The Basics of R

Instructor: Yang Fu

Preliminary

R for Windows: https://cran.r-project.org/bin/windows/base/
R for Mac OS X: https://cran.r-project.org/bin/macosx/

R studio: https://rstudio.com/products/rstudio/download/#download

Content

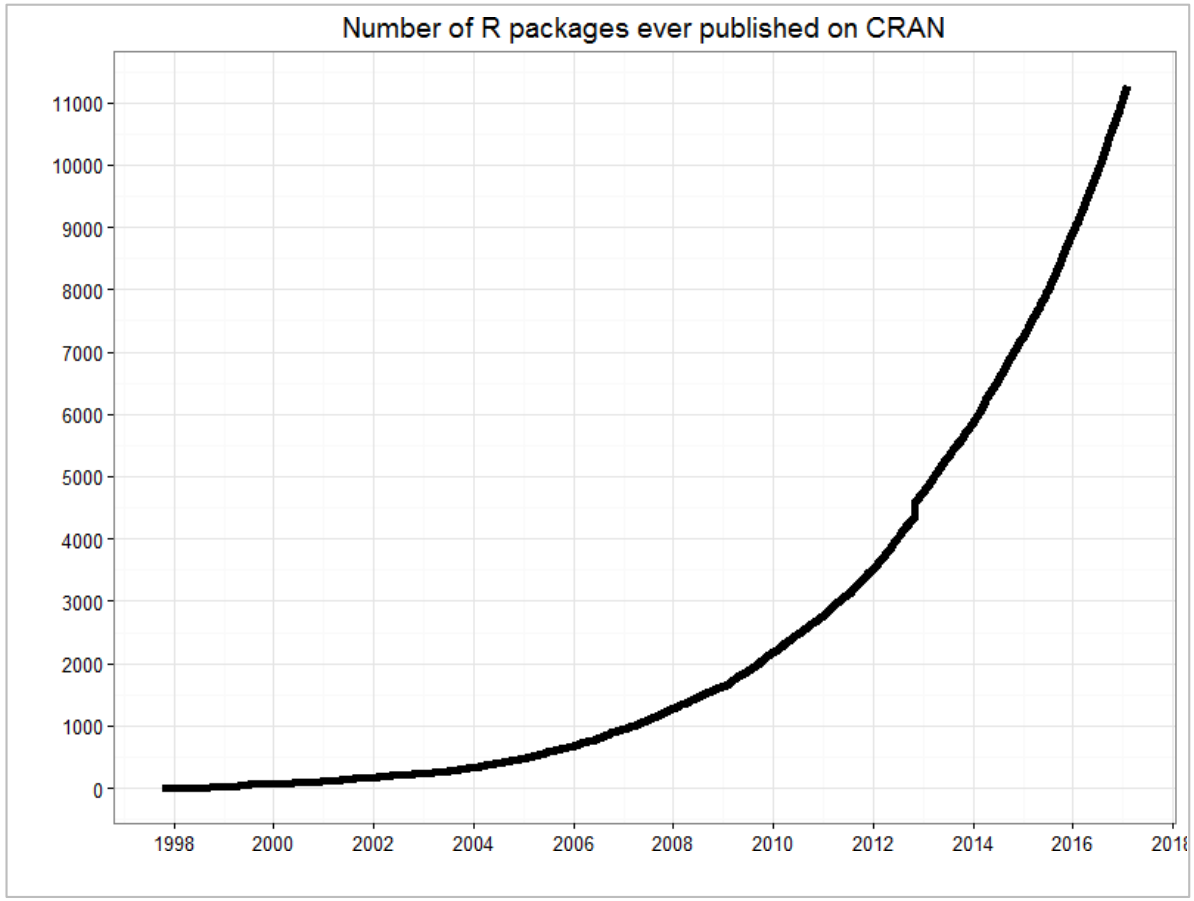
- Getting Started with R and Rstudio
- R Language Basics
- Data Structures
- Writing Functions
- Developing Workflows with R Scripts

Why R & What is R?

https://www.r-project.org/

- A free software environment for statistical computing and graphics
- R is the primary tool for statistical research
- Over 11,000 add-on packages available in CRAN

