…Loading Data with fread

```
# R Libraries
library("reticulate")
library("skimr")

knitr::opts_chunk$set(
  fig.width = 15,
  fig.height = 8,
  out.width = '100%')


# Install Python packages
lapply(c("datatable", "pandas"), function(package) {
       conda_install("r-reticulate", package, pip = TRUE)
})

# Python libraries
from datatable import *
import numpy as np
import re
import pprint
```

We tried to download both the origin zipped data directly from the EPA website (see link below), and the *.csv* from the Tidy Tuesday website, but were unsuccessful in both cases using Python and R versions of fread. We were able to download the Tidy Tuesday *.csv* link with *fread* in `data.table` but not `datatable`, and the error message didn't give us enough information to figure it out. The documentation for `data.table` fread is among the most extensive of any function we know, while still thin for datatable's version so far. In the end, we manually downloaded and unzipped the file from the EPA's website, and uploaded from our local drive.

```
# Data dictionary, EPA vehicles zip and Tidy Tuesday vehicles
csv links
#Data dictionary https://www.fueleconomy.gov/feg/ws/index.shtml#fuelType1
#EPA zip data set https://www.fueleconomy.gov/feg/epadata/vehicles.csv.zip
#Tidy Tuesday csv data set https://raw.githubusercontent.
com/rfordatascience/tidytuesday/master/data/2019/2019-10-15/big_epa_cars.csv

# Load vehicles
big_mt = fread("~/Desktop/David/Projects/general_working/mt_
cars/vehicles.csv")

# Dimensions
big_mt.shape

## (42230, 83)
```

The list of all 83 variables below, and we can see that there are several pertaining to fuel efficiency, emissions, fuel type, range, volume and some of the same attributes that we all know from *mtcars* (ie: cylinders, displacement, make, model and transmission). As mentioned, gross horsepower and weight are missing, but carburetors, acceleration and engine shape are also absent. We have all classes of vehicles sold, so get vehicle class information (*VClass*) not available in *mtcars* which is only cars. We will discuss further down, changes to the weight cutoffs on some of the categories over time make *VClass* of questionable use.

```
# Set up pprint params and print
pp = pprint.PrettyPrinter(width=80, compact = True)
pp.pprint(big_mt.names)
```

```
## ('barrels08', 'barrelsA08', 'charge120', 'charge240',
'city08', 'city08U',
##  'cityA08', 'cityA08U', 'cityCD', 'cityE', 'cityUF',
'co2', 'co2A',
##  'co2TailpipeAGpm', 'co2TailpipeGpm', 'comb08', 'comb08U',
'combA08',
##  'combA08U', 'combE', 'combinedCD', 'combinedUF',
'cylinders', 'displ', 'drive',
##  'engId', 'eng_dscr', 'feScore', 'fuelCost08',
'fuelCostA08', 'fuelType',
##  'fuelType1', 'ghgScore', 'ghgScoreA', 'highway08',
'highway08U', 'highwayA08',
##  'highwayA08U', 'highwayCD', 'highwayE', 'highwayUF',
'hlv', 'hpv', 'id', 'lv2',
##  'lv4', 'make', 'model', 'mpgData', 'phevBlended', 'pv2',
'pv4', 'range',
##  'rangeCity', 'rangeCityA', 'rangeHwy', 'rangeHwyA',
'trany', 'UCity', 'UCityA',
##  'UHighway', 'UHighwayA', 'VClass', 'year',
'youSaveSpend', 'guzzler',
##  'trans_dscr', 'tCharger', 'sCharger', 'atvType',
'fuelType2', 'rangeA',
##  'evMotor', 'mfrCode', 'c240Dscr', 'charge240b',
'c240bDscr', 'createdOn',
##  'modifiedOn', 'startStop', 'phevCity', 'phevHwy',
'phevComb')
```

