1. R as Calculator

You can start with playing around with R, use it as a calculator

1 + 3

*#> [1] 4*

6\*(5-1)

*#> [1] 24*

Some useful operators:

* +: addition
* -: subtraction
* \*: multiplication
* /: division
* x %% y: modulus (Remainder from division)
* x %/% y: integer division
* ^ or \*\*: exponentiation, e.g. 3^2 or 3\*\*2 to compute 3232

**?** What is the output of the following code?

* -1^2 = ?
* (-1)^2 = ?

You can also try out some more complicated (fancier) computations, such as Trigonometric functions (high school nightmare):

* sin(): sine
* cos(): cosine
* tan(): tangent
* exp(): Exponential with base *e*
* log(): Logarithm
* log10(): Logarithm with the base 10
* sqrt(): Square root
* abs(): Absolute value
* round(): Round the value
* floor(): Round down the value
* ceiling(): Round up the value
* factorial(): Factorial function
* gamma(): Gamma function
* digamma(): Digamma function
* …

**?** What are the solutions to the following equations

* sin(π2π2) = ?
* cos(ππ) = ?
* tan(π4π4) = ?

**?** Can you verify if the function factorial returns the correct result of 5!5!?

5\*4\*3\*2\*1

*#> [1] 120*

factorial(5)

*#> [1] 120*

*Exercise Pack 1*

1. Calculate the volume of a sphere (ball) whose raduis r=2r=2
2. What is 35∘C35∘C in Fahrenheit (∘F∘F)?
3. Use 5 different θθ values to calculate the result of sin(θ)2+cos(θ)2sin(θ)2+cos(θ)2
4. In a right triangle with sides a=b=2<ca=b=2<c, how long is cc? (Use Pythagorean Theorem)

2. Basic R Objects

Declare value

Use <- to declare a value to an object.

x<- 1

x

*#> [1] 1*

of course, using = to assign the value also works.

x = 1

x

*#> [1] 1*

Use class, mode or typeof to check the type of the object.

x<- 1.1

class(x)

*#> [1] "numeric"*

mode(x)

*#> [1] "numeric"*

typeof(x)

*#> [1] "double"*

Working environment

List all the objects under the current working environment.

ls()

* rm() remove object
* object.size() memory used by the object

Remove all the objects under the current working environment.

rm(list = ls())

Numbers

Numeric (real number)

x<- 0.8

class(x)

*#> [1] "numeric"*

Complex number

x<- 3+0i

class(x)

*#> [1] "complex"*

Scientific notation

x<- 9.6e-4

Infinity (use is.infinite or is.finite to test)

x<- 1/0

x

*#> [1] Inf*

is.infinite(x)

*#> [1] TRUE*

is.finite(x)

*#> [1] FALSE*

Not a number (undefined result, use is.nan to test)

x<- 0/0

x

*#> [1] NaN*

is.nan(x)

*#> [1] TRUE*

Null object (use is.null to test)

x<- NULL

is.null(x)

*#> [1] TRUE*

Not available/missing value (use is.na to test)

x<- NA

is.na(x)

*#> [1] TRUE*

is.nan(x)

*#> [1] FALSE*

Use identical to check if two objects are identical

x<- 1e-3

y<- 0.001

identical(x, y)

*#> [1] TRUE*

Logical

x<- TRUE

x

*#> [1] TRUE*

y<- FALSE

y

*#> [1] FALSE*

or

x<- T

x

*#> [1] TRUE*

y<- F

y

*#> [1] FALSE*

Some logical operators

* !: not
* ==: exactly equal to
* !=: not equal to
* &: and
* |: or
* <: less than
* <=: less than or equal to
* >: greater than
* >=: greater than or equal to
* **?** !!!T | !F = ?
* **?** T > F = ?
* **?** T + F = ?

Strings

a <- "hello"

a

*#> [1] "hello"*

class(a)

*#> [1] "character"*

print("Hello R!")

*#> [1] "Hello R!"*

Functions

Define your own function with function.

my\_square<- **function**(x){

x^2

}

my\_square(4)

*#> [1] 16*

or

my\_plus<- **function**(x, y){

x + y

}

my\_plus(2, 3)

*#> [1] 5*

Note that the last element in the function will be returned as the output value. Or you can use return to specify your output value.

my\_square<- **function**(x){

**return**(x^2)

x^3 *# does not affect the output*

}

my\_square(4)

*#> [1] 16*

*Exercise Pack 2*

1. Check identical(as.integer(5), 5.0) and as.integer(5.0) == 5, which one is TRUE?
2. List all object under the current environment. Check the memory used by the first object in the list.
3. Check the types of the following objects (choose one from class, typeof, and mode)

* Inf - Inf
* Inf + Inf
* 0/0
* sin(Inf)
* Inf/0

1. Complete the following function to compute the area of an ellipse where a and b are the axes.

ellipse\_area<- **function**(a, b){

}

ellipse\_area(3, 5)

3. Data structures

Vectors

All the elements in a vector should be of the same object type.

