Lab 1 of STAT462

tags: MOC of Lectures
Date: 2025-02-18-星期二

Lecture Material
Index: Data Mining – Outline
previous:
next:

Lectures

New Lecture

Lecture Material
Course Echo360 link

The recommended texts are:
If you are looking for some materials for beginners (or even advanced level), those are really good starting points:

- For zero R experience:
  - Getting Started with R and RStudio
- For data science:
  - r4ds-1e, r4ds-2e
  - YaRrr!
- For visualization:
  - R Graphics Cookbook
  - Fundamentals of Data Visualization
- For authoring a markdown file:
  - Reproducible reports with R markdown
  - R Markdown: The Definitive Guide
  - bookdown: Authoring Books and Technical Documents with R Markdown
  - Quarto

There are several ways to interact this this material. We shall begin by viewing this as an HTML page in a web browser and copying fragments of code into a Console session of RStudio.

Start a session of RStudio and be ready to copy and paste the grey-box styled code like the one shown below.

```
# Everything after the "#" symbol is a comment line of R code.

# It is used to annotate code and make it more readable

# Comments should briefly explain what lines of code are supposed to be doing

print("Welcome to R") # this displays the string "Welcome to R" on the screen.
```

Notice that you are able to copy and paste multiple lines at one time. R will process them in sequence.

::: boxTask

Your first task:

- [x] ~~1. After this box, create a new code block by doing one of the following:~~
  - (the shortcut Ctrl + Alt + I can be used, or
  - (in visual mode) you click on insert -> Executable Cell -> R, or

- (in source mode) you type in ```{r} [press enter] ```.

- [x] ~~2. In the code block (after {r}) and before the closing three quotes: Display your name using the `print` command.~~
- [x] ~~3. In order to see an effect of your changes in the grey code block, you need to run it. In every code block there is a small green triangle which you need to press in order to run this specific block. There are also commands for running several, or all code blocks, which is sometimes useful if you need to refresh the whole document. Also, there are shortcuts for running a code block.~~

Shortcut

- https://support.posit.co/hc/en-us/articles/200711853-Keyboard-Shortcuts-in-the-RStudio-IDE

- [x] ~~4. Hone your googling skills by finding out which keyboard shortcut runs a code block (so you don't need to click on the symbol).~~
  - ~~cmd+enter~~
- [x] ~~5. Don't worry about the weird "[1]" at the beginning of the output. We'll get to this later.~~

:::

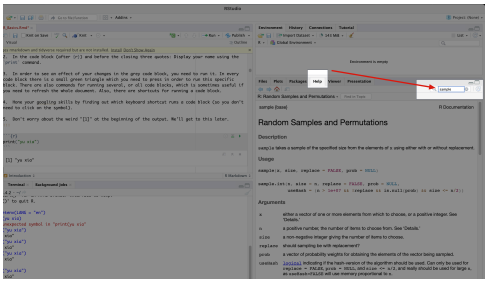
Some tricks

Getting helps

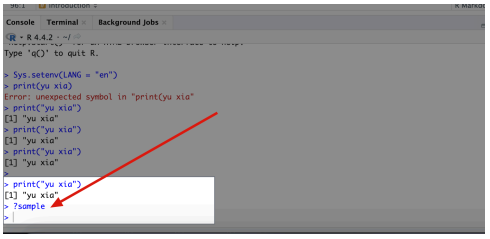
Other than asking our tutors for help, there are plenty of resources available for you to explore on you own. A very important source is the documentation, which provides the detailed explanation of how a library/function works.

For example, if we want to learn the details about a function called `sample()`, we can access to the documentation by the following ways:

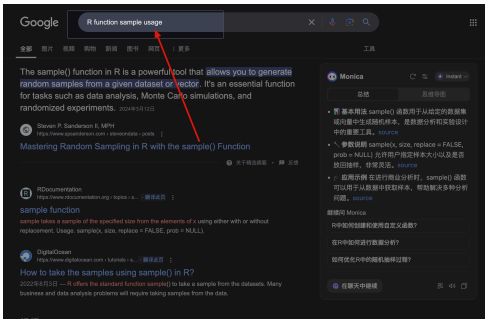
- [x] ~~In `Help` panel search `sample`~~



- [x] ~~Execute `?sample` in a script or in the Console~~



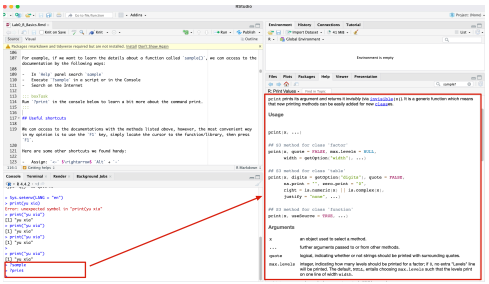
- [x] ~~Search on the Internet~~



~~::: boxTask~~

Run `?print` in the console below to learn a bit more about the command print.

