Post02-Thanh-La

Data Manipulation

Introduction

What is good code? Good code has aspects of readability and compactness. Which are things that make a data manipulation package popular.

In this post, I will talk about the `data.table` package that allows for very efficient data manipulation. It only depends on the base R code, so dependency packages are not needed to use this package.

The data.table package allows for efficient use of memory, fast access and loading of data. It's simple and compact syntax allows for powerful expressions to be formed for data transformation.

Motivation

In this post, I will go over some benefits of the package through examples. The goal of this post is to go over basic functions in the package that is typically included in data manipulation packages. An important operator I will cover is the `:=` operator that utilizes in place memory overwrites to dynamically update data table objects.

Background

There is no need for mathematical background or knowledge of other functions in R, only simple operations with data.frame objects.

A quick overview on the data: The data on frogs describes a survey of frogs, recording climate, setting, geographic information in the presence or absence of frogs within a region. The other data describes the NBA players and their performance in the league. More information is provided in the link in the References section.

Discussions

Before going through the tutorial, I want to touch on some key points of data preparation and why it is important.

According to `Why data preparation is an important part of data science?`

Data preparation consists of

1. Data cleaning
2. Data Integration
3. Data Transformation
4. Data Reduction
5. Data discretization

A study found that

- 60% of the time in organizing and cleaning data.
- 19% of the time is spent in collecting datasets.
- 9% of the time is spent in mining the data to draw patterns.
- 3% of the time is spent in training the datasets.
- 4% of the time is spent in refining the algorithms.
- 5% of the time is spent in other tasks.

The results were concluded from a survey on 80 data scientists. You can find more information about it on the link in the resources section.

Data preparation allows for insight on the study before performing in depth analysis. It helps guide our intuition and raise concerns that may not have been considered during the data collection process.

According to `IBM: IT Trends`, data preparation allows for

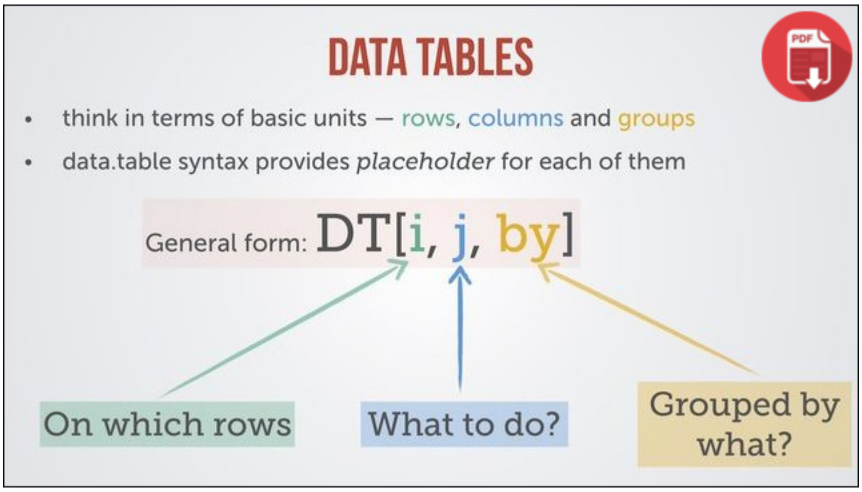
- Credibility of data
- Simplifies the process and provides actionable information
- Makes it easy to explain data to employees and stakeholders

More information about this can be found in the resources below.

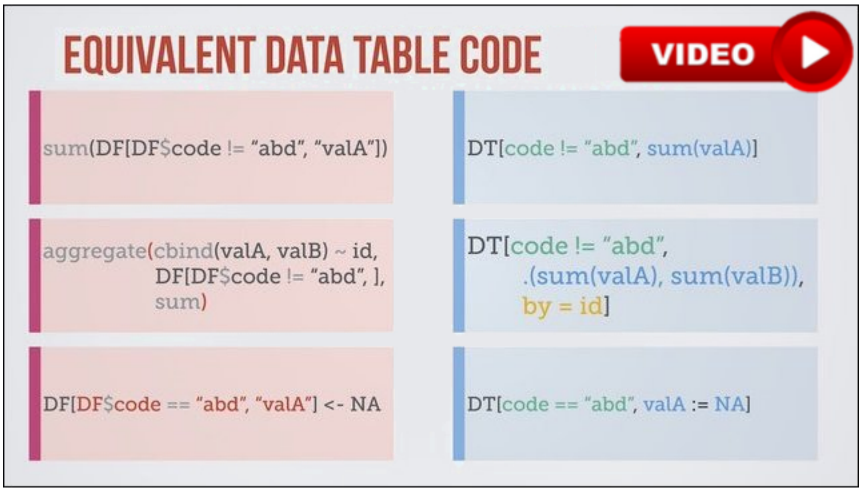
Tutorial data.table: Examples and further Discussions

The idea behind data.table is to access the data table like an sql query

Here is a helpful image to picture formulation of the commands



Here is an equivalence that describe how you can produce the same output with data.table expressions.