Set-up Thoughts from R Perspective

There were a couple of things about the set-up for `datatable`, which weren't apparent coming over from `data.table` as an R user. The first was to use `from dt import *` at the outset to avoid having to reference the package short name every time within the frame. From a Python perspective, this is considered bad practice, but we are only going to do it for that one package because it makes us feel more at home. The second was to use `export_names()` in order to skip having to use the f operator or quotation marks to reference variables. In order to do this, we had to create a dictionary of names using the names list from above, and each of their f expressions extracted with `export_names` in a second list. We then used update from the local environment to assign all of the dictionary values to their keys as variables. From then on, we can refer to those variable without quotation marks or the f operator (although any new variables created would still need f or quotation marks). We weren't sure why this is not the default behavior, but it is easily worked around for our purposes. These two possibly not "Pythonic" steps brought the feel of `datatable` a lot closer to the usual R `data.table` (ie: without the package and expression short codes).

Basic Filter and Select Operations

A few lines of some key variables are shown in the code below, and it is clear that they need significant cleaning to be of use. One difference with R `data.table` can be seen below with filtering. Using our `year_filter` in i (the first slot), the 1204 2019 models are shown below. Unlike R `data.table`, we refer to year outside of the frame in an expression, and then call it within i of the frame. The columns can be selected within `()` or `[]` in *j* (the second slot) as shown below, and new columns can be created within `{}`.

```
# Key variables for year 2019
year_filter = (year == 2020)
print(big_mt[year_filter, (year, make, model, trany, evMotor,
VClass)])
```

```
##      | year  make     model                        trany
evMotor             VClass
## ---- + ----  -------  ---------------------------
-------------------------------  ------------------
----------------------------------
##    0 | 2020  Toyota   Corolla
Automatic (AV-S10)                                    Compact
Cars
##    1 | 2020  Toyota   Corolla Hybrid
Automatic (variable gear ratios)  202V Ni-MH          Compact
Cars
##    2 | 2020  Toyota   Corolla                      Manual
6-spd                                           Compact Cars
##    3 | 2020  Toyota   Corolla XSE
Automatic (AV-S10)                                    Compact
Cars
##    4 | 2020  Toyota   Corolla
Automatic (variable gear ratios)                      Compact
Cars
##    5 | 2020  Toyota   Corolla                      Manual
6-spd                                           Compact Cars
##    6 | 2020  Toyota   Corolla XLE
Automatic (variable gear ratios)                      Compact
Cars
##    7 | 2020  Kia      Soul
Automatic (variable gear ratios)                      Small
Station Wagons
##    8 | 2020  Kia      Soul Eco dynamics
Automatic (variable gear ratios)                      Small
Station Wagons
##    9 | 2020  Kia      Soul                         Manual
6-spd                                           Small Station
Wagons
##   10 | 2020  Kia      Soul
Automatic (AM-S7)                                     Small
Station Wagons
##   11 | 2020  Kia      Sportage FWD
Automatic (S6)                                        Small
```

```
Sport Utility Vehicle 2WD
##   12 | 2020  Kia      Sportage FWD
Automatic (S6)                                             Small
Sport Utility Vehicle 2WD
##   13 | 2020  Kia      Telluride FWD
Automatic (S8)                                             Small
Sport Utility Vehicle 2WD
##   14 | 2020  Kia      Sportage AWD
Automatic (S6)                                             Small
Sport Utility Vehicle 4WD
##    … |    …   …        …                                  …
…                   …
## 1199 | 2020  Porsche  718 Cayman GT4                     Manual
6-spd                                               Two Seaters
## 1200 | 2020  Bentley  Mulsanne
Automatic (S8)                                             Midsize
Cars
## 1201 | 2020  Porsche  Cayenne e-Hybrid
Automatic (S8)                   99 kW DC Brushless
Standard Sport Utility Vehicle 4WD
## 1202 | 2020  Porsche  Cayenne e-Hybrid Coupe
Automatic (S8)                   99 kW DC Brushless
Standard Sport Utility Vehicle 4WD
## 1203 | 2020  Porsche  Taycan 4S Perf Battery Plus
Automatic (A2)                   120 kW ACPM         Large
Cars
##
## [1204 rows x 6 columns]
```

We usually like to make a quick check if there are any duplicated rows across the whole our dataFrame, but there isn't a `duplicated()` function yet in datatable. According to [How to find unique values for a field in Pydatatable Data Frame](), the `unique()` function also doesn't apply to groups yet. In order to work around this, identifying variables would have to be grouped, counted and filtered for equal to 1, but we weren't sure yet exactly which variables to group on. We decided to pipe over to pandas to verify with a simple line of code that there were no duplicates, but hope this function will be added in the future.

