R Programming for Engineering Biology (EB501, KAIST 2023)

Haseong Kim

2023-11-10

Table of contents

Chapter 1

Course introduction

1.1 Introduction

KAIST Graduate School of Engineering and Biology lecture notes (2023.11)

1.2 Table of Contents

1.3 Data analysis

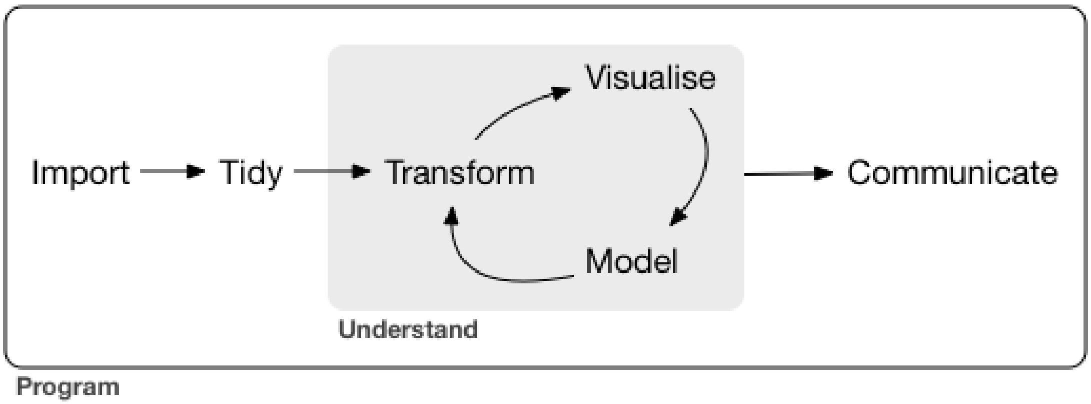


Figure 1.1: from https://r4ds.had.co.nz/

Hadley Wickham, the representative developer of posit, explains data science with the picture above at the beginning of R for Data Science. The original data must be imported into R in Tidy format and can be transformed at any time into a form suitable for analysis. And you always have to directly verify the data through visualization and use the model to perform the desired analysis. After analysis, objective verification is required through sharing. R is the best language that can perform all these processes quickly and efficiently. From a data science perspective, programming surrounds the above tools.

1.4 License

Chapter 2

Introduction

2.1 R/RStudio Introduction

R is an open source programming language widely used in research fields such as statistics, biostatistics, and genetics. It is derived from the S language, and its strengths include data analysis and visualization packages. Recently, as interest in artificial intelligence has increased significantly, the demand for the Python language has increased, but it is still a language that has many users, especially in terms of medium- and small-scale data preprocessing, visualization, and statistical analysis.

Rstudio is the key driving force behind the R language gaining attention and securing a large user base. Posit, which developed Rstudio, has its own top-level open source development team and has developed major packages related to data analysis and visualization such as tidyverse, tidymodel, and shiny, and plays a key role in disseminating technology by holding conferences on a regular basis. Tidyverse has made so many changes and improved efficiency that data analysis using R can be divided into before and after tidyverse.

2.2 Installing R/RStudio

To use R, you must first install the core program of the R language and then install RStudio, an IDE (Integrated Development Environment) for the R language. Rstudio is an open source-based integrated development environment (IDE) for the R language that provides convenient functions for R programming.

2.2.1 Installing R

- Visit the R official website [https://www.r-project.org/]. Click CRAN located at the top of the left menu.
- Select one of the Korean mirror sites from the list.
- Click ‘Download R for Windows’ and then go to the ‘base’ link.
- Click ‘Download R xxx for Windows’ to download the executable program.
- Run the downloaded Rx.xx-win.exe file on your local computer (as of November 2023, the latest version of R is 4.3.2).

- Complete the R language software installation by following the instructions in the installation program.

2.2.2 Installing RStudio

- Visit Posit's Rstudio official [https://posit.co/] website, then click 'Products > RStudio IDE' at the top of the page.
- Click ‘Download Rstudio Desktop’ under ‘Open Source Edition’ Free.
- Click ‘Download Rstudio desktop for windows’ in ‘Download Rstudio’ to start downloading.
- Run the downloaded RStudio-xxxexe file on your local computer (as of November 2023, the latest version of RStudio Desktop is 2023.09).
- Follow the installation guide to complete the installation.

2.2.3 Posit Cloud

Posit Cloud provides RStudio in the cloud, allowing users to use RStudio directly in their browser without installation or setup.

- Visit posit.cloud and register as a user (you can use your Google account).
- Log in and select ‘Cloud Free’ to get started.
- In this environment, 1GB of RAM and 1 CPU are provided free of charge.

2.3 RStudio Interface and Keyboard Shortcuts

RStudio provides a code editing window, console window, environment, and file panel. Main

keyboard shortcuts

- Code execution: Ctrl + Enter
- Go to console window: Ctrl + 2
- Go to code editing window: Ctrl + 1
- Save: Ctrl + S
- Code commenting: Ctrl + Shift + C

2.4 Starting a Project

You can start a new project in 'RStudio' by going to 'File > New Project'.

- Hello World Example
- Create a new R file and execute the following code:

```
mystring <-"Hello\nworld!" cat
(mystring)
print(mystring)
```

2.5 Getting Help

R provides extensive help data, and you can find help and examples for specific functions with the following commands:

```
help("mean")
?mean
example("mean")
help.search("mean"
) ??mean
help(package="MASS")
```

RStudio Cheat Sheet is a document in the form of a cheat sheet that allows you to understand various R functions at a glance. For more information, see Posit Cheatsheets.

2.6 R packages and datasets

An R package is a bundle of functions and data sets. By importing and using codes or functions created by others, you can reduce the effort of writing code and quickly introduce and use convenient and proven functions. For example, the sd() function is a function provided by the stats package, so there is no need to create and use a separate function for calculating standard deviation.

```
library(UsingR)
```

2.6.1 Installing and loading packages

Packages can be obtained from repositories such as CRAN or Bioconductor, and you can use RStudio or the install.packages() function in the console window as shown below.

```
install.packages("UsingR")
```

2.6.2 Checking the installation directory

You can check the installation directory of R and packages with the following command.

```
. libPaths()
path.package()
```

2.6.3 Using package data

You can also use the data contained in the package, and use the data() function to copy the package data to the user workspace.

```
data(rivers)
length(rivers)
class(rivers)
data(package="UsingR")
library(HistData)
head(Cavendish)
str(Cavendish)
```

Chapter 3

Quarto and Markdown

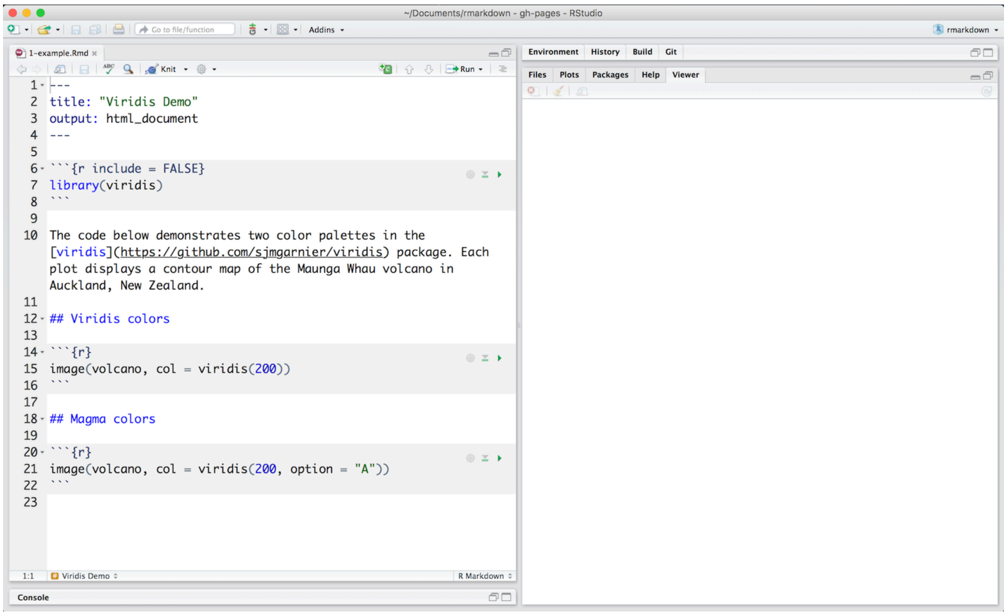
3.1 Introduction

Quarto is a workbook system that allows you to combine code and reports used in data science. Just as you can perform programming code and data analysis in word processors such as Microsoft Word or Hangul, Quarto allows you to write documents using text-based Markdown syntax and convert them to output in various formats. These outputs include HTML, PDF, Word documents, slideshows, books, websites, and more.

More detailed information about Quarto can be found on the Quarto official website. The website includes a Quarto introduction video and Quarto official site manual, and also provides a cheat sheet that can be used as a reference when using Quarto.

3.2 Quarto's basic operating principles

Quarto is written based on plain text and is saved as a file with the extension .qmd. The following text files are typical examples of Quarto files:



~~~
---
title: "Viridis Demo"
output: html_document
---

```{r include = FALSE}
library(viridis)
~~~

The code below demonstrates two color palettes in the
[viridis](https://github.com/sjmgarnier/viridis) package. Each
plot displays a contour map of the Maunga Whau volcano in
Auckland, New Zealand.

Viridis colors

```{r}
image(volcano, col = viridis(200))
```

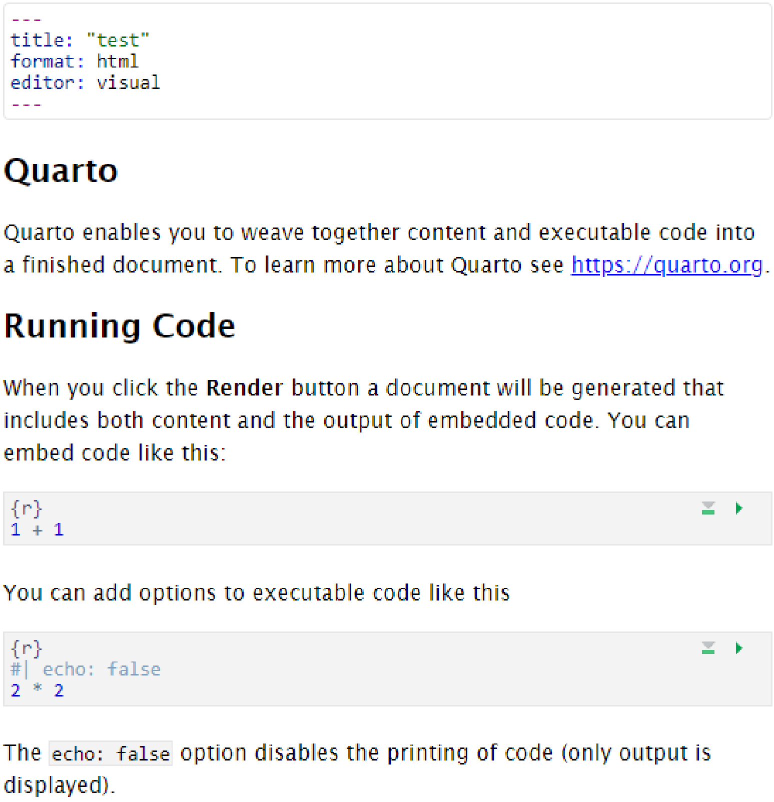
Magma colors

```{r}
image(volcano, col = viridis(200, option = "A"))
```

```

Quarto documents contain mainly three types of content: First, YAML headers define the metadata of the document and, like JSON, are used to store data. Second, code chunks are surrounded by backticks ("'), and can execute code from various languages  such as R and Python. Third, the content is written in plain text and Markdown grammar.

```
- - -
title: "Lecture"
format:
  html:
```
```

```
  toc: true
  toc-floating: true
  toc-depth: 2
  number-sections: true
- - -
```

These Quarto files can be converted into documents in the desired format using the render command. The screen below is what is displayed when creating a basic Quarto document. To render in HTML format, specify HTML in YAML and use the render function as shown below. Alternatively, you can create a PDF or HTML document using the Render button at the top of the Rstudio code input window.



```
---
title: "test"
format: html
editor: visual
---
```

Quarto

Quarto enables you to weave together content and executable code into a finished document. To learn more about Quarto see https://quarto.org.

Running Code

When you click the **Render** button a document will be generated that includes both content and the output of embedded code. You can embed code like this:

```
{r}
1 + 1
```

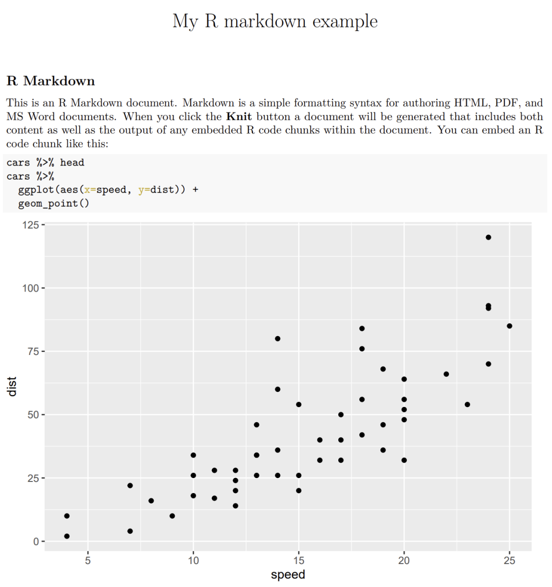
You can add options to executable code like this

```
{r}
#| echo: false
2 * 2
```

The `echo: false` option disables the printing of code (only output is displayed).

Quarto internally uses the knitr package to run code from R and other languages and convert them into .md files, and the .md files are ultimately converted into documents in various formats such as HTML, PDF, and Word through pandoc.

~~~
```{r}
# | eval=F

quarto::quarto_render("test.qmd",output_format ="html") ```
~~~



My R markdown example

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
cars %>% head
cars %>%
  ggplot(aes(x=speed, y=dist)) +
  geom_point()
```

3.3 Enter code

Here's how to insert code chunks in Quarto: You can use the shortcut Ctrl + Alt + I (Windows) or Cmd + Option + I (macOS). The following options are available to determine whether code chunks are executed and visible:

- include: false - The code runs, but the results and code are not displayed in the document.
- echo: false - The code is executed and the results are included, but the code is hidden.
- eval: false - Do not execute code, only display it in the document.
- message: false, warning: false, error: false - Hides messages, warnings, and errors that occur during code execution.
- fig.cap: "..." - Adds a caption to the figure generated by code.

R code chunk example:

The code runs, but does not display results or code.

~~~~
```{r}
# | include=FALSE

summary(cars)
```
~~~~

The code runs and the results are included, but the code is hidden.

~~~~
```{r}
# | echo=FALSE

summary(cars)
```
~~~~

Does not execute code; only displays it in the document.

~~~~
```{r}
# | eval=FALSE

summary(cars)
```
~~~~

Python code chunk example:

~~~
```{python}
# | eval: false
x="hello, python in Quarto"
print(x.split(' ')) ```
~~~

In Quarto documents, you can use code chunks and Markdown syntax to format content and run code from a variety of programming languages. Detailed information and usage instructions for Quarto can be found in the Quarto official documentation.

In Quarto, you can use 'r' to insert R code outside of a code chunk (Inlinecode). In addition to the R language, the code chunk function can also be used for various programming languages such as Python, SQL, Bash, Rcpp, Stan, JavaScript, and CSS. However, in order for these languages to be usable, there must be an engine (interpreter) that executes the languages, and in the case of Python, a package called reticulate is responsible for this function. When you install this package, an open source package for virtual environment and data analysis called miniconda is automatically installed.

~~~
```{r}
# | eval: false
library(reticulate)
```
~~~

After loading the reticulate library above, you can run the Python code chunk below.

~~~
```{python}
# | eval: false
x="hello, python in R"
print(x.split(' ')) ```
~~~

3.4 Markdown syntax

Markdown is a plain text-based markup language. The markup language uses tags, etc. to specify the data structure of the document. The method of using these tags is called a markup language. The most representative example is HTML.

```
<html>
  <head>
    <title> Hello HTML </title> </
    head>
  <body>
  Hello markup world!
  </body>
</html>
```

Markdown also uses several tags to define the structure of the document. For detailed information, please refer to the Pandoc Markdown document. The philosophy of the Markdown language is that it is a document that is easy to read and write. It is written based on plain text, so it is easy to use (the same reason many people still use notepad), and there is no difficulty in reading it even if tags are included. If you look at the examples of the HTML language above and the markdown file below, you can easily understand the difference.

In Markdown, you cannot break a line by typing Enter once. <br>Alternatively, you can enter two spaces at the end of the sentence.

```
This line break<br>
It's possible
```

If you look at some markdown tags, there is a header created by adding #.

```
# A level-one header
# # A level-two header
# # # A level-three header
```

```
# A level-one header {#l1-1}
# # A level-two header {#l2-1}
# # # A level-three header {#l3-1}

# A level-one header {#l1-2}
# # A level-two header {#l2-2}
# # # A level-three header {#l3-2}
```

There are block quotations that represent quoted phrases.

> This is a block quote. This paragraph has two lines

```
>This is block quote. This paragraph has two lines
```

> This is a block quote.
>
> > A block quote within a block quote.

```
>This is a block quote.
>
> > A block quote within a block quote.
```

Italic

```
*Italic*
```

Bold

```
* * Bold**
```

Naver link

```
[Naver link](https://www.naver.com/)
```

Insert the image directly and center it.

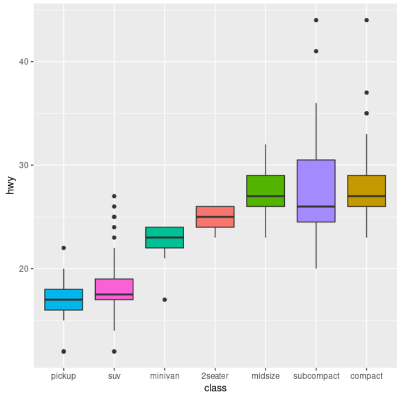


Figure 3.1: Distribution of highway fuel efficiency according to car model

```
<center>
![Distribution of highway fuel efficiency according to car model](images/rmarkdown/000002.png){width="200"} </
center>
```

Lists:

Ordered lists use numeric headers, while unordered lists use dashes or other bullet points.

1. First
2. Second
3. Third

- Item 1
- Item 2
- Item 3
    - Item 3-1
    - Item 3-2

```
1. First
2. Second
3. Third

- Item 1
- Item 2
- Item 3
  - Item 3-1
  - Item 3-2
```

For reference, in order to express the source code as is, #| Use echo: fanced or two curly braces (fanced echo help). For regular text, use three backticks or (“‘) indentation.

3.5 styles

You can insert a CSS file into the YAML header as shown below and apply the style to the content corresponding to the corresponding class or id.

