Chapter 23: data.table

Data.table is a package that extends the functionality of data frames from base R, particularly improving on their performance and syntax. See the package's Docs area at Getting started with data.table for details.

Section 23.1: Creating a data.table

A data.table is an enhanced version of the data.frame class from base R. As such, its `class()` attribute is the vector `"data.table" "data.frame"` and functions that work on a data.frame will also work with a data.table. There are many ways to create, load or coerce to a data.table.

Build

Don't forget to install and activate the `data.table` package

```
library(data.table)
```

There is a constructor of the same name:

```
DT <- data.table(
  x = letters[1:5],
  y = 1:5,
  z = (1:5) > 3
)
#    x y     z
# 1: a 1 FALSE
# 2: b 2 FALSE
# 3: c 3 FALSE
# 4: d 4  TRUE
# 5: e 5  TRUE
```

Unlike `data.frame`, `data.table` will not coerce strings to factors:

```
sapply(DT, class)
#               x           y           z
#     "character"   "integer"   "logical"
```

Read in

We can read from a text file:

```
dt <- fread("my_file.csv")
```

Unlike `read.csv`, `fread` will read strings as strings, not as factors.

Modify a data.frame

For efficiency, data.table offers a way of altering a data.frame or list to make a data.table in-place (without making a copy or changing its memory location):

```
# example data.frame
DF <- data.frame(x = letters[1:5], y = 1:5, z = (1:5) > 3)
# modification
setDT(DF)
```

Note that we do not `<-` assign the result, since the object `DF` has been modified in-place. The class attributes of the

data.frame will be retained:

```
sapply(DF, class)
#         x         y         z
#  "factor" "integer" "logical"
```

Coerce object to data.table

If you have a `list`, `data.frame`, or `data.table`, you should use the `setDT` function to convert to a `data.table` because it does the conversion by reference instead of making a copy (which `as.data.table` does). This is important if you are working with large datasets.

If you have another R object (such as a matrix), you must use `as.data.table` to coerce it to a `data.table`.

```
mat <- matrix(0, ncol = 10, nrow = 10)

DT <- as.data.table(mat)
# or
DT <- data.table(mat)
```

Section 23.2: Special symbols in data.table

.SD

`.SD` refers to the subset of the `data.table` for each group, excluding all columns used in `by`.

`.SD` along with `lapply` can be used to apply any function to multiple columns by group in a `data.table`

We will continue using the same built-in dataset, `mtcars`:

```
mtcars = data.table(mtcars) # Let's not include rownames to keep things simpler
```

Mean of all columns in the dataset by *number of cylinders*, `cyl`:

```
mtcars[ , lapply(.SD, mean), by = cyl]

#   cyl      mpg     disp        hp     drat       wt     qsec        vs        am     gear
carb
#1:   6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286 0.4285714 3.857143
3.428571
#2:   4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909 0.7272727 4.090909
1.545455
#3:   8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000 0.1428571 3.285714
3.500000
```

Apart from `cyl`, there are other categorical columns in the dataset such as `vs`, `am`, `gear` and `carb`. It doesn't really make sense to take the `mean` of these columns. So let's exclude these columns. This is where `.SDcols` comes into the picture.

.SDcols

`.SDcols` specifies the columns of the `data.table` that are included in `.SD`.

Mean of all columns (continuous columns) in the dataset by *number of gears* `gear`, and *number of cylinders*, `cyl`, arranged by `gear` and `cyl`:

```
# All the continuous variables in the dataset
```

```
cols_chosen <- c("mpg", "disp", "hp", "drat", "wt", "qsec")

mtcars[order(gear, cyl), lapply(.SD, mean), by = .(gear, cyl), .SDcols = cols_chosen]

#   gear cyl    mpg     disp       hp     drat       wt    qsec
#1:    3   4 21.500 120.1000  97.0000 3.700000 2.465000 20.0100
#2:    3   6 19.750 241.5000 107.5000 2.920000 3.337500 19.8300
#3:    3   8 15.050 357.6167 194.1667 3.120833 4.104083 17.1425
#4:    4   4 26.925 102.6250  76.0000 4.110000 2.378125 19.6125
#5:    4   6 19.750 163.8000 116.5000 3.910000 3.093750 17.6700
#6:    5   4 28.200 107.7000 102.0000 4.100000 1.826500 16.8000
#7:    5   6 19.700 145.0000 175.0000 3.620000 2.770000 15.5000
#8:    5   8 15.400 326.0000 299.5000 3.880000 3.370000 14.5500
```

Maybe we don't want to calculate the `mean` by groups. To calculate the mean for all the cars in the dataset, we don't specify the `by` variable.

```
mtcars[ , lapply(.SD, mean), .SDcols = cols_chosen]

#        mpg     disp       hp     drat      wt     qsec
#1: 20.09062 230.7219 146.6875 3.596563 3.21725 17.84875
```

Note:

- It is not necessary to define `cols_chosen` beforehand. `.SDcols` can directly take column names
- `.SDcols` can also directly take a vector of columnnumbers. In the above example this would be `mtcars[ , lapply(.SD, mean), .SDcols = c(1,3:7)]`

.N

`.N` is shorthand for the number of rows in a group.

```
iris[, .(count=.N), by=Species]

#      Species count
#1:     setosa    50
#2: versicolor    50
#3:  virginica    50
```

Section 23.3: Adding and modifying columns

`DT[where, select|update|do, by]` syntax is used to work with columns of a data.table.

