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

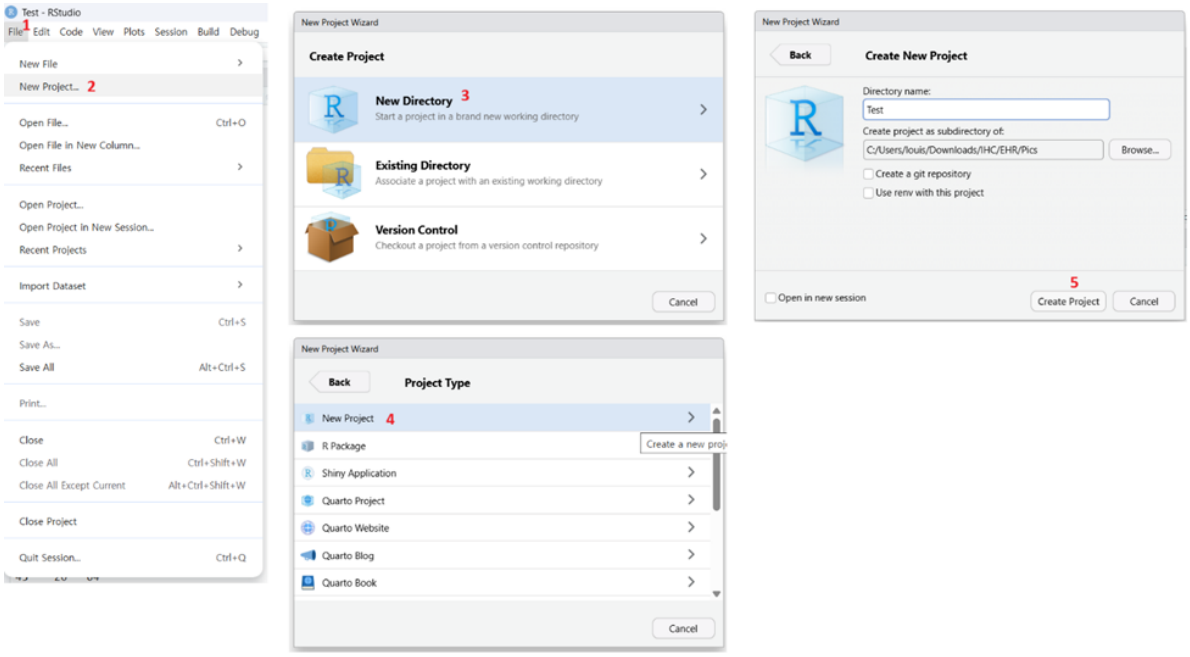
R is a programming language. R Studio is a UI that helps you write, run, and organize your R code. Let's explore how R Studio looks and what each section does.

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.

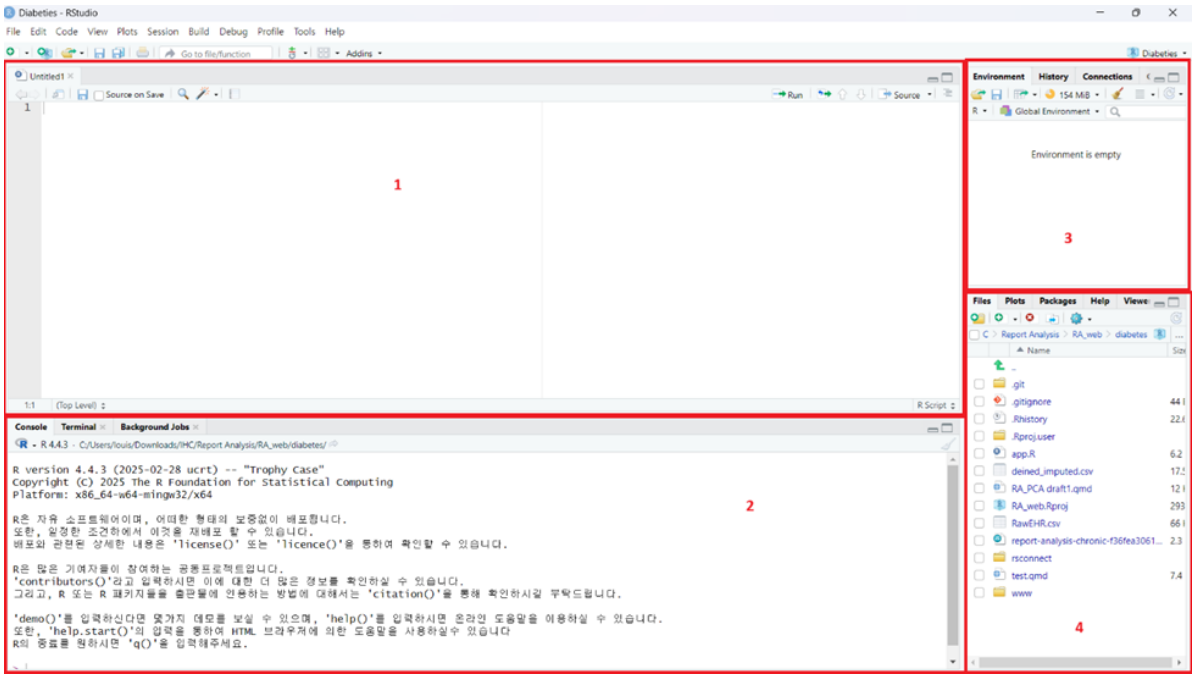
1. Click "File -> New Project -> New Directory -> New Project"
2. Choose location of folder you want to be, a good name, and click "Create Project"



