data.table

sunsik

1. Creating data.table

We can build data.table just like we build data.frame:

```
set.seed(2007)
library(dplyr)
library(data.table)
custom_dt <- data.table(A = rep(letters[2:1], each = 4),
                        B = rep(1:4, each = 4),
                        C = sample(8))
custom_df <- data.frame(A = rep(letters[2:1], each = 4),
                        B = rep(1:4, each = 4),
                        C = sample(8))
```

2. Basic facts on data.table

Primary unit is row

Below code implies that **primary unit of data.table is row**, unlike data.frame(list of columns).

```
custom_dt[1:2]
```

```
##    A B C
## 1: b 1 6
## 2: b 1 2
```

```
custom_df[1:2] %>% head()
```

```
##   A B
## 1 b 1
## 2 b 1
## 3 b 1
## 4 b 1
## 5 a 2
## 6 a 2
```

If selecting column are needed, one can make use of [[]]. Note that one can only select single column using double brackets.

```
custom_dt[[1]] %>% head()
```

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

```
custom_dt[[1:2]] %>% head()
```

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

data.table is also data.frame

Since data.table is also a data.frame, data.frame related process is also applicable to the data.table.

```
class(custom_dt)
```

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

```r
custom_dt %>%
  group_by(A) %>%
  summarize(Bmean = mean(B))
```

```
## # A tibble: 2 x 2
##   A     Bmean
##   <chr> <dbl>
## 1 a         3
## 2 b         2
```

3. Subsetting data.table

Basically, spaces for subsetting operation on data.table can be categorized as "i, j, by" part which stands for i, j, by of **DT[i, j, by]** respectively.

(1) Row filtering("i" part)

Filtering conditions on data table can be specified as **numeric(row index) or logical vector**.

```r
# setting filtering condition as 1, 3, 5 to select 1, 3, 5th row
custom_dt[seq(1, 5, by = 2)]
```

```
##    A B C
## 1: b 1 6
## 2: b 1 1
## 3: a 2 7
```

```r
# boolean filter to select rows whose values of A column and B column are "b", 1
custom_dt[A == 'b' & B == 1]
```

```
##    A B C
## 1: b 1 6
## 2: b 1 2
## 3: b 1 1
## 4: b 1 4
```

(2) Column selection("j" part)

Column selection can be done in the form of **list or expression**. Rule of thumb in column selection is that **"any column that is not specified will not be called"**.

1) In a list form

You'll **get data.table as a result**

```r
custom_dt[A == 'b' & B == 1, list(B, C)]
```

```
##    B C
## 1: 1 6
## 2: 1 2
## 3: 1 1
## 4: 1 4
```

Note that a list can be abbreviated as .().

```r
custom_dt[A == 'b' & B == 1, .(B, C)]
```

```
##    B C
## 1: 1 6
## 2: 1 2
## 3: 1 1
## 4: 1 4
```

Additionally, **manipulating columns while selecting it is also possible**.

```
custom_dt[1:5, .(B = B + 1)]
```

```
##    B
## 1: 2
## 2: 2
## 3: 2
## 4: 2
## 5: 3
```

(as we can see, since B is the only column that is specified in the list, B becomes the only column that is called, which supports the rule of thumb above)

2) In a expression form

Selection using expression can be used when trying to **extract single column with certain manipulation as a vector**. What expression means is the single verse enveloped by parentheses.

```
custom_dt[1:5, (B + 1)]
```

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

(3) Grouped subsetting("by" part)

Specifying grouping variable in "by" space enables identical operation that group_by & summarize operation in dplyr package does.

```
# identical to group_by(A) %>% summarize(maxB = max(B)).
custom_dt[, .(maxB = max(B)), by = A]
```

```
##    A maxB
## 1: b    3
## 2: a    4
```

Note that **multiple grouping** is also possible if **grouping variables are specified as a list**.

```
custom_dt[, .(maxC = max(C)), by = .(A, B)]
```

```
##    A B maxC
## 1: b 1    6
## 2: a 2    8
## 3: b 3    6
## 4: a 4    8
```

