R Guide

Jaewon Kim

2025-12-13

Table of contents

Welcome to “Introduction to R”

Here, you will learn how to use R for data analysis. Try practice questions, and feel free to contact me anything you have questions.

1 Structure of R Studio

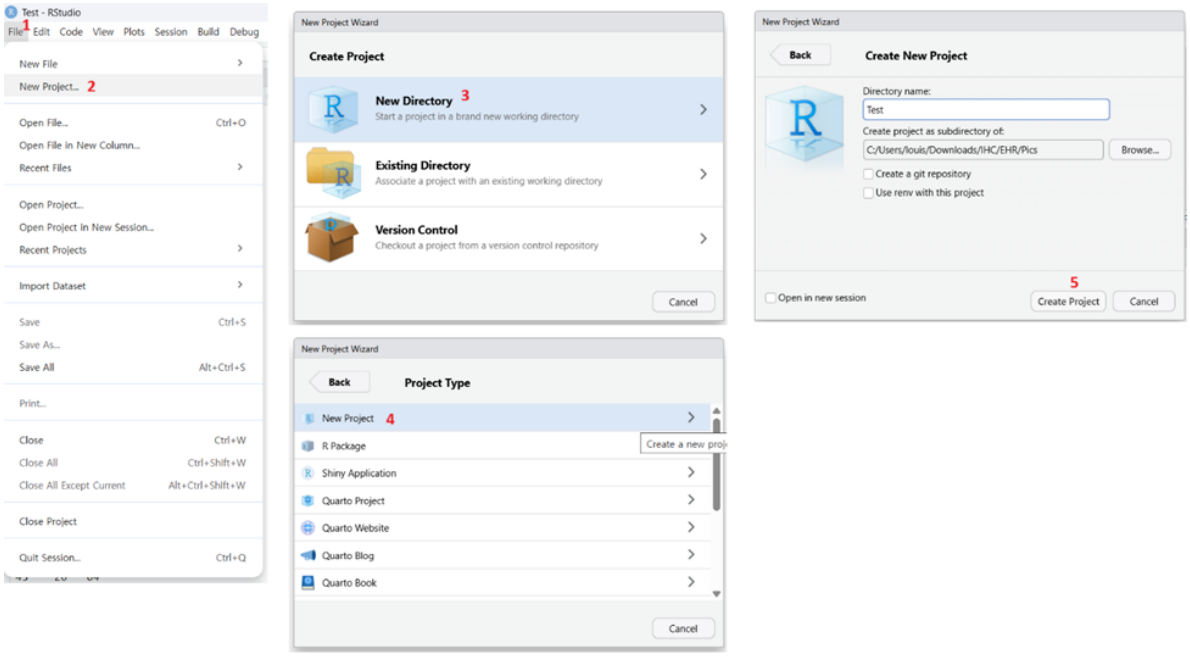
R is a programming language. R Studio is a UI that helps you write, run, and organize your R code. Let's explore how R Studio looks and what each section does.

1.1 Creating a Project

When we write codes and save data, it's important to keep everything organized so that R can remember where things were saved. To tell R Studio where to save and find your files, we use project. It's like a home for your codes and data.

Let's make a new project.

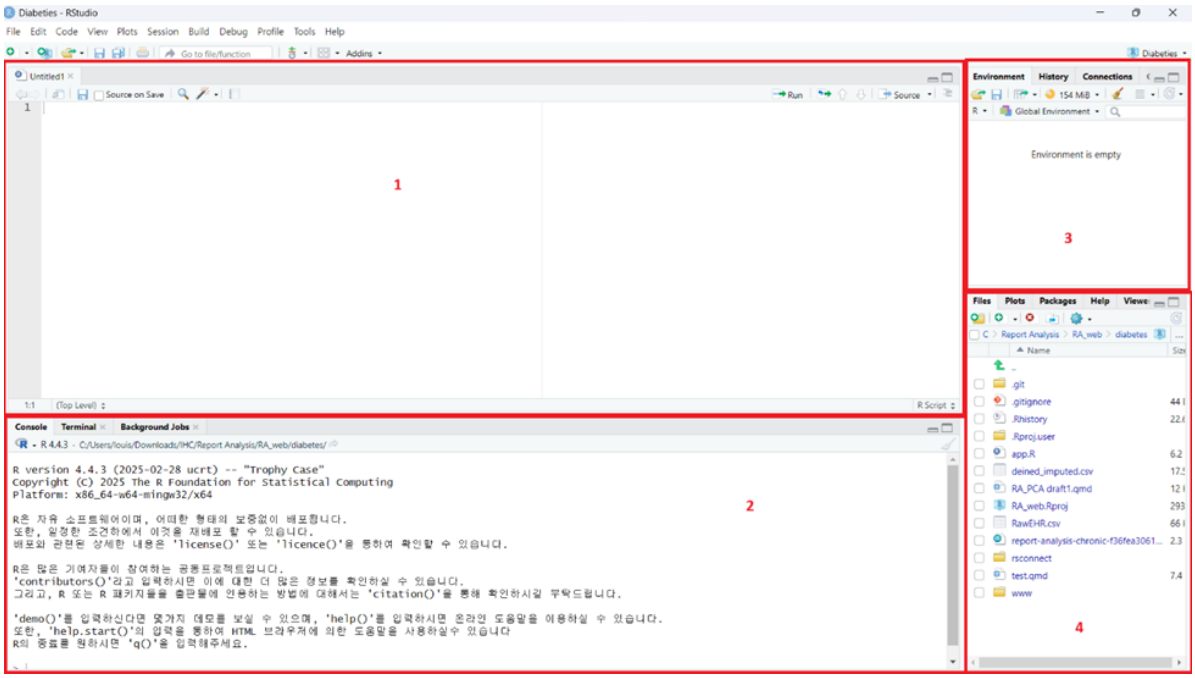
1. Click "File -> New Project -> New Directory -> New Project"
2. Choose location of folder you want to be, a good name, and click "Create Project"



Congrats! You just made your first project in R. Now let's start exploring what features are there in R Studio.

1.2 UI of R Studio

On the main page of R Studio, you'll see 4 main sections.



1 – Script Editor: This is where you write and save your code.

2 – Console: It's a communication window between you and the computer. You can give him commands to execute, and console will show you what it's doing.

3 – Environment: You'll work with countless variables. This section shows what variables you've saved as you run your code.

4 – Directories: Sometimes you need to load data or files for your code. This section shows which files are in your working folder. Default is folder where your project is.

1.3 Using Console

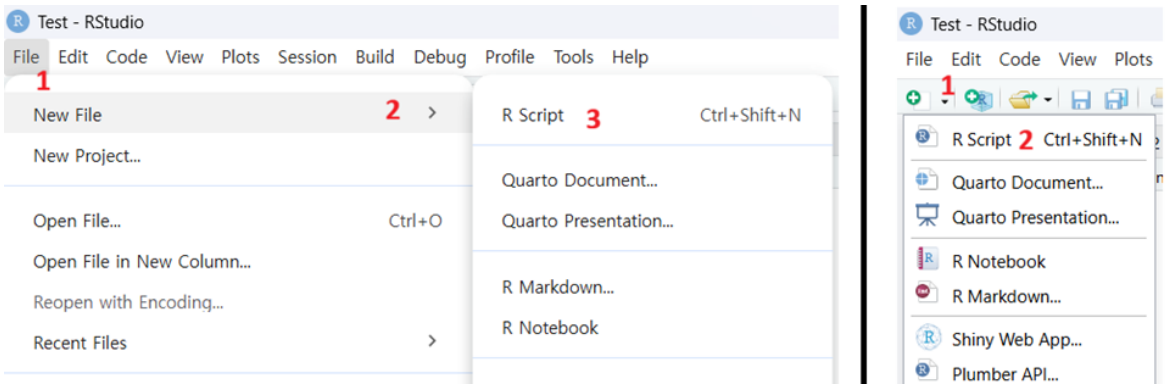
So Console is where actual works are done. Let's try using it. Input 1 + 1 and press enter. You'll see it's printing 2, the result of what you asked R to calculate.

```
> 1 + 1
[1] 2
>
```

However, Console is one-time-communicator. It can't remember anything you say or R executes, making you to retype everything to do same job. This gets annoying fast, especially if your code gets long. How can we make this easy? We'll figure it out in next the next section.