```
. header1 {
color: red;
}
```

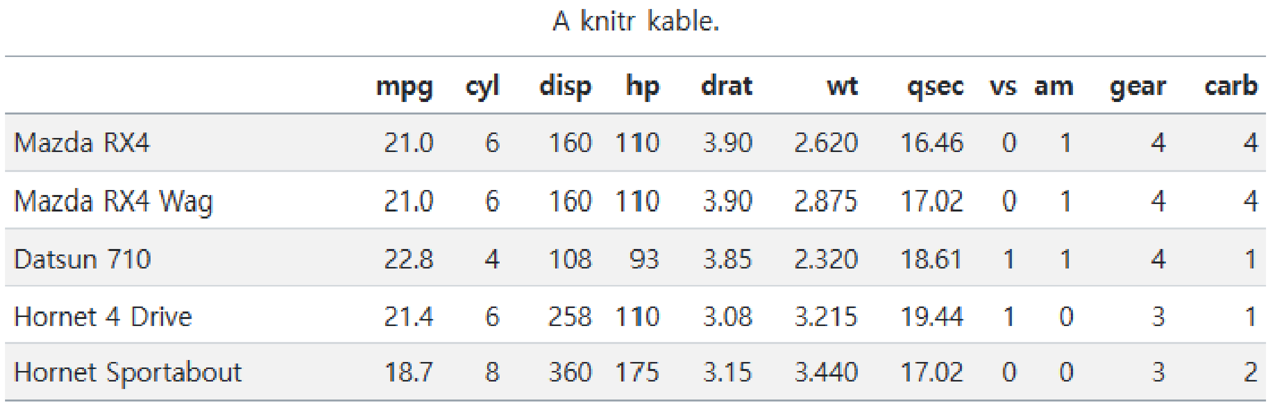
After creating the style.css file as above, use it in the quarto file as follows:

~~~
- - -
title: "test"
format:
  html:
    css: styles.css
- - -

```{r}
# | classes: header

2 + 2
```
~~~

3.6 Table

You can use the kable function to create a table included in a Quarto document in the desired direction. mtcars is data in data frame format.

```
knitr::kable( mtcars[One:5, ],
 caption =“A knitr kable.”

)
```



A knitr kable.

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2

3.7 YAML header

The most important function of YAML in Quarto files is to specify the output format, and you can also set the title, author, date, etc.

```
- - -
title: "R Programming"
subtitle: "Using Quarto"
format:
  html:
    css: style.css
    includes:
      in_header: header.html
      after_body: footer.html
    toc: true
    toc_float: true
    number_sections: true
mainfont: NanumGothic
- - -
```

3.8 Output format

Quarto's output format settings can be flexibly adjusted to suit the types and needs of the documents you create, allowing you to create high-quality documents in a variety of formats. More detailed information about Quarto documentation can be found in the official Quarto documentation.

3.9 Practice problems

3.9.1 Exercise 1: Creating a basic HTML document

Create simple HTML documents using Quarto. This document should include an introduction, a few sections, and a chunk of R code.

- Create a new Quarto document

- Set the title of the document to “My Quarto”
- Add format: html to YAML header
- Add self-introduction section and data analysis section in Markdown language.
- The “Data Analysis” section includes code chunks showing summary statistics for the mtcars dataset.

```
summary(mtcars)
```

3.9.2 Exercise 2: Creating a slideshow

Create a simple slideshow using Quarto. This slideshow must include several slides and at least one interactive R code chunk.

- Create a new Quarto document and name the file “Slideshow Exercise”.
- Add format: revealjs to YAML header
- Write a brief introduction and theme of the slideshow on the first slide.
- The second slide shows mpg vs. mtcars dataset using ggplot2. cyl graph display

```
library(ggplot2)
ggplot(mtcars, aes(x=cyl, y=mpg)) + geom_point()
```

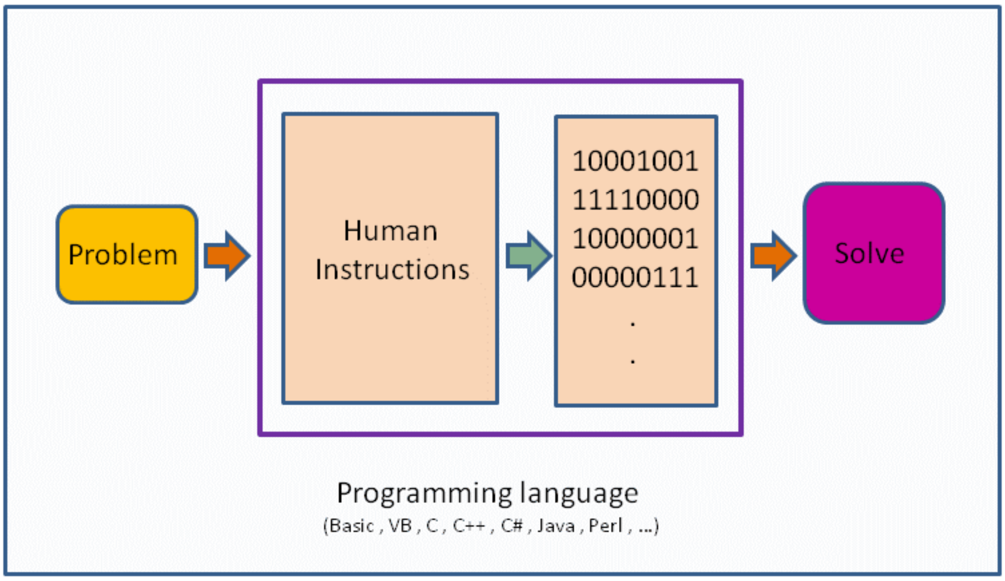
- Third slide includes conclusion and summary

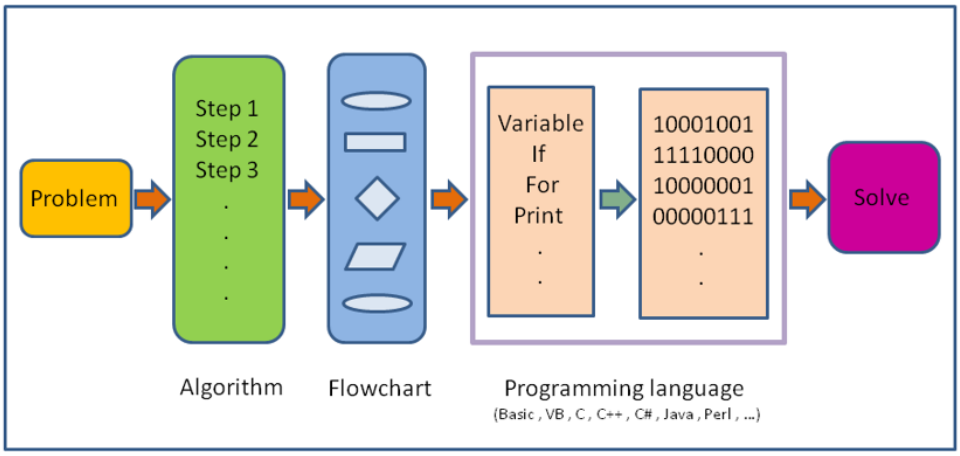
Chapter 4

R-programming

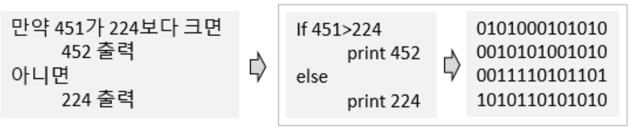
4.1 What is a programming language

A programming language is a tool that converts human-understandable code created to solve arbitrary problems into machine-understandable code. In other words, it refers to a language for communication between humans and machines, and a program is viewed as a set of logical operations to solve a problem.





Problem
두 값 451와 224 중 높은 값을 출력하라



```
만약 451가 224보다 크면
        452 출력
아니면
        224 출력
```

```
If 451>224
        print 452
else
        print 224
```

```
0101000101010
0010101001010
0011110101101
1010110101010
```

As a programming language, R has some common concepts like other programming languages (variables, data types, functions, conditionals, loops), but it also has different purposes and functions from other languages. In this chapter, we will learn about variables (variable, Object), data types (Class), functions (Function), conditional statements, and loop statements (Flow control).

4.2 Console calculator

R is a scripting language that allows you to perform calculations right in the console. You can see the results right away by entering the following calculation into the console and pressing Enter. For reference, previously performed commands can be viewed by pressing the up and down arrows while the cursor is in the console, and can be reused by pressing Enter. You can use ; to perform two commands at the same time.

```
2+2
(((2–One)^2+(One–3)^2)^(One/2) 2
+2;2-2
```

Exercises

1) Write R code that calculates the following formulas.

$$\sqrt{(4 + 3)(2 + 1)}$$

$$2_3 + 3_2$$

$$\frac{0.25 - 0.2}{\sqrt{0.2(1 - 0.2)/100}}$$

4.2.1 Terminology

- Session: R language execution environment
- Console: Window for entering commands
- Code: Collection of R programming variables/control statements
- Object: All objects (data structures) used in programming, such as variables and functions.
  - array: 1D, 2D, 3D, ... A collection of shape values
  - vector: Collection of one-dimensional values combine function c() EX: c(6, 11, 13, 31, 90, 92)
  - matrix: A collection of two-dimensional values (composed of values of the same type)
  - data frame: A collection of two-dimensional values (different types of values can be configured)
  - list: Takes various objects such as vector, matrix, data.frame, and list as elements.
- function: Performs a specific function, consists of [function name, input value (arguments), output value (return)]
- Data (value): Value - Data type
  - Integers
  - doubles/numerics
  - logicals
  - characters
  - factor: categorical
- Conditionals:
  - if, ==, & (AND), | (OR) Ex: (2 + 1 == 3) & (2 + 1 == 4)
  - for, while: number of repetitions

4.3 Data and variables

4.3.1 Data

The meaning of data is numbers that represent facts.

- McDonough Information Economics (1963)
    - Wisdom: Patterned knowledge
    - Knowledge: Valuable information
    - Information: Meaningful data
    - Data: List of simple facts

General data can be classified according to its properties as follows:

1. Categorical Data
    - Qualitative data, which can be expressed in numbers, but those numbers have no numerical meaning.
    - Nominal: Indicates categories without order or ranking. Example: person's name.
    - Ordinal: Indicates a category that has an order or rank. Example: Arrival order for a running race.
2. Numerical Data
    - It is expressed as a number, and this number reflects the properties of the data.
    - Interval: Data that has an order and equal intervals, but does not have an absolute '0'. Example: Athletes' finishing time.
    - Ratio: This is data in which there is an absolute '0' and ratio comparisons are possible. Example: End point transit time compared to departure time.

이름	등수	도착	걸린시간
둘리	1	13:12	1:12
희동	5	14:30	2:30
길동	2	13:30	1:30
철수	4	14:00	2:00
영희	3	13:50	1:50

4.3.2 Variables

A variable can be understood as a space to store data.

- Assignment operator (<- OR =)
    - Valid object name <- value
    - Shortcut: Alt + - (the minus sign)
- Built-in variables

```
x <-2
y <- x̂2–2*x+One y

x <-"two"
some_data <-9.8
pi
```

- Variable naming method
    - Characters (letters), numbers, "_", "."
    - A and a are different symbols
    - Names are effectively unlimited in length

```
i_use_snake_case <-One
otherPeopleUseCamelCase <-2
some.people.use.periods <-3
And_aFew.People_RENOUNCEconvention <-4
```

In R, a variable is a container that holds data, and as mentioned before, each data has a type with different characteristics, so the variable also has a type appropriate for it. The main data types used in R are as follows.

1. Numeric (numerical data)

- Discrete: Data made up of individual values. Example: count, number of times.
- Continuous: Data consisting of continuous values. Example: height, weight.
- Date and time: Data representing date and time.

2. Factors (categorical data)

- Categorical data used to group data.
- Used when character data is used as categorical data. Example: Data identifier.

3. Character (character data)

- Data representing a text string

4. Logical (logical data)

- Data expressed as Boolean values TRUE or FALSE

The following is Iris data, one of the most commonly used data in R, and you can check each data type as follows.

```
data(iris)
iris
class(iris$Sepal.Length)
str(iris)
glimpse(iris)
```

4.4 Object (Data structure)

It refers to all objects used in programming, such as variables and functions. The data types of the variables mentioned earlier can be viewed as the types of data values, and data types such as vector, data.frame, and list, which will be discussed later, can be viewed as the types of variable structures.

4.4.1 vector

Vector is the basic data structure in R. It can be broadly divided into three types depending on the type of stored value, such as numeric vector, logical vector, and character vector. You can use the class() function to find out the type of a value. It is created using the combine function c(), and values can be added sequentially. It is used to express the following Univariate (Single variable).

One， 2， …，

```
x <-c(10.4,5.6,3.1,6.4,21.7) class
(x)
y <-c("X1","Y2","X3","Y4")
```

```
class(y)
z <-c(T, F, F, T)
class(z)
```

4.4.1.1 numeric

Vectors of numeric type can be created using various convenience functions, such as:

```
One:5
seq(One,5,by=One) seq(0,100
,by=10) seq(0,100,length.out=
11)?seq


rep(5,times=10)
rep(One:3,times=4)
rep(One:3,each=3)
```

Exercises

1) Store only odd numbers from 1 to 100 in a variable named odds (using the seq() function).

Indexing is a method used to refer to some data in array type (vector, matrix, data.frame, etc.) data. [and] are used and are referred to as numbers indicating the position.

```
x[One]
x[One:3]
i <-One:3
x[i]
x[c(One,2,4)]
y[3]
```

It is also referred to by the name of the location.

```
head(precipice)
precip[One]
precip[2:10]precip[
c(One,3,5)] precip[-
One]
precip["Seattle Tacoma"]
precip[c("Seattle Tacoma","Portland")]
precip[2] <-10
```

For reference, vectors can use the following builtin functions to find out the attribute of the corresponding variable. Attributes refer to characteristics that a vector type variable can have, such as element name, type, and length.

```
head(precipice)
class(precipice)
length(precipice)
names(precipice)

test_scores <-c(100,90,80)
names(test_scores) <-c("Alice","Bob","Shirley")
test_scores
```

4.4.1.2 logical

Logical vectors are vectors whose elements are True or False. Remember that the first letter begins with a capital letter, and can also be expressed with a single letter, such as T or F. Logical values are also used when returning judgment results for specific conditions. In this case, after determining the conditions, the corresponding values are extracted using an indexing method (using which, any, all, etc.). Also, learn how to use the frequently used sample function.

```
x <-One:20
x>13
temp <- x>13
class(temp)

ages <-c(66,57,60,41,6,85,48,34,61,12) ages<30

which(ages<30) i <-
which(ages<30) age[i]

any(ages<30)
all(ages<30)

random_number <-sample(c(One:10),2)
```

Exercises

1) Store numbers from 1 to 100 in a variable named evens, and extract only even numbers and print them (using which() function).

2) Use the sample function to randomly select one sample from the odds and evens variables and store it in mynumbers.

3) Find out which even number was selected and print it (using which and indexing).

4.4.1.3 character

In the case of character vectors, it is convenient to know how to use the paste() function, which is often used in handling strings. The paste() function is mainly used to attach different strings. For reference, the function to split a string is strsplit(). The parameter that specifies the character that goes between the pasted characters in paste() is sep, and the character that becomes the standard for cutting in the strsplit() function is specified with the split parameter (check ?split or ?paste).

```
paste("X","Y","Z",sep="_")
paste(c("Four","The"),c("Score","quick"),c("and","fox"),sep="_") paste(
"X",One:5,sep="") paste(c("X","Y"),One:10,sep="")


x <-c("X1","Y2","X3","Y4","X5")
paste(x[One], x[2]) paste(x[One],
x[2],sep="") paste(x,collapse="_")


strsplit("XYZ",split="")
sort(c("B","C","A","D"))
```

Exercises

1) Store the string “Capital of South Korea is Seoul” in a variable called m, then extract “Capital of South Korea” separately and store it in m2 (using substr()).

2) Randomly extract 10 letters from the LETTERS built-in function, store them in the myletters variable, and concatenate them (using paste) to create one sentence (String).

3) Sort the letters in the myletters variable in alphabetical order (using sort) and connect them to create one sentence (String).

4.4.1.4 factors

The Factor type is an object for storing categorical data and is specially created and used in the R language. It is created using the factor() function, and the created object has a characteristic value indicating a category called level as follows.

For example, when five children each hold red, blue, yellow, red, and blue colored paper, the values representing the types of colors are red, blue, and yellow. Regardless of what color of paper the five children are holding, they have three categories of values.

```
x <-c("Red","Blue","Yellow","Red","Blue") y
<-factor(x)
y
```

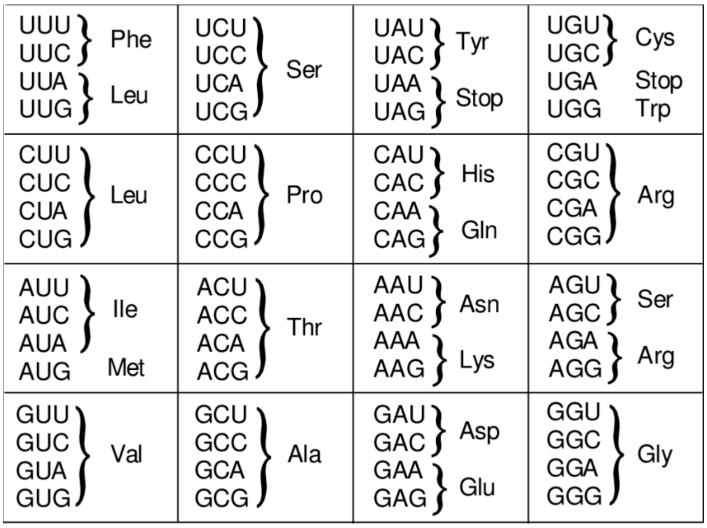
When adding data of a new category, you must first add the corresponding level and save the value as follows.

```
levels(y)
y[One] <-"Gold"
y

levels(y) <-c(levels(y),"Gold")
levels(y)
y
y[One] <-"Gold"
y
```

By default, the order displayed in the level of the factor is the position (sorting) order. To change this, you can change the order using the levels function as follows.