Manipulating grouping variable while specifying it is also possible.

```
# Below code shows that subsetting operation is executed in by -> row -> column order.
# That is, R understands code below in following order:
# (1) by remainder3, (2) filter the rows whose value in C is less than or equal to 4 and then (3) selec
custom_dt[C <= 4, .(maxC = max(C)), by = .(remainder3 = B %% 3)]
```

```
##    remainder3 maxC
## 1:          1    4
## 2:          2    3
## 3:          0    4
```

(4) Chaining

It is important to note that variables are not generated dynamically. For example, code below causes an error because maxC is not created when creating maxone.

```
custom_dt[C <= 4, .(maxC = max(C), maxone = maxC + 1), by = .(remainder3 = B %% 3)]
```

For such situation, chaining operation is prepared as in dplyr(or magrittr) which can be implemented as:

```
custom_dt[C <= 4, .(maxC = max(C)), by = .(remainder3 = B %% 3)][
  , .(maxone = maxC + 1), by = remainder3
]
```

```
##    remainder3 maxone
## 1:          1      5
## 2:          2      4
## 3:          0      5
```

4. Built-in variables in a data.table object

(1) .N : number of data

Every data.table has default variable .N, which stands for **the Number of data**. Below code counts the number of data grouped by A and B.

```
custom_dt[order(A), .(count = .N), by = .(A, B)]
```

```
##    A B count
## 1: a 2     4
## 2: a 4     4
## 3: b 1     4
## 4: b 3     4
```

(2) .SD and .SDcols

1) .SD : subset of the data which is grouped according to "by" variable

Every data.table has default variable .SD, which stands for **the Subset of Data**. More specificly, it stands for subsets of data resulted from "by" specification. So if by is not specified, .SD stands for whole data and if specified, specific subsets of the data.

```
custom_dt[, lapply(.SD, mean)] # (a)
```

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

```
##     A   B   C
## 1: NA 2.5 4.5
```

```
custom_dt[, lapply(.SD, mean), by = .(A, B)]
```

```
##    A B    C
## 1: b 1 3.25
## 2: a 2 5.75
## 3: b 3 3.25
## 4: a 4 5.75
```

2) .SDcols : specification of column(s) of .SD

Note in (a), since just subset of the data is specified, it applies lapply to every columns and thus produces warning message(presence of character variable A). This can be fixed if we **specify the columns of such subset**, and .SDcols is being readied for such need.

```
custom_dt[, lapply(.SD, mean), .SDcols = c('B', 'C')] # specification by colname
```

```
##       B   C
## 1: 2.5 4.5
```

```
custom_dt[, lapply(.SD, mean), .SDcols = 2:3] # specification by column index
```

```
##       B   C
## 1: 2.5 4.5
```

```
custom_dt[, lapply(.SD, mean), .SDcols = names(custom_dt) != 'A'] # specification by boolean
```

```
##       B   C
## 1: 2.5 4.5
```

Since .SD itself refers to subset of a data, we can conduct indexing/filtering on .SD directly.

```
# 1. Data is grouped according to the unique value of A column(by = 'A'), and .SD refers to such data.
# 2. Among such data, .SD[B <= 2] excludes rows whose B value is greater than 2.
# 3. Then it applies mean function to 'B', 'C', which is specified in .SDcols(i.e. error doesn't occur)
custom_dt[, lapply(.SD[B <= 2], mean), .SDcols = c('B', 'C'), by = 'A']
```

```
##    A B    C
## 1: b 1 3.25
## 2: a 2 5.75
```

Note that every columns are called, because every one of them are designated.

5. Mutating data.table permanently

Reference(:=) is the operation that **directly alters column(s) of data.table**, unlike subsetting operations introduced above.

