Lecture Note for Applied Econometrics

Yuta Toyama

Last updated: 2019/04/18

Contents

Chapter 1

Preface

Welcome to Applied Econometrics using R!

1.1 About this

This lecture note is maintained by Yuta Toyama.

1.2 Update: April 16, 2019

- Update chapter 3 (basic programming)
- Upload chapter 4 (Data frame)
- Upload chapter 5 (Exercise 1)

1.3 Acknowledgement (as of April 16, 2019)

- Chapter 2 through 4 are largely based on Applied Statistics with `R`. https://daviddalpiaz.github.io/appliedstats/

Chapter 2

Introduction to the course

2.1 What is econometrics?

1. Estimating economic relationships
    1. Demand curve $\log(Q_t) = \alpha_0 + \alpha_1 P_t + \epsilon_t$
    2. Production function $Y_{it} = A_{it} K_{it}^{\alpha} L_{it}^{\beta}$
2. Testing economic theory
    - Does adverse selection exists in insurance markets?
    - Are consumers rational?
3. Determine the effect of a given intervention (causal inference)
    - What is the effect of increasing minimum wage on employment?
    - Do mergers increase the output price?
    - Does democracy cause economic growth? (a series of works by Acemoglu, Robinsohn, and their co-authors).
    - Effects of going to private colleges on your future earnings.
    - Note: Some questions may have underlying economic models, others may not.
4. Describe the data (prediction/forecasting)
    - How does the distribution of wage look like?
    - Relationship between electricity consumption and temperature (possibly nonlinear).
    - Related to machine learning (ML).

2.2 Why do we need to learn computation

1. Conduct statistical and empirical analysis using your own data set
    1. Construct the data set
    2. Describe the data
    3. Run regression or estimate an economic object
    4. Make tables and figures that show the results of your analysis.
2. Verify the econometric theory through numerical simulations.
    - Ex. Asymptotic theory considers the case when the sample size is large enough (i.e., $N \to \infty$)
        - Law of large numbers, central limit theorem
        - How well is the asymptotic approximation?
    - **Monte Carlo simulations**

- We will learn both aspects in this course.

2.3 Why do we use R?

- Many alternatives: Stata, Matlab, Python, etc...

1. Free software!!
   - Stata and Matlab are expensive.
   - Though you can use Matlab through the campus license from this April.
2. Good balance between flexibility in programming and easy-to-use for econometric analysis
   - Stata is easy to use for econometric analysis, but hard to write your own program.
   - Matlab is the opposite.
   - You can do everything with R, including data construction, regression analysis, and complicated structural estimation.
3. Many users
   - Popular in engineering.
   - Many packages being developed (especially important for recently popular tools.)

- Note: Python seems also good, though I have not used it before.

Chapter 3

Introduction of `R` and `R studio`

3.1 Getting Started

- You can use R/R studio in the PC room.
- However, I strongly recommend you install R/Studio in your laptop and bring it to the class.
- Install in the following order
  1. R: https://www.r-project.org/
  2. Rstudio: https://www.rstudio.com/
- Now open Rstudio.

3.2 Helps

- The RStudio team has developed a number of "cheatsheets" for working with both `R` and RStudio.
- This particular cheatsheet for Base `R` will summarize many of the concepts in this document.

3.3 Quick tour of Rstudio

- There are four panels
  1. Source: Write your own code here.
  2. Console:
  3. Environment/History:
  4. Files/Plots/Packages/Help:
- In the Source panel,
  - Write your own code.
  - Save your code in `.R` file
  - Click `Run` command to run your entire code.
- In the concole panel,
  - After clicking `Run` in the source panel, your code is evaluated.
  - You can directly type your code here to implement.

3.4 Basic Calculations

To get started, we'll use `R` like a simple calculator.

Addition, Subtraction, Multiplication and Division

Math	R	Result
$3+2$	`3 + 2`	5
$3-2$	`3 - 2`	1
$3 \cdot 2$	`3 * 2`	6
$3/2$	`3 / 2`	1.5

Exponents

Math	R	Result
3^2	`3 ^ 2`	9
$2^{(-3)}$	`2 ^ (-3)`	0.125
$100^{1/2}$	`100 ^ (1 / 2)`	10
$\sqrt{100}$	`sqrt(100)`	10

Mathematical Constants

Math	R	Result
π	`pi`	3.1415927
e	`exp(1)`	2.7182818

Logarithms

- Note that we will use ln and log interchangeably to mean the natural logarithm.
- There is no `ln()` in R, instead it uses `log()` to mean the natural logarithm.