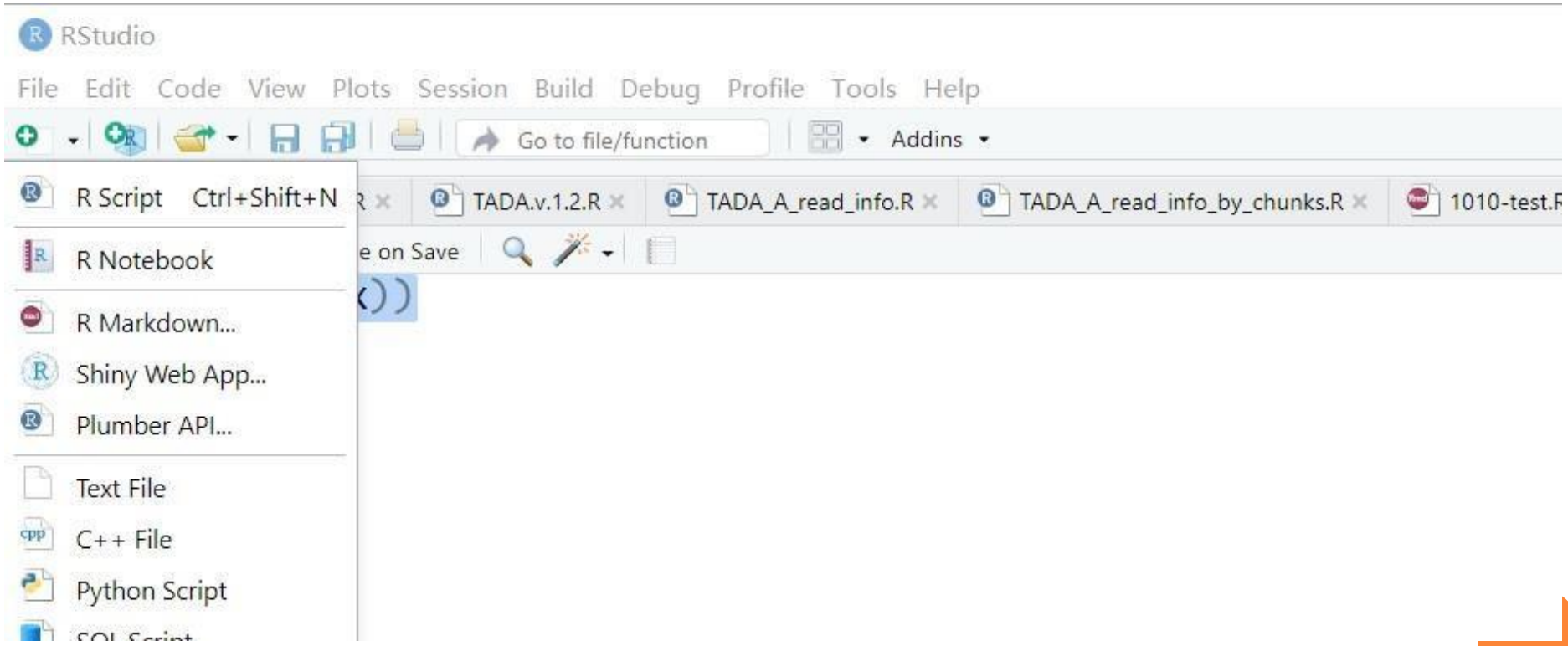
  New techniques are available to the public without delay