Use c to create a *vector*

*## number*

exp\_1<- c(1, 2, 3, 4, 5)

exp\_1

*#> [1] 1 2 3 4 5*

*## logical*

exp\_2<- c(TRUE, FALSE, FALSE, TRUE)

exp\_2

*#> [1] TRUE FALSE FALSE TRUE*

*## string*

exp\_3<- c("I", "am", "a", "meaningless", "example")

exp\_3

*#> [1] "I" "am" "a" "meaningless" "example"*

or use vector to define an empty vector

emp\_vec<- vector()

emp\_vec

*#> logical(0)*

Use seq to create a vector with sequential numbers

a<- seq(from = 1, to = 5, by = 1)

a

*#> [1] 1 2 3 4 5*

or just simply:

a<- 1:5

a

*#> [1] 1 2 3 4 5*

Use rep to create a vector with replicate elements

b<- rep(x = 1, times = 3)

b

Use sample to create a vector with random numbers

s<- sample(x = 1:100, size = 5)

s

*#> [1] 24 11 63 75 85*

Set the seed with set.seed function before sampling if you want to reproduce the result.

sample(1:100, 5)

*#> [1] 98 29 45 99 79*

sample(1:100, 5)

*#> [1] 99 37 66 26 43*

set.seed(123)

sample(1:100, 5)

*#> [1] 29 79 41 86 91*

set.seed(123)

sample(1:100, 5)

*#> [1] 29 79 41 86 91*

* min() and max(): minimum value and maximum value within a vector
* which.min() and which.max(): index of the minimal element and maximal element of a vector
* pmin() and pmax(): element-wise minima and maxima of several vectors
* sum() and prod(): sum and product of the elements of a vector
* cumsum() and cumprod(): cumulative sum and product of the elements of a vector

s<- sample(1:100, 5)

s

*#> [1] 5 53 88 54 44*

min(s)

*#> [1] 5*

max(s)

*#> [1] 88*

which.min(s)

*#> [1] 1*

which.max(s)

*#> [1] 3*

Concatenate vectors

vec\_1<- c(1, 1, 1)

vec\_2<- c(2, 2, 2)

vec\_join<- c(vec\_1, vec\_2)

vec\_join

*#> [1] 1 1 1 2 2 2*

vec\_3<- c(3, 3, 3)

vec\_join<- c(vec\_1, vec\_2, vec\_3)

vec\_join

*#> [1] 1 1 1 2 2 2 3 3 3*

Subset a vector

a<- c(1, 2, 3, 4, 5)

*# extract with indices*

a[c(1, 3, 5)]

*#> [1] 1 3 5*

*# extract with logicals*

a[c(T, F, T, F, T)]

*#> [1] 1 3 5*

a %% 2 == 1

*#> [1] TRUE FALSE TRUE FALSE TRUE*

a[(a %% 2 == 1)]

*#> [1] 1 3 5*

*# omit*

a[-c(2, 4)]

*#> [1] 1 3 5*

a[-which(a %% 2 == 0)]

*#> [1] 1 3 5*

NA values in a vector

a<- c(1, NA, 2, NA, 3)

a

*#> [1] 1 NA 2 NA 3*

b<- c(1, 2, 3, 4, 5)

b \* c(1, NA, 1, NA, 1)

*#> [1] 1 NA 3 NA 5*

*# replace NA with 0*

a[is.na(a)]<- 0

a

*#> [1] 1 0 2 0 3*

Vectorized computation

a<- c(1, 2, 3, 4)

b<- c(5, 6, 7, 8)

a + b

*#> [1] 6 8 10 12*

a \* b

*#> [1] 5 12 21 32*

* **?** a = c(1, 2, 3, 4) and b = c(1, 2, 3) What is the value of a\*b?