```
library(MASS)
str(Cars93)
x <- Cars93$Origin
plot(x)
levels(x) <-c("non-USA","USA")
levels(x)
plot(x)
```

Exercises



UUU UUC Phe UUA UUG Leu	UCU UCC UCA UCG Ser	UAU UAC Tyr UAA UAG Stop	UGU UGC Cys UGA Stop UGG Trp
CUU CUC CUA CUG Leu	CCU CCC CCA CCG Pro	CAU CAC His CAA CAG Gln	CGU CGC CGA CGG Arg
AUU AUC AUA Ile AUG Met	ACU ACC ACA ACG Thr	AAU AAC Asn AAA AAG Lys	AGU AGC Ser AGA AGG Arg
GUU GUC GUA GUG Val	GCU GCC GCA GCG Ala	GAU GAC Asp GAA GAG Glu	GGU GGC GGA GGG Gly

1) Create a categorical variable (factor) with the amino acids Phe, Leu, and Ser as values.

2) Consider whether each amino acid and the nucleotide triplets (codons) that code for that amino acid can be stored in some type of variable.

4.4.1.5 Missing values

If a specific value is “Not available” or “Missing value,” NA is used to notify data abnormalities at the corresponding element of the vector. Therefore, if NA is included in a general operation, the result of the operation becomes NA to indicate incompleteness of the data. The is.na() function is a function that checks whether the variable has an NA value. In addition to this, the following special values are used in R.

- NA: Not available, The value is missing
- NULL: a reserved value
- NaN: Not a number (0/0)
- Inf: (1/0)

```
hip_cost <-c(10500,45000,74100,N.A.,83500) sum
(hip_cost)
sum(hip_cost,na.rm=TRUE) ?
sum
```

4.4.1.6 Useful functions

The following are useful functions used with vector variables.

```
z <-sample(One:10,100, T)
head(z)
sort(z)
order(z)
table(z)
p <- z/sum(z) round
(p,digits=One)
```

You can use the is function to check whether the data type matches the type you intended. If you type is. in the console window and wait a moment, you can see various is sums.

```
is.na(One)
is.numeric(One)
is.logical(TRUE)
is.data.frame("A")
is.character("A")
```

The as function is a function that converts data types.

```
digits <-runif(10)*10
class(digits)
digits_int <-as.integer(digits)
class(digits_int)
digits_char <-as.character(digits_int)
class(digits_char)
digits_num <-as.numeric(digits_char)
class(digits_num)
```

4.4.2 matrix

A matrix is a two-dimensional matrix filled only with data values of the same format (numberic, character, logical). The method of creating a matrix is as follows. Enter the number of rows and columns in the nrow and ncol parameters, and put the value in each cell in the data parameter located at the very front (check the parameter name with ?matrix). Matrix indexing is a method used when storing or referencing (retrieving) values in a matrix. It is controlled using square brackets immediately after the matrix variable name, and the front of the square brackets, separated by commas, represents the row index, and the back of the brackets represents the column index.

```
mymat <-matrix(0,nrow=100,ncol=3)# One
mymat[,One] <-One:100# 2 mymat[,2] <-seq(
One,200,2)#3 mymat[,3] <-seq(2,200,2)# 4
```

If a name is given to a row or column of the matrix, you can refer to it by enclosing the name in quotation marks (“). The names of rows or columns can be created or changed with rownames() or colnames(). To determine the number of rows or columns, use the nrow() or ncol() function.

```
colnames(mymat)
colnames(mymat) <-c("A","B","C")
colnames(mymat)
colnames(mymat)[2] <-"D"
colnames(mymat)
rownames(mymat) <-paste(“No”,One:nrow(mymat),sep="")
rownames(mymat)
```

When referring to multiple rows or columns, you must use the combine function as shown below to group them, and when adding or subtracting a scalar value (one random number), vector / matrix operations are performed by default.

```
mymat[c(2,3,4,5),2]#5
mymat-One#6
mysub <- mymat[,2]-mymat[,One]#7
sum(mysub)#8
sum(mysubˆ2)#8
```

Exercises

1) Create a 10 x 2 matrix with 20 randomly selected numbers from 1 to 100 in a variable called score (use sample()).

2) Enter the row names of the score in character format: Name1, Name2, ... , specify Name10 (use paste())

3) Name the score column as math and eng in character format.

4) Add the numbers of the first and second columns of this matrix and store them in a variable called total_score.

5) Store the index (using the order() function) indicating the ascending order of total_score in a variable called o.

6) Rearrange the scores in o-order and save them in the score_ordered variable.

4.4.3 data.frame

A data frame has the same form as a matrix, but the difference is that each column is a vector type variable, and each variable can store values in different modes. You can refer to each configuration variable using the $ symbol. Since one column line is one variable, new variables can also be pasted in column form. In other words, each row represents a sample, each column represents a variable, and the number of samples (row length, vector length) of each variable must be the same. It can be considered the most preferred data type in R-based data analysis.

```
# #data.frame
ids <-One:10
ids
idnames <-paste("Name", ids,sep="")
idnames
students <-data.frame(ids, idnames)
students
class(students$ids) class
(students$idnames)
students$idnames
str(students)
```

```
students <-data.frame(ids, idnames,stringsAsFactors =F)
class(students$idnames) students$idnames

students[One,]
str(students)
```

In data frames, you can index by variable name using $.

```
# # data frame indexing
students$ids
students[,One]
students[,"ids"]
```

Exercises

1) Enter 10 randomly selected numbers from 1 to 100 into the variable called math.

2) Enter 10 randomly selected numbers from 1 to 100 in the variable called eng.

3) Name1, Name2, ... in character type in the variable called students. , specify Name10 (use paste())

4) Name the values stored in the vectors math and eng as the names stored in the students variable.

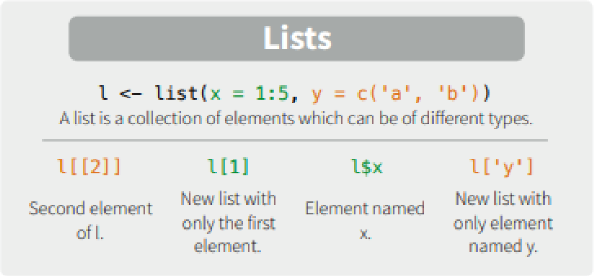
5) Create a data.frame called score with math and eng vectors.

6) Delete math and eng variables (using rm())

7) Add the math and eng of the score data frame and save them in a variable called total_score.

4.4.4 list

A list is like a data frame in that it is a collection of variables, but unlike a data frame, where all variables must have the same length, a list is different in that it can collect variables of different lengths. In other words, there are two types of data types that can contain two variables in the R language: list and data frame. The list variable type can have multiple elements in the form of a vector, and the length of each vector can be different. In indexing of a list, [ ] returns a list and [[ ]] returns vector elements.



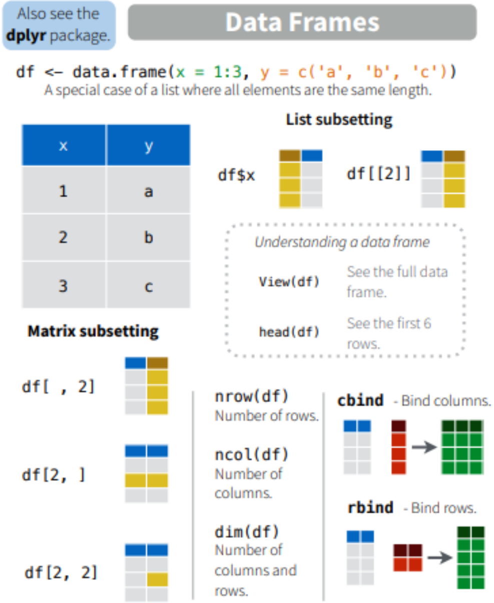
Lists

```
l <- list(x = 1:5, y = c('a', 'b'))
```

A list is a collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

```
# # list
parent_names <-c("Fred","Mary")
number_of_children <-2 child_ages
<-c(4,7,9)
data.frame(parent_names, number_of_children, child_ages) lst <-
list(parent_names, number_of_children, child_ages) lst[One]

lst[[One]]
class(lst[One])
class(lst[[One]])
lst[[One]][One]
lst[[One]][c(One,2)]
```



Also see the **dplyr** package.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`

A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting

`df$x` `df[[2]]`

Understanding a data frame

`View(df)` See the full data frame.

`head(df)` See the first 6 rows.

Matrix subsetting

`df[ , 2]`

`df[2, ]`

`df[2, 2]`

`nrow(df)` Number of rows.

`ncol(df)` Number of columns.

`dim(df)` Number of columns and rows.

cbind - Bind columns.

rbind - Bind rows.

Exercises

1) In the amino acid example above, create three vector variables with each codon of Phe, Leu, and Ser as elements, and make them into one list variable named aalist.

2) Make the aalist list into an aadf variable in data.frame format (the data structure can be changed and saved).

4.5 Functions

A function is a collection of codes that perform a function desired by the user and is designed to be easily used repeatedly.

4.5.1 A script in R

Before learning the concept of functions, let's learn about executing commands using scripts. The way to perform the function the user wants through R programming is to create and run a script as follows. Let's learn how to generally execute script commands using R. The following example is a script command that calculates and outputs the average of input values. There is a function called mean() that is provided by default in the R base package, but I did not use it, but instead used the sum() and length() functions.

```
numbers <-c(0.452,1.474,0.22,0.545,1.205,3.55) cat(
"Input numbers are", numbers,"\n") numbers_mean
<-sum(numbers)/length(numbers)
out <-paste("The average is ", numbers_mean,".\n",sep="") cat
(out)
```

Although it varies depending on the situation, usually when executing the above script, an R file is created and a function called source() is used to read and execute the entire file at once. You can create a new R file called myscript.R, save the above code, and run it as follows. Please note that the above file must be saved in the same location as the current working directory.

```
source("myscript.R")
```

However, there are a few problems with running it this way: First, every time the input value changes, you need to open the file and save the changed value. If you want to do other processing on the result, you must also edit the file directly. Additionally, if all variables are used as global variables and the code becomes complex, there is a high possibility of interference between variables.

4.5.2 Build a function

A function has a structure that receives specific data as input, performs the desired function, and then returns the resulting data. Functions can generally be implemented in the following format:

```
my_function_name <- function(parameter1, parameter2, ... ){
 # # any statements
 return(object)
}
```

For example, you can create the my_sine function as follows, where the parameter is x and y is a local variable that stores the return value.

```
my_sine <- function(x){
   y <-sin(x)
   return(y)
}
```

The created function can be used as follows. The created function is executed once at the beginning and can be used after registering it in the running R session. Here, the value pi passed to the function is called argument. You must enter the same number of transfer arguments as the number of parameters defined in the function. Please note that parameter and argument are words that many people confuse. In this example, the variable

```
my_sine(pi)
my_sine(90)
sin(90)
```

- Terminology
  - function name: my_sine
  - parameter: x
  - argument: pi
  - return value: y

Now, let's change the code used in the above script (myscript.R) into a function. The parameter that receives numbers (pass argument) is x, the function name is mymean, and it is a sum that returns the average value (numbers_mean).

```
numbers <-c(0.452,1.474,0.22,0.545,1.205,3.55)

mymean <- function(x){
 cat("Input numbers are",x,"\n")
 numbers_mean <-sum(x)/length(x)
 out <-paste("The average is ", numbers_mean,".\n",sep="") cat
 (out)
 return(numbers_mean)
}

retval <-mymean(numbers)
cat(retval)
```

If you open a file called myscript.R and create a function code as shown below in addition to the written script, you can use the source() function to read the function into the session and use it right away. When you create a function like the one above, you can change the input value at any time and easily perform additional operations on the return value.

```
new_values   <-c(One:10) retval <-
mymean(new_values) retval
```

Exercises

1) Write code to store 1, 3, 5, 7, and 9 in variable x and 2, 4, 6, 8, and 10 in variable y.

2) Write code that stores the value added by x and y in z.

3) Write a function named mysum, write code that takes two variables as input, adds them up, and returns the result.

4) Write a function named mymean, write code that takes two variables as input, calculates the average, and returns the result.

Exercises

1) Implement a function to calculate the (sample) standard deviation named mysd in the myscript.R file (do not use the sd() function, but use the following standard deviation formula)

$$= \sqrt{\frac{\sum(\; - \quad (\;))^2}{h(\;) - 1}}$$

The code is as follows

~~~~
```{r}

mysd <- function(x){
 numbers_sd <-sqrt(sum((x-mymean(x))^2)/(length(x)-One))
 return(numbers_sd)
}
```
~~~~

2) Store values   from 1 to 100 in x and use the mysd function to find the standard deviation.

3) Use the mymean function and mysd function written earlier to standardize x and save it as z. The standardized formula is:

$$= \frac{- \quad (\;)}{(\;)}$$

4) Create a data.frame named y with x and z variables as elements.

4.5.3 local and global variables

As you use a function, you need to clearly understand which variables are used inside the function and which variables are used outside the function. Looking at the following code, global variables x and y are used independently from local variables x and y.

```
my_half <- function(x){ y
 <- x/z
 cat("local variable x:",x,"\n") cat(
 "local variable y:",y,"\n") cat(
 "global variable z:", z,"\n")
 return(y)
}
y <-100
x <-20
z <-30
cat("Global variable x:",x,"\n")
cat("Global variable y:",y,"\n")
cat("Global variable z:", z,"\n")
```

```
my_half(5)

my_half <- function(x, z){ y
 <- x/z
 cat("local variable x:",x,"\n")
 cat("local variable y:",y,"\n")
 cat("local variable z:", z,"\n")
 return(y)
}

my_half(5,10)
```

Functions such as log and sin are built-in functions, so be careful not to create functions with the same name.

```
x <- pi
sin(x)
sqrt(x)
log(x)
log(x,10)
x <-c(10,20,30) x+
x
mean(x)
sum(x)/length(x)
```

4.5.4 Vectorized functions

One of the reasons why R was initially competitive compared to other programming languages was this vector operation function. If you want to apply a specific function or operation to each element contained in a vector variable, in traditional languages such as C or Java, the desired operation is performed by looping as many loops as the number of elements. However, R's vector operation function can execute functions or perform operations on elements in a vector without a separate loop.

```
x <-c(10,20,30) x+
x
sqrt(x)
sin(x)
log(x)
x-mean(x)

length(x)
test_scores <-c(Alice =87,Bob =72,James=99)
names(test_scores)
```

Exercises

The following is the weight of five people before and after the diet program.

Before 78 72 78 79 105 after 67 65 79 70 93

1) Save each in variables named before and after, then calculate the amount of change in weight value and save it in a variable named diff.

2) Find the sum, average, and standard deviation of the values stored in diff.

Exercises

1) There are four students: “John,” “James,” “Sara,” and “Lilly.” Their ages are 21, 55, 23, and 53. Create a variable called ages, store each age, and then create a function named who to print the names of people over 50 years old.

- Store age in variable called ages, use c() function, store in vector format
- Use the names() function to name each element of each ages vector.
- Use the which() function to find the index where the age is greater than 50 and store the corresponding index values in idx.
- Store the value with the index corresponding to idx in ages in sel_ages.
- Use the names() function to store the names of sel_ages in sel_names.
- Referring to the explanation above, create a function named who50 that takes a parameter called input and returns the names of people over 50 years old called sel_names.
- The usage of the who50 function is who50(ages).

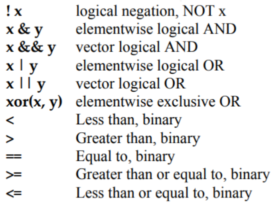
4.6 Flow control

4.6.1 if statements

The use of control statements in R is largely similar to other programming languages. First, if is used in the following format, and an expression for judging a specific condition is contained within ().

```
if(condition){
  expr_1
}else{
  expr_2
}
```

In particular, condition must be a conditional statement for one element that returns only one T or F value. The above code is a command to execute expr_1 if condition is True and execute expr_2 if condition is False. The comparison operators used in condition are as follows.



<table>
<tr><th></th><th></th></tr>
<tr><td><b>! x</b></td><td>logical negation, NOT x</td></tr>
<tr><td><b>x &amp; y</b></td><td>elementwise logical AND</td></tr>
<tr><td><b>x &amp;&amp; y</b></td><td>vector logical AND</td></tr>
<tr><td><b>x | y</b></td><td>elementwise logical OR</td></tr>
<tr><td><b>x || y</b></td><td>vector logical OR</td></tr>
<tr><td><b>xor(x, y)</b></td><td>elementwise exclusive OR</td></tr>
<tr><td><b>&lt;</b></td><td>Less than, binary</td></tr>
<tr><td><b>&gt;</b></td><td>Greater than, binary</td></tr>
<tr><td><b>==</b></td><td>Equal to, binary</td></tr>
<tr><td><b>&gt;=</b></td><td>Greater than or equal to, binary</td></tr>
<tr><td><b>&lt;=</b></td><td>Less than or equal to, binary</td></tr>
</table>

```
x <-2
if(x%%2==One){
  cat("Odd")
}else{
  cat("Even")
}

x <-5
if(x>0&x<4){
  print("Positive number less than four")
}

if(x>0)print("Positive number")

x <--5
if(x>0){
  print("Non-negative number") }
else if(x<=0&x> -5){
  print("Negative number greater than -5") }
else {
  print("Negative number less than -5")
}

if(x>0)
```

```
 print("Non-negative number")
 else
 print("Negative number")
```

4.6.2 ifelse statements

if can be used to compare only one condition. However, variables contain multiple values in vector format, and when performing vector operations, the results also come out in vector format, but it is difficult for the if statement to process them all at once. ifelse can compensate for these shortcomings and process multiple values at once.

```
ifelse(condition, return value when True, return value when False)
```

Ifelse can quickly return the desired value, but has the disadvantage of not being able to perform additional commands for each condition.

```
x <-c(One:10)
if(x>10){
  cat("Big")
}else{
  cat("Small")
}

ifelse(x>10,"Big","Small")
```

Exercises

1) The following is the formula for calculating the median, and different formulas are used depending on whether the length of x (n) is odd or even. Using the following formula and code, create a function named mymedian and create a function that finds and returns the median of the input values. (%% remainder operation, use of if statement, refer to intermediate value code below)

$$( ) = \{2\ 2] \qquad \frac{One}{}[\ \ - \ + \frac{One}{2}[1 + \ \ \overline{2}] \quad \text{if is even}$$

$$[+1\frac{}{2}] \quad \text{if is odd}$$

```
sorted_x <-sort(x)
# If the number is even
retval <- sort_x[n/2]/2+sort_x[One+(n/2)]/2
# If it is an odd number
retval <- sort_x[(n+One)/2]
```

4.6.3 for, while, repeat

The for statement is used when you want to repeatedly execute specific code. It is available in the following format:

```
for(var in seq){
  expression
}
```

var is a local variable that changes with each iteration and is used within {}. seq is a variable in vector format and the values are sequentially stored in var each time it is repeated.

```
x <-One:10
for(i in x){
  cat(i,"\n")
  flush.console()
}

sum_of_i <-0
for(i inOne:10){ sum_of_i <-
  sum_of_i+i
  cat(i," ", sum_of_i,"\n");flush.console()
}
```

The while statement, like the for statement, is used when you want to repeatedly execute specific code. The syntax used is as follows. cond contains a conditional statement that returns True or False. If it is True, the expressions are executed while continuously repeating. This repetition continues until cond becomes False.

```
while(cond){
  expression
}
```

When using the while statement, you must set a variable called an indicator and change its value each time it is repeated, as shown below. Otherwise, an infinite loop problem may occur.

