補足：Rの基本

講師：遠山祐太

最終更新：2024-11-16

はじめに

- Rのプログラミングに関する入門事項についての解説。
- 注意：暫く更新していないため、タイポやエラーなどが多々ある可能性。
- 謝辞：本スライドは Applied Statistics with R に基づいています。

データ

データ型

R はいくつもデータの型をもつ。

- Numeric（数値型）
  - R には Integer（整数型）と Double（実数型）がある。数値を扱うときの基本の型である。
  - Examples: `1`, `1.0`, `42.5`
- Logical（論理型）
  - `TRUE` と `FALSE` の真偽二値のみとる。
  - `T` と `F` とも書けるが、**非推奨**。
  - `R では `NA` も logical である。
- Character（文字型）
  - 例：`"a"`/`"Statistics"`/`"1 plus 2."`

データ構造

- R は基本的なデータ構造をもつ。
- データ構造は以下のいずれもある。
  - 同質的（全ての要素が同じデータ型をもつ）
  - 異質的 （要素が異なるデータ型をもつ）

次元	同質的	異質的
1	ベクトル（vector）	リスト（list）
2	行列（matrix）	データフレーム（data frame）
3+	配列（array）	

ベクトル

ベクトル

ベクトルの基本

- R における多くの操作で**ベクトル**を使う。
  - R の**添字は 1 からはじまる。**
- R でベクトルを作成するのによく用いられるのは`c()` である（combine の略）。

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

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

代入

- ベクトルを**変数**に格納したければ、**代入演算子（assignment operator）** `=` を用いる。
  - 変数 `x` にベクトルを格納しているので、`x` と入力すればベクトルをよびだせる。

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

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

```
# 次も同じ
x <- c(1, 3, 5, 7, 8, 9)
x
```

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

代入に関するTips

- R では = と <- が代入演算子として問題なく使える（大体の場合は）。
  - 問題になる例は The R Inferno を参照。
- R では、行の先頭に # をつけるとコメント行としてみなされ、実行されない。

数字の列

- 数列の作成には `:` 演算子を使うのが最もお手軽である。

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

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

- かっこをつけると、R は
  - 変数 y に数列を格納する。
  - 変数 y をコンソールに表示する。

ベクトル作成に便利な関数

- 数列の作成には `seq()` 関数の汎用性が高い。

```
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 3.2 3.3
## [20] 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

- `from`・`to`・`by` はあってもなくてもよい（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 3.2 3.3
## [20] 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

続・ベクトル作成に便利な関数

- `rep()` 関数は同じ値を指定の回数出力する。

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

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

- よって、`rep()` 関数は要素が等しい数列の作成に便利である。

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

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

ここまでのまとめ

- 4 種類のことなる手法をみてきた。

    1. `c()`
    2. `:`
    3. `seq()`
    4. `rep()`
    - 一緒に使われることも多い。

```
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  3 42
## [26]  2  3  4
```

ベクトルの長さ

- ベクトルの長さは `length()` 関数で取得できる。

```
length(x)
```

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

```
length(y)
```

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

要素

- 四角かっこ `[]` をつけると、ベクトルの要素を取得できる。
- `x[1]` は `x` の第一要素を返す。Rにおいては添字が1からはじまることに注意！

```
x
```

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

```
x[1]
```

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

```
x[3]
```

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

要素続き

- 特定の要素を省いたものを取得することもできる。以下では第二要素を除いている。

```
x[-2]
```

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

- 添字はベクトルでもよい。

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

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

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

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

要素続き

-論理値ベクトルを添字に用いてもよい。

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

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

```
x[z]
```

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

ベクトル化

- R の強みの一つは、ベクトル化された（vectorized）操作が実装されていることである。
    - この点に関する理解が足りないと、R は遅いと信じ込むことになる。
- ベクトルに `log()` などの関数を作用させると、ベクトルの各要素に対して操作が施されたものが出力される。

```
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
```

ベクトル化続き

```
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 2.828427
##  [9] 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
```

論理演算子

演算子	読み方	例	出力
`x < y`	x は y より小さい	`3 < 42`	TRUE
`x > y`	x は y より大きい	`3 > 42`	FALSE
`x <= y`	x は y 以下	`3 <= 42`	TRUE
`x >= y`	x は y 以上	`3 >= 42`	FALSE
`x == y`	x は y に等しい	`3 == 42`	FALSE
`x != y`	x は y に等しくない	`3 != 42`	TRUE
`!x`	x でない	`!(3 > 42)`	TRUE
`x|y`	x または y	`(3 > 42)|TRUE`	TRUE
`x & y`	x かつ y	`(3 < 4) & (42 > 13)`	TRUE

論理演算子の例

- 論理演算子はベクトル化されている。

