Data frames and Apply

Statistical Computing, STA3005

Monday Jan 22, 2023

Last chapter: Indexing and iteration

- Three ways to index vectors, matrices, data frames, lists: integers, Booleans, names
- Boolean on-the-fly indexing can be very useful
- Named indexing will be especially useful for data frames and lists
- Indexing lists can be a bit tricky (beware of the difference between `[ ]` and `[[ ]]`)
- `if()`, `else if()`, `else`: standard conditionals
- `ifelse()`: shortcut for using `if()` and `else` in combination
- `switch()`: shortcut for using `if()`, `elseif()`, and `else` in combination
- `for()`, `while()`: standard loop constructs
- Don't overuse explicit `for()` loops, vectorization is your friend!
- apply and map functions: can also be very useful (we'll see them today and next week, respectively)

Part I: Data frames

Data frames

The format for the "classic" data table in statistics: **data frame**. Lots of the "really-statistical" parts of the R programming language presume data frames

- Think of each row as an observation/case
- Think of each column as a variable/feature
- Not just a matrix because variables can have different types
- Both rows and columns can be assigned names

(此没有 row name)

Difference between data frames and lists? Each column in a data frame must have the same length (each element in the list can be of different lengths)

Creating a data frame

Use `data.frame()`, similar to how we create lists

```
my.df = data.frame(nums=seq(0.1,0.6,by=0.1), chars=letters[1:6]
                   bools=sample(c(TRUE,FALSE), 6, replace=TRUE)
my.df
```

```
##   nums chars bools
## 1  0.1     a FALSE
## 2  0.2     b  TRUE
## 3  0.3     c  TRUE
## 4  0.4     d  TRUE
## 5  0.5     e  TRUE
## 6  0.6     f  TRUE
```

```
# Recall, a list can have different lengths for different el
my.list = list(nums=seq(0.1,0.6,by=0.1), chars=letters[1:12]
               bools=sample(c(TRUE,FALSE), 6, replace=TRUE))
my.list
```

```
## $nums
## [1] 0.1 0.2 0.3 0.4 0.5 0.6
##
## $chars
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
##
## $bools
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE
```

Indexing a data frame

- By rows/columns: similar to how we index matrices
- By columns only: similar to how we index lists

```
my.df[,1] # Also works for a matrix
```

```
## [1] 0.1 0.2 0.3 0.4 0.5 0.6
```

```
my.df[,"nums"] # Also works for a matrix
```

```
## [1] 0.1 0.2 0.3 0.4 0.5 0.6
```

```
my.df$nums # Doesn't work for a matrix, but works for a list
```

```
## [1] 0.1 0.2 0.3 0.4 0.5 0.6
```

```
my.df$chars # Ditto
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

Creating a data frame from a matrix

Often times it's helpful to start with a matrix, and add columns (of different data types) to make it a data frame

```
class(state.x77) # Built-in matrix of states data, 50 states x 8 variables
```

```
## [1] "matrix" "array"
```

```
head(state.x77)
```

```
##            Population Income Illiteracy Life Exp Murder H
## Alabama          3615   3624        2.1    69.05   15.1
## Alaska            365   6315        1.5    69.31   11.3
## Arizona          2212   4530        1.8    70.55    7.8
## Arkansas         2110   3378        1.9    70.66   10.1
```

```
## California      21198   5114        1.1    71.71   10.3
## Colorado         2541   4884        0.7    72.06    6.8
```

```
class(state.region) # Factor of regions for the 50 states
```

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

(一种 data type，包含两个信息：{ level：unique data 数 / index)

```
head(state.region)
```

```
## [1] South West  West  South West  West
## Levels: Northeast South North Central West
```

```
class(state.division) # Factor of divisions for the 50 state
```

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

```
head(state.division)
```

```
## [1] East South Central Pacific           Mountain
## [5] Pacific            Mountain
## 9 Levels: New England Middle Atlantic South Atlantic ...
```