:::



Useful shortcuts

We can access to the documentations with the methods listed above, however, the most convenient way in my opinion is to use the `F1` key, simply locate the cursor to the function/library, then press `F1`.

Here are some other shortcuts we found handy:
Windows:

- Assign: `<-` → `Alt` + `-`
- Piping: `%>%` → `Ctrl` + `Shift` + `M`
- Comment: `#` → `Ctrl` + `Shift` + `C` (works on multiple lines as well)
- Make text italic/cursive: `Ctrl` + `I` (works on multiple lines as well)
- Keyboard shortcut reference: `Alt` + `Shift` + `K`

Mac:

- Assign: `<-` → `Opt` + `-`
- Piping: `%>%` → `Cmd` + `Shift` + `M`

```
在 RStudio 中，使用 `cmd + Shift + M` 快捷键可以插入 `%>%` 操作符，这个操作符被称为“管道操作符”（pipe operator）。它主要用于将一个表达式的输出作为下一个表达式的输入，从而实现代码的链式调用。
It is mainly used to take the output of one expression as the input to the next expression, thus enabling chain calls to code.
```

below is a e.g. for %>% (piping)

```
library(dplyr)

# 使用管道操作符
result <- df %>%
  filter(column1 > 10) %>%
  select(column2, column3) %>%
  summarize(mean_value = mean(column2))
```

- Comment: `#` → `Ctrl` + `Shift` + `C` (works on multiple lines as well)

```
# comment line
```

- [ ] Make text italic/cursive: `Ctrl` + `I` (works on multiple lines as well)

```
# Line which can not apply Ctrl+I to italic
```

- Keyboard shortcut reference: `Alt` + `Shift` + `K`

You may wonder, what does *assign* mean? What is *pipe* and when do we need it? You'll find answers in the following sections.

Packages

What is a package?

A package is a set of R functions and data sets that are bound together in a single resource. So far, we have been using functions from the **base** package which is available as a language foundation.

We could have written the `max()` as `base::max()`, they are referring to the same function. ==There are some functions share the exact same name from different packages (the `select()` function is a good example of this), and which one will be used may not be predictable. ==In situations like this, reverting to the full name solves the issue `select()`

You can load a package with the R function `library()`:

```
library(tidyverse)
```

Note: The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. We will be using it very very very very often.

tidyverse package intro

The local R installation must know about this package, if we want to load a package smoothly. During the labs, you are assured that every official package has already been installed but when you switch to another environment, say your own laptop, this is unlikely to be the case. Official packages are referred to as Comprehensive R Archive Network (CRAN) packages.

```
install.packages("tidyverse") # install tidyverse
installed.packages() # check installation status
library(tidyverse) # load package
```

Installing packages

The easiest way to install a package is to let RStudio do it for you. Locate the lower−right panel of RStudio and select the *Packages* tab.Use the install button to locate and install packages.

Doing the same thing through the console involves the `install.packages()` function.

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

Installing tidyverse may take longer than other packages and it is very likely to be unsuccessful at the first attempt. This is common, tidyverse is a collection of R packages, it may require some other packages to be installed prior, try to follow the hint from the error message(it is also an essential skill) to see what is missing.

:::{.boxNote}

It is a good idea to stick to the same working environment if it is possible(for eg. your own device or the MADS server rather than any random machine/serves on campus), so that you do not need to install new libraries or change Global Options into your favour every time you switch to another environment.

:::

What have you learned?

- Packages need to be downloaded and installed on your personal laptops
- You have no need to do this on the MADS server
- Using RStudio to manage your packages is the best strategy

Variables

Assigning values

In R we can assign a variable (myVar) a value. Normally each line represents a complete program instruction. In this case **assign *27* to the variable *myVar***.

We can print the value several ways. The most convenient way is to put the variable on a line of its own.