1.4 Creating a script

Script is a file that saves your code so that you don't have to rewrite same thing every time. Let's make a script. There are two ways to create a script.



Script and Quarto Document are most common files used to save the codes. Difference between to are what the default text is.

Type	Description
Script (Talking to computer)	Plain texts are considered as code, and you need to use special characters to insert "comments." It's for purely performing code it's written.
Quarto Document (Talking to human)	Plain texts are considered as text, and you need to insert "Chunks" to tell R Studio 'These are codes I want to run.' You can export Quarto Doc as beautiful report, which Script can't.

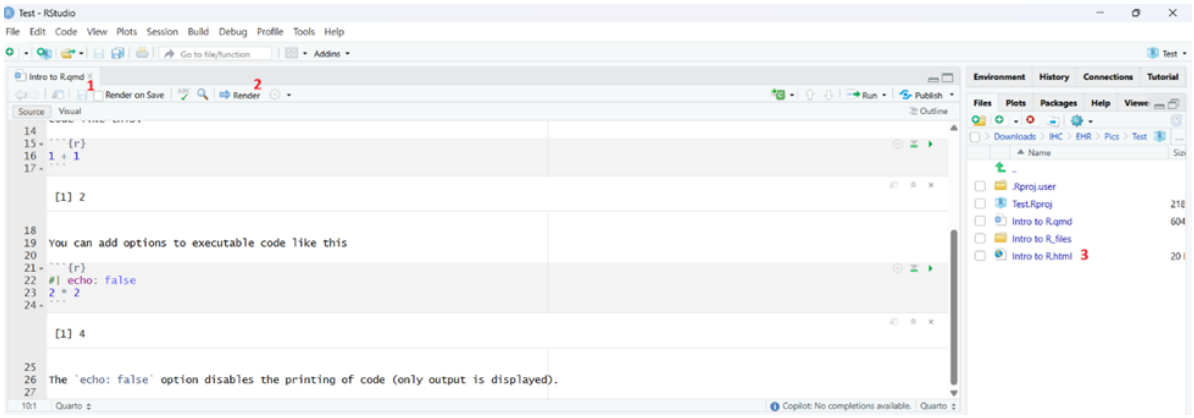
This year, we'll use Quarto Doc as we need to create reports and share codes to others. Create Quarto Doc. You can use whatever title and Author name you want. Leave other options as it is.

1 – # in the doc indicates the title. The more # you put, the smaller the text size gets. # (biggest) -> ####(Smallest)

2 – Area that wrapped with `{r}` and is called "Chunk," where codes you want to run goes. # in the chunks indicates comment, which will not get executed.

3 – When you click Run, you'll see which part of doc you want to execute. If you click specific chunk and "Run Current Chunk," only clicked chunk will be executed. This is one advantage of Quarto doc too.

Now, save your quarto doc and Render. You'll see that (Title).html file is created in your directory (1->2->3 marked in the picture). Congrats! You just created your first report using R. When you open the file, you'll see how quarto doc became visually pretty and clear.



2 Data Types and Variables

2.1 Data Types

Now, let's talk about how data are saved in R. R classifies data into four categories: Number, Text, Boolean, Special.

Type	Description
Number	Any numbers. They're classified into two categories. Integer – Type of number WITHOUT decimals. E.g.) 1, 10, 100 Float – Type of number WITH decimals. E.g) 1.24, 2.00 (Not int because of ".00"!)
Text	Anything you put into " ", regardless of what's inside. E.g) "Hi", "121"
Boolean	A logic data. It's either "True" or "False"
Special	Values that cannot be specified Null – Empty value without determined data type (Number/Text/Boolean) NA – Missing value. Known data type but nothing inside. NaN – Unable to mathematically define. E.g.) Saving "Hi" as int/float inf – Yes, infinite number. They're under special characters!

2.2 Variables

Now we know that there are multiple types of data. But how do we tell R to remember them? That's where variables come in! Variables are names that store data values.

You can declare a variable by using the assignment operator '<-' or '='.

```
Test1 <- 3
Test2 = 4
```

Once you declare the variable, you'll see them in the Environment tab. Also, you'll be able to call them by simply typing their names.

```
Test1
```

```
[1] 3
```

```
Test2
```

```
[1] 4
```

If it's not working, it's probably due to variable names. There're specific rules in naming - First character must be dot(.) or letter. It cannot be number or special character. Second character and later can be anything - You cannot use other function's name as variable name (sum, mean, etc. We'll cover functions as we go on) - Variable names are case-sensitive. "test" and "Test" are different variables. - Avoid using spaces in variable names. Use dot(.) or underscore(_) instead. E.g.) my.variable, my_variable

2.3 Simple Calculations

Here, we'll review basic operators in program language.

Type	Operation
Addition	+
Subtraction	-
Multiplication	*
Remainder	%
Exponents	/

Try some on the script you made! Also, you can use variables in calculations.

```
Test1 + Test2
```

```
[1] 7
```

There are multiple functions that will make calculations easy

Operation	**Description**
ceiling(x)	Round x up as "int" E.g.) 3.4 (float) becomes 3 (int)
floor(x)	Round x down as "int"
trunc(x)	Drop decimals as "float" E.g.) 3.4 (float) becomes 3.0 (float)
round(x, digits = n)	Round x as "float" at nth decimal place. default (If you omit "digits =n") is to round at ones place.
sqrt(x)	$\sqrt{x}$
exp(x)	e^x
log(x, base = n)	$\log_n(x)$. Default is base with n = e
sin(x), cos(x), etc.	Trig functions!
Factorial(n)	n! = 1 x 2 x 3 x … x n
Others	There are many other function!

2.4 Logical Calculations

Sometimes you need more than simple calculations, such as "which number is bigger?" Return values of logical calculations are Boolean (True/False).

Type	Operation
Comparison	<, >, <=, >=
Equal	==
Not Equal	!=
Or	|
And	&

3 Vectors

Last chapter, we reviewed what variables are. Imagine we're storing heights of 50 students. Declaring 50 variables like "Height1", "Height2", ..., "Height50" takes forever. Instead, we can use "vector" to store multiple values in a single variable.

Vector is a 1-dimensional sequence of "same type" of data. For example, vector with int can only store int values. Length of Vector is equal to number of values stored in the vector.

3.1 Vectors with Numbers

3.1.1 Ways to create vectors

```
y <- c(2, 4, 6, 8, 10)
y
```

```
[1]  2  4  6  8 10
```

Again, vector must be same data type. For example, if you create one with mix of int and float, all values will be float. Why? because all int can be float by adding ".0" but float cannot be int without losing data. Let's see an example.

```
x <- c(1,3,5,7,9.2)
x
```

```
[1] 1.0 3.0 5.0 7.0 9.2
```

Recall c() meant to "connect." This means you can connect multiple vectors into one as long as they match data type.

```
z <- c(x, y)
z
```

```
 [1]  1.0  3.0  5.0  7.0  9.2  2.0  4.0  6.0  8.0 10.0
```

Question: What will happen if we flip the order of element? As you can see below, resulting order of elements in the vector also changes.

```
z <- c(y, x)
z
```

```
 [1]  2.0  4.0  6.0  8.0 10.0  1.0  3.0  5.0  7.0  9.2
```

Now we learned how to create vectors with number. But what if we want to label something and need 1 to 100? Typing them manually will take some time. Instead, there's a short cut to create continuous numbers with ":" operator. There there two points to remember when using :.