- The "where" part is the `i` argument
- The "select|update|do" part is the `j` argument

These two arguments are usually passed by position instead of by name.

Our example data below is

```
mtcars = data.table(mtcars, keep.rownames = TRUE)
```

Editing entire columns

Use the `:=` operator inside `j` to assign new columns:

```
mtcars[, mpg_sq := mpg^2]
```

Remove columns by setting to `NULL`:

```
mtcars[, mpg_sq := NULL]
```

Add multiple columns by using the `:=` operator's multivariate format:

```
mtcars[, `:=`(mpg_sq = mpg^2, wt_sqrt = sqrt(wt))]
# or
mtcars[, c("mpg_sq", "wt_sqrt") := .(mpg^2, sqrt(wt))]
```

If the columns are dependent and must be defined in sequence, one way is:

```
mtcars[, c("mpg_sq", "mpg2_hp") := .(temp1 <- mpg^2, temp1/hp)]
```

The `.()` syntax is used when the right-hand side of `LHS := RHS` is a list of columns.

For dynamically-determined column names, use parentheses:

```
vn = "mpg_sq"
mtcars[, (vn) := mpg^2]
```

Columns can also be modified with `set`, though this is rarely necessary:

```
set(mtcars, j = "hp_over_wt", v = mtcars$hp/mtcars$wt)
```

Editing subsets of columns

Use the `i` argument to subset to rows "where" edits should be made:

```
mtcars[1:3, newvar := "Hello"]
# or
set(mtcars, j = "newvar", i = 1:3, v = "Hello")
```

As in a data.frame, we can subset using row numbers or logical tests. It is also possible to use a "join" in `i`, but that more complicated task is covered in another example.

Editing column attributes

Functions that edit attributes, such as `levels<-` or `names<-`, actually replace an object with a modified copy. Even if only used on one column in a data.table, the entire object is copied and replaced.

To modify an object without copies, use `setnames` to change the column names of a data.table or data.frame and `setattr` to change an attribute for any object.

```
# Print a message to the console whenever the data.table is copied
tracemem(mtcars)
mtcars[, cyl2 := factor(cyl)]

# Neither of these statements copy the data.table
setnames(mtcars, old = "cyl2", new = "cyl_fac")
setattr(mtcars$cyl_fac, "levels", c("four", "six", "eight"))

# Each of these statements copies the data.table
names(mtcars)[names(mtcars) == "cyl_fac"] <- "cf"
levels(mtcars$cf) <- c("IV", "VI", "VIII")
```

Be aware that these changes are made by reference, so they are *global*. Changing them within one environment

affects the object in all environments.

```
# This function also changes the levels in the global environment
edit_levels <- function(x) setattr(x, "levels", c("low", "med", "high"))
edit_levels(mtcars$cyl_factor)
```

Section 23.4: Writing code compatible with both data.frame and data.table

Differences in subsetting syntax

A `data.table` is one of several two-dimensional data structures available in R, besides **`data.frame`**, **`matrix`** and (2D) **`array`**. All of these classes use a very similar but not identical syntax for subsetting, the `A[rows, cols]` schema.

Consider the following data stored in a **`matrix`**, a **`data.frame`** and a `data.table`:

```
ma <- matrix(rnorm(12), nrow=4, dimnames=list(letters[1:4], c('X', 'Y', 'Z')))
df <- as.data.frame(ma)
dt <- as.data.table(ma)

ma[2:3]  #---> returns the 2nd and 3rd items, as if 'ma' were a vector (because it is!)
df[2:3]  #---> returns the 2nd and 3rd columns
dt[2:3]  #---> returns the 2nd and 3rd rows!
```

If you want to be sure of what will be returned, it is better to be *explicit*.

To get specific **rows**, just add a comma after the range:

```
ma[2:3, ]  # \
df[2:3, ]  #  }---> returns the 2nd and 3rd rows
dt[2:3, ]  # /
```

But, if you want to subset **columns**, some cases are interpreted differently. All three can be subset the same way with integer or character indices *not* stored in a variable.

```
ma[, 2:3]          #  \
df[, 2:3]          #   \
dt[, 2:3]          #    }---> returns the 2nd and 3rd columns
ma[, c("Y", "Z")]  #   /
df[, c("Y", "Z")]  #  /
dt[, c("Y", "Z")]  # /
```

However, they differ for unquoted variable names

```
mycols <- 2:3
ma[, mycols]               # \
df[, mycols]               #  }---> returns the 2nd and 3rd columns
dt[, mycols, with = FALSE] # /

dt[, mycols]               # ---> Raises an error
```

In the last case, `mycols` is evaluated as the name of a column. Because `dt` cannot find a column named `mycols`, an error is raised.