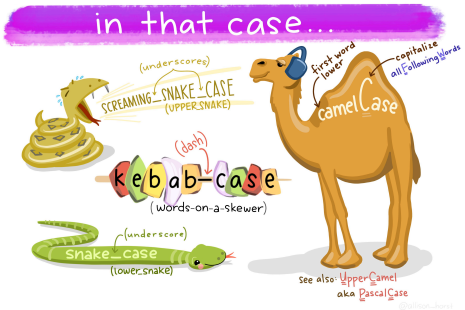
```
myVar <- 27

myVar
```

::: boxTask

Change the code above so that myVar now contains your name. Note that strings need to be enclosed by quotes (or otherwise R will interpret them as variable names). Run the code snippet with the little green triangle symbol and check whether your name is being shown on screen correctly.

:::

Variables in R are **case sensitive**. That means ***myVar*** and ***myvar*** would be separate variables. This could be a common mistake, especially for beginners.



In R, variables typically contain a list of things rather than just a single thing. Of course a list can also contain just a single value as a special case.\

One way to create a list of values is the `c()` function.

```
myVar <- c(27,28,29,30,67,92)

myVar
```

You can read the `c()` as meaning "column", or "concatenate". Mathematically, the variable is a "vector".

There is more than one way to assign a variable. In these labs we shall use the **<-** assignment operator. We could also use `assign("myVar", c(27,28,29,30,67,92))` . There is another style that you will frequently encounter: `myVar = c(27,28,29,30,67,92)` (with an "=" instead of "<−"). For *our* purposes these are equivalent.

Variable Types

What types of variables does R support?

Here are some common types of variable.

- Numeric variables − these can be decimal or integer
- Character string variables − these can be varying lengths of letters, numbers & punctuation
- Logical − TRUE or FALSE

- Factor – short strings that occur frequently e.g. "Low" / "Medium" / "High"
- Date – dates and times displayed using a particular format
- Data frame – used to store tabular data with rows and columns

```
myNum<-c(1, 2, 3, pi, 0.1, -1/6)

myNum
```

```
myChar <- c("The ", "quick", " brown fox jumped ", "over the ", "lazy dog 27 times", ".")

myChar
```

```
myLog <- c(TRUE, TRUE, FALSE, TRUE, F, T)

myLog
```

Below is an example of an ordered factor.

```
myFactor <- factor(x = c("High", "Low", "Low", "Medium", "Low", "High"), levels = c("Low", "Medium",
"High"), ordered = TRUE)

myFactor
```

```
myDate <- c(Sys.time() - 1,

            Sys.time(),

            Sys.time() + 1,

            Sys.Date() - 1,

            Sys.Date(),

            Sys.Date() + 1

            )

myDate
```

`Sys.time()` and `Sys.Date()` return different types of outcome, but all elements in `c()` should be in the same type, we will discuss more about this later.

On the other hand, data frame can store the values with different types in tabular fashion.

```
myDf <- data.frame(myNum, myChar, myLog, myDate, myFactor)

myDf
```

::: boxTask

Give it a go:

In the last code block, change `myDf` to `View(myDf)` , what's the difference?

:::

Print layout

```
runif(10, max = 50)
```

What is going on with the `[1]` ?

When R prints a vector, it does so in a particular style. Suppose a vector was 100 long. How would you locate the 76th item. Let's try that

```
set.seed(99)

runif(100, max = 50)
```

Using the [??] indices it is possible to locate the 76th case as "4.52055981"

Consider lists

Suppose we consider grocery shopping lists. We might think of things on the list as being the things we need to buy. For example:

- Coffee
- Milk
- Biscuits (chocolate)
- Carrots
- Rice (basmati)
- Juice (apple)

This looks like a character variable. Let's create this in R and print it. Notice that we are using the explicit **print** rather than the implicit one.

```
shopping <- c("Coffee", "Milk", "Biscuits (chocolate)", "Carrots", "Rice (basmati)", "Juice (apple)")

print(shopping)
```

Suppose we want to reuse this list from one week to the next. What do we need to change?\

Now it becomes about whether an item is actually needed this week. How can we do this in R?\

We want to shift from a list of characters to a logical list *that is labelled*.

```
shopping <- c(`Coffee` = TRUE, `Milk` = TRUE, `Biscuits (chocolate)` = TRUE, `Carrots` = FALSE, `Rice
(basmati)` = FALSE, `Juice (apple)` = FALSE, `Juice (orange)` = TRUE)

print(shopping)
```

We have created a list of TRUE/FALSE entries against the names of the shopping items. That was easy.

