Introduction to Programming with R - June 8

Eric Archer (eric.archer@noaa.gov, 858-546-7121)

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
`#`: comment
`#'`: flowing 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/

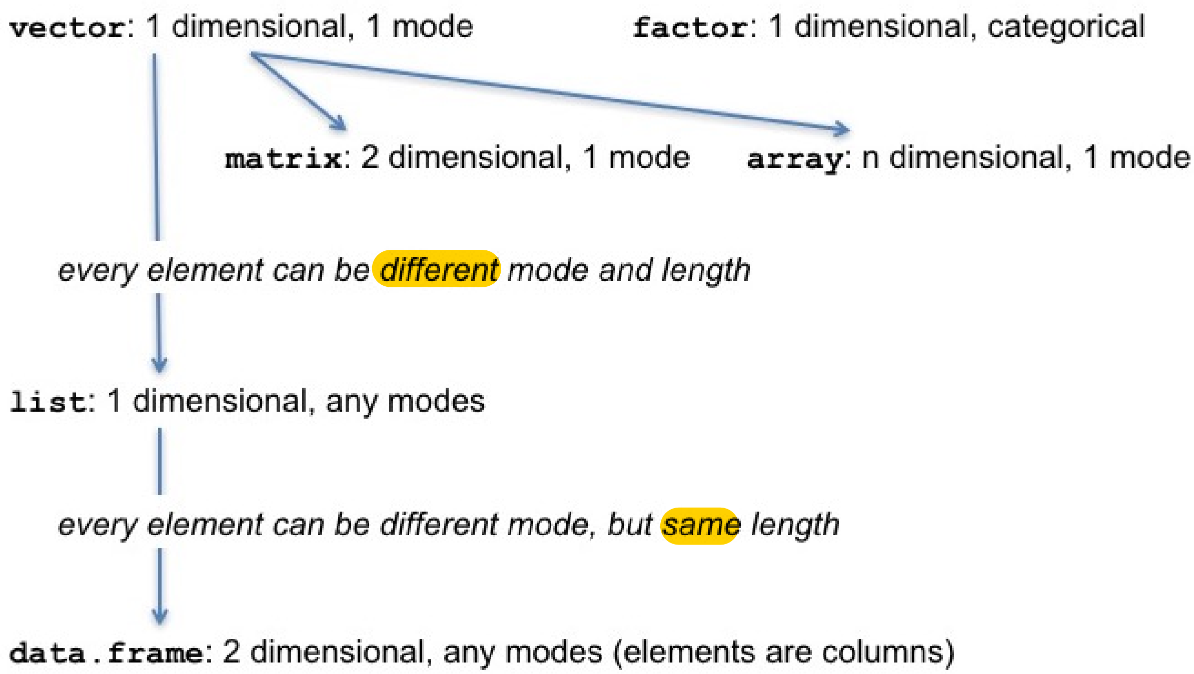
Data structures

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)

It can be useful to think of data structures being related like this:



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 Logical, will show "TRUE" or "FALSE"

Vectors

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

```r
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
```

could think of this as combine/column/category

```
[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:

	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

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
```

using the "letters" allows you to index from the alphabet

```
 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
```

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
```

i.e. 10 is divisible by 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

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

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

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

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

Character vectors

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

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

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

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

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

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

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

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

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

```r
# 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"
```

```r
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

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

```
[1] TRUE
```

```r
all(x)
```

```
[1] FALSE
```

```r
# Negate the vector
!x
```

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

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

Read back every other value

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

```r
# Just the TRUE values
x[x]
```

```
[1] TRUE TRUE TRUE
```

```r
# Logical vectorization
x & T
```

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

```r
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

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

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

=1,2,3,4, defaults to alphabetical/numeric order

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

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

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

```
[1] FALSE
```

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

```
[1] FALSE
```

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

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

```r
# 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:

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

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

```r
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`.

