R Basics 1

Instructor: Yuta Toyama

Last updated: 2020-03-30

Section 1

Data

Acknowledgement

This note is largely based on Applied Statistics with R.
https://daviddalpiaz.github.io/appliedstats/

Data Types

`R` has a number of basic data *types*.

- Numeric
  - Also known as Double. The default type when dealing with numbers.
  - Examples: `1`, `1.0`, `42.5`
- Logical
  - Two possible values: `TRUE` and `FALSE`
  - You can also use `T` and `F`, but this is *not* recommended.
  - `NA` is also considered logical.
- Character
  - Examples: `"a"`, `"Statistics"`, `"1 plus 2."`

Data Structures

- R also has a number of basic data *structures*.
- A data structure is either
  - homogeneous (all elements are of the same data type)
  - heterogeneous (elements can be of more than one data type).

Dimension	**Homogeneous**	**Heterogeneous**
1	Vector	List
2	Matrix	Data Frame
3+	Array	

Section 2

Vector

Vectors

Basics of vectors

- Many operations in R make heavy use of **vectors**.
  - Vectors in R are indexed starting at 1.
- The most common way to create a vector in R is using the c() function, which is short for "combine.""

```
c(1, 3, 5, 7, 8, 9)
```

```
## [1] 1 3 5 7 8 9
```

Assignment

- If we would like to store this vector in a **variable** we can do so with the **assignment** operator =.
  - The variable x now holds the vector we just created, and we can access the vector by typing x.

```
x = c(1, 3, 5, 7, 8, 9)
x
```

```
## [1] 1 3 5 7 8 9
```

```
# The following does the same thing.
x <- c(1, 3, 5, 7, 8, 9)
x
```

```
## [1] 1 3 5 7 8 9
```

- The operator = and <- work as an assignment operator.
  - You can use both. This does not matter usually.
  - If you are interested in the weird cases where the difference matters, check out The R Inferno.
- In `R` code the line starting with # is `comment`, which is ignored when you run the fode.

A sequence of numbers.

- The quickest and easiest way to do this is with the `:` operator, which creates a sequence of integers between two specified integers.

```
(y = 1:100)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [91]  91  92  93  94  95  96  97  98  99 100
```

- By putting parentheses around the assignment,
  - `R` both stores the vector in a variable called `y` and
  - automatically outputs `y` to the console.

Useful functions for creating vectors

- Use the `seq()` function for a more general sequence.

```
seq(from = 1.5, to = 4.2, by = 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.
## [20] 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

- Here, the input labels `from`, `to`, and `by` are optional.

```
seq(1.5, 4.2, 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.
## [20] 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

- We have now seen four different ways to create vectors:
    1. `c()`
    2. `:`
    3. `seq()`
    4. `rep()`
- They are often used together.

Length

- The length of a vector can be obtained with the `length()` function.

```
length(x)
```

```
## [1] 6
```

```
length(y)
```

```
## [1] 100
```

Subsetting

- Use square brackets, `[]`, to obtain a subset of a vector.
- We see that `x[1]` returns the first element.

```
x
```

```
## [1] 1 3 5 7 8 9
```

```
x[1]
```

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

```
x[3]
```

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

- We can also exclude certain indexes, in this case the second element.

```
x[-2]
```

```
## [1] 1 5 7 8 9
```

- We can subset based on a vector of indices.

```
x[1:3]
```

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

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

```
## [1] 1 5 7
```

- We could instead use a vector of logical values.

```r
z = c(TRUE, TRUE, FALSE, TRUE, TRUE, FALSE)
z
```

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

```r
x[z]
```

```
## [1] 1 3 7 8
```

Vectorization

- One of the biggest strengths of R is its use of vectorized operations.
  - Frequently the lack of understanding of this concept leads of a belief that R is *slow*.
- When a function like log() is called on a vector x, a vector is returned which has applied the function to each element of the vector x.

```
x = 1:10
x + 1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
2 * x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```r
2 ^ x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```r
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
##  [9] 3.000000 3.162278
```

```r
log(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.79
##  [8] 2.0794415 2.1972246 2.3025851
```