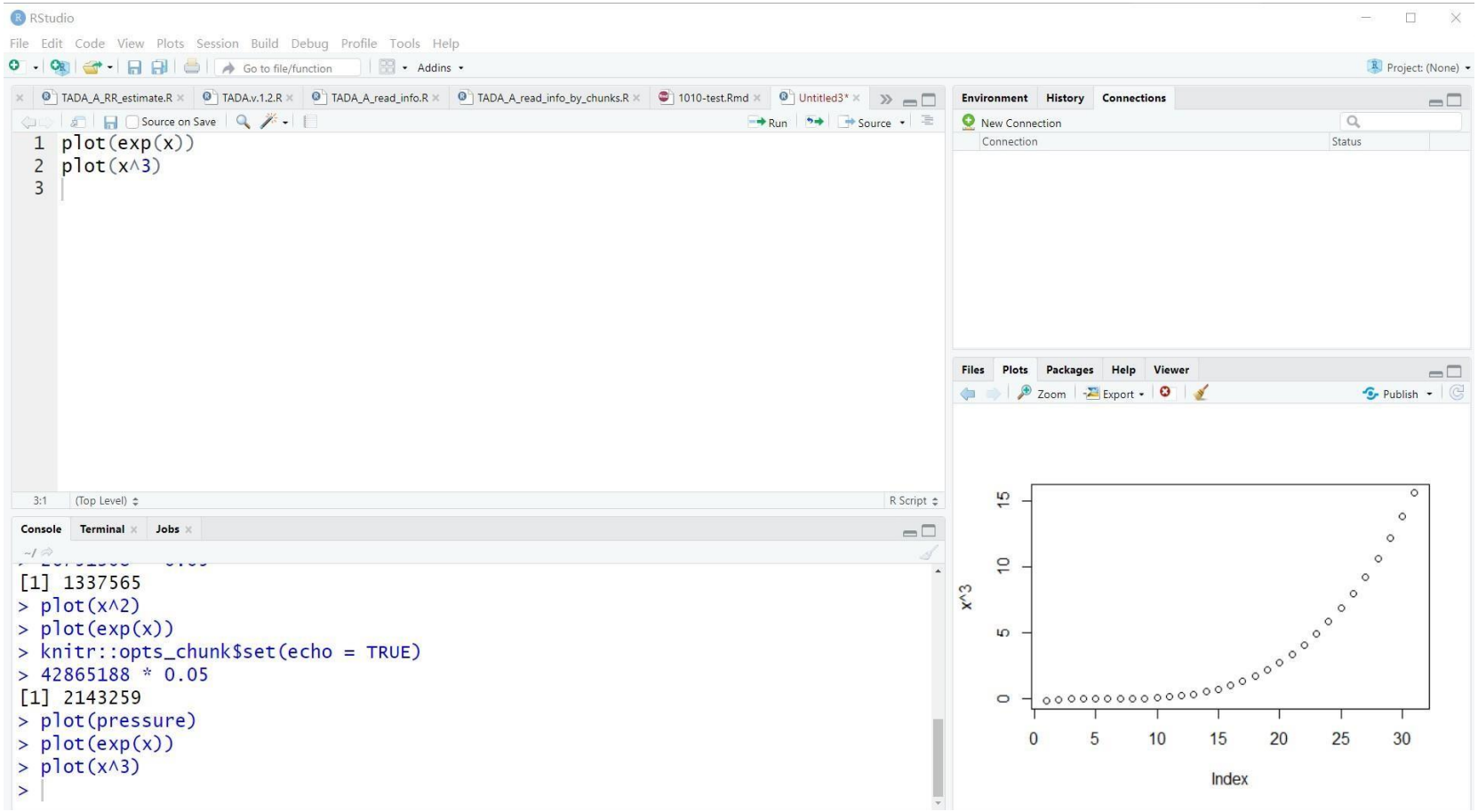
https://blog.revolutionanalytics.com/2017/01/cran-10000.html

Introduction to RStudio

a free, open source R integrated development environment. It provides a built in editor, works on all platforms (including on servers) and supports many features useful for working in R: syntax highlighting, quick access to R's help system, plots visible alongside code, and integration with version control.



