# Introduction and R Basics

## STA 032: Gateway to data science Lecture 2

Jingwei Xiong

April 5, 2023

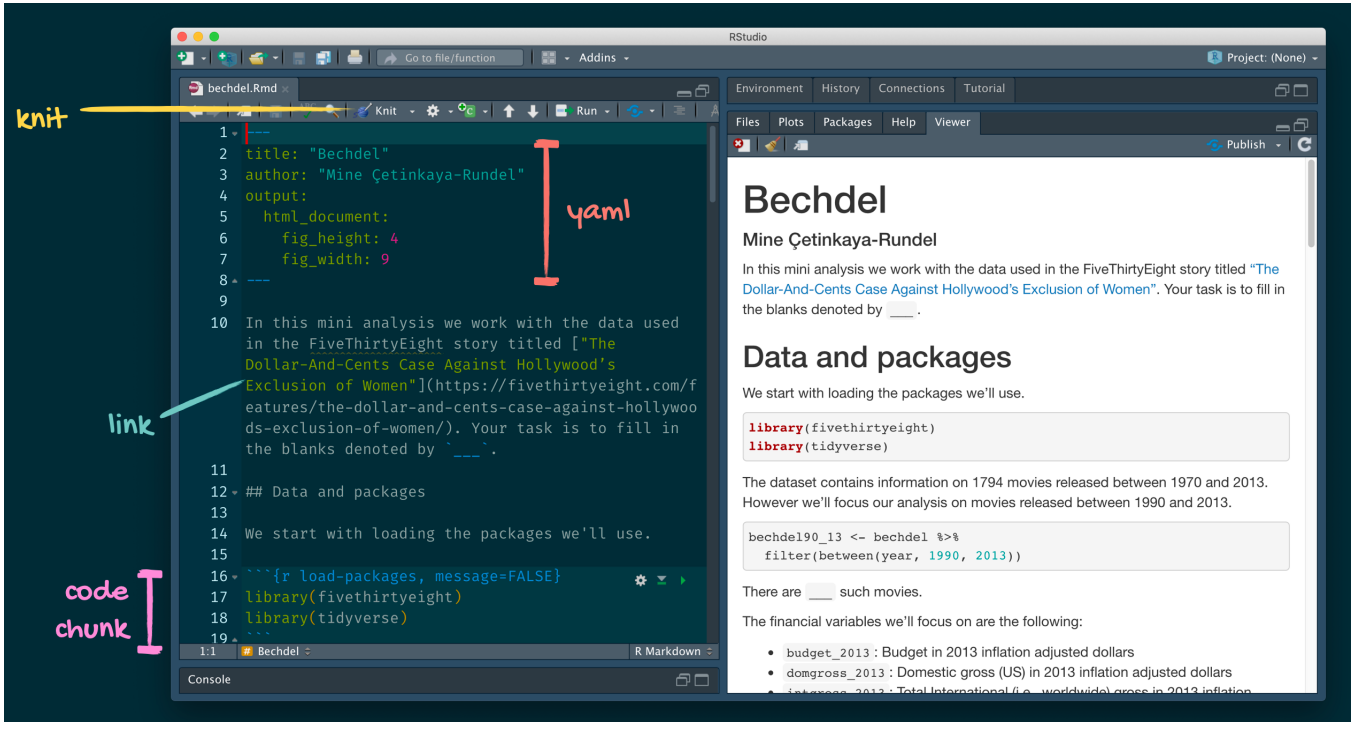
# Recap

- R is a free, open-source statistical programming language
- R is also an environment for statistical computing and graphics
- It is easily extensible with packages



- RStudio is a convenient interface for R called an **IDE** (integrated development environment)
- RStudio is not a requirement for programming with R, but it's very commonly used by R programmers and data scientists

# R Markdown




Hopefully you would have made your first R Markdown document. In this quarter all homework will be done with R markdown.

- Recap how to knit, how to create a code chunk.