Logical Operators

Operator	Summary	Example	Result
`x < y`	`x` less than `y`	`3 < 42`	TRUE
`x > y`	`x` greater than `y`	`3 > 42`	FALSE
`x <= y`	`x` less than or equal to `y`	`3 <= 42`	TRUE
`x >= y`	`x` greater than or equal to `y`	`3 >= 42`	FALSE
`x == y`	`x`equal to `y`	`3 == 42`	FALSE
`x != y`	`x` not equal to `y`	`3 != 42`	TRUE
`!x`	not `x`	`!(3 > 42)`	TRUE
`x | y`	`x` or `y`	`(3 > 42) | TRUE`	TRUE
`x & y`	`x` and `y`	`(3 < 4) & (42 > 13)`	TRUE

- Logical operators are vectorized.

```
x = c(1, 3, 5, 7, 8, 9)
x > 3
```

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

```
x < 3
```

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

```
x == 3
```

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

```
x != 3
```

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

```r
x == 3 & x != 3
```

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

```r
x == 3 | x != 3
```

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

- This is extremely useful for subsetting.

```
x[x > 3]
```

```
## [1] 5 7 8 9
```

```
x[x != 3]
```

```
## [1] 1 5 7 8 9
```

Short exercise

1. Create the vector $z = (1, 2, 1, 2, 1, 2)$, which has the same length as x.
2. Pick up the elements of x which corresponds to 1 in the vector z.

Section 3

Matrix

Matrix Operation: Basics

- R can also be used for **matrix** calculations.
  - Matrices have rows and columns containing a single data type.
- Matrices can be created using the `matrix` function.

```
x = 1:9
X = matrix(x, nrow = 3, ncol = 3)
X
```

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

- We are using two different variables:
  - lower case `x`, which stores a vector and
  - capital `X`, which stores a matrix.

- By default the `matrix` function reorders a vector into columns, but we can also tell `R` to use rows instead.

```
Y = matrix(x, nrow = 3, ncol = 3, byrow = TRUE)
Y
```

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

- a matrix of a specified dimension where every element is the same, in this case 0.

```
Z = matrix(0, 2, 4)
Z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
```

- Matrices can be subsetted using square brackets, `[]`.
- However, since matrices are two-dimensional, we need to specify both a row and a column when subsetting.
- Here we get the element in the first row and the second column.

```
X
```

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

```
X[1, 2]
```

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

- We can also subset an entire row or column.

```
X[1, ]
```

```
## [1] 1 4 7
```

```
X[, 2]
```

```
## [1] 4 5 6
```

- Matrices can also be created by combining vectors as columns, using `cbind`, or combining vectors as rows, using `rbind`.

```
x = 1:9
rev(x)
```

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

```
rep(1, 9)
```

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

```
rbind(x, rev(x), rep(1, 9))
```

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

- When using `rbind` and `cbind` you can specify "argument" names that will be used as column names.

```
cbind(col_1 = x, col_2 = rev(x), col_3 = rep(1, 9))
```

```
##       col_1 col_2 col_3
##  [1,]     1     9     1
##  [2,]     2     8     1
##  [3,]     3     7     1
##  [4,]     4     6     1
##  [5,]     5     5     1
##  [6,]     6     4     1
##  [7,]     7     3     1
##  [8,]     8     2     1
##  [9,]     9     1     1
```

Matrix calculations

- Perform matrix calculations.

```
x = 1:9
y = 9:1
X = matrix(x, 3, 3)
Y = matrix(y, 3, 3)
X
```

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

```
Y
```

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

```r
X + Y
```

```
##      [,1] [,2] [,3]
## [1,]   10   10   10
## [2,]   10   10   10
## [3,]   10   10   10
```

```r
X - Y
```

```
##      [,1] [,2] [,3]
## [1,]   -8   -2    4
## [2,]   -6    0    6
## [3,]   -4    2    8
```

```r
X * Y
```

```
##      [,1] [,2] [,3]
## [1,]    9   24   21
## [2,]   16   25   16
## [3,]   21   24    9
```

```r
X / Y
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.1111111 0.6666667 2.333333
## [2,] 0.2500000 1.0000000 4.000000
## [3,] 0.4285714 1.5000000 9.000000
```

- Note that `X * Y` is **not** matrix multiplication.
- It is element by element multiplication. (Same for `X / Y`).