Introduction to RStudio

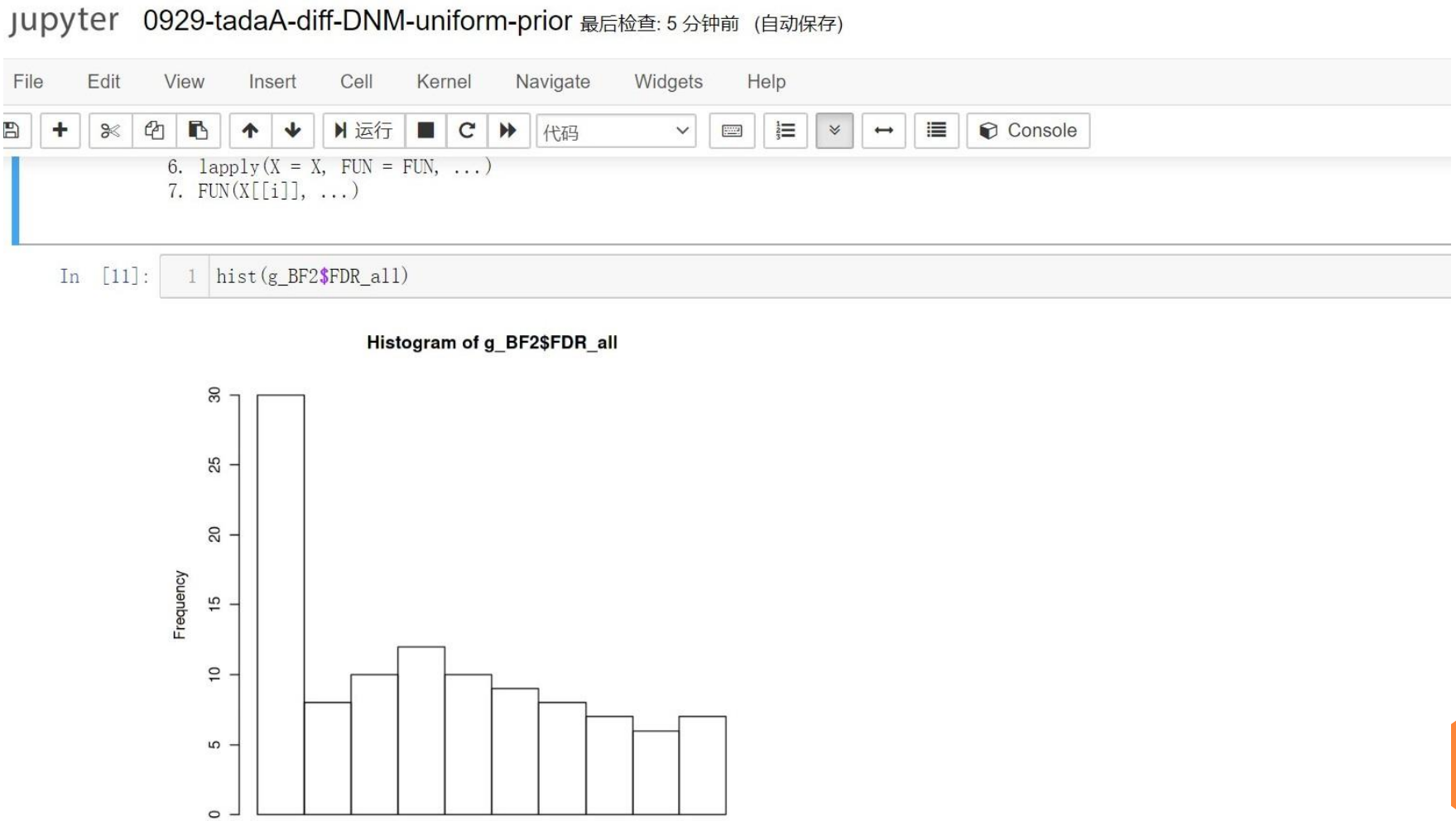


Run a certain line: ctrl + enter

Run all lines: ctrl + shift + enter

Alternative for Rstudio

Jupyter notebook with IR kernel



Tip: Good Enough Practices for Scientific Computing

Creating a new project

Good Enough Practices for Scientific Computing gives the following recommendations for project organization:

- Put each project in its own directory, which is named after the project.
- Put text documents associated with the project in the doc directory.
- Put raw data and metadata in the data directory, and files generated during cleanup and analysis in a results directory.
- Put source for the project's scripts and programs in the src directory, and programs brought in from elsewhere or compiled locally in the bin directory.
- Name all files to reflect their content or function.

Creating an R Markdown document

File -> New File -> RMarkdown.

- An (optional) YAML header surrounded by —s;
- R code chunks surrounded by ```s;
- text mixed with simple text formatting.