Factors

Represente categorical data with specifying levels (e.g. gender, education). A factor is stored as a vector of integers with corresponding labels.

x<- c("Python user", "R user", "C++ user", "R user", "C++ user", "JAVA user", "R user")

f\_x<- factor(x)

f\_x

*#> [1] Python user R user C++ user R user C++ user JAVA user*

*#> [7] R user*

*#> Levels: C++ user JAVA user Python user R user*

levels(f\_x)

*#> [1] "C++ user" "JAVA user" "Python user" "R user"*

nlevels(f\_x)

*#> [1] 4*

class(f\_x)

*#> [1] "factor"*

summary(f\_x)

*#> C++ user JAVA user Python user R user*

*#> 2 1 1 3*

or assign the labels you prefer

x<- c(1, 2, 1, 2, 1, 1, 1)

f\_x<- factor(x, labels = c("male", "female"))

f\_x

*#> [1] male female male female male male male*

*#> Levels: male female*

summary(f\_x)

*#> male female*

*#> 5 2*

or by spliting a vector into groups with the function cut

x<- c(12, 64, 47, 36, 31, 64, 25, 34, 6, 89)

f\_x<- cut(x, c(0, 14, 64, 100))

f\_x

*#> [1] (0,14] (14,64] (14,64] (14,64] (14,64] (14,64] (14,64]*

*#> [8] (14,64] (0,14] (64,100]*

*#> Levels: (0,14] (14,64] (64,100]*

levels(f\_x)<- c("child", "labor", "aged")

summary(f\_x)

*#> child labor aged*

*#> 2 7 1*

Matrices

Define a matrix

x<- matrix(1:15, nrow = 3, ncol = 5, byrow = F)

x

*#> [,1] [,2] [,3] [,4] [,5]*

*#> [1,] 1 4 7 10 13*

*#> [2,] 2 5 8 11 14*

*#> [3,] 3 6 9 12 15*

x<- matrix(1:15, nrow = 3, ncol = 5, byrow = T)

x

*#> [,1] [,2] [,3] [,4] [,5]*

*#> [1,] 1 2 3 4 5*

*#> [2,] 6 7 8 9 10*

*#> [3,] 11 12 13 14 15*

Subset a matrix

x<- matrix(1:15, nrow = 3, ncol = 5)

x[2,]

*#> [1] 2 5 8 11 14*

x[,1]

*#> [1] 1 2 3*

x[2,1:3]

*#> [1] 2 5 8*

x[1:2,c(1,3)]

*#> [,1] [,2]*

*#> [1,] 1 7*

*#> [2,] 2 8*

Some useful functions and operators for matrix computations:

* %\*%: matrix multiplication
* %o%: outer product
* crossprod(): cross product
* t(): tranpose matrix
* diag(): diagnal
* det(): calculate the determinant of the matrix
* solve(): obtain the inverse matrix

A<- matrix(sample(1:10, 4), 2, 2)

B<- matrix(sample(1:10, 6), 2, 3)

A

*#> [,1] [,2]*

*#> [1,] 10 6*

*#> [2,] 5 9*

B

*#> [,1] [,2] [,3]*

*#> [1,] 2 10 8*

*#> [2,] 9 1 5*

A%\*%B

*#> [,1] [,2] [,3]*

*#> [1,] 74 106 110*

*#> [2,] 91 59 85*

t(A) %\*% A

*#> [,1] [,2]*

*#> [1,] 125 105*

*#> [2,] 105 117*

crossprod(A)

*#> [,1] [,2]*

*#> [1,] 125 105*

*#> [2,] 105 117*

solve(A)

*#> [,1] [,2]*

*#> [1,] 0.15000000 -0.1000000*

*#> [2,] -0.08333333 0.1666667*

Arrays

Define an array

x<- array(1:24, dim = c(4, 3, 2))

x

*#> , , 1*

*#>*

*#> [,1] [,2] [,3]*

*#> [1,] 1 5 9*

*#> [2,] 2 6 10*

*#> [3,] 3 7 11*

*#> [4,] 4 8 12*

*#>*

*#> , , 2*

*#>*

*#> [,1] [,2] [,3]*

*#> [1,] 13 17 21*

*#> [2,] 14 18 22*

*#> [3,] 15 19 23*

*#> [4,] 16 20 24*

x[3,,]

*#> [,1] [,2]*

*#> [1,] 3 15*

*#> [2,] 7 19*

*#> [3,] 11 23*

x[3,2,]

*#> [1] 7 19*

x[3,2,1]

*#> [1] 7*

Lists

x<- list(name = "miina", age = 25, score = 1, pass = T, gender = "female")

length(x)

*#> [1] 5*

x$name

*#> [1] "miina"*

x[2]