One of the goals of this tutorial is to create expressions with data.table functions that can be performed by redundant data.frame function calls.

Data Used:

In this tutorial I will be using the frogs data from the DAAG package To load it: run this code

```
#package containing frogs data
library(DAAG)
```

```
## Loading required package: lattice
```

```
#dependency package
library(lattice)

#loading frogs data as frogs
data(frogs)
help(frogs)
```

The other data sets used are the NBA-2017 statistics and the player information. It can be found on the link below, or in our class github folder under data.

The links below provide descriptions for these data sets.

Time complexity benefits

There are multiple ways to read in the data. To demonstrate the runtime power of the table.data `fread()` function for reading in data, we will run it on the large data set of NBA 2017 statistics

```
#load library
library(data.table)
```

```
## Warning: package 'data.table' was built under R version 3.4.2
```

```
system.time(temp_ <- read.csv("nba2017-stats.csv", header=TRUE,sep=","))
```

```
##    user  system elapsed
##   0.010   0.001   0.018
```

```
system.time(temp_ <- fread("nba2017-stats.csv", header=TRUE,sep=","))
```

```
##    user  system elapsed
##   0.002   0.001   0.004
```

```
print("dimension:")
```

```
## [1] "dimension:"
```

```
dim(temp_)
```

```
## [1] 441  22
```

As you can see, this is almost an instant load for fread compared to the regular read.csv function. However, the times will vary depending on personal machines, but overall fread will perform much better than read.csv.

```
df <- read.csv("nba2017-stats.csv", header=TRUE,sep=",")
DT <- data.table(df)
class(DT)
```

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

We can either read in csv files directly to create data.table objects or instantiate them from regular data frames. Similarly, you can cast data.frame objects with the function `as.data.table()`.

Notice that the data.table is a class of data.table and data.frame. This is to say that the data.table is a subclass of data.frame, therefore we are able to apply regular operations on the data.table object as we would on a data.frame object.

Accessing data

Basics

Let's take a look at the frog data

```
frogs_DT <- data.table(frogs)
#accessing rows with index in i
frogs_DT[c(1:5),]
```

```
##    pres.abs northing easting altitude distance NoOfPools NoOfSites
## 1:        1      115    1047     1500      500       232         3
## 2:        1      110    1042     1520      250        66         5
## 3:        1      112    1040     1540      250        32         5
## 4:        1      109    1033     1590      250         9         5
## 5:        1      109    1032     1590      250        67         5
##      avrain  meanmin  meanmax
## 1: 155.0000 3.566667 14.00000
## 2: 157.6667 3.466667 13.80000
## 3: 159.6667 3.400000 13.60000
## 4: 165.0000 3.200000 13.16667
## 5: 165.0000 3.200000 13.16667
```

```
#accessing as data.frame
frogs_DT[1:5]
```

```
##    pres.abs northing easting altitude distance NoOfPools NoOfSites
## 1:        1      115    1047     1500      500       232         3
## 2:        1      110    1042     1520      250        66         5
## 3:        1      112    1040     1540      250        32         5
## 4:        1      109    1033     1590      250         9         5
## 5:        1      109    1032     1590      250        67         5
##      avrain  meanmin  meanmax
## 1: 155.0000 3.566667 14.00000
## 2: 157.6667 3.466667 13.80000
## 3: 159.6667 3.400000 13.60000
## 4: 165.0000 3.200000 13.16667
## 5: 165.0000 3.200000 13.16667
```

As expected, both methods work for accessing data. As a data.table and as a data.frame

```
#accessing columns by select j
frogs_DT[c(1:5),.(northing, easting) ]
```

```
##    northing easting
## 1:      115    1047
## 2:      110    1042
## 3:      112    1040
## 4:      109    1033
## 5:      109    1032
```

```
#return types

#as list
typeof(frogs_DT[c(1:5),.(northing, easting) ])
```

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

```
#as vector
typeof(frogs_DT[c(1:5),c(northing, easting) ])
```

```
## [1] "double"
```