```r
# 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] d b d b c a c a a b
Levels: a b c d
```

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

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

= levels

```r
# 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] d b d b c a c a a b
Levels: c b d a
```

factor values still in same order, but order of levels is now different

```r
# 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] d b d b c a c a a b
Levels: c e b d a
```

```r
# Omitting a level causes all values with that level to be NA
xf.lvl <- factor(x, levels = c("b", "d", "a"))
xf.lvl
```

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

```r
# Labels will match order of levels
xf.lbl <- factor(x, labels = c("Z", "Y", "X", "W"))
xf.lbl
```

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

```r
# 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
```

Matrices

Matrices are always two-dimensional objects having a certain number of rows and columns. They contain only one kind (atomic mode) of data (e.g., numeric, character, logical). They are created by supplying a vector of values to the `matrix()` function and specifying how many rows and/or how many columns to dimension it by

```
# Create a matrix
x <- 1:24
mat <- matrix(x, nrow = 4)
mat
```

Don't have to specify number of columns, will determine 6 automatically

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

```
# How many elements are in the matrix?
length(mat)
```

```
[1] 24
```

```
#How many rows and columns?
nrow(mat)
```

```
[1] 4
```

```
ncol(mat)
```

```
[1] 6
```

Cells are selected by `[row, column]`

```
mat[2, 3]
```

```
[1] 10
```

Selecting a single row or single column returns a vector

```
mat[3, ]
```

read back row three across all columns

```
[1]  3  7 11 15 19 23
```

```
mat[, 4]
```

read back column four down all rows

```
[1] 13 14 15 16
```

Use `drop = F` to select a single row or column and return a matrix

```
mat[4, , drop = F]
```

drop maintains matrix structure, rather than returning a vector

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    4    8   12   16   20   24
```

```
mat[, 2, drop = F]
```

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

Select several rows or columns

this automatically retains the matrix structure, but resets row/column numbering

```
mat[c(1, 3, 4), ]
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    3    7   11   15   19   23
[3,]    4    8   12   16   20   24
```

```
mat[, 2:5]
```

```
     [,1] [,2] [,3] [,4]
[1,]    5    9   13   17
[2,]    6   10   14   18
[3,]    7   11   15   19
[4,]    8   12   16   20
```

Select rows, exclude columns

```
mat[1:3, -(2:4)]
```

```
     [,1] [,2] [,3]
[1,]    1   17   21
[2,]    2   18   22
[3,]    3   19   23
```

Change a value in the matrix

```
mat[2, 5] <- NA
```

Change an entire column

```
mat[, 3] <- 100:103
```

Adding a column or row

```
mat.plus.col <- cbind(mat, 100:103)
mat.plus.row <- rbind(300:307, mat)
```

```
Warning in rbind(300:307, mat): number of columns of result is not a
multiple of vector length (arg 1)
```

Assign row and column names

```
rownames(mat) <- c("first", "second", "third", "fourth")
colnames(mat) <- letters[1:ncol(mat)]
```

for when you know how many columns there are

labels columns with letters from 1st to last column (ncol)

Choose rows and columns by name

```
mat["first", c("e", "c", "d")]
```

```
  e   c   d 
 17 100  13 
```

Choose columns by logical vectors

```
mat[, c(T, T, F, F, T, F)]
```

```
       a b  e
first  1 5 17
second 2 6 NA
third  3 7 19
fourth 4 8 20
```

Transpose a matrix

```r
t(mat)
```

```
  first second third fourth
a     1      2     3      4
b     5      6     7      8
c   100    101   102    103
d    13     14    15     16
e    17     NA    19     20
f    21     22    23     24
```