```
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
```

論理演算子の例

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

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

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

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

論理演算子を用いた要素取得

- これはベクトルの要素の取得（切り出し）に大変便利である。

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

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

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

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

ちょっと練習

1. 大きさが x と等しいベクトル $z = (1, 2, 1, 2, 1, 2)$ を作成せよ。
2. z のある要素と等しい x の要素を取り出せ。

行列

行列の操作

- R は**行列**の計算もできる。
  - 行列では、同じデータ型をもつ要素が行（row）と列（column）に格納されている。
- `matrix()` 関数で作成できる。

```
x = 1:9
X = matrix(x, nrow = 3, ncol = 3)
X
```

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

- ここではふたつの変数を用いている。
  - 小文字の `x` はベクトル
  - 大文字の `X` は行列

行列の操作

- デフォルトでは `matrix()` は列に沿ってベクトルの要素を並べかえるので、行に沿って並べかえたければ `byrow = TRUE` とする。

```
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
```

- 全ての要素が等しい行列は次のように作成できる。

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

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

行列の操作

- 行列の要素も `[]` で取得できる。
    - 行列は二次元なので、添字もふたつ指定しなければならない。
- `X` の1 行 2 列目の要素 X_{12} を取得するなら次のようにする。

```
X
```

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

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

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

行列の操作

- 行・列のみを指定して切り出すこともできる。

```
X[1, ]
```

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

```
X[, 2]
```

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

行列の操作

- ベクトルを行で結合（rbind()）したり、列で結合（cbind()）したりして、行列を作成することもできる。

```
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
```

行列の操作

- これらの関数を用いるときは、行名・列名をつけることができる。

```
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
```

行列計算

- 行列計算の実例を示す。

```
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
```

足し算・引き算

```
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
```

要素ごとの掛け算・割り算

```
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
```

- `X * Y` は**行列の積ではない。**行列の要素ごとの積 $X \odot Y$ である。
- `X / Y` も同様に、要素ごとの商である。

行列の積

- 行列の積 XY には `%*%` を使う。

```
X %*% Y
```

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

- 転置行列 $X^\top$ を得るには `t()` を使う。

```
t(X)
```

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

逆行列

- `solve()` は、行列 Z が正則ならば逆行列 Z^{-1} を返す。

```
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
```

逆行列の確認

- 逆行列がちゃんと計算されているか確かめるために、 $Z^{-1}Z$ を計算してみよう。もし正しければ、単位行列が出力されるはずである。

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

```
##               [,1]          [,2]         [,3]
## [1,] 1.000000e+00 -6.245005e-17 0.000000e+00
## [2,] 8.326673e-17  1.000000e+00 5.551115e-17
## [3,] 2.775558e-17  0.000000e+00 1.000000e+00
```

```
diag(3)
```

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

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

ちょっと練習

- 次の連立方程式（同時方程式）を行列を用いて計算せよ。

$$\begin{aligned} 2x_1 + 3x_2 &= 10 \\ 5x_1 + x_2 &= 20 \end{aligned}$$

- ヒント：上の方程式を、2×2 の行列 A を用いて $Ax = y$ の形に書き直してみよう。

行列の特徴

- 行列の次元や情報を取得するための関数が用意されている。

```
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
```

行列の特徴

```
colSums(X)
```

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

```
rowMeans(X)
```

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

```
colMeans(X)
```

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

対角要素・単位行列

- `diag()` 関数に行列を入力すると、対角要素が出力される。

```
diag(Z)
```

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

- 数列を与えると対角行列が出力される。

```
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
```

単位行列

- 数値を与えると、その値に応じた次元の単位行列が出力される。

```
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
```

リスト

リスト

- リストは異質な型の要素を許す一次元のデータ構造である。
  - 添字はひとつ
  - 要素の型は問わない

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

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

例

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

リストの要素取得

- リストの要素はふたつの方法で取得できる。
  1. `$` 演算子
  2. `[]`

```
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
```

リストの要素取得

```
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
```

データフレーム

- 別途説明する。

制御構造

if 文

- `if` 文は次のような構造をしている。

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

例：x と y の大小比較

```
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
```

ifelse 関数

- R には ifelse() 関数が用意されている。
  - 与えられた条件に基づいて値を出力する。

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

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

- ifelse() はベクトルに適用できるのがよい。

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

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

for 文

- `for` 文は、一定の回数だけ同じ手続きを繰り返す。

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

x
```

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

for 文

- for 文はプログラミング言語で一般に利用される制御構造である。
- R ではベクトル化演算が実装されているので、できる限り使わない方がよい。
  - R の for 文は遅いということで悪名高い。

```
x <- 11:15
x <- x * 2
x
```

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

関数

関数

- 関数を使うには、
    - 関数名を入力し、
    - かっこの中に引数を入力し、
    - かっこを閉じる。
- **引数（argument）**とは関数が用いる変数のことである。

```
# これはデモ
# !NOT RUN
function_name(arg1 = 10, arg2 = 20)
```