# Today

- Data types
- Operators
- Vectors
- objects and environment

# Data types

- **Logical/Booleans**: binary values: `TRUE` or `FALSE` in R
- **Integers**: whole numbers (positive, negative or zero)
- **Double**: a floating point number, like $1.87 \times 10^6$
- **Characters**: e.g., letters; R displays it using double quotes. **strings** = sequences of characters
- **Complex**: complex numbers, e.g., 1 + 2i (rarely used in data analysis)
- **Dates**: Time objects (We will cover it later)

## Note:

- each of these data types can have missing or ill-defined values: NA, NaN, etc.
- Integers and doubles are both *numeric*

# Functions related to data types

`typeof()` function returns the type

`is.`*foo*`()` functions return Booleans (TRUE or FALSE) for whether the argument is of type *foo*

`as.`*foo*`()` (tries to) "cast" its argument to type *foo* --- to translate it sensibly into a *foo*-type value

`is.na()` checks if the value is NA

```
typeof(7)
```

```
[1] "double"
```

```
is.numeric(7)
```

```
[1] TRUE
```

# Functions related to data types

```
is.character(7)
is.character("7")
is.character("seven")
is.na("na")
is.na(NA)
```

# Functions related to data types

```
is.character(7)
```

```
[1] FALSE
```

```
is.character("7")
```

```
[1] TRUE
```

```
is.character("seven")
```

```
[1] TRUE
```

```
is.na("7")
```

```
[1] FALSE
```

# Functions related to data types

```
as.character(5/6)
as.numeric(as.character(5/6))
6*as.numeric(as.character(5/6))
```

# Functions related to data types

```
as.character(5/6)
```

```
[1] "0.833333333333333"
```

```
as.numeric(as.character(5/6))
6*as.numeric(as.character(5/6))
```

# Functions related to data types

```
as.character(5/6)
```

```
[1] "0.833333333333333"
```

```
as.numeric(as.character(5/6))
```

```
[1] 0.8333333
```

```
6*as.numeric(as.character(5/6))
```

# Functions related to data types

```
as.character(5/6)
```

```
[1] "0.833333333333333"
```

```
as.numeric(as.character(5/6))
```

```
[1] 0.8333333
```

```
6*as.numeric(as.character(5/6))
```

```
[1] 5
```

> Small summary: any numbers are numeric, anything inside qoute signs "" are string.

# Operators

- **Unary** operators take a single input, e.g., – for arithmetic negation, ! for Boolean
- **Binary** operators take two inputs and give a output, e.g., usual arithmetic operators, modulo
- See https://stat.ethz.ch/R-manual/R-devel/library/base/html/Arithmetic.html for more details

```
7 + 5
```

```
[1] 12
```

```
7 - 5
```

```
[1] 2
```

```
7*5
```

```
[1] 35
```

```
7^5
```

```
[1] 16807
```

```
7/5
```

```
[1] 1.4
```

```
7 %% 5
```

```
[1] 2
```

```
7 %/% 5
```

```
[1] 1
```

These operators represent exponentiation (^), division (/), modulus (%%), and integer division (%/%).

# Arithmetic Operators vs. Logical Operators

- Operators on previous slides were arithmetic operators
- **Logical operators** are **comparisons** (also binary operators; they take two objects, like numbers, and give a Boolean)

```
7 > 5
```

```
[1] TRUE
```

```
7 < 5
```

```
[1] FALSE
```

```
7 >= 7
```

```
[1] TRUE
```

Additional comparisons:

```
7 <= 5
```

```
[1] FALSE
```

```
7 == 5
```

```
[1] FALSE
```

```
7 != 5
```

```
[1] TRUE
```

- "and" and "or" are also logical operators

```
(5 > 7) & (6*7 == 42)
(5 > 7) | (6*7 == 42)
```

(what should these give?)

- "and" and "or" are also logical operators

```
(5 > 7) & (6*7 == 42)
```

```
[1] FALSE
```

```
(5 > 7) | (6*7 == 42)
```

```
[1] TRUE
```

The & operator returns TRUE if both sides of the operator are true, and the | operator returns TRUE if at least one side of the operator is true.

# Assignment operators: Data can have names

We can give names to data objects; these give us **variables**

A few variables are built in:

```
pi
```

```
[1] 3.141593
```

```
letters
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
[18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Variables can be arguments to functions or operators, just like constants:

```
pi*10
```

```
[1] 31.41593
```

Most variables are created with the **assignment operator**, <- or =

The value of a variable can be retrieved by typing its name, e.g., myVar.

