Lecture1. Introducing R and Coding Basics

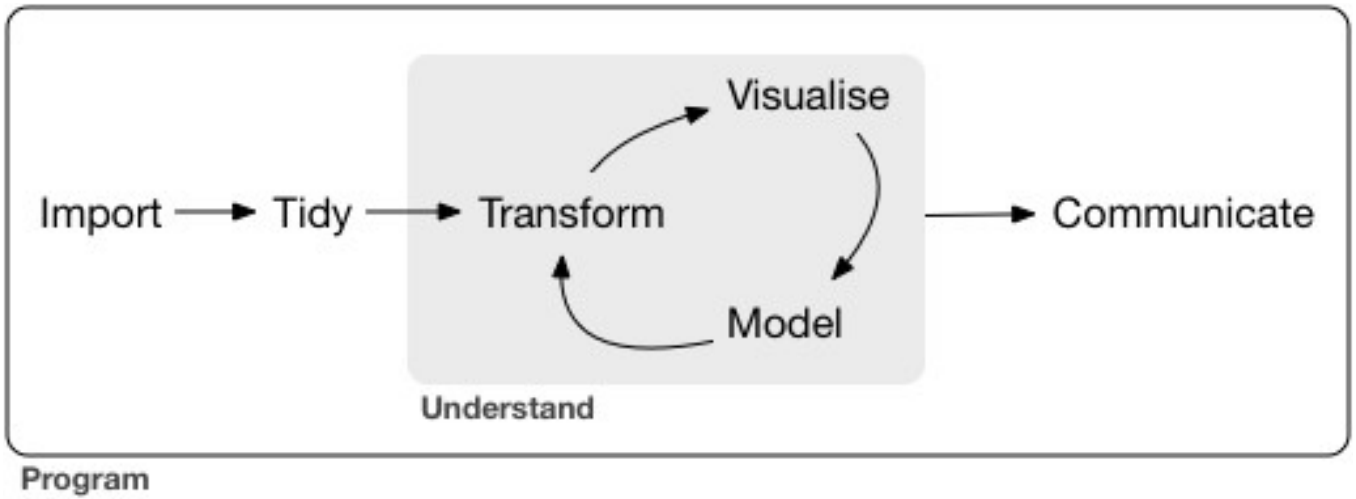
Liang Chen, liang.chen@whu.edu.cn

1st Week

Introduction

- Why this couse?
- The era of big data: online shopping, social media, and so on.
- Goal of this course: R language

What will you learn?



- Prerequisite: Probability, Statistics, and Econometrics

Grades: to be announced

- Assignments
- Attendance
- Exams

Install R and RStudio

- R
- RStudio
- Packages

Install R and RStudio

- R
- RStudio
- Packages: library(tidyverse)

Roadmap

- Coding basics
- Vectors

Coding basics (1)

- You can use R as a calculator:

```
1 / 200 * 30
```

```
## [1] 0.15
```

```
(59 + 73 + 2) / 3
```

```
## [1] 44.66667
```

```
sin(pi / 2)
```

```
## [1] 1
```

Coding basics (2)

-You can create new objects with <-:

```
x <- 3 * 4
```

All R statements where you create objects, **assignment** statements, have the same form:

```
object_name <- value
```

When reading that code say “object name gets value” in your head.

You will make lots of assignments and <- is a pain to type. Don’t be lazy and use =: it will work, but it will cause confusion later. Instead, use RStudio’s keyboard shortcut: Alt + - (the minus sign).

Coding basics (2)

- What's in a name?

Object names must start with a letter, and can only contain letters, numbers, _ and .. You want your object names to be descriptive, so you'll need a convention for multiple words. I recommend **snake_case** where you separate lowercase words with _.

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

We'll come back to code style later, in [functions].

Coding basics (2)

- You can inspect an object by typing its name.

```
x
```

```
## [1] 12
```

Make another assignment:

```
this_is_a_really_long_name <- 2.5
```

To inspect this object, try out RStudio's completion facility: type "this", press TAB, add characters until you have a unique prefix, then press return.

Ooops, you made a mistake! `this_is_a_really_long_name` should have value 3.5 not 2.5. Use another keyboard shortcut to help you fix it. Type "this" then press Cmd/Ctrl + ↑. That will list all the commands you've typed that start those letters. Use the arrow keys to navigate, then press enter to retype the command. Change 2.5 to 3.5 and rerun.