- Matrix multiplication uses %*%.

```
X %*% Y
```

```
##      [,1] [,2] [,3]
## [1,]   90   54   18
## [2,]  114   69   24
## [3,]  138   84   30
```

- t() which gives the transpose of a matrix

```
t(X)
```

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

- `solve()` which returns the inverse of a square matrix if it is invertible.

```
Z = matrix(c(9, 2, -3, 2, 4, -2, -3, -2, 16), 3, byrow = TRUE)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    9    2   -3
## [2,]    2    4   -2
## [3,]   -3   -2   16
```

```
solve(Z)
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.12931034 -0.05603448 0.01724138
## [2,] -0.05603448  0.29094828 0.02586207
## [3,]  0.01724138  0.02586207 0.06896552
```

- To verify that `solve(Z)` returns the inverse, we multiply it by `Z`.

```r
solve(Z) %*% Z
```

```
##                [,1]          [,2]         [,3]
## [1,] 1.000000e+00 -6.245005e-17 0.000000e+00
## [2,] 8.326673e-17  1.000000e+00 5.551115e-17
## [3,] 2.775558e-17  0.000000e+00 1.000000e+00
```

```r
diag(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```r
all.equal(solve(Z) %*% Z, diag(3))
```

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

Exercise

- Solve the following simultanoues equations using matrix calculation

$$
\begin{aligned}
2x_1 + 3x_2 = &\quad 10 \\
5x_1 + x_2 = &\quad 20
\end{aligned}
$$

- Hint: You can write this as $Ax = y$ where A is the 2-times-2 matrix, x and y are vectors with the length of 2.

Getting information of matrix

- Matrix specific functions for obtaining dimension and summary information.

```
X = matrix(1:6, 2, 3)
X
```

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

```
dim(X)
```

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

```
rowSums(X)
```

```
## [1]  9 12
```

```r
colSums(X)
```

```
## [1]  3  7 11
```

```r
rowMeans(X)
```

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

```r
colMeans(X)
```

```
## [1] 1.5 3.5 5.5
```

- The `diag()` function can be used in a number of ways. We can extract the diagonal of a matrix.

```
diag(Z)
```

```
## [1]  9  4 16
```

- Or create a matrix with specified elements on the diagonal. (And `0` on the off-diagonals.)

```
diag(1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    3    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    5
```

- Or, lastly, create a square matrix of a certain dimension with `1` for every element of the diagonal and `0` for the off-diagonals.

Section 4

List

List

- A list is a one-dimensional heterogeneous data structure.
  - It is indexed like a vector with a single integer value,
  - but each element can contain an element of any type.

```
# creation
list(42, "Hello", TRUE)
```

```
## [[1]]
## [1] 42
##
## [[2]]
## [1] "Hello"
##
## [[3]]
## [1] TRUE
```

```r
ex_list = list(
  a = c(1, 2, 3, 4),
  b = TRUE,
  c = "Hello!",
  d = function(arg = 42) {print("Hello World!")},
  e = diag(5)
)
```

- Lists can be subset using two syntaxes,
    1. the $ operator, and
    2. square brackets [].

```
# subsetting
ex_list$e
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```r
ex_list[1:2]
```

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] TRUE
```

```
ex_list[c("e", "a")]
```

```
## $e
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
##
## $a
## [1] 1 2 3 4
```

```r
ex_list["e"]
```

```
## $e
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```r
ex_list$d
```

```
## function(arg = 42) {print("Hello World!")}
```

Data Frames

- We will talk about Dataframe in the next chapter.

Section 5

Control flow

if/else syntax

- The if/else syntax is:

```
if (...) {
  some R code
} else {
  more R code
}
```

- Example: To see whether x is large than y.

```
x = 1
y = 3
if (x > y) {
  z = x * y
  print("x is larger than y")
} else {
  z = x + 5 * y
  print("x is less than or equal to y")
}
```

```
## [1] "x is less than or equal to y"
```

```
z
```

```
## [1] 16
```

- R also has a special function `ifelse()`
  - It returns one of two specified values based on a conditional statement.

```
ifelse(4 > 3, 1, 0)
```

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

- The real power of `ifelse()` comes from its ability to be applied to vectors.