Here notice we do not need the `$` operator that we usually need to access data frames. Additionally, `.()` serves as an alias for `list()`. This way we can access multiple columns. To keep the returned object as a data.frame object, we need to wrap output in a list to ensure it preserves its type, otherwise a vector would be produced. Remember that a data.frame is a list.

```
#restoring to data.frame mode: "with=" argument
frogs_DT[c(1:5),c("northing", "easting"), with=FALSE ]
```

```
##    northing easting
## 1:      115    1047
## 2:      110    1042
## 3:      112    1040
## 4:      109    1033
## 5:      109    1032
```

Notice here that we pass the column names as a string. By using the `with` argument, we temporarily access the data as a data.frame type for usual accessing.

```
#accessing columns in an iterative fashion
frogs_DT[,northing:distance, with=FALSE ]
```

```
##      northing easting altitude distance
##   1:      115    1047     1500      500
##   2:      110    1042     1520      250
##   3:      112    1040     1540      250
##   4:      109    1033     1590      250
##   5:      109    1032     1590      250
##  ---
## 208:      143    1047     1300     3000
## 209:      299     730     1580     8000
## 210:      287     730     1640     9000
## 211:      269     729     1660     9500
## 212:      186    1108     1440     2750
```

```
names(frogs_DT)
```

```
##  [1] "pres.abs"  "northing"  "easting"   "altitude"  "distance"
##  [6] "NoOfPools" "NoOfSites" "avrain"    "meanmin"   "meanmax"
```

This method will access the columns in order from left to right with respect to the data structure.

Filtering

We can subset data by passing expressions in the i, and j components

```
#filter conditions
frogs_DT[northing > 100 & easting > 100, .(northing, pres.abs)]
```

```
##      northing pres.abs
##   1:      115        1
##   2:      110        1
##   3:      112        1
##   4:      109        1
##   5:      109        1
##  ---
## 202:      143        0
## 203:      299        0
## 204:      287        0
## 205:      269        0
## 206:      186        0
```

Here we can filter the data based on the `northing` and `easting` variables to find a correlation of whether a frog was present or not for a location more than 100 distance from the reference point with respect to `northing` and `easting`.

We have already seen how to select columns by specifying the column names, however, we can also exclude columns with the `!` and `-` negation prefix.

```
frogs_DT[, -c("northing","easting","altitude","distance",
              "NoOfPools","NoOfSites"), with=FALSE]
```

```
##      pres.abs   avrain  meanmin  meanmax
##   1:        1 155.0000 3.566667 14.00000
##   2:        1 157.6667 3.466667 13.80000
##   3:        1 159.6667 3.400000 13.60000
##   4:        1 165.0000 3.200000 13.16667
##   5:        1 165.0000 3.200000 13.16667
##  ---
## 208:        0 132.3333 4.200000 15.80000
## 209:        0 138.0000 2.433333 13.40000
## 210:        0 150.3333 2.266667 12.76667
## 211:        0 157.0000 2.266667 12.53333
## 212:        0 142.3333 3.766667 14.53333
```

```
frogs_DT[, !c("northing","easting","altitude","distance",
              "NoOfPools","NoOfSites"), with=FALSE]
```

```
##      pres.abs   avrain  meanmin  meanmax
##   1:        1 155.0000 3.566667 14.00000
##   2:        1 157.6667 3.466667 13.80000
##   3:        1 159.6667 3.400000 13.60000
##   4:        1 165.0000 3.200000 13.16667
##   5:        1 165.0000 3.200000 13.16667
##  ---
## 208:        0 132.3333 4.200000 15.80000
## 209:        0 138.0000 2.433333 13.40000
## 210:        0 150.3333 2.266667 12.76667
## 211:        0 157.0000 2.266667 12.53333
## 212:        0 142.3333 3.766667 14.53333
```

```
#this will also work for removing unitary chunks of columns at once
frogs_DT[, !(northing:meanmin), with=FALSE]
```

```
##      pres.abs  meanmax
##   1:        1 14.00000
##   2:        1 13.80000
##   3:        1 13.60000
##   4:        1 13.16667
##   5:        1 13.16667
##  ---
## 208:        0 15.80000
## 209:        0 13.40000
## 210:        0 12.76667
## 211:        0 12.53333
## 212:        0 14.53333
```

Functional mapping

How do we actually make use of these subsetting operations to analyze the data? Well, we can actually pass in functions in the jth component to return summary statistical values of the data

```
#applying a function on a column
frogs_DT[, .(mean(distance))]
```

```
##          V1
## 1: 1932.547
```

```
#regular data frames
#frogs[,mean(frogs)]
```

Here the `.()` is optional. Notice that this kind of operation is not allowed in the regular setting of a data.frame. Try running the commented out command on the last line and see what you get.