Coding basics (2)

-Make yet another assignment:

```
r_rocks <- 2 ^ 3
```

Let's try to inspect it:

```
r_rock
#> Error: object 'r_rock' not found
R_rocks
#> Error: object 'R_rocks' not found
```

There's an implied contract between you and R: it will do the tedious computation for you, but in return, you must be completely precise in your instructions. Typos matter. Case matters.

Coding basics (3)

- Paried quotation and parentheses

Type this code and notice you get similar assistance with the paired quotation marks:

```
x <- "hello world"
```

Quotation marks and parentheses must always come in a pair. RStudio does its best to help you, but it's still possible to mess up and end up with a mismatch. If this happens, R will show you the continuation character "+":

```
> x <- "hello
+
```

The + tells you that R is waiting for more input; it doesn't think you're done yet. Usually that means you've forgotten either a " or a). Either add the missing pair, or press ESCAPE to abort the expression and try again.

Coding basics (3)

- Calling functions

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

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

```
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

If you make an assignment, you don't get to see the value. You're then tempted to immediately double-check the result:

```
y <- seq(1, 10, length.out = 5)
y
```

```
## [1]  1.00  3.25  5.50  7.75 10.00
```

This common action can be shortened by surrounding the assignment with parentheses, which causes assignment and "print to screen" to happen.

```
(y <- seq(1, 10, length.out = 5))
```

```
## [1]  1.00  3.25  5.50  7.75 10.00
```

Practice

1. Why does this code not work?

```
my_variable <- 10
my_var??able
```

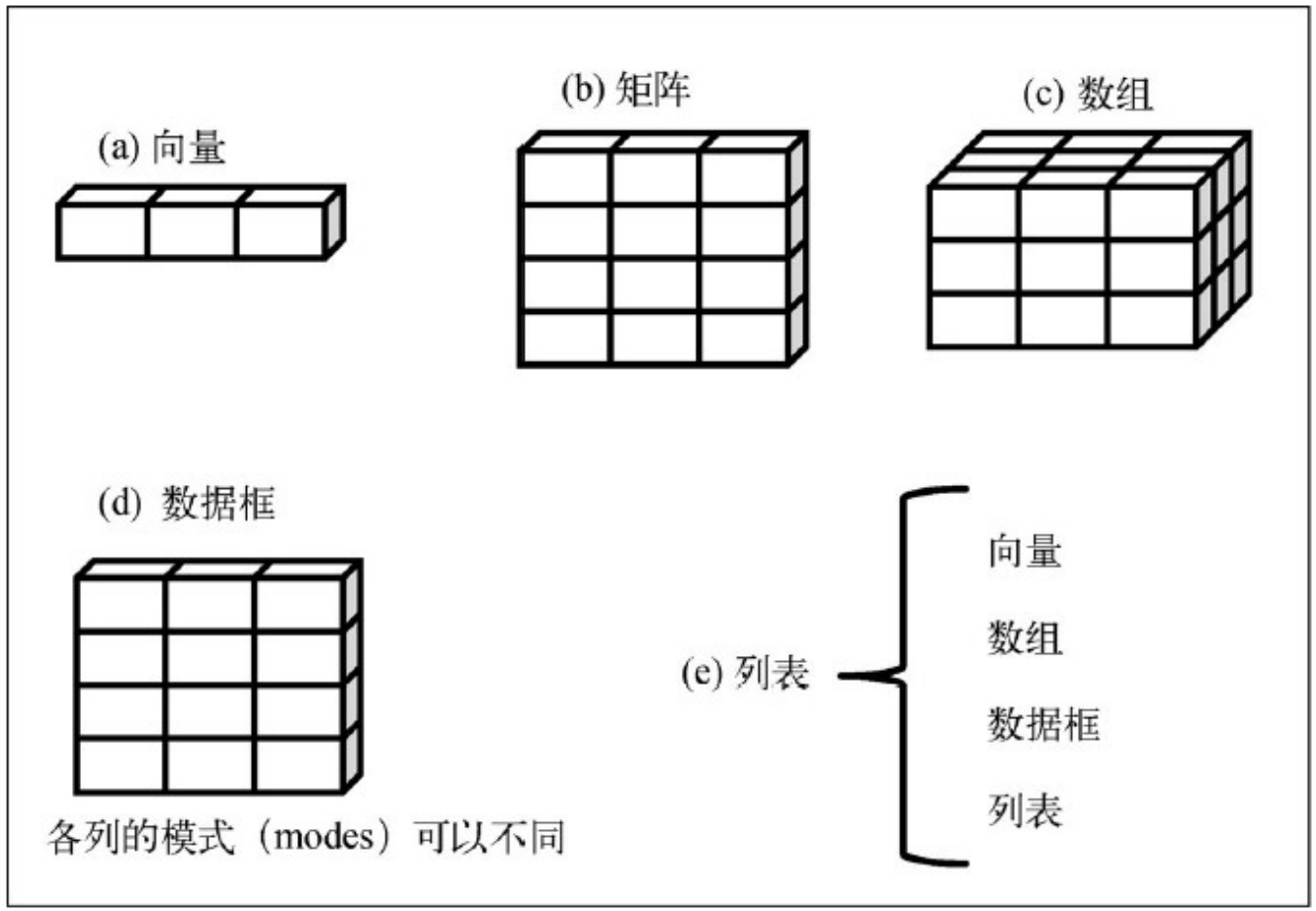
```
## Error in help.search("able", fields = "my_var", package = NULL): incorrect field specification
```

Look carefully! (This may seem like an exercise in pointlessness, but training your brain to notice even the tiniest difference will pay off when programming.)

2. Press Alt + Shift + K. What happens? How can you get to the same place using the menus?

Vectors(1)

Data Structure



Vectors(2)

Creating Vectors

- Using c()