Aggregate New Variable and Sort

We can see that below that `eng_dscr` is unfortunately blank 38% of the time, and high cardinality for the rest of the levels. A small percentage are marked *"GUZZLER"* and *"FLEX FUELS"*. in a few cases, potentially helpful information about engine like V-6 or V-8 are included with very low frequency, but not consistently enough to make sense try to extract. Another potentially informative variable, `trans_dscr` is similarly blank more than 60% of the time. It seems unlikely that we could clean these up to make it useful in an analysis, so will probably have to drop them.

```
print(big_mt[:, {'percent' : int32(count() *
100/big_mt.nrows) }, by(eng_dscr)]\
            [:,:, sort(-f.percent)])
```

```
##      | eng_dscr                     percent
```

```
## --- + --------------------------  -------
##   0 |                                   38
##   1 | (FFS)                             20
##   2 | SIDI                              14
##   3 | (FFS) CA model                     2
##   4 | (FFS)      (MPFI)                  1
##   5 | (FFS,TRBO)                         1
##   6 | FFV                                1
##   7 | (121)      (FFS)                   0
##   8 | (122)      (FFS)                   0
##   9 | (16 VALVE) (FFS)      (MPFI)       0
##  10 | (16-VALVE) (FFS)                   0
##  11 | (16-VALVE) (FFS)      (MPFI)       0
##  12 | (16-VALVE) (FFS,TRBO)              0
##  13 | (164S)     (FFS)      (MPFI)       0
##  14 | (16VALVES) (FFS)                   0
##   … | …                                  …
## 556 | VTEC       (FFS)                   0
## 557 | VTEC-E                             0
## 558 | VTEC-E     (FFS)                   0
## 559 | Z/28                               0
## 560 | new body style                     0
##
## [561 rows x 2 columns]
```

Separate and Assign New Variables

As shown above, `trany` has both the transmission-type and gear-speed variables within it, so we extracted the variable from big_mt with `to_list()`, drilled down one level, and used regex to extract the transmission and gear information needed out into trans and gear. Notice that we needed to convert the lists back into columns with dt.Frame before assigning as new variables in big_mt.

In the third line of code, we felt like we were using an R `data.table`. The `{}` is used group by *trans* and *gear*, and then to create the new *percent* variable in-line, without affecting the other variables in *big_mt*. We tried to round the decimals in percent, but couldn't figure it out so far. Our understanding is that there is no `round()` method yet for `datatable`, so we multiplied by 100 and converted to integer. We again called `export_names()`, to be consistent in using non-standard evaluation with the two new variables.

```
big_mt['trans'] = Frame([re.sub('[\s\(].*$','', s) for s in
big_mt[:, 'trany'].to_list()[0]])
big_mt['gear'] = Frame([re.sub('A\w+\s|M\w+\s','', s) for s
in big_mt[:, 'trany'].to_list()[0]])
gear, trans= big_mt[:, ('gear', 'trans')].export_names()

# Summarize percent of instances by transmission and speed
print(big_mt[:, { 'percent' : int32(count() * 100
/big_mt.nrows) }, by(trans, gear)]\
            [0:13, : , sort(-f.percent)])
```

```
##    | trans      gear                      percent
```

```
## -- + ---------  ---------------------  -------
##  0 | Automatic  4-spd                       26
##  1 | Manual     5-spd                       19
##  2 | Automatic  (S6)                         7
##  3 | Automatic  3-spd                        7
##  4 | Manual     6-spd                        6
##  5 | Automatic  5-spd                        5
##  6 | Automatic  (S8)                         4
##  7 | Automatic  6-spd                        3
##  8 | Manual     4-spd                        3
##  9 | Automatic  (variable gear ratios)       2
## 10 | Automatic  (AM-S7)                      1
## 11 | Automatic  (S5)                         1
## 12 | Automatic  7-spd                        1
##
## [13 rows x 3 columns]
```