```
# Combine these into a data frame with 50 rows and 10 column
state.df = data.frame(state.x77, Region=state.region, Divisi
class(state.df)
```

```
## [1] "data.frame"
```

```
head(state.df) # Note that the first 8 columns name carried
```

```
##            Population Income Illiteracy Life.Exp Murder H
## Alabama          3615   3624        2.1    69.05   15.1
## Alaska            365   6315        1.5    69.31   11.3
```

```
## Arizona          2212   4530        1.8    70.55    7.8
## Arkansas         2110   3378        1.9    70.66   10.1
## California      21198   5114        1.1    71.71   10.3
## Colorado         2541   4884        0.7    72.06    6.8
##              Region           Division
## Alabama     South East South Central
## Alaska        West            Pacific
## Arizona       West           Mountain
## Arkansas    South West South Central
## California    West            Pacific
## Colorado      West           Mountain
```

Adding columns to a data frame

To add columns: we can either use `data.frame()`, or directly define a new named column

① ② ③ cbind（但需要再手动修改 column name）

```
# First way: use data.frame() to concatenate on a new column
state.df = data.frame(state.df, Cool=sample(c(T,F), nrow(sta
head(state.df, 4)
```

```
##          Population Income Illiteracy Life.Exp Murder HS.
## Alabama        3615   3624        2.1    69.05   15.1
## Alaska          365   6315        1.5    69.31   11.3
## Arizona        2212   4530        1.8    70.55    7.8
## Arkansas       2110   3378        1.9    70.66   10.1
##          Region           Division  Cool
## Alabama   South East South Central  TRUE
## Alaska     West            Pacific FALSE
## Arizona    West           Mountain FALSE
## Arkansas  South West South Central  TRUE
```

```
# Second way: just directly define a new named column
state.df$Score = sample(1:100, nrow(state.df), replace=TRUE)
head(state.df, 4)
```

```
##          Population Income Illiteracy Life.Exp Murder HS.
## Alabama        3615   3624        2.1    69.05   15.1
## Alaska          365   6315        1.5    69.31   11.3
## Arizona        2212   4530        1.8    70.55    7.8
```

```
## Arkansas       2110   3378        1.9    70.66   10.1
##          Region           Division  Cool Score
## Alabama   South East South Central  TRUE    62
## Alaska     West            Pacific FALSE    92
## Arizona    West           Mountain FALSE    46
## Arkansas  South West South Central  TRUE    72
```

Deleting columns from a data frame

To delete columns: we can either use negative integer indexing, or set a column to `NULL`

```
# First way: use negative integer indexing
state.df = state.df[,-ncol(state.df)]
head(state.df, 4)
```

```
##          Population Income Illiteracy Life.Exp Murder HS.
## Alabama        3615   3624        2.1    69.05   15.1
## Alaska          365   6315        1.5    69.31   11.3
## Arizona        2212   4530        1.8    70.55    7.8
## Arkansas       2110   3378        1.9    70.66   10.1
##          Region           Division  Cool
## Alabama   South East South Central  TRUE
## Alaska     West            Pacific FALSE
## Arizona    West           Mountain FALSE
## Arkansas  South West South Central  TRUE
```

```
# Second way: just directly set a column to NULL
state.df$Cool = NULL
head(state.df, 4)
```

```
##          Population Income Illiteracy Life.Exp Murder HS.
## Alabama        3615   3624        2.1    69.05   15.1
## Alaska          365   6315        1.5    69.31   11.3
## Arizona        2212   4530        1.8    70.55    7.8
## Arkansas       2110   3378        1.9    70.66   10.1
##          Region           Division
## Alabama   South East South Central
## Alaska     West            Pacific
```

```
## Arizona    West           Mountain
## Arkansas   South West South Central
```

Reminder: Boolean indexing

With matrices or data frames, we'll often want to access a subset of the rows corresponding to some condition. You already know how to do this, with Boolean indexing