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

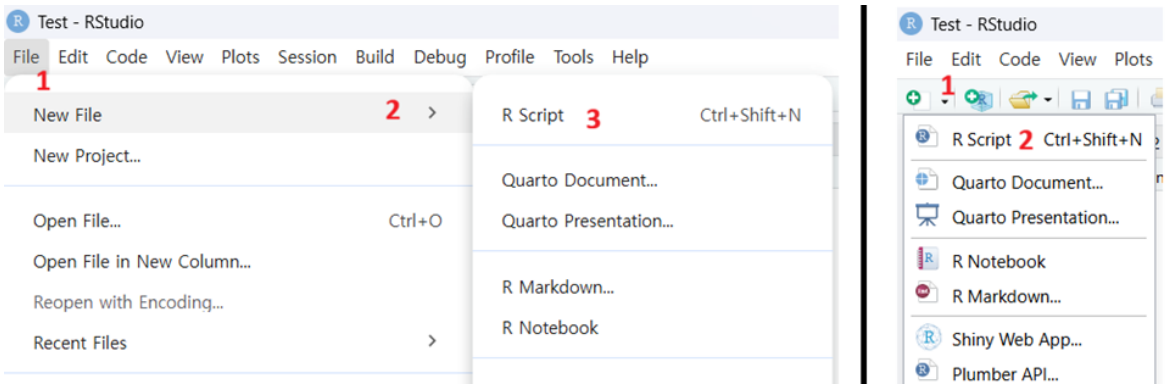
So Console is where actual works are done. Let's try using it. Input 1 + 1 and press enter. You'll see it's printing 2, the result of what you asked R to calculate.

```
> 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.

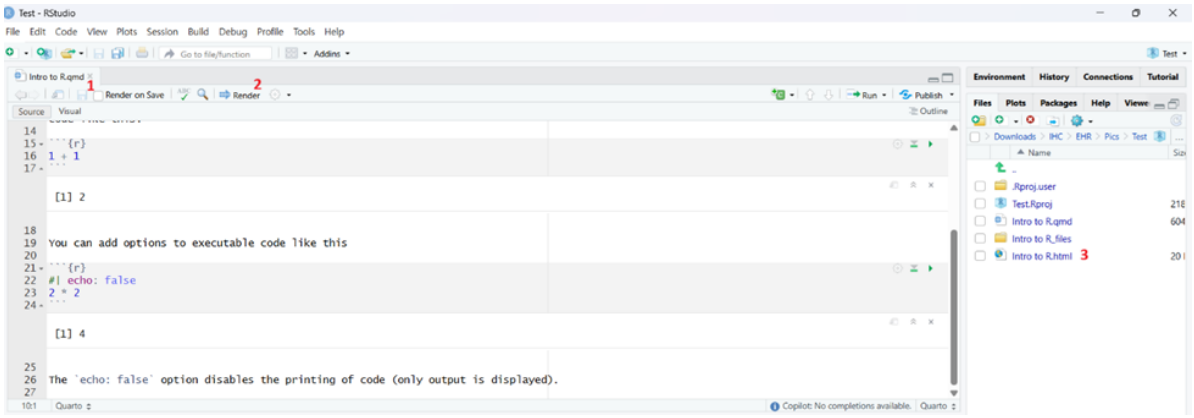
This year, we'll use Quarto Doc as we need to create reports and share codes to others. Create Quarto Doc. You can use whatever title and Author name you want. Leave other options as it is.

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.

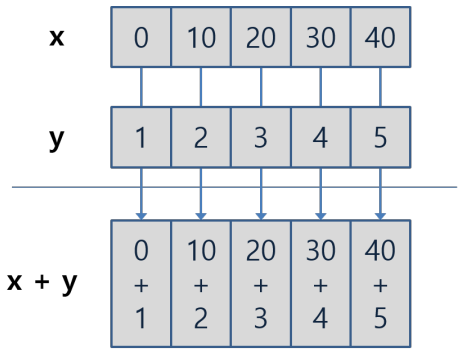
```r
x <- seq(0, 40, by = 10)
y <- c(1:5)

x + y
```

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

```r
x * y
```

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



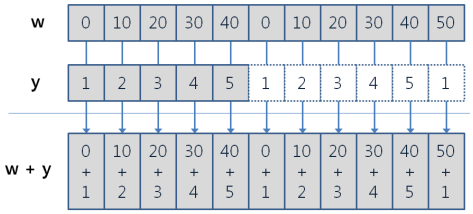
- Recycling rule What if the length of vectors are different? In this case, R will recycle the shorter vector until it matches the length of longer vector. Check the example below.

```
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
```