Add, subtract, multiply, or divide a matrix by a scalar

```r
mat * 5
```

```
        a  b   c  d   e   f
first   5 25 500 65  85 105
second 10 30 505 70  NA 110
third  15 35 510 75  95 115
fourth 20 40 515 80 100 120
```

```r
mat / 3
```

```
               a        b        c        d        e        f
first  0.3333333 1.666667 33.33333 4.333333 5.666667 7.000000
second 0.6666667 2.000000 33.66667 4.666667       NA 7.333333
third  1.0000000 2.333333 34.00000 5.000000 6.333333 7.666667
fourth 1.3333333 2.666667 34.33333 5.333333 6.666667 8.000000
```

```r
mat ^ 2
```

```
        a  b     c   d   e   f
first   1 25 10000 169 289 441
second  4 36 10201 196  NA 484
third   9 49 10404 225 361 529
fourth 16 64 10609 256 400 576
```

Add a column and a matrix

```r
mat + 1000:1003
```

```
          a    b    c    d    e    f
first  1001 1005 1100 1013 1017 1021
second 1003 1007 1102 1015   NA 1023
third  1005 1009 1104 1017 1021 1025
fourth 1007 1011 1106 1019 1023 1027
```

Row and column sums or means

```r
rowSums(mat)
```

```
 first second  third fourth 
   157     NA    169    175 
```

```r
colMeans(mat)
```

```
    a     b     c     d     e     f 
  2.5   6.5 101.5  14.5    NA  22.5 
```

Arrays

Arrays are multi-dimensional objects that also contain only a single atomic mode of data. They are indexed the same way as matrices, but created by specifying the number of dimensions.

```
# 1 dimensional array (= vector)
arr.vec <- array(x)
arr.vec
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
[24] 24
```

```
# 2 dimensional array (= matrix)
arr.mat <- array(x, dim = c(3, 8))
arr.mat
```

3 rows, 8 columns

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    4    7   10   13   16   19   22
[2,]    2    5    8   11   14   17   20   23
[3,]    3    6    9   12   15   18   21   24
```

```
# 3 dimensional array
arr.3d <- array(x, dim = c(3, 4, 2))
arr.3d
```

split into 2 "sub arrays", each with 3 rows and 4 columns

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Lists

Lists are one-dimensional objects where each element can be any kind of object.

```
x <- list(1, letters[1:5], matrix(100:119, 5))
x
```

this makes three lists, that are sublists of x.
In other words, x is a list of lists

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

[[2]]
[1] "a" "b" "c" "d" "e"

[[3]]
     [,1] [,2] [,3] [,4]
[1,]  100  105  110  115
```

```
[2,]  101  106  111  116
[3,]  102  107  112  117
[4,]  103  108  113  118
[5,]  104  109  114  119
```

```
str(x)
```

```
List of 3
 $ : num 1
 $ : chr [1:5] "a" "b" "c" "d" ...
 $ : int [1:5, 1:4] 100 101 102 103 104 105 106 107 108 109 ...
```

```
class(x)
```

```
[1] "list"
```

```
mode(x)
```

```
[1] "list"
```

A useful piece of information is that lists are special vectors:

```
is.list(x)
```

```
[1] TRUE
```

```
is.vector(x)
```

```
[1] TRUE
```

If you use a single bracket (`[`) to index a list, you will get a list back:

```
y <- x[2]
str(y)
```

this takes the 2nd list of x, and puts into "y"

```
List of 1
 $ : chr [1:5] "a" "b" "c" "d" ...
```

```
length(y)
```

y contains 1 list, so the length is 1

```
[1] 1
```

To get the actual object back, you have to use double brackets (`[[`):

```
z <- x[[2]]
str(z)
```

now, z is the string of letters, rather than a list

```
 chr [1:5] "a" "b" "c" "d" "e"
```

```
length(z)
```

```
[1] 5
```

List elements can have names and they can be used for indexing like vectors, but single brackets still return a list and double brackets return the object:

```
x2 <- list(first = 1, lets = letters[1:5], third = matrix(30:53, 4))
x2["first"]
```

```
$first
[1] 1
```

```
x2[["first"]]
```

```
[1] 1
```