Note: For versions of the `data.table` package priorto 1.9.8, this behavior was slightly different. Anything in the column index would have been evaluated using `dt` as an environment. So both **`dt`**`[, 2:3]` and **`dt`**`[, mycols]` would

return the vector `2:3`. No error would be raised for the second case, because the variable `mycols` does exist in the parent environment.

Strategies for maintaining compatibility with data.frame and data.table

There are many reasons to write code that is guaranteed to work with **`data.frame`** and `data.table`. Maybe you are forced to use **`data.frame`**, or you may need to share some code that you don't know how will be used. So, there are some main strategies for achieving this, in order of convenience:

1. Use syntax that behaves the same for both classes.
2. Use a common function that does the same thing as the shortest syntax.
3. Force `data.table` to behave as **`data.frame`** (ex.: call the specific method **`print.data.frame`**).
4. Treat them as **`list`**, which they ultimately are.
5. Convert the table to a **`data.frame`** before doing anything (bad idea if it is a huge table).
6. Convert the table to `data.table`, if dependencies are not a concern.

Subset rows. Its simple, just use the `[, ]` selector, *with* the comma:

```
A[1:10, ]
A[A$var > 17, ]  # A[var > 17, ] just works for data.table
```

Subset columns. If you want a single column, use the `$` or the `[[ ]]` selector:

```
A$var
colname <- 'var'
A[[colname]]
A[[1]]
```

If you want a uniform way to grab more than one column, it's necessary to appeal a bit:

```
B <- `[.data.frame`(A, 2:4)

# We can give it a better name
select <- `[.data.frame`
B <- select(A, 2:4)
C <- select(A, c('foo', 'bar'))
```

Subset 'indexed' rows. While **`data.frame`** has **`row.names`**, `data.table` has its unique `key` feature. The best thing is to avoid **`row.names`** entirely and take advantage of the existing optimizations in the case of `data.table` when possible.

```
B <- A[A$var != 0, ]
# or...
B <- with(A, A[var != 0, ])  # data.table will silently index A by var before subsetting

stuff <- c('a', 'c', 'f')
C <- A[match(stuff, A$name), ]  # really worse than: setkey(A); A[stuff, ]
```

Get a 1-column table, get a row as a vector. These are easy with what we have seen until now:

```
B <- select(A, 2)    #---> a table with just the second column
C <- unlist(A[1, ])  #---> the first row as a vector (coerced if necessary)
```

Section 23.5: Setting keys in data.table

Yes, you need to SETKEY pre 1.9.6

In the past (pre 1.9.6), your `data.table` was sped up by setting columns as keys to the table, particularly for large tables. [See intro vignette page 5 of September 2015 version, where speed of search was 544 times better.] You may find older code making use of this setting keys with 'setkey' or setting a 'key=' column when setting up the table.

```
library(data.table)
DT <- data.table(
  x = letters[1:5],
  y = 5:1,
  z = (1:5) > 3
)

#> DT
#   x y     z
#1: a 5 FALSE
#2: b 4 FALSE
#3: c 3 FALSE
#4: d 2  TRUE
#5: e 1  TRUE
```

Set your key with the `setkey` command. You can have a key with multiple columns.

```
setkey(DT, y)
```

Check your table's key in tables()

```
tables()

> tables()
     NAME NROW NCOL MB COLS  KEY
[1,] DT      5    3  1 x,y,z y
Total: 1MB
```

Note this will re-sort your data.

```
#> DT
#   x y     z
#1: e 1  TRUE
#2: d 2  TRUE
#3: c 3 FALSE
#4: b 4 FALSE
#5: a 5 FALSE
```

Now it is unnecessary

Prior to v1.9.6 you had to have set a key for certain operations especially joining tables. The developers of data.table have sped up and introduced a `"on="` feature that can replace the dependency on keys. See SO answer here for a detailed discussion.

In Jan 2017, the developers have written a vignette around secondary indices which explains the "on" syntax and allows for other columns to be identified for fast indexing.

Creating secondary indices?

In a manner similar to key, you can `setindex(DT, key.col)` or `setindexv(DT, "key.col.string")`, where DT is your data.table. Remove all indices with `setindex(DT, NULL)`.

See your secondary indices with `indices(DT)`.

Why secondary indices?

This **does not sort** the table (unlike key), but does allow for quick indexing using the "on" syntax. Note there can be only one key, but you can use multiple secondary indices, which saves having to rekey and resort the table. This will speed up your subsetting when changing the columns you want to subset on.

Recall, in example above y was the key for table DT:

```
DT
# x y     z
# 1: e 1  TRUE
# 2: d 2  TRUE
# 3: c 3 FALSE
# 4: b 4 FALSE
# 5: a 5 FALSE

# Let us set x as index
setindex(DT, x)

# Use indices to see what has been set
indices(DT)
# [1] "x"

# fast subset using index and not keyed column
DT["c", on ="x"]
#x y     z
#1: c 3 FALSE

# old way would have been rekeying DT from y to x, doing subset and
# perhaps keying back to y (now we save two sorts)
# This is a toy example above but would have been more valuable with big data sets
```