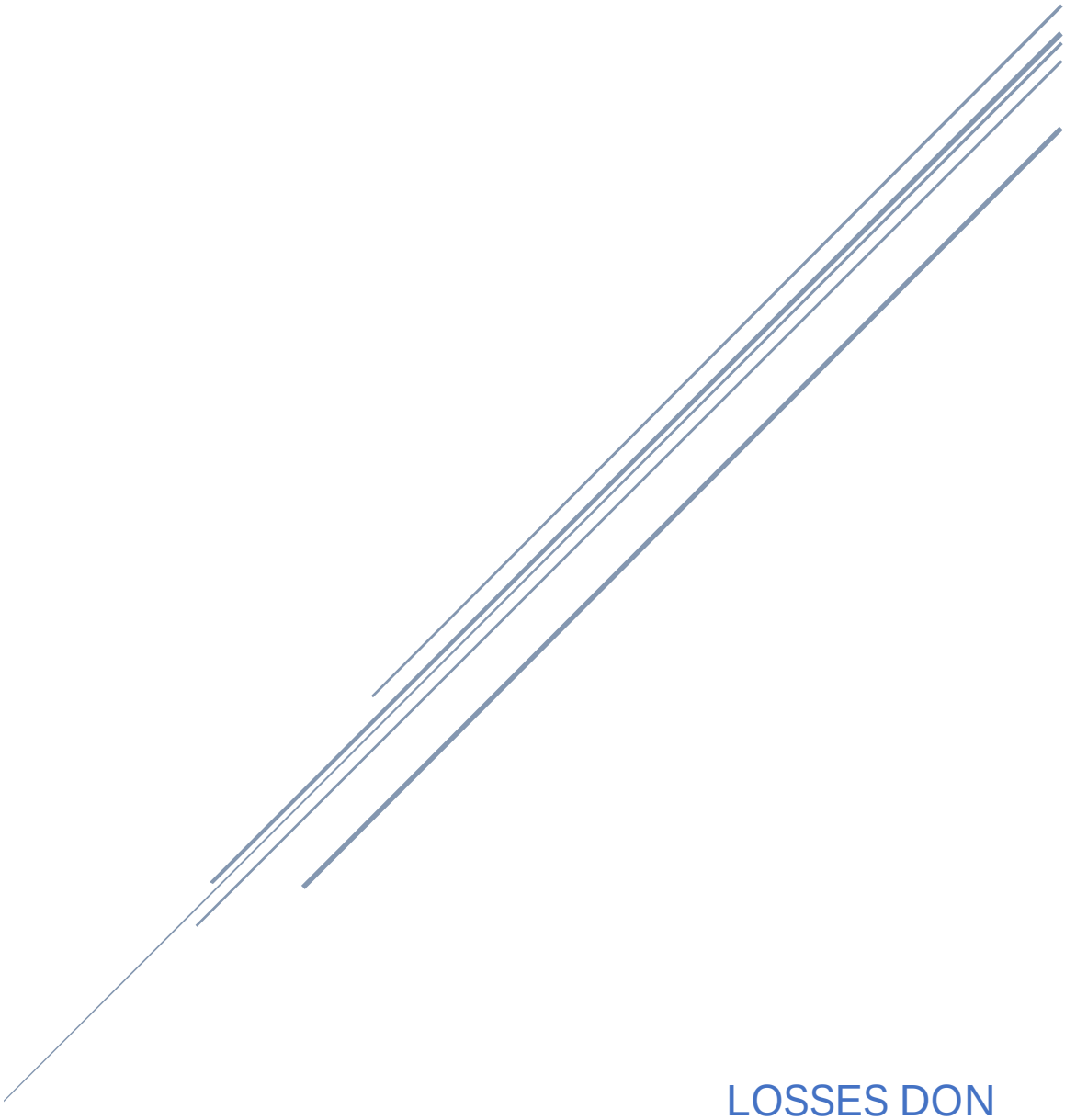
A BRIEF INTRODUCTION TO R

HANDOUT BOOK



LOSSES DON

2018/11/7

CONTENTS

CHARAPTER 0

INTRODUCTION

1. WELCOME

Welcome to ***The Brief Introduction to R***.

R is a programming language designed for statistic and analysis. In this talk, we will learn the basic concepts in R script together, we'll get in touch with how to use R script for the most basic data manipulation and plotting operations.

The tutorial code and presentation is attached, in case you need them, for any question while learning, feel free to contact me at losses@m-b.science.

2. PREPARE TOOLS AND PACKAGES

2.1 DOWNLOAD R AND R STUDIO

We could get **R** from CRAN, for reader in Asia, it's recommended to get R from Tsinghua University's mirror: https://mirrors.tuna.tsinghua.edu.cn/CRAN/.

RStudio is an IDE for R programmers, it's free and also opensource, you can get it from RStudio's official website: https://www.rstudio.com/products/rstudio/download/#download.