The dollar sign ($) is a special operator for lists with names that returns the same thing as double brackets:

```
x2$first
```

```
[1] 1
```

List names can be changed with **names**:

```
names(x2) <- c("a.number", "some.letters", "a.matrix")
x2
```

```
$a.number
[1] 1

$some.letters
[1] "a" "b" "c" "d" "e"

$a.matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   30   34   38   42   46   50
[2,]   31   35   39   43   47   51
[3,]   32   36   40   44   48   52
[4,]   33   37   41   45   49   53
```

A list can contain a list, and if you know the names, you can chain the **$**:

```
x2$new.element <- list(numbers = 1:5, matrix = matrix(11:25, 3))
x2
```

```
$a.number
[1] 1

$some.letters
[1] "a" "b" "c" "d" "e"

$a.matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   30   34   38   42   46   50
[2,]   31   35   39   43   47   51
[3,]   32   36   40   44   48   52
[4,]   33   37   41   45   49   53

$new.element
$new.element$numbers
[1] 1 2 3 4 5

$new.element$matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   14   17   20   23
[2,]   12   15   18   21   24
[3,]   13   16   19   22   25
```

```
x2$new.element$matrix
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   14   17   20   23
[2,]   12   15   18   21   24
[3,]   13   16   19   22   25
```

To remove an element from a list, you assign `NULL` to that element:

```
x2$some.letters <- NULL
x2
```

```
$a.number
[1] 1

$a.matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   30   34   38   42   46   50
[2,]   31   35   39   43   47   51
[3,]   32   36   40   44   48   52
[4,]   33   37   41   45   49   53

$new.element
$new.element$numbers
[1] 1 2 3 4 5

$new.element$matrix
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   14   17   20   23
[2,]   12   15   18   21   24
[3,]   13   16   19   22   25
```

Lists can be grown using the `c` function:

```
x <- list(a = 1, b = 2:6, c = letters)
z <- c(x, g = T)
z
```

x was the first list of lists, and g = t is the list you're adding to that list, then renaming the orig list

```
$a
[1] 1

$b
[1] 2 3 4 5 6

$c
 [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"

$g
[1] TRUE
```

We use `dimnames` to add names to arrays. They have to be specified as lists:

```
arr <- array(1:24, dim = c(3, 4, 2))
dimnames(arr) <- list(letters[1:3], LETTERS[1:4], c("one", "two"))
arr
```

rows columns sub-arrays

```
, , one

  A B C  D
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12

, , two
```

```
   A  B  C  D
a 13 16 19 22
b 14 17 20 23
c 15 18 21 24
```

Data Frames

Data frames are two-dimensional objects that are normally used to represent data where the rows are observations and the columns are variables. This is the tidy data concept

```
ids <- c(1213, 2435, 5367, 6745, 3592)
loc <- c("north", "north", "north", "west", "south")
len <- c(9.9, 4.5, 7.7, 3.4, 2.0)
wght <- c(270, 130, 235, 90, 88)

df <- data.frame(id = ids, location = loc, len = len, wt = wght)

str(df)
```

```
'data.frame':   5 obs. of  4 variables:
 $ id      : num  1213 2435 5367 6745 3592
 $ location: Factor w/ 3 levels "north","south",..: 1 1 1 3 2
 $ len     : num  9.9 4.5 7.7 3.4 2
 $ wt      : num  270 130 235 90 88
```

```
nrow(df)
```

```
[1] 5
```

```
ncol(df)
```

```
[1] 4
```

Data frames are actually special lists where every column is an element that is the same length:

```
is.data.frame(df)
```

```
[1] TRUE
```

```
is.list(df)
```

```
[1] TRUE
```

```
is.vector(df)
```

```
[1] FALSE
```

```
length(df)
```
returns the number of columns (i.e. variables)

```
[1] 4
```