```
i <-10
f <-One
while(i>One){
  f <- i*f
  i <- i-One
  cat(i, f,"\n")
}
f
factorial(10)
```

The repeat command is a command that unconditionally repeats the code in a block without conditions. Therefore, you need code to stop in the middle of a block, and this command is break.

```
repeat{
  expressions
  if(cond) break
}

i <-10
f <-One
repeat {
  f <- i*f
  i <- i-One
  cat(i, f,"\n") if(i
  <One) break
}
f
factorial(10)
```

4.6.4 Avoiding Loops

In R, it is recommended to avoid using loop statements if possible. This is also because loops execute more slowly than in other languages. However, in R, various methods are provided that can achieve the same result as executing a loop statement much faster than executing the loop statement. We will gradually learn about such techniques.

```
x <-One:1E7
sum(x)
system.time(sum(x))
```

```
st <-proc.time()
total <-0
for(i inOne:length(x))
  { total <- total+x[i]
}
ed <-proc.time
()ed-st
```

Exercises

1) Create a users variable (data.frame) with the names and ages of the following four people as data.

```
user_score <-c(90,95,88,70) user_names <-c(
"John","James","Sara","Lilly")
```

2) If each person's score is less than 80, write code that prints Name Score: Fail. If it is greater than 80, write code that prints Name Score: Pass. For example, John's score is greater than 80, so print John 90: Pass (using the for, print function)

4.6.5 dplyr::if_else

4.6.6 dplyr::case_when

4.7 Object Oriented Programming (Advanced)

OOP is called object-oriented programming. Using OOP, you can more clearly establish the concept of the problem you want to solve through programming and make complex code clearer. However, OOP in R requires a more difficult conceptual understanding than other languages. There are S3, S4, and Reference classes. S3 and S4 use generic functions and are different from the OOP concept used in other languages. Reference classes are similar to OOP concepts used in other languages and can be used using the R6 package.

Chapter 5

Data transform classic

5.1 Introduction

General data analysis can be carried out through repeated performance of data preprocessing (conversion), visualization, and modeling (statistical analysis). In R, data types in the data.frame format are mainly used (recently the tibble format), so it is necessary to learn various functions to handle data.frame-based data. In this lecture, we will learn functions for data preprocessing (conversion) along with functions that read or write data.frame data.

Let's first briefly summarize the types of objects that store the data we learned earlier.

- Vectors - Objects that store data of the same type (Numeric, character, factor, ...). The index uses [,].
- Lists - Can have multiple vectors as elements, and each element vector can be any data type, including letters or numbers, and can have different lengths. In indexing of a list, [ ] returns a list and [[ ]] returns a vector.
- Matrices - It is a two-dimensional matrix filled with data of the same type, and the index is in the form of [i, j], where i represents row and j represents column. The matrix command is used to create a matrix, and the default setting is to fill in all column values from the left and then proceed to fill in the next column value. You can also fill the row first through byrow=T. Row and column names can be set as rownames and colnames, and you can connect two matrices or a matrix and a vector with rbind and cbind (in the case of rbind and cbind, larger matrices use more computer resources).
- data.frame - An object type that has the characteristics of both a list and a matrix. Like a list, several vector variables of different types are attached to a column to form a matrix. However, unlike a list, the length of each variable must be the same (length of row). Each variable can be indexed (accessed) with the $ symbol, and like a matrix, indexing in the form of [i,j] is also possible.

5.2 Reading and writing

Reading or writing data from files into R can be essential in general data analysis processes. In this lecture, we will learn how to use commonly used text files and Excel files.

5.2.1 Text file

For convenience, we will first look at the data writing process. Among the data in the ggplot2 example, msleep data is a dataset that contains information about the sleep patterns of various animals. Let's create additional data using only some data that understands the overall structure of the data using the str or dplyr::glimpse function and then save it separately in a file.

```
library(tidyverse)
data(msleep)
str(msleep)
glimpse(msleep)

mydf <-data.frame(brainwt =msleep$brainwt,
        sleep_total =msleep$sleep_total)
class(mydf)
```

```
mydf <- msleep[,c("brainwt","sleep_total")]
class(mydf)

mydf <- msleep|>dplyr::select(brainwt, sleep_total)
class(mydf)
```

The last tidyverse-style mydf data generation is the most recommended method. The method of writing to a file is as follows. Various file writing functions are provided depending on the package, but the above two files are the most frequently used functions included in R's basic package called utils. Please look at the help with ?write.table, etc. and especially learn the function's arguments (quote, row.names, col.names, sep).

```
write.table(mydf,file="table_write1.txt") write.table(mydf,file=
"table_write2.txt",quote=F) write.table(mydf,file="table_write3.txt",quote=F,
row.names=F) write.table(mydf,file="table_write4.txt",quote=F,row.names=F,
sep="\t") write.table(mydf,file="table_write5.csv",quote=F,row.names=F,sep=
",")
```

Most text files can be saved as csv or txt files as shown below and opened with Notepad to check. When reading, you can specify as an option whether to read the separator (sep parameter) or header (header parameter).

```
dat <-read.csv("table_write5.csv")
head(dat)
str(dat)
glimpse(dat)
```

If you open the table_write5.csv file, you can see that it is separated from the header by “,” as follows. If you look at the help of the read.csv function, you can see that the parameters head and sep of this function are set to T and , by default. In addition to read.csv, you can read text files using functions such as read.table and read.delim.

Additionally, to see the relationship between sleep time and brain weight, you can visualize the data as follows:

```
plot(y=mydf$brainwt,x=mydf$sleep_total) plot(y=
log(mydf$brainwt),x=mydf$sleep_total)
```

To remove NA, use the na.omit function.

```
mydf2 <-na.omit(mydf)
mycor <-cor(log(mydf2$brainwt), mydf2$sleep_total) fit
<-lm(log(mydf2$brainwt)~mydf2$sleep_total) summary
(fit)
plot(y=log(mydf2$brainwt),x=mydf2$sleep_total);abline(fit);text(50,170,round(mycor,2))
```

5.2.2 Excel file

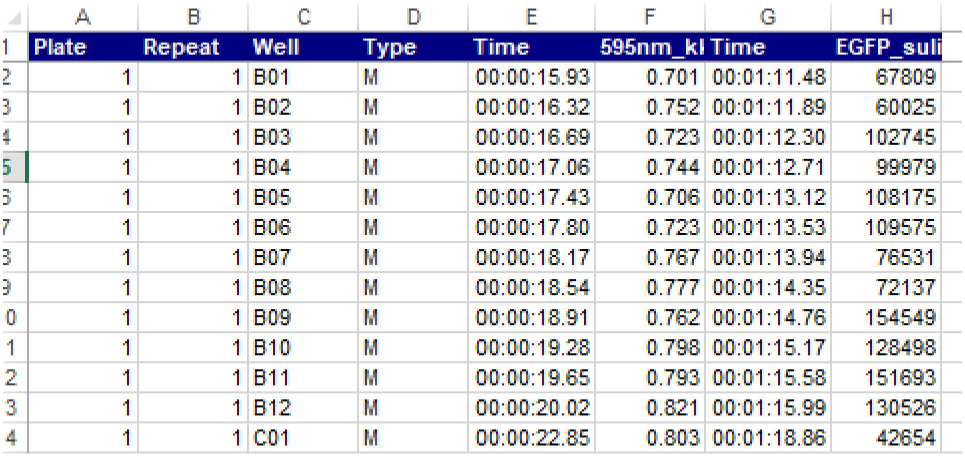
In addition to text files, Excel files can be read or written using an R package called readxl. The package can be installed in the following way and data can be read using a function called read_excel. readxl is one of the tidyverse packages.

```
# install.packages("readxl")
library(readxl)
```

The practice file is actual data obtained using a fluorescence reader by culturing fluorescent cells and can be downloaded from plate_reader.xls. When reading the contents of a file using the read_excel function, the basic data type is tibble. tibble is a recently used R object that is similar to data.frame, but the difference is that the type, name, and rowname of the input value cannot be arbitrarily changed.

```
dat <-read_excel("examples/plate_reader.xls",sheet=One,skip =0,col_names=T)
```

There are two types of Excel files (600 , Fluorescence) data is saved. The data is stored in the first sheet as follows.



	A	B	C	D	E	F	G	H
1	Plate	Repeat	Well	Type	Time	595nm_kl	Time	EGFP_suli
2	1	1	B01	M	00:00:15.93	0.701	00:01:11.48	67809
3	1	1	B02	M	00:00:16.32	0.752	00:01:11.89	60025
4	1	1	B03	M	00:00:16.69	0.723	00:01:12.30	102745
5	1	1	B04	M	00:00:17.06	0.744	00:01:12.71	99979
6	1	1	B05	M	00:00:17.43	0.706	00:01:13.12	108175
7	1	1	B06	M	00:00:17.80	0.723	00:01:13.53	109575
8	1	1	B07	M	00:00:18.17	0.767	00:01:13.94	76531
9	1	1	B08	M	00:00:18.54	0.777	00:01:14.35	72137
0	1	1	B09	M	00:00:18.91	0.762	00:01:14.76	154549
1	1	1	B10	M	00:00:19.28	0.798	00:01:15.17	128498
2	1	1	B11	M	00:00:19.65	0.793	00:01:15.58	151693
3	1	1	B12	M	00:00:20.02	0.821	00:01:15.99	130526
4	1	1	C01	M	00:00:22.85	0.803	00:01:18.86	42654

When reading the third sheet containing protocol details, set the sheet option to 3. Skip=3 and do not use separate column names, so you can read them with col_names=T.

```
dat <-read_excel("examples/plate_reader.xls",sheet=3,skip =3,col_names=F)
```

For reference, to save as an Excel file, you can use tidyverse's writexl package or write the data in a csv file, then open the csv file in Excel and save it as an xlsx file.

```
library(writexl)
dat <-read_excel("examples/plate_reader.xls",sheet=One)
write_xlsx(dat,path ="examples/plate_reader.xlsx")
```

5.3 Classical way of data transformation

Functions such as subset, filter, merge, split, and select described below are basic functions used to effectively convert arbitrary data. However, tidyverse-type data conversion functions are used more often than using the above functions individually. You can refer to the explanation below if necessary.

5.4 Subset

Data storage in R typically uses data.frame or matrix types. There may be times when you need to import data from only some columns or rows of this data and store or analyze them separately. At this time, indexing can be used to select and use some data, and the subset function also provides this selection function. This is a function that can select both banks and columns. The following airquality data measures New York's air quality by date in 1973. Let’s create a new dataset with only the remaining data excluding NA. When you use the is.na function, it returns TRUE if the data is NA, and FALSE if it is not NA.

```
glimpse(air quality)

is.na(airquality$Ozone)
ozone_complete1 <- airquality[!is.na(airquality$Ozone),]
glimpse(ozone_complete1)
ozone_complete2 <-filter(airquality,!is.na(Ozone))
glimpse(ozone_complete2)
```

ozone_complete1 and ozone_complete2 above show the same results. However, ozone_complete2 code is more intuitive and readable than ozone_complete1. In particular, you can maintain the conciseness and readability of the code by accessing the variable directly using the variable name Ozone rather than using $ as airquality$ozone. Also, the select option of subset

You can also select variables using &(AND) and |(OR) operators to set two or more conditions. In the select option below, - means to exclude the relevant variable.

```
ozone_complete3 <-subset(airquality,!is.na(ozone),select=c(ozone, temp, month, day))
ozone_complete4 <-subset(airquality,!is.na(ozone)& !is.na(solar.r),select=c(-month,-day))
```

However, you can see that the conversion process performed sequentially in the above code also becomes more complex and less readable as multiple processes are repeated.

Exercises

1) Create a data.frame named df consisting of Temp and Ozone variables from the airquality data (but exclude all samples (columns) with NA).

5.5 Merging and Splitting

The merge function is a function that performs the function of integrating two or more data sets. In particular, unlike rbind or cbind, the join is performed based on data that is common to the two data being combined or on one side of the data. Referring to ?merge, you can perform these settings with the following options: by, by.x, by.y, all, all.x, all.y. Let's understand through a simple example.

I have two datasets with 10 people, each representing their age and gender. However, df1 has only age information, and df2 only has gender information, and only three people (indexes 4, 5, and 6) have provided both information. Now, let's combine the two datasets using merge.

```
# # merge
df1 <-data.frame(id=c(One,2,3,4,5,6),age=c(30,41,33,56,20,17)) df2 <-
data.frame(id=c(4,5,6,7,8,9),gender=c("f","f","m","m","f","m"))

df_inner <-merge(df1, df2,by="id",all=F) df_outer <-
merge(df1, df2,by="id",all=T) df_left_outer <-merge
(df1, df2,by="id",all.x=T) df_right_outer <-merge
(df1, df2,by="id",all.y=T)
```

If the IDs of the two datasets are different or need to be combined based on different criteria, you can use the by.x and by.y options instead of by.

The split function is responsible for dividing data by specific criteria, and the criteria can be given in the form of a factor-type vector. For example, let's separate the data based on the month variable in the airquality data.

```
str(air quality)
g <-factor(airquality$Month)
airq_split <-split(airquality, g)
class(airq_split)
str(airq_split)
```

As shown above, airq_split is a list type with a length of 5 (May, June, July, August, September), and you can see that each element is composed of a data.frame type of different sizes.

5.6 Transformation

In R, changes to existing data can include adding, deleting, and transforming new variables and adding, deleting, and transforming samples. You can use these functions using methods such as merge, split, rbind, cbind, and changing values using indexing that you learned earlier. Also, the most intuitive way is to extract the variables you need from the existing dataset and create a new dataset using the data.frame command.

In addition to these methods, if you use within, you can easily create a variation of a specific variable and a new dataset reflecting it. Let's look at an example of using the with function, as well as an example of using the within function to transform data. The with and within functions are not commonly used functions in R. Additionally, these functions are often provided by packages such as dplyr, so they are essential.

It's not something you have to learn. However, they are good tools to help you understand the concept, and they are still functions that high-level R users use in their code, so it is good to know them.

```
# # without with
ozone_complete <- airquality[!is.na(airquality$Ozone),"Ozone"]
temp_complete <- airquality[!is.na(airquality$Temp),"Temp"] print
(mean(ozone_complete)) print(mean(temp_complete))


# # with
with(airquality, {
 print(mean(Ozone[!is.na(Ozone)]))
 print(mean(Temp[!is.na(Temp)])) })
```

As shown in the with function above, instead of accessing variables using $, within the with function (within {, }), you can directly access the variable name in the data.frame, thus improving the conciseness and readability of the code.

The within function, like the with function, allows access to the variable within {, } using only the name of the variable, but the difference is that it returns the input data and the changed variable(s). The example below is code that creates a new dataset by converting the Fahrenheit temperature of airquality data to Celsius temperature. Please compare it with the code using data.frame. If there are many variables to reference within the dataset, you can maintain code readability and conciseness simply by shortening the code in the airquality$xxx expression.

```
newairquality <-within(airquality,
 { celsius =round(((5*(Temp-32))/9,2) })

head(newairquality)

# #data.frame
celsius <-round(((5*(airquality$Temp-32))/9,2)
newairquality <-data.frame(airquality, celsius)
head(newairquality)
```

Exercises

1) Convert the values divided into hour, minute, and second of the following df into seconds, store them in a variable called seconds, and then create a df2 data set added to the existing df (using the within function).

```
df <-data.frame(hour=c(4,7,One,5,8),
                minute=c(46,56,44,37,39),
                second=c(19,45,57,41,27))
```

5.7 Babies example

Let's use the babies data from the UsingR package to perform an analysis to determine the relationship between maternal smoking status and newborn weight. There are some things you did not learn in this lecture, but please refer to them as you follow the code. First, load the UsingR package. Data where the mother's gestation period (gestation) is marked as 999 is clearly an error and is treated as NA.

```
library(UsingR)
head(babies)
# # a simple way to check out the data
plot(babies$gestation)
babies$gestation[babies]$gestation>900] <-N.A. str
(babies)
```

Use the within function as shown below to reduce the inconvenience of repeatedly entering babies$ and improve readability. In the same way, you can also perform NA processing on the error values of the dwt (father's weight) variable.

```
new_babies <-within(babies,
 {gestation[gestation==999] <-N.A.
 dwt[dwt==999] <-N.A. })

str(new_babies)
```

The smoke variable is a categorical variable indicating smoking status, and the values 0, 1, 2, and 3 are meaningless. Let's also write code to convert it to a factor type variable with a human-readable label.

```
str(babies$smoke) new_babies <-
within(babies, {gestation[gestation
 ==999] <-N.A. dwt[dwt==999] <-
 N.A. smoke =factor(smoke) levels
 (smoke) =list( "never"=0,


  "smoke now"=One,
  “until current pregnancy”=2,
  “once did, not now”=3) })

str(new_babies$smoke)
```

Now you can analyze your pregnancy period and smoking status. You can use t-test or ANOVA to analyze whether there is a difference in period by smoking group.

```
fit <-lm(gestation~smoke, new_babies)
summary(fit)## t-test results anova(fit)
```

Briefly looking at the results, summary(fit) shows the results of three t-tests. never vs. In the case of smoke new, the t-value is -1.657, which shows that the pregnancy period for those who smoke is significantly reduced compared to those who do not smoke. In comparison, there appears to be no difference if you do not currently smoke (never vs. until current pregnancy or never vs. once did, not now).

Now let's create a newdf for the sample if smoke now or if the age is less than 25 years old (use subset function, select id, gestation, age, wt, smoke variables). If you then draw a scatterplot of weight and pregnancy period using ggplot, you can see that there is not much difference, but among women who smoke, those who weigh less tend to have a shorter pregnancy period.

```
newdf <-subset(new_babies, (smoke=="smoke now"|smoke=="never")&age<25,select=c(id, gestation, age, wt,
# ggplot(newdf, aes(x=wt, y=gestation, color=smoke)) +
# geom_point(size=3, alpha=0.5) +
# facet_grid(.~smoke) +
# theme_bw()
```

5.8 Useful functions

We introduce functions that are frequently used or useful in data analysis using R, such as the various R programming techniques and functions we have learned so far. These are functions that first compare elements and extract only common or unique elements.

```
# match(), %in%, intersect()

x <-One:10
y <-5:15
match(x, y)
x%in%y
intersect(x, y)

#unique()
```

```
unique(c(x, y))
```

The following are string-related functions that are useful in sequence data analysis, etc.

