

R Guide

Jaewon Kim

2025-12-13

Table of contents

Welcome to “Introduction to R”	4
1 Structure of R Studio	5
1.1 Creating a Project	5
1.2 UI of R Studio	6
1.3 Using Console	6
1.4 Creating a script	7
2 Data Types and Variables	9
2.1 Data Types	9
2.2 Variables	9
2.3 Simple Calculations	10
2.4 Logical Calculations	11
3 Vectors	12
3.1 Vectors with Numbers	12
3.1.1 Ways to create vectors	12
3.1.2 Calculations with Vectors	14
3.1.3 Functions helpful for numeric vectors	15
3.2 Vectors with Boolean	17
3.2.1 Functions you should know for Boolean vectors	17
3.3 Vectors with Text	18
3.3.1 Functions helpful for text vectors	19
3.4 Other vectors with special data tpyes	21
3.4.1 Vectors with NA	21
3.4.2 Vectors with NaN	22
3.4.3 Index	22
4 Matrix	23
4.1 Creating a matrix	23
4.2 Index of matrix	25
4.3 Calculation of matrix	29
4.4 Functions regarding matrix	32
5 Array	35
5.1 Creating an array	35

5.2	Calculation of Array	37
5.3	Advanced caluclation	37
5.3.1	Solving equations with matrix	37
5.3.2	Eigenvalues and Eigenvectors	38
6	Summary of Vector, Matrix, and Array.	39

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

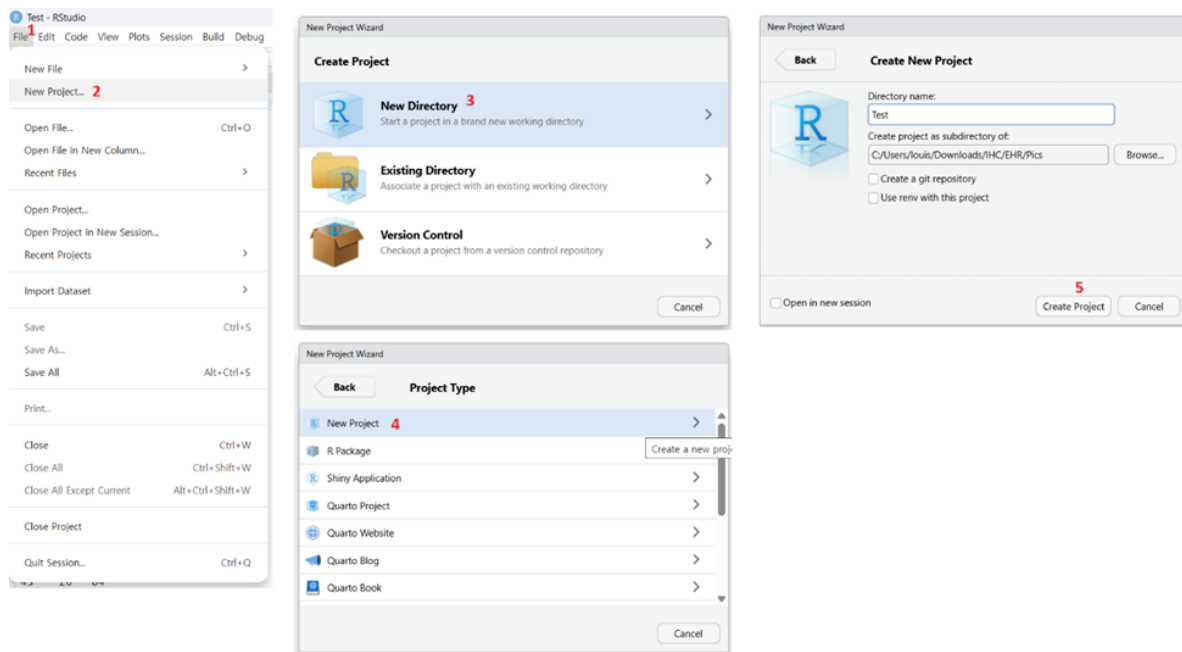
R is a programming language. R Studio is a UI that helps you write, run, and organize your R code. Let's explore how R Studio looks and what each section does.