So how do we apply functions to multiple columns? We can use the `.SD` and `.SDcols` along with `lapply()`. Where `.SD` is a data.frame and the `.SDcols` is a character vector that lists the select columns as its parameters.

```
#parameters to analyze
arg_ <- c("northing", "easting", "NoOfSites")
frogs_DT[,lapply(.SD, mean), .SDcols=arg_]
```

```
##    northing  easting NoOfSites
## 1: 228.1887 1004.585  2.938679
```

By utilizing the mappings of `lapply()` and the parameters `.SD`, `.SDcols` as place holders, we can map the input function onto all the columns of interest.

Grouping

Now we will talk about analyzing data that is more categorical. The nba-2017 player data will reveal the strength of the data.table package.

```
nba_DT <- fread("nba2017-players.csv", header=TRUE,sep=",")
```

As mentioned, more information about the data used in this tutorial can be found by following the links in the resources section.

For the purpose of grouping, we will now learn to us the `by=` parameter

```
nba_DT[,sum(games), by=team]
```

```
##     team  V1
##  1:  BOS 892
##  2:  CLE 730
##  3:  TOR 793
##  4:  WAS 712
##  5:  ATL 762
##  6:  MIL 847
##  7:  IND 794
##  8:  CHI 774
##  9:  MIA 769
## 10:  DET 850
## 11:  CHO 747
## 12:  NYK 839
## 13:  ORL 794
## 14:  PHI 715
## 15:  BRK 778
## 16:  GSW 945
## 17:  SAS 955
## 18:  HOU 652
## 19:  LAC 864
## 20:  UTA 882
## 21:  OKC 782
## 22:  MEM 824
## 23:  POR 794
## 24:  DEN 775
## 25:  NOP 594
## 26:  DAL 751
## 27:  SAC 700
## 28:  MIN 796
## 29:  LAL 775
## 30:  PHO 800
##     team  V1
```

Here we can find out how many games each team has played. The operations in the jth component preserves ordering of the original data.

Ordering

To order the data, we can take advantage of the `keyby=` argument and the `order()` function.

```
nba_DT[,.(sum(games)), keyby=.(team, position)]
```

```
##      team position  V1
##   1:  ATL        C 144
##   2:  ATL       PF 167
##   3:  ATL       PG 169
##   4:  ATL       SF 203
##   5:  ATL       SG  79
##  ---
## 146:  WAS        C 206
## 147:  WAS       PF  78
## 148:  WAS       PG 158
## 149:  WAS       SF 106
## 150:  WAS       SG 164
```

The `keyby=` argumente orders by team and position in this case. However, for a less constrained order specification, we can use `order()`.

```
nba_DT[order(player, -team)]
```

```
##               player team position height weight age experience
##   1:    A.J. Hammons  DAL        C     84    260  24          0
##   2:    Aaron Brooks  IND       PG     72    161  32          8
##   3:    Aaron Gordon  ORL       SF     81    220  21          2
##   4:   Adreian Payne  MIN       PF     82    237  25          2
##   5:      Al Horford  BOS        C     82    245  30          9
##  ---
## 437: Wilson Chandler  DEN       SF     80    225  29          8
## 438:    Yogi Ferrell  DAL       PG     72    180  23          0
## 439:     Zach LaVine  MIN       SG     77    189  21          2
## 440:   Zach Randolph  MEM       PF     81    260  35         15
## 441:   Zaza Pachulia  GSW        C     83    270  32         13
##                                    college   salary games minutes points
##   1:                     Purdue University   650000    22     163     48
##   2:                  University of Oregon  2700000    65     894    322
##   3:                 University of Arizona  4351320    80    2298   1019
##   4:             Michigan State University  2022240    18     135     63
##   5:                 University of Florida 26540100    68    2193    952
##  ---
## 437:                     DePaul University 11200000    71    2197   1117
## 438:                    Indiana University   207798    36    1046    408
## 439: University of California, Los Angeles  2240880    47    1749    889
## 440:             Michigan State University 10361445    73    1786   1028
## 441:                                        2898000    70    1268    426
##      points3 points2 points1
##   1:       5      12       9
##   2:      48      73      32
##   3:      77     316     156
##   4:       3      20      14
##   5:      86     293     108
##  ---
## 437:     110     323     141
## 438:      60      82      64
## 439:     120     206     117
## 440:      21     412     141
## 441:       0     164      98
```

Here we are ordering alphabetically from A-Z based on the player's names, but reverse order Z-A for team. the `-` indicates descending order. The `order()` function as a part of data.table package actually runs much faster than the `base::order()`, try it on a very large data set.