1. It has fixed increment of 1.
2. It follows the order from the left to the right. Check the example below.

```
1:10
```

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

```
10:1
```

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

```
0.2:3
```

```
[1] 0.2 1.2 2.2
```

What if you want to create vector with different increment? You can use seq() function. It has three main arguments - from, to, by. "from" is where the sequence starts, "to" is where it ends, and "by" is the increment.

```
seq(1, 10)
```

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

```r
seq(1, 10, by=2)
```

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

```r
seq(length=5, from=1, by=2)
```

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

If you want to create vector with repeated values, you can use rep() function. It has two main arguments - "x" and "times". "x" is the value you want to repeat, and "times" is how many times you want to repeat it.

```r
rep(5, times=4)
```

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

3.1.2 Calculations with Vectors

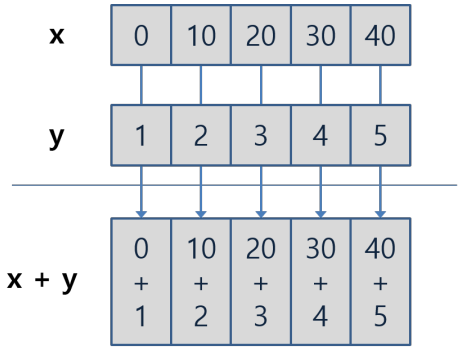
- Element wise operation Just like how we perform calculations with single numbers, we can do the same with vectors. One strong advantage of R is that it's one of rare programming languages that can perform "vectorized" calculations. This means you can perform calculations using same position of different vectors without complex logics. Let's perform example calculation using vectors we created.

```r
x <- seq(0, 40, by = 10)
y <- c(1:5)

x + y
```

```
[1]  1 12 23 34 45
```

```r
x * y
```

```
[1]   0  20  60 120 200
```



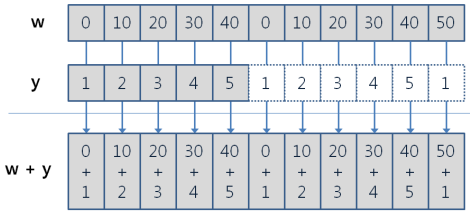
- Recycling rule What if the length of vectors are different? In this case, R will recycle the shorter vector until it matches the length of longer vector. Check the example below.

```
z <- c(rep(x, times=2), 50)
z + y
```

```
Warning in z + y:
```

```
 [1]  1 12 23 34 45  1 12 23 34 45 51
```

As you can see, addition is performed even though length of two vectors didn't match. The warning sign is indicating that length's are not multiple of each other, so it shows what we mean by "recycle until it matches the length." If it's difficult to grasp, there's a visualization below.



3.1.3 Functions helpful for numeric vectors

There are many functions that can be used with numeric vectors. Here are some of the most commonly used ones. While you might not end up not using them all, try them out and see which ones are useful for you!

Operation	Discription
length(x)	Return length of the vector
sum(x)	Return sum of all elements in the vector
mean(x)	Return average of all elements
median(x)	Return median of all elements
var(x)	Return variance of all elements
sd(x)	Return standard deviation of all elements
max(x)	Return highest number in the vector
min(x)	Return lowest number in the vector
range(x)	Return the range (highest and lowest) number
rank(x)	Return the "ranking" of each element from smallest to biggest
sort(x)	Reorganize vector from smallest to biggest elements
order(x)	Return index of smallest to biggest element's location
which.max(x)	Return the highest number's index
which.min(x)	Return the lowest number's index
which(x)	Return index of whatever element that satisfies the condition

rank/sort/order can be confusing, so here's the example.

```
x <- c(11,14,12,13,15)
x
```

```
[1] 11 14 12 13 15
```

```
# Rank returns the ranking of each element in entire vector
rank(x)
```

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

```
# Order returns Where the element is located in the original vector if we sorted it
order(x)
```

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

```
# Sort returns the vector sorted from smallest to biggest
sort(x)
```

```
[1] 11 12 13 14 15
```

3.2 Vectors with Boolean

Just like numeric vectors, you can create vectors with Boolean values (TRUE/FALSE). You can use c() function to create them. You can use T/F instead too.

```
booleanvec <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
booleanvec
```

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

Special point about the Boolean vectors is that you can use logical operators to create vectors. Let's say you want to know which students scored higher than 60 on their final.

```
scores <- c(55, 70, 65, 40, 90)
passed <- scores > 60
passed
```

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

3.2.1 Functions you should know for Boolean vectors

While it's not literally a boolean logic, there are some functions that can be useful when working with Boolean vectors: any/all/if/else

any() statement returns TRUE if at least one element in the vector is TRUE.

```
any(passed)
```

```
[1] TRUE
```

all() statement returns TRUE only if all elements in the vector are TRUE.

```
all(passed)
```

```
[1] FALSE
```

if and else statements are fundemental for all programming languages. They allow you to control the flow of your program based on conditions. However, we'll discuss about it briefly here since other contents are heavy as well. If you want to learn more about it, let me know anytime.

if statement returns true when the condition is met. It's like a gateway before executing next code. "Do if condition is met? Then move to next line of code. If not, then skip to else statement (or end the if statement)."

else statement is executed when the if condition is not met. It's like the alternative path when the if condition fails.

Both if and else statements are wrapped around {}.

Here's the example:

```
Iftest <- c(1,3,5,7,9)

if (any(Iftest > 5))
  {
  print("At least one element is greater than 5")
} else
  {
  print("No elements are greater than 5")
}
```

```
[1] "At least one element is greater than 5"
```

For readability, R allows you to combine ifelse into a single line. It looks like this: ifelse(Condition, Value if TRUE, Value if FALSE)

```
ifelse(any(Iftest > 5), "At least one element is greater than 5", "No elements are greater th
```

```
[1] "At least one element is greater than 5"
```

3.3 Vectors with Text

Just like other two types, this is just vector with a sequence of text values. You can create them using c() function.

```r
names <- c("Banana", "Apple", "Carrot", "Grape", "Whatelse")
names
```

```
[1] "Banana"   "Apple"    "Carrot"   "Grape"    "Whatelse"
```

Unlike numbers, you can't add or subtract texts. So there's other ways to handle manipulation of text vectors.

3.3.1 Functions helpful for text vectors

First, we should check if all elements are text, as vector require everything to be same type. But in huge data set, it might be tough to manually check everything. So we have ".ascharacter()" function to convert everything into text.

```r
mixedvec <- c("Apple", 2, "Banana", 4.5, TRUE)
mixedvec <- as.character(mixedvec)
mixedvec
```

```
[1] "Apple"  "2"      "Banana" "4.5"    "TRUE"
```

Try code by yourself. Did you notice that all inputs are as text when they were originally text, int, float, bool?

Now, let's look at how we can combine and split text vectors like addition/substraction. Both cases, they follow same element wise operation and recycle rule like numeric vectors.

For combining text vectors, we can use paste() function. If you want to add space between combined texts, use sep=" " argument. One point is that you can paste how many vectors you want, but you can call sep =" " one once per line. So if you want to have different seperators between multiple vectors, you'll need to call paste() multiple times. So it'll look like this:

```r
names <- c("Banana", "Apple", "Carrot", "Grape", "Whatelse")
price <- c("1.00", "2.50", "0.75")
combined <- paste(names, price, sep=": $") #Combine name and price with ": $" in between
combined #Do you notice recycling rule here?
```

```
[1] "Banana: $1.00"   "Apple: $2.50"    "Carrot: $0.75"   "Grape: $1.00"
[5] "Whatelse: $2.50"
```

For spliting text vectors, we can use strsplit() function. When you split, you use split=" " argument to specify where to split.

```
splitnames <- strsplit(combined, split=": ")
splitnames #Name and price are seperated using ":" as the split point
```

```
[[1]]
[1] "Banana" "$1.00"

[[2]]
[1] "Apple" "$2.50"

[[3]]
[1] "Carrot" "$0.75"

[[4]]
[1] "Grape" "$1.00"

[[5]]
[1] "Whatelse" "$2.50"
```

There are many other functions that's helpful with handling vectors. However, due to it's amount, we'll mention only few that's commonly used.