```
fib = c(1, 1, 2, 3, 5, 8, 13, 21)
ifelse(fib > 6, "Foo", "Bar")
```

```
## [1] "Bar" "Bar" "Bar" "Bar" "Bar" "Foo" "Foo" "Foo"
```

for loop

- A `for` loop repeats the same procedure for the specified number of times

```
x = 11:15
for (i in 1:5) {
  x[i] = x[i] * 2
}

x
```

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

- Note that this `for` loop is very normal in many programming languages.
- In `R` we would not use a loop, instead we would simply use a vectorized operation.
    - `for` loop in `R` is known to be very slow.

```
x = 11:15
x = x * 2
x
```

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

Section 6

Function

Functions

- To use a function,
  - you simply type its name,
  - followed by an open parenthesis,
  - then specify values of its arguments,
  - then finish with a closing parenthesis.
- An **argument** is a variable which is used in the body of the function.

```
# The following is just a demonstration,
# not the real function in R.
function_name(arg1 = 10, arg2 = 20)
```

- We can also write our own functions in R.

Example

- Example: "standardize" variables

$$\frac{x - \bar{x}}{s}$$

- When writing a function, there are three thing you must do.
  1. Give the function a name. Preferably something that is short, but descriptive.
  2. Specify the arguments using `function()`
  3. Write the body of the function within curly braces, `{}`.

```
standardize = function(x) {
  m = mean(x)
  std = sd(x)
  result = (x - m) / std
  return(result)
}
```

- Here the name of the function is `standardize`,
- The function has a single argument `x` which is used in the body of function.
- Note that the output of the final line of the body is what is returned by the function.

- Let's test our function
- Take a random sample of size `n = 10` from a normal distribution with a mean of 2 and a standard deviation of 5.

```
test_sample = rnorm(n = 10, mean = 2, sd = 5)
test_sample
```

```
##  [1] -1.5143403 10.7411552 -2.2773664  6.6904636 -5.3841708
##  [7] 11.2472866  3.2674091  0.1412592  1.7623680
```

```
standardize(x = test_sample)
```

```
##  [1] -0.79748119  1.43811087 -0.93666895  0.69920204 -1.503
##  [7]  1.53043708  0.07478391 -0.49547420 -0.19975888
```

- The same function can be written more simply.

```
standardize = function(x) {
  (x - mean(x)) / sd(x)
}
```

- When specifying arguments, you can provide default arguments.

```
power_of_num = function(num, power = 2) {
  num ^ power
}
```

- Let's look at a number of ways that we could run this function to perform the operation 10^2 resulting in 100.

```
power_of_num(10)
```

```
## [1] 100
```

```
power_of_num(10, 2)
```

```
## [1] 100
```

```
power_of_num(num = 10, power = 2)
```

```
## [1] 100
```

```
power_of_num(power = 2, num = 10)
```

```
## [1] 100
```

- Note that without using the argument names, the order matters. The following code will not evaluate to the same output as the previous example.

```
power_of_num(2, 10)
```

```
## [1] 1024
```

- Also, the following line of code would produce an error since arguments without a default value must be specified.

```
power_of_num(power = 5)
```

- To further illustrate a function with a default argument, we will write a function that calculates sample variance two ways.
- By default, the function will calculate the unbiased estimate of σ^2, which we will call s^2.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x-\bar{x})^2$$

- It will also have the ability to return the biased estimate (based on maximum likelihood) which we will call $\hat{\sigma}^2$.

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x-\bar{x})^2$$

```r
get_var = function(x, unbiased = TRUE) {

  if (unbiased == TRUE){
    n = length(x) - 1
  } else if (unbiased == FALSE){
    n = length(x)
   }

  (1 / n) * sum((x - mean(x)) ^ 2)
}
```

```r
get_var(test_sample)
```

```
## [1] 30.05223
```

```r
get_var(test_sample, unbiased = TRUE)
```

```
## [1] 30.05223
```

```r
var(test_sample)
```

```
## [1] 30.05223
```

- We see the function is working as expected, and when returning the unbiased estimate it matches R's built in function `var()`. Finally, let's examine the biased estimate of σ^2.

```
get_var(test_sample, unbiased = FALSE)
```

```
## [1] 27.047
```