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.

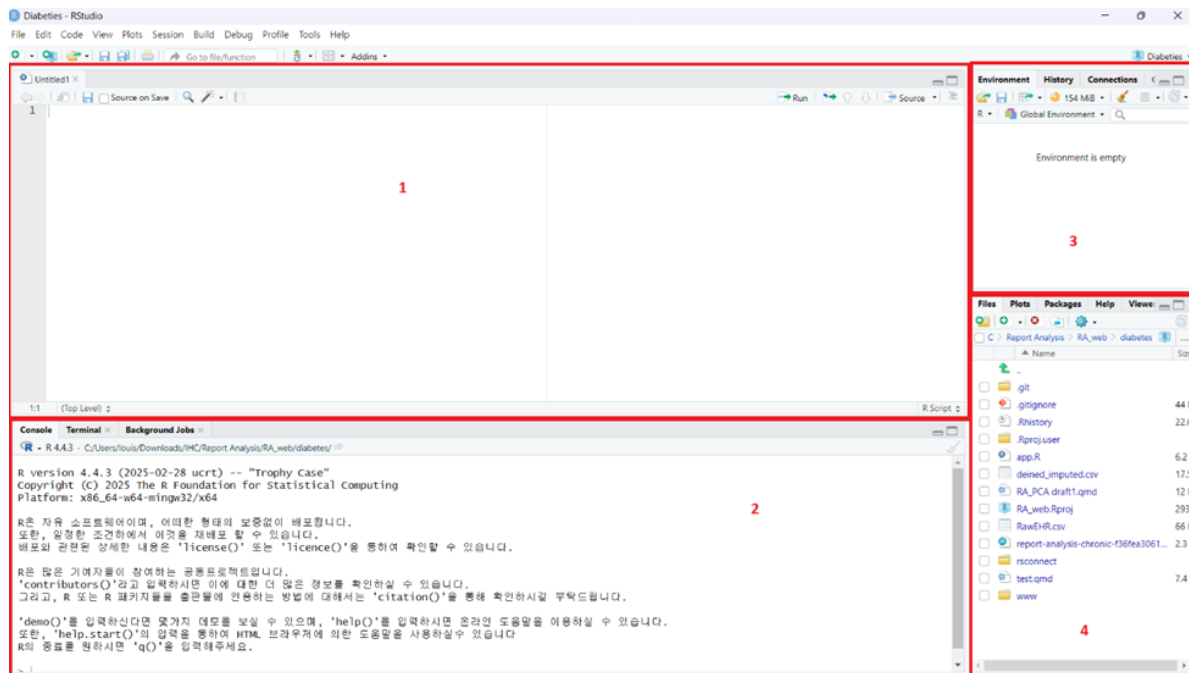
1. Click “File -> New Project -> New Directory -> New Project”
2. Choose location of folder you want to be, a good name, and click “Create Project”



