory, important to keep limits in mind. Especially when reading in a new dataset to R. BOTE are easy for it.
Suppose data frame with 1,500,000 rows and 120 columns all of which are numeric. How much memory to store numeric data? 1,500,000 x 120 x 8 bytes/numeric = 1,440,000,000 bytes = 1,400,000,000 / 2^20 bytes/MB = 1,373.29 MB = 1.34 GB.

Using the readr Package

Recently developed to deal with reading large flat files quickly, replacement for read.table() and read.csv(). Ananogous in readr are read_table() and read_csv().
For the most part, can use these instead.

```
library(readr)
#teams <- read_csv("data/team_standings.csv")
#teams
```

By default, will open and read lint-by-line. Will also, read first few rows of table to figure out type of column.
Can instead specify type of each column with col_types argument. Good idea in general.
col_types accepts a compact representation. "cc" says first and second columns are characters.
read_csv also reads compressed files automatically, no need to decompress.
Can specify column type in detailed fashion using various col_* functions.

Using Textual and Binary Formats for Storing Data

Data can be stored in a variety of ways including strutured forms like CSV. Intermediate format that is textual, but not as simple as CSV.
One can create a more descriptive representation of an R object by using the dput() or dump() functions; useful because resulting textual format is editable and recoverable in case of corruption. Preserve metadata (sacrificing readability) so user doesn't have to specify it again.
Work much better with VC programs to track meaningful changes.
Downsides include not space inefficiency and partial readibility.

Using dput() and dump()

One way to pass data around is deparsing the R object with dput() and reading it back in with dget().

```
# Create a data frame
y <- data.frame(a = 1, b = "a")
# Print 'dput' output to console
dput(y)
```

```
## structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA,
## -1L))
```

Notice that the dput() output is in the form of R code and that it preserves meta data like class, row names, and column names.
Output can also be saved directly to a file

```r
# Send 'dput' output to a file
dput(y, file = "y.R")
# Read in 'dput' output from a file
new.y <-dget("y.R")
new.y
```

```
##   a b
## 1 1 a
```

Multiple objects can be deparsed at once using the dump function and read back in using source

```r
x <- "foo"
y <- data.frame(a = 1L, b = "a")
dump(c("x","y"),file = "data.R")
rm(x,y)
# Inverse of dump is source
source("data.R")
str(y)
```

```
## 'data.frame':    1 obs. of  2 variables:
##  $ a: int 1
##  $ b: chr "a"
```

```r
x
```

```
## [1] "foo"
```

Binary Formats

The complement to the textual format is the binary format. Use for efficiency purposes or if no useful way to represent data with text. Also, can lose precision on numeric data when converting to and from textual format.
Convert R object to binary: save(), save.image(), serialize().
Individual objects with save() function

```r
a <- data.frame(x = rnorm(100), y = runif(100))
b <- c(3, 4.4, 1/3)
# Save 'a' and 'b' to a file
save(a, b, file = "mydata.rda")
# Load 'a' and 'b' into workspace
load("mydata.rda")
```

Save lots of objects with save.image()

```r
# Save everything
save.image(file = "mydata.RData")
# Load all objects in this file
load("mydata.RData")
```

Example used .rda for save and .RData for save.image. Can use other formats but these are common and accessible.
serialize() converts individual R objects to binary format that can be communicated across an arbitrary connection (file or network).
Coded as a raw vector in hexadecimal.

```r
x <- list(1, 2, 3)
serialize(x,NULL)
```

```
##  [1] 58 0a 00 00 00 03 00 04 02 02 00 03 05 00 00 00 00 05 55 54 46 2d 38 00 00
## [26] 00 13 00 00 00 03 00 00 00 0e 00 00 00 01 3f f0 00 00 00 00 00 00 00 00 00
## [51] 0e 00 00 00 01 40 00 00 00 00 00 00 00 00 00 0e 00 00 00 01 40 08 00 00
## [76] 00 00 00 00
```

Can be sent to a file, but better of using save for that.
Benefit to serialize: perfectly represent R object in an exportable format, without losing precision or metadata.

Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files or more exotic things. - file, opens connection to file - gzfile, opens connection to file compressed with gzip - bzfile, opens connection to file compressed with bzip2 - url, opens connection to webpage.

Think of connections as translator that lets you talk to objects outside of R.
## File Connections Connections to file using file()

```r
str(file)
```

```
## function (description = "", open = "", blocking = TRUE, encoding = getOption("encoding"),
##     raw = FALSE, method = getOption("url.method", "default"))
```

Arguments:

- description, name of file
- open, mode to open file
    - 'r' read mode
    - 'w' write mote (and initialize new file)
    - 'a' appending
    - 'rb', 'wb', 'ab' readin writing appending in binary

Most of connection is dealt with in background and we don't deal with it.

```r
# Create connection to 'foo.txt'
#con <- file("foo.txt")
# Open connetion in read mode
#open(con, "r")
# Read from connection
#data <- read.csv(con)
# Close connection
```

```r
#close(con)

# this is all the same as:
#data <- read.csv("foo.txt")
```

In the background, read.csv() opens a connection, reads, and closes the connection.