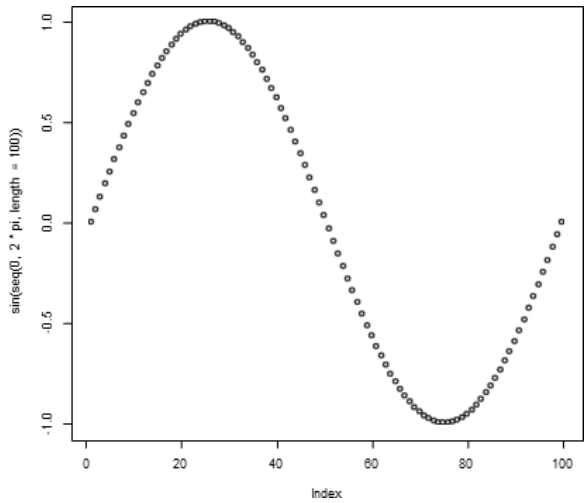
You can use the “Knit” button in the RStudio IDE to render the file and preview the output with a single click.

Markdown syntax:
https://markdown-zh.readthedocs.io/en/latest/

How to use R?

Simple Examples

```
> log2(32)
[1] 5
> sqrt(2)
[1] 1.414214
>seq( 0, 5 , length=6)
[1] 0 1 2 3 4 5
> plot(sin(seq(0, 2*pi, length=100)))
```



Assignment and Variable names

- Three ways to assign variables

  a = 6 (usually used for arguments)

  a <- 6 (common way to assign a value)

  6 -> a (rarely used)

- Naming rules

  1. Can include letters, numbers, . , and _
  2. Names are case sensitive
  3. Must start with . or a letter

  Example: `my.dat <- data.frame(x , y)`

Challenge 1

Which of the following are valid R variable names?

1. min_height
2. max.height
3. _age
4. .mass
5. MaxLength
6. min-length
7. 2widths
8. celsius2kelvin

Solution to challenge 1

1. min_height
2. max.height
3. _age
4. .mass → hidden variable
5. MaxLength
6. min-length
7. 2widths
8. celsius2kelvin

Data types in R

- Primitive (atomic) ***data types***

  Character 字符型, e.g. “abc”, “3”, “?”

  Numeric (integer, double)

  - double双整型
  - Integer整型, e.g. 2L (the L tells R to store this as an integer)

  Logical, e.g. TRUE(T), FALSE(F)

  Complex, e.g. 1+0i　a+bi（a、b均为实数），a 称为实部，b 称为虚部，i 称为虚数单位

- Out of these, *vectors*, *lists* and more ***data structures*** can be built.

Data structures in R

Data structures in R can be organized by

- their dimensionality (1D, 2D, or nD), and
- whether they're *homogeneous* (all contents must be of the same type) or *heterogeneous* (the contents can be of different types).

	Homogeneous	Heterogeneous
1D	Vector	List
2D	Matrix	Data frame
nD	Array	

str() is short for structure and it gives a compact, human readable description of any R data structure.

Vectors

- 1D data structure of the same type, usually created with c()
- four common types of atomic vectors : logical, integer, double (often called numeric), and character typeof()

```
dbl_var <- c(1, 2.5, 4.5) # double
# With the L suffix, you get an integer rather than a double
int_var <- c(1L, 6L, 10L) # integer
# Use TRUE and FALSE (or T and F) to create logical vectors
log_var <- c(TRUE, FALSE, T, F) # logical
chr_var <- c("these are", "some strings") # character
```

```
> typeof(c(1,2,3))
[1] "double"
> typeof(c(1L,2L,3L))
[1] "integer"
```

c(NA) logical
c(NA, 1) vector?

- Vectors are always flat, even if you nest c()'s, e.g. these two expressions give the same vector:

```
c(1, c(2, c(3, 4)))
c(1, 2, 3, 4)
```

Vector

Missing values are specified with NA, which is a logical vector of length 1. NA will always be coerced to the correct type if used inside c().

```
> typeof(c(NA))
[1] "logical"
> typeof(c(NA,1))
[1] "double"
```

Coercion

All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be coerced to the most flexible type. The coercion rules go: logical -> integer -> double -> character, where -> can be read as are transformed into. You can try to force coercion against this flow using the as. Functions.