Math	R	Result
$\log(e)$	`log(exp(1))`	1
$\log_{10}(1000)$	`log10(1000)`	3
$\log_2(8)$	`log2(8)`	3
$\log_4(16)$	`log(16, base = 4)`	2

Trigonometry

Math	R	Result
$\sin(\pi/2)$	`sin(pi / 2)`	1
$\cos(0)$	`cos(0)`	1

3.5 Getting Help

- In using R as a calculator, we have seen a number of functions: `sqrt()`, `exp()`, `log()` and `sin()`.
- To get documentation about a function in R, simply put a question mark in front of the function name

and RStudio will display the documentation, for example:

```
?log
?sin
?paste
?lm
```

3.6 Installing Packages

- One of the main strengths of `R` as an open-source project is its package system.
- To install a package, use the `install.packages()` function.
  - Think of this as buying a recipe book from the store, bringing it home, and putting it on your shelf.

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

- Once a package is installed, it must be loaded into your current `R` session before being used.
  - Think of this as taking the book off of the shelf and opening it up to read.

```
library(ggplot2)
```

- Once you close `R`, all the packages are closed and put back on the imaginary shelf.
- The next time you open `R`, you do not have to install the package again, but you do have to load any packages you intend to use by invoking `library()`.

Chapter 4

Data and Programming

4.1 Data Types

`R` has a number of basic data *types*.

- Numeric
  - Also known as Double. The default type when dealing with numbers.
  - Examples: `1`, `1.0`, `42.5`
- Logical
  - Two possible values: `TRUE` and `FALSE`
  - You can also use `T` and `F`, but this is *not* recommended.
  - `NA` is also considered logical.
- Character
  - Examples: `"a"`, `"Statistics"`, `"1 plus 2."`

4.2 Data Structures

- `R` also has a number of basic data *structures*.
- A data structure is either
  - homogeneous (all elements are of the same data type)
  - heterogeneous (elements can be of more than one data type).

Dimension	**Homogeneous**	**Heterogeneous**
1	Vector	List
2	Matrix	Data Frame
3+	Array	

4.3 Vectors

4.3.1 Basics of vectors

- Many operations in `R` make heavy use of **vectors**.
  - Vectors in `R` are indexed starting at `1`.
- The most common way to create a vector in `R` is using the `c()` function, which is short for "combine."

```
c(1, 3, 5, 7, 8, 9)
```

```
## [1] 1 3 5 7 8 9
```

- If we would like to store this vector in a **variable** we can do so with the **assignment** operator `=`.
  - The variable `x` now holds the vector we just created, and we can access the vector by typing `x`.

```
x = c(1, 3, 5, 7, 8, 9)
x
```

```
## [1] 1 3 5 7 8 9
```

```
# The following does the same thing.
x <- c(1, 3, 5, 7, 8, 9)
x
```

```
## [1] 1 3 5 7 8 9
```

- The operator `=` and `<-` work as an assignment operator.
  - You can use both. This does not matter usually.
  - If you are interested in the weird cases where the difference matters, check out The R Inferno.
- In `R` code the line starting with `#` is `comment`, which is ignored when you run the fode.

- A vector based on a sequence of numbers.
- The quickest and easiest way to do this is with the `:` operator, which creates a sequence of integers between two specified integers.

```
(y = 1:100)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

- By putting parentheses around the assignment,
  - `R` both stores the vector in a variable called `y` and
  - automatically outputs `y` to the console.

4.3.2 Useful functions for creating vectors

- Use the `seq()` function for a more general sequence.

```
seq(from = 1.5, to = 4.2, by = 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
## [18] 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

- Here, the input labels `from`, `to`, and `by` are optional.

```
seq(1.5, 4.2, 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
## [18] 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

- The `rep()` function repeat a single value a number of times.

```
rep("A", times = 10)
```

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

- The `rep()` function can be used to repeat a vector some number of times.

```
rep(x, times = 3)
```

```
##  [1] 1 3 5 7 8 9 1 3 5 7 8 9 1 3 5 7 8 9
```

- We have now seen four different ways to create vectors:
  1. `c()`
  2. `:`
  3. `seq()`
  4. `rep()`
- They are often used together.

```
c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
```

```
##  [1]  1  3  5  7  8  9  1  3  5  7  9  1  3  5  7  9  1  3  5  7  9  1  2
## [24]  3 42  2  3  4
```

- The length of a vector can be obtained with the `length()` function.

```
length(x)
```

```
## [1] 6
```

```
length(y)
```

```
## [1] 100
```

4.3.3 Subsetting

- Use square brackets, `[]`, to obtain a subset of a vector.
- We see that `x[1]` returns the first element.

