

# Introduction and R Basics

STA 032: Gateway to data science Lecture 3

Jingwei Xiong

April 7, 2023

# Reminders

- Homework 1 has been assigned: (Due April 17 midnight, cover lecture 1-4)
  - start as soon as possible
  - PDF files only
  - Submission through Gradescope (accessible through Canvas)
  - If you get stuck, it's encouraged to communicate with your mate about solution.
  - But you should type your solution by your own.
  - If you collaborate with others, write their names in your submission
- Office hours:
  - TBD

# Today

- Vectors arithmetics
- Logical subsetting
- Installing packages
- Function basics

# Questions from last class: NA vs. NaN

- NaN means "not a number" and it means there is a result, but it cannot be represented by the computer

```
0 / 0 # note that 1 / 0 returns Inf
```

```
[1] NaN
```

- NA means missing; when working with data sets this is the more common one you will encounter
- `is.na()` returns TRUE for both missing values (NA) and NaN

```
is.na(0 / 0)
```

```
[1] TRUE
```

```
NA + NaN
```

```
[1] NA
```

- For more, see <https://jameshoward.us/2016/07/18/nan-versus-na-r/>.

# Recap: Vector covered last lecture:

- How to create a vector: `c()`, `1:5`, `seq()`, `vector(length = 7)`
- How to subset a vector using index
- How to subset a vector using name

And today we will continue with more on vectors.

# Vector arithmetic with a constant

In R, arithmetic operations on vectors occur *element-wise*. For a quick example, suppose we have height in inches:

```
inches <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)
```

and want to convert to centimeters. Notice what happens when we multiply inches by 2.54:

```
inches * 2.54
```

```
[1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.42 170.18  
[10] 177.80
```

In the line above, we multiplied each element by 2.54. Similarly, if for each entry we want to compute how many inches taller or shorter than 69 inches, the average height for males, we can subtract it from every entry like this:

```
inches - 69
```

```
[1]  0 -7 -3  1  1  4 -2  4 -2  1
```

# Vector arithmetic: Two vectors

If we have two vectors of the same length, and we sum them in R, they will be added entry by entry as follows:

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} + \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} a + e \\ b + f \\ c + g \\ d + h \end{pmatrix}$$

The same holds for other mathematical operations, such as  $-$ ,  $*$  and  $/$ .

# Example: Vector arithmetic: Two vectors

```
x <- c(7, 8, 10, 45)
y <- c(-7, -8, -10, -45)
x + y
```

```
[1] 0 0 0 0
```

```
x * y
```

```
[1] -49 -64 -100 -2025
```

```
x^c(1, 0, -1, 0.5)
```

```
[1] 7.000000 1.000000 0.100000 6.708204
```



# Recycling

- R will also implicitly coerce the length of vectors.
- This is called vector **recycling**:
  - When a shorter vector is combined with a longer one, elements of the shorter vector are repeated or recycled, to make it the same length as the longer vector.

```
x <- c(7, 8, 10, 45)  
x + c(-7, -8)
```

```
[1] 0 0 3 37
```

Single numbers are vectors of length 1 for purposes of recycling:

```
2*x
```

```
[1] 14 16 20 90
```

# Vectorized functions: Examples

Most built-in functions are vectorized, meaning that they will operate on a vector of numbers. Here are some examples about functions taking vector as input:

```
sample(1:10) + 100
```

```
[1] 109 110 102 101 104 108 105 107 103 106
```

(what does `sample()` do?)

Operator also work as functions:

```
x
```

```
[1] 7 8 10 45
```

```
x > 9 # pairwise comparisons, where the scalar 9 is recycled
```

```
[1] FALSE FALSE TRUE TRUE
```

# Vectorized functions

Lots of functions take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, `sum()`: return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

# Vector subsetting using conditions

Because Boolean operators work elementwise, We can use logical operators and comparison operators to specify a condition, and R will return Boolean vector that is TRUE for the elements that meet the condition and FALSE for the others.

```
ages <- c(23, 35, 28, 19, 42, 30, 38, 27)
(ages > 25) & (ages < 40)
```

```
[1] FALSE TRUE TRUE FALSE FALSE TRUE TRUE TRUE
```

- Use the Boolean vector as an index for the original vector.

```
ages[(ages > 25) & (ages < 40)]
```

```
[1] 35 28 30 38 27
```

- To get the number of components that satisfy a certain condition: This works because R coerces the TRUE and FALSE values to 1 and 0, respectively.

```
sum((ages > 25) & (ages < 40)) # another example of coercion
```

```
[1] 5
```

# The which() Function

This function returns the indices of the elements in a vector that satisfy a certain condition.