Set Key and Join

We wanted to create a Boolean variable to denote if a vehicle had an electric motor or not. We again used `{}` to create the variable in the frame, but don't think it is possible to update by reference so still had to assign to *is_ev*. In the table below, we show the number of electric vehicles rising from 3 in 1998 to 149 this year. Unfortunately,

```
# Create 'is_ev' within the frame
big_mt['is_ev'] = big_mt[:, { 'is_ev' : evMotor != '' }]
is_ev = big_mt[:, 'is_ev'].export_names()
ann_models = big_mt[:, {'all_models' : count()}, by(year)]
ev_models = big_mt[:, {'ev_models' : count() }, by('year',
'is_ev')]\
                  [(f.is_ev == 1), ('year', 'ev_models')]
ev_models.key = "year"
print(ann_models[:, :, join(ev_models)]\
                [:, { 'all_models' : f.all_models,
                      'ev_models' : f.ev_models,
                      'percent' : int32(f.ev_models * 100 /
f.all_models) },
                      by(year)]\
                [(year > 1996), :])
```

```
##    | year  all_models  ev_models  percent
## -- + ----  ----------  ---------  -------
##  0 | 1997         762         NA       NA
##  1 | 1998         812          3        0
##  2 | 1999         852          7        0
##  3 | 2000         840          4        0
##  4 | 2001         911          5        0
##  5 | 2002         975          2        0
##  6 | 2003        1044          1        0
##  7 | 2004        1122         NA       NA
##  8 | 2005        1166         NA       NA
##  9 | 2006        1104         NA       NA
```

```
## 10 | 2007        1126         NA       NA
## 11 | 2008        1187         23        1
## 12 | 2009        1184         27        2
## 13 | 2010        1109         34        3
## 14 | 2011        1130         49        4
## 15 | 2012        1152         55        4
## 16 | 2013        1184         68        5
## 17 | 2014        1225         77        6
## 18 | 2015        1283         76        5
## 19 | 2016        1262         95        7
## 20 | 2017        1293         92        7
## 21 | 2018        1344        103        7
## 22 | 2019        1335        133        9
## 23 | 2020        1204        149       12
## 24 | 2021          73          6        8
##
## [25 rows x 4 columns]
```

Using Regular Expressions in Row Operations

Next, we hoped to extract wheel-drive (2WD, AWD, 4WD, etc) and engine type (ie: V4, V6, etc) from model. The `re_match()` function is helpful in filtering rows in i. As shown below, we found almost 17k matches for wheel drive, but only 718 for the engine size. Given that we have over 42k rows, we will extract the wheels and give up on the engine data. It still may not be enough data for wheels to be a helpful variable.

```
# Regex match with re_match()
print('%d of rows with wheels info.' %
(big_mt[model.re_match('.*(.WD).*'), model].nrows))
```

```
## 16921 of rows with wheels info.
```

```
print('%d of rows with engine info.' %
(big_mt[model.re_match('.*(V|v)(\s|\-)?\d+.*'),
model].nrows))
```

```
## 718 of rows with engine info.
```