```
x
```

```
## [1] 1 3 5 7 8 9
```

```
x[1]
```

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

```
x[3]
```

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

- We can also exclude certain indexes, in this case the second element.

```
x[-2]
```

```
## [1] 1 5 7 8 9
```

- We can subset based on a vector of indices.

```
x[1:3]
```

```
## [1] 1 3 5
```

```
x[c(1,3,4)]
```

```
## [1] 1 5 7
```

- We could instead use a vector of logical values.

```
z = c(TRUE, TRUE, FALSE, TRUE, TRUE, FALSE)
z
```

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

```
x[z]
```

```
## [1] 1 3 7 8
```

4.4 Vectorization

- One of the biggest strengths of R is its use of vectorized operations.
  - Frequently the lack of understanding of this concept leads of a belief that R is *slow*.
  - R is not the fastest language, but it has a reputation for being slower than it really is.)
- When a function like `log()` is called on a vector `x`, a vector is returned which has applied the function to each element of the vector `x`.

```
x = 1:10
x + 1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
2 * x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
2 ^ x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
log(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

4.5 Logical Operators

Operator	Summary	Example	Result
`x < y`	x less than y	`3 < 42`	TRUE
`x > y`	x greater than y	`3 > 42`	FALSE
`x <= y`	x less than or equal to y	`3 <= 42`	TRUE
`x >= y`	x greater than or equal to y	`3 >= 42`	FALSE
`x == y`	xequal to y	`3 == 42`	FALSE
`x != y`	x not equal to y	`3 != 42`	TRUE
`!x`	not x	`!(3 > 42)`	TRUE
`x | y`	x or y	`(3 > 42) | TRUE`	TRUE
`x & y`	x and y	`(3 < 4) & ( 42 > 13)`	TRUE

- Logical operators are vectorized.

```
x = c(1, 3, 5, 7, 8, 9)
```

```
x > 3
```

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

```
x < 3
```

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

```
x == 3
```

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

```
x != 3
```

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

```
x == 3 & x != 3
```

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

```
x == 3 | x != 3
```

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

- This is extremely useful for subsetting.

```
x[x > 3]
```

```
## [1] 5 7 8 9
```

```
x[x != 3]
```

```
## [1] 1 5 7 8 9
```

4.5.0.1 Short exercise

1. Create the vector $z = (1, 2, 1, 2, 1, 2)$, which has the same length as x.
2. Pick up the elements of x which corresponds to 1 in the vector z.

4.6 Matrices

4.6.1 Basics

- `R` can also be used for **matrix** calculations.
- Matrices have rows and columns containing a single data type.
- Matrices can be created using the `matrix` function.

```
x = 1:9
x
```

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

```
X = matrix(x, nrow = 3, ncol = 3)
X
```

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

```
## [3,]    3    6    9
```

- We are using two different variables:
  - lower case `x`, which stores a vector and
  - capital `X`, which stores a matrix.
- By default the `matrix` function reorders a vector into columns, but we can also tell `R` to use rows instead.

```
Y = matrix(x, nrow = 3, ncol = 3, byrow = TRUE)
Y
```

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

- a matrix of a specified dimension where every element is the same, in this case `0`.

```
Z = matrix(0, 2, 4)
Z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
```

- Matrices can be subsetted using square brackets, `[]`.
- However, since matrices are two-dimensional, we need to specify both a row and a column when subsetting.
- Here we get the element in the first row and the second column.

```
X
```

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

```
X[1, 2]
```

```
## [1] 4
```

- We could also subset an entire row or column.

```
X[1, ]
```

```
## [1] 1 4 7
```

```
X[, 2]
```

```
## [1] 4 5 6
```

- Matrices can also be created by combining vectors as columns, using `cbind`, or combining vectors as rows, using `rbind`.

```
x = 1:9
rev(x)
```

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

```
rep(1, 9)
```

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

```
rbind(x, rev(x), rep(1, 9))
```

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

- When using `rbind` and `cbind` you can specify "argument" names that will be used as column names.

```
cbind(col_1 = x, col_2 = rev(x), col_3 = rep(1, 9))
```

```
##       col_1 col_2 col_3
##  [1,]     1     9     1
##  [2,]     2     8     1
##  [3,]     3     7     1
##  [4,]     4     6     1
##  [5,]     5     5     1
##  [6,]     6     4     1
##  [7,]     7     3     1
##  [8,]     8     2     1
##  [9,]     9     1     1
```

4.6.2 Matrix calculations

- Perform matrix calculations.

```
x = 1:9
y = 9:1
X = matrix(x, 3, 3)
Y = matrix(y, 3, 3)
X
```