For example, let's say we have a vector of test scores:

```
scores <- c(78, 82, 91, 64, 87, 76, 93, 80)
```

We can use the which() function to return the indices of the scores that are greater than 80:

```
which(scores > 80)
```

```
[1] 2 3 5 7
```

We can also use the `which()` function to return the indices of the minimum or maximum values in a vector.

```
which.min(scores)
```

```
[1] 4
```

With these index we can easily subset the vector.

```
scores[which(scores > 80)] # subset by index
```

```
[1] 82 91 87 93
```

```
scores[scores > 80] # subset by boolean values
```

```
[1] 82 91 87 93
```

```
# Why they are the same?  
scores[which.max(scores)]
```

```
[1] 93
```

# The %in% Operator

Another useful operator for working with vectors in R is the %in% operator. This operator allows us to test whether the elements of one vector are present in another vector.

For example, let's say we have a vector of names:

```
names <- c("Alice", "Bob", "Charlie", "David", "Eve")
```

We can use the %in% operator to test whether a vector of search terms is present in the names vector:

```
search_terms <- c("Bob", "Eve", "Frank")  
search_terms %in% names
```

```
[1] TRUE TRUE FALSE
```

```
names[names %in% c("Bob", "Charlie", "Frank")]
```

```
[1] "Bob"      "Charlie"
```

# Comparison operators

When we want to compare two vectors element-wise, we can use Boolean operators like `==`

However, to compare **whole** vectors, best to use `identical()`:

```
x; y
```

```
[1]  7  8 10 45
```

```
[1] -7 -8 -10 -45
```

```
x == -y
```

```
[1] TRUE TRUE TRUE TRUE
```

```
identical(x, -y)
```

```
[1] TRUE
```



# Example 1: counting the number of missing values

It's common to encounter missing values (NA) in vectors when working with real-world data. In R, we can use the `is.na()` function to check for missing values in a vector, and the `sum()` function to count the number of missing values.

```
myNAvec = c(1,2,NA,4,NA)
is.na(myNAvec)
```

```
[1] FALSE FALSE  TRUE FALSE  TRUE
```

```
sum(is.na(myNAvec))
```

```
[1] 2
```

## Example 2: Calculate the summation

If we want to compute:  $1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots + \frac{1}{1000^2}$

We first define a vector contains numbers from 1 to 1000, then square that vector, and then use single number 1 divided by the squared vector. By vector arithmetic, the result will be 1 divided by each element of the vector, which will be a vector of  $1, \frac{1}{2^2}, \dots, \frac{1}{1000^2}$ .

Then use the function `sum` we can obtain the summation. In one line, it is:

```
sum(1 / (1:1000)^2 )
```

```
[1] 1.643935
```

# Installing R packages

- What you get after your first install is base R
- extra functionality comes from add-ons available from developers
- R makes it very easy to install packages from within R. For example, type this in console

```
install.packages("tidyverse")  
install.packages("ggplot2")  
install.packages("dslabs")
```

After we install the package, we can then load the package into our R sessions using the library function:

```
library(tidyverse)  
library(dslabs)
```

If you want to use the add-on functions in the package, you need to library the package first.

# Functions introduction

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

1	2	3	4	5	6	7
---	---	---	---	---	---	---

- The data analysis process are a series of **functions** applied to the data.
- We also used the function `sqrt` to solve the quadratic equation.
- Prebuilt R functions **do not** appear in the workspace because you did not define them, but they are available for immediate use.