```
#substr()
x <-"Factors, raw vectors, and lists, are converted"
substr(x,One,6)

# grep()
grep("raw", x)

# grepl()
grepl("raw", x) if(
grepl("raw", x)){ cat(
  “I found raw!”)
}

x <-paste(LETTERS,One:100,sep="")
grep("A", x)
x[grep("A", x)]

grepl("A", x) r <-
grepl("A", x) if(r){

  cat(“Yes, I found A”) }
  else{
  cat("No A")
}

# strsplit()
x <-c("Factors, raw vectors, and lists, are converted","vectors, or for, strings with") y
<-strsplit(x,split=", ")

#unlist()
unlist(y)

y <-strsplit(x,split="") ychar <-unlist(y) ycount
<-table(y2)ycount_sort <-sort
(ycount)ycount_sort <-sort(ycount,
decreasing =T) ycount_top <- ycount_sort[
One:5] ycount_top_char <-names
(ycount_top)


# toupper(), tolower()
toupper(ycount_top_char)
```

Exercises

Among the built-in datasets, state.abb is an abbreviation for the 50 states of the United States.

1) Select the week containing the double letter A and save it in x (use grep or grepl)

2) Infix state.abb, remove the names stored in x and store them in y (using match() or %in%)

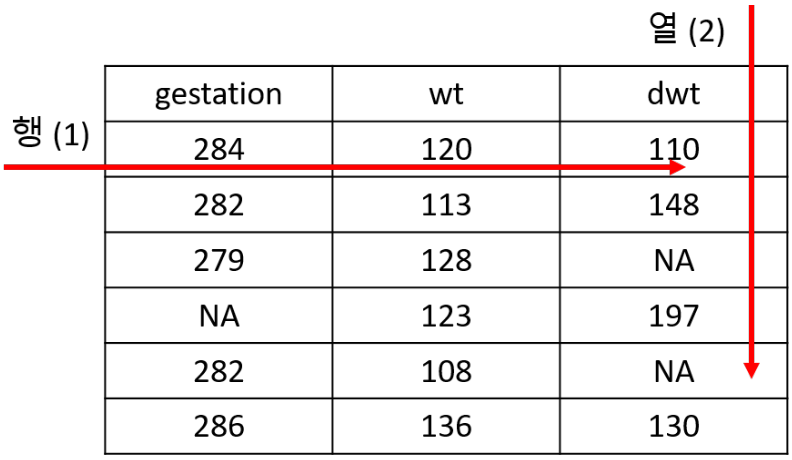
3) Find the number of alphabets used in state.abb and find the most used alphabet (using strsplit(), table(), etc.)

5.9 Apply

Apply is not a function for transforming data, but rather a function that allows you to efficiently perform repetitive tasks for each element, group, row, or column when handling data. Proper use of the apply series of functions has many advantages, including not only efficiency and convenience, but also code simplicity. For easy understanding, take the colMean function as an example. colMean is a function that calculates the average of all corresponding values by column or row. When using apply, the same function can be performed using the apply function and the mean function as follows. there is. Below is what is done on the new_babies data (created above) after clearing the babies data.

```
library(UsingR)
head(babies)
df <-subset(babies,select=c(gestation, wt, dwt))
colMeans(df,na.rm=T) apply(df,2, mean,na.rm=T)
```

As shown above, colMeans and apply show the same results. As the value of the second argument, margin (see ?apply), 2 is used here, and it operates as follows depending on whether the margin value is 1 or 2.



gestation	wt	dwt
284	120	110
282	113	148
279	128	NA
NA	123	197
282	108	NA
286	136	130

In addition to mean, various functions can be used, and you can also create and use an arbitrary function as shown below. In the code below, the definition of the function is entered directly as function(x)..., but you can create a function in advance like the mysd function below and then apply apply using the function name.

```
apply(df,2, s.d.,na.rm=T) apply
(df,2, function(x){ xmean <-
 mean(x,na.rm=T) return
 (xmean)
 })
```

The apply function can be used in place of loop (for, while, etc.) statements, which operate slowly in R, and can show fast calculation speed even for large matrices.

```
n <-40
m <-matrix(sample(One:100,n,replace=T),ncol=4)
mysd <- function(x){
 xmean <-sum(x)/length
 (x)tmpdif <- x-xmean
 xvar <-sum(tmpdif^2)/(length(x)-One)
 xsd <-sqrt(xvar) return(xsd)

}

# # for
results <-rep(0,nrow(m))
for(i inOne:nrow(m)){
  results[i] <-mysd(m[i,])
}
print(results)
apply(m,One, mysd)
apply(m,One, s.d.)
```

In addition to the apply function, various apply series functions such as sapply, lapply, and mapply can be used. First, lapply is used for list data, not matrix type data, and repeatedly performs the given function for each list element. Sapply is similar to lapply, but functions can be applied to vectors, lists, data frames, etc., and returns the results as vectors or matrices. do.

```
x <-list(a=One:10,b=exp(-3:3),logic=c(T,T,F,T))
mean(x$a)
lapply(x, mean)
sapply(x, mean)

x <-data.frame(a=One:10,b=exp(-4:5))
sapply(x, mean)

x <-c(4,9,16)
sapply(x, sqrt)
sqrt(x)

y <-c(One:10)
sapply(y, function(x){2*x})y*
2
```

As in the last example, you can also create and apply arbitrary functions like sapply or lapply. If you look closely, y is a vector with 10 values, and the function is repeatedly applied to each element (value) of this vector. In the function, x serves to receive the value of each element in turn, so values from 1 to 10 enter the function and the number multiplied by 2 is returned. Therefore, the result is the same as y*2, which is a vector operation, but it has the advantage of being able to be freely used by defining the desired function. In the case of a list, it is used as follows.

```
y <-list(a=One:10,b=exp(-3:3),logic=c(T,T,F,T))
myfunc <- function(x){
  return(mean(x,na.rm=T))
}
lapply(y, myfunc)
unlist(lapply(y, myfunc))
```

In other words, x in myfunc receives each element of list y, y[[1]], y[[2]], and y[[3]], and performs the mean operation. As a result, the average value of each list element is returned, and the unlist function converts the return value in list form to vector form.

5.10 purrr

TODO

Exercises

The following is a data set that divides the airquality data performed previously by month. Using this dataset, write appropriate code for the following step-by-step process to create data in data.frame format that stores the average values of temperature and ozone concentration for each month.

```
# # dataset
g <-factor(airquality$month)
airq_split <-split(airquality, g)
```

1) Write an ozone_func function that calculates the ozone average of the following df (however, the input receives an object in data.frame format and the output is the average value (one integer value). When using the mean function, if the data includes NA, na.rm =T option applied)

```
# #Maydata.frame df
<- airq_split[[One]]
#
# write your code here for ozone_func function
#

# # Usage
ozone_func(df)
# #output
#23.61538
```

2) Use the lapply and ozone_func functions to obtain the monthly ozone average value of the airq_split list data and save it in vector format in ozone_means.

3) Create a temp_func function in the same way as 1) and 2) above and save the average value of monthly temp in temp_means in vector format.

4) Use the two variable values obtained above and save them as a data.frame named air_means.

Exercises

1) Use the following code to download the file, save it in myexp, and check the data structure and names of samples.

```
myexp<-read.csv("https://github.com/greendaygh/kribbr2022/raw/main/examples/gse93819_expression_valu
header=T)
```

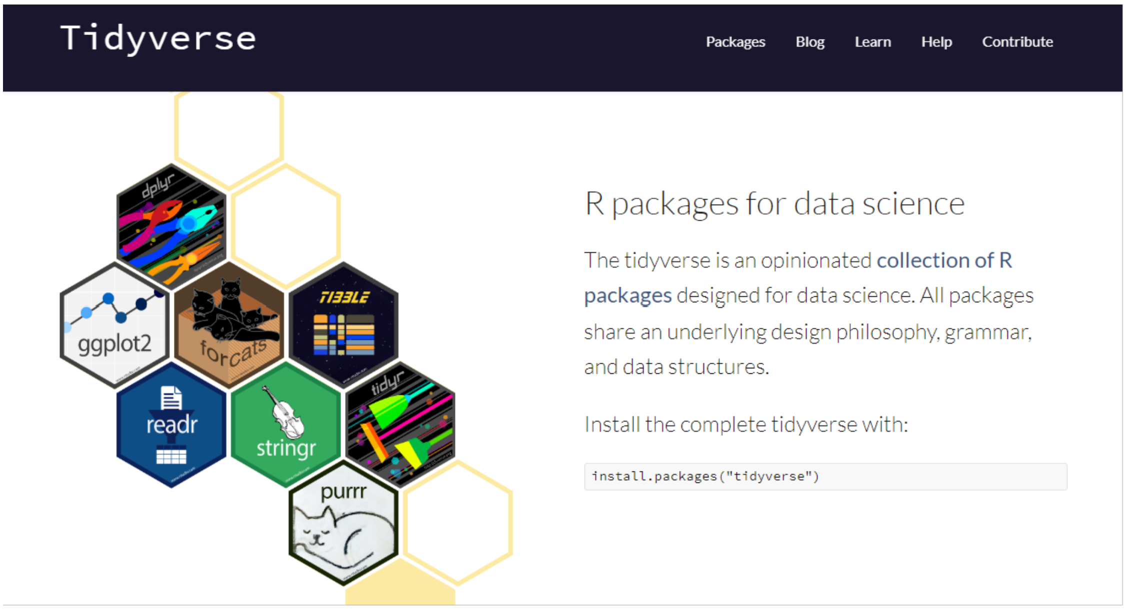
2. Divide the 1st to 10th sample (column) data of myexp by myexp1 and the 11th to 20th sample data by myexp2.
3. Find the average for each row of myexp1, find the average for each row of myexp2 in myexp1mean, and save it in myexp2mean (using apply).
4. Create a data.frame called myexpmean by combining myexp1mean and myexp2mean (using cbind, caution required).
5. Use plot to draw a scatter plot of the two averages.
6. Find the difference between the two variables in myexpmean and store it in a variable called mydiff.
7. Draw a histogram (bar graph) of the values of mydiff.

Chapter 6

Data transform tidyverse

6.1 Introduction

tidyverse (https://www.tidyverse.org/) is a collection of original R-based packages for data science. It was created by Hadley Wickham, the core expert of Rstudio, and can perform data analysis more easily and efficiently than existing tools.



R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Data science does not have a wide range of concepts and methods. However, the purpose of tidyverse is to provide highly efficient tools that are the core for data analysis, and its philosophy can be summarized in the following picture.

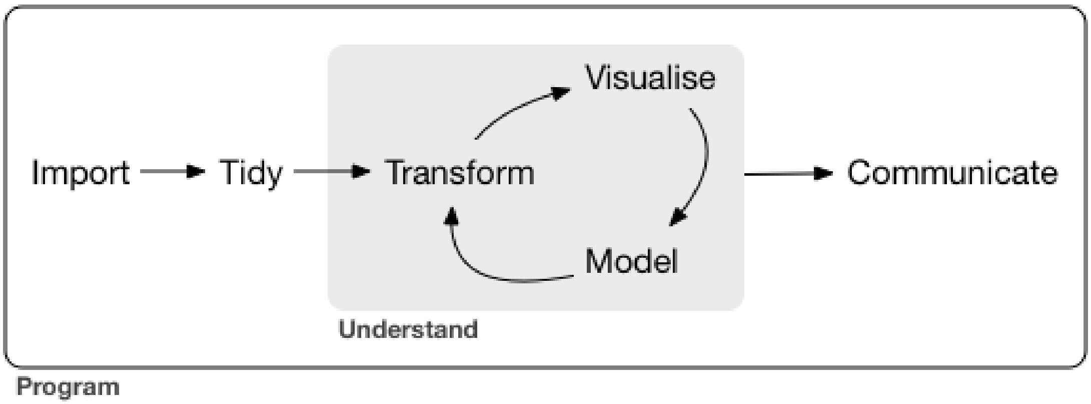


Figure 6.1: from https://r4ds.had.co.nz/

6.2 Tibble object type

R is a language with a relatively long history of more than 20 years, and the data type in the form of data.frame is most commonly used. However, as the function that was useful at the time revealed some shortcomings over time, a new type of tibble object format was created that complemented the shortcomings in the form of a package while maintaining the existing code. Most R code still uses data types in the form of data.frame, but please note that tibble is used by default in tidyverse.

```
library(tidyverse)

tb <-tibble(
  x =One:5,
  y =One,
  z =x^2+y
)
tb

iris
as_tibble(iris)
```

A tibble differs from a data.frame in a few ways: In the case of data.frame, when converting the type, there were cases where the type of the value was forcibly changed or the name of the internal variable was changed, but tibble does not allow this. Samples (rows) cannot be renamed. Also, when printing, the information that appears in the output is different. Lastly, in data.frame, the type for the subset may change, but the tibble does not change.

```
x <-One:3
y <-list(One:5,One:10,One:20)

data.frame(x, y)
tibble(x, y)
```

Another difference from data.frame in tibble is that one column can be a list-type variable rather than a vector-type variable.

```
names(data.frame`(crazy name=One))
names(tibble(`crazy name=One))
```

Additionally, the (x) reference scope of the variables used is different as follows.

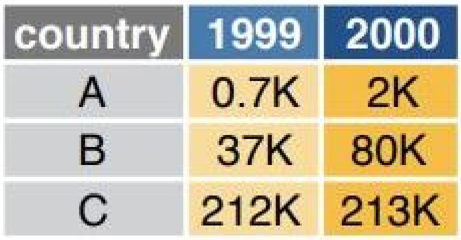
```
data.frame(x =One:5,y =x^2)
tibble(x =One:5,y =x^2)
```

```
df1 <-data.frame(x =One:3,y =3:One)
class(df1)
class(df1[,One:2])
class(df1[,One])

df2 <-tibble(x =One:3,y =3:One)
class(df2)
class(df2[,One:2])
class(df2[,One])
class(df2$x)
```

6.3 Tidy data structure

Distinguishing between data variables and values   is an essential process for proper data analysis. This may be especially important in the case of complex and large-sized data, but in most cases, classification is based on experience. Tidy data provides clear information about these variables and values.

It is one of the data structures for classification and utilization (Hadley Wickham. Tidy data. The Journal of Statistical Software, vol. 59, 2014).



country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

tidy data has the following characteristics.

- Each variable has only one column.
- Each sample has only one row.
- Each observation has only one cell corresponding to it.

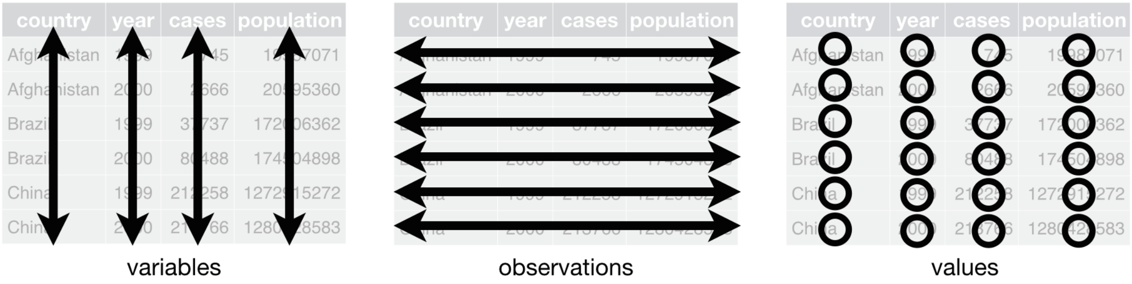


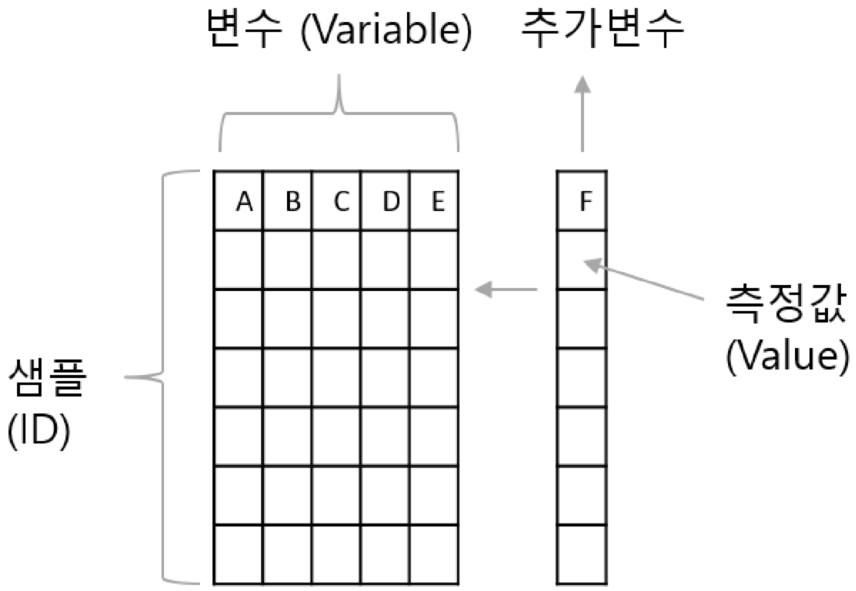
Figure 6.2: from https://r4ds.had.co.nz/

The reason why it is necessary to maintain this data structure is primarily for the efficiency of data analysis. Additionally, when data is stored and managed consistently, it becomes easier to learn and utilize analysis tools that deal with it. In particular, dplyr, ggplot2 and tidyverse packages work with tidy data format. And as variables are placed in columns, the performance of most R functions that support vector operations is maximized.

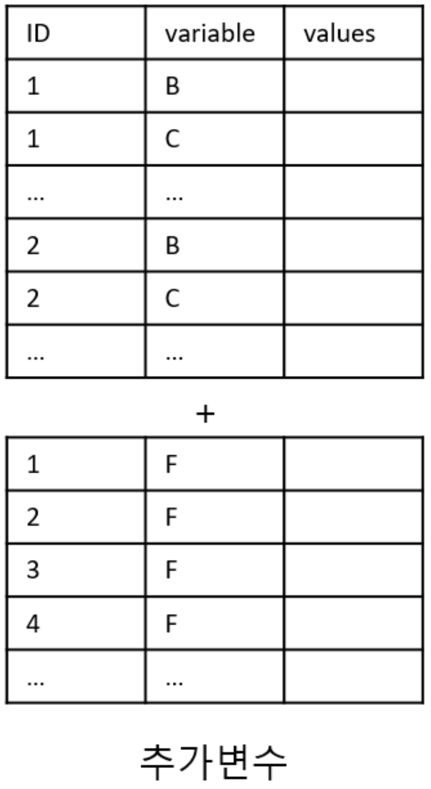
```
dat <-read_excel("examples/plate_reader.xls",sheet=One,skip =0,col_names=T)

head(dat)
glimpse(dat)
```

The above data is typical long type data. Each variable appears in only one column, and each sample has only one row. If you measure one more platter, additional data will be added to the same column further down. However, the criteria for determining whether arbitrary data is of long type or wide type may vary depending on the purpose. In this case, when drawing a plot on a two-dimensional plane with a specific objective, it is easy to judge if you consider what data will be used to draw the picture.

Tidy data is also known as Long type data. For reference, in the case of wide data, as sample data increases, the data is expanded in a way that it is piled in rows and new variables are piled in columns. The general format found in Excel is as shown below.



For long type data, you only need to remember three variables: ID, variable, and value. Looking at the wide data case above, you can see that the three factors ID, variable, and value are the main components. When converted to Long type, any variable that can refer to a sample can become an ID, and two or more variables can be designated as IDs. For reference, when specifying an ID, the ID must use a variable with non-overlapping values if possible to properly function as an identifier. When using the Long type, the number of rows increases even if the number of data variables increases, so there are advantages such as consistency in coding and the ability to create groups of variables for analysis. Below is an illustration showing when data is added to long type data when a new variable F is added.



ID	variable	values
1	B	
1	C	
...	...	
2	B	
2	C	
...	...	

+

1	F	
2	F	
3	F	
4	F	
...	...	

추가변수

6.4 Pipe operator

To utilize the tidyverse package, an understanding of pipe operators is required. In addition to the %>% pipe operator provided by magrittr, you can use the native pipe operator (|>) recently introduced in R 4.1.0. The method of operation simply receives the result of the left code of the pipe operator as output and accepts it as the input (value of the first parameter) of the right code (shortcut key: Shift+Ctrl+m).

In the following example, instead of the normal use of a function like sin(pi), you can use pi |> sin() to get the same result. It provides a significantly improved method in terms of code readability and efficiency compared to using multiple sums overlapping, such as cos(sin(pi)).