```
# Compare the averages of the Frost column between states in
# Pacific divisions
mean(state.df[state.df$Division == "New England", "Frost"])
```

```
## [1] 145.3333
```

```
mean(state.df[state.df$Division == "Pacific", "Frost"])
```

```
## [1] 49.6
```

`subset()` : extract rows based on a condition

The `subset()` function provides a convenient alternative way of accessing rows for data frames

```
# Using subset(), we can just use the column names directly
# using $)
state.df.ne.1 = subset(state.df, Division == "New England")
# Get same thing by extracting the appropriate rows manually
state.df.ne.2 = state.df[state.df$Division == "New England",]
all(state.df.ne.1 == state.df.ne.2)
```

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

```
# Same calculation as in the last slide, using subset()
mean(subset(state.df, Division == "New England")$Frost)
```

```
## [1] 145.3333
```

```
mean(subset(state.df, Division == "Pacific")$Frost)
```

```
## [1] 49.6
```

Part II: Factors

Creating a factor from a vector

Factors are the data objects which are used to categorize the data and store it as a combination of a integer vector and a character vector, known as `levels`. Use `factor()` function to create a factor

index

level

```
gender.vec <- c("male", "female", "female", "male")
gender.vec
```

```
## [1] "male"   "female" "female" "male"
```

```
gender.fac <- factor(gender.vec)
gender.fac
```

```
## [1] male   female female male
## Levels: female male
```

```
levels(gender.fac) # character vector
```

```
## [1] "female" "male"
```

(按字母表顺序)

```
as.numeric(gender.fac) # integer vector
```

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

- `levels` contains the unique set of values, which are taken by `as.character(gender.vec)` and sorted in alphabetical order.
- The integer vector indicates the indices of each element in `levels`.

Changing the order of levels

Sometimes, the order of levels is meaningful. We expect to set `levels` in specific rather than alphabetical order.

```
quality <- factor(c("low", "high", "medium", "high", "low",
quality
```

```
## [1] low    high   medium high   low    medium high
## Levels: high low medium
```

```
as.numeric(quality)
```

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

```
quality.adj <- factor(quality, levels = c("low", "medium", "
quality.adj
```

```
## [1] low    high   medium high   low    medium high
## Levels: low medium high
```

```
as.numeric(quality.adj)
```

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

重新定义 levels

```
levels(quality) = c("low", "medium", "high")
quality
```

```
## [1] medium low    high   low    medium high   low
## Levels: low medium high
```

注意：该方法并不是修改顺序，而是把所有的 high 换成 low，low 换成 medium，medium 换成 high

Converting factors to a vector

Converting from a factor to a vector may cause problems:

① 错误的方法

```
num.fac <- factor(c(3.4, 1.2, 5))
as.numeric(num.fac)   (返回 index)
```

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

② 第一种方法

```
num.1 <- levels(num.fac)[num.fac]
num.1
```

= c("1.2", "3.4", "5") = c(2, 1, 3)

```
## [1] "3.4" "1.2" "5"   (strings. 因为 levels 的 type 均为 string)
```

```
is.vector(num.1)
```

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

```
as.numeric(num.1)
```

```
## [1] 3.4 1.2 5.0
```

③ 第二种方法

```
num.2 <- as.character(num.fac)
as.numeric(num.2)
```

```
## [1] 3.4 1.2 5.0
```

- Convert a character factor to a vector by `as.character`
- Cannot directly convert a numeric factor to a vector by `as.numeric`: first, convert to a character vector, and then covert to a numeric vector
- Factors are needed for many functions, like `tapply()` and `split()` we will introduce in this chapter

Part III: `apply()` function

The apply family

R offers a family of **apply functions**, which allow you to apply a function across different chunks of data. Offers an alternative to explicit iteration using `for()` loop; can be simpler and faster, though not always. Summary of functions:

- `apply()` : apply a function to rows or columns of a matrix or data frame
- `lapply()` : apply a function to elements of a list or vector
- `sapply()` : same as the above, but simplify the output (if possible)
- `tapply()` : apply a function to levels of a factor vector

`apply()` : rows or columns of a matrix or data frame

The `apply()` function takes inputs of the following form:

- `apply(x, MARGIN=1, FUN=my.fun)` , to apply `my.fun()` across rows of a matrix or data frame `x`
- `apply(x, MARGIN=2, FUN=my.fun)` , to apply `my.fun()` across columns of a matrix or data frame `x`