```
myVar <- 100
myVar
```

```
[1] 100
```

```
myVar2 <- 10
myVar*myVar2
```

```
[1] 1000
```

The assignment operator also changes values:

```
myVar3 <- myVar*myVar2
myVar3
```

```
[1] 1000
```

```
myVar3 <- 30
myVar3
```

```
[1] 30
```

Using names and variables makes code easier to design, easier to debug, less prone to bugs, easier to improve, and easier for others to read

Avoid "magic constants" (using numbers directly in source code); use named variables

# Example: The root formula

Suppose a high school student asks us for help solving several quadratic equations of the form $ax^2 + bx + c = 0$. The quadratic formula gives us the solutions:

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

In R we can **define** variables and write expressions with these variables, similar to how we do so in math, and obtain a numeric solution.

```
a = 1
b = 1
c <- -1
```

```
a = 1
b = 1
c <- -1
```

Now these values are saved in variables, to obtain a solution to our equation, we use the quadratic formula:

```
(-b + sqrt(b^2 - 4*a*c) ) / ( 2*a )
```

```
[1] 0.618034
```

```
(-b - sqrt(b^2 - 4*a*c) ) / ( 2*a )
```

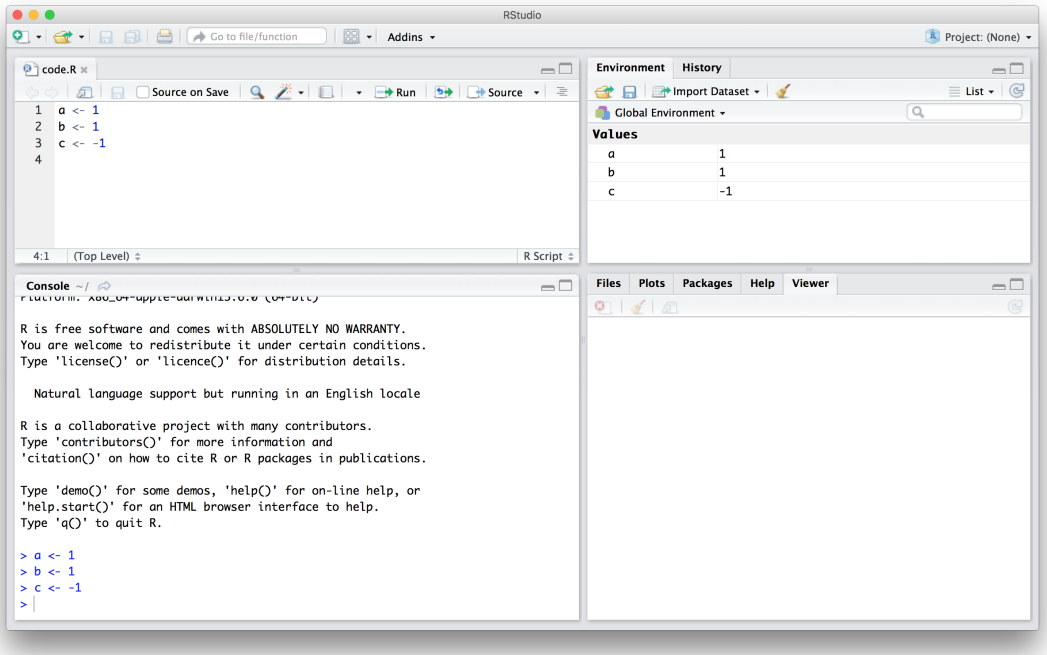
```
[1] -1.618034
```

# The workspace

We use the term *object* to describe stuff that is stored in R. Variables are examples, but objects can also be more complicated entities such as functions, datasets, which are described later.

As we define objects in the console, we are actually changing the *workspace*.

In RStudio, the *Environment* tab shows the objects in the workspace:



# object not found

In the environment, we should see `a`, `b`, and `c`. If you try to recover the value of a variable that is not in your workspace, you receive an error.

For example, if you type `x` you will receive the following message:

```
> x
Error: object 'x' not found
```

This error tells you that you asked the R to tell you something it doesn't know! (Because you haven't defined yet)

This is the **most common** error for beginners. A typo of object name can lead to this.

# Aside: R markdown

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.

# First data structure: vectors

Group related data values into one object, a **data structure**

A **vector** is a sequence of values, *all of the same type* (what are data types?)

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

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

```
is.vector(x)
```

```
[1] TRUE
```

> `c()` function creates a vector containing all its arguments in order,
> c stands for *concatenate*:

# Subsetting

`x[1]` is the first element, `x[4]` is the 4th element
`x[-4]` is a vector containing **all but** the fourth element

Vector of indices:

```
x
```

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

```
x[c(2, 4)]
```

```
[1]  8 45
```

Vector of negative indices

```
x[c(-1, -3)]
```

```
[1]  8 45
```

(why that, and not `8 10`?)

# length

To get the length of the vector, use `length()`

```
length(x)
```

```
[1] 4
```

# Creating vectors

In addition to `c()` function, the function `vector(length = 7)` returns an empty vector of length 7; helpful for filling things up later

```
weeklyHours <- vector(length = 7)
weeklyHours[5] <- 8
weeklyHours
```

```
[1] 0 0 0 0 8 0 0
```

The colon operator produces a sequence

```
mySeq <- 2:5
mySeq
```

```
[1] 2 3 4 5
```

The sequences defined above are particularly useful if we want to access, say, the first two elements:

```
x[1:2]
```

```
[1] 7 8
```

# Creating vectors

Many other ways to produce sequences, e.g.,

```
(mySeq <- seq(from = 1, to = 10, by = 2))
```

```
[1] 1 3 5 7 9
```

seq() function takes three arguments: from, to and by (optional).

from and to are the first and last numbers in the sequence, and by specifies the increment between numbers in the sequence.

(Enclosing an assignment statement in parentheses prints the result)

# Vector can have names for each element

Remember can create vectors using the function `c`, which stands for *concatenate*:

```
codes <- c(380, 124, 818)
```

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variable names.

```
country <- c("italy", "canada", "egypt")
```

In R you can also use single quotes:

```
country <- c('italy', 'canada', 'egypt')
```

# Vector can have names for each element

By now you should know that if you type:

```
country <- c(italy, canada, egypt)
```

you receive an error because the variables `italy`, `canada`, and `egypt` are not defined. If we do not use the quotes, R looks for variables with those names and returns an error.

We can also assign names using the `names` functions:

```
codes <- c(380, 124, 818)
country <- c("italy","canada","egypt")
names(codes) <- country
codes
```

```
italy canada  egypt
  380    124    818
```

# Vector can subset by names

If the elements have names, we can also access the entries using these names. Below are two examples.

```
codes["canada"]
```

```
canada
   124
```

```
codes[c("egypt","italy")]
```

```
egypt italy
  818   380
```

It is the same as we subset by index.

```
codes[c(3,1)]
```

```
egypt italy
  818   380
```

# Vectors with additional attributes: Factors

- Factors are built on top of integer vectors
- These have a fixed and known set of possible values.
- Factors have two components: level numbers (integers) and level labels (characters)

```
tmp <- factor(c("BS", "MS", "PhD", "MS"))
tmp
```

```
[1] BS  MS  PhD MS
Levels: BS MS PhD
```

```
as.integer(tmp)
```

```
[1] 1 2 3 2
```

> In summary, factors are a useful data type in R for representing categorical data with a fixed and known set of possible values.

# Vectors with additional attributes: Dates

- Dates and date-times are built on top of numeric vectors
- Dates are represented internally as the number of days since the origin, 1 Jan 1970

```
z <- as.Date("2020-01-01")
z
```

```
[1] "2020-01-01"
```

```
typeof(z)
```

```
[1] "double"
```

```
str(z)
```

```
 Date[1:1], format: "2020-01-01"
```

# Vectors with additional attributes: Dates

- Dates and date-times are built on top of numeric vectors
- Dates are represented internally as the number of days since the origin, 1 Jan 1970

```
z <- as.Date("2020-01-01")
z
```

```
[1] "2020-01-01"
```

```
typeof(z)
```

```
[1] "double"
```

```
str(z)
```

```
 Date[1:1], format: "2020-01-01"
```

# Vectors with additional attributes: Dates

```
z
```

```
[1] "2020-01-01"
```

```
as.integer(z)
```

```
[1] 18262
```

```
as.integer(z) / 365 # roughly 50 yrs
```

```
[1] 50.03288
```

> In summary, date vectors are useful for representing dates in R, and they are stored as numeric values representing the number of days since the origin.

We will talk more about packages later on in the class, but the `lubridate` package is particularly useful for dealing with dates.

# Summary

- We write programs by composing functions to manipulate data
- The basic data types represent Booleans, numbers, and characters
- Data structures group related values together
- Vectors group values of the same type
- Vector creating and subsetting

# Reading:

- R for Data Science Chapter 4, 20
- Additional reading: Matloff Chapter 2
- Chapter 2:R basics