Opertiona	Description
nchar(x)	Return character counts of each elements in the vector x. *Literally all characters, including space, are counted!
substr(x, start, stop)	Return character counts of each element in the vector x, starting from 'start'th character to the 'stop'th character.
grep(target, x, ignore.case=T, fixed=T)	Return the index of element in vector x that contains "target". ignore.case asks if you want grep() to be case sensitive (T as yes). fixed asks if you want to use Regex (We don't be discussing about this here. Keep it true)
sub(target, replacement, x, ignore.case=T, fixed=T)	If any element in vector x contains "target," replace it to "replacement."
toupper(x) tolower(x)	change all characters in the vector x into upper/lower case

3.4 Other vectors with special data tpyes

Earlier, we said there's a special data types. During data analysis, handling these becomes essential as we don't know how data that doesn't align with your dataset will influence the general trend (hence analysis result). So let's briefly look at how to create vectors with these special data types, especially Na and NaN.

3.4.1 Vectors with NA

NA is used to represent missing values with undetermined data type. For example, if you're collecting final exam scores but completely missed one student's score, you can use NA to represent it.

How we do process NA without manually checking all data? We can use is.na() and na.omit() functions. is.na() checks if there is NA values, and na.omit() removes all NA values from the vector.

```
scores <- c(90, 85, NA, 70, 95, NA, 80)
is.na(scores)  #It'll return TRUE if there's NA in the vector
```

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

```
scores <- na.omit(scores)  #It'll remove all NA
scores
```

```
[1] 90 85 70 95 80
attr(,"na.action")
[1] 3 6
attr(,"class")
[1] "omit"
```

Look at the output of na.omit(). Second/Fourth line - attr() shows what type of work is by the code. Ignore them First line - Vector after removing NA values Third line - Attribute showing which index values were removed due to NA Fifth line - What's done by the code (removing NA values)

However, using is.na() and na.omit() everytime can be time-consuming in big dataset. How can we "ignore" them without removing them? Many functions in R have "na.rm" argument. Setting na.rm=T will ignore NA values during calculation.

```r
scores <- c(90, 85, NA, 70, 95, NA, 80)
mean(scores, na.rm=T)  #Calculate mean while ignoring NA values
```

```
[1] 84
```

3.4.2 Vectors with NaN

NaN is used to represent mathematically undefined values. For example, if you try to divide 0 by 0, the result is undefined even thought we know that data type is numbers. In this case, R will return NaN. Just like NA, we can use is.nan() function to check if there's NaN values in the vector. To know if vector actually contains NaN, you must use is.nan() because is.na() will return TRUE for both NA and NaN.

3.4.3 Index

I mentioned "index" multiple times before, and you probably grasped somewhat sense of what it is. Index is a location of element in the vector. In R, index starts from 1 (not 0 like other programming languages). You can use index to call specific elements in the vector by indicating interested index in []. Here's the example:

```r
x <- c(10, 20, 30, 40, 50)
x[3]  #Call the 3rd element in the vector x
```

```
[1] 30
```

But one thing to be careful is that you cannot call two indicies at once. Try x[3,5] and see if it works.
Instead, you can use c() to connect multiple indices.

```r
x <- c(10, 20, 30, 40, 50)
x[c(2,4)]  #Call 2nd and 4th elements in the vector x
```

```
[1] 20 40
```

4 Matrix

Vector was a 1-dimensional data, meaning each vector can hold one type. For example, if we create vector of heights, all elements must be height numbers. However, what if we want need to hold details about the data? For example, height by gender?

Matrix is a 2-dimensional data structure, holding rows and columns. Just like vector, matrix must hold one type of data.

4.1 Creating a matrix

There's two ways to declare matrix.

1. Dividing vector into rows and columns with matrix()

```
c <- 1:10

m <- matrix(c, nrow = 2, ncol = 5)
m
```

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

Here, did you notice the order of elements? R fills the matrix by column first, then row. If you want to fill by row first, use byrow = TRUE option.

```
m <- matrix(c, nrow = 2, ncol = 5, byrow = TRUE)
m
```

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

2. Combining vectors with rbind() and cbind() rbind() combines vectors by row, while cbind() combines by column. This means that rbind() will stack vectors on top of each other, while cbind() will place them side by side.

```
v1 <- c(1, 2, 3)
v2 <- c(4, 5, 6)
m_row <- rbind(v1, v2)
m_row
```

```
   [,1] [,2] [,3]
v1    1    2    3
v2    4    5    6
```

```
m_col <- cbind(v1, v2)
m_col
```

```
     v1 v2
[1,]  1  4
[2,]  2  5
[3,]  3  6
```

One thing to consider is the recycling rule. Whenever you're working with data sets, having different dimensions will cause shorter data set to repeat.

```
v1 <- c(1,2,3)
v3 <- c(7, 8)
m_row2 <- rbind(v1, v3)
```

```
Warning in rbind(v1, v3): number of columns of result is not a multiple of
vector length (arg 2)
```

```
m_row2
```

```
   [,1] [,2] [,3]
v1    1    2    3
v3    7    8    7
```

With a same logic, you can create matrix by combining vector with matrix or matrix with matrix using bind() functions.

```r
c <- 1:10
rbind(c, 11:20)
```

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

4.2 Index of matrix

Just like vector, you can access specific elements of matrix using index. However, since matrix has two dimensions, you need to specify both row and column index.

When you index, you use matrix[row, column] format.Emptying either means "all of them," and you can also use range like how we created 1,2,3,4 with 1:4.

```r
test <- matrix(1:25, nrow=5, ncol=5)
test
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

```r
test[2, 3]  # Access element at 2nd row, 3rd column
```

```
[1] 12
```

```r
test[ , 4]  # Access all rows in 4th column
```

```
[1] 16 17 18 19 20
```

```r
test[5, 1:3]  # Access 1st-3rd column in 5th row
```

```
[1]  5 10 15
```

There is special about matrix in R. If you use minus, like [, -2], it means "Don't display (whatever you decide)". This means that you can manipulate matrix! Look at example below. Do you notice that 6-10 are missing?

```
test[ , -2]  # Access all rows except 2nd column
```

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

```
test <- test[ , -2]  # By saving "test matrix without 2nd column", you can remove 2nd column
```

Like how matrix fills in order (like column first, then row), when you remove rows or columns, you can manipulate matrix beyond simply removing specific rows or columns. Look at example below. For example, 1:5 mean "1st to 5th row." Then what will happen if we do 5:1?

```
test[5:1, ]
```

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

Because we're displaying test matrix from 5th row to 1st row, the order of rows get reversed! Pause and Think: what would test[order(a[, 2]),] do?

Like how we filtered out the matrix, we can manipulate them using logical operations. In this case, it only displays elements that's TRUE. Check example below.

```
people <- c("John", "Jane", "Jim", "Jill", "Jack")
Age <- c(28, 34, 29, 42, 23)
Height <- c(175, 160, 180, 165, 170)

Information <- cbind(people, Age, Height) # Imagine what output it would be!
Information[ , c(T,F,T)]  # Display only 1st and 3rd column (those with TRUE)
```

```
     people Height
[1,] "John" "175"
[2,] "Jane" "160"
[3,] "Jim"  "180"
[4,] "Jill" "165"
[5,] "Jack" "170"
```

```r
Information[Information[ ,2] > 30, ]  # Display only rows where Age (2nd column) is greater t
```

```
     people Age  Height
[1,] "Jane" "34" "160"
[2,] "Jill" "42" "165"
```

However, there is issue with the structure of matrix when we combine multiple vectors. Let's say you want to investigate average bloodpressure of a patient

```r
# Data with visiting date, bloodpressure, and patient id
record <- matrix(c("2023-01-01", 120, 123,
                   "2023-01-02", 130, 124,
                   "2023-01-01", 125, 456,
                   "2023-01-02", 135, 456), nrow=4, ncol=3, byrow=TRUE)
record
```

```
     [,1]         [,2]  [,3]
[1,] "2023-01-01" "120" "123"
[2,] "2023-01-02" "130" "124"
[3,] "2023-01-01" "125" "456"
[4,] "2023-01-02" "135" "456"
```

```r
Avg_BP <- mean(as.numeric(record[, 2]))  # Think, why do we need as.numeric() ?
Avg_BP
```

```
[1] 127.5
```

But you realized that you forgot the heights of patients, and you create new matrix.