```
x1 <- c(1,3,5,7,9)
x1
```

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

```
x2 <- c("One","Two","Three")
x2
```

```
## [1] "One"   "Two"   "Three"
```

```
x3 <- c(TRUE,F,T,NA)
x3
```

```
## [1]  TRUE FALSE  TRUE    NA
```

```
x4 <- 5
x4
```

```
## [1] 5
```

```
is.vector(x4)
```

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

Vectors(3)

Creating Vectors

- Using :

```
1:8
```

```
## [1] 1 2 3 4 5 6 7 8
```

- Using rep()

```
x5 <- rep(x = "a", times = 5)
x5
```

```
## [1] "a" "a" "a" "a" "a"
```

- Using seq()

```
x6 <- seq(from = 1, to = 10, by = 0.5)
x6
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
## [15]  8.0  8.5  9.0  9.5 10.0
```

Vectors(4)

Every vector has two key properties:

1. Its **type**, which you can determine with `typeof()`.

```
typeof(letters)
```

```
## [1] "character"
```

```
typeof(1:10)
```

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

2. Its **length**, which you can determine with `length()`.

```
x <- list("a", "b", 1:10)
length(x)
```

```
## [1] 3
```

Vectors(5)

Data types:

- logical, integer, double, and character.
- Logical: Take only three possible values: FALSE, TRUE, and NA. NULL vs. NA

```
c(TRUE, TRUE, FALSE, NA)
```

```
## [1]  TRUE  TRUE FALSE    NA
```

```
c(T, T, F, NA)
```

```
## [1]  TRUE  TRUE FALSE    NA
```

Results of comparision operations

```
5 > 3
```

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

```
5 < 3
```

```
## [1] FALSE
```

```
5 == 3
```

```
## [1] FALSE
```

Vectors(6)

Data types: Numeric

- Integer and double vectors are known collectively as numeric vectors.
- In R, numbers are doubles by default.
- To make an integer, place an `L` after the number:

```
typeof(1)
```

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

```
typeof(1L)
```

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

```
1.5L
```

```
## [1] 1.5
```

The distinction between integers and doubles is not usually important, but there are two important differences that you should be aware of:

1. Doubles are approximations. Doubles represent floating point numbers that can not always be precisely represented with a fixed amount of memory. This means that you should consider all doubles to be approximations. For example, what is square of the square root of two?

```
x <- sqrt(2) ^ 2
x
```

```
## [1] 2
```

```
x - 2
```

```
## [1] 4.440892e-16
```

This behaviour is common when working with floating point numbers: most calculations include some approximation error. Instead of comparing floating point numbers using `==`, you should use `dplyr::near()` which allows for some numerical tolerance.

2. Integers have one special value: `NA`, while doubles have four: `NA`, `NaN`, `Inf` and `-Inf`. All three special values `NaN`, `Inf` and `-Inf` can arise during division:

```
c(-1, 0, 1) / 0
```

```
## [1] -Inf  NaN  Inf
```

Avoid using `==` to check for these other special values. Instead use the helper functions `is.finite()`, `is.infinite()`, and `is.nan()`:

	0	Inf	NA	NaN
`is.finite()`		x		
`is.infinite()`		x		

	0	Inf	NA	NaN
is.na()			x	x
is.nan()				x

Vectors(7)

Character

Character vectors are the most complex type of atomic vector, because each element of a character vector is a string, and a string can contain an arbitrary amount of data.

Using vectors (1)

Now that you understand the different types of atomic vector, it's useful to review some of the important tools for working with them. These include:

1. How to convert from one type to another, and when that happens automatically.
2. How to tell if an object is a specific type of vector.
3. What happens when you work with vectors of different lengths.
4. How to name the elements of a vector.
5. How to pull out elements of interest.

Using vectors (2)

How to convert from one type to another?

There are two ways to convert one type of vector to another: Explicit coercion and Implicit coercion

1. Explicit coercion happens when you call a function like `as.logical()`, `as.integer()`, `as.double()`, or `as.character()`. Whenever you find yourself using explicit coercion, you should always check whether you can make the fix upstream, so that the vector never had the wrong type in the first place.

```
as.logical(3.6)
```

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

```
as.logical(0)
```

```
## [1] FALSE
```

```
as.integer(1.51)
```

```
## [1] 1
```

```
as.double(5L)
```

```
## [1] 5
```

```
as.character(TRUE)
```

```
## [1] "TRUE"
```

```
as.character(1)
```

```
## [1] "1"
```

Using vectors (3)

How to convert from one type to another?

- Implicit coercion happens when you use a vector in a specific context that expects a certain type of vector. For example, when you use a logical vector with a numeric summary function, or when you use a double vector where an integer vector is expected.

Because explicit coercion is used relatively rarely, and is largely easy to understand, I'll focus on implicit coercion here.

You've already seen the most important type of implicit coercion: using a logical vector in a numeric context. In this case `TRUE` is converted to `1` and `FALSE` converted to 0. That means the sum of a logical vector is the number of trues, and the mean of a logical vector is the proportion of trues:

```
x <- sample(1:20, 100, replace = TRUE)
y <- x > 10
sum(y)  # how many are greater than 10?
```

```
## [1] 48
```

```
mean(y) # what proportion are greater than 10?
```

```
## [1] 0.48
```

You may see some code (typically older) that relies on implicit coercion in the opposite direction, from integer to logical:

```

```

```
if (length(x)) {
  # do something
}
```

In this case, 0 is converted to `FALSE` and everything else is converted to `TRUE`. I think this makes it harder to understand your code, and I don't recommend it. Instead be explicit: `length(x) > 0`.

Using vectors (4)

How to convert from one type to another?

It's also important to understand what happens when you try and create a vector containing multiple types with `c()`: the most complex type always wins.

```
typeof(c(TRUE, 1L))
```

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

```
typeof(c(1L, 1.5))
```

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

```
typeof(c(1.5, "a"))
```

```
## [1] "character"
```

An atomic vector can not have a mix of different types because the type is a property of the complete vector, not the individual elements. If you need to mix multiple types in the same vector, you should use a list, which you'll learn about shortly.

Using vectors (5)

Test functions

Sometimes you want to do different things based on the type of vector. One option is to use `typeof()`. Another is to use a test function which returns a `TRUE` or `FALSE`. Base R provides many functions like `is.vector()` and `is.atomic()`, but they often returns surprising results. Instead, it's safer to use the `is_*` functions provided by purrr, which are summarised in the table below.

	lgl	int	dbl	chr	list
`is_logical()`	x				
`is_integer()`		x			
`is_double()`			x		
`is_numeric()`		x	x		
`is_character()`				x	
`is_atomic()`	x	x	x	x	
`is_list()`					x
`is_vector()`	x	x	x	x	x

Each predicate also comes with a "scalar" version, like `is_scalar_atomic()`, which checks that the length is 1. This is useful, for example, if you want to check that an argument to your function is a single logical value.

Using vectors (6)

Scalars and recycling rules

As well as implicitly coercing the types of vectors to be compatible, R will also implicitly coerce the length of vectors. This is called vector **recycling**, because the shorter vector is repeated, or recycled, to the same length as the longer vector.

This is generally most useful when you are mixing vectors and “scalars”. I put scalars in quotes because R doesn’t actually have scalars: instead, a single number is a vector of length 1. Because there are no scalars, most built-in functions are **vectorised**, meaning that they will operate on a vector of numbers. That’s why, for example, this code works:

```
sample(10) + 100
```

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

```
runif(10) > 0.5
```

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

Using vectors (7)

Scalars and recycling rules

In R, basic mathematical operations work with vectors. That means that you should never need to perform explicit iteration when performing simple mathematical computations.

It's intuitive what should happen if you add two vectors of the same length, or a vector and a "scalar", but what happens if you add two vectors of different lengths?