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

```
Y
```

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

```
X + Y
```

```
##      [,1] [,2] [,3]
## [1,]   10   10   10
## [2,]   10   10   10
## [3,]   10   10   10
```

```
X - Y
```

```
##      [,1] [,2] [,3]
## [1,]   -8   -2    4
## [2,]   -6    0    6
## [3,]   -4    2    8
```

```
X * Y
```

```
##      [,1] [,2] [,3]
## [1,]    9   24   21
## [2,]   16   25   16
## [3,]   21   24    9
```

```
X / Y
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.1111111 0.6666667 2.333333
## [2,] 0.2500000 1.0000000 4.000000
## [3,] 0.4285714 1.5000000 9.000000
```

- Note that `X * Y` is **not** matrix multiplication.
- It is element by element multiplication. (Same for `X / Y`).
- Matrix multiplication uses `%*%`.
- `t()` which gives the transpose of a matrix

```
X %*% Y
```

```
##      [,1] [,2] [,3]
## [1,]   90   54   18
## [2,]  114   69   24
## [3,]  138   84   30
```

```
t(X)
```

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

- `solve()` which returns the inverse of a square matrix if it is invertible.

```
Z = matrix(c(9, 2, -3, 2, 4, -2, -3, -2, 16), 3, byrow = TRUE)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    9    2   -3
## [2,]    2    4   -2
## [3,]   -3   -2   16
```

```
solve(Z)
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.12931034 -0.05603448 0.01724138
## [2,] -0.05603448  0.29094828 0.02586207
## [3,]  0.01724138  0.02586207 0.06896552
```

- To verify that `solve(Z)` returns the inverse, we multiply it by `Z`.
  - We would expect this to return the identity matrix.
  - However we see that this is not the case due to some computational issues.
  - However, `R` also has the `all.equal()` function which checks for equality, with some small tolerance which accounts for some computational issues.

```
solve(Z) %*% Z
```

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

```
## [1,] 1.0000000000000000000000 -0.0000000000000006245005
## [2,] 0.0000000000000008326673  1.0000000000000022204460
## [3,] 0.0000000000000002775558  0.0000000000000000000000
##                              [,3]
## [1,] 0.0000000000000000000000
## [2,] 0.0000000000000005551115
## [3,] 1.0000000000000000000000
```

```
diag(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
all.equal(solve(Z) %*% Z, diag(3))
```

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

4.6.2.1 Exercise

- Solve the following simultanoues equations using matrix calculation

$$2x_1 + 3x_2 = 105x_1 + x_2 = 20$$

- Hint: You can write this as $Ax = y$ where A is the 2-times-2 matrix, x and y are vectors with the length of 2.

4.6.3 Getting information for matrix

- `R` has a number of matrix specific functions for obtaining dimension and summary information.

```
X = matrix(1:6, 2, 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(X)
```

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

```
rowSums(X)
```

```
## [1]  9 12
```

```
colSums(X)
```

```
## [1]  3  7 11
```

```
rowMeans(X)
```

```
## [1] 3 4
```

```
colMeans(X)
```

```
## [1] 1.5 3.5 5.5
```

- The `diag()` function can be used in a number of ways. We can extract the diagonal of a matrix.

```
diag(Z)
```

```
## [1]  9  4 16
```

- Or create a matrix with specified elements on the diagonal. (And `0` on the off-diagonals.)

```
diag(1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    3    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    5
```

- Or, lastly, create a square matrix of a certain dimension with `1` for every element of the diagonal and `0` for the off-diagonals.

```
diag(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

4.7 Lists

- A list is a one-dimensional heterogeneous data structure.
  - It is indexed like a vector with a single integer value,
  - but each element can contain an element of any type.

```
# creation
list(42, "Hello", TRUE)
```

```
## [[1]]
## [1] 42
##
## [[2]]
## [1] "Hello"
##
## [[3]]
## [1] TRUE
```

```
ex_list = list(
  a = c(1, 2, 3, 4),
  b = TRUE,
  c = "Hello!",
  d = function(arg = 42) {print("Hello World!")},
  e = diag(5)
)
```

- Lists can be subset using two syntaxes,
  1. the `$` operator, and
  2. square brackets `[]`.