Chaining expressions

We can chain these commands by having the form `DT[...][...][...]...`

```
nba_DT[,.(sum(games)), by=team][order(team)]
```

```
##     team  V1
##  1:  ATL 762
##  2:  BOS 892
##  3:  BRK 778
##  4:  CHI 774
##  5:  CHO 747
##  6:  CLE 730
##  7:  DAL 751
##  8:  DEN 775
##  9:  DET 850
## 10:  GSW 945
## 11:  HOU 652
## 12:  IND 794
## 13:  LAC 864
## 14:  LAL 775
## 15:  MEM 824
## 16:  MIA 769
## 17:  MIL 847
## 18:  MIN 796
## 19:  NOP 594
## 20:  NYK 839
## 21:  OKC 782
## 22:  ORL 794
## 23:  PHI 715
## 24:  PHO 800
## 25:  POR 794
## 26:  SAC 700
## 27:  SAS 955
## 28:  TOR 793
## 29:  UTA 882
## 30:  WAS 712
##     team  V1
```

Here we are able to summarize the data by finding the number of games each team has played and then sorting the information alphabetically by team name.

`:=` Operator

Now we will explore adding and deleting information to data columns with the `:=` operator. This operator does assignments in place as opposed to the method of recreating existing structure, making the changes and then the reassignment the data to the original name. By doing in place overwrites, we save temporal memory.

```
#augment data
vec_ <- rep(c("random 1","random2"), times=dim(frogs_DT)[1]/2)
frogs_DT[,c("random_vals"):=.(vec_)]
head(frogs_DT)
```

```
##    pres.abs northing easting altitude distance NoOfPools NoOfSites
## 1:        1      115    1047     1500      500       232         3
## 2:        1      110    1042     1520      250        66         5
## 3:        1      112    1040     1540      250        32         5
## 4:        1      109    1033     1590      250         9         5
## 5:        1      109    1032     1590      250        67         5
## 6:        1      106    1018     1600      500        12         4
##      avrain  meanmin  meanmax random_vals
## 1: 155.0000 3.566667 14.00000    random 1
## 2: 157.6667 3.466667 13.80000     random2
## 3: 159.6667 3.400000 13.60000    random 1
## 4: 165.0000 3.200000 13.16667     random2
## 5: 165.0000 3.200000 13.16667    random 1
## 6: 167.3333 3.133333 13.06667     random2
```

In the same fashion, we can augment the data.table object by multiple columns at once.

Notice how we do not need to assign anything, the operation makes edits in place and does not need reassignment `<-` as is usually done.

```
#edit columns
frogs_DT[random_vals == "random 1", random_vals := "not random" ]
head(frogs_DT)
```

```
##    pres.abs northing easting altitude distance NoOfPools NoOfSites
## 1:        1      115    1047     1500      500       232         3
## 2:        1      110    1042     1520      250        66         5
## 3:        1      112    1040     1540      250        32         5
## 4:        1      109    1033     1590      250         9         5
## 5:        1      109    1032     1590      250        67         5
## 6:        1      106    1018     1600      500        12         4
##      avrain  meanmin  meanmax random_vals
## 1: 155.0000 3.566667 14.00000  not random
## 2: 157.6667 3.466667 13.80000     random2
## 3: 159.6667 3.400000 13.60000  not random
## 4: 165.0000 3.200000 13.16667     random2
## 5: 165.0000 3.200000 13.16667  not random
## 6: 167.3333 3.133333 13.06667     random2
```

```
#to delete columns
frogs_DT[, c("random_vals") := NULL]
names(frogs_DT)
```

```
##  [1] "pres.abs"  "northing"  "easting"   "altitude"  "distance"
##  [6] "NoOfPools" "NoOfSites" "avrain"    "meanmin"   "meanmax"
```

The `:=` proves very useful in altering existing data in place. To create new copies use the `copy()` to create what is called a deepcopy. More information about this can be found on the `cran-r project data.table general` link below.

Conclusion

data.table package allows for very compact data subseting and transformation. Combining all the aspects and functionalities of the object, we can write clean code for data transformation.

References and Resources

"Why data preparation is an important part of data science?"

"IBM: IT Trends"

"Github data.table"

"R: frog data description"

"NBA statistics"

"Advanced tips and tricks with data.table"

"cran-r project data.table := operator"

"cran-r project data.table general"