*#> $age*

*#> [1] 25*

x[[3]]

*#> [1] 1*

x["pass"]

*#> $pass*

*#> [1] TRUE*

x[["gender"]]

*#> [1] "female"*

Data Frames

A data frame generalized matrix in which each column may have different object types. It can be also seen as aa list of colume vectors with all equal length, thus, the way to extract the colums is the same as how you do on a list.

toy\_dat<- data.frame(id = 1:5, age = c(15, 5, 11, 10, 95), city = c("Tampere", "Pori", "Tampere", "Helsinki", "Turku"))

toy\_dat

*#> id age city*

*#> 1 1 15 Tampere*

*#> 2 2 5 Pori*

*#> 3 3 11 Tampere*

*#> 4 4 10 Helsinki*

*#> 5 5 95 Turku*

toy\_dat$id

*#> [1] 1 2 3 4 5*

toy\_dat[2]

*#> age*

*#> 1 15*

*#> 2 5*

*#> 3 11*

*#> 4 10*

*#> 5 95*

Import and Export Dataset

The example dataset steam\_subset.csv can be found [here](https://raw.githubusercontent.com/HumbleLu/IntroR/master/steam_subset.csv). (Right click -> Save as). The colums are seperated with comma( , ) and the first line is the column names.

Read the data set from a file with read.table, use functions head and str to check the dataset

steam<- read.table(file = "steam\_subset.csv", sep = ",", header = T)

head(steam)

*#> UserId Level Showcases Comments Badges*

*#> 1 1 17 1 24 10*

*#> 2 2 55 2 105 47*

*#> 3 3 0 0 11 0*

*#> 4 4 16 1 15 12*

*#> 5 5 52 4 98 38*

*#> 6 6 27 2 5 41*

str(steam)

*#> 'data.frame': 500 obs. of 5 variables:*

*#> $ UserId : int 1 2 3 4 5 6 7 8 9 10 ...*

*#> $ Level : int 17 55 0 16 52 27 57 14 21 71 ...*

*#> $ Showcases: int 1 2 0 1 4 2 5 1 2 7 ...*

*#> $ Comments : int 24 105 11 15 98 5 1024 16 25 111 ...*

*#> $ Badges : int 10 47 0 12 38 41 72 10 21 66 ...*

or use read.csv to read the file

steam<- read.csv(file = "steam\_subset.csv", header = T, sep = ",")

head(steam)

*#> UserId Level Showcases Comments Badges*

*#> 1 1 17 1 24 10*

*#> 2 2 55 2 105 47*

*#> 3 3 0 0 11 0*

*#> 4 4 16 1 15 12*

*#> 5 5 52 4 98 38*

*#> 6 6 27 2 5 41*

str(steam)

*#> 'data.frame': 500 obs. of 5 variables:*

*#> $ UserId : int 1 2 3 4 5 6 7 8 9 10 ...*

*#> $ Level : int 17 55 0 16 52 27 57 14 21 71 ...*

*#> $ Showcases: int 1 2 0 1 4 2 5 1 2 7 ...*

*#> $ Comments : int 24 105 11 15 98 5 1024 16 25 111 ...*

*#> $ Badges : int 10 47 0 12 38 41 72 10 21 66 ...*

Computation on the variables

mean(steam$Level)

*#> [1] 37.648*

sd(steam$Level)

*#> [1] 68.99508*

Attach a dataset

*# stick to it*

**attach**(steam)

mean(Level)

*#> [1] 37.648*

sd(Level)

*#> [1] 68.99508*

*# get rid of it*

**detach**(steam)

Save the dataset to a csv file

write.csv(toy\_dat, file = "toy.csv", row.names = F)

*Exercise Pack 3*

1. Create a vector z of length 10 with variance equals to 0 and mean equals to 5. Verify with function mean and var.
2. Create a function which returns the sum of the maximum and the mininum value of the input vector
3. [!!! Only for practice (answers proviede below)] Create a function which approximates the sin function with 4th-order taylor series.

sin\_approx<- **function**(x){

x - x^3 / factorial(3)

}

sin\_approx(0)

*#> [1] 0*

sin(0)

*#> [1] 0*

sin\_approx(1)

*#> [1] 0.8333333*

sin(1)

*#> [1] 0.841471*

1. Compute the CV (coefficient of variation) values of variables Level and Badges in the steam dataset.
2. Create a toy data frame object with at least 5 rows and 3 columns, save it to a .csv file.