```
apply(state.x77, MARGIN=2, FUN=min) # Minimum entry in each
```

```
## Population     Income Illiteracy   Life Exp     Murder
##     365.00    3098.00       0.50      67.96       1.40
##       Area
##    1049.00
```

```
apply(state.x77, MARGIN=2, FUN=max) # Maximum entry in each
```

```
## Population     Income Illiteracy   Life Exp     Murder
##    21198.0     6315.0        2.8       73.6       15.1
##       Area
##   566432.0
```

```r
apply(state.x77, MARGIN=2, FUN=which.max) # Index of the max
```

```
## Population     Income Illiteracy   Life Exp     Murder
##          5          2         18         11          1
##       Area
##          2
```

```r
apply(state.x77, MARGIN=2, FUN=summary) # Summary of each co
```

```
##         Population  Income Illiteracy Life Exp Murder HS
## Min.        365.00 3098.00      0.500  67.9600  1.400  37
## 1st Qu.    1079.50 3992.75      0.625  70.1175  4.350  48
## Median     2838.50 4519.00      0.950  70.6750  6.850  53
## Mean       4246.42 4435.80      1.170  70.8786  7.378  53
## 3rd Qu.    4968.50 4813.50      1.575  71.8925 10.675  59
## Max.      21198.00 6315.00      2.800  73.6000 15.100  67
```

Applying a custom function

For a custom function, we can just define it before hand, and the use `apply()` as usual

```r
# Our custom function: trimmed mean
trimmed.mean = function(v) {
  q1 = quantile(v, prob=0.1)
  q2 = quantile(v, prob=0.9)
  return(mean(v[q1 <= v & v <= q2]))
}

apply(state.x77, MARGIN=2, FUN=trimmed.mean)
```

```
##  Population      Income  Illiteracy    Life Exp      Murd
##  3384.27500  4430.07500     1.07381    70.91775     7.297
##       Frost        Area
##   104.68293 56575.72500
```

We'll learn more about functions later (don't worry too much at this point about the details of the function definition)

Applying a custom function "on-the-fly"

Instead of defining a custom function before hand, we can just define it "on-the-fly". Sometimes this is more convenient

```
# Compute trimmed means, defining this on-the-fly
apply(state.x77, MARGIN=2, FUN=function(v) {
  q1 = quantile(v, prob=0.1)
  q2 = quantile(v, prob=0.9)
  return(mean(v[q1 <= v & v <= q2]))
})
```

在使用 apply() 时定义 function

```
##  Population      Income  Illiteracy    Life Exp      Murd
##  3384.27500  4430.07500     1.07381    70.91775     7.297
##       Frost        Area
##   104.68293 56575.72500
```

Applying a function that takes extra arguments

Can tell `apply()` to pass **extra arguments** to the function in question. E.g., can use:
`apply(x, MARGIN=1, FUN=my.fun, extra.arg.1, extra.arg.2)`, for two extra arguments `extra.arg.1`, `extra.arg.2` to be passed to `my.fun()`

```
# Our custom function: trimmed mean, with user-specified per
trimmed.mean = function(v, p1, p2) {
  q1 = quantile(v, prob=p1)
  q2 = quantile(v, prob=p2)
  return(mean(v[q1 <= v & v <= q2]))
}
apply(state.x77, MARGIN=2, FUN=trimmed.mean, p1=0.01, p2=0.99)
```

(无论有没有 default value)

function 的从左数第一个未被 specify 的 argument 会被 assign 成 columns

specify arguments