Data frames are indexed the same way as matrices:

```
df[1, ]
```
This time, it will return the column/row names of what you're indexing too

```
    id location len  wt
1 1213    north 9.9 270
```

```
df[, "len"]
```

```
[1] 9.9 4.5 7.7 3.4 2.0
```

```
df[, c("id", "wt")]
```

```
    id  wt
1 1213 270
2 2435 130
3 5367 235
4 6745  90
5 3592  88
```

Columns can also be returned as a vector using the `$`:

```
df$wt
```

```
[1] 270 130 235  90  88
```

Data frames are often indexed by a column within the data frame itself. For instance, we want to select only the rows where sepal length is less than 4.8:

```
# `iris` is a sample data set included in base R
summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
 Median :5.800   Median :3.000   Median :4.350   Median :1.300  
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
       Species  
 setosa    :50  
 versicolor:50  
 virginica :50  
```

```
# extract rows where
iris[iris$Sepal.Length < 4.8, ]
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
7           4.6         3.4          1.4         0.3  setosa
9           4.4         2.9          1.4         0.2  setosa
14          4.3         3.0          1.1         0.1  setosa
23          4.6         3.6          1.0         0.2  setosa
30          4.7         3.2          1.6         0.2  setosa
39          4.4         3.0          1.3         0.2  setosa
42          4.5         2.3          1.3         0.3  setosa
43          4.4         3.2          1.3         0.2  setosa
48          4.6         3.2          1.4         0.2  setosa
```

Notice that when we do this, we are placing the condition in the row slot of the indexing brackets. The point of this is that we are creating a logical condition as long as there are rows and using this logical vector to index.

Here's a more complex example:

```
iris[iris$Sepal.Length > 5 & iris$Petal.Length < 1.5 & iris$Species == "setosa", ]
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
15          5.8         4.0          1.2         0.2  setosa
17          5.4         3.9          1.3         0.4  setosa
18          5.1         3.5          1.4         0.3  setosa
29          5.2         3.4          1.4         0.2  setosa
34          5.5         4.2          1.4         0.2  setosa
37          5.5         3.5          1.3         0.2  setosa
```

We can also choose which columns to return at the same time:

```
iris[iris$Sepal.Width < 2.5, c("Species", "Sepal.Length")]
```

```
       Species Sepal.Length
42      setosa          4.5
54  versicolor          5.5
58  versicolor          4.9
61  versicolor          5.0
63  versicolor          6.0
69  versicolor          6.2
81  versicolor          5.5
82  versicolor          5.5
88  versicolor          6.3
94  versicolor          5.0
120  virginica          6.0
```

Note that the vector that you use to index a data.frame does not have to be in the data.frame itself. It just has to be as long as there are rows in the data.frame:

```
petal.area <- iris$Petal.Length * iris$Petal.Width
summary(petal.area)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  0.110   0.420   5.615   5.794   9.690  15.870 
```

```
iris[petal.area > 14, ]
```

petal.area was something you created based on info in "iris", but you can use it in indexing

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
101          6.3         3.3          6.0         2.5 virginica
110          7.2         3.6          6.1         2.5 virginica
118          7.7         3.8          6.7         2.2 virginica
119          7.7         2.6          6.9         2.3 virginica
136          7.7         3.0          6.1         2.3 virginica
145          6.7         3.3          5.7         2.5 virginica
```

If the indexing vector is shorter, then vectorization happens:

```
# first 10 rows
head(iris, 10)
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
```

```
4           4.6         3.1          1.5         0.2  setosa
5           5.0         3.6          1.4         0.2  setosa
6           5.4         3.9          1.7         0.4  setosa
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
```

```r
# extract every third row
head(iris[c(T, F, F), ], 4)
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
7           4.6         3.4          1.4         0.3  setosa
10          4.9         3.1          1.5         0.1  setosa
```

The `subset` function is a convenient way to index a data.frame without using the `$` notation:

```r
subset(iris, Petal.Width > 2.3, c("Species", "Petal.Width", "Petal.Length"))
```