```r
# Data with visiting date, height, bloodpressure, and patient id
record_fix <- matrix(c("2023-01-01", 150, 120, 123,
                       "2023-01-02", 150, 130, 124,
                       "2023-01-01", 150, 125, 456,
                       "2023-01-02", 150, 135, 456), nrow=4, ncol=4, byrow=TRUE)
Avg_BP <- mean(as.numeric(record_fix[, 2]))  # Run the same code again
Avg_BP
```

```
[1] 150
```

After fixing the matrix, running same code that gave us average blood pressure is giving us wrong result! This is because adding new column shifted location of existing information. Then how can we track which information is where? This is where rownames() and colnames() takes place to assign titles to rows and columns.

```r
colnames(record_fix) <- c("Date", "Height", "BloodPressure", "PatientID")
record_fix
```

```
     Date         Height BloodPressure PatientID
[1,] "2023-01-01" "150"  "120"         "123"
[2,] "2023-01-02" "150"  "130"         "124"
[3,] "2023-01-01" "150"  "125"         "456"
[4,] "2023-01-02" "150"  "135"         "456"
```

Now, we can access information by their names, not by their index. This way, even if we add new information, we can still access existing information without worrying about shifting index.

```r
Avg_BP <- mean(as.numeric(record_fix[ , "BloodPressure"]))
Avg_BP
```

```
[1] 127.5
```

Great fix! By the way, review old matrices like "Information" or m_row, and compare to "record." Do you notice two have colnames while one doesn't? This is where creating matrix by combining vectors (rbind, cbind) is better than creating matrix by dividing single vector (how we made record). When you use rbind() and cbind(), they recognize names of vectors, thinking "these values belong under the name of the vector!" As result, functions automatically assign colnames() or rownames() using vector names.

One last thing, matrix is 2x2. What if you only give one index like how we did in vector indexing? In matrix, inputing one index means matrix will behave like a vector. So, like how matrix fills by column first and then rows, matrix[i] will return index after scanning column and row order.

```
last <- matrix(1:9, nrow=3, ncol=3)
last
```

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

```
last[5] # Since it's 3x3, 5th index will be [2,2]. 1st column (3 elements), and 2nd column's
```

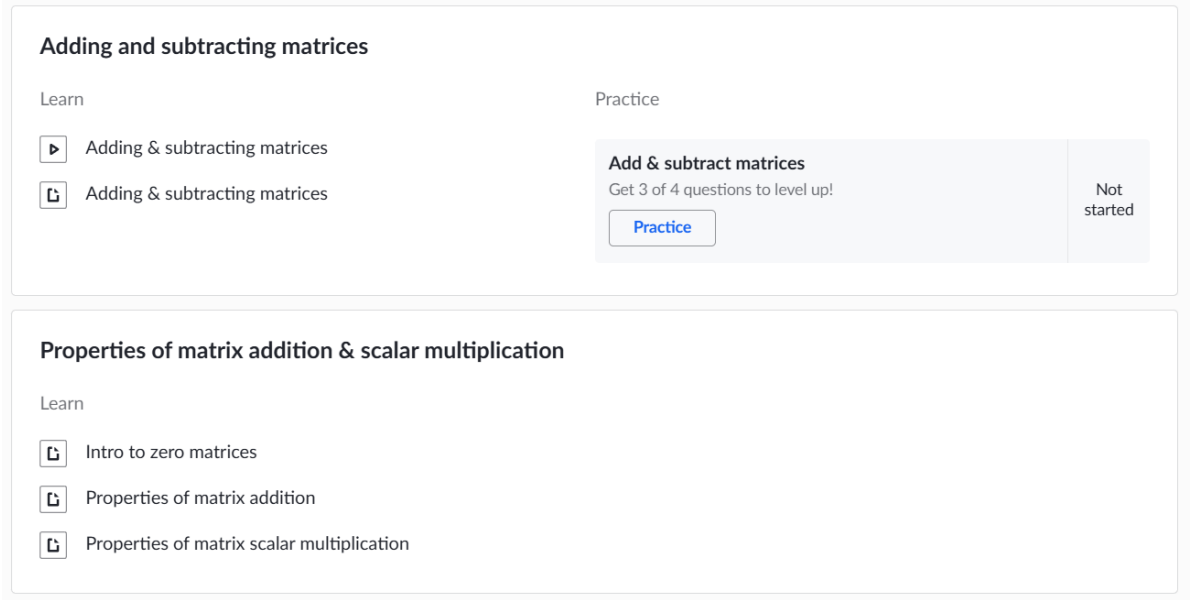
```
[1] 5
```

```
last[last > 5]  # Returns all elements greater than 5 as 1-dimensional vector
```

```
[1] 6 7 8 9
```

4.3 Calculation of matrix

Before we get into calculation of matrix, you should know the math behind matrix calculation. Please review concepts regarding matrix, adding/subtracting/multiplying matrices. Others are good to know but optional. Link is **here**



Now let's try calculating matrix in R. First, creating two matrices with same dimension.

```
A <- matrix(1:9, nrow=3, ncol=3)
B <- matrix(9:1, nrow=3, ncol=3)
A
```

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

```
B
```

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

```
A + B
```

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

```
B - A
```

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

```
A * B
```

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

```
A / B
```

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

Great! R is doing element-wise calculation. This means that R is adding/subtracting/multiplying/dividing each element in same position. For example, A[1,1] + B[1,1], A[1,2] + B[1,2], and so on.

One question, do matrix follow recycling rule? Try making two matricies with different dimension and adding them. You'll see the error - R cannot perform calculations that violates fundemental rules of math. However, when you're using vector and matrix, vectors can be recycled.

```
C <- c(0, 10, 100)

A + C
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]   12   15   18
[3,]  103  106  109
```

However, we should keep in mind that vector should be shorter than the number of elements in the matrix. If vector is longer, what are we adding leftover vector elements with?

4.4 Functions regarding matrix

###nrow(), ncol(), dim()

How do we know size of unknown matrix? nrow() and ncol() returns number of rows and columns, while dim() returns both as vector. dim() returns column size first! Don't get confused with [row, column] in indexing.

```
m <- matrix(1:12, nrow=3, ncol=4)
nrow(m)
```

```
[1] 3
```

```
ncol(m)
```

```
[1] 4
```

```
dim(m)
```

```
[1] 3 4
```

4.4.1 rownames(), colnames()

Earlier, we saw how we can assign row names and column names by rownames() <- c(names). However, can we do the opposite? Like, checking names of cols and rows instead of assigning? Yes! Instead of assigning through <-, you can use rownames(matrix) or colnames(matrix) to check names of rows and columns.

```
colnames(m)
```

```
NULL
```

```
rownames(m)
```

```
NULL
```

Here, we get Null (empty). Why is that? Look at how m looks like and think about it!

4.4.2 t()

Transpose matrix is matrix with switched rows and columns (for example, 2x3 becomes 3x2). In R, you can do this by t() function.

```
m
```

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

```
t(m)
```

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

###Moving between matrix and vector

Sometimes, you may want to convert matrix into vector or vice versa. You can do this by as.vector() or c(). as.vector() unstrings matrix into single line (= vector). And do you remember how giving one input feature in indexing make matrix behave like vector? c() function does the same thing. If you do c(matrix) without telling what else to combine with, c() will combine matrix itself into one string, making it vector.