We used regex to extract whether the model was *2WD*, *4WD*, etc as *wheels* from *model*, but most of the time, it was the same information as we already had in drive. It is possible that our weakness in Python is at play, but this would have been a lot simpler in R, because we wouldn't have iterated over every row in order to extract part of the row with regex. We found that there were some cases where the 2WD and 4WD were recorded as 2wd and 4wd. The `replace()` function was an efficient solution to this problem, replacing matches of *'wd'* with *'WD'* over the entire frame.

```
# Extract 'wheels' and 'engine' from 'model'
reg = re.compile(r'(.*)(.WD|4x4)(.*)', re.IGNORECASE)
big_mt[:, 'wheels'] = Frame([reg.match(s).group(2) if
reg.search(s) else '' for s in big_mt[:, model].to_list()
[0]])
wheels = big_mt[:, 'wheels'].export_names()
```

```
# Fix problem notations
big_mt.replace("\dwd", "\dWD")

# Summarize total count for all years
cols = ['make', 'model', 'cylinders', 'wheels', 'drive']
print(big_mt[(f.wheels != ''), cols]\
            [:, count(), by(f.wheels, cylinders, drive)]\
            [0:14:, :, sort(-f.count)])
```

```
##    | wheels  cylinders  drive                       count
## -- + ------  ---------  --------------------------  -----
##  0 | 2WD             8  Rear-Wheel Drive             2616
##  1 | 2WD             6  Rear-Wheel Drive             2255
##  2 | 4WD             6  4-Wheel or All-Wheel Drive   1637
##  3 | 4WD             8  4-Wheel or All-Wheel Drive   1481
##  4 | 2WD             4  Rear-Wheel Drive             1063
##  5 | 4WD             4  4-Wheel or All-Wheel Drive    984
##  6 | AWD             6  All-Wheel Drive              771
##  7 | FWD             4  Front-Wheel Drive            638
##  8 | AWD             4  All-Wheel Drive              629
##  9 | 2WD             4  Front-Wheel Drive            508
## 10 | FWD             6  Front-Wheel Drive            497
## 11 | 2WD             6  Front-Wheel Drive            416
## 12 | AWD             4  4-Wheel or All-Wheel Drive    368
## 13 | 4WD             8  4-Wheel Drive                361
##
## [14 rows x 4 columns]
```

Reshaping

There was no such thing as an 4-wheel drive SUVs back in the 80's, and we remember the big 8-cylinder Oldsmobiles and Cadillacs, so were curious how these models evolved over time. datatable doesn't yet have `dcast()` or `melt()`, so we had to pipe these out `to_pandas()` and then use `pivot_table()`. Its likely that a lot of the the many models where wheel-drive was unspecified were *2WD*, which is still the majority of models. We would have liked to show these as whole numbers, and there is a workaround in datatable to convert to integer, but once we pivoted in `pandas`, it reverted to float. We can see the first *AWD* models starting in the late 80s, and the number of 8-cylinder cars fall by half. There are are a lot fewer annual new car models now than in the 80s, but were surprised how many fewer 4-cylinders.

```
# Summarize by year again having to move to pandas to pivot
print(big_mt[:, count(), by(f.wheels,
year)].to_pandas().pivot_table(index='wheels',
columns='year', values='count'))
```

```
## year      1984    1985   1986   1987   1988  ...   2017
2018   2019   2020  2021
## wheels                                         ...
##         1184.0  1057.0  698.0  732.0  677.0  ...  798.0
821.0  797.0  706.0  46.0
## 2WD      472.0   430.0  338.0  310.0  262.0  ...   89.0
```

```
97.0  110.0   94.0   4.0
## 4WD      304.0   208.0  174.0  201.0  187.0  ...  107.0
119.0  131.0  131.0   5.0
## 4x4        NaN     NaN    NaN    2.0    2.0  ...    1.0
1.0    NaN    NaN   NaN
## AWD        NaN     NaN    NaN    2.0    2.0  ...  186.0
197.0  195.0  180.0  10.0
## FWD        1.0     4.0    NaN    NaN    NaN  ...  104.0
96.0   88.0   78.0   5.0
## RWD        3.0     2.0    NaN    NaN    NaN  ...    8.0
13.0   14.0   15.0   3.0
##
## [7 rows x 38 columns]
```

```
print(big_mt[:, count(), by(cylinders,
year)].to_pandas().pivot_table(index='cylinders',
columns='year', values='count'))
```

```
## year         1984   1985   1986   1987   1988  ...   2017
2018   2019   2020  2021
## cylinders                                     ...
## 2.0           6.0    5.0    1.0    3.0    3.0  ...    1.0
2.0    2.0    2.0   NaN
## 3.0           NaN    6.0    9.0   11.0   13.0  ...   26.0
22.0   22.0   19.0   7.0
## 4.0        1020.0  853.0  592.0  625.0  526.0  ...  563.0
590.0  585.0  523.0  44.0
## 5.0          39.0   20.0   18.0   26.0   17.0  ...    1.0
2.0    2.0    2.0   NaN
## 6.0         457.0  462.0  323.0  296.0  325.0  ...  416.0
449.0  440.0  374.0  17.0
## 8.0         439.0  351.0  263.0  282.0  241.0  ...  211.0
219.0  224.0  222.0   4.0
## 10.0          NaN    NaN    NaN    NaN    NaN  ...    7.0
8.0    4.0    6.0   NaN
## 12.0          3.0    2.0    3.0    4.0    5.0  ...   38.0
27.0   20.0   21.0   1.0
## 16.0          NaN    NaN    NaN    NaN    NaN  ...    NaN
1.0    1.0    1.0   NaN
##
## [9 rows x 38 columns]
```