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

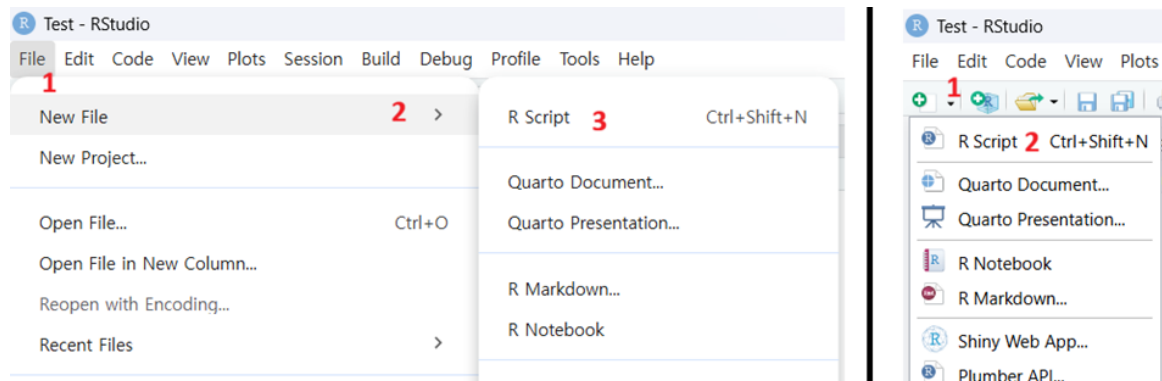
So Console is where actual works are done. Let's try using it. Input `1 + 1` and press enter. You'll see it's printing 2, the result of what you asked R to calculate.

```
> 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

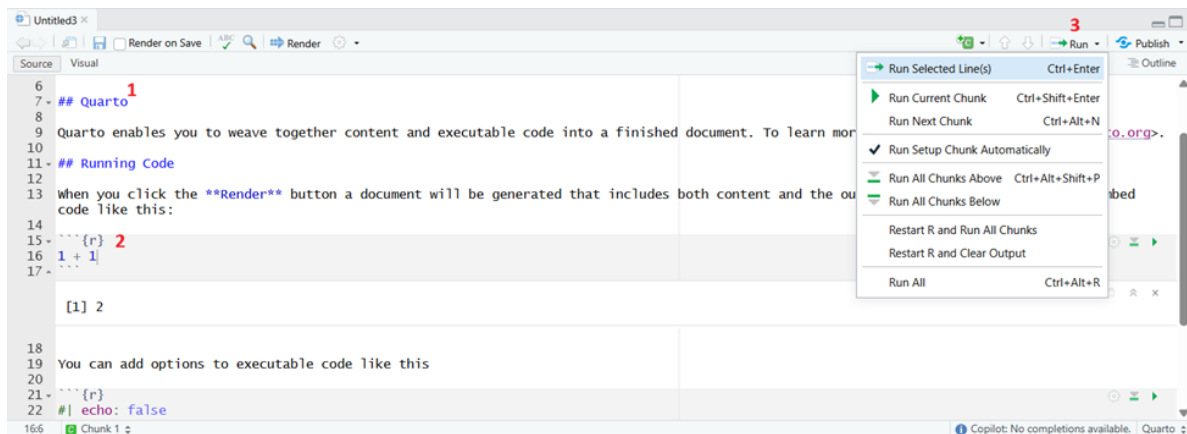
Script is a file that saves your code so that you don't have to rewrite same thing every time. Let's make a script. There are two ways to create a script.



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.

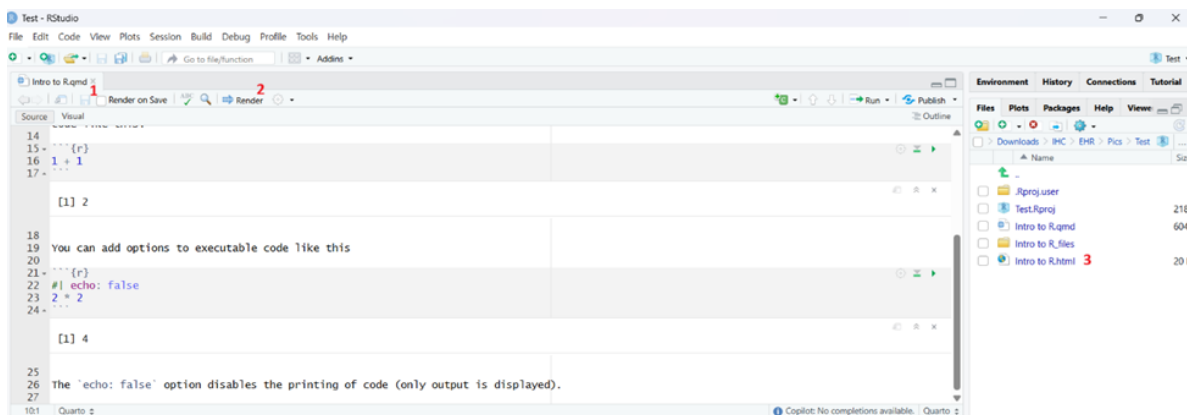


1 – # in the doc indicates the title. The more # you put, the smaller the text size gets. # (biggest) -> ####(Smallest)

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)	$n! = 1 \times 2 \times 3 \times \dots \times n$
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 are 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
```

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

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

```
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.