2.2 DOWNLOAD PACKAGES

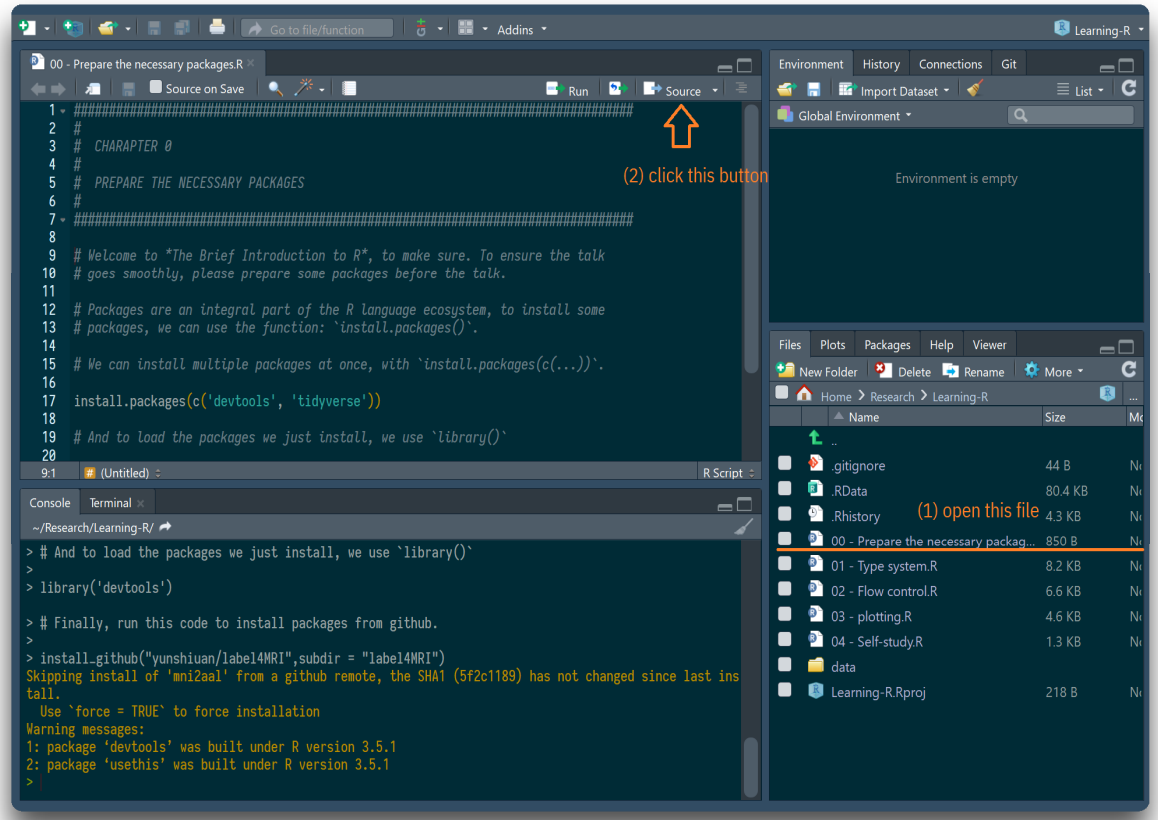
To ensure the talk goes smoothly, please prepare some packages before the talk.

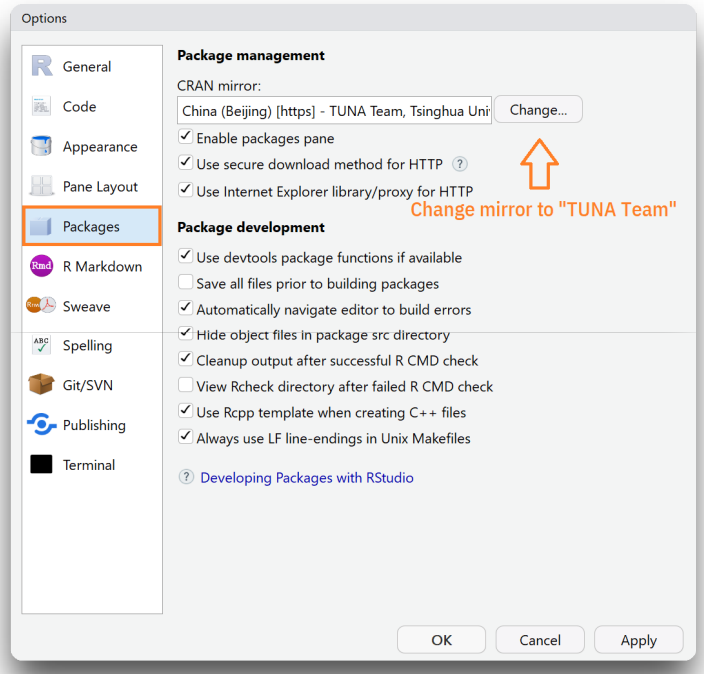
The sample code in this section corresponds to the `"00 - Prepare the necessary packages.R"` file in the **tutorial code package**, so you don't need to manually enter the code into R.
After completing the RStudio installation, open `"Learning-R.Rproj"` in the tutorial code package to start R Studio;

find `"00 - Prepare the necessary packages.R"` in the lower right side of the window, double-click to open the file;

execute the script by clicking "Source" button on the top-right corner of code window, to complete the package installation.

Chinese users may need to modify the mirror address of package repository to avoid download failures caused by Internet censorship. You can set the option by clicking tools → Global options → Packages → CRAN Mirror. For details, please refer to the screenshot.


00 - Prepare the necessary packages.R
Source on Save
Run
Source
(2) click this button
Environment
History
Connections
Git
Import Dataset
List
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
New Folder
Delete
Rename
More
Home › Research › Learning-R
Name
Size
.gitignore
44 B
.RData
80.4 KB
.Rhistory
4.3 KB
(1) open this file
00 - Prepare the necessary packag...
850 B
01 - Type system.R
8.2 KB
02 - Flow control.R
6.6 KB
03 - plotting.R
4.6 KB
04 - Self-study.R
1.3 KB
data
Learning-R.Rproj
218 B
Console
Terminal
~/Research/Learning-R/
Skipping install of 'mni2aal' from a github remote, the SHA1 (5f2c1189) has not changed since last install.
Use `force = TRUE` to force installation
Warning messages:
1: package 'devtools' was built under R version 3.5.1
2: package 'usethis' was built under R version 3.5.1


Options
General
Code
Appearance
Pane Layout
Packages
R Markdown
Sweave
Spelling
Git/SVN
Publishing
Terminal
Package management
CRAN mirror:
China (Beijing) [https] - TUNA Team, Tsinghua Uni
Change...
Enable packages pane
Use secure download method for HTTP
Use Internet Explorer library/proxy for HTTP
Change mirror to "TUNA Team"
Package development
Use devtools package functions if available
Save all files prior to building packages
Automatically navigate editor to build errors
Hide object files in package src directory
Cleanup output after successful R CMD check
View Rcheck directory after failed R CMD check
Use Rcpp template when creating C++ files
Always use LF line-endings in Unix Makefiles
Developing Packages with RStudio
OK
Cancel
Apply

Here are what we did with the script.

Packages are an integral part of the R language ecosystem, to install some packages, we can use the function: `install.packages()`.

We can install multiple packages at once, with `install.packages(c(...))`:

```
install.packages(c('devtools', 'tidyverse'))
```

And to load the packages we just install, we use `library()`:

```
library('devtools')
```

Finally, run this code to install packages from GitHub.

```
install_github("yunshiuan/label4MRI",subdir = "label4MRI")
```

CHARAPTER 1

TYPE SYSTEM

Type system is one of the most basic elements of a programming, it helps us organize and calculate data in a reasonable way.

All variable has a type, and R calls it **“Class”**. In the following sections, we will introduce how to **create, query and operate** different types of variables.

1. VECTOR

1.1 CREATING VECTOR

Vector is the most basic variable type in the world of R. Each vector contains one or multiple atomic elements, like:

```
# To inspect values:

as.numeric(gender)

# and to inspect levels, we can:

levels(gender)
```

We can create a vector using the function: `c()`, or special grammar sugar:

```
# We can create a vectpr using the function: c():

c(1, 2, 3, 4, 5)
c('CS', 'STS', 'TPJ')
c(T, F, F, T, F)

# A convinent way to create a sequence is:

1:5
```

And, we can assign the vector to a variable in the following fashion:

```
temp_var <- 1:5
c(T, F, F, T, F) -> temp_var
temp_var = c('CS', 'STS', 'TPJ')

# Inspect a variable
temp_var
```

1.2 QUERYING VECTOR

How can we get one or more elements from a vector? Here're some example:

```r
# To get a single element:

temp_var[0]

# To get multiple elements:

temp_var[1:2]
temp_var(c(1, 2))
temp_var(c(F, T, T))
```

1.3 OPERATING VECTOR

We can try some basic mathematical calculations:

```r
1 + 1    # Addition
2 - 1    # Subtraction
1 * 2    # Multiplication
1 / 5    # Division
10 %% 3  # Modulus
```

And we can operating logical vector with following operators:

```r
T & F                 # And
T | F                 # Or
c(T, T, T) & c(T, F, T)
c(T, T, F) | c(T, T, T)
```

In the world of R, we can also use operator to operate each element in a vector with the following code:

```r
c(1, 2, 3, 4, 5) + 2
1:5 * 2:6
c(F, F, T) * 2
```

2. FACTOR

Factor is a special kind of vector, we call it **compound vector**, It corresponds to the concept of **nominal variables** in statistics.

Say, there are five members in a team, and we want to list their gender, using factor is a good choice.

```r
gender <- factor(c('male', 'male', 'female', 'male', 'female'))
gender
```

Factor is made up of values and levels, to inspect them, we can run the following code:

```r
# To inspect values:

as.numeric(gender)

# and to inspect levels, we can:

levels(gender)
```

R will map levels to values and return the factor vector, it's necessary while we are facing a bunch of data, using character vector may consume a lot of memory, and potential data mistake may not easy to be checked out.

3. LIST

3.1 CREATING LIST

Why list? Try the following code:

```
c(2.2, 'Frontal Lobe', T, 1)
```

You will get a character vector, because R require each element in a vector must have the same type, if the types are not consistent, it will try to convert them, that's a pitfall while programming using R.

To solve this problem, we can use list. Now, let's create a list:

```
sub <- list(2.2, 'Frontal Lobe', T, 1)
sub <- list(oxytocin = 2.2, peak_aal = 'Frontal Lobe', in_love = T, group = 1)

sub
```

If we want to store different types of variable in a single object, we should consider using list.

Each element of a list has a name (or index) and its value, in the example above, `[[1]]` and `[['oxytocin']]` is the index and `[1] 2.2` is the value.

3.2 QUERYING LIST

We can get the **value** of each element in this fashion:

```
sub[[1]]
sub[['oxytocin']]
sub$oxytocin
sub$`oxytocin`
```

And if you want to get a sublist, we can:

```
sub['oxytocin']
sub[c('oxytocin', 'group')]
```

4. DATA FRAME

4.1 CREATING DATA FRAME

It's a quite common requirement to list different types of information in a table (Looks like the data in SPSS, SAS or Excel), data frame can help us to finish the task.

To create a data frame, we use the function: `data.frame()`:

```
research_cases <- data.frame(
  case_id = c(30, 35, 10, 12),
  bleeding = c(T, T, T, T),
  surgery = c(T, F, T, F),
  name = c('Hao', 'Si', 'Yang', 'Na')
)

research_cases
```

Now, we've created a data frame with two rows and four columns.

Data frame has some notable features:

- **LENGTH:** The length of each column must be consistent, running the following code will cause an error;

```
# FOLLOWING CODE WILL CAUSE AN ERROR!

data.frame(
  case_id = c(30, 35, 10, 12),
  bleeding = c(T, T, T),
  surgery = c(T, F, T),
  name = c('Hao', 'Si', 'Yang')
)
```

- **BROADCASTING:** If not consistent, R will try to broadcast the data automatically;

```r
data.frame(
  case_id = c(30, 35, 10, 12),
  bleeding = c(T),              # Equivalent to `c(T, T, T, T)`
  surgery = c(T, F),            # Equivalent to `c(T, F, T, F)`
  name = c('Hao', 'Si', 'Yang')
)
```

- **SPECIAL LIST:** Data frame is a kind of **compound list**, it means that data frame shares a lot of feature with list, like querying method;

```r
as.list(research_cases)
```

- **CHARACTER TO FACTOR**: The character vector will be transformed to factor while creating the data frame.

```r
class(research_cases$name)
```

sometimes you may want to keep it as character, we can:

```r
# With base
research_cases$name <- as.character(research_cases$name)

# With margrittr

library(magrittr)
research_cases$name %<>% as.character

# Let's inspect it's class now:

class(research_cases$name)
```

4.2 QUERYING DATA FRAME

To get the **value** of specific column, we can:

```
research_cases[['name']]
research_cases$name
research_cases$`name`
```

To get a sub data frame, we use single brackets. Say, if we want to get specific column or multiple columns:

```
research_cases[1]
research_cases[1:2]
research_cases[, 1]
research_cases[, 1:2]
```

To get specific columns and rows:

```
research_cases[1, 1:2]
research_cases[2:3, 1:2]
research_cases[1:3, c('surgery', 'name')]
```

To get rows:

```
research_cases[1:2, ]
```

PLEASE NOTICE THAT: If you are trying to query values from a single row, R will return its value but not a sub data frame.

```r
# Will return vectors
research_cases[1, 1]
research_cases[2:4, 1]
research_cases[2:4, 'bleeding']
```

4.3 OPERATING DATA FRAME

Sometimes, we may get data fragment from different source, to analysis the datacombining them is necessary.

To combine the data row by row, we can use `rbind()`:

```r
research_cases_1 <- data.frame(
  case_id = c(30, 35),
  bleeding = c(T, T),
  surgery = c(T, F),
  name = c('Hao', 'Si')
)
research_cases_2 <- data.frame(
  case_id = c(10, 12),
  bleeding = c(T, T),
  surgery = c(T, F),
  name = c('Yang', 'Na')
)

rbind(research_cases_1, research_cases_2)
```

To combine the data column by column, we can use `cbind()`:

```r
research_cases_1 <- data.frame(
  case_id = c(10, 12, 16),
  name = c('Yang', 'Na', 'DanDan')
)

research_cases_2 <- data.frame(
  surgery = c(F, F, T),
  bleeding = c(T, T, F)
)

cbind(research_cases_1, research_cases_2)
```

The data from different source may have different row order, to merge them correctly, we have to use `merge()`:

```r
research_cases_names <- data.frame(
  case_id = c(15, 16, 10, 12),
  name = c('MuDan', 'DanDan', 'Yang', 'Na')
)
research_cases_detail <- data.frame(
  case_id = c(10, 12, 15, 16),
  surgery = c(F, F, F, T),
  bleeding = c(T, T, F, F)
)

merge(research_cases_names, research_cases_detail, by = 'case_id')
```

For advanced data operating and cleaning, `tidyverse` may help a lot, to get detail, please visit: https://www.tidyverse.org/packages/.

CHARAPTER 2

FLOW CONTROL

1. CONDITION, LOOP, AND FUNCTION

1.1 CONDITION

If some condition meets, run a specific block of code is a common sense while coding, like other languages, R use the following structure to express the condition branch.

```r
sub_in_love <- 'inLove'

if (sub_in_love == 'inLove') {
  print('yes')
} else {
  print('no')
}

# Or

if (sub_in_love == 'inLove') {
  print('yes')
}
```

In R, we can also use the function `'ifelse'` to make it more expressive:

```r
ifelse(sub_in_love == 'inLove', 'yes', 'no')
```

1.2 BASIC LOOP

We can iterate through each element in a vector in this way:

```r
for ( i in 1:5 ) {
  print(i)
}
```

BUT PLEASE NOTICE: THIS TRAVERSAL METHOD IS VERY INEFFICIENT, IT'S HIGHLY RECOMMENDED NOT USING "FOR" IN THE CASE OF LARGE DATA OPERATIONS, LIKE fMRI DATA!

1.3 FUNCTION

Defining frequently used codes as a function is a good practice which can increase the maintainability and readability of our code, let's try defining a simple function, to say hello to someone:

```r
say_hello <- function(x) {
  paste0('Hello, ', x, '!')
}

# and call it:

say_hello('Mike')
```

2 ADVANCED LOOP

In the world of R, a more efficient way to travers variables is functional loop, there are a lot function to help us finish different type of task, the basic idea of functional loop is defining a function, the traverse function will pass each element as a parameter to the function you defined.

we'll introduce three most used functions in psychology research here:

2.1 LAPPLY

`lapply` will travers each element of the specific variable and return a list of returned result.

Here're some example:

```r
#say hello to a lot of people:

members <- c('Alice', 'Mike', 'Joe', 'Tony')
lapply(members, say_hello)

# Or report the SCL-90 scores of different members:

scores <- list(
  list(name = 'Alice', score = 90),
  list(name = 'Mike', score = 100),
  list(name = 'Joe', score = 70)
)

lapply(scores, function(x) {
  paste0('The SCL90 score of ', x$name, ' is ', x$score, '.')
})
```

2.2 SAPPLY

`sapply` is a simplified version of `lapply`, it will try to return a vector if it's possible.

```r
sapply(members, say_hello)

sapply(scores, function(x) {
  paste0('The SCL90 score of ', x$name, ' is ', x$score, '.')
})
```

Some programmers object to using this function because the type of result it returns is indeterminate (list or vector), which affects the robustness of the program. However, for some specific scenarios `sapply` is a more efficient way to solve our problem.

2.3 MAPPLY

`mapply` accepts multiple variables of equal length and sequentially passes each element of these variables to our function.

Say, if we have a lot of patients, and someone asked us to generate a list to guide the visitors to their ward.

```r
research_cases <- data.frame(
  case_id = c(30, 35, 10, 12),
  bleeding = c(T, T, T, T),
  surgery = c(T, F, T, F),
  name = I(c('Hao', 'Si', 'Yang', 'Na'))
)

query_patient_room <- function(name, bleeding, surgery) {
  room_type <- ifelse(bleeding && !surgery, 'ICU ward', 'General ward')
  paste0('Please meet ', name, ' in the ', room_type, '.')
}

mapply(
  query_patient_room,
  research_cases$name,
  research_cases$bleeding,
  research_cases$surgery
)
```

3. ADVANCED FLOW CONTROL

Before getting started to learn this section, let's load some libraries and data.

```r
# Load the library we need to use:

library(mni2aal)

# Load the data we want to use:

coordinates = read.csv('./data/MNI_coordinates.csv')

# Inspect the data to make sure R read it properly:

coordinates
```

3.1 PIPELINE OPERATOR

Pipeline operator is a very convenient way to simplify our code. Say, we want to transform the MNI coordinate to AAL region name, we need to:

```r
# Transform each coordinate to AAL region name:

aal_names_list <- mapply(
  mni_to_region_name,
  coordinates$x,
  coordinates$y,
  coordinates$z
)

# It's not enough, because is not structured, this will cause a lot of inconvinent
# while further data analysis, so let's convert the result to a data frame:

# - Rotate the list, it will turn the list to a row-wise matrix:

aal_names_long_matrix <- t(aal_names_list)

# - Turn the matrix to a data frame:

aal_names_df <- as.data.frame(aal_names_long_matrix)

# - Combine the result with original data frame:

cbind(coordinates, aal_names_df)
```

We created a lot of tempory variables, each variable consumes a certain amount of memory. While analysis a bunch of data (like 10K+), unnecessary memory consumption is very very "impressive", to avoid this, we can write the code in this way:

```r
cbind(
  coordinates,
  as.data.frame(
    t(
      mapply(
        mni_to_region_name,
        coordinates$x,
        coordinates$y,
        coordinates$z
      )
    )
  )
)
```

But wait! PARENTHESES HELL! No humans can understand what the code does, the code looks like a pasta twist!

The pipeline operator in the `tidyverse` library (or `magrittr`, a sub library of `tidyverse`) can aid this situation.

```
# Load the library:

library(tidyverse)

# Clean our code:

mapply(mni_to_region_name, coordinates$x, coordinates$y, coordinates$z) %>%
  t %>%
  as.data.frame %>%
  cbind(coordinates, .)
```

The pipeline operator will pass the calculating result on the left side to the function on the right side, if not specified, the result will appear as the first parameter, we can write a '.' on any place to change where to put the result (see the last line of example code).

CHARAPTER 3

DATA I/O AND PLOTTING

1. DATA I/O

1.1 READING DATA

We can use `read.csv()` to read data from internet or local hard disk:

```
# From internet:

bf <- read.csv('http://sc.bnu.edu.cn/learning-r/big5.csv')

# From hard disk:

bf <- read.csv('./data/big5.csv')

# and inspect the data:

bf
```

2. PLOTTING WITH GGPLOT

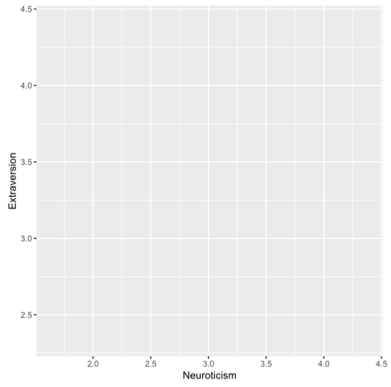
`ggplot` is a powerful tool to generate beautiful, color-blinded friendly plot for your research, let's load it firstly:

```
library(tidyverse)
# Or use `library(ggplot2)` to load it indepently.
```

2.1 DOTPLOT

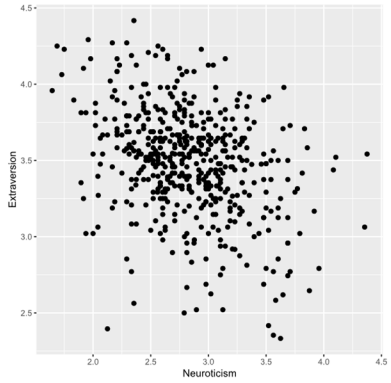
Now, Let's create a coordinate paper:

```
p0 <- ggplot(bf, aes(Neuroticism, Extraversion))
p0
#                                          #
```



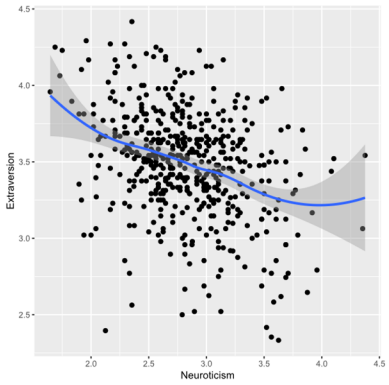
And draw some points on it:

```
p1 <- p0 + geom_point()
p1
#                                          #
```



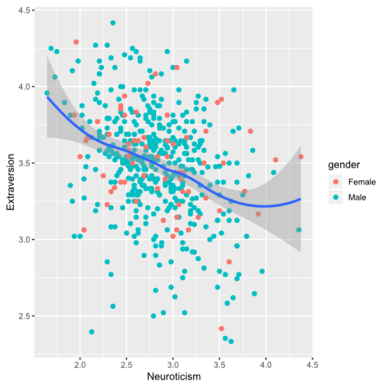
A regression line seems really pro, right?

```
p2 <- p1 + stat_smooth()
p2
#                                   #
```



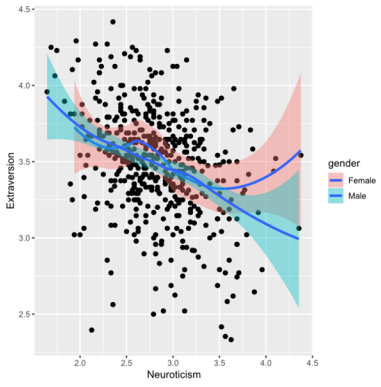
Anyone want to sperate different gender for different color?

```
p3 <- p0 +
      geom_point(aes(color = gender)) +
      stat_smooth()
p3
#                                   #
```



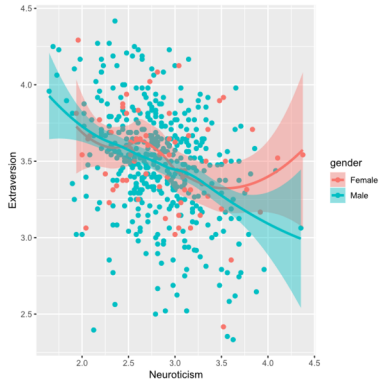
Or two regression lines?

```
p4 <- p0 +
      geom_point() +
      stat_smooth(aes(fill = gender))
p4
#                                   #
```



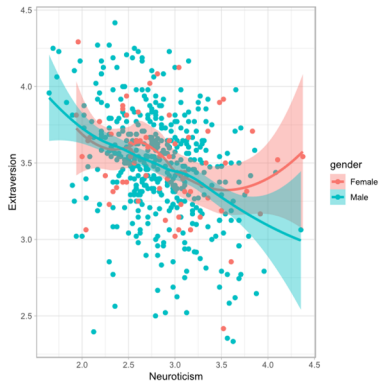
Or both?

```
p5 <- ggplot(
        bf, aes(
          Neuroticism, Extraversion,
          color = gender, fill = gender
        )
      ) +
      geom_point() +
      stat_smooth()
p5
#                                    #
```



Maybe we want to change a theme to make the plot looks cleaner:

```
p6 <- p5 + theme_light()
p6
#                                    #
```



2.2 HOW DOES IT WORKS?

While we are trying to plotting with ggplot, the first thing is creating a coordinate paper with `ggplot()`:

- The first parameter of this function is the data to be used while plotting, it should be a data frame;
- The second parameter is `mapping`, we specify which column of data is drawn on the X axis, and which column of data is drawn on the Y axis with `aes()`.
- `aes` is the abbr. of **aesthetic**, it will create a mapping option set for our plot, not only for mapping the data to which axis, but also mapping the group to different fill color, line color, or even dot size, opacity and more. To get detailed information, please read: ***Ggplot2: Elegant Graphics for Data Analysis***

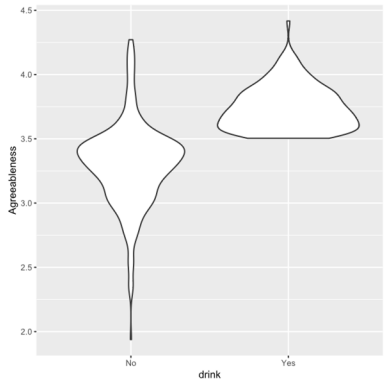
- Next, we stack a new layer above the coordinate paper, with operator `+`, it's quite intuitive. in this example, we stacked a dot plot with `geom_point()`, and a regression line with `stat_smooth()`. To know more plotting layers, please check out:
  - https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
  - https://ggplot2.tidyverse.org/

That's it, ggplot, a plotting package. In the following parts of this tutorial, we'll try to draw more different kinds of plots with it.

2.3 VIOLIN PLOT

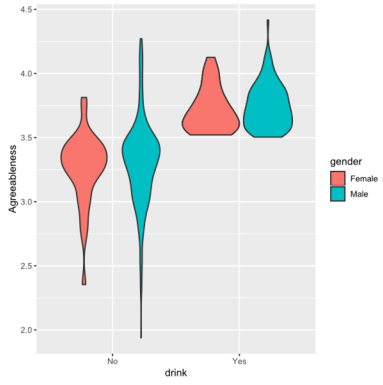
Violin plot is a convinent way to show the distribution of our data:

```
p0 <- ggplot(bf, aes(drink, Agreeableness)) +
      geom_violin()
p0
#                                          #
```



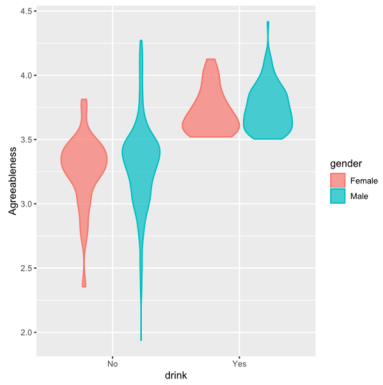
Of course, we can use `aes()` to mapping gender as color, as we used to do:

```
p1 <- ggplot(bf, aes(drink, Agreeableness)) +
      geom_violin(aes(fill = gender))
p1
#                                          #
```