```
# subsetting
ex_list$e
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```
ex_list[1:2]
```

```
## $a
## [1] 1 2 3 4
##
## $b
## [1] TRUE
```

```
ex_list[1]
```

```
## $a
## [1] 1 2 3 4
```

```
ex_list[c("e", "a")]
```

```
## $e
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
##
## $a
## [1] 1 2 3 4
```

```
ex_list["e"]
```

```
## $e
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```
ex_list$d
```

```
## function(arg = 42) {print("Hello World!")}
```

4.8 Data Frames

- We will talk about Dataframe in the next chapter.

4.9 Programming Basics -Control flow-

4.9.1 if/else

- The if/else syntax is:

```
if (...) {
  some R code
} else {
  more R code
}
```

- Example: To see whether x is large than y.

```
x = 1
y = 3
if (x > y) {
  z = x * y
  print("x is larger than y")
} else {
  z = x + 5 * y
  print("x is less than or equal to y")
}
```

```
## [1] "x is less than or equal to y"
```

```
z
```

```
## [1] 16
```

- `R` also has a special function `ifelse()`
  - It returns one of two specified values based on a conditional statement.

```
ifelse(4 > 3, 1, 0)
```

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

- The real power of `ifelse()` comes from its ability to be applied to vectors.

```
fib = c(1, 1, 2, 3, 5, 8, 13, 21)
ifelse(fib > 6, "Foo", "Bar")
```

```
## [1] "Bar" "Bar" "Bar" "Bar" "Bar" "Foo" "Foo" "Foo"
```

4.10 `for` loop

- A `for` loop repeats the same procedure for the specified number of times

```
x = 11:15
for (i in 1:5) {
  x[i] = x[i] * 2
}

x
```

```
## [1] 22 24 26 28 30
```

- Note that this `for` loop is very normal in many programming languages.
- In `R` we would not use a loop, instead we would simply use a vectorized operation.

  - `for` loop in `R` is known to be very slow.

```
x = 11:15
x = x * 2
x
```

```
## [1] 22 24 26 28 30
```

4.11 Functions

- To use a function,
  - you simply type its name,
  - followed by an open parenthesis,
  - then specify values of its arguments,
  - then finish with a closing parenthesis.
- An **argument** is a variable which is used in the body of the function.

```
# The following is just a demonstration, not the real function in R.
function_name(arg1 = 10, arg2 = 20)
```

- We can also write our own functions in `R`.
- Example: "standardize" variables

$$\frac{x - \bar{x}}{s}$$

- When writing a function, there are three thing you must do.
  1. Give the function a name. Preferably something that is short, but descriptive.
  2. Specify the arguments using `function()`
  3. Write the body of the function within curly braces, `{}`.

```
standardize = function(x) {
  m = mean(x)
  std = sd(x)
  result = (x - m) / std
  return(result)
}
```

- Here the name of the function is `standardize`,
- The function has a single argument `x` which is used in the body of function.
- Note that the output of the final line of the body is what is returned by the function.
- Let's test our function
- Take a random sample of size `n = 10` from a normal distribution with a mean of `2` and a standard deviation of `5`.

```
test_sample = rnorm(n = 10, mean = 2, sd = 5)
test_sample
```

```
##  [1]  5.883677 -3.618779  6.728653  1.217414  4.882808  6.729246  2.781511
##  [8]  3.438332  4.843239  7.607464
```

```
standardize(x = test_sample)
```

```
##  [1]  0.5488951 -2.2945834  0.8017425 -0.8474198  0.2493987  0.8019200
##  [7] -0.3793853 -0.1828406  0.2375584  1.0647143
```

- The same function can be written more simply.

```
standardize = function(x) {
  (x - mean(x)) / sd(x)
}
```

- When specifying arguments, you can provide default arguments.

```
power_of_num = function(num, power = 2) {
  num ^ power
}
```

- Let's look at a number of ways that we could run this function to perform the operation `10^2` resulting in `100`.

```
power_of_num(10)
```

```
## [1] 100
```

```
power_of_num(10, 2)
```

```
## [1] 100
```

```
power_of_num(num = 10, power = 2)
```

```
## [1] 100
```

```
power_of_num(power = 2, num = 10)
```

```
## [1] 100
```

- Note that without using the argument names, the order matters. The following code will not evaluate to the same output as the previous example.

```
power_of_num(2, 10)
```

```
## [1] 1024
```

- Also, the following line of code would produce an error since arguments without a default value must be specified.