Lists of lists

Suppose we have kept the shopping lists we have used for the last 12 months.\

Also, suppose we labelled these lists with the date.\

This is how we might populate variables in R with this data. This way assumes we never misspell a grocery item or describe the same thing differently – put that objection out of your head for now.

```
Jan06 <- c(`Coffee` = FALSE, `Milk` = TRUE, `Biscuits (chocolate)` = FALSE, `Carrots` = TRUE, `Rice
(basmati)` = FALSE, `Juice (apple)` = FALSE, `Juice (orange)` = TRUE)

Jan11 <- c(`Coffee` = TRUE, `Milk` = TRUE, `Biscuits (chocolate)` = TRUE, `Carrots` = FALSE, `Rice
(basmati)` = TRUE, `Juice (apple)` = TRUE, `Juice (orange)` = FALSE)

Jan15 <- c(`Coffee` = FALSE, `Milk` = TRUE, `Biscuits (chocolate)` = TRUE, `Carrots` = FALSE, `Rice
(basmati)` = FALSE, `Juice (apple)` = TRUE, `Juice (orange)` = FALSE)
```

Obviously we are going to stop at 3 lists because this is a teaching exercise and not an endurance test.

How do we combine these things into a list of lists?

```
shoppingHistory <- list(Jan06, Jan11, Jan15)

shoppingHistory
```

Hold on, earlier we used `c()` and now we have switched to `list()`. What is going on?

A *"vector"* is a special list in which the values are

- single things
- of the same type

::: boxTask

What happens if you change `list()` to `c()` above?

:::

::: boxTask

Give it a go:

What if we force different types of value into the same vector? Try `c("1", 2)` and `c(1, 2)`, what's the difference?

:::

A *"list"* is a general collection of anything (including other lists) and the entries can be different types.

```
print(shoppingHistory)
```

We were able to label out shopping items; can we label out shopping lists?

```
shoppingHistory = list(`Jan06`=Jan06, `Jan11`=Jan11, `Jan15`=Jan15)

print(shoppingHistory)
```

Later we will mostly work with `data.frame`s instead of lists, see the next lab.

What have you learned?

- Everything in R is a list or built up using lists

- A vector(`c()`) is a special list of the same type of singular thing (e.g. all numeric values)
- Lists can be labelled – this is optional and can be useful for navigating through the list

Operators

Numeric operators are processes like **multiplication, addition, power, log, sin**, etc\

Logical operators are processes like **and, or, not** etc\

Character operators are processes like **concatenation, splitting, length** etc

How do we use operators with lists of things?

Numeric operators

Suppose we populate variables A and B in the following way:

```
A <- c(19,21,4,17,5)

B <- c(5,11,9,13,19)
```

What does the multiplication of A and B look like? What does it produce?

```
A * B
```

The result is an element−wise multiplication.

What does A cubed look like? What does it produce?

```
A^3
```

The result is an element−wise cubing.

What does A to the power of B look like? What does it produce?

```
A^B
```

The result is 19^5, 21^{11}, 4^9 etc.

::: boxTask

Create a new codeblock and compute the sum of vectors `A` and `B`.

:::

This should be starting to look sensible. It is not always so obvious though. Suppose we want the largest value in A

```
max(A)
```

We are not done yet. Suppose we want the largest value in A or B

```
max(c(A,B))
```

We had to create a new variable which was the combination of the two lists and then test that for the maximum. We are not finished yet. Suppose we wanted the pair−wise maximum of A and B, i.e. a vector that always picks the largest option between A[i] and B[i] at every index i. This is done by `pmax` .

```
pmax(A,B)
```

What we have avoided using is `max(A,B)` . So what does this produce?

```
max(A,B)
```

Clearly this is the same as `max(c(A,B))` . It was not doing pair wise operation − that requires `pmax()` .

Let's consider what happens when the lengths of the two vectors is not the same. Firstly lets use a vector of length 1

```
B*10
```

This should be identical to `B * c(10)` . Let's check...

```
B*c(10)
```

When the lengths do not match, the shorter vector is repeated until it reaches the right length. In this case the 10 was repeated five times to match the length of B.

Now let's try multiplying by c(10,100)

```
B*c(10,100)
```

Notice two things:

1 The calculation proceeded and produced: [1] 50 1100 90 1300 190 2 There was a warning objecting to what you were trying to do: *longer object length is not a multiple of shorter object length*

Since, 5 is not a multiple of 2, the rule about repeating the shorter variable does not make much sense − hence the warning.