Vector

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
[1] 0 0 1
```

as.character(), as.double(), as.integer(), or as.logical()

Challenge 2

Predict the type of following vectors:

```
a <- c("a", 1)
b <-c(TRUE, 1)
c <- c(1L, 10)
d <- c(a, b, c)
```

```
# "character"
# "double"
# "double"
# "character"
```

Lists

Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using list() instead of c():

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
```

```
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
```

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

[[2]]
[1] "a"

[[3]]
[1]  TRUE FALSE  TRUE

[[4]]
[1] 2.3 5.9
```

Compare the results of list() and c()

c() will combine several lists into one. If given a combination of atomic vectors and lists, c() will coerce the vectors to lists before combining them.

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
```

```
List of 2
 $ :List of 2
  ..$ : num 1
  ..$ : num 2
 $ : num [1:2] 3 4
```

```
str(y)
```

```
List of 4
 $ : num 1
 $ : num 2
 $ : num 3
 $ : num 4
```

Attributes

The three most important attributes:

- Names names()
- Dimensions dim() 二维及以上
- Class class()

```
x <- c(a = 1, b = 2, c = 3)
x
```

```
a b c
1 2 3
```

```
names(x)
```

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

```
y <- c(a = 1, 2, 3)
names(y)
```

```
[1] "a" ""  ""
```

```
z <- c(1, 2, 3)
names(z)
```

```
NULL
```

Factors

A ***factor*** is a variable that can only take a limited number of values, which are defined as ***levels***.

```
x <- factor(c("a", "b", "b", "a"))
levels(x)          #[1] "a" "b"

# You CANNOT use values that are not in the levels
x[2] <- "c"        # ERROR

# Factors can be converted to characters or integers
as.character(x)    #[1] "a" "b" "b" "a"
as.integer(x)      #[1] 1 2 2 1
```

```
x=c(1,2,3)
class(x)
class(factor(x))
```

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

Subsetting

six different ways we can subset any kind of object, and three different subsetting operators for the different data structures.

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
x
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

Accessing elements using their indices

In many programming languages (C and python, for example), the first element of a vector has an index of 0. In R, the first element is 1.

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

Subsetting

If we ask for a number outside of the vector, R will return missing values:

```
x[6]
<NA>
  NA
```

This is a vector of length one containing an NA, whose name is also NA.

If we ask for the 0th element, we get an empty vector:

```
x[0]
named numeric(0)
```

Skipping and removing elements

If we use a negative number as the index of a vector, R will return every element except for the one specified:

```
x[-2]
  a   c   d   e
5.4 7.1 4.8 7.5
```

How to get:

```
b   c   d
6.2 7.1 4.8
```

```
x[c(-1, -5)] # or x[-c(1,5)]
```

Subsetting by name

We can extract elements by using their name, instead of index:

```
x[c("a", "c")]
  a   c
5.4 7.1
```

This is usually a much more reliable way to subset objects: the position of various elements can often change when chaining together subsetting operations, but the names will always remain the same!

Unfortunately we can't skip or remove elements so easily.

To skip (or remove) a single named element:

```
x[-which(names(x) == "a")]
  b   c   e
6.2 7.1 7.5
```

Subsetting by name

```
names(x) == "a"
[1] TRUE FALSE FALSE FALSE
```

which then converts this to an index:

```
which(names(x) == "a")
[1] 1
```

Only the first element is TRUE, so which returns 1. Now that we have indices the skipping works because we have a negative index!

Skipping multiple named indices is similar, but uses a different comparison operator:

```
x[-which(names(x) %in% c("a", "c"))]
  b   e
6.2 7.5
```

How about non-unique names?

```
x <- 1:3
x
[1] 1 2 3
names(x) <- c('a', 'a', 'a')
x
a a a
1 2 3
x['a']  # only returns first value
a
1
x[which(names(x) == 'a')]  # returns all three values
a a a
1 2 3
```

Subsetting through other logical operations

We can also more simply subset through logical operations:

```
x[c(TRUE, TRUE, FALSE, FALSE)]
a a
1 2
```

Note that in this case, the logical vector is also recycled to the length of the vector we're subsetting!

```
x[c(TRUE, FALSE)]
a a
1 3
```

Since comparison operators evaluate to logical vectors, we can also use them to succinctly subset vectors:

```
x[x > 7]
named integer(0)
```

Challenge 3

Given the following code:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

```
  a.  b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

Come up with at least 3 different commands that will produce the following output:

```
  b.  c   d
6.2 7.1 4.8
```

```
x[c(2,3,4)]
x[c("b","c","d")]

x[-c(1,5)]

x[which(names(x) %in%
c("b","c","d"))]
```

Matrices and Arrays

- Adding a *dim* attribute to a vector allows it to behave like a multi-dimensional ***array***. A special case of the array is the ***matrix***, which has two dimensions.
- Matrices and arrays are created with `matrix()` and `array()`, or by using the assignment form of `dim()`.