```
      Species Petal.Width Petal.Length
101 virginica         2.5          6.0
110 virginica         2.5          6.1
115 virginica         2.4          5.1
137 virginica         2.4          5.6
141 virginica         2.4          5.6
145 virginica         2.5          5.7
```

Coercion

Many objects can be coerced from one class to another using `as.<class>` functions. If you have a `numeric` vector, it can be coerced to `character` or `logical`:

```r
as.character(1:5)
```

```
[1] "1" "2" "3" "4" "5"
```

```r
# when going from numeric to logical, 0 = FALSE, all other numbers are TRUE
as.logical(c(-1, -0.5, 0, 1, 3.5, 6))
```

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

Going from `character` to `numeric` or `logical`:

```r
as.numeric(c("-5", "0.3", "3.14x", "hello", "a4"))
```

```
Warning: NAs introduced by coercion
```

```
[1] -5.0  0.3   NA   NA   NA
```

```r
as.logical(c("hello", "T", "false", "True", "n", "1"))
```

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

Going from `logical` to `character` or `numeric`:

```r
as.character(c(T, F, TRUE, FALSE))
```

```
[1] "TRUE"  "FALSE" "TRUE"  "FALSE"
```

```
as.numeric(c(T, F, TRUE, FALSE))
```

```
[1] 1 0 1 0
```

When coercing a `logical` to `numeric` `T = 1` and `F = 0`. This has some useful properties. To count the number of elements that meet a condition, we can use this feature with the `sum` function:

```
x <- sample(1:5, 100, replace = T)
x
```

```
  [1] 2 2 2 2 1 5 3 1 2 2 4 4 5 4 2 4 3 3 3 5 4 3 1 4 1 1 5 5 2 2 2 4 1 2 5
 [36] 4 2 1 3 4 4 2 4 1 5 2 5 2 2 2 2 1 3 1 1 5 1 5 5 4 2 2 1 4 3 4 4 2 5 4
 [71] 3 1 3 3 2 1 2 4 1 5 4 4 3 4 3 1 1 4 3 3 1 5 1 1 4 5 3 2 1 3
```

```
sum(x == 1)
```

```
[1] 22
```

Likewise, to calculate the proportion of things that meet a condition, we use the same trick with `mean`:

```
mean(x <= 2)
```

```
[1] 0.46
```

Missing data (NAs)

Missing data is denoted in R with `NA` and has to be explicitly tested for and handled specially. To test if values are equal to NA, you can't use `==`, you have to use `is.na()`

```
x <- c(1, NA, 3, 6, NA)
x == NA
```

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

```
is.na(x)
```

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

To remove NAs from a vector, use `na.omit()`:

```
x2 <- na.omit(x)
x2
```

```
[1] 1 3 6
attr(,"na.action")
[1] 2 5
attr(,"class")
[1] "omit"
```

```
str(x2)
```

```
 atomic [1:3] 1 3 6
 - attr(*, "na.action")=Class 'omit'  int [1:2] 2 5
```

To identify rows in a data frame without NAs, use `complete.cases`:

```
iris.na <- iris
iris.na[1, "Sepal.Length"] <- NA
iris.na[8, "Petal.Length"] <- NA
```

```r
iris.na[12, c("Sepal.Length", "Petal.Length")] <- NA
summary(iris.na)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
 Median :5.800   Median :3.000   Median :4.400   Median :1.300  
 Mean   :5.855   Mean   :3.057   Mean   :3.788   Mean   :1.199  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
 NA's   :2                       NA's   :2                       
       Species  
 setosa    :50  
 versicolor:50  
 virginica :50  
```

```r
i <- complete.cases(iris.na)
i
```

```
  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
 [12] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [56]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [67]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [78]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [89]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[100]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[111]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[122]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[133]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[144]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
summary(iris.na[i, ])
```