```
Testmat <- matrix(1:9, nrow=3, ncol=3)
as.vector(Testmat)
```

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

```
c(Testmat)
```

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

4.4.3 Apply()

Often, we need to perform calculation throughout row or column of the matrix (e.g. finding average height by each person, finding total score by each student). Instead of writing all elements individually, we can use apply() function.

apply() has three main arguments (inputs). X (target), Margin (which direction do you want to apply the function? 1 is row, 2 is column), and Function (work you want to perform)

```
Scores <- matrix(c(90, 85, 88,
                   78, 92, 80,
                   85, 87, 90), nrow=3, byrow=TRUE)
colnames(Scores) <- c("Math", "Science", "English")
Scores
```

```
     Math Science English
[1,]   90      85      88
[2,]   78      92      80
[3,]   85      87      90
```

```
#Let's find average score of each subjects
math_avg <- (Scores[1,1] + Scores[2,1] + Scores[3,1]) / 3
math_avg
```

```
    Math
84.33333
```

```
#Doing this for each column is annoying. Let's use apply()
exam_avg <- apply(Scores, 2, mean) # Find mean (function) of matrix (Scores) by column (2)
exam_avg
```

```
    Math  Science  English
84.33333 88.00000 86.00000
```

5 Array

Vector is 1-dimensional, matrix is 2-dimensional. What if we want to hold more than 2 dimensions? For example, what if we want to hold height and weight by gender? This is where array comes in. Array is multi-dimensional data structure, meaning it can hold 3 or more dimensions. However, like vector and matrix, array can only hold one type of data.

5.1 Creating an array

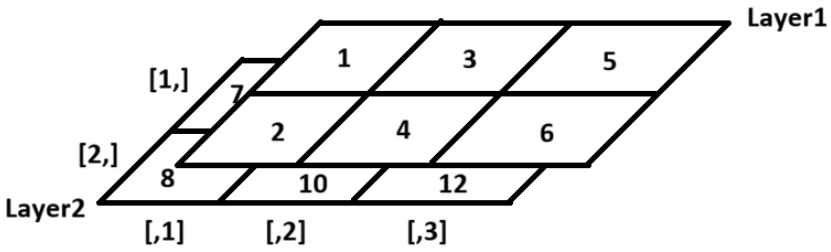
We can declare by matrix in a similar way to how we declared vectors. 1. Define matricies you want to stack 2. Use array() function to stack them * Remember, dimension should be identical to be able to stack them.

```
mat1 <- matrix(1:6, nrow=2, ncol=3)
mat2 <- matrix(7:12, nrow=2, ncol=3)
arr <- array(c(mat1, mat2), dim = c(2, 3, 2)) # 2 rows, 3 columns, 2 layers
arr
```

```
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Unlike how matrix or vectors were single layer, you'll see that 3D array has layers, shown as ", , 1" and ", , 2". Arrays fill data just like matrix. Column first, then rows (next columns), then layers. In the same way, you can index them. Array[row, column, layer]. If you expand it to 4,5,6th dimensions? You just keep adding commas! For example, array[row, column, layer, 4th dimension, 5th dimension]

It is difficult to imagine it, so I'll visualize 3D array we made (arr) below.



Just like colnames and rownames in matrix, you can assign names to each dimension using dimnames() function.

```
dimnames(arr) <- list(
  Row = c("R1", "R2"),
  Column = c("C1", "C2", "C3"),
  Layer = c("L1", "L2")
)
arr
```

```
, , Layer = L1

    Column
Row  C1 C2 C3
  R1  1  3  5
  R2  2  4  6

, , Layer = L2

    Column
Row  C1 C2 C3
  R1  7  9 11
  R2  8 10 12
```

Now, you can index using names too!

```
arr["R1", "C2", "L1"]
```

```
[1] 3
```

However, there is one thing that you should be careful. Let's look at the example below

```
arr[1, 2:3, 1:2] # Access 1st row, 2nd-3rd column, all layers
```

```
      Layer
Column L1 L2
    C2  3  9
    C3  5 11
```

We asked them to display first row, 2nd-3rd column, all layers: 3D data. However, output is 2x2 matrix, a 2D data. Like this, when R can display result in lower dimension, it will do it. If you want to keep the original dimension, use drop = False argument. If you look at the example below, you'll see that output is seperated into two layers again.

```
arr[1, 2:3, 1:2, drop = FALSE]
```

```
, , Layer = L1

    Column
Row  C2 C3
  R1  3  5

, , Layer = L2

    Column
Row  C2 C3
  R1  9 11
```

5.2 Calculation of Array

Omitted. Everything is same as Matrix calculation.

5.3 Advanced caluclation

5.3.1 Solving equations with matrix

Matrix and Array can be used in a lot of places. Due to its compact representation of high dimnesional data, we often use matrix to represent equations with multiple dimension. For example,

$3x + 2y = 5$ $4x + 6y = 12$

can be written as [3, 2, 5]; [4, 6,12] (; is used to seperate rows). So, we condensed multiple equations into one 2x3 matrix. This also means that we can use matrix to solve complex equations.

Using the matrix above as example, we have two parts: variables (x and y), and constants (5 and 12). We can separate them into two matrices:

```
A <- matrix(c(3, 2,
              4, 6), nrow=2, byrow=TRUE) # Coefficient matrix
B <- matrix(c(5, 12), nrow=2) # Constant matrix
colnames(A) <- c("x", "y")
```

Now, you can just use solve() to find values of x and y!

```
solution <- solve(A, B)
solution
```

```
  [,1]
x  0.6
y  1.6
```

5.3.2 Eigenvalues and Eigenvectors

This is core concept of linear algebra, which will help you understand more advanced machine learning algorithms. However, it's difficult to understand without proper math background, so we'll come back to this later. For now, watch this playlist to get introduction to linear algebra. I highly recommend this.

Link: Linear Algebra by 3Blue1Brown

6 Summary of Vector, Matrix, and Array.

Throughout Chapter 3 and 4, we reviewed basic concepts of vector (1D), matrix (2D), and array (Multiple D). While we defined three different data structure, are they really different? Think about this. We declared matrix by combining vectors into two dimensions. Then, we declared 3D array by stacking matrices. However, way to higher dimension is just repeating same process: adding a new axis by combining existing structure. So, doesn't that mean matrix is vector and array is also vector in the end? You're right, it is.

So array is often seen as 'generalized vector' or 'generalized matrix.' While vector is tied to 1D and matrix is tied to 2D, array can build as much as dimensions you want using vector. Next time, we'll explore the list (like a shopping list or to-do list). You'll be able to verify using code how vector to array are all same thing. Then, keep this in your mind and see you in the next chapter!

7 List

7.1 Object and Class

So far we looked at vector, matrix, and array. However, one limitation was that all elements in them need to be same data type. Revisit "Information" matrix where we stored patient's visit date, blood pressure, and patient ID. All elements were character type ("text") despite of bp/ID being number because "Date" was in text format.

But why is that really the case? Why can we declare vectors with text, numbers, or logic, but not two at the same time?

To answer this, we need to understand what object and type are.

7.1.1 What is Object?

Single component we decide to store was called variable. Then how about some kind of produces like vectors? Single unit of whatever data that R can recall, like vectors, matrix, and functions, are called "Objects."

In stored data's perspective, Objects are divded into two types. "Atomic" and "Generic (or List)" List, we'll talk about it later.

Atomic object is what we've been reviewing: A vector, data we can store with single data type. Some examples are vectored made out of logical, integer, character, float (double), complex, etc.

7.1.1.1 Characteristics of Object

All object has two intrinsic characteristics: Type and Length.

Type is literally type of data we discussed in Chapter 2. Nothing more to say. Throughout previous chapters, we talked about how to declare the vectors. However, we never talked about how to check data type of random vector. We use function called typeof() for that.

```
a <- c(T, F, T, T); typeof(a)
```

```
[1] "logical"
```

```
b <- c(1.2, 0.3, 3.5, 2.3); typeof(b)
```

```
[1] "double"
```

Length is another characteristic of object. It tells how many elements are in the object. For example, vector with 4 elements will have length of 4. We can check length of object with length() function.

```
length(a)
```

```
[1] 4
```

Intrinsic characteristics are intrinsic because they can't be changed once object is created. For example, if we create a vector with character type, we can't change it to numeric type unless we manually code it to be so.