Logical operators

Suppose we populate variables J and K in the following way. Note that `T` and `F` are shorthands for `TRUE` and `FALSE` :

```
J <- c(T,F,F,T,F)

K <- c(F,T,F,T,T)
```

The characters `!` , `&` , `|` have a special meaning in logical expressions (as they do in most programming languages). *R also supports shortcut Boolean (&& and ||) but these are reserved for single valued logical expressions rather than multiple values expressions − therefore we do not get many opportunities to shortcut Boolean expressions.*

To ***AND*** J and K together, we do the following

```
J & K
```

To ***OR*** J and K together, we do the following

```
J | K
```

To evaluate ***NOT*** J, we do the following

```
!J
```

How do we check if J has ***any TRUE*** values?

```
any(J)
```

How do we check if K has ***all FALSE*** values?

```
all(!K)
```

Notice that `any()` and `all()` only ever return a single value – never a list.

We can also switch between numeric and Boolean.

```
L <- A > 10

L
```

Checking for equality is done with the `==` operator (NOT `=`).

```
print(A)

print(A == 4)
```

We can think of *TRUE* being 1 and *FALSE* being 0. When we use a logical variable in a numeric expression, it is tolerated just fine. Do make sure that the effect is what you intend and not a typo. Tricks like this allow efficient code, for e.g, we want to know how many values are greater than the threshold in a vector generated randomly.

```
nums <- rnorm(100)

nums

sum(nums>0)
```

::: boxTask

Give it a go:

What if we want to know how many numbers in **nums** are greater than −1 but smaller than 1?

:::

::: boxTask

A bit tricky:

Use the ideas in this section to create a list containing all numbers between 1 and 100 which are divisible by 2 or 3, but not by 15. You will need the following things:

1. Regarding divisibility: A number x is divisible by p if and only if `x %% p` is equal to 0.

2. `1:100` creates a list with all numbers between 1 and 100.

:::

Character operators

The big issue with character variables is how non−intuitive the concatenation can be. Once you master this you have done the difficult part.

Character variables should NOT be thought of as a sequence of letters that make up a character string. In many programming languages this is the paradigm − not so in R. One simple difference is the length of a variable with character strings and the length of those characters.

```
V <- c("The best day of my life")

length(V)
```

The number of characters in a variable can be found using the **nchar()** function:

```
nchar(V)
```

To split a string into words we can use `strsplit()`:

```
strsplit(V, split = " ")
```

Notice also that the output is a list of one thing: a vector of character strings (you can see that this is the case by the 1[1] symbols before the actual output). This is so that `strsplit()` can operate on vectors longer than 1

```
strsplit(c(V, "This is another sentence."), split = " ")
```

```
words <- unlist(strsplit(V, split = " "))

words
```

Notice that the `unlist()` function was introduced to convert the (unnecessary) list back into a vector of words.

Now we turn to concatenation. Suppose we wanted to join the words we just split back into a single string with a dot as a word separator. We need to apply a function called `paste()`

```
paste(words, collapse = ".")
```

The collapse parameters controls whether the concatenation is going to be element wise or vector wise. In this case we are collapsing the list into a single value.

We could have choose to append something to the end of each word (say a dash).

```
paste(words, "-")
```

Notice that a space has crept in. To get rid of that we must change a default parameter and try again.

```
paste(words, "-", sep = "")
```

::: boxTask

In the next code block there is a phone number with inserted dashes which you are supposed to get rid of. Using the techniques presented above, write code so that `phone_without_dashes` contains "027283124221" (without just typing it in there, imagine you have a long database of thousands of phone numbers in that format).

:::

```
phone <- "027-283-124-221"

phone_without_dashes <- NULL
```

What have you learned?

- Variables honor element−wise behavior − be aware of `max()` / `pmax()`
- Length mismatches will *NOT* fail − warnings are there to guide you
- Logical variables can be used in numeric expressions
- Character variables are lists of strings not necessarily letters/digits etc
- `paste()` will do the concatenation but you need to think about how the concatenation should work.

End of lab0

```
Dataview (inline field '='): Error:
-- PARSING FAILED -----------------------------------------------

> 1 | =
    | ^

Expected one of the following:

'(', 'null', boolean, date, duration, file link, list ('[1, 2, 3]'), negated field, number,
object ('{ a: 1, b: 2 }'), string, variable
```