SIOB 296 Introduction to Programming with R

Eric Archer (eric.archer@noaa.gov)

Week 1: January 8, 2019

R console, RStudio, scripts, data structures, vectors, indexing

The R Book

- 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:

```r
help(mean)
```

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

```r
?median
```

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

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

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

```r
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`.

Writing and running scripts

Scripts are text files containing commands and comments written in an order as if they were executed on the command line. They can be executed with `source("filename.r")`, or if loaded into an R editor, run piece by piece or all together. In RStudio, see commands and shortcuts under the Code menu option.
Code style is an important habit to cultivate. Being consistent in your syntax, spacing, and naming will help

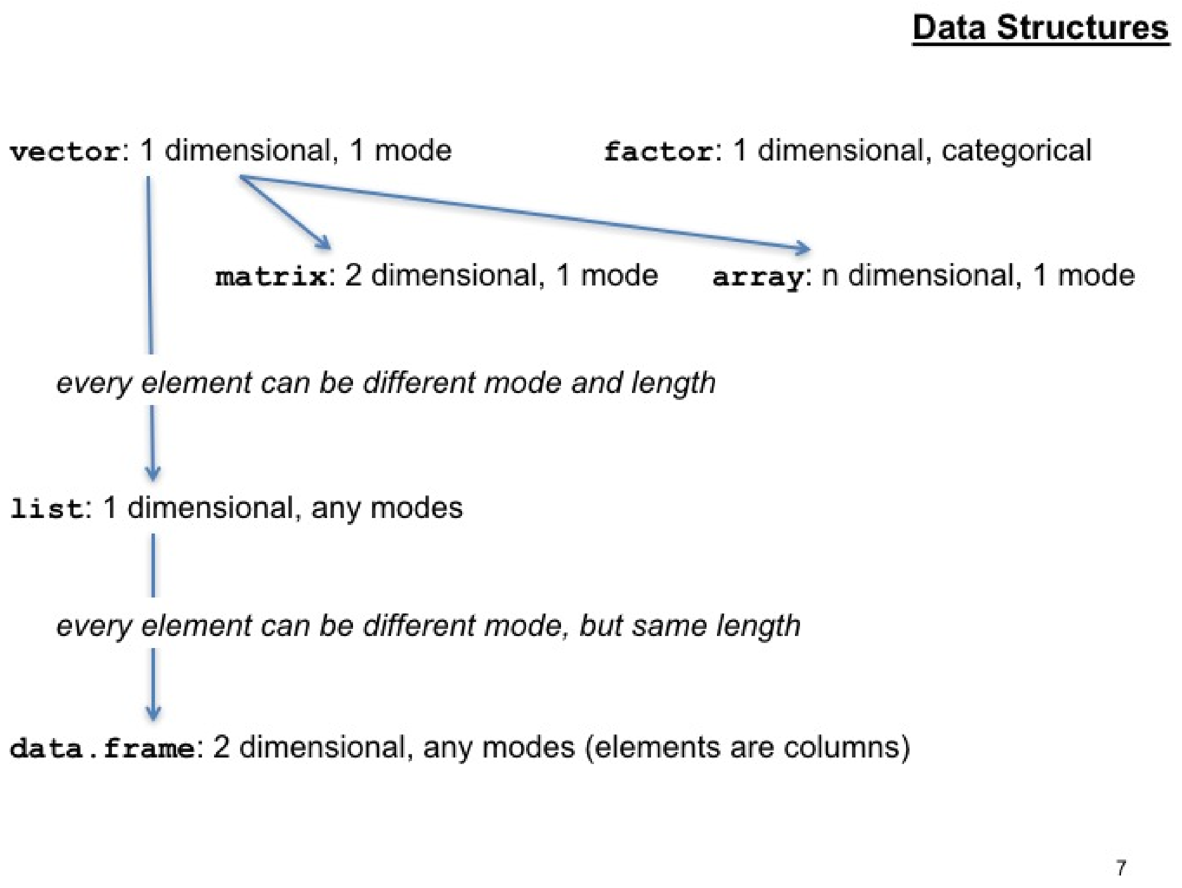
you create, edit, and understand your code later. There are many good style guides that you can follow. Feel free to mix and match from them choosing what works best for you. Here are a few:

- Google's: https://google.github.io/styleguide/Rguide.xml
- Hadley Wickham's: http://adv-r.had.co.nz/Style.html
- https://csgillespie.wordpress.com/2010/11/23/r-style-guide/
- http://jef.works/R-style-guide/

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)


Data Structures
vector: 1 dimensional, 1 mode
factor: 1 dimensional, categorical
matrix: 2 dimensional, 1 mode
array: n dimensional, 1 mode
every element can be different mode and length
list: 1 dimensional, any modes
every element can be different mode, but same length
data.frame: 2 dimensional, any modes (elements are columns)
7

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 `<-`.