4. Computations

Loops

For loop

**for**(i **in** c(1, 3, 5)){

print(i)

}

*#> [1] 1*

*#> [1] 3*

*#> [1] 5*

or

**for**(i **in** seq(1, 5, 2)){

print(i)

}

*#> [1] 1*

*#> [1] 3*

*#> [1] 5*

While loop

i<- 1

**while**(i <= 5){

print(i)

i<- i + 2

}

*#> [1] 1*

*#> [1] 3*

*#> [1] 5*

Conditional statement

if statement

x<- 2

**if**(x > 0){

print("Positive number")

}

*#> [1] "Positive number"*

if … else statement

x<- -1

**if**(x > 0){

print("Positive number")

} **else** {

print("Not a positive number")

}

*#> [1] "Not a positive number"*

or

ifelse(test = x>0, yes = "Positive number", no = "Not a positive number")

*#> [1] "Not a positive number"*

if … else ladder

x<- 0

**if**(x > 0){

print("Positive number")

} **else** **if**(x < 0){

print("Negative number")

} **else**{

print("Zero")

}

*#> [1] "Zero"*

More Functions

You can set up a default input

hello<- **function**(obj = "R"){

print(paste("Hello", obj, "!"))

}

hello()

*#> [1] "Hello R !"*

hello("World")

*#> [1] "Hello World !"*

A function can also generate a function. For example, the volume of a dd-dimensional hypersphere with radius rr is πd2Γ(d2+1)rdπd2Γ(d2+1)rd.

hypersphere<- **function**(d){

**function**(r){

(pi^(d/2)/gamma(d/2 + 1)) \* (r^d)

}

}

A circle is a 2-dimensional case:

circle<- hypersphere(2)

circle(1)

*#> [1] 3.141593*

A ball is a 3-dimensional case:

ball<- hypersphere(3)

ball(1)

*#> [1] 4.18879*

or

hypersphere(2)(1)

*#> [1] 3.141593*

hypersphere(3)(1)

*#> [1] 4.18879*

Operators are also functions

1 + 1

*#> [1] 2*

"+"(1,1)

*#> [1] 2*

**?** What is the output of "\*\*"(1, 2) ?

You can also define your operator with %

"%negative prod%"<- **function**(a, b){

a \* b \* (-1)

}

2 %negative prod% 3

*#> [1] -6*

Apply functions

apply takes a matrix, the MARGIN detering the row-wise (1) or column-wise (2) computation.

a<- matrix(1:6, 2, 3)

a

*#> [,1] [,2] [,3]*

*#> [1,] 1 3 5*

*#> [2,] 2 4 6*

apply(a, 1, sum)

*#> [1] 9 12*

apply(a, 2, sum)

*#> [1] 3 7 11*

lapply takes a list or a vector, returns a list.

lapply(c(1, 2, 3, 4), **function**(x) x + 1)

*#> [[1]]*

*#> [1] 2*

*#>*

*#> [[2]]*

*#> [1] 3*

*#>*

*#> [[3]]*

*#> [1] 4*

*#>*

*#> [[4]]*

*#> [1] 5*

sapply takes a list or a vector, returns a vector

sapply(c(1, 2, 3, 4), **function**(x) x + 1)

*#> [1] 2 3 4 5*

*Exercise Pack 4*

1. Use conditional statement to complete the following greeting function. The function prints "good morning", "good afternoon", "good evening" or "good night" according to the current hour.

greeting<- **function**(hour = lubridate::hour(Sys.time())){

}

greeting()

Note. lubridate::hour(Sys.time()) returns the current hour (0-24). If the package lubridate is not installed, use:

greeting<- **function**(hour = as.numeric(format(strptime(Sys.time(), "%Y-%m-%d %H:%M:%S") , "%H"))){

}

greeting()

1. Write a for loop to sum up all the elements of a vector, compare time consumption with simply using the function sum. Use the function Sys.time() to record the time, for example:

start\_t<- Sys.time()

*#computation*

end\_t<- Sys.time()

end\_t - start\_t

1. Write a loop to print out all the prime numbers smaller than 50.
2. Make an operator %$>€% to detect if the amount of money of the left hand side is greater than the right hand side. Where the left hand side is holding US dollars ($) and the right hand side is holding Euros (€). The exchange rate is 1€ = 1.1$.
3. Use apply to compute the coefficient of variation (CV) of each column in the steam dataset.
4. Use sapply to calcluate the object size of all the objects under current environment. Hint: you will need functions ls() and object.size().