```
c <- 1:6
dim(c) <- c(3,2) 行，列
```

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

```
a <- matrix(1:6, ncol=2, nrow=3)
ncol(a)
nrow(a)
rownames(a) <- c("A", "B")
colnames(a) <- c("a","b", "c")
```

- Example of using a matrix: The expression values for 10,000 genes for 30 tissue biopsies could be stored as a 10000x30 matrix.

Matrices(-like) functions and operator

To create a matrix:

```
# matrix() command to create matrix A with rows and cols
A=matrix(c(54, 49, 49, 41, 26, 43, 49, 50, 58, 71), nrow=5, ncol=2))
B=matrix(1, nrow=4, ncol=4)
```

To access matrix elements:

```
# matrix_name[row_no, col_no]
A[2,1]    # 2nd row, 1st column element
A[3,]     # 3rd row
A[,2]     # 2nd column of the matrix
A[2:4, c(3,1)] # submatrix of 2nd-4th # elements of the 3rd and 1st columns
A["KC",] # access row by name, "KC"
```

Statistical operations:

```
rowSums(A)
colSums(A)
rowMeans(A)
colMeans(A)
# max of each columns
apply(A, 2, max)
# min of each row
apply(A, 1, min)
```

Element by element options:

```
2*A+3; A+B; A*B; A/B;
```

Matrix/vector multiplication:

```
A %*% B;
```

Data frames

- A ***data frame*** is the most common way of storing data in R.
- It is supposed to represent the typical data table that researchers come up with, like a spreadsheet.
- It shares properties of both the matrix and the list. (Different columns may have different types.)

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE)
df
# subsetting column of x
df$x
# subsetting the first row
df[1,]
# subsetting rows where df$x>1
df[df$x>1,]
```

Note that the data.frame()'s default behaviour which turns strings into factors. Use stringAsFactors = FALSE to suppress this behaviour!

Read and write data files

- Reading a table of data can be done with read.table()

  Values are read into R as an object of data frame Can specify reading or discarding of headers

```
a <- read.table("a.txt")
HousePrice <- read.table("houses.data", header=TRUE)
```

- Use save() or write.table() functions to write data to file

```
save(x, file="x.Rdata") write.table(x,
file="x.txt", sep="\t")
```

Install.packages("data.table")

Other useful functions

Use cbind() to add a new column to a data frame.

Use rbind() to add a new row to a data frame.

Use na.omit() to remove rows from a data frame with NA values.

Use levels() and as.character() to explore and manipulate factors

Use str(), nrow(), ncol(), dim(), colnames(), rownames(), head() and typeof() to understand structure of the data frame

Read in a csv file using read.csv()

Use x%in%y or match to match columns in dataframes

Use an index created by match or merge to merge two dataframes

Writing your own functions

- Writing functions in R is defined by an assignment like:
  `fct <- function(arg1,arg2) { function_commands; }`
- Arguments may have default values (they become optional)

The general syntax for R functions is:

```
fun_name <- function(args) {
# body, containing R expressions
return(value)
}
```

Challenge 4

a = c(NA,1,2,3,4,5)
Write a function meanRemoveNA

Hint: na.rm=TRUE

```
### write your own function
```{r}
a = c(NA,1,2,3,4,5)
mean(a)

meanRemoveNA <- function(x){
  mean(x, na.rm=TRUE)
  }


mean(a)
meanRemoveNA(a)
```
```

Conditional statements

```
# if
if (x == some_value) {
        # do some stuff in here
} else if (x == other_value) {
        # elseif is optional
} else {
        #else is optional
}

#for
for (element in some_vector) {
        # iteration happens here
}

#while
while (something_is_true) {
        # do some stuff
}
```

