

SIOB 296 Introduction to Programming with R

Eric Archer (eric.archer@noaa.gov)

Week 1: April 4, 2017

Reading

The Book of R:

Chapter 1, pages 3-5 (installing R)

Chapter 2 (Numeric, Arithmetic, Assignment and Vectors)

Chapter 4 (Non-Numeric Values)

Chapter 6, pages 103-114 (Some Special Values)

The Art of R:

Chapters 1, 2, 6.1

R Console

- commands and assignments executed or evaluated immediately
- separated by new line (Enter/Return) or semicolon
- recall commands with ↑ or ↓
- case sensitive

NB: EVERY command is executing some function and returns something

Help

There are several ways of getting help. The most common is just the `help` command:

```
help(mean)
```

This can be shortened to just `?` in most cases:

```
?median
```

For some special functions, topics, or operators, you should use quotes:

```
help("[")
```

The examples in help pages can be run using the `example` function:

```
example(mean)
```

```
mean> x <- c(0:10, 50)
```

```
mean> xm <- mean(x)
```

```
mean> c(xm, mean(x, trim = 0.10))
```

```
[1] 8.75 5.50
```

Finally, if you don't know the name of the function, but you know a keyword, you can use `help.search`:

```
help.search("regression")
```

Workspace

The contents of the workspace can be viewed with `ls()`:

```
ls()
```

```
[1] "x"  "xm"
```

Useful workspace functions

`rm()`: remove an object

`rm(list = ls())`: remove all objects in the workspace

`save.image()`: save all objects in the workspace

`load(".rdata")`: load saved workspace

`history()`: view saved history

`#`: comment

Math

The R console can be used as a powerful calculator where both complex and simple calculations can be made on the fly:

```
4 + 5
```

```
[1] 9
```

```
5 / 23
```

```
[1] 0.2173913
```

```
1 / 1.6 + 1
```

```
[1] 1.625
```

```
(-5 + sqrt(5 ^ 2 - (4 * 3 * 2))) / (2 * 3)
```

```
[1] -0.6666667
```

Other common mathematical operators can be found with `?Arithmetic`.

Data structures

There are six basic storage **modes** that you will encounter in most of your R work:

logical: TRUE, FALSE, T, F

integer: whole numbers (e.g., 1, -1, 15, 0)

double: double precision decimals (e.g., 3.14, 1e-5, 2.0)

character: character strings (e.g., "Hello World", "I love R", "22.3")

list: A collection of objects that can be of different modes

function: A set of commands initiated by a call that takes arguments and returns a value

There are six basic object **classes** that you should become familiar with:

vector: One dimensional, all elements are of same mode

factor: One dimensional, categorical data represented by integers mapped to levels

matrix: Two dimensional, all elements are of same mode

array: Multi-dimensional, all elements are of same mode

list: One dimensional, elements can be of different modes

data.frame: Two dimensional, each column is an element of same length (rows)

Special Values

NULL: Empty object or object does not exist NA: Missing data

NaN: Not a Number (0/0)

Inf / -Inf: Infinity (1/0)

Object Information

str: Display the structure of an object

mode: The storage mode of an object

class: The class of an object

is.<class>: Test if an object is of a given class

Vectors

Objects are assigned values using the “left arrow” (<-) operator, like this:

```
x <- 1
x
```

```
[1] 1
```

You can also use = for assignment, but I seriously recommend not getting into the habit of doing that. It can actually make code harder to read because = is used in a slightly different context. I have found it better to be consistent and stick with <-.

```
# The ':' operator creates a numeric vector incrementing by 1
```

```
x <- 1:10
x
```

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

```
# The `c` function creates a vector containing the arguments inside
```

```
y <- c("a", "b", "d")
y
```

```
[1] "a" "b" "d"
```

```
str(x)
```

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

```
is.numeric(x)
```

```
[1] TRUE
```

```
class(y)
```