```
power_of_num(power = 5)
```

- To further illustrate a function with a default argument, we will write a function that calculates sample variance two ways.
- By default, the function will calculate the unbiased estimate of σ^2, which we will call s^2.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x - \bar{x})^2$$

- It will also have the ability to return the biased estimate (based on maximum likelihood) which we will call $\hat{\sigma}^2$.

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x - \bar{x})^2$$

```
get_var = function(x, unbiased = TRUE) {

  if (unbiased == TRUE){
    n = length(x) - 1
```

```
  } else if (unbiased == FALSE){
    n = length(x)
   }

  (1 / n) * sum((x - mean(x)) ^ 2)
}
```

```
get_var(test_sample)
```

```
## [1] 11.16791
```

```
get_var(test_sample, unbiased = TRUE)
```

```
## [1] 11.16791
```

```
var(test_sample)
```

```
## [1] 11.16791
```

- We see the function is working as expected, and when returning the unbiased estimate it matches `R`'s built in function `var()`. Finally, let's examine the biased estimate of σ^2.

```
get_var(test_sample, unbiased = FALSE)
```

```
## [1] 10.05112
```

Chapter 5

Data frame

5.1 Introduction

- A **data frame** is the most common way that we store and interact with data in this course.

```
example_data = data.frame(x = c(1, 3, 5, 7, 9, 1, 3, 5, 7, 9),
                          y = c(rep("Hello", 9), "Goodbye"),
                          z = rep(c(TRUE, FALSE), 5))
```

- A data frame is a **list** of vectors.
  - Each vector must contain the same data type
  - The difference vectors can store different data types.

```
example_data
```

```
##    x       y     z
## 1  1   Hello  TRUE
## 2  3   Hello FALSE
## 3  5   Hello  TRUE
## 4  7   Hello FALSE
## 5  9   Hello  TRUE
## 6  1   Hello FALSE
## 7  3   Hello  TRUE
## 8  5   Hello FALSE
## 9  7   Hello  TRUE
## 10 9 Goodbye FALSE
```

- `write.csv` save (or export) the dataframe in `.csv` format.

5.2 Load csv file

- We can also import data from various file types in into `R`, as well as use data stored in packages.
- Read `csv` file into R.
  - `read.csv()` function as default
  - `read_csv()` function from the `readr` package. This is faster for larger data.

```
# install.packages("readr")
library(readr)
example_data_from_csv = read_csv("example-data.csv")
```

- Note: This particular line of code assumes that the file `example_data.csv` exists in your current working directory.
- The current working directory is the folder that you are working with. To see this, you type

```
getwd()
```

```
## [1] "C:/Users/Yuta/Dropbox/Teaching/2019S_Applied_Econometrics_JPN_ENG/Material_Github"
```

- If you want to set the working directory, use `setwd()` function

```
setwd(dir = "directory path" )
```

5.3 Examine dataframe

- Inside the `ggplot2` package is a dataset called `mpg`. By loading the package using the `library()` function, we can now access `mpg`.

```
library(ggplot2)
```

- Three things we would generally like to do with data:
  - Look at the raw data.
  - Understand the data. (Where did it come from? What are the variables? Etc.)
  - Visualize the data.
- To look at the data, we have two useful commands: `head()` and `str()`

```
head(mpg, n = 10)
```

```
## # A tibble: 10 x 11
##    manufacturer model displ  year   cyl trans drv     cty   hwy fl    cla~
##    <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <ch>
##  1 audi         a4      1.8  1999     4 auto~ f        18    29 p     com~
##  2 audi         a4      1.8  1999     4 manu~ f        21    29 p     com~
##  3 audi         a4      2    2008     4 manu~ f        20    31 p     com~
##  4 audi         a4      2    2008     4 auto~ f        21    30 p     com~
##  5 audi         a4      2.8  1999     6 auto~ f        16    26 p     com~
##  6 audi         a4      2.8  1999     6 manu~ f        18    26 p     com~
##  7 audi         a4      3.1  2008     6 auto~ f        18    27 p     com~
##  8 audi         a4 q~   1.8  1999     4 manu~ 4        18    26 p     com~
##  9 audi         a4 q~   1.8  1999     4 auto~ 4        16    25 p     com~
## 10 audi         a4 q~   2    2008     4 manu~ 4        20    28 p     com~
```

- The function `str()` will display the "structure" of the data frame.
  - It will display the number of **observations** and **variables**, list the variables, give the type of each variable, and show some elements of each variable.
  - This information can also be found in the "Environment" window in RStudio.

```
str(mpg)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    234 obs. of  11 variables:
##  $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
##  $ model       : chr  "a4" "a4" "a4" "a4" ...
##  $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr  "f" "f" "f" "f" ...
##  $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
```

```
##  $ hwy          : int  29 29 31 30 26 26 27 26 25 28 ...
##  $ fl           : chr  "p" "p" "p" "p" ...
##  $ class        : chr  "compact" "compact" "compact" "compact" ...
```

- `names()` function to obtain names of the variables in the dataset

```
names(mpg)
```

```
##  [1] "manufacturer" "model"        "displ"        "year"
##  [5] "cyl"          "trans"        "drv"          "cty"
##  [9] "hwy"          "fl"           "class"
```

- To access one of the variables **as a vector**, we use the `$` operator.