```
  Sepal.Length    Sepal.Width    Petal.Length    Petal.Width  
 Min.   :4.300   Min.   :2.00   Min.   :1.000   Min.   :0.10  
 1st Qu.:5.100   1st Qu.:2.80   1st Qu.:1.600   1st Qu.:0.30  
 Median :5.800   Median :3.00   Median :4.400   Median :1.30  
 Mean   :5.861   Mean   :3.05   Mean   :3.804   Mean   :1.22  
 3rd Qu.:6.400   3rd Qu.:3.30   3rd Qu.:5.100   3rd Qu.:1.80  
 Max.   :7.900   Max.   :4.40   Max.   :6.900   Max.   :2.50  
       Species  
 setosa    :47  
 versicolor:50  
 virginica :50  
```

Directories and files

Working directory

The working directory in R is the default location where files are written to and read from. To see where that currently is, use `getwd()`

```
getwd()
```

```
[1] "/Users/ericarcher/Desktop/2018 City R Class/Day 1"
```

The working directory can be changed programmatically with `setwd()`, and a character vector of the directory contains viewed with `dir()`:

```
# The contents of this directory
dir()
```

```
 [1] "DataStructures.jpg"             "free text.txt"
 [3] "Indexing.jpg"                   "Intro to R - Week 1 June 8.Rmd"
 [5] "Intro_to_R_-_Week_1_June_8.pdf" "Intro_to_R_-_Week_1_June_8.Rmd"
 [7] "test ws.rdata"                  "test.csv"
 [9] "x.r"                            "xy.rdata"
```

```
# Move up a directory
setwd("..")
# Show the contents of this directory
dir()
```

```
[1] "2018 City R Class.Rproj"         "Archer 2018 Training Quote.docx"
[3] "Day 1"                           "Homework"
[5] "Intro to R Syllabus.Rmd"         "Prep"
```

The `pattern` argument of `dir()` allows you to filter the files that are returned:

```
dir(pattern = "jpg")
```

```
[1] "DataStructures.jpg" "Indexing.jpg"
```

Reading and writing data

R workspaces (.Rdata): `save`, `load`

The entire workspace can be saved to disk with `save.image()`. R workspace/object files are binary files that cannot be read by anything but R. They usually end in ".rdata", or ".rda".

```
rm(list = ls())
x <- 1
y <- 2
z <- 3
save.image(file = "test ws.rdata")
```

The file can be read back into the workspace with `load()`:

```
rm(list = ls())
ls()
```

```
character(0)
```

```r
load("test ws.rdata")
ls()
```

```
[1] "x" "y" "z"
```

Individual objects can be saved with `save()`:

```r
save(x, y, file = "xy.rdata")
rm(list = ls())
load("xy.rdata")
ls()
```

```
[1] "x" "y"
```

Text tables (.csv): `write.table`, `read.table`

Data in tabular format, such as matrices or data frames are saved to and read from disk with `write.table` and `read.table` and their wrappers, most commonly `write.csv` and `read.csv`:

```r
x <- data.frame(nums = 51:60, lets = letters[1:10])
write.csv(x, file = "test.csv")
rm(list = ls())
df <- read.csv("test.csv")
df
```

```
    X nums lets
1   1   51    a
2   2   52    b
3   3   53    c
4   4   54    d
5   5   55    e
6   6   56    f
7   7   57    g
8   8   58    h
9   9   59    i
10 10   60    j
```

You'll notice that there is a new column, "X" that has the numbers 1-10 in it. This is because by default, `write.csv` writes a file with the rownames in the first column. To change this behavior, set the argument `row.nams = FALSE` in `write.csv`.