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

```
mode(x)
```

```
[1] "numeric"
```

Indexing

There are three ways to index any object in R:

Numeric: Using integers to reference the element number

Character: If the object has “names”, using characters to specify those names

Logical: Return only the elements that match TRUE values

Numeric Indexing

```
x <- 21:30
x

[1] 21 22 23 24 25 26 27 28 29 30
# The fifth element
x[5]

[1] 25
# The first three elements
x[1:3]

[1] 21 22 23
# The first, fifth, and sixth elements
x[c(1, 5, 6)]

[1] 21 25 26
# Numerical indexing returns elements in the order they were requested
x[c(8, 9, 3)]

[1] 28 29 23
# Replication of elements is allowed and will be accommodated
x[c(4, 6, 5, 6, 4)]

[1] 24 26 25 26 24
# Any numeric vector is allowed
x[c(1:4, 5, 10:8)]

[1] 21 22 23 24 25 30 29 28
# Negative numbers return all elements except the negative value
x[-3]

[1] 21 22 24 25 26 27 28 29 30
# Don't fall into this trap
x[-1:5]

Error in x[-1:5]: only 0's may be mixed with negative subscripts
# What you probably mean is this
x[-(1:5)]

[1] 26 27 28 29 30
```

Character Indexing

To use character indexing, you have to provide **names** to the vector

```
names(x) <- letters[1:10]
x

  a  b  c  d  e  f  g  h  i  j
21 22 23 24 25 26 27 28 29 30
```

```
str(x)
```

```
Named int [1:10] 21 22 23 24 25 26 27 28 29 30
- attr(*, "names")= chr [1:10] "a" "b" "c" "d" ...
```

Then, elements can be specified by name

```
x["d"]
```

```
d
24
```

```
x[c("f", "a")]
```

```
f a
26 21
```

Specific names can be changed by referencing the `names(x)` vector

```
names(x)[4] <- "fourth"
x["fourth"]
```

```
fourth
24
```

Logical indexing

The third way to index is using logical vectors. Only elements matching `TRUE` values are returned

```
y <- 1:4
y[c(T, T, F, T)]
```

```
[1] 1 2 4
```

Here are the primary logical operators:

`!` : Not - negates the value (`!T = F`, `!F = T`)

`&` : And - Result is T if both values are T (`T & T = T`, `T & F = F`, `F & F = F`)

`|` : Or - Result is T if one value is T (`T | T = T`, `T | F = T`, `F | F = F`)

`<`, `>` : Less, greater than

`<=`, `>=` : Less than or equal to, greater than or equal to

`==` : Equal to

`!=` : Not equal to

`any()` : Returns T if any value is T

`all()` : Returns T if all values are T

```
x <- 50:20
x
```

```
[1] 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28
[24] 27 26 25 24 23 22 21 20
```

```
x[x < 30]
```

```
[1] 29 28 27 26 25 24 23 22 21 20
```

```
x[x < 40 & x > 25]
```

```
[1] 39 38 37 36 35 34 33 32 31 30 29 28 27 26
```

```
x[x < 25 | x > 43]
```

```
[1] 50 49 48 47 46 45 44 24 23 22 21 20
```

Vectorization

A key component of R operations on vectors is the idea of “vectorization”. In essence, this means that operations between multiple R vectors will tend to recycle elements in the smaller object to the size of the larger object. This is most easily seen in vector algebra:

```
# Add two vectors of equal length
1:5 + 21:25
```

```
[1] 22 24 26 28 30
```

```
# Add two vectors where one is a multiple of the other
1:10 + 1:2
```

```
[1] 2 4 4 6 6 8 8 10 10 12
```

```
# Add two vectors where one is not the multiple of the other
1:10 + 1:3
```

Warning in 1:10 + 1:3: longer object length is not a multiple of shorter object length

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

Vectorization can be used in logical indexing too

```
# Select every other element
x <- 1:10
x[c(T, F)]
```