# Functions introduction

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

1	<u>2</u>	3	4	5	6	7
---	----------	---	---	---	---	---

In general, we need to use **parentheses** to evaluate a function. If you type `ls`, the function is not evaluated and instead R shows you the code that defines the function.

```
ls
```

# Functions introduction

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

1	2	<u>3</u>	4	5	6	7
---	---	----------	---	---	---	---

If you type `ls()` the function is evaluated and then we see objects in the workspace.

```
ls()
```

[1]	"ages"	"from"	"inches"	"input"
[5]	"myNAvec"	"names"	"output_file"	"proc"
[9]	"render_args"	"render_fn"	"scores"	"search_terms"
[13]	"self_contained"	"to"	"x"	"y"

# Functions introduction

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

1	2	3	<u>4</u>	5	6	7
---	---	---	----------	---	---	---

Unlike `ls`, most functions require one or more **arguments**. Below is an example of how we assign an object to the argument of the function `log`. Remember that we earlier defined `a` to be 1:

```
a=1  
log(8)
```

```
[1] 2.079442
```

```
log(a)
```

```
[1] 0
```

# Functions introduction

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

1	2	3	4	<u>5</u>	6	7
---	---	---	---	----------	---	---

However, some arguments are required and others are optional. You can determine which arguments are optional by noting in the help document that a default value is assigned with `=`. Defining these is optional. For example, the base of the function `log` defaults to `base = exp(1)` making `log` the natural log by default.

You can change the default values by simply assigning another object:

```
log(8, base = 2)
```

```
[1] 3
```



# Functions introduction

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

1	2	3	4	5	<u>6</u>	7
---	---	---	---	---	----------	---

Note that we have not been specifying the argument `x` as such:

```
log(x = 8, base = 2)
```

```
[1] 3
```

The above code works, but we can save ourselves some typing: if no argument name is used, R assumes you are entering arguments in the order shown in the help file.

# Functions introduction

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

1	2	3	4	5	6	<u>7</u>
---	---	---	---	---	---	----------

So by not using the names, it assumes the arguments are  $x$  followed by base:

```
log(8,2)
```

```
[1] 3
```

If using the arguments' names, then we can include them in whatever order we want:

```
log(base = 2, x = 8)
```

```
[1] 3
```

To specify arguments, we must use `=`, and cannot use `<-`.

# R Markdown Revisit

In RStudio, you can start an R markdown document by clicking on **File, New File, the R Markdown**.

You will then be asked to enter a title and author for your document.

You can also decide what format you would like the final report to be in: HTML, PDF, or Microsoft Word.

It will generate a template file.

As a convention, we use the **Rmd suffix** for these files.

In the template, you will see several things to note.

# The YAML header

At the top you see:

```
---  
title: "Untitled"  
author: 'Jingwei Xiong'  
date: "2023/1/10"  
output: html_document  
---
```

The things between the `---` is the header. We actually don't need a header, but it is often useful. You can define many other things in the header than what is included in the template. We don't discuss those here, but much information is available online. The one parameter that we will highlight is `output`. By changing this to, say, `pdf_document`, we can control the type of output that is produced when we compile.

# R code chunks

In various places in the document, we see something like this:

```
`` `{r}  
summary(pressure)  
```
```

These are the code chunks. When you compile the document, the R code inside the chunk, in this case `summary(pressure)`, will be evaluated and the result included in that position in the final document.

This applies to plots as well; the plot will be placed in that position. We can write something like this:

```
`` `{r}  
plot(pressure)  
```
```

By default, the code will show up as well. To avoid having the code show up, you can use an argument. To avoid this, you can use the argument `echo=FALSE`. For example:

```
```{r, echo=FALSE}  
summary(pressure)  
```
```

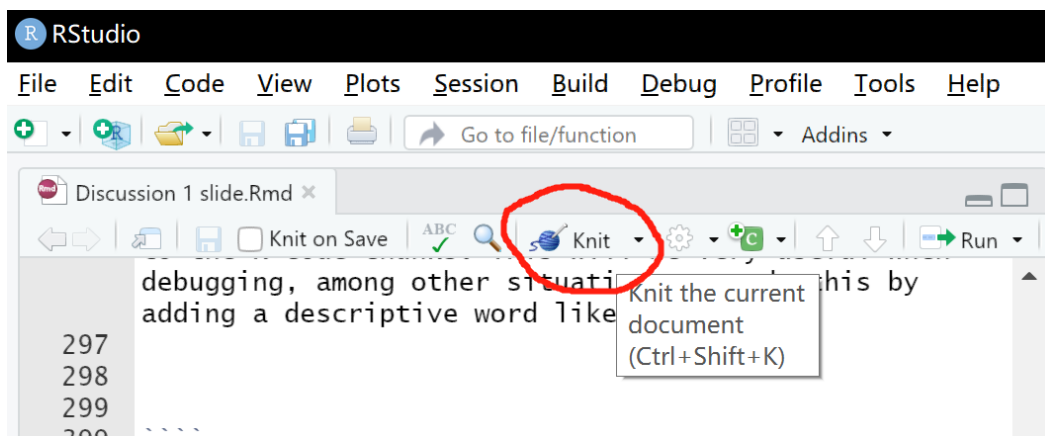
If you want to only show the code but not run the code, you can use the argument `eval=FALSE`.

```
```{r, eval=FALSE}  
summary(pressure)  
```
```