Reading Lines of a Text File

Lines of text file can be read line by line with readLines(). Useful for unstructured and nonstandard data.

```r
# Open connection to gz-compressed file
# con <- gzfile("words.gz")
# x <- readLines(con,10)
# x
```

For more structured text like CSV, other functions like read.csv() and read.table() exits.
In above, gzfile() is used to create connections to compressed data with gzip algorithm. Lets you not unzip files.
Complementary function writeLines() to write character vector element wise to each line.

Reading from a URL Connection

readLines() is useful for reading in lines of webpages that are text files stored on a server. url() function navigates communication between computer and web server.

```r
# Open a url connection for reading
con <- url("https://www.jhu.edu","r")

# Read the webpage
x <- readLines(con)

# Print first few lines
head(x)
```

```
## [1] "<!doctype html>"                    ""
## [3] "<html class=\"no-js\" lang=\"en\">" "  <head>"
## [5] "    <script>"                       "    dataLayer = [];"
```

Reading a simple page is sometimes useful, more commonly used to read data stored on web servers.
Using URL connections can be useful for producing a reproducible analysis, because code documents where the data cam from and how it was obtained. Preferable to opening a webpage and downloading dataset by hand. If server is changed, can break code.

Subsetting R Objects

Three operators to extract subsets of R Objects

- The “[]” operator always returns an object of the same class as the original, can be used to select multiple elements of object
- [[]] operator is used to extract elements of a list or data frame. Used to extract single element and class of returned object will not necessarily be a list or data frame.
- $ operator is used to extract elements of a list by literal name.

Subsetting a Vector

Vectors are basic objects and are subsetted using the [] operator.

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]
```

```
## [1] "a"
```

```
x[2]
```

```
## [1] "b"
```

```
# Can be used to extract multiple elements by passing an inteer sequence.
x[1:4]
```

```
## [1] "a" "b" "c" "c"
```

```
# Sequence doesn't need to be in order
x[c(1, 3, 4)]
```

```
## [1] "a" "c" "c"
```

```
# Can also pass logical sequence. For example, elements of x that come alphabetically after letter a
u <- x > "a"
x[u]
```

```
## [1] "b" "c" "c" "d"
```

```
# More compactly:
x[x > "a"]
```

```
## [1] "b" "c" "c" "d"
```

Subsetting a Matrix

Subsetted in usual way with (i, j) indices. Row x column

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

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
# access elements using indices
x[1,2]
```

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

```r
x[2,1]
```

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

```r
# Missing index indicates entire row/column
x[1,]
```

```
## [1] 1 3 5
```

```r
x[,2]
```

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

Dropping Matrix Dimensions

By default, when you index a single element a vector length 1 is returned. Can turn this off.

```
x <- matrix(1:6, 2, 3)
x[1,2]
x[1, 2, drop = FALSE]
# Similar use to keep matrix dimensions for extracting entire row.
x[1,]
x[1, , drop = FALSE]
```

Be careful of R's automatic dropping of dimensions

Subsetting Lists

```r
x <- list(foo = 1:4, bar = 0.6)
x
```

```
## $foo
## [1] 1 2 3 4
##
## $bar
## [1] 0.6
```

```r
## The [[]] operator can be used to extract a single element from a list
x[[1]]
```

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

```r
## Can also use named indices or $ operator to extract by name
x[["bar"]]
```

```
## [1] 0.6
```

```r
x$bar
```

```
## [1] 0.6
```

```r
# Don't need quotes on $ operator.
# [[ can be used with computed indices, $ operator can only be used with names
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"

# Computed index for "foo"
x[[name]]
```

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

```r
# element name doesn't exist!
x$name
```

```
## NULL
```

Subsetting Nested Elements of a List

The [[]] operator can take an integer sequence

```r
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
# Get 3rd element of the first element
x[[c(1,3)]]
```

```
## [1] 14
```

```r
# Alternative
x[[1]][[3]]
```

```
## [1] 14
```

```r
# first element of second element
x[[c(2,1)]]
```

```
## [1] 3.14
```

Extracting Multiple Elements of a List

The [] operator can extract multiples

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

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

Note that x[c(1,3)] is not the same as x[[c(1,3)]]!

Removing NA Values

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

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

```
# What if there are multiple R objects and you want to take the subset with no missing values of those
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"
```

```
# Can be used on data frames too
head(airquality)
```

```
##   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
```

```r
good <- complete.cases(airquality)
head(airquality[good, ])
```

```
##   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
## 8    19      99 13.8   59     5   8
```