Conditional statements

Examples:

```
quantity <- 25
# Set the if-else statement
if (quantity > 20) {
    print('You sold a lot!')
} else {
    print('Not enough for today')
}
```

```
quantity <- 10
# Create multiple condition statement
if (quantity <20) {
      print('Not enough for today')
} else if (quantity > 20 &&quantity <= 30) {
       print('Average day')
} else {
      print('What a great day!')
}
```

Loops

Examples:

```
for (year in  c(2010,2011,2012,2013,2014,2015)){
   print(paste("The year is", year))
}
```

```
i  <- 1
while (i  < 6) {
   print(i)
   i  <- i+ 1
}
```

Functions of lapply, sapply and apply

- The apply family provides an easier and faster way than for-loop
- **lapply**(li, function)

  Apply function to each element of the list li ; return a list
- **sapply**(li, function)

  Like lapply, but try to simply the result by converting it into a vector or array of appropriate size

```
li <- list("klaus","martin","georg")
sapply(li, toupper)
1    "KLAUS"    "MARTIN" "GEORG"

fct <- function(x) { return(c(x, x^2, x^3)) }
sapply(1:5, fct)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    4    9   16   25
[3,]    1    8   27   64  125
```

Functions of lapply, sapply and apply

- **apply**(mat, margin, function)

  Apply the function along some dimension of the matrix, according to margin, and return a vector or array of appropriate size

  If margin=1, apply by rows; if margin=2, then apply by columns

```
x <- matrix( 1:12, nrow=4, ncol=3)
x
 [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
apply( x, 1, sum)
[1] 15 18 21 24
apply( x, 2, sum)
[1] 10 26 42
```

Challenge 5

Write your funciton and get the following result.
(you may use for loop)

```
[1] "1 a"
[1] "1 b"
[1] "1 c"
[1] "2 a"
[1] "2 b"
[1] "2 c"
[1] "3 a"
[1] "3 b"
[1] "3 c"
```

Challenge 5

Reference solution:

```
for(i in 1:3){
  for(j in c('a', 'b', 'c')){
    print(paste(i,j))
  }
}
```

Challenge 6

Import “tss_test.txt” and get promoter regions of all protein-coding genes.

(upstream 1kb, downstream 1kb)

The outcome should be a data frame with a header.

(“chr”, “start”, “end”, “geneSymbol”)

R Packages

- R functions and datasets are organized into packages
  - Packages *base* and *stats* include many of the built-in functions of R
  - CRAN provides thousands of packages contributed by R users
- Package contents are only available when loaded
  - Load a package with `library(pkgname)`
- Packages must be installed before they can be loaded
  - Use `library()` to see installed packages
- To install or update a package
  - `install.packages(pkgname)`
  - `update.packages(pkgname)`
  - `github`
- You can also install from command line, if the package source is downloaded

```
R CMD INSTALL pkgname.tar.gz
```

Managing your environment

There are a few useful commands you can use to interact with the R session.

ls will list all of the variables and functions stored in the global environment (your working R session):

```
ls() # to list files use list.files() function
```

You can use rm to delete objects you no longer need:

```
rm(x)
```

If you have lots of things in your environment and want to delete all of them, you can pass the results of ls to the rm function:

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

Managing your environment

rm(list = ls())

In this case we've combined the two. Like the order of operations, anything inside the innermost parentheses is evaluated first, and so on.

In this case we've specified that the results of ls should be used for the list argument in rm. When assigning values to arguments by name, you must use the = operator!!

If instead we use <-, there will be unintended side effects, or you may get an error message:

rm(list <- ls())
Error in rm(list <- ls()): ... must contain names or character strings

Managing your environment

Print out your current version of R, as well as any packages you have loaded.

```
sessionInfo()
```

```
R version 3.6.3 (2020-02-29)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 18362)

Matrix products: default

Random number generation:
 RNG:     Mersenne-Twister
 Normal:  Inversion
 Sample:  Rounding
```

Materials for learning R

- Programming with R by Software Carpentry

  https://swcarpentry.github.io/r-novice-inflammation/

- R for genomics by Data Carpentry

  https://datacarpentry.org/R-genomics/

- R Tutorial from TutorialsPoint

  https://www.tutorialspoint.com/r/index.htm

- R manuals

  edited by the R Development Core Team

  https://cran.r-project.org/doc/manuals/r-release/R-intro.html

  https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf