Review of the course "R for Data Science" Part 01(Talk 01~04)

By Haoran Nie @ HUST Life ST

Multi-omics data analysis and visualisation, #1

> Talk 01

This section has nothing to explain :)

R language basics, part 1

> Talk 02

Fundamental Data Type

The most basic data types include **numbers**, **logical symbols** and **strings** and are the basic building blocks of the other data types.

Simple Data Types

This includes vectors and matrices, both of which can contain multiple values of a certain basic data type, such as a **matrix** consisting of multiple numbers, a **vector** consisting of multiple strings, and so on. However, **they can only contain a single data type.**

```
c(100, 20, 30) ## Interger vector
c("String", "Array", "It's me".) ## String vector
c(TRUE, FALSE, TRUE, T, F) ## A logic vector
```

As shown above, arrays are usually defined with the function `c()`. In addition, a `vector` containing consecutive integers can be defined using the `:` operator.

Conversion between data types

1. Automatic Conversion

   A `vector` can contain only one basic data type. Therefore, when defining arrays, if the input values are mixed, certain basic data types are automatically converted to other types to ensure consistency of the numeric types; this is called `coerce` in English, and has the meaning of forced conversion. The priority of this conversion is:

   - Logical types -> numeric types
   - Logical Type -> String
   - numeric type -> string

2. Manual switchover

   In addition to the automatic conversion, you can manually convert the types of the elements in a vector:

- Checking the type of a variable `class()`
- Checking of classes `is.type()`
- Conversion of classes `as.type()`

Some special values in matrices

- `NA` (Not Available) missing values
- `NaN` (Not a Number)is meaningless
- `-Inf` Negative Infinity
- `Inf` Positive Infinity
- `NULL` Null

Some functions to determine these special values:

- `is.na()`
- `is.finite()`
- `is.infinite()`

Vectors and Arrays

Both are arrays. A `vector` is a one-dimensional array and a matrix is a two-dimensional array. This means.

- There can be more dimensional arrays
- High-dimensional arrays, like `vector` and matrices, can contain only one basic data type.
- Higher dimensional arrays can be defined by the `array()` function.

Vector maniulation

```
dim(m);
nrow(m);
ncol(m);
range(m); ## Available when the content is numeric
summary(m); ## Can also be used in vector
```

Extra:

- Incorporation `ab = c(a, b)`
- Take part `ab[1]`
- Replacement of individual values `ab[1] = c`
- Replacing multiple values `ab[c(2, 3)] = c("Weihua", "Chen")`
- Naming elements and replace values `names(ab) = as.character(ab)`
- Reverse `rev(1:10)`

- Sort&order

```
lts = sample(LETTERS[1:20])
sort(lts)
```

- Fetch one line or multiple lines

```
# (There's already some data in workspace)

$ m
> (List the content of matrix "m")

$ m[1, ]
> (List the first row of matrix 'm')

$ m[1:2, ]
> (List the first two rows of matrix 'm')
```

  You can also let the console to fetch multiple lines as the order you give.

  `m[c("row_B", "row_A")]`

  The console will output the contents of matrix "m" in the order of "row_B" and then "row_A".

- Fetch one column or multiple columns

  As can be seen from the same principle, I only list codes here

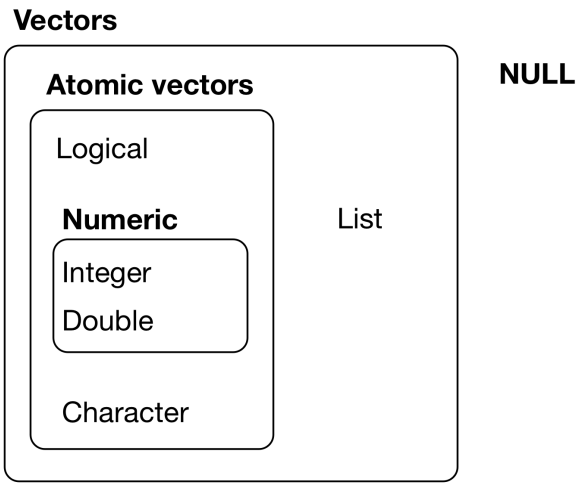
```
m[ , 1]
m[ , c(1:2)]
m[ , c("col_B", "col_A")]
```

- Fetch parts `m[1:2, 2:3]`
- Replacement

```
m[1, ] = c(10)
m[, "C"] = c(230, 140)
m[1:2,] = matrix( 1:6, nrow=2)
m[1, c("C", "B")] = matrix(110:111, nrow = 1)
```

- Transparent `t(m)`

The hierarchy of R's vector types



You can use function `typeof()` to know the type of a vector.

Here are some examples of other `is.xxx()` function:

```
is.null( NULL )
is.numeric( NA )
is.numeric( Inf );
is.list(); # This is a function which can take the place of "typeof()"
is.logical()
is.character()
is.vector();
# more ...
```

R language basics, part 2

Talk 03

`data.frame`

What is a `data.frame`?

```
library(tidyverse);
library(kableExtra)
kbl(head(mpg),
    booktabs = T)
```

Here's the result:

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact

Usage of `head()` and `tail()`

- `head()` is a function to display the first rows of some data (vectors etc.)
- `tail()` is a function to display the last rows of some data (vectors etc.)

Components of `data.frame` and common functions

Components:

- Two-dimensional table
- consists of different columns; each column is a vector, different columns can have different data types, but a column contains only one data type (`int`, `num`, `chr` ...)
- Each column has the same length

Common functions:

```r
nrow() # Show the number of rows
ncol() # Show the number of columns
dim()  # Show the dimension
```

Structure of `data.frame` & `tibble`

`str(mpg)`

This command shows the structure of the tibble `mpg`:

```
## tibble [234 x 11] (S3: tbl_df/tbl/data.frame)
##  $ manufacturer: chr [1:234] "audi" "audi" "audi" "audi" ...
##  $ model       : chr [1:234] "a4" "a4" "a4" "a4" ...
##  $ displ       : num [1:234] 1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int [1:234] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int [1:234] 4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr [1:234] "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr [1:234] "f" "f" "f" "f" ...
##  $ cty         : int [1:234] 18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int [1:234] 29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr [1:234] "p" "p" "p" "p" ...
##  $ class       : chr [1:234] "compact" "compact" "compact" "compact" ...
```

Make a new `data.frame`

You can use the function `data.frame()` to make a new `data.frame`

```r
data2 =
        data.frame(
          data = sample(1:100, 10),
    group = sample(LETTERS[1:3], 10, replace = TRUE)
    data2 = 0.1
  )
```

How to add row(s)/col(s) to an existing data.frame

Create the "table header" first, then populate the `data.frame`

```r
df2 =
        data.frame(
    x = character(),
    y = integer(),
    z = double() ,
    stringsAsFactors = FALSE
  )

df2 =
        rbind(
    df2,
    data.frame(
      x = "a",
      y = 1L,
      z = 2.2
    )
  )

df2 =
```

```
    rbind(
  df2,
  data.frame(
    x = "b",
    y = 2,
    z = 4.4
  )
)
```

ATTENTION

- Use `rbind()` function to add rows, use `cbind()` function to add columns.
- Define the new line using `data.frame()` function, the "header" needs to be the same as the merged table.

You can also use these functions to bind several data.frames.

`tibble`

`tibble` is kind of similar to `data.frame`.

Make new `tibble`

`tibble` related functionality is provided by the `tibble` or `tidiverse` packages.

Almost all of the functions that you'll use in this book produce tibbles, as tibbles are one of the unifying features of the tidyverse. Most other R packages use regular data frames, so you might want to coerce a data frame to a tibble. You can do that with `as_tibble()`:

```
as_tibble(iris)
#> # A tibble: 150 × 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>
#> 1          5.1         3.5          1.4         0.2 setosa
#> 2          4.9         3            1.4         0.2 setosa
#> 3          4.7         3.2          1.3         0.2 setosa
#> 4          4.6         3.1          1.5         0.2 setosa
#> 5          5           3.6          1.4         0.2 setosa
#> 6          5.4         3.9          1.7         0.4 setosa
#> #   144 more rows
```

Another way to create a tibble is with `tribble()`, short for **tr**ansposed tibble. `tribble()` is customised for data entry in code: column headings are defined by formulas (i.e. they start with `~`), and entries are separated by commas. This makes it possible to lay out small amounts of data in easy to read form.

```
tribble(
  ~x, ~y, ~z,
  #--/--/----
  "a", 2, 3.6,
  "b", 1, 8.5
)
#> # A tibble: 2 × 3
#>   x         y     z
```

```
#>   <chr> <dbl> <dbl>
#> 1 a         2   3.6
#> 2 b         1   8.5
```

- `add_row()`
- `add_column()`

Manipulate the tibble

> See "Manipulate the `data.frame`"

`tibble` to `data.frame`

- `as.data.frame()`
- `as_tibble()`

e.g.

```
library(tibble)
as.data.frame(head(as_tibble(iris)))
```

Differences between tibble and data.frame

Tibble evaluates columns sequentially

```
rm(x,y) # Delete possible x, y
tibble(x = 1:5, y = x^2); # You can do this with tibble
data.frame(x = 1:5, y = x ^ 2); # But data.frame doesn't work.
```

`data.frame` causes trouble when fetching `subset` operations

```
df1 =
        data.frame(x = 1:3, y = 3:1)
class(df1[, 1:2])
```

```
#> [1] "data.frame"
```

```
# Subset operation :takes a column and expects a data.frame ()
class(df1[, 1]) # The result is a vector ...
```

```
#> [1] "integer"
```

```
## Tibble doesn't.
df2 =
        tibble(x = 1:3, y = 3:1)
class(df2[, 1]) ## Tibble forever
```

```
#> [1] "tbl_df" "tbl" "data.frame"
```

`tibble` allows controlled data type conversion

> There's no proper example here.
> :_(

Recycling

```r
data.frame(a = 1:6, b = LETTERS[1:2]) # data.frame CAN!!!
```

OUTPUT

```
#   a b
# 1 1 A
# 2 2 B
# 3 3 A
# 4 4 B
# 5 5 A
# 6 6 B
```

```r
tibble(a = 1:6, b = LETTERS[1:2]); ## But tibble CAN'T!!!
```

OUTPUT

```
# Error:
# ! Tibble columns must have compatible sizes. ## * Size 6: Existing data.
# * Size 2: Column `b`.
#   Only values of size one are recycled.
```

ATTENTION!

The recycling of `tibble` is limited to lengths of 1 or equal; `data.frame` is just divisible.

`data.frame` will do partial matching, while `tibble` will NEVER do it.

```r
df = data.frame(abc = 1)
df$ab; # Unwanted result ...

df2 = tibble(abc = 1)
df2$a; # Produce a warning and return NULL
```

OUTPUT

```
# Warning: Unknown or uninitialised column: `a`.
# NULL
```

Advanced tips for using `data.frame` and `tibble`

- `attach()`
- `detach()`
- `with()`
- `within()`

Following is the introduction (Produced by ChatGPT)

These functions—`attach()`, `detach()`, `with()`, and `within()`—are incredibly useful when working with data frames or tibbles in R, aiding in smoother workflows and code readability. Here's a breakdown of their functionality:

`attach()` and `detach()`

- **Purpose**: These functions allow you to temporarily attach a data frame to the search path, making its columns directly accessible by their names.
- **Usage**:
  - `attach(df)` attaches the specified data frame `df`.
  - `detach(df)` detaches the specified data frame `df`.
- **Example**:

```r
data(mtcars) # Loading a sample dataset
attach(mtcars) # Attaching mtcars

# Now, columns can be accessed directly
summary(mpg)
mean(mpg)

detach(mtcars) # Detaching mtcars
```

- **Note**: While convenient, using `attach()` can sometimes lead to confusion or unintended consequences, such as masking variables in your environment. It's often recommended to avoid using `attach()` due to potential side effects.

`with()`

- **Purpose**: `with()` allows you to execute expressions in an environment where the data frame's columns can be referenced without using `$`.
- **Usage**:
  - `with(data, expr)` evaluates `expr` in the context of the specified data frame `data`.
- **Example**:

```r
data(mtcars) # Loading a sample dataset

with(mtcars, {
  mean(mpg)
  summary(cyl)
  })
```

- **Advantage**: It helps avoid repetitive use of the data frame name while working with its columns.

`within()`

- **Purpose**: Similar to `with()`, `within()` allows modification of a data frame by evaluating expressions within it.
- **Usage**:
  - `within(data, expr)` modifies `data` according to `expr` and returns the modified data frame.
- **Example**:

```r
data(mtcars) # Loading a sample dataset

modified_mtcars <- within(mtcars, {
  mpg_square <- mpg^2
  hp_doubled <- hp * 2
  })
head(modified_mtcars) # Checking the modified data frame
```

- **Advantage**: `within()` is useful when you want to create or modify columns within the data frame without having to repeatedly refer to the data frame name.

Remember, while these functions can streamline your code, it's crucial to use them judiciously to avoid unexpected behavior or cluttering your global environment.

Here's the console print output for the examples provided earlier:

```r
# Using attach() and detach()
data(mtcars) # Loading a sample dataset
attach(mtcars) # Attaching mtcars

# Now, columns can be accessed directly
summary(mpg)
# Output:
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  10.40   15.43   19.20   20.09   22.80   33.90

mean(mpg)
# Output:
# [1] 20.09062

detach(mtcars) # Detaching mtcars

# Using with()
data(mtcars) # Loading a sample dataset

with(mtcars, {
  mean(mpg)
  # Output:
  # [1] 20.09062

  summary(cyl)
  # Output:
  #   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  #   4.00   4.00    6.00    6.188   8.00    8.00
})

# Using within()
data(mtcars) # Loading a sample dataset

modified_mtcars <- within(mtcars, {
  mpg_square <- mpg^2
  hp_doubled <- hp * 2
})
head(modified_mtcars) # Checking the modified data frame
# Output:
#                   mpg cyl disp  hp drat    wt  qsec vs am gear carb mpg_square hp_doubled
# Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4     441.00       220
# Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4     441.00       220
```

```
# Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1     519.84       186
# Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1     457.96       220
# Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2     349.69       350
# Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1     327.61       210
```

File IO

Read from files

Using functions from the `readr` package

```
# readr is part of tidyverse
library(tidyverse) # or alternatively
library(readr)
```

Some available functions:

- `read_csv()`: comma separated (CSV) files
- `read_tsv()`: tab separated files
- `read_delim()`: general delimited files
- `read_fwf()`: fixed width files
- `read_table()`: tabular files where columns are separated by white-space. read_log(): web log files

  Full documentation of the package is available through https://www.rdocumentation.org/packages/readr/versions/1.3.1.

Usage

- Read with predifined column types

```
myiris2 =
        read_csv("../data/talk03/iris.csv",
           col_types =
                   cols(
              Sepal.Length = col_double(),
              Sepal.Width = col_double(),
              Petal.Length = col_double(),
              Petal.Width = col_double(),
              Species = col_character()
            )
          )
```

- To read from other formats, you can try the following packages:

  Similar to Python

  - `haven` - SPSS, Stata, and SAS files
  - `readxl` - excel files (.xls and .xlsx) DBI - databases
  - `jsonlite` - json
  - `xml2` - XML
  - `httr` - Web APIs
  - `rvest` - HTML (Web Scraping)

Write to files

Use the following functions to write object(s) to external files:

> Default parameters are listed.
>
> More related documents can be found in https://r4ds.had.co.nz/data-import.html?q=file#writing-to-a-file.

- Comma delimited file:

```
write_csv(
  x,
  path,
  na = "NA",
  append = FALSE,
  col_names = !append
)
```

- File with arbitrary delimiter:

```
write_delim(
  x,
  path,
  delim = " ",
  na ="NA",
  append = FALSE,
  col_names = !append
)
```

- CSV for excel:

```
write_excel_csv(
  x,
  path,
  na = "NA",
  append = FALSE,
  col_names = !append
)
```

- String to file:

```
write_file(
  x,
  path,
  append = FALSE
)
```

- String vector to file, one element per line:

```
write_lines(
  x,
  path,
  na = "NA",
  append = FALSE
)
```

- Object to RDS file:

```
write_rds(
  x,
  path,
  compress =
          c(
      "none",
      "gz",
      "bz2",
      "xz"
    ),
  ...
)
```

- Tab delimited files:

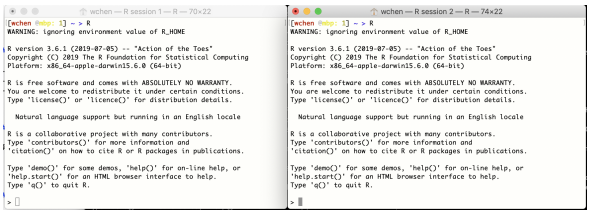
```
write_tsv(
  x,
  path,
  na = "NA",
  append = FALSE,
  col_names = !append
)
```

R language basics, part 3: factor

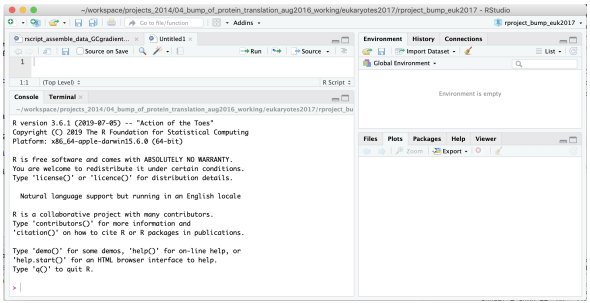
Talk 04

IO and working enviroment management

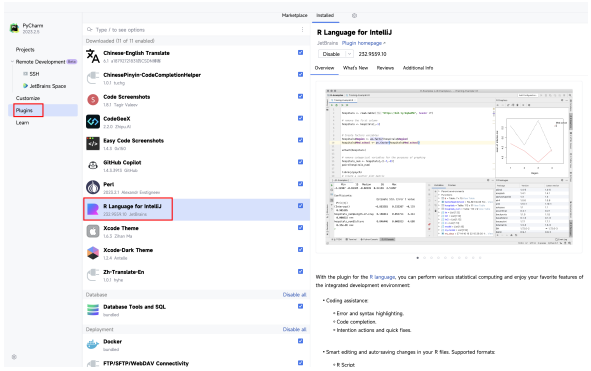
Each R session is a separate **work space** containing its own data, variables, and operation history.



Each RStudio session is automatically associated with a R session



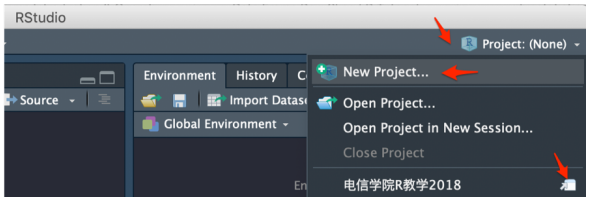
If you want to coding with R using PyCharm or other JetBrains IDE (i.e. IntelliJ, CLion, etc.), remember to install the *R Language Plug-in*.



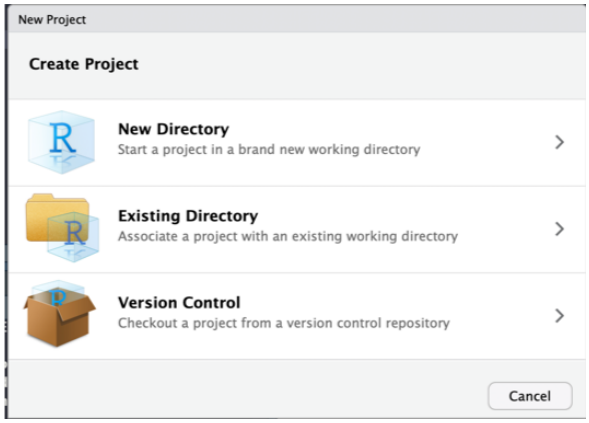
Start a new RStudio session by creating a new project

To start a new session in PyCharm, simply press the bottom corner and select a new session.

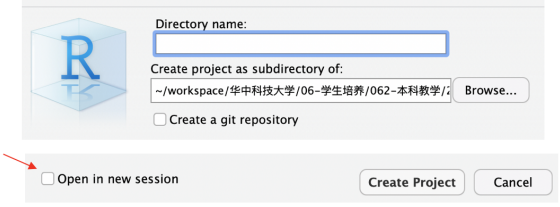
- Click the Project button in the upper right corner and select New Project in the pop-up menu ...



- Select: New directory -> New Project in the popup window



- Enter a new directory name, choose its mother directory ...



Working Space

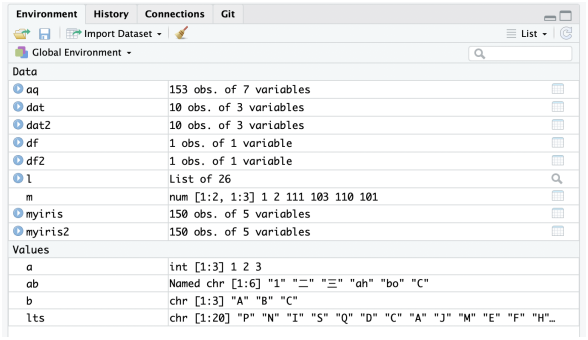
Current workspace, including all loaded data, packets and homebrew functions.

Variables can be managed with the following code:

```
ls() # Show all the variables in current workspace/session
rm(x) # Remove a variable
rm(list = ls()) # Remove ALL variables in current workspace/session
```

Variables in working space in RStudio

The "Environment" window in the upper right corner of RStudio shows all the variables of the current workspace.



Save and restore work space

```
# Save all loaded variables into an external .RData file
save.image(file = "prj_r_for_bioinformatics_aug3_2019.RData")
# Restore (load) saved work space
load(file = "prj_r_for_bioinformatics_aug3_2019.RData")
```

Notes:

- Existing variables will be kept, however, those will the same names will be replaced by loaded variables
- Please consider using `rm(list=ls())` to remove all existing variables to have a clean start
- You may need to reload all the packages

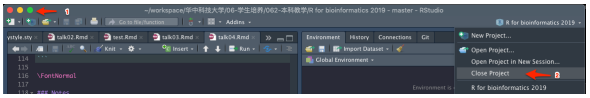
Save selected variables

Sometimes you need to transfer processed data to a collaborator ...

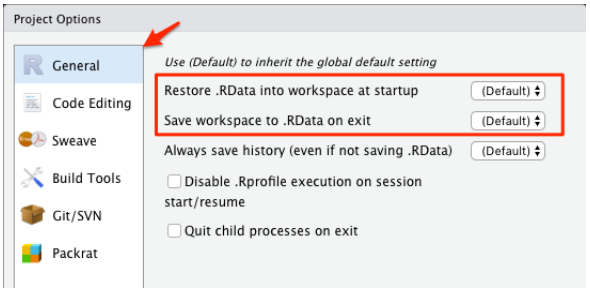
```
# Save selected variables to external
save(
  city,
  country,
  file="1.RData"
)
# You can specify directory name
load("1.RData")
```

Close and (re)open a project

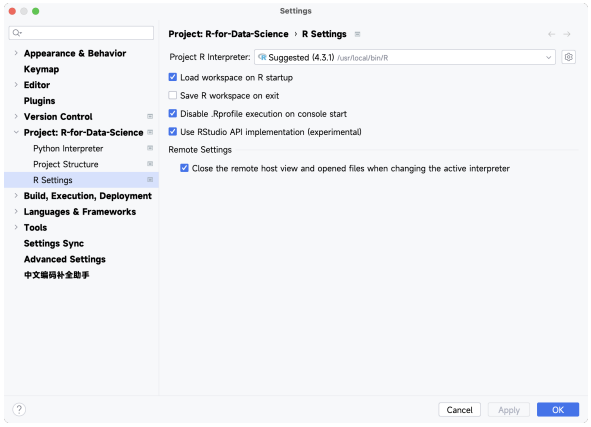
- To close a project



- In RStudio and similar IDEs, there are some preferences to choose



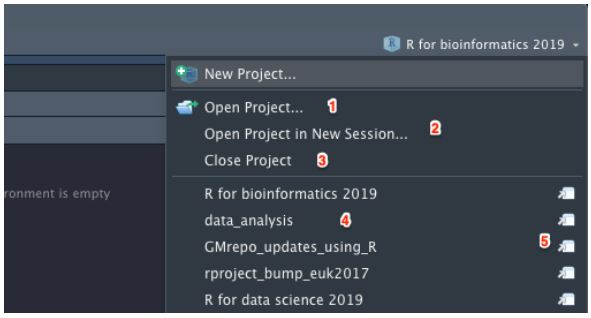
The UI in PyCharm



Notes:

- Save on exit
- Load on opening
- When the data is large, the loading time may be too long ...

Open a project



When in PyCharm, simply drag the working directory to its main window, remember to trust the project.

Factors

Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data).

It will limit the selection of input data.

Play around with `levels()`

Here are instructions of modifying factor levels

> Based on the textbook

The levels are terse and inconsistent. Let's tweak them to be longer and use a parallel construction. Like most rename and recoding functions in the tidyverse, the new values go on the left and the old values go on the right:

```
load(gss_cat)

mutate(
  partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat"
  )
)

count(partyid)
```

```
#> # A tibble: 10 × 2
#>   partyid                  n
#>   <fct>                <int>
#> 1 No answer              154
#> 2 Don't know               1
#> 3 Other party            393
#> 4 Republican, strong    2314
#> 5 Republican, weak      3032
#> 6 Independent, near rep 1791
#> #  4 more rows
```

Use this technique with care: if you group together categories that are truly different you will end up with misleading results.

The order of the `levels` determines the sorting order.

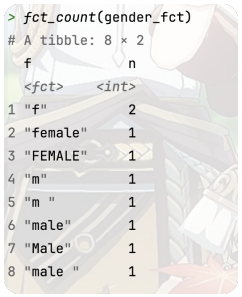
Use factor to clean data

> Usage of `fct_xxx()` functions.

Suppose I have a set of gender data that is written in a very irregular way:

```r
gender =
        c("f", "m ", "male ","male", "female", "FEMALE", "Male", "f", "m")

gender_fct =
  as.factor(gender)

fct_count(gender_fct)
```

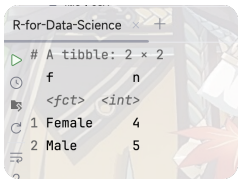
The output looks like this:



```
> fct_count(gender_fct)
# A tibble: 8 × 2
  f             n
  <fct>     <int>
1 "f"           2
2 "female"      1
3 "FEMALE"      1
4 "m"           1
5 "m "          1
6 "male"        1
7 "Male"        1
8 "male "       1
```

Now I request to replace with Female, Male.

```r
gender_fct =
  fct_collapse(
    gender,
    Female = c("f", "female", "FEMALE"),
    Male = c("m ", "m", "male ", "male", "Male")
  )

fct_count(gender_fct)
```



```
R-for-Data-Science
# A tibble: 2 × 2
  f           n
  <fct>   <int>
1 Female      4
2 Male        5
```

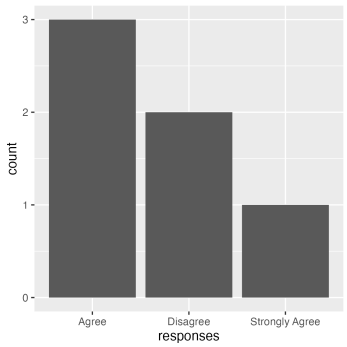
You can also use `fct_relabel()` to do the same thing

```r
fct_relabel(
  gender,
  ~ ifelse(
    tolower(
      substring(., 1, 1)) == "f",
    "Female",
    "Male"
  )
)
```

Usage of factors in drawing plots

```r
library(ggplot2)

responses =
```

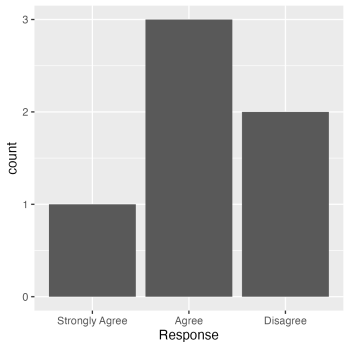
```
  factor(
    c("Agree", "Agree", "Strongly Agree", "Disagree", "Disagree", "Agree")
  )

response_barplot =
  ggplot(
    data = data.frame(responses),
    aes(x = responses)
  ) +
  geom_bar()
```



By default, `factor` is sorted alphabetically.

`ggplot2` also plots `factor` in that order, so you can adjust the `factor` to adjust the drawing order.

```
res =
  data.frame(responses)
# Sort by level of agreement from strong -> weak
res$res =
  factor(
    res$res,
    levels =
       c("Strongly Agree", "Agree", "Disagree")
  )

response_barplot2 =
  ggplot(
    data = res,
    aes(x = res)
  ) +
  geom_bar() +
  xlab("Response")
```



You can also use the parameter `ordered` to let others know that your `factor` is ordered properly.

```
responses =
  factor(
    c("Agree", "Agree", "Strongly Agree", "Disagree", "Disagree", "Agree"),
    ordered = TRUE
  )
```

```
> is.ordered(responses)
[1] TRUE
```

Using `factor` to vhange values

You can use `recode()` in `dplyr` package to change `value`

`dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

- `mutate()` adds new variables that are functions of existing variables
- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

These all combine naturally with `group_by()` which allows you to perform any operation "by group". You can learn more about them in `vignette("dplyr")`. As well as these single-table verbs, dplyr also provides a variety of two-table verbs, which you can learn about in `vignette("two-table")`.

> Based on the introduction on the official website of `dplyr`.

Here's an example:

```
x =
  factor(
    c("alpha", "beta", "gamma", "theta", "beta", "alpha")
  )

x =
  recode(
    x,
    alpha = "a",
    beta = "b",
    gamma = "c",
    theta = "d"
  )
```

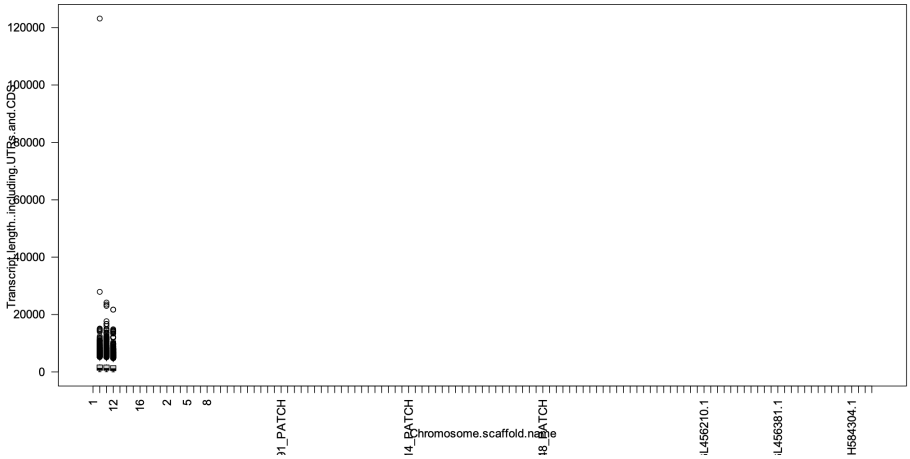
```
+     theta = "d"
+   )
> str(x)
 Factor w/ 4 levels "a","b","c","d": 1 2 3 4 2 1

>
```

Delete useless `levels`

```
mouse.genes =
  read.delim(
    file = "data/talk04/mouse_genes_biomart_sep2018.txt",
    sep = "\t",
    header = T,
    stringsAsFactors = T
  )
```

```
+   )
> str(mouse.genes)
'data.frame':   138532 obs. of  6 variables:
 $ Gene.stable.ID                        : Factor w/ 55029 levels "ENSMUSG00000000001",..: 17960 17959 17958 17957 17956 17955 17954 17953 17952 17951 ...
 $ Transcript.stable.ID                  : Factor w/ 138532 levels "ENSMUST00000000001",..: 17312 17311 17310 17309 17308 17307 17306 17305 17304 17303 ...
 $ Protein.stable.ID                     : Factor w/ 65897 levels "","ENSMUSP00000000001",..: 1 1 16260 1 16259 16258 1 1 1 16257 ...
 $ Transcript.length..including.UTRs.and.CDS.: int  67 67 1144 69 519 1824 71 59 67 1378 ...
 $ Transcript.type                       : Factor w/ 48 levels "3prime_overlapping_ncRNA",..: 18 18 24 18 24 24 18 18 18 24 ...
 $ Chromosome.scaffold.name              : Factor w/ 117 levels "1","10","11",..: 115 115 115 115 115 115 115 115 115 115 ...
```

If you draw a plot without deleting the useless `levels`, you will get this result:



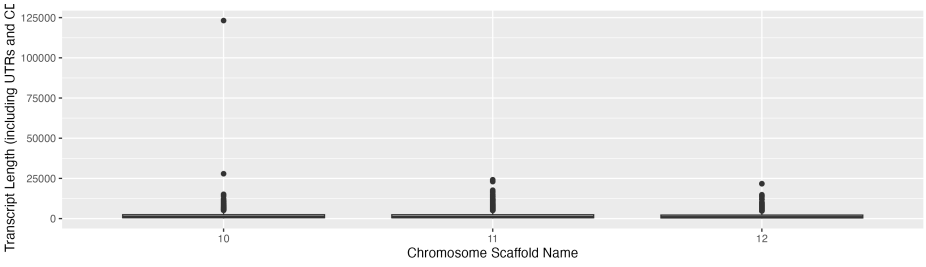
But when you delete the useless `level` using these commands:

```
mouse.chr_10_12$Chromosome.scaffold.name =
  droplevels(mouse.chr_10_12$Chromosome.scaffold.name)
```

You will see that:

```
+   droplevels( mouse.chr_10_12$Chromosome.scaffold.name
> levels(mouse.chr_10_12$Chromosome.scaffold.name)
[1] "10" "11" "12"
```

Then, you'll get the plot like this:

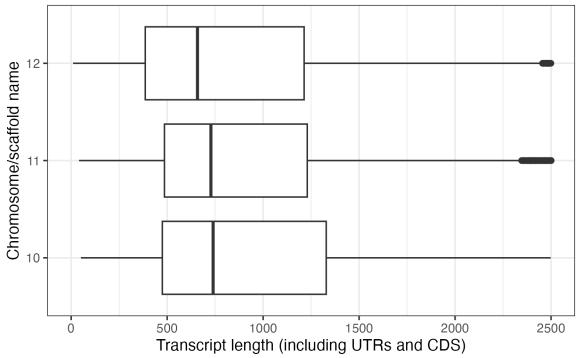


Source code:

```
mouse_gene_plot02 =
  ggplot(
    mouse.chr_10_12,
    aes(
      x = Chromosome.scaffold.name,
      y = Transcript.length..including.UTRs.and.CDS.
    )
  ) +
  geom_boxplot() +
  labs(
    x = "Chromosome Scaffold Name",
    y = "Transcript Length (including UTRs and CDS)"
  )
```

You can also use `tibble` to solve these problems:

```
mouse.tibble =
  read_delim(
    file = "data/talk04/mouse_genes_biomart_sep2018.txt",
    delim = "\t",
    quote = "",
    show_col_types = FALSE
  )

mouse.tibble.chr10_12 =
  mouse.tibble %>% filter(
    `Chromosome/scaffold name` %in% c("10", "11", "12"))


mouse_gene_plot03 =
    ggplot(
        mouse.tibble.chr10_12,
        aes(
        x = Chromosome.scaffold.name,
        y = Transcript.length..including.UTRs.and.CDS.
        )
    ) +
    geom_boxplot() +
    labs(
        x = "Chromosome",
        y = "Transcript length (bp)"
    ) +
    coord_flip() +
    ylim(0, 2500) +
    theme_bw()
```



Advance usage

- Use `reorder()` function to reorder the level.

```
x = reorder(
  `Chromosome/scaffold name`,
  `Transcript length (including UTRs and CDS)`,
  median
)
```

- Use `forcats::fct_reorder()` to reorder factors

```
x = fct_reorder(
  `Chromosome/scaffold name`,
  `Transcript length (including UTRs and CDS)`,
  median
)
```