```
mpg$year
```

```
##   [1] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008
##  [15] 2008 1999 2008 2008 2008 2008 2008 1999 2008 1999 1999 2008 2008 2008
##  [29] 2008 2008 1999 1999 1999 2008 1999 2008 2008 1999 1999 1999 1999 2008
##  [43] 2008 2008 1999 1999 2008 2008 2008 2008 1999 1999 2008 2008 2008 1999
##  [57] 1999 1999 2008 2008 2008 1999 2008 1999 2008 2008 2008 2008 2008 2008
##  [71] 1999 1999 2008 1999 1999 1999 2008 1999 1999 1999 2008 2008 1999 1999
##  [85] 1999 1999 1999 2008 1999 2008 1999 1999 2008 2008 1999 1999 2008 2008
##  [99] 2008 1999 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 2008 2008
## [113] 1999 1999 2008 1999 1999 2008 2008 2008 2008 2008 2008 2008 1999 1999
## [127] 2008 2008 2008 2008 1999 2008 2008 1999 1999 1999 2008 1999 2008 2008
## [141] 1999 1999 1999 2008 2008 2008 2008 1999 1999 2008 1999 1999 2008 2008
## [155] 1999 1999 1999 2008 2008 1999 1999 2008 2008 2008 2008 1999 1999 1999
## [169] 1999 2008 2008 2008 2008 1999 1999 1999 1999 2008 2008 1999 1999 2008
## [183] 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008 1999 1999 1999
## [197] 2008 2008 1999 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008
## [211] 2008 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 1999 1999 1999
## [225] 1999 2008 2008 1999 1999 2008 2008 1999 1999 2008
```

```
mpg$hwy
```

```
##   [1] 29 29 31 30 26 26 27 26 25 28 27 25 25 25 25 24 25 23 20 15 20 17 17
##  [24] 26 23 26 25 24 19 14 15 17 27 30 26 29 26 24 24 22 22 24 24 17 22 21
##  [47] 23 23 19 18 17 17 19 19 12 17 15 17 17 12 17 16 18 15 16 12 17 17 16
##  [70] 12 15 16 17 15 17 17 18 17 19 17 19 19 17 17 17 16 16 17 15 17 26 25
##  [93] 26 24 21 22 23 22 20 33 32 32 29 32 34 36 36 29 26 27 30 31 26 26 28
## [116] 26 29 28 27 24 24 24 22 19 20 17 12 19 18 14 15 18 18 15 17 16 18 17
## [139] 19 19 17 29 27 31 32 27 26 26 25 25 17 17 20 18 26 26 27 28 25 25 24
## [162] 27 25 26 23 26 26 26 26 25 27 25 27 20 20 19 17 20 17 29 27 31 31 26
## [185] 26 28 27 29 31 31 26 26 27 30 33 35 37 35 15 18 20 20 22 17 19 18 20
## [208] 29 26 29 29 24 44 29 26 29 29 29 29 23 24 44 41 29 26 28 29 29 29 28
## [231] 29 26 26 26
```

- We can use the `dim()`, `nrow()` and `ncol()` functions to obtain information about the dimension of the data frame.

```
dim(mpg)
```

```
## [1] 234  11
```

```
nrow(mpg)
```

```
## [1] 234
```

```
ncol(mpg)
```

```
## [1] 11
```

5.4 Subsetting data

- Subsetting data frames can work much like subsetting matrices using square brackets, `[,]`.
- Here, we find fuel efficient vehicles earning over 35 miles per gallon and only display `manufacturer`, `model` and `year`.

```
mpg[mpg$hwy > 35, c("manufacturer", "model", "year")]
```

```
## # A tibble: 6 x 3
##   manufacturer model       year
##   <chr>        <chr>      <int>
## 1 honda        civic       2008
## 2 honda        civic       2008
## 3 toyota       corolla     2008
## 4 volkswagen   jetta       1999
## 5 volkswagen   new beetle  1999
## 6 volkswagen   new beetle  1999
```

- An alternative would be to use the `subset()` function, which has a much more readable syntax.

```
subset(mpg, subset = hwy > 35, select = c("manufacturer", "model", "year"))
```

- Lastly, we could use the `filter` and `select` functions from the `dplyr` package which introduces the `%>%` operator from the `magrittr` package.