But Question here:

```
a <- c(1, 2, 3, 4)
b <- matrix(c(1:4), nrow = 2, ncol = 2)
a
```

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

```
b
```

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

Consider two following data. Both have same length (4), and same data tpye (int). There intrinsic characteristics are same, but they look different (one is 1D vector, another is 2D matrix). How do R distinguish different objects with identical intrinsic characteristics?

7.1.2 Class

Class solves the question. Class is an "atrribute" that you can assign to the object to tell R how to treat the object.

For example, vector "a" and matrix "b" both have same length (4) and same data type (int). However, R treats them differently because their class is different. Vector "a" is stored with class "numeric" while matrix "b" has class "matrix." We can check class of objects by class().

```
class(a) # This was vector
```

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

```
class(b) # This was matrix
```

```
[1] "matrix" "array"
```

7.1.3 Attributes

Another question is, "Okay, now we know that it's the class that differentiate vector and matrix. But, how do R know the matrix is 2x2?, not 4x1?"

When we declare objects, R saves all the accessory information (people call it meta data. It's anything besides intrinsic characteristics, like class) as "Attributes." You can check all attributes using attributes() function. Let's try it.

```
attributes(a)
```

```
NULL
```

```
attributes(b)
```

```
$dim
[1] 2 2
```

As you can see, vector "a" has no attributes, meaning that it's just a simple 1D structure. However, matrix "b" has dimension of 2,2 which means 2x2. This dimension attribute ($dim) tells R that this object is 2D structure with 2 rows and 2 columns.

7.1.3.1 Editing attributes

Earlier, we said that intrinsic characteristics can't be changed once object is created. However, attributes can be changed after creation. Let's try changing dimension attribute of vector "a" to make it 2x2 matrix. We use attr() function to edit them.

```
a
```

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

```
attr(a, "dim")
```

```
NULL
```

```
# Like rownames(), function(target, detail) displays detail of the target. Here, we're askin

attr(a, "dim") <- c(2, 2) # Like assigning rownames, now "<-" assigns (2,2) to dim attribute
a # Now a is 2x2 matrix!
```

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

```
class(a) # And class of a is now matrix!
```

```
[1] "matrix" "array"
```

There are many attributes, such as dimnames() (name of each dimension), names() (name of each element in 1D structure), etc. You can check all attributes using attributes() function. You can look them up and play with it!

7.2 Summary & List

So what was the answer to the question at the beginning of "Object and Class," what was the point of whole section about them?

Consider that intrinsic characteristics are saved and doesn't change when objects are declared. This also means that different type of data are saved and recalled through different pathway. For example, integer vector are stored where only integer exists. In the same way, method

that vector is loaded for calculation is also something only integer can understand. Therefore, vector/matrix/array can hold only single data type.

However, sometimes we need to save multiple data type like patient information vector (visiting date in character, vital signs in numeric). This is where "Generic (because it's not atomic)" object, or "List" comes in. List is special object that can hold multiple data type at the same time. Each element in list can be different data type, and even different structure (vector, matrix, array, another list, etc)!

Each element in the list is called "Component."

7.3 Creating a List

Just like atomic objects, list are declared through list() function in two ways: unnamed list and named list.

- Unnamed list: list(Compoenent1, Component2, Component3))
- Named list: list(Name1 = Component1, Name2 = Component2, Name3 = Component3)

Let's try an example.

```
my_list <- list(Name = c("Patient1", "Patient2", "Patient3"), #Component 1
                Age =  c(30, 45, 28), #Component 2
                Diabetic = c(F, T, F), #Component 3
                Visit.Date = matrix(c("2023-12-1",
                                      "2024-2-16",
                                      "2025-3-20"),
                                      nrow = 3, byrow = TRUE) #Component 4
)

print(my_list)
```

```
$Name
[1] "Patient1" "Patient2" "Patient3"

$Age
[1] 30 45 28

$Diabetic
[1] FALSE  TRUE FALSE

$Visit.Date
```

```
     [,1]
[1,] "2023-12-1"
[2,] "2024-2-16"
[3,] "2025-3-20"
```

Now you see all Character, Numeric, Logical, and even Matrix are stored in single object. Here, we have each components named like "Name," but it will show up as [[1]], [[2]], etc., if they're unnamed.

One important thing to remember while creating a list is that component's name isn't automatically assigned like how creating vector using cbind() named columns whether you name them or not.

```
var1 = c(3,5)
var2 = c(7,9)

cbind(var1, var2) # Columns are named var1 and var2 automatically
```

```
     var1 var2
[1,]    3    7
[2,]    5    9
```

```
list(var1, var2)
```

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

[[2]]
[1] 7 9
```

So like the my_list, you must manually assign names to list, like variable1 = var1.

7.4 Indexing List

Unlike vectors, List use different notation for index: listname[index]. Check the exmaple below:

```
class(my_list[1])
```

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

```
class(my_list[[1]])
```

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

One is list, and another is character. What's the difference? To remember the difference, think of level of information you need to call.

- Single bracket [ ]: Index of one level deeper (like calling component in vector X). So In list, [] will call entire "Component" itself. For example, my_list[i] will print entire i-th component. However, this is still a list object because list had two factors: Name and value.

```
my_list[c(1, 2)] # Calls entire first and second component
```

```
$Name
[1] "Patient1" "Patient2" "Patient3"

$Age
[1] 30 45 28
```

- Double bracket [[ ]]: Index of two level deeper. So In list, [[]] will call specific "element" inside the component. For example, my_list[[i]] will return contents in i-th component. This time, this is vector since it only prints out values without name. Check the example, can you spot the difference?

```
my_list[[1]] # Calls elements inside first component (Patient1, Patient2, Patient3)
```

```
[1] "Patient1" "Patient2" "Patient3"
```

```
my_list[1]
```

```
$Name
[1] "Patient1" "Patient2" "Patient3"
```

Important thing you must remember is that [] recalls component as list (list with 1 component), while [[]] recalls component as vector. So, for example, my_list[1][2] won't work since my_list[1] has only first element (2nd element of my_list[1] is non-existent). However, my_list[[1]][2] will work since my_list[[1]] is vector with 3 elements.

```
my_list[1][2] # As you can see, it prints out NULL.
```

```
$<NA>
NULL
```

```
my_list[[1]][2] # Calls 2nd element in first component (Patient2
```

```
[1] "Patient2"
```

Like vectors, you can also use name to call specific component/element. A lot of times, people use its alternatives, $, to call component by name. It's easier to type. There output is same!

```
my_list[["Age"]]
```

```
[1] 30 45 28
```

```
my_list$Age
```

```
[1] 30 45 28
```

Also, like vectors, you can recall elements in component by adding another index.

```
my_list$Name[2:3] # Calls 2nd and 3rd element in "Name" component
```

```
[1] "Patient2" "Patient3"
```

One special point about indexing an list in R is that you can call component by portion of name as long as it's unique. Let's try it.

```
my_list$Visit
```

```
     [,1]
[1,] "2023-12-1"
[2,] "2024-2-16"
[3,] "2025-3-20"
```

Even though name of that component was "Visit.Date," R was able to recall it because no other component started with "Visit." However, be careful when you have multiple components with similar names.

7.5 Editing List

Let's say you created a list. However, what if you need to change information to update record? Let's look at how we can do it.

7.5.1 Adding Component

You can add component to list by simply adding a new component to the list.

```
my_list$Height
```

```
NULL
```

Right now, my_lit doesn't have "height component" that it returns NULL. Let's add it.