Combining Levels of Variables with High Cardinality

With 35 distinct levels often referring to similar vehicles, *VClass* also needed to be cleaned up. Even in R `data.table`, we have been keenly awaiting the implementation of `fcase`, a `data.table` version of the `dplyr case_when()` function for nested control-flow statements. We made a separate 16-line function to lump factor levels (not shown). In the first line below, we created the vclasses list to drill down on the *VClass* tuple elements as strings. In the second line, we had to iterate over the resulting strings from the 0-index of the tuple to extract wheel-drive from a list-comprehension. We printed out the result of our much smaller list of lumped factors, but there are still problems with the result. The EPA changed the cutoff for a “Small

Pickup Truck" from 4,500 to 6,000 lbs in 2008, and also used a higher cut-off for "small" SUV's starting in 2011. This will make it pretty hard to us *VClass* as a consistent variable for modeling, at least for Pickups and SUVs. As noted earlier, if we had the a weight field, we could have easily worked around this.

```
# Clean up vehicle type from VClass
vclasses = [tup[0] for tup in big_mt[:,
'VClass'].to_tuples()]
big_mt['VClass'] = Frame([re.sub('\s\dWD$|\/\dwd$|\s\-\s
\dWD$', '', x) if re.search(r'WD$|wd$', x) is not None else x
for x in vclasses])
big_mt['VClass'] = Frame([collapse_vclass(line[0]) for line
in big_mt[:, 'VClass'].to_tuples()])

# Show final VClass types and counts
print(big_mt[:, count(), VClass][:,:, sort(-f.count)])
```

```
##    | VClass                   count
## -- + -----------------------  -----
##  0 | Small Car                16419
##  1 | Midsize Car               5580
##  2 | Standard Pickup Trucks    4793
##  3 | Sport Utility Vehicle     4786
##  4 | Large Car                 2938
##  5 | Small Pickup and SUV      2937
##  6 | Special Purpose Vehicle   2457
##  7 | Vans                      1900
##  8 | Minivan                    420
##
## [9 rows x 2 columns]
```