```
##   Population       Income   Illiteracy     Life Exp
##  3974.125000  4424.520833     1.136735    70.882708     7
```

```
##        Frost         Area
##   104.895833 61860.687500
```

What's the return argument?

What kind of data type will `apply()` give us? Depends on what function we pass. Summary, say, with `FUN=my.fun()`:

- If `my.fun()` returns a single value, then `apply()` will return a vector
- If `my.fun()` returns k values, then `apply()` will return a matrix with k rows (note: this is true *regardless* of whether `MARGIN=1` or `MARGIN=2`)
- If `my.fun()` returns different length outputs for different inputs, then `apply()` will return a list
- If `my.fun()` returns a list, then `apply()` will return a list

Optimized functions for special tasks

Don't overuse the apply paradigm! There's lots of special functions that **optimized** are will be both simpler and faster than using `apply()`. E.g.,

- `rowSums()`, `colSums()`: for computing row, column sums of a matrix
- `rowMeans()`, `colMeans()`: for computing row, column means of a matrix
- `max.col()`: for finding the maximum position in each row of a matrix

Combining these functions with logical indexing and vectorized operations will enable you to do quite a lot. E.g., how to count the number of positives in each row of a matrix?

```
x = matrix(rnorm(1e8), 10000, 10000)
# Don't do this (much slower for big matrices)
system.time(apply(x, MARGIN=1, function(v) { return(sum(v >
```

```
##    user  system elapsed
##    2.14    0.94    3.13
```

```r
# Do this instead (much faster, simpler)
system.time(rowSums(x > 0))
```

```
##    user  system elapsed
##    0.66    0.08    0.74
```

- `system.time()` returns a vector of three times, where `elapsed` represents the actual running time. The definition of `user` and `system` times is complicated (You can find more details from the help file of `proc.time()`)

Part IV: `lapply()`, `sapply()`, `tapply()` functions

`lapply()`: elements of a list or vector

The `lapply()` function takes inputs as in: `lapply(x, FUN=my.fun)`, to apply `my.fun()` across elements of a list or vector `x`. The output is always a list

```r
my.list
```

```
## $nums
## [1] 0.1 0.2 0.3 0.4 0.5 0.6
##
## $chars
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
##

## $bools
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE
```

```r
lapply(my.list, FUN=mean) # Get a warning: mean() can't be a
```

```
## Warning in mean.default(X[[i]], ...): argument is not num
## returning NA
```

```
## $nums
## [1] 0.35
##
## $chars
## [1] NA
##
## $bools
## [1] 0.3333333
```

```
lapply(my.list, FUN=summary)
```

```
## $nums
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.100   0.225   0.350   0.350   0.475   0.600
##
## $chars
##    Length     Class      Mode
##        12 character character
##
## $bools
##    Mode   FALSE    TRUE
## logical       4       2
```

`sapply()` : elements of a list or vector

The `sapply()` function works just like `lapply()`, but tries to **simplify** the return value whenever possible. E.g., most common is the conversion from a list to a vector

```
sapply(my.list, FUN=mean) # Simplifies the result, now a vec
```

```
## Warning in mean.default(X[[i]], ...): argument is not num
## returning NA
```

```
##      nums     chars     bools
## 0.3500000        NA 0.3333333
```

```
sapply(my.list, FUN=summary) # Can't simplify, so still a li
```

```
## $nums
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.100   0.225   0.350   0.350   0.475   0.600
##
## $chars
##    Length     Class      Mode
##        12 character character
##
## $bools
##    Mode   FALSE    TRUE
## logical       4       2
```

`tapply()` : levels of a factor vector

The function `tapply()` takes inputs as in: `tapply(x, INDEX=my.index, FUN=my.fun)` , to apply `my.fun()` to subsets of entries in `x` that share a common level in `my.index`