```
1:10 + 1:2
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

Here, R will expand the shortest vector to the same length as the longest, so called recycling. This is silent except when the length of the longer is not an integer multiple of the length of the shorter:

```
1:10 + 1:3
```

```
## Warning in 1:10 + 1:3: 长的对象长度不是短的对象长度的整倍数
```

```
##  [1]  2  4  6  5  7  9  8 10 12 11
```

While vector recycling can be used to create very succinct, clever code, it can also silently conceal problems. For this reason, the vectorised functions in tidyverse will throw errors when you recycle anything other than a scalar. If you do want to recycle, you'll need to do it yourself with `rep()`:

```
tibble(x = 1:4, y = 1:2)
```

```
## Error in tibble(x = 1:4, y = 1:2): 没有"tibble"这个函数
```

```
tibble(x = 1:4, y = rep(1:2, 2))
```

```
## Error in tibble(x = 1:4, y = rep(1:2, 2)): 没有"tibble"这个函数
```

```
tibble(x = 1:4, y = rep(1:2, each = 2))
```

```
## Error in tibble(x = 1:4, y = rep(1:2, each = 2)): 没有"tibble"这个函数
```

Using vectors (8)

Naming vectors

All types of vectors can be named. You can name them during creation with `c()`:

```
c(x = 1, y = 2, z = 4)
```

```
## x y z
## 1 2 4
```

Or set the names after creating the vector using names()

```
a <- 1:3
names(a) <- c("x", "y", "z")
```

Using vectors (9)

Subsetting Vector (1)

- [] is the subsetting function.
- There are FIVE types of things that you can subset a vector with:
- A numeric vector containing only integers. The integers must either be all positive, all negative, or zero

1. A numeric vector containing only integers. The integers must either be all positive, all negative, or zero.

   Subsetting with positive integers keeps the elements at those positions:

```
x <- c("one", "two", "three", "four", "five")
x[c(3, 2, 5)]
```

```
## [1] "three" "two"   "five"
```

   By repeating a position, you can actually make a longer output than input:

```
x[c(1, 1, 5, 5, 5, 2)]
```

```
## [1] "one"  "one"  "five" "five" "five" "two"
```

   Negative values drop the elements at the specified positions:

```

```

```
x[c(-1, -3, -5)]
```

```
## [1] "two"  "four"
```

It's an error to mix positive and negative values:

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

```
## Error in x[c(1, -1)]: 只有负下标里才能有零
```

The error message mentions subsetting with zero, which returns no values:

```
x[0]
```

```
## character(0)
```

This is not useful very often, but it can be helpful if you want to create unusual data structures to test your functions with.

2. Subsetting with a logical vector keeps all values corresponding to a `TRUE` value. This is most often useful in conjunction with the comparison functions.

```
x <- c(10, 3, NA, 5, 8, 1, NA)

# All non-missing values of x
x[!is.na(x)]
```

```
## [1] 10  3  5  8  1
```

```
# All even (or missing!) values of x
x[x %% 2 == 0]
```

```
## [1] 10 NA  8 NA
```

3. If you have a named vector, you can subset it with a character vector:

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
```

```
## xyz def
##   5   2
```

Like with positive integers, you can also use a character vector to duplicate individual entries.

4. The simplest type of subsetting is nothing, `x[]`, which returns the complete `x`. This is not useful for subsetting vectors, but it is useful when subsetting matrices (and other high dimensional structures) because it lets you select all the rows or all the columns, by leaving that index blank. For example, if `x` is 2d, `x[1, ]` selects the first row and all the columns, and `x[, -1]` selects all rows and all columns except the first.

5. using subset() function

```
 x <- 1:10
subset(x, x>5)
```

```
## [1]  6  7  8  9 10
```

```
 y <- c(rep(TRUE, 10), rep(NA,5))
 y
```

```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE   NA   NA   NA   NA
## [15]   NA
```

```
  subset(y, is.na(y))
```

```
## [1] NA NA NA NA NA
```

To learn more about the applications of subsetting, reading the “Subsetting” chapter of Advanced R: http://adv-r.had.co.nz/Subsetting.html#applications.