```
library(dplyr)

pi|>sin()
sin(pi)
pi|>sin()|>cos()
cos(sin(pi))
```

The pipe operator can be especially effective in situations where matrix editing/operation functions such as dplyr's group_by, split, filter, and summary, which will be explained next, are frequently used in various combinations. Typically, after the first step in the pipeline, you indent each line two spaces. If each character is on a separate line, it is indented an additional two spaces.

|> uses _ as a placeholder and magrittr uses . as a %>% placeholder. magrittr's %>% is a pipe operator that has been widely used in R data analysis for a long time, providing advanced functionality and flexibility, but recently the use of the native pipe operator (|>) has been recommended, both operators for data processing in R. It is an important tool that makes analysis more efficient and intuitive.

In the following code, x is entered as the first parameter of paste, so you can see that "1a", "2a", "3a", "4a", and "5a" are output with the x values in front of a.

```
x <-One:5
x|>paste("a",sep="")
```

Let's consider the case of calculating the average for each column of a specific data set and calculating the sum of each average. R provides a special function called colMeans to calculate the average for each column. After that, you can use the sum function to get the final desired value. You can compare this code with the code when using the |> operator.

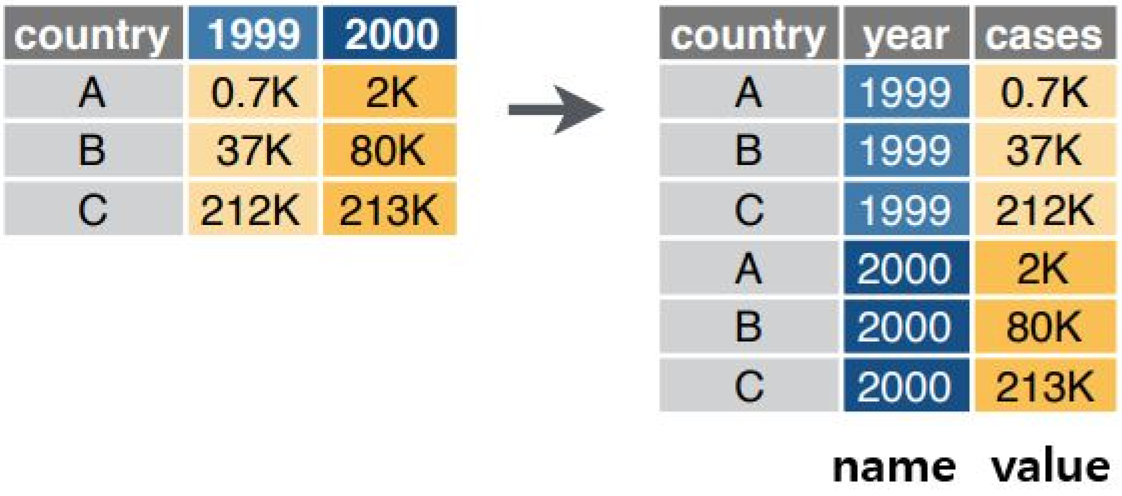
```
x <-data.frame(x=c(One:100),y=c(201:300))
sum(colMeans(x))

x <-data.frame(x=c(One:100),y=c(201:300)) x|
>
  colMeans()|>
  sum()
```

If you want to accept the output of the left statement as input to the second parameter, you can use placeholder_. The round function can set two parameters, and if you want to pass the value to the second parameter called digits using the pipe operator, it can be expressed as follows.

```
6|>round(pi,digits=_)
round(pi,digits=6)
```

6.5 Pivoting

Generally, the data type obtained is wide type, and to convert it to long type, pivot_longer and pivot_wider of the tidyr package belonging to the tidyverse package are used. Previously, you could use the melt function in the reshape2 package and, vice versa, the dcast function, but the tidyr package is widely used. The operation of converting wide type data to long type or converting long type to wide type is called pivoting.



country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

Airquality data can be viewed as wide type data or long type data. As mentioned earlier, depending on the purpose, airquality data can be wide or long type. Airquality data is a collection of measured values for several airquality indicators on a specific date. If Ozone and Solar.R are the only variables needed for analysis, they can be viewed as long type data with two variables.

```
airquality
myair2 <- airquality|>
  dplyr::select(Ozone, Solar.R, Day, Month)

myair2
ggplot(myair2,aes(x=Solar.R,y=Ozone))+
  geom_point()

fit <-lm(myair2$Ozone~myair2$Solar.R)
summary(fit)
```

However, if each column is viewed as a categorical data value that can represent airquality, the airquality data becomes wide data. If you change this data to long type, you can identify the data by setting the ID to date. However, since the date is divided into two variables, Month and Day, the two variables are used as identification variables (ID) as follows. To confirm, let's perform a type conversion using only the top 5 data.

```
airquality

myair <- airquality[One:5,]
myair

myair_long <-pivot_longer(my air,c("Ozone",“Solar.R”,"Wind","Temp"))
myair_long

myair_long <- myair|>
  pivot_longer(c("Ozone",“Solar.R”,"Wind","Temp"))
  myair_long

myair_long2 <- myair|>
  pivot_longer(c(Ozone, Solar.R, Wind, Temp))
  myair_long2

myair_long3 <- myair|>
  pivot_longer(!c(Month, Day))
  myair_long3
```

The name and value of the variable names of the generated long type data can be changed by specifying the following parameters.

```
myair_long <- myair|>
 pivot_longer(c(Ozone, Solar.R, Wind, Temp),
        names_to ="Type", values_to
        ="Observation") myair_long
```

You can also convert long type data to wide type data.

```
myair_long|>
 pivot_wider(
  names_from =Type,
  values_from =Observation)
```

Long type data like the above is mainly used to create graphs using ggplot. Data visualization using R is performed by editing wide data with the dplyr package, converting it to long data with the pivot_longer function, and then using ggplot. For more specific information about the two data formats, please refer to the following links. https://www.theanalysisfactor.com/wide-and-long-data/

```
library(ggplot2)
data(msleep)
head(msleep)

ggplot(msleep,aes(x =brainwt,y =sleep_total))+
 geom_point()+
 scale_x_log10()+
 xlab("Brain Weight (log scale)")+
 ylab("Total Sleep Time (hours)")+
 ggtitle("Relationship between Brain Weight and Total Sleep Time")
```

6.6 Separating and uniting

When analyzing data, there are often cases where the values of two or more variables are stored in one column or two variables need to be combined into one column. In the former case, you can divide it into two variables (columns) using the separate() function, and in the latter case, you can use the unite() function to merge the two variables into one value. The following is an example of merging the Month and Day variables in airquality data into one column to create a variable called Date.

```
newairquality <- airquality|> unite
 (Date, Month, Day,sep="."
 )newairquality
```

Using the separate() function, you can divide the value of the variable into two variables (columns) as follows.

```
newairquality|>
 separate(col=Date,into =c("Month","Day"),sep ="\\.")
```

6.7 dplyr

dplyr (https://dplyr.tidyverse.org/) was created mainly by Hadley Wickham, who developed ggplot2, and is a core package of tidyverse (https://www.tidyverse.org/) along with ggplot2. . dplyr is a newly created package that improves the size of data handling, speed of analysis, and convenience. It can be said to be a tool created by adding matrix operation functions such as the existing apply and matrix editing functions such as subset, split, and group.

The plyr package, which can be considered the predecessor of dplyr, is described as follows. A set of tools for a common set of problems: you need to split up a big data structure into homogeneous pieces, apply a function to each piece and then combine all the results back together. In other words, split-apply-combine is a tool created to easily perform three operations. The reason R is attracting attention in data analysis compared to other languages is the development of matrix operation functions such as split and apply, and dplyr is designed to make them more convenient to use.

Now let’s use the function provided by the dplyr package. The important functions that make up dplyr are:

- select() - Select variables (columns)
- filter() - select samples (rows)
- arrange() - Change the sort order of samples
- mutate() - Create a new variable
- summarise() - Create a representative value
- group_by() - Perform calculations by group
- join() - Used to merge two tibbles or data.frames
- Helper functions that can be used in combination (within a function) with the above functions (particularly filter, select, mutate, summarise) can be used together (can also be used independently).
  - across
  - if_any
  - if_all
  - everything
  - starts_with
  - end_with
  - contains

These functions provide powerful performance when used with %>%. The summarise function is a function that calculates statistical values of specific values, and other functions can be viewed as functions for editing matrices. Let's walk through a simple example and explore each feature to understand why dplyr is so widely used and what its advantages are.

6.7.1 select

select() selects and displays the variable (column) of interest from the given data set.

```
head(iris)
iris|>
 select(Species,everything())|>
 head(5)
iris|>select(Species,everything()) iris
|>select(-Species)
```

The following helper functions can be useful like the select function.

starts_with(“abc”) - Variable name with a string starting with “abc” ends_with(“xyz”) - Variable name with a string ending with “xyz” contains(“ijk”) - Variable containing the string “ijk” Namematches(“(.)\1”) - regular expression, repeated characters

```
iris|>select(starts_with('S')) iris|>
select(obs =starts_with('S'))
```

Below is how to use the matches function. You can select variables by applying more complex patterns, and you can also apply regular expression patterns when using the grep function.

```
iris2 <-rename(iris,aavar =Petal.Length)
select(iris2,matches("(.)\\One"))
```

```
tmp <-iris[,3:5]
colnames(iris)[grep("^S",colnames(iris))]
iris[,grep("^S",colnames(iris))] tmp
```

Below (.)\\1 refers to the variable name in which one character . (any character) is used once more\\1. This is the only aa in aavar, so aavar is selected. In grep, the ^ mark indicates the beginning, so ^S will be the letter starting with S. Therefore, in the case of grep("^S", colnames(iris)), any column name starting with S returns True, otherwise, it returns False.

6.7.2 filter

You can use the filter function to select data (samples) that meet the desired conditions.

```
library(dplyr)

head(iris)
iris|>
 filter(Species=="setosa")

iris|>
 filter(Species=="setosa"|Species=="versicolor")

iris|>
 filter(Species=="setosa"&Species=="versicolor")

iris|>
 filter(Species=="setosa"|Species=="versicolor")|>
 dim()
```

The parameters separated by , in filter are conditions bound with and logic. As you saw in the last lecture, in R, and means &, or means |, and not means ! It can be used as and the conditions separated by , in the filter can be considered the same as and.

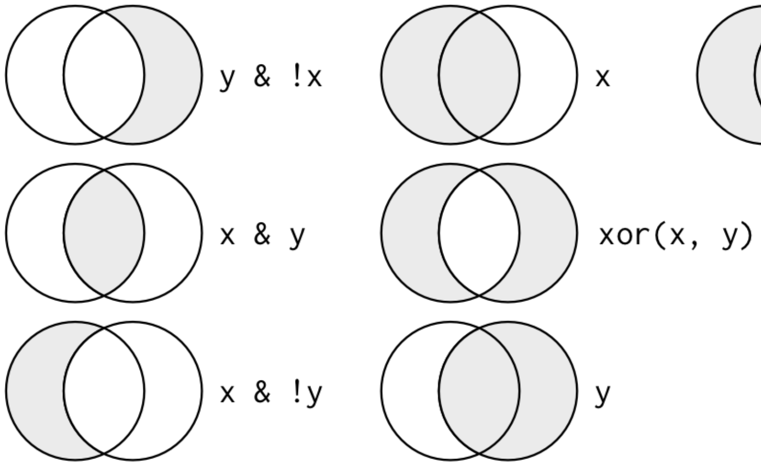


Image from (https://r4ds.had.co.nz/)

6.7.3 arrange

arrange() performs the function of changing the array order of samples, that is, the row order, in order of value size based on a specified variable. By default, sorting is done in the order of increasing size. If you want to order the size to decrease, you can use the desc function.

```
iris|>arrange(Sepal.Length) iris|>arrange(
desc(Sepal.Length)) iris|>arrange
(Sepal.Length, Sepal.Width)
```

6.7.4 mutate

The mutate() function provides the ability to add new variables and can be seen as similar to within(), which we learned earlier. As shown below, the mutate function creates a new variable called sepal_ratio and returns it along with the existing iris data.

```
iris2 <- iris|>mutate(sepal_ratio =Sepal.Length/Sepal.Width)
head(iris2)
```

6.7.5 summary

summarise() creates one summary/representative value from the values of specific variables in data.frame. The summarize function is useful when used in parallel with the group_by() function rather than used alone. Using the summarize_all() function executes the specified function for all variables. In particular, the summarise function can be used in combination with helper functions such as across, if_any, and if_all as follows.

```
iris|>summarise(mean(Sepal.Length),m=mean(Sepal.Width)) iris|
>
 group_by(Species)|>
 summarise(mean(Sepal.Width))

iris|>
 group_by(Species)|>
 summary_all(mean)

iris|>
 group_by(Species)|>
 summarise(across(everything(), mean))


iris|>
 group_by(Species)|>
 summary_all(sd)

iris|>
 group_by(Species)|>
 summarise(across(everything(), sd))
```

6.7.6 join

The join function is a function that performs the function of merging data. There are four types of functions (left_join(), 'right_join(), 'inner_join(), 'full_join()), and they basically perform the function of automatically merging common samples using variables with common names (keys). The function is performed based on the value of the parameter specified in ‘by’.

```
df1 <-data.frame(id=c(One,2,3,4,5,6),age=c(30,41,33,56,20,17)) df2 <-
data.frame(id=c(4,5,6,7,8,9),gender=c("f","f","m","m","f","m"))

inner_join(df1, df2,by="id")
left_join(df1, df2,"id")
right_join(df1, df2,"id")
full_join(df1, df2,"id")

# vs.
cbind(df1, df2)
```

6.8 Code comparison

Now let's compare the code that calculates the average using split, apply, and combine with the code created using the dplyr package. This code analyzes iris data and calculates the average and standard deviation of the calyx length (Sepal.length) for each variety, as well as the number of samples.

split plays the role of dividing iris data based on Species, a factor type variable, and lapply plays the role of performing an arbitrary function function(x)... for each element of each list using iris_split, a list type data. do. Combine the final path into the finaldata.frame.

```
iris_split <-split(iris, iris$Species)
iris_means <-lapply(iris_split, function(x){mean(x$Sepal.Length)}) iris_sd
<-lapply(iris_split, function(x){sd(x$Sepal.Length)}) iris_cnt <-lapply
(iris_split, function(x){length(x$Sepal.Length)}) iris_df <-data.frame(unlist
(iris_cnt),unlist(iris_means),unlist(iris_sd))
```

Below is the code using the dplyr package.

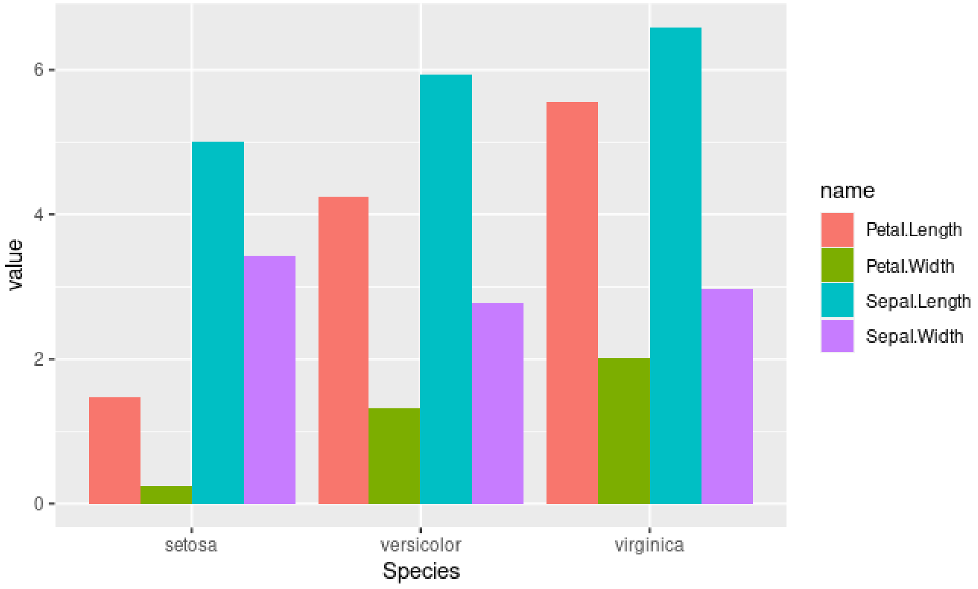
```
iris_df <- iris |>
  group_by(Species) |>
  summarise(n=n(),mean=mean(Sepal.Length),sd=sd(Sepal.Length))
```

As seen above, when using the dplyr package, the result is the same, but it shows advantages in terms of code readability and efficiency. This means receiving iris data, dividing it into groups specified in Species, and applying the desired function to the target column. The following is code to calculate the average for all variables.

```
iris_mean_df <- iris |>
  group_by(Species) |>
  summarise(across(everything(), mean))
```

We will learn more about ggplot next time, but let's create a bar graph for each average.