```r
x <- data.frame(nums = 51:60, lets = letters[1:10])
write.csv(x, file = "test.csv", row.names = FALSE)
rm(list = ls())
df <- read.csv("test.csv")
df
```

```
   nums lets
1    51    a
2    52    b
3    53    c
4    54    d
5    55    e
6    56    f
7    57    g
8    58    h
9    59    i
```

```
10    60    j
```

```r
str(df)
```

```
'data.frame':   10 obs. of  2 variables:
 $ nums: int  51 52 53 54 55 56 57 58 59 60
 $ lets: Factor w/ 10 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10
```

Also, notice that the `lets` column is read in as a factor. This is the default behavior of `read.csv` and can be changed with the `stringsAsFactors` argument:

```r
df <- read.csv("test.csv", stringsAsFactors = FALSE)
str(df)
```

```
'data.frame':   10 obs. of  2 variables:
 $ nums: int  51 52 53 54 55 56 57 58 59 60
 $ lets: chr  "a" "b" "c" "d" ...
```

Free text (.txt): `write`, `scan`

Free text can be written and read with `write` and `scan`. With `write`, one line is written per call and the `append` argument is used to add to an existing file:

```r
fname <- "free text.txt"
write("Hello, I am the first line", file = fname)
write("...and I am the second line in the file", file = fname, append = TRUE)
write("I'll be the third line to end it all", file = fname, append = TRUE)
```

`scan` will read a text file into a vector of a type specified by the `what` argument. See the Details section in `?scan` for more info. To read the text file above:

```r
rm(list = ls())
x <- scan("free text.txt", what = "character")
x
```

```
 [1] "Hello," "I"      "am"     "the"    "first"  "line"   "...and"
 [8] "I"      "am"     "the"    "second" "line"   "in"     "the"
[15] "file"   "I'll"   "be"     "the"    "third"  "line"   "to"
[22] "end"    "it"     "all"
```

```r
# here the delimiter is the end of line character ("\n") so each line is a single element in the return
z <- scan("free text.txt", what = "character", sep = "\n")
z
```

```
[1] "Hello, I am the first line"
[2] "...and I am the second line in the file"
[3] "I'll be the third line to end it all"
```

R scripts (.r): `dump`, `source`

R objects can be written to files as a form of the code used to create them using `dump`. These files usually end in ".R"

```r
x <- matrix(1:24, nrow = 4)
dump("x", file = "x.r")
```

These files can be read back in using `source`.

```r
rm(list = ls())
source("x.r")
ls()
```

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

`source` is used to execute R commands stored in text files. It is the command you will use to execute saved scripts.

Homework

Answer all questions in a script (.R) file. Use comments (`#` or `#'`) to explain steps in code.

1. Compute the following values:
    - 27 times 38 minus 17
    - natural logarithm of (56 divided by 4)
    - square root of (4 times 13)
    - 6 squared divided by 2
2. Create two vectors:
    - 160, decreasing sequential integers by 10, 20, 15, 10, 5
    - 56, 57, 58, 59, sequential integers by 2, 85, 86
      (If you can't create these two vectors, then create one vector from 20 to 35 and another from 205 to 190)
3. Multiply the two vectors above and assign to a new vector.
4. What are the 3rd, 10th, and 13th elements of the vector from 3?
5. How many elements of the vector in 3 are greater than 6000?
6. What is the mean, median, and sum of the vector in 3?
7. Change the 4th and 8th elements of the vector in 3 to 4 and 8.
8. Using the `USArrests` data.frame, extract a vector of the percent of the population living in urban areas (`UrbanPop`). Create names for this vector from the states in the row names of the data.frame.
9. What is the percentage of the population in urban areas for Kansas, California, and Kentucky?
10. What is the average urban population percentage in New England (Maine, Vermont, New Hampshire, Massachusetts, Rhode Island, and Connecticut)?
11. What is the average murder/assault ratio in the entire `USArrests` data.frame?
12. Create a copy of the `USArrests` data.frame and add a column for the murder/assault ratio.
13. Write a .csv file of this modified data.frame.
14. Read the .csv file from 13 to a new data.frame.
15. Extract rows from the data.frame read in 14 that have a murder/assault ratio less than the mean.