Selecting Multiple Columns with Regex

In the chunk (below), we show how to select columns from the big_mt names tuple by creating the measures selector using regex matches for the key identifier columns and for integer mileage columns matching '08'. This seemed complicated and we couldn't do it in line within the frame as we would have with `data.table` .SD = patterns(). We also wanted to reorder to move the identifier columns (year, make and model) to the left side of the table, but couldn't find a equivalent `setcolorder` function. There is documentation about multi-column selection, but we couldn't figure out an efficient way to make it work. We show the frame with the `year_filter` which we set up earlier.

```
# Regex search for variable selection
measures = [name for name in big_mt.names if
re.search(r'make|model|year|08$', name)]

# Print remaining cols with measures filter
print(big_mt[year_filter,  measures])
```

```
##      | barrels08  barrelsA08  city08  cityA08  comb08
combA08  fuelCost08  fuelCostA08  highway08  highwayA08  make
model                        year
## ---- + ---------  ----------  ------  -------  ------
```

```
-------  ----------  -----------  ---------  ----------
-------  ---------------------------  ----
##    0 |   9.69441       0           31         0      34
0         800            0         40           0  Toyota
Corolla                             2020
##    1 |   6.33865       0           53         0      52
0         500            0         52           0  Toyota
Corolla Hybrid                      2020
##    2 |  10.3003        0           29         0      32
0         850            0         36           0  Toyota
Corolla                             2020
##    3 |   9.69441       0           31         0      34
0         800            0         38           0  Toyota
Corolla XSE                         2020
##    4 |   9.98818       0           30         0      33
0         800            0         38           0  Toyota
Corolla                             2020
##    5 |   9.98818       0           29         0      33
0         800            0         39           0  Toyota
Corolla                             2020
##    6 |  10.3003        0           29         0      32
0         850            0         37           0  Toyota
Corolla XLE                         2020
##    7 |  10.987         0           27         0      30
0         900            0         33           0  Kia
Soul                                2020
##    8 |  10.6326        0           29         0      31
0         900            0         35           0  Kia
Soul Eco dynamics                   2020
##    9 |  12.2078        0           25         0      27
0        1000            0         31           0  Kia
Soul                                2020
##   10 |  11.3659        0           27         0      29
0         950            0         32           0  Kia
Soul                                2020
##   11 |  12.6773        0           23         0      26
0        1050            0         30           0  Kia
Sportage FWD                        2020
##   12 |  14.3309        0           20         0      23
0        1200            0         28           0  Kia
Sportage FWD                        2020
##   13 |  14.3309        0           20         0      23
0        1200            0         26           0  Kia
Telluride FWD                       2020
##   14 |  14.3309        0           22         0      23
0        1200            0         26           0  Kia
Sportage AWD                        2020
##    … |          …            …         …         …       …
…            …            …          …            …  …         …
…
## 1199 |  17.3479        0           16         0      19
0        2000            0         23           0  Porsche
```

```
718 Cayman GT4              2020
## 1200 |  27.4675        0          10         0       12
0        3150           0         16           0  Bentley
Mulsanne                    2020
## 1201 |  10.5064        0.426      20        45       21
41        1800        1400         22           37  Porsche
Cayenne e-Hybrid            2020
## 1202 |  10.5064        0.426      20        45       21
41        1800        1400         22           37  Porsche
Cayenne e-Hybrid Coupe      2020
## 1203 |   0.294          0          68         0       69
0         950           0         71           0  Porsche
Taycan 4S Perf Battery Plus  2020
##
## [1204 rows x 13 columns]
```

Selecting Columns and Exploring Summary Data

We looked for a Python version of `skimr`, but it doesn't seem like there is an similar library (as is often the case). We tried out `pandas profiling`, but that had a lot of dependencies and seemed like overkill for our purposes, so decided to use `skim_tee` on the table in a separate R chunk (below). It was necessary to convert to `pandas` in the Python chunk (above), because we couldn't figure out how to translate a `datatable` back to a `data.frame` via `reticulate` in the R chunk.