```r
# The ':' operator creates a numeric vector incrementing by 1
x <- 1:10
x
```

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

```r
# The `c` function creates a vector containing the arguments inside
y <- c("a", "b", "d")
y
```

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

```r
str(x)
```

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

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

```
[1] TRUE
```

```r
class(y)
```

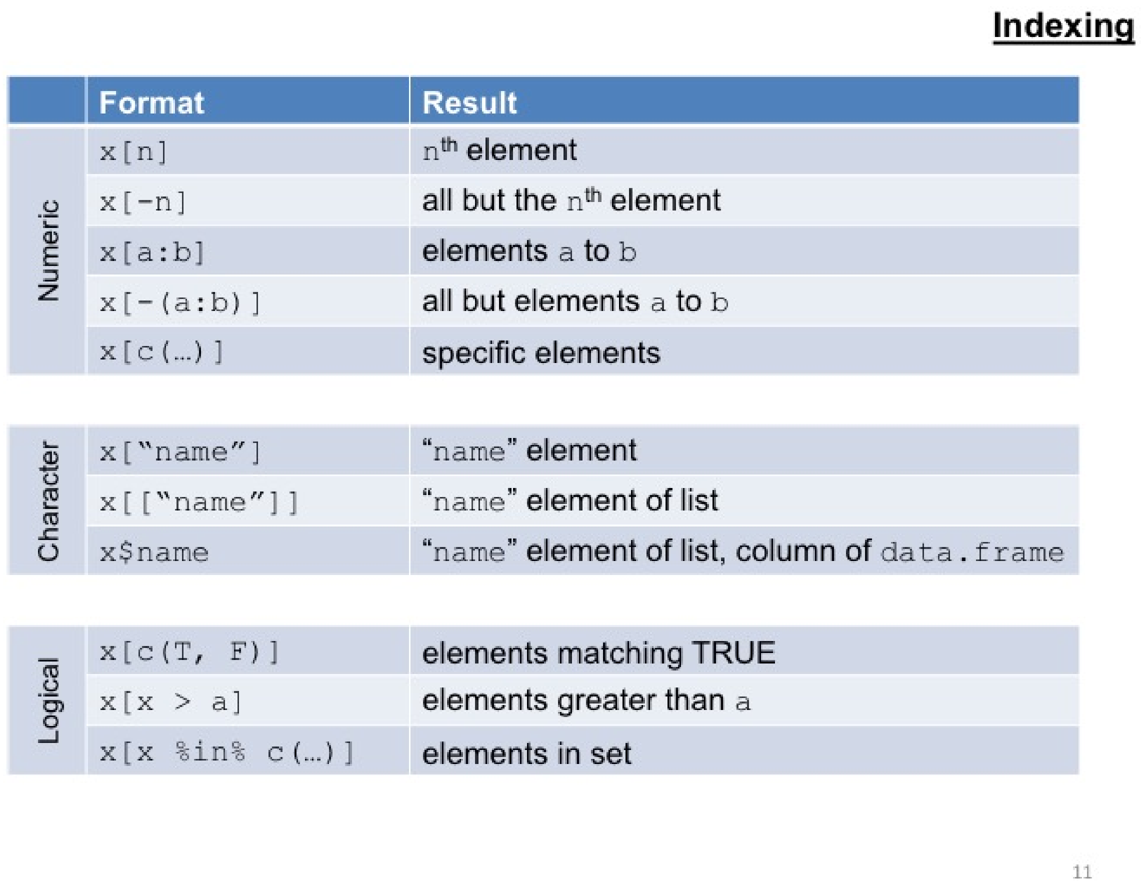
```
[1] "character"
```

```r
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



Indexing

	Format	Result
Numeric	`x[n]`	n^th element
	`x[-n]`	all but the n^th element
	`x[a:b]`	elements `a` to `b`
	`x[-(a:b)]`	all but elements `a` to `b`
	`x[c(…)]`	specific elements
Character	`x["name"]`	"`name`" element
	`x[["name"]]`	"`name`" element of list
	`x$name`	"`name`" element of list, column of `data.frame`
Logical	`x[c(T, F)]`	elements matching TRUE
	`x[x > a]`	elements greater than `a`
	`x[x %in% c(…)]`	elements in set

11

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
```

```r
# Replication of elements is allowed and will be acommodated
x[c(4, 6, 5, 6, 4)]
```

```
[1] 24 26 25 26 24
```

```r
# Any numeric vector is allowed
x[c(1:4, 5, 10:8)]
```

```
[1] 21 22 23 24 25 30 29 28
```

```r
# Negative numbers return all elements except the negative value
x[-3]
```

```
[1] 21 22 24 25 26 27 28 29 30
```

```r
# Don't fall into this trap
x[-1:5]
```

```
Error in x[-1:5]: only 0's may be mixed with negative subscripts
```

```r
# What you probably mean is this
x[-(1:5)]
```

```
[1] 26 27 28 29 30
```

Assign values to elements using indexing

```r
x[3:5] <- c(10, 20, 30)
```

Character Indexing

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

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

```
 a  b  c  d  e  f  g  h  i  j 
21 22 10 20 30 26 27 28 29 30 
```

```r
str(x)
```

```
 Named num [1:10] 21 22 10 20 30 26 27 28 29 30
 - attr(*, "names")= chr [1:10] "a" "b" "c" "d" ...
```

Then, elements can be specified by name

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

```
 d 
20 
```

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

```
 f  a 
26 21 
```

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

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

```
fourth 
    20 
```

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
```

Indexing with NAs

```
x <- c(1, 2, NA, 3)
```

Simple indexing: return values equal to 2:

```
x[x == 2]
```

```
[1]  2 NA
```

What does the logical vector that we're using to index look like?

```
x == 2
```

```
[1] FALSE  TRUE    NA FALSE
```

We know that the indexing operator will return the TRUE values and not the FALSE values, but what does it do with NA values?

```
z <- 1:3
z[c(NA, NA, NA)]
```

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

```r
z[c(TRUE, NA, TRUE)]
```

```
[1]  1 NA  3
```

```r
z[c(TRUE, NA, FALSE)]
```

```
[1]  1 NA
```

```r
z[c(FALSE, NA, FALSE)]
```

```
[1] NA
```

The bottom line is that when indexing with a logical vector, TRUE values are returned, FALSE values are not, and NA values are NA (because you don't know if they are TRUE or FALSE).