(1) Attatching/Detatching single column

Below code shows the usage of reference.

```
custom_dt[, D := B + C]
custom_dt %>% head(3)
```

```
##    A B C D
## 1: b 1 6 7
## 2: b 1 2 3
## 3: b 1 1 2
```

In the same perspective, we can delete existing columns by referencing NULL to the column like:

```
custom_dt[, D := NULL]
custom_dt %>% head(3)
```

```
##    A B C
## 1: b 1 6
## 2: b 1 2
## 3: b 1 1
```

(2) Attatching/Detatching multiple columns

We envelope reference sign with two tick marks(‘, one with tilde button).

```
custom_dt[, `:=`(B = B + 1,
                 D = B + C,
                 E = abs(B - C))][
                   , `:=`(G = D * E,
                          H = 1)
                   ]
head(custom_dt)
```

```
##    A B C  D E  G H
## 1: b 2 6  7 5 35 1
## 2: b 2 2  3 1  3 1
## 3: b 2 1  2 0  0 1
## 4: b 2 4  5 3 15 1
## 5: a 3 7  9 5 45 1
## 6: a 3 8 10 6 60 1
```

As mentioned above, since variables are not dynamically generated, code below results in error.

```
custom_dt[, `:=`(D = B + C,
                 B = B + 1,
                 E = abs(B - C),
                 G = D * E,
                 H = 1)]
```

When removing multiple columns, input should be vector, whether it is character(colname) or numeric(index)

```
# Deleting G, H columns
custom_dt[, LETTERS[7:8] := NULL]
head(custom_dt)
```

```
##    A B C  D E
## 1: b 2 6  7 5
## 2: b 2 2  3 1
## 3: b 2 1  2 0
## 4: b 2 4  5 3
## 5: a 3 7  9 5
## 6: a 3 8 10 6
```

```
# Deleting 2nd, 4th columns
custom_dt[, c(2, 4) := NULL]
head(custom_dt)
```

```
##    A C E
## 1: b 6 5
## 2: b 2 1
## 3: b 1 0
## 4: b 4 3
## 5: a 7 5
## 6: a 8 6
```

(3) Efficiency of another level: set() family

```
random_dt <- as.data.table(matrix(rnorm(10^3 * 10^3), nrow = 10^3))
```

1) set() function

However, if number of manipulations to make gets larger, writing all the specific code might not be efficient. For that, data.table package provides set function, to ensure **for-loop style manipulation** on data.table. By using set function inside for loop, it implements reference(:=) repeatively.

set function consists of following arguments:

`set(DT, index, column, value)`

- **DT** : data.table to manipulate
- **index** : optional space for specifying **row index** to manipulate(ex. which() with is.na())
- **column** : iterator
- **value** : value to replace. If index = NULL, **length of value should equal to that of random_data[, iterator]**

```r
for (i in 1:ncol(random_dt)) set(random_dt, NULL, i, round(random_dt[[i]]))
# By this for-set loop, every column(1:ncol(random_dt)) is substituted to round of itself.
head(random_dt[, 1:5])
```

```
##    V1 V2 V3 V4 V5
## 1:  0 -2  0 -2  1
## 2:  2 -1  1  0 -1
## 3:  1  0  0  2 -1
## 4: -1  1 -1  1  1
## 5:  1  1  1  0  1
## 6: -2 -1 -1  1  0
```

2) setnames() function

setnames function also iteratively uses reference implicitly to change column names, thus result in **fast substitution on column names**. It has following arguments:

`setnames(DT, old, new)`

(length(old) == length(new) must hold)

```r
# This substitutes first character of colnames whose index is even number to "S"
setnames(random_dt,
         names(random_dt)[1:ncol(random_dt) %% 2 == 0],
         paste0("S", (2 * 1:(length(random_dt) / 2))))
names(random_dt)[1:10]
```

```
##  [1] "V1"  "S2"  "V3"  "S4"  "V5"  "S6"  "V7"  "S8"  "V9"  "S10"
```

3) setcolorder() function

Provides really intuitive, **fast way of changing the order of the columns**. The argument 'neworder' can be character vector of colnames or numeric vector of colindexes.

`setcolorder(DT, neworder)`

(length(neworder) == length(DT) must hold)

```r
setcolorder(random_dt, ncol(random_dt):1)
names(random_dt)[1:8]
```

```
## [1] "S1000" "V999"  "S998"  "V997"  "S996"  "V995"  "S994"  "V993"
```