```
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.

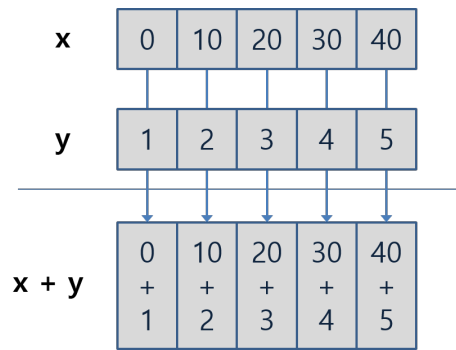
```
x <- seq(0, 40, by = 10)
y <- c(1:5)

x + y
```

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

```
x * y
```

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



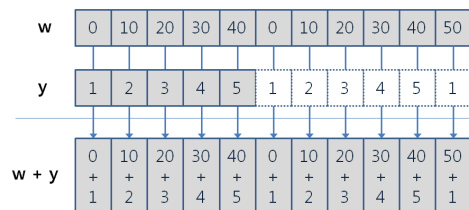
- Recycling rule What if the length of vectors are different? In this case, R will recycle the shorter vector until it matches the length of longer vector. Check the example below.

```
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 fundamental 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 than 5")
```

```
[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.

```
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 `as.character()` function to convert everything into text.

```
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:

```
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 splitting 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
<code>nchar(x)</code>	Return character counts of each elements in the vector x. *Literally all characters, including space, are counted!
<code>substr(x, start, stop)</code>	Return character counts of each element in the vector x, starting from 'start'th character to the 'stop'th character.
<code>grep(target, x, ignore.case=T, fixed=T)</code>	Return the index of element in vector x that contains "target". ignore.case asks if you want <code>grep()</code> 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)
<code>sub(target, replacement, x, ignore.case=T, fixed=T)</code>	If any element in vector x contains "target," replace it to "replacement."
<code>toupper(x)</code> <code>tolower(x)</code>	change all characters in the vector x into upper/lower case

3.4 Other vectors with special data types

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.

```
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 though 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:

```
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 indices at once. Try `x[3,5]` and see if it works.

Instead, you can use `c()` to connect multiple indices.

```
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.

```
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.

```
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
```

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

```
[1] 12
```

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

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

```
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"

```

```

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

```

```

      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

```

# 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"

```

```

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.

```
# 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.

```
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.

```
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, inputting 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](#)

Adding and subtracting matrices

Learn

-  Adding & subtracting matrices
-  Adding & subtracting matrices

Practice

Add & subtract matrices




Get 3 of 4 questions to level up!

[Practice](#)

Not
started

Properties of matrix addition & scalar multiplication

Learn

-  Intro to zero matrices
-  Properties of matrix addition
-  Properties of matrix scalar multiplication

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 matrices with different dimension and adding them. You'll see the error - R cannot perform calculations that violates fundamental 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

1. `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
```

2. `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!

3. `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
```

4. 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
```

5. `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 matrices 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

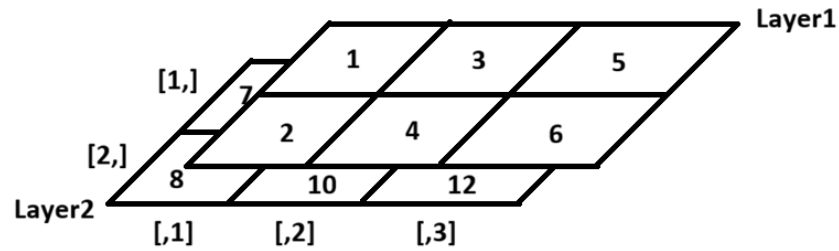
	[,1]	[,2]	[,3]
[1,]	1	3	5
[2,]	2	4	6

, , 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 separated 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 calculation

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 dimensional data, we often use matrix to represent equations with multiple dimension. For example,

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

can be written as $\begin{bmatrix} 3 & 2 & 5 \\ 4 & 6 & 12 \end{bmatrix}$ (; is used to separate rows). So, we condensed multiple equations into one 2×3 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!