```
library(ggplot2)

iris_mean_df2 <- iris_mean_df |>
  pivot_longer(-Species)

ggplot(iris_mean_df2,aes(x=Species,y=value,fill=name))+
  geom_bar(stat="identity",position="dodge")
```



Chapter 7

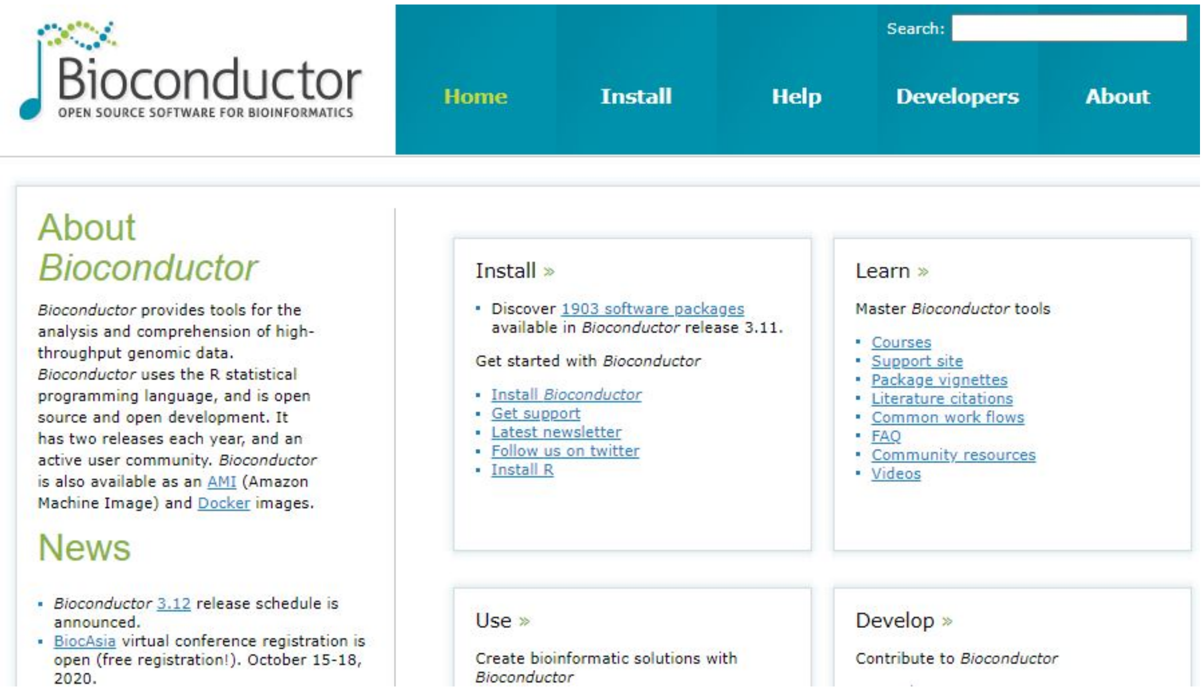
Bioconductor

7.1 Introduction

- https://www.bioconductor.org

Bioconductor is a collection of R-based data, methods, and packages for bioinformatics. It started as a platform for microarray data analysis in 2002 and currently consists of more than 2000 packages. R is a distributed open source, but Bioconductor is maintained by full-time developers. It is not distributed on CRAN, requires more essential materials (vignettes, etc.) than CRAN, and is subject to a high level of quality control. Bioconductor has scheduled releases every six months to ensure that all bioconductor packages work harmoniously and without conflicts.

Please refer to here for available packages.



About *Bioconductor*

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. *Bioconductor* uses the R statistical programming language, and is open source and open development. It has two releases each year, and an active user community. *Bioconductor* is also available as an AMI (Amazon Machine Image) and Docker images.

News

- *Bioconductor* 3.12 release schedule is announced.
- BiocAsia virtual conference registration is open (free registration!). October 15-18, 2020.

Install »

- Discover 1903 software packages available in *Bioconductor* release 3.11.

Get started with *Bioconductor*

- Install *Bioconductor*
- Get support
- Latest newsletter
- Follow us on twitter
- Install R

Learn »

Master *Bioconductor* tools

- Courses
- Support site
- Package vignettes
- Literature citations
- Common work flows
- FAQ
- Community resources
- Videos

Use »

Create bioinformatic solutions with *Bioconductor*

Develop »

Contribute to *Bioconductor*

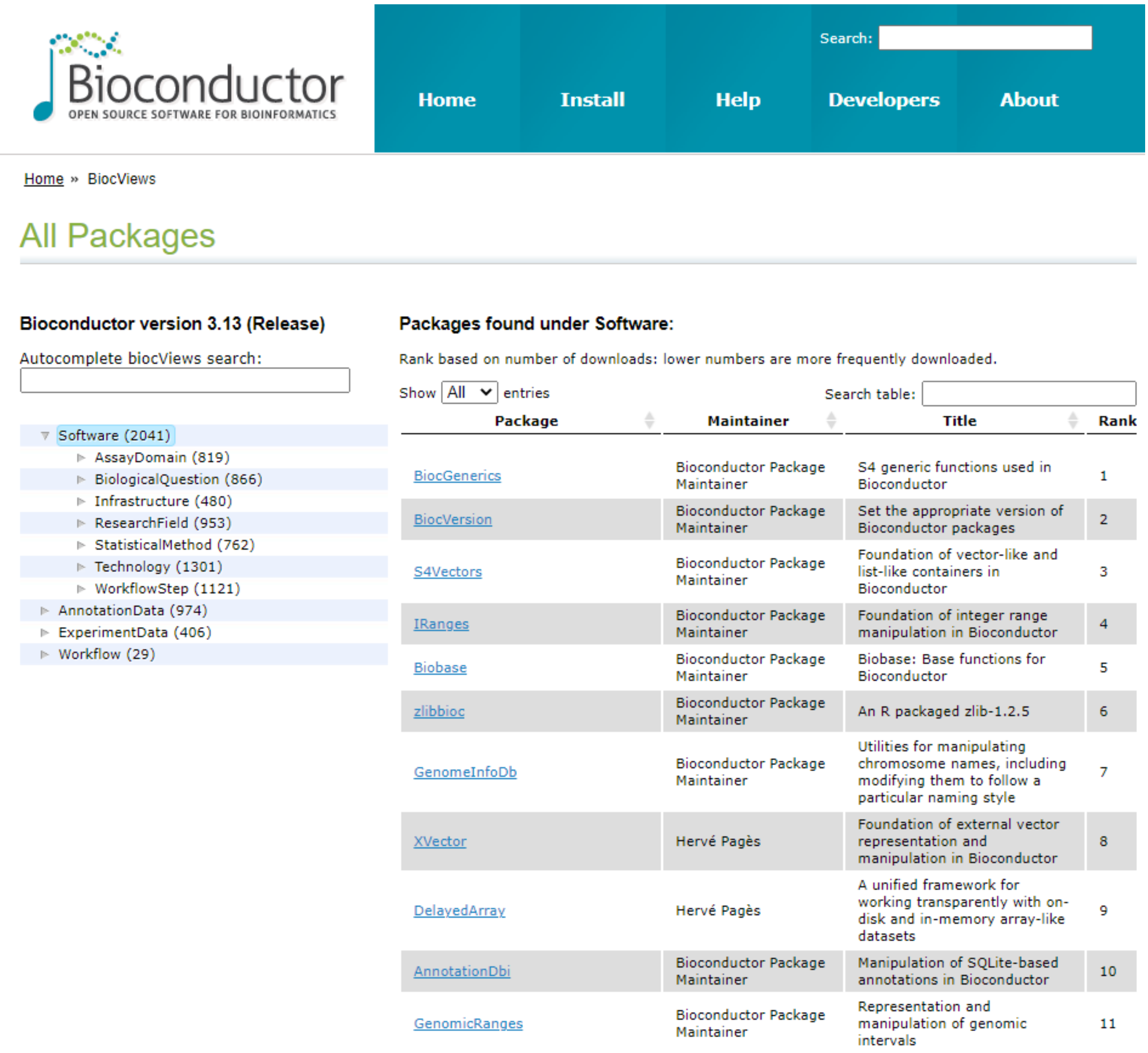
The Bioconductor core development group is developing a data structure to allow users to more conveniently handle genome-scale data. The main features of Bioconductor are as follows:

- Provides tools for management and statistical analysis of large-scale genotype data such as genome-scale sequence and expression
- Integration and management of quantitative data to identify relationships between molecular-level phenomena and phenotypic-level conditions such as growth and disease.

7.2 Packages

Main screen >> Use >> Software, Annotation, Experiment

- Software: Algorithm/tool collection for data analysis
- Annotation: Gene symbol/ID mapping, gene classification based on gene ontology, location of exons, transcripts, genes, etc. on the genome, protein function, etc. See Annotation > Packagetype
- Experiment data: Verified experimental data
- Workflow: Collection of processes for specific data analysis RNA-seq, ChIP seq, copy number analysis, microarray methylation, classic expression analysis, flow cytometry, etc.



Home » BiocViews

All Packages

Bioconductor version 3.13 (Release)

Autocomplete biocViews search:

- Software (2041)
  - AssayDomain (819)
  - BiologicalQuestion (866)
  - Infrastructure (480)
  - ResearchField (953)
  - StatisticalMethod (762)
  - Technology (1301)
  - WorkflowStep (1121)
- AnnotationData (974)
- ExperimentData (406)
- Workflow (29)

Packages found under Software:

Rank based on number of downloads: lower numbers are more frequently downloaded.

Show All entries   Search table:

Package	Maintainer	Title	Rank
BiocGenerics	Bioconductor Package Maintainer	S4 generic functions used in Bioconductor	1
BiocVersion	Bioconductor Package Maintainer	Set the appropriate version of Bioconductor packages	2
S4Vectors	Bioconductor Package Maintainer	Foundation of vector-like and list-like containers in Bioconductor	3
IRanges	Bioconductor Package Maintainer	Foundation of integer range manipulation in Bioconductor	4
Biobase	Bioconductor Package Maintainer	Biobase: Base functions for Bioconductor	5
zlibbioc	Bioconductor Package Maintainer	An R packaged zlib-1.2.5	6
GenomeInfoDb	Bioconductor Package Maintainer	Utilities for manipulating chromosome names, including modifying them to follow a particular naming style	7
XVector	Hervé Pagès	Foundation of external vector representation and manipulation in Bioconductor	8
DelayedArray	Hervé Pagès	A unified framework for working transparently with on-disk and in-memory array-like datasets	9
AnnotationDbi	Bioconductor Package Maintainer	Manipulation of SQLite-based annotations in Bioconductor	10
GenomicRanges	Bioconductor Package Maintainer	Representation and manipulation of genomic intervals	11

Annotation resources can be divided into several levels as follows.

- ChipDb: Lowest level, Affymatrix Chip information
- OrgDb: Functional annotations of specific organisms
- TxDb/EnsDb: Transcriptome information, location information
- OrganismDb: meta-packages for OrgDb, TxDb
- BSgenome Actual base information of a specific organism
- Others GO.db; KEGG.db
- AnnotationHub:
- biomaRt:

To install the package provided by Bioconductor, please install BiocManager first and then install the package. BiocManager also has a function called available() to search for available packages (with specific characters). For example, when installing a package called IRanges, enter IRanges in the Search box at the top right of bioconductor or in the search box of the software package list to find the package and perform installation as follows.

```
if (!requireNamespace("BiocManager",quietly =TRUE))
   install.packages("BiocManager")

BiocManager::install("IRanges")
# # . libPaths()
```

Exercises

1) OrganismDb has information including OrgDb, TxDb, and GO.db packages in the form of a meta-package. Find and install Homo.sapiens with human information in OrganismDB.

7.3 Learning and support

Each package has a landing page containing information such as title, author, maintainer, description, references, and installation instructions, and detailed explanations and examples are provided for the functions within the package. For an example, see IRanges' landing page. Vignettes are one of the important features of bioconductor: documents that provide detailed explanations of how to use the package with R code.

```
library(IRanges)

vignette(package="IRanges")
browseVignettes("IRanges")
vignette("IRangesOverview",package="IRanges")

ir1 <-IRanges(start=One:10,width=10:One)
ir1
class(ir1)
methods(class="IRanges")

example(IRanges)
?IRanges
??IRanges
```

Main page >> Learn >> Support site There are several related Q&As on the bulletin board, so you can get help with similar problems.

7.4 OOP - Class, Object and Method

Object-oriented programming (OOP) has been widely used since the 1990s as a way to solve the code complexity that arises when programming complex problems.

R is also an object-oriented programming language. However, R uses different and difficult concepts compared to other languages. There are various types of classes used in R, including base type, S3, S4, RC, and R6. Among these, S3 has been used a lot, and classes in the S4 format that complement the shortcomings of S3 and R6 are mainly used (AdvancedR?) . In this lecture, we will only cover S3 format classes.

There are many reasons for using classes, but it can be said that they are used to structure and easily manage complex data. The concepts you need to know are Class, Object, and Method. In fact, everything in R is an Object, and the definition of these Objects is a Class.

```
df <-data.frame(x=c(One:5),y=LETTERS[One:5])df

class(df)
```

Above, df is called a variable, but it is also an object. The class of df is data.frame. Anyone can create as many classes as they want.

```
class(df) <-"myclass"
df
class(df)

class(df) <-c("data.frame","myclass") df

class(df)
```

However, not all objects originate from OOP, for example base objects.

```
x <-One:10
class(x)
attr(x,"class")

mtcars
attr(mtcars,"class")
```

You can think of a method as a function that performs some function specialized for the classes above.

```
mt <-matrix(One:9,3,3)
df <-data.frame(One:3,4:6,7:9)

class(mt)
class(df)
str(mt)
str(df)


diamonds <- ggplot2::diamonds

summary(diamonds$carat)
summary(diamonds$cut)

methods(class="data.frame")
```

The above summary, str, etc. are methods called generic functions. To find out what information is available about the methods available for each class, use a function called methods(). For detailed information about object-oriented programming in R, see Advanced R.

Exercises

1) For the following two types of objects, create a mysummary function that calculates the average if the class is integer and the ratio (using the table function) if the class is character.

```
x <-c(One:10)
y <-c("A","G","G","T","A")
```

7.5 Bioconductor OOP

Genome-scale experiments and annotations handled by bioconductor are representative types of complex data. The OOP concepts in Bioconductor are as follows:

class - Defining the framework of a complex biological data structure object -

An entity in which a specific class is specifically implemented

method - performs a function for a specific class

For example, if we look at OrganismDb, which is a class of Homo.sapience installed earlier, it is as follows.

```
library(Homo. sapiens)
class(Homo. sapiens)
?OrganismDb
```

> The OrganismDb class is a container for storing knowledge about existing Annotation packages and the relationships between these resources. The purpose of this object and it's associated methods is to provide a means by which users can conveniently query for data from several different annotation resources at the same time using a familiar interface.

```
homo_seq <-seqinfo(Homo. sapiens)
class(homo_seq)
?Seqinfo
```

> A Seqinfo object is a table-like object that contains basic information about a set of genomic sequences. ...

```
length(homo_seq)
seqnames(homo_seq)
```

In the bioconductor, large amounts of information is structured and stored in object form, and can be read using the library() function, and the information of the object can be read using various functions.

Exercises

1) Obtain the top 10 genes and top 10 exons from Homo.sapiens information.

```
genes(Homo.sapiens)[One:10]
exons(Homo.sapiens)[One:10]
```

Chapter 8

Biostrings

8.1 Introduction

Biological sequences such as DNA or amino acids, including high-throughput sequencing data, can be analyzed by various packages of Bioconductor, and the Biostrings package is especially used as a key tool to effectively utilize biological sequences.

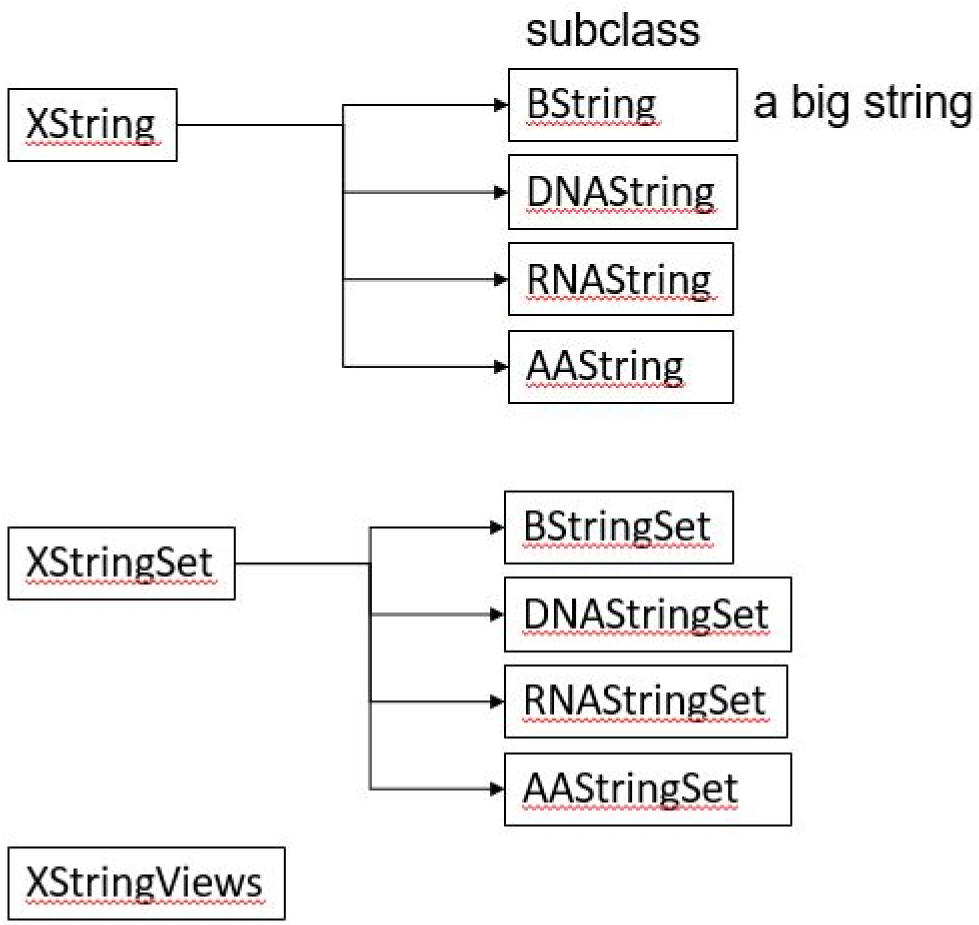
8.2 Working with sequences

Biostrings is a package that provides various functions for handling biological strings such as DNA, RNA, and amino acids. In particular, it is a package frequently used for simple sequence analysis by providing sequence comparison functions such as pattern search in sequences and Smith-Waterman local alignments and Needleman-Wunsch global alignments (sippl1999biological?). The installation method for the Biostrings package is as follows.