By default, the code will run and the output will be shown.

# Knit your first rmd file

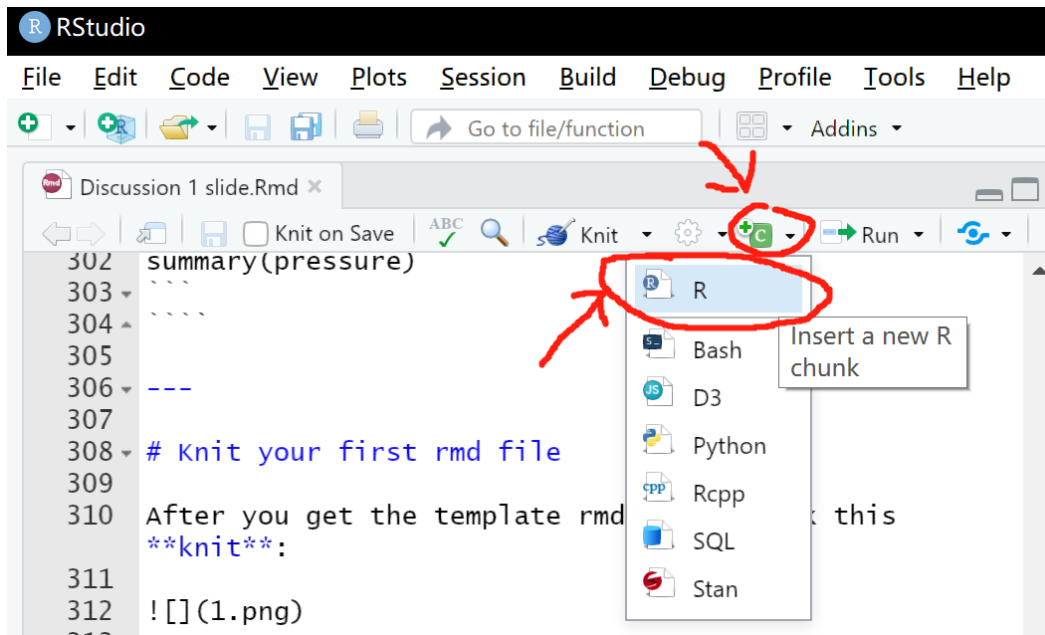
After you get the template rmd file, click this **knit**:



This button will process your source code into the final document, if your code has no error.

# Insert a new code chunk

To insert a new code chunk, click this:

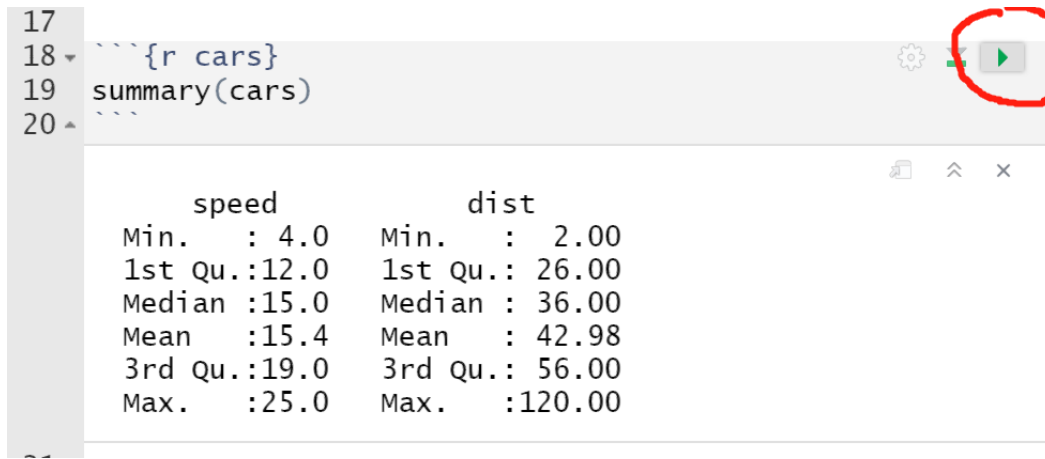


This button will insert a new code chunk in the current cursor line.



# Run scripts in the code chunk

To run scripts in a code chunk, click this:



The screenshot shows a code chunk in RStudio with the following R code:

```
17  
18 {r cars}  
19 summary(cars)  
20
```

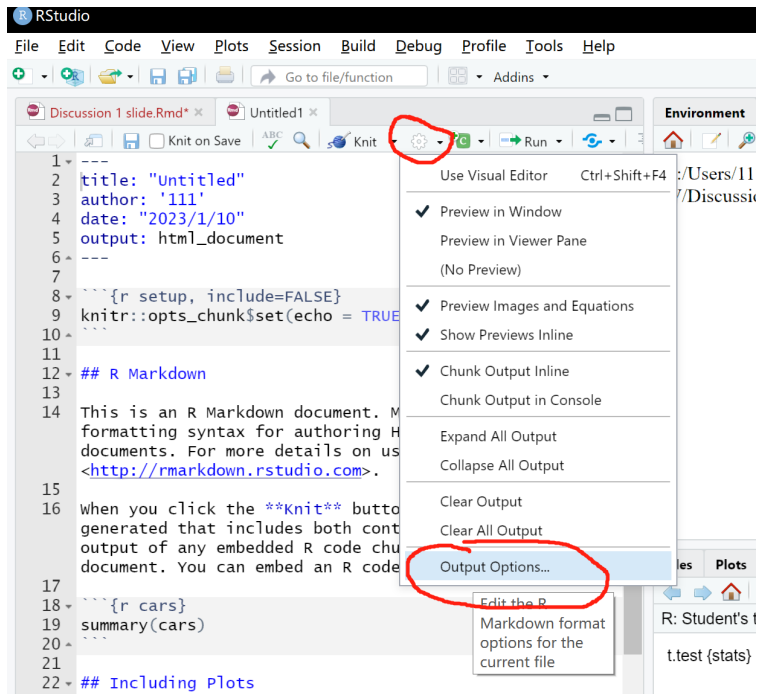
The 'Run' button (a green play icon) is circled in red. Below the code, the output of the `summary(cars)` function is displayed:

| speed    |      | dist     |        |
|----------|------|----------|--------|
| Min. :   | 4.0  | Min. :   | 2.00   |
| 1st Qu.: | 12.0 | 1st Qu.: | 26.00  |
| Median : | 15.0 | Median : | 36.00  |
| Mean :   | 15.4 | Mean :   | 42.98  |
| 3rd Qu.: | 19.0 | 3rd Qu.: | 56.00  |
| Max. :   | 25.0 | Max. :   | 120.00 |

This button will copy all of the codes inside that code chunk into the console, and run it.

# Knit settings

You can find the knit settings here:



You can change to word using the output format.

# Revisit: R markdown and console environment

The environment of your R Markdown document is separate from the Console!

First, run the following in the console

```
x <- 2  
x * 3
```

Then, add the following in an R chunk in your R Markdown document and try to knit it.

```
x * 3
```

What happens? Why the error?

## Explanation:

When adding the `x * 3` command to an R chunk in the R Markdown document, an error will occur because the variable `x` was not defined within the R markdown environment. The R Markdown environment is separate from the console environment, so any variables created or functions defined in the console will **not carry over to the R Markdown document** unless they are specifically included or imported.

# Example on Rmarkdown: How to do the homework

Create new code chunks; Run codes; Write your own response.

# Next week: Datasets

# Readings

- R for Data Science Chapter 20, 27
- Additional reading: Matloff Chapter 2
- Chapter 2:R basics
- R markdown tutorial