- 自前の関数を作成することもできる。

例

- 例：変数の標準化

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

- 関数を作成するときは、
  1. 関数名を与える。短く、かつわかりやすい名前をつけること。
  2. `function()` を使って引数を指定する。
  3. `{}` のなかに関数が行う操作を記述する。

関数の定義

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

- 関数名は `standardize`
- 引数は `x`
- `return()` で出力

関数を使ってみる

- 関数をテストしてみよう。
- $\mathcal{N}(2, 25)$ から 10 標本を取り出す。

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

```
##  [1]  0.3420878 -2.8632418 -4.2518933 -2.1620763  5.5012461  6.2950079
##  [7] -1.0157150 -0.6918263  3.0929875  3.4847199
```

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

```
##  [1] -0.1189401 -1.0034068 -1.3865861 -0.8099298  1.3046588  1.5236865
##  [7] -0.4936071 -0.4042345  0.6401329  0.7482261
```

デフォルトの引数

- デフォルトの引数を指定することもできる。

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

例

- 関数の挙動をみるために、さまざまなやり方を試してみる。全て $10^2 = 100$ を計算する。

```
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
```

注意：引数の順序

- 引数名を指定しない場合、引数の順序が問題になってくる。以下では、10^2 ではなく 2^{10} が計算される。

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

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

- デフォルトの値が定まっていない引数に入力を与えない場合、エラーになる。

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

標本分散

- 標本分散を二通りで計算する関数を作成し、デフォルトの引数の挙動について詳しくみてみよう。
- デフォルトでは、不偏分散 σ^2 を計算する。これを s^2 とする。

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

- 最尤法に基づく、不偏ではない標本分散の推定値 $\hat{\sigma}^2$ も計算する。

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

関数の定義

```
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)
}
```

結果 1

```
get_var(test_sample)
```

```
## [1] 13.13356
```

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

```
## [1] 13.13356
```

```
var(test_sample)
```

```
## [1] 13.13356
```

結果2

- 作成した関数が期待通りの挙動をし、R に実装されている `var()` 関数と値が一致した。次に、不偏でない σ^2 の推定値を計算する。

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

```
## [1] 11.8202
```

Data frame

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.

中身を見る

```
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
```

データフレームをCSVファイルで保存する

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

```
write.csv(example_data, "example-data.csv", row.names = FALSE)
```

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.

```
library(readr)
example_data_from_csv = read.csv("example-data.csv")
```

Note:

- 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/taho1/OneDrive/ドキュメント/GitHub/Applied_Econometrics_JPN/02_R_Programming"
```

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

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

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)
```

Examine dataframe

- 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 = 5)
```

```
## # A tibble: 5 × 11
##   manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
##   <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
## 1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compa…
## 2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa…
## 3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compa…
## 4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compa…
## 5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p     compa…
```

`str()` function

- 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)
```

```
## tibble [234 × 11] (S3: tbl_df/tbl/data.frame)
##  $ manufacturer: chr [1:234] "audi" "audi" "audi" "audi" ...
##  $ model       : chr [1:234] "a4" "a4" "a4" "a4" ...
##  $ displ       : num [1:234] 1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int [1:234] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int [1:234] 4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr [1:234] "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr [1:234] "f" "f" "f" "f" ...
##  $ cty         : int [1:234] 18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int [1:234] 29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr [1:234] "p" "p" "p" "p" ...
##  $ class       : chr [1:234] "compact" "compact" "compact" "compact" ...
```

`names()` function

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

```
names(mpg)
```

```
##  [1] "manufacturer" "model"        "displ"        "year"         "cyl"
##  [6] "trans"        "drv"          "cty"          "hwy"          "fl"
## [11] "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 2008
##  [16] 1999 2008 2008 2008 2008 2008 1999 2008 1999 1999 2008 2008 2008 2008 2008
##  [31] 1999 1999 1999 2008 1999 2008 2008 1999 1999 1999 1999 2008 2008 2008 1999
##  [46] 1999 2008 2008 2008 2008 1999 1999 2008 2008 2008 1999 1999 1999 2008 2008
##  [61] 2008 1999 2008 1999 2008 2008 2008 2008 2008 2008 1999 1999 2008 1999 1999
##  [76] 1999 2008 1999 1999 1999 2008 2008 1999 1999 1999 1999 1999 2008 1999 2008
##  [91] 1999 1999 2008 2008 1999 1999 2008 2008 2008 1999 1999 1999 1999 1999 2008
## [106] 2008 2008 2008 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 2008 2008
```

Dimension of dataframe

- 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
```

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 × 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
```

Subseting data (cont.d)

- 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"))
```

dplyr package

- 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)
```