```
if (!requireNamespace("BiocManager",quietly =TRUE))
  install.packages("BiocManager")

BiocManager::install("Biostrings")
```

```
library(Biostrings)
```

The Biostrings package basically defines three classes: XString, XStringSet, and XStringViews. XString is a class for handling a single strand of biological sequence such as DNA, RNA, or AA, and XStringSet is a class for handling multiple strands.

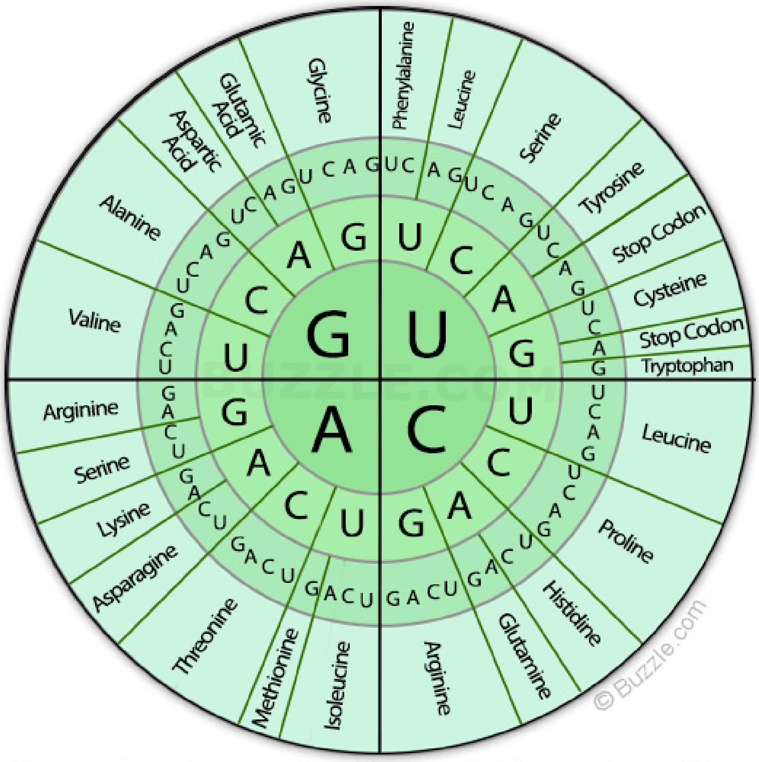


You can create an object using the DNAString function, and in addition to 'A', 'C', 'G', and 'T', '-' (insertion) and 'N' are allowed.

```
dna1 <-DNAString(“ACGT?”) dna1
<-DNAString("ACGT-N") dna1[
One]
dna1[2:3]

dna2 <-DNAStringSet(c("ACGT","GTCA","GCTA")) dna2[
One]
dna2[[One]]
dna2[[One]][One]
```

The following built-in variables are automatically saved when the Biostrings package is loaded and have predefined biological sequences. IUPAC (International Union of Pure and Applied Chemistry)

```
DNA_BASES
DNA_ALPHABET
IUPAC_CODE_MAP
GENETIC_CODE
```



To decode the codon, move from the center circle towards the periphery.

Using the above variables, you can randomly obtain a DNA sequence using the sample() function as follows. DNA_BASES is a vector with a length of 4, and to select 10 of them, replace=T must be used.

```
x0 <-sample(DNA_BASES,10,replace =T)x0

s1 <-"ATG"
s2 <-"CCC"
s3 <-paste(s1, s2,sep="")s3

x1 <-paste(x0,collapse="")x1
```

For related functions, refer to Cheat sheet

8.2.1 XString

The XString class can be divided into subclasses of DNAString, RNAString, and AAString. In the DNAString class, the length function calculates the number of nucleic acids (in DNAStringSet type variables, length is the number of DNA strands), and the number of nucleic acids can be obtained with the nchar function. toString is a function that converts the DNAString type into a simple string, and information such as complementary sequences and reverse complementary sequences can also be found using complement, reverseComplement, etc.

```
x0 <-paste(sample(DNA_BASES,10,replace =T),collapse="") x1 <-
DNAString(x0) class(x0)

class(x1)
length(x1)
(x1)
complement(x1)
Biostrings::complement(x1)
reverseComplement(x1)
```

The indexing of DNAString is the same as that of a vector (string), and the indexing of DNAStringSet is the same as that of a list.

```
# # indexing
x1[One]
x1[One:3]
subseq(x1,start=3,end=5)
subseq(x1,3,5)

# # letter frequency
alphabetFrequency(x1,baseOnly=TRUE,as.prob=TRUE)
letterFrequency(x1,c("G","C"),as.prob=TRUE)
```

Exercises

1) Create a random gene sequence with a length of 30 characters, including a start codon and a stop codon.

2) AA_ALPHABET is a built-in variable that stores the amino acid sequence alphabet defined by IUPAC. Create a random gene sequence of length 10 containing “M” and “*”.

8.2.2 XStringSet

XStringSet can also be divided into DNAStringSet, RNAStringSet, and AAStringSet, and the DNAStringSet class can be considered a set of multiple DNAStrings. The length function is the number of DNA strings, and the length of each string can be obtained with the width or nchar function. Other functions used in most DNAStrings can be used in the same way.

```
x0 <-c("CTC-NACCAGTAT","TTGA","TACCTAGAG") x1
<-DNAStringSet(x0) class(x0)

class(x1)
names(x1)
names(x1) <-c("A","B","C")
length(x1)
width(x1)
subseq(x1,2,4
)x1[[One]]
x1[One]


x3 <-DNAString("ATGAGTAGTTAG")x4
<-c(x1,DNAStringSet(x3))x4[-One]

x4
alphabetFrequency(x1,baseOnly=TRUE,as.prob=TRUE)
letterFrequency(x1,c("G","C"),as.prob=TRUE) rowSums(
letterFrequency(x1,c("G","C"),as.prob=TRUE)) subseq(x4,2,4)
```

RNA or amino acids can also be applied in the same way and converted to XStringSet using the c function.

```
x1 <-paste(sample(AA_ALPHABET,10,replace =T),collapse="") x2 <-
paste(sample(AA_ALPHABET,10,replace=T),collapse="")

x3 <-AAString(x1)x4
<-AAString(x2)

AAStringSet(c(x1, x2))
AAStringSet(c(x3, x4))
```

Exercises

1) Create a 36bp long DNA (random) sequence with a start codon and a stop codon.

2) Create 10 random sequences like above and convert them to DNAStringSet.

Below is the most intuitive way to use for. In other words, the method is to create an x0 variable with 10 storages in advance and create and store the sequences one by one by going through a for statement.

```
x0 <-rep("",10) for(i in
One:length(x0)){
  tmp <-paste(sample(DNA_BASES,30,replace =T),collapse="") x0[i]
  <-paste("ATG", tmp,"TAG",sep="")
}
x0
```

Let’s make the above code into a function. When creating random DNA, the same code is used repeatedly, only the length is different. In this case, it is convenient to use the DNA length as an input parameter to be determined by the user and create a function that receives the parameter and creates DNA.

```
data(DNA_BASES)
random_dna <- function(len){
  tmp <-paste(sample(DNA_BASES, len,replace =T),collapse="") x0 <-
  paste("ATG", tmp,"TAG",sep="") return(x0)

}
random_dna(len=30)
random_dna(len=40)
```

Please note that the len value passed as a parameter is used in len of the sample function.

Now, when repeatedly creating 10 sequences with a length of 30bp, if you run the above function 10 times repeatedly using the for statement as before, you will get the same result. If you create a function like the one above, you can reuse it at any time when creating a DNA sequence.

```
x0 <-rep("",10) for(i inOne:
length(x0)){ x0[i] <-
  random_dna(30)
}
x0
```

However, R has matrix operation functions such as apply, so you can conveniently execute repetitive statements without using the for statement. The replicate function has the same function as apply and can be used for list or vector variables. In other words, it repeatedly executes the function the user wants as follows and returns a result with a length equal to the number of repetitions.

```
x0 <-replicate(10,random_dna(30))x0

x1 <-DNAStringSet(x0)x1
```

3. Calculate the GC ratio of 10 sanitary sequences and draw a bar graph.

Let's change the above x0 strings to XStringSet, calculate the GC rate, and draw a bargraph. Since gc_ratio is a 10x2 table that stores the ratio value of G and C, think of it as representing 10 sequences and the GC ratio of each sequence on the x-axis and drawing the ratio value on the y-axis, then specify the aes and parameters of ggplot appropriately.

bar plot using ggplto2

```
x1 <-DNAStringSet(x0)
gc_ratio1 <-letterFrequency(x1,c("G","C"),as.prob=TRUE)
gc_ratio2 <-rowSums(gc_ratio1) barplot(gc_ratio2,beside=T)


names(gc_ratio2) <-paste("seq",One:length(gc_ratio2),sep="")
```

```
barplot(gc_ratio2,beside=T)

data.frame(gc_ratio2)|>
 rownames_to_column()|>
 ggplot(aes(x=rowname,y=gc_ratio2,fill=rowname))+
 geom_bar(stat="identity")+ scale_y_continuous(limits =c
 (0,One))+ scale_fill_brewer(palette ="green")+
 theme_bw()
```

8.2.3 XStringView

XStringView, another class of Biostrings, provides an interface that allows users to view the DNA, RNA, and AA sequences of the XString class as they wish. Instructions for use are as follows:

```
x2 <- x1[[One]]
Views(x2,start=One,width=20) Views(x2,
start=One,end=4) Views(x2,start=c(One,3),
end=4) Views(x2,start=c(One,3,4),width=20)
Views(x2,start=c(One,3,4),width=20) i <-
Views(x2,start=c(One,3,4),width=20)
```

As shown below, you can view several sequence fragments for one sequence, and the gaps function is a function that shows the sequence in the remaining sections excluding the section of the sequence view given as a parameter. The successiveviews function displays the number of sequences from the first sequence to the number given in the parameter width, and uses the rep() function to display the sequence from the beginning to the end.

```
v <-Views(x2,start=c(One,10),end=c(3,15))
gaps(v)

successfulViews(x2,width=20)
successfulViews(x2,width=rep(20,2))
successfulViews(x2,width=rep(20,3))
```

Exercises

1) Create a random DNA sequence of 1000bp in length and write a code to view the length in 40bp units.

You can use the random_dna() function created earlier. Since you need to use the successiveViews function, it needs to be converted to DNAString, and the number of repetitions is automatically calculated using rep() depending on the length of the sequence.

8.3 Sequence read and write

Using readDNAStringSet or writeXStringSet in the Biostrings package, you can read and write basic DNA/RNA/AA sequences and apply them to file types such as fasta and fastq.

```
x1 <-DNAStringSet(x0)
writeXStringSet(x1,"myfastaseq.fasta",format="fasta")

names(x1) <-"myfastaseq"
writeXStringSet(x1,"myfastaseq.fasta",format="fasta")

myseq <-readDNAStringSet("myfastaseq.fasta",format="fasta")
myseq
```

Multiple DNA fragments divided by successiveViews can be saved to myfastaseqs.fasta and read again.

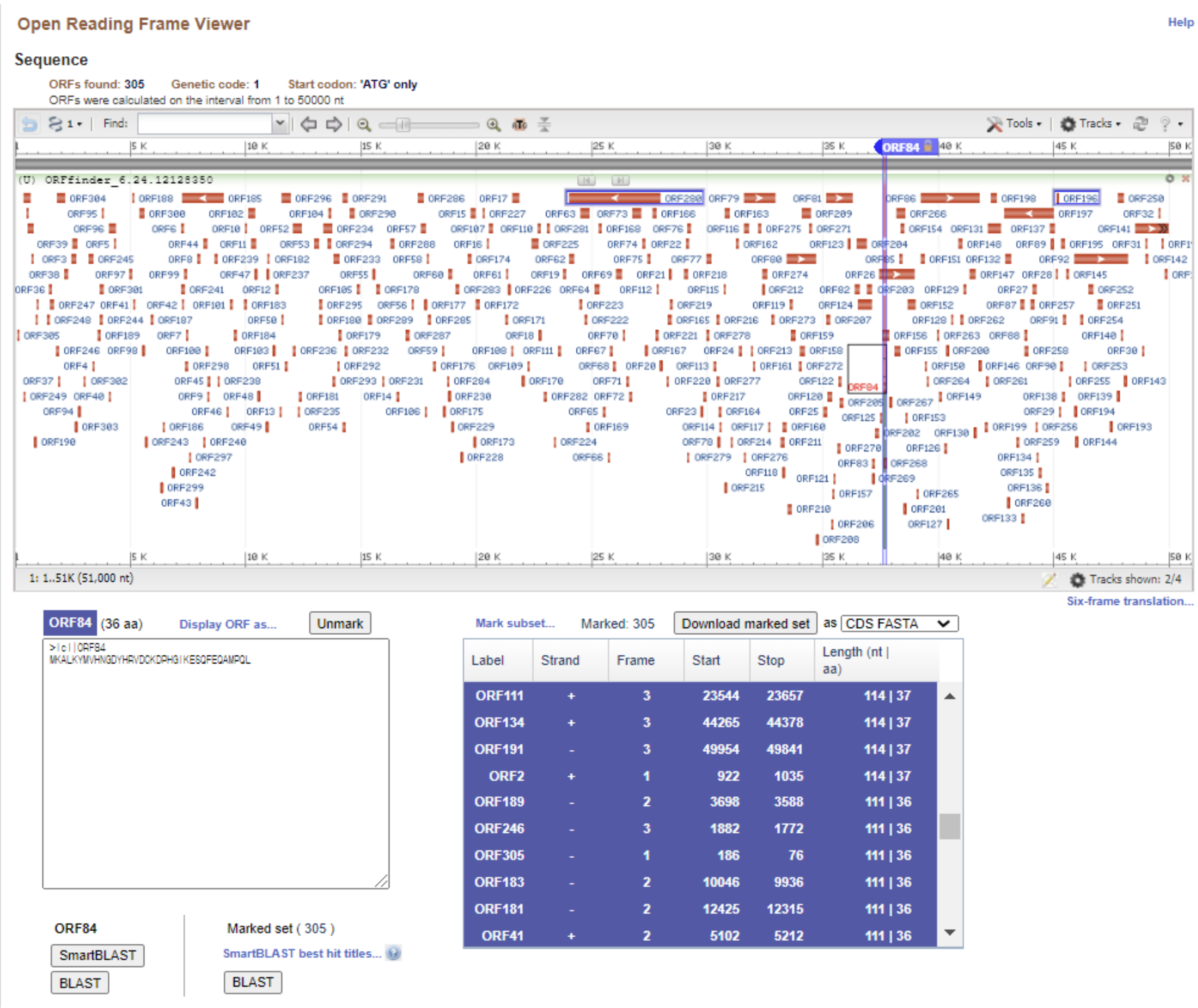
```
myseqs <-DNAStringSet(sv)
names(myseqs) <-paste("myseqs",One:length(myseqs),sep="")
writeXStringSet(myseqs,"myfastaseqs.fasta",format="fasta")
```

8.4 Sequence statistics

The oligonucleotideFrequency is a sum that counts the number of all nucleic acids in the sequence according to the options called width and step. yeastSEQCHR1, used next, is built-in data included in the Biostrings package and contains information about the first chromosome of yeast.

```
data(yeastSEQCHR1)#Biostrings
yeast1 <-DNAString(yeastSEQCHR1)

oligonucleotideFrequency(yeast1,3)
dinucleotideFrequency(yeast1)
trinucleotideFrequency(yeast1)

tri <-trinucleotideFrequency(yeast1,as.array=TRUE)tri
```

Let’s search for ORF to get amino acid information. Information about the first chromosome of yeast is annotated, but we will use the tool for learning. There are already many types of ORF search tools available, but in this lecture, we will use orfinder provided by NCBI.



Open Reading Frame Viewer

Sequence

ORFs found: 305 Genetic code: 1 Start codon: 'ATG' only
ORFs were calculated on the interval from 1 to 50000 nt

ORF84 (36 aa)

```
>lcl|ORF84
MKALKYMVHNGDYHRVDCKDPHGIKESQFEQAMPQL
```

Marked: 305 Download marked set as CDS FASTA

Label	Strand	Frame	Start	Stop	Length (nt | aa)
ORF111	+	3	23544	23657	114 | 37
ORF134	+	3	44265	44378	114 | 37
ORF191	-	3	49954	49841	114 | 37
ORF2	+	1	922	1035	114 | 37
ORF189	-	2	3698	3588	111 | 36
ORF246	-	3	1882	1772	111 | 36
ORF305	-	1	186	76	111 | 36
ORF183	-	2	10046	9936	111 | 36
ORF181	-	2	12425	12315	111 | 36
ORF41	+	2	5102	5212	111 | 36

ORF84

Marked set (305)

```
my_ORFs <-readDNAStringSet("yeast1orf.cds")
hist(nchar(my_ORFs),br=100)
codon_usage <-trinucleotideFrequency(my_ORFs,step=3)
global_codon_usage <-trinucleotideFrequency(my_ORFs,step=3,simplify.as="collapsed")

colSums(codon_usage)==global_codon_usage
names(global_codon_usage) <- GENETIC_CODE[names(global_codon_usage)]
codonusage2 <-split(global_codon_usage,names(global_codon_usage))
global_codon_usage2 <-sapply(codonusage2, sum)
```

Information about the yeast first chromosome can be found in bioconductorannotationDataOrgDb or bioconductorannotationDataTxDb.

```
# BiocManager::install("org.Sc.sgd.db")
library(org.Sc.sgd.db) class
(org.Sc.sgd.db)
?org.Sc.sgd.db
ls("package:org.Sc.sgd.db")
columns(org.Sc.sgd.db)
mykeys <-keys(org.Sc.sgd.db,keytype ="ENTREZID")[One:10]
AnnotationDbi::select(org.Sc.sgd.db,
              keys=mykeys,
              columns =c("ORF","DESCRIPTION"),
              keytype="ENTREZID")
```

TxDb

```
BiocManager::install("TxDb.Scerevisiae.UCSC.sacCer3.sgdGene")
library(TxDb.Scerevisiae.UCSC.sacCer3.sgdGene) class
(TxDb.Scerevisiae.UCSC.sacCer3.sgdGene) columns
(TxDb.Scerevisiae.UCSC.sacCer3.sgdGene) methods(class=class
(TxDb.Scerevisiae.UCSC.sacCer3.sgdGene)) ygenes <-genes
(TxDb.Scerevisiae.UCSC.sacCer3.sgdGene)
```

```
library(tidyverse)

mydat <- global_codon_usage2 |>
  data.frame |>
  rownames_to_column |>
  rename(codon ="rowname",freq =".")

ggplot(mydat,aes(x=codon,y=freq))+
  geom_bar(stat="identity")

mydat
AMINO_ACID_CODE[mydat$codon]
```

Exercises

1) Using AMINO_ACID_CODE, convert the 1st abbreviation in the above graph to 3rd abbreviation, rotate the label vertically by 90 degrees, apply y-axis label “Frequency”, x-axis label “Amino acid code”, theme option “theme_bw”, etc. Draw again (revisit ggplot2)

8.5 Pattern matching

The Biostrings package provides a matchPattern function that searches for whether a specific pattern exists in a subject sequence. If you find one pattern in multiple subject sequences, use the vmatchPattern function, and if you find multiple patterns in one subject sequence, use the matchPDict function.

```
length(coi)
hits <-matchPattern("ATG", yeast1,min.mismatch=0,max.mismatch=0) hits

class(hits)
methods(class="XStringViews")
ranges(hits)

hits <-vmatchPattern("ATG", my_ORFs,min.mismatch=0,max.mismatch=0)
stack(hits)
```