```
my_list$Height <- c(170, 180, 160) # Adding Height component
my_list
```

```
$Name
[1] "Patient1" "Patient2" "Patient3"

$Age
[1] 30 45 28

$Diabetic
[1] FALSE  TRUE FALSE

$Visit.Date
     [,1]
[1,] "2023-12-1"
[2,] "2024-2-16"
[3,] "2025-3-20"

$Height
[1] 170 180 160
```

Wow! By adding Height component, now we see that list is extended.However, think about this: what will happen if we add componenet to way far back of the last element?

```
my_list[[7]] <- c("A+", "B", "O-") # Adding Blood Type component at 6th index. Our Last compo
my_list[5:7]
```

```
$Height
[1] 170 180 160

[[2]]
NULL

[[3]]
[1] "A+" "B"  "O-"
```

As you can see, we skipped 6th component slot and added bloodtype to 7th. R will automatically fill skipped componenet in the list with NULL.

7.5.2 Editing Component

In the same way with adding the component, we can edit elements within the component by overwriting existing component (assigning new content to desired index).

```
my_list$Height <- c(4,5,6) # Changing Height into 4, 5, 6
my_list$Height
```

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

```
my_list[[5]] <- c(150, 160, 170) # You can also edit using number index
my_list$Height
```

```
[1] 150 160 170
```

Do you remember how we could recall partial list (component as list) using []? In the same way, you can edit multiple components of list using []. However, remember that assignment (new edits) must be list, since list[] acts as a list, not vector.

```
my_list[1:2]
```

```
$Name
[1] "Patient1" "Patient2" "Patient3"

$Age
[1] 30 45 28
```

```
my_list[1:2] <- list(Name = c("NewPatient1", "NewPatient2", "NewPatient3"),
                             Age = c(35, 50, 40)) # Editing Name and Age component at once
my_list[1:2]
```

```
$Name
[1] "NewPatient1" "NewPatient2" "NewPatient3"

$Age
[1] 35 50 40
```

Now, name and age of patients are changed at once!

One thing to be careful is that recycling rule applies in the list too. So if I try to add a component with different length, R will recycle the values.

```
my_list[9:11] <- list(c("Text"), 12) # Adding Weight component with length
```

```
Warning in my_list[9:11] <- list(c("Text"), 12): number of items to replace is
not a multiple of replacement length
```

```
my_list[9:11] # Now, you'll see "Text" twice because of recycling
```

```
[[1]]
[1] "Text"

[[2]]
[1] 12

[[3]]
[1] "Text"
```

7.5.3 Removing Component

Recall how skipping component resulted in Null, empty. We can use this to remove component from the list! Simpy assign Null to the component you want to remove. It's like editing a list, but with empty component.

```
my_list[8:11] <- NULL # Removing my_list[9:11] we just added. But think, why [8:11], not[9:1
my_list[[7]] <- NULL # Removing blood type component
```

Now, try calling blood type by my_list[[7]]. It will return error since it doesn't exist anymore.

7.5.4 Vector to list, list to Vector

Recall how we could change data type among vectors, like number to character? In the same way, we can convert vector to list, and list to vector.

```
Test_Vector <- c(10, 20, 30, 40)
Test_List <- as.list(Test_Vector) # Converting vector to list
Test_List # List to vector will be as.vector()
```

```
[[1]]
[1] 10

[[2]]
[1] 20

[[3]]
[1] 30

[[4]]
[1] 40
```

Also, we can connect different lists using c() like vectors.

```
Another_List <- list("Text1", "Text2")
Combined_List <- c(Test_List, Another_List)
Combined_List
```

```
[[1]]
[1] 10

[[2]]
[1] 20

[[3]]
[1] 30

[[4]]
[1] 40

[[5]]
[1] "Text1"

[[6]]
[1] "Text2"
```

One thing to be careful is the hierarchy of list. When you use c(), you're putting two vectors next to each other (equally). This results a list with single layer (component1, componenet2, etc, like example above). However, if you create a list by combining other lists, combined list will be compoenent of new list, creating multiple layers.

```
Combined_List2 <- list(Sublist1 = Test_List, Sublist2 = Another_List)
Combined_List2
```

```
$Sublist1
$Sublist1[[1]]
[1] 10

$Sublist1[[2]]
[1] 20

$Sublist1[[3]]
[1] 30

$Sublist1[[4]]
[1] 40


$Sublist2
$Sublist2[[1]]
```

```
[1] "Text1"

$Sublist2[[2]]
[1] "Text2"
```

Now, you'll see that both Test_list and Another_List became components under name of Sublist1, and Sublist2, respectively (2 layered list). Then, could we transform multi-layered list into vector? Yes, we can use unlist() function for that.

```
unlist(Combined_List2)
```

```
Sublist11 Sublist12 Sublist13 Sublist14 Sublist21 Sublist22
     "10"      "20"      "30"      "40"   "Text1"   "Text2"
```

7.6 Functions with List

Many functions we learned can be applied to list. However, apply() doesn't work on list because list is a generic object with different components, while apply() requires same data type as it perform same work throughout the object.

7.6.1 lapply()

lapply works by apply(list, function). Let's look at the example.

```
Sample1 <- list(A = c(1,2,3), B = c(4,5,6), C = c(7,8,9))
Sample1
```

```
$A
[1] 1 2 3

$B
[1] 4 5 6

$C
[1] 7 8 9
```

If we want to find mean of each component, using mean() requires us to calculate through three lines of code (mean() per component). Using, lapply does the same in single line.

```r
lapply(Sample1, mean)
```

```
$A
[1] 2

$B
[1] 5

$C
[1] 8
```

Here, remember that lapply() applies function to each componenet and return the result in identical structure to the input (so list in this case).

7.6.2 sapply()

Many times, we want the calculation to return in simplified structure like vector so that we don't have to index list every time (which is more complex and time consuming). sapply() does the same thing with lapply(), but return the result in vector or matrix.

```r
sapply(Sample1, mean)
```

```
A B C
2 5 8
```

```r
sapply(Sample1, range)
```

```
     A B C
[1,] 1 4 7
[2,] 3 6 9
```

Now, instead of multi components from lapply ($A, $B, $C), we have single vector with three elements. This is easier to use in many cases. If result requires multiple rows/columns, like range(), output will be matrix.

7.6.3 mapply()

So lapply() and sapply() apply function to single list. But what if we have multiple lists and want to apply function to them at once? mapply() is the answer. mapply() works by mapply(function, list1, list2, ...). Let's look at example.

```
mList <- list(A = c(1,2,3), B = c(4,5,6), C = c(7,8,9))
mList2 <- list(D = c(10, 11, 12, 13), E = c(14, 15), F = c(16, 17, 18))

mapply(c, mList, mList2) # Perform c() of each component in mList and mList2
```

```
$A
[1]  1  2  3 10 11 12 13

$B
[1]  4  5  6 14 15

$C
[1]  7  8  9 16 17 18
```

As you can see, each component in mList and mList2 are combined using c(). Two points to notice here: 1. mapply() is order sensitive. If you put mList2 first, mList2 will be at the front of the result. 2. When combining components with different names, names of first list are prioritized (Here, six components are combined into three components. Since A, B, C are from first list, they're kept as names). 3. mapply() performs dimension reduction

What do we mean by #3? Look at the output below:

```
mList3 <- list(G = c(10, 11, 12), H = c(13, 14, 15), I = c(16, 17, 18))
mapply(c, mList, mList3)
```

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

Here, we have a single matrix instead of list with three components. Why is this? Review the length of each component.

- mList and mList2: Since mList and mList2's component pairs (A/D, B/E, C/F) had different lengths, mapply() is required to print output as seperate components: a list.
- mList and mList3: However, in mList and mList3, all componenets (A/G, B/H, C/I) had same length (3). Therefore, mapply() was able to combine them into single matrix instead of list with three components, reducing the dimension.

Like this, mapply() will try to be useful and print most simplified structure possible.

7.7 Closing

List, which combines multiple data types into single object, is absolute must-know concept in R as many real world data mixes data types (like title and values). Also, this concept of list will be foundation of the highlight of data structure, a dataframe (we'll cover this in next chapter). Then, see you in next chapter!