When we did convert, we discovered there were some problems mapping NA's which we will show below. We suspect it isn't possible to pass a `datatable` to `data.table`, and this might be the first functionality we would vote to add. There is a sizable community of `data.table` users who are used to the syntax, and as we are, might be looking to port into Python (rather than learn `pandas` directly). As `reticulate` develops, opening this door seems to make so much sense. Below, we again run `export_names()` in order to also prepare the newly generated variables for non-standard evaluation within the frame, and then filtered for the 21 columns we wanted to keep.

```
# List of cols to keep
cols = ['make',
        'model',
        'year',
        'city08',
        'highway08',
        'comb08',
        'VClass',
        'drive',
        'fuelType1',
        'hlv',
        'hpv',
        'cylinders',
        'displ',
        'trans',
        'gear',
        'wheels',
        'is_ev',
```

```
        'evMotor',
        'guzzler',
        'tCharger',
        'sCharger']

# Select cols and create pandas version
big_mt_pandas = big_mt[:, cols].to_pandas()
```

```
# Skimr
skim_tee(py$big_mt_pandas)
```

```
## ── Data Summary ────────────────────────
##                            Values
## Name                       data
## Number of rows             42230
## Number of columns          21
## _______________________
## Column type frequency:
##   character                12
##   logical                  1
##   numeric                  8
## ________________________
## Group variables            None
##
## ── Variable type: character ──────────────────────────────
──────────────────────
##    skim_variable n_missing complete_rate   min   max empty
n_unique whitespace
##  1 make                  0             1     3    34     0
137          0
##  2 model                 0             1     1    47     0
4217          0
##  3 VClass                0             1     4    23     0
9          0
##  4 drive                 0             1     0    26  1189
8          0
##  5 fuelType1             0             1     6    17     0
6          0
##  6 trans                 0             1     0     9    11
3          0
##  7 gear                  0             1     0    22    11
34          0
##  8 wheels                0             1     0     3 25265
7          0
##  9 evMotor               0             1     0    51 41221
171          0
## 10 guzzler               0             1     0     1 39747
4          0
## 11 tCharger              0             1     0     1 34788
2          0
## 12 sCharger              0             1     0     1 41352
2          0
```

```
##
## ── Variable type: logical ──────────────────────────────
────────────────────────
##   skim_variable n_missing complete_rate   mean count
## 1 is_ev                 0             1 0.0239 FAL: 41221,
TRU: 1009
##
## ── Variable type: numeric ──────────────────────────────
────────────────────────
##   skim_variable n_missing complete_rate    mean    sd
p0    p25   p50    p75
## 1 year                  0           1     2002.   11.4
1984 1991    2003 2012
## 2 city08                0           1       18.5   8.36
6   15      17   21
## 3 highway08             0           1       24.6   8.03
9   20      24   28
## 4 comb08                0           1       20.8   8.06
7   17      20   23
## 5 hlv                   0           1        1.99  5.92
0    0       0    0
## 6 hpv                   0           1       10.2  27.9
0    0       0    0
## 7 cylinders           240           0.994    5.71  1.76
2    4       6    6
## 8 displ               238           0.994    3.29  1.36
0    2.2     3    4.3
##      p100 hist
## 1 2021
## 2  150
## 3  132
## 4  141
## 5   49
## 6  195
## 7   16
## 8    8.4
```

In the result above, we see a lot of challenges if we had hoped to have appropriate data to build a model to predict mpg over time. Many variables, such as evMotor, tCharger, sCharger and guzzler, are only available in a small number of rows. When we set out on this series, we hoped we would be able to experiment with modeling gas mileage for every year just like mtcars, but that seems unlikely based on the available variables.

Conclusion

It took us a couple of months to get up and running with R `data.table`, and even with daily usage, we are still learning its nuance a year later. We think the up-front investment in learning the syntax, which can be a little confusing at first, has been worth it. It is also less well documented than dplyr or pandas. We learned so much about `data.table` from a few blog posts such as Advanced tips and tricks with data.table and A data.table and dplyr tour. The goal of this post is to help to similarly fill the gap for `datatable`.

Python `datatable` is promising, and we are grateful for it as familiar territory as we learn Python. We can't tell how much of our difficulty has been because the package is not as mature as `data.table` or our just inexperience with Python. The need to manually set variables for non-standard evaluation, to revert to pandas to accomplish certain tasks (ie: reshaping) or the challenges extracting and filtering data from nested columns. It was still not easy to navigate the documentation and there were areas where the documentation was not Also, it would be appreciated to seamlessly translate between a `datatable` and `data.table`.