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

```
# Select every third element
x[c(T, F, F, F)]
```

```
[1] 1 5 9
```

Character vectors

```
x <- c("A", "b", "C")
x[2]
```

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

```
# Add names with vector
y <- 1:3
names(y) <- x
```

```
# Select using logical
x[x == "C"]
```

```
[1] "C"
```

```
x[x != "b"]
```

```
[1] "A" "C"
```

```

# Index one vector with another
x[y == 2]

[1] "b"

# Two special values that provide a vector of lower and upper case letters:
letters

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"

LETTERS

[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
[18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

```

Logical vectors

```

x <- c(T, F, T, F, F, T)
any(x)

[1] TRUE

all(x)

[1] FALSE

# Negate the vector
!x

[1] FALSE TRUE FALSE TRUE TRUE FALSE

# Every other value
x[c(F, T)]

[1] FALSE FALSE TRUE

# Just the TRUE values
x[x]

[1] TRUE TRUE TRUE

# Logical vectorization
x & T

[1] TRUE FALSE TRUE FALSE FALSE TRUE

x | c(F, T)

[1] TRUE TRUE TRUE TRUE FALSE TRUE

```

Factors

Factors are special vectors where the unique values are stored as numbers and mapped to character levels

```

x <- factor(c("yellow", "blue", "green", "blue", "Blue", "yellow"))
x

[1] yellow blue green blue Blue yellow
Levels: blue Blue green yellow

```

```
# Notice that the values are numerics  
str(x)
```

```
Factor w/ 4 levels "blue","Blue",...: 4 1 3 1 2 4
```

```
# ... but the class isn't  
is.numeric(x)
```

```
[1] FALSE
```

```
# ... nor is it character  
is.character(x)
```

```
[1] FALSE
```

```
# Here's the class  
class(x)
```

```
[1] "factor"
```

```
# and the storage mode  
mode(x)
```

```
[1] "numeric"
```

The numeric and original character vectors can be obtained by **coercion** using the `as.<class>` set of functions:

```
as.numeric(x)
```

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

```
as.character(x)
```

```
[1] "yellow" "blue"   "green"  "blue"   "Blue"   "yellow"
```

A factor has both levels and labels. The levels are the set of values that might have existed in the original vector and the labels are the representations of the levels.

```
# The sample function takes a random sample from a vector with or without replacement  
x <- sample(x = letters[1:4], size = 10, replace = TRUE)  
xf <- factor(x)  
xf
```

```
[1] a c b a a b d d d b
```

```
Levels: a b c d
```

```
# Here are the levels  
levels(xf)
```

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

```
# We can change the order of the levels (note doesn't change order of values in vector)  
xf.lvl <- factor(x, levels = c("c", "b", "d", "a"))  
xf.lvl
```

```
[1] a c b a a b d d d b
```

```
Levels: c b d a
```

```
# Adding a level that doesn't exist has no effect on data, but includes level in list of levels  
xf.lvl <- factor(x, levels = c("c", "e", "b", "d", "a"))  
xf.lvl
```

```
[1] a c b a a b d d d b
```



```
Levels: c e b d a
```

```
# Omitting a level causes all values with that level to be NA
```

```
xf.lvl <- factor(x, levels = c("b", "d", "a"))
```

```
xf.lvl
```

```
[1] a    <NA> b    a    a    b    d    d    d    b
```

```
Levels: b d a
```

```
# Labels will match order of levels
```

```
xf.lbl <- factor(x, labels = c("Z", "Y", "X", "W"))
```

```
xf.lbl
```

```
[1] Z X Y Z Z Y W W W Y
```

```
Levels: Z Y X W
```

```
# But you must have as many labels as levels
```

```
xf.lbl <- factor(x, labels = c("Z", "Y", "X"))
```

```
Error in factor(x, labels = c("Z", "Y", "X")): invalid 'labels'; length 3 should be 1 or 4
```