```
# Compute the mean and sd of the Frost variable, within each
tapply(state.x77[,"Frost"], INDEX=state.region, FUN=mean)
```

```
##     Northeast         South North Central          West
##      132.7778       64.6250      138.8333      102.1538
```

```
tapply(state.x77[,"Frost"], INDEX=state.region, FUN=sd)
```

```
##     Northeast         South North Central          West
##      30.89408      31.30682      23.89307      68.87652
```

`split()` : split by levels of a factor

The function `split()` split up the rows of a data frame by levels of a factor, as in: `split(x, f=my.index)` to split a data frame `x` according to levels of `my.index`

```
# Split up the state.x77 matrix according to region
state.by.reg = split(data.frame(state.x77), f=state.region)
```

```r
class(state.by.reg) # The result is a list
```

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

```r
names(state.by.reg) # This has 4 elements for the 4 regions
```

```
## [1] "Northeast"     "South"         "North Central" "West
```

```r
class(state.by.reg[[1]]) # Each element is a data frame
```

```
## [1] "data.frame"
```

```r
# For each region, display the first 3 rows of the data fram
lapply(state.by.reg, FUN=head, 3)
```

```
## $Northeast
##               Population Income Illiteracy Life.Exp Murde
## Connecticut         3100   5348        1.1    72.48    3.
## Maine               1058   3694        0.7    70.39    2.
## Massachusetts       5814   4755        1.1    71.83    3.
##
## $South
##            Population Income Illiteracy Life.Exp Murder HS.
## Alabama          3615   3624        2.1    69.05   15.1
## Arkansas         2110   3378        1.9    70.66   10.1
## Delaware          579   4809        0.9    70.06    6.2
##
## $`North Central`
##            Population Income Illiteracy Life.Exp Murder HS.
## Illinois        11197   5107        0.9    70.14   10.3
## Indiana          5313   4458        0.7    70.88    7.1
## Iowa             2861   4628        0.5    72.56    2.3
##
## $West
##              Population Income Illiteracy Life.Exp Murder H
## Alaska              365   6315        1.5    69.31   11.3
```

```
## Arizona           2212   4530        1.8    70.55    7.8
## California       21198   5114        1.1    71.71   10.3
```

```r
# For each region, average each of the 8 numeric variables
lapply(state.by.reg, FUN=function(df) {
  return(apply(df, MARGIN=2, mean))
})
```

```
## $Northeast
##   Population       Income   Illiteracy     Life.Exp
##  5495.111111  4570.222222     1.000000    71.264444     4
##        Frost         Area
##   132.777778 18141.000000
##
## $South
##  Population      Income  Illiteracy    Life.Exp      Murd
##  4208.12500  4011.93750     1.73750    69.70625    10.581
##       Frost        Area
##    64.62500 54605.12500
##
## $`North Central`
##  Population      Income  Illiteracy    Life.Exp      Murd
##  4803.00000  4611.08333     0.70000    71.76667     5.275
##       Frost        Area
##   138.83333 62652.00000
##
## $West
##   Population       Income   Illiteracy     Life.Exp
## 2.915308e+03 4.702615e+03 1.023077e+00 7.123462e+01 7.215
##        Frost         Area
## 1.021538e+02 1.344630e+05
```

Summary

- Data frames are a representation of the “classic” data table in R: rows are observations/cases, columns are variables/features
- Each column can be a different data type (but must be the same length)
- Factors represent a vector by categories (`levels`) and integer indices

- `subset()` : function for extracting rows of a data frame meeting a condition
- `split()` : function for splitting up rows of a data frame, according to a factor variable
- `apply()` : function for applying a given routine to rows or columns of a matrix or data frame
- `lapply()` : similar, but used for applying a routine to elements of a vector or list
- `sapply()` : similar, but will try to simplify the return type, in comparison to `lapply()`
- `tapply()` : function for applying a given routine to groups of elements in a vector or list, according to a factor variable