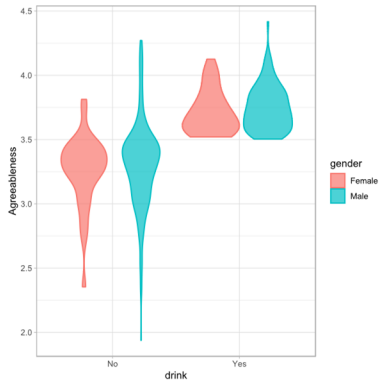
And we can beautify the plot by specifying opacity (alpha), and border color:

```
p2 <- ggplot(bf, aes(drink, Agreeableness)) +
      geom_violin(
        alpha = 0.7,
        aes(fill = gender, color = gender)
      )
p2
#                                          #
```



Finally, a theme if you want:

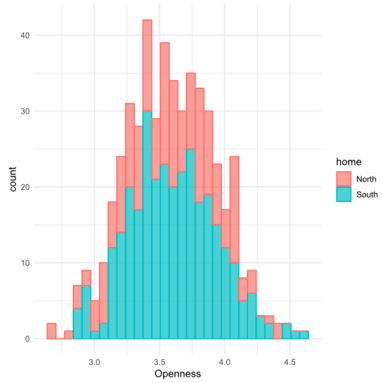
```
p3 <- p2 + theme_light()
p3
#                                          #
```



2.4 HISTOGRAM

Another way to show the distribution of our data is histogram:

```
ggplot(bf, aes(Openness, color = home)) +
  geom_histogram(
    alpha = 0.7,
    aes(fill = home)
  ) +
  theme_minimal()
#                                          #
```

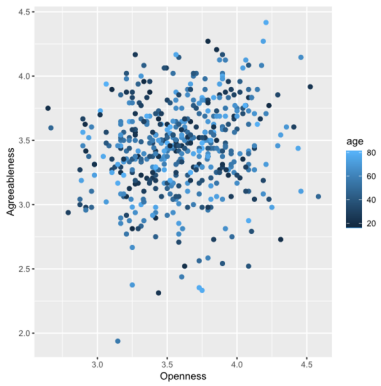


2.5 MULTIPLOT

Sometimes, we may need to paint different parts of data into different plot ggplot2 has two function `facet_grid` and `facet_wrap`, helping us finish this task, they have different using scenario but we'll only introduce `facet_grid` today.

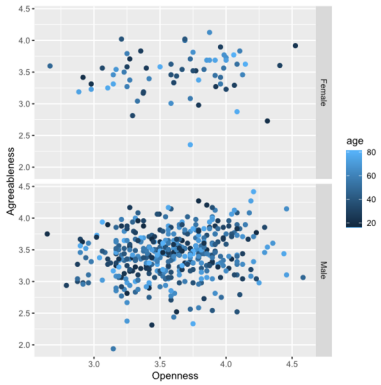
Firstly, let's create a dotplot, different color of the dots represents different age:

```
p0 <- ggplot(bf, aes(Openness, Agreeableness)) +
      geom_point(aes(color = age))
#                                              #
```



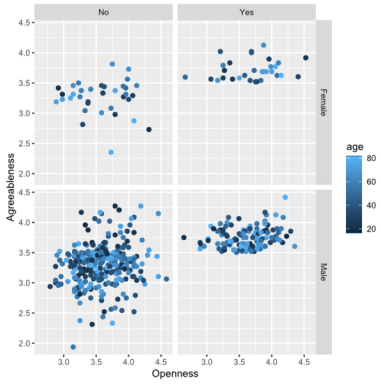
Next, if we want to plot different gender to different plot, we can:

```
p0 + facet_grid(rows = vars(gender))
#                                              #
```



Finally, to inspect whither there're some differences among drinking people and no drinking people, we can:

```
p0 + facet_grid(
  rows = vars(gender),
  cols = vars(drink)
)
#                                              #
```



CHARAPTER 4

SELF-STUDY

1.GETTING HELP

R have a build in document system, to know how to use some function or operator, we can use the question mark:

```
# Document of the function or operator
?sapply
?`+`

# Document of a package
??tidyverse
```

2. RESOURCE

2.1 BOOKS

Here are some books for R beginners:

- ***Learning R: A Step-by-Step Function Guide to Data Analysis*** by Richard Cotton
- ***Ggplot2: Elegant Graphics for Data Analysis*** by Hadley Wickham
- ***R in Action: Data Analysis and Graphics with R*** by Robert Kabacoff
- ***R Graphics 2nd Edition*** by Paul Murrell

2.2 SITES

- ***R-bloggers***: https://www.r-bloggers.com/
- ***Tidyverse***: http://tidyverse.org/
- ***Cheatsheets***: https://github.com/rstudio/cheatsheets

CREDICT

Font: IBM Plex Sans, IBM Plex Serif, Iosevka

Code Snippet: Carbon

Typography tool: Microsoft Word 2016

Made by Losses Don with ❤.