```
library(dplyr)
mpg %>%
filter(hwy > 35) %>%
select(manufacturer, model, year)
```

- I will give you an assignment about `dplyr` package in the `DataCamp` as a makeup lecture.

Chapter 6

Exercise 1

- Due date: April 22th (Monday) 11pm

Rules for Problem Sets

- If you are enrolled in Japanese class (i.e., Wednesday 2nd), you can use both Japanese and English to write your answer.
- Submit your solution through `CourseN@vi`.
- Submit both your answer and R script.
- Using `Rmarkdown` would be appreciated, though not mandatory.
  - `Rmarkdown` introduction in Japanese: https://kazutan.github.io/kazutanR/Rmd_intro.html
  - `Rmarkdown` introduction in English: https://rmarkdown.rstudio.com/articles_intro.html
- I might cover `Rmarkdown` in the course later.

6.1 Update (as of 10am, April 18th)

- Please calculate the standard deviation, not the variance in your simulation. The parameter you set when drawing the random number is the mean μ and the standard deviation σ in normal distribution.
- Use `seq` function to create the sequence of the sample sizes that you use in the simulation.

6.2 Question: Examine the law of large numbers through numerical simulations

Consider the random sample of $\{x_i\}_{i=1}^N$ drawn from the random variable X. The law of large numbers implies that

$$\frac{1}{N}\sum_{i=1}^{N} x_i \xrightarrow{p} E[X]$$

In other words, the sample mean converges to the population mean in probability as the sample size goes to infinity (i.e., $N \to \infty$).

Similary, the sample variance also converges to the population variance in probability

$$\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 \xrightarrow{p} V[X]$$

** (Note on 4/18) This implies that the sample standard deviation also converges to the population standard deviation ** (This is an application of the law of large numbers, though it is a bit involved to prove this.)

The goal of this problem set is to demonstrate these two properties through numerical simulations. Here is what we are going to do:

1. For a certain sample size N, draw N random numbers from the normal distribution with known mean and standard deviation.
2. Calculate the sample mean and the sample variance for the "data" you draw.
3. Repeat this for many different sample sizes.
4. Examine to see whether the sample mean and **standard deviation** are getting closer to the true value, which you set when you draw the random numbers, as the sample size gets larger.

6.2.1 How to implement

I explain how to implement this in `R` step by step below.

1. Prepare a function like this
    1. There are two inputs: (1) a vector that contains the data $\{x_i\}_{i=1}^N$ and (2) the indicator of whether you calculate the mean or the standard deviation.

```
fun_something = function(firstinput, secondinput){

  # Two inputs: firstinput, secondinput
  # One output: output

  # Do something.

  return(output)

}
```

    2. Use if/else sentence. Example:

```
# "secondinput" is the name of the input variable in your function
if ( secondinput == "mean"){
  # calculate mean of the data (firstinput)

} else if ( secondinput == "sd"){
  # Calculate standard deviation of the data (firstinput)
}
```

    3. Use `return` function to define the output of the function.
2. Construct a vector that contains the sample size you want to use in your simulation. For example:

```
samplesize_vec = seq(from = 100, to = 100000, by = 100)
```

Here, let's try 100 different sample sizes that ranges from 100 to 100000.

3. Prepare two vectors that contain the result in the `forloop` below. Since we are trying 100 different sample sizes, let's create a vector with the length of 100.

```
# Hint:
# numeric(k) returns a zero vector with the length of k
# length( vector) returns the length of `vector`
```

```
# result_mean = ....
# result_sd = ....
```

4. To create the random draw from the normal distribution, use below

```
# You can choose the mean and the standard deviation as you like.
rnorm(n = 100, mean = 2, sd = 5)
```

4. Use `forloop` to calculate both mean and the standard deviation for each sample size. For example:

```
for (i in 1:length(samplesize_vec)){

  # Draw the random number

  # Calculate the mean using the function you construct.

  # Calculate the standard deviation using the function you construct.

}
```

5. Plot the result with `ggplot2`.
    1. Install the package if you have not done it yet.
    2. Load `ggplot2` by `library(ggplot2)`
    3. Use `qplot` command to make a figure

```
# Create plot and save it as the variable `plot1`
plot1 <- qplot(x = samplesize_vec, y = yourresult, geom = "line")

# print "plot1"
print(plot1)

# save the plot as PNG file
ggsave(file = "filename.png", plot = plot1)
```

6.2.2 What to submit

Your answer should include

1. The true value of mean and variance you choose in your simulation.
2. The plot that describes the relationship between the sample mean (variance) and the sample size.
3. Explain what the plots from your simulation indicate.