Summarising, Aggregating, and Grouping data in Python Pandas


Pandas – Python Data Analysis Library

I’ve recently started using Python’s excellent Pandas library as a data analysis tool, and, while finding the transition from R’s excellent data.table library frustrating at times, I’m finding my way around and finding most things work quite well.

One aspect that I’ve recently been exploring is the task of grouping large data frames by different variables, and applying summary functions on each group. This is accomplished in Pandas using the “groupby()” and “agg()” functions of Panda’s DataFrame objects.

Update: Pandas version 0.20.1 in May 2017 changed the aggregation and grouping APIs. This post has been updated to reflect the new changes.

A Sample DataFrame

In order to demonstrate the effectiveness and simplicity of the grouping commands, we will need some data. For an example dataset, I have extracted my own mobile phone usage records. I analyse this type of data using Pandas during my work on KillBiller. If you’d like to follow along – the full csv file is available here.

The dataset contains 830 entries from my mobile phone log spanning a total time of 5 months. The CSV file can be loaded into a pandas DataFrame using the pandas.DataFrame.from_csv() function, and looks like this:

<table>
<thead>
<tr>
<th>index</th>
<th>date</th>
<th>duration</th>
<th>item</th>
<th>month</th>
<th>network</th>
<th>network_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15/10/14 06:58</td>
<td>34.429</td>
<td>data</td>
<td>2014-11</td>
<td>data</td>
<td>data</td>
</tr>
<tr>
<td>1</td>
<td>15/10/14 06:58</td>
<td>13.000</td>
<td>call</td>
<td>2014-11</td>
<td>Vodafone</td>
<td>mobile</td>
</tr>
<tr>
<td>2</td>
<td>15/10/14 14:46</td>
<td>23.000</td>
<td>call</td>
<td>2014-11</td>
<td>Meteor</td>
<td>mobile</td>
</tr>
<tr>
<td>3</td>
<td>15/10/14 14:48</td>
<td>4.000</td>
<td>call</td>
<td>2014-11</td>
<td>Tesco</td>
<td>mobile</td>
</tr>
<tr>
<td>4</td>
<td>15/10/14 17:27</td>
<td>4.000</td>
<td>call</td>
<td>2014-11</td>
<td>Tesco</td>
<td>mobile</td>
</tr>
<tr>
<td>5</td>
<td>15/10/14 18:55</td>
<td>4.000</td>
<td>call</td>
<td>2014-11</td>
<td>Tesco</td>
<td>mobile</td>
</tr>
<tr>
<td>6</td>
<td>16/10/14 06:58</td>
<td>34.429</td>
<td>data</td>
<td>2014-11</td>
<td>data</td>
<td>data</td>
</tr>
<tr>
<td>7</td>
<td>16/10/14 15:01</td>
<td>602.000</td>
<td>call</td>
<td>2014-11</td>
<td>Three</td>
<td>mobile</td>
</tr>
<tr>
<td>8</td>
<td>16/10/14 15:12</td>
<td>1050.000</td>
<td>call</td>
<td>2014-11</td>
<td>Three</td>
<td>mobile</td>
</tr>
<tr>
<td>9</td>
<td>16/10/14 15:30</td>
<td>19.000</td>
<td>call</td>
<td>2014-11</td>
<td>voicemail</td>
<td>voicemail</td>
</tr>
<tr>
<td>10</td>
<td>16/10/14 16:21</td>
<td>1183.000</td>
<td>call</td>
<td>2014-11</td>
<td>Three</td>
<td>mobile</td>
</tr>
<tr>
<td>11</td>
<td>16/10/14 22:18</td>
<td>1.000</td>
<td>sms</td>
<td>2014-11</td>
<td>Meteor</td>
<td>mobile</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
The main columns in the file are:

1. **date**: The date and time of the entry
2. **duration**: The duration (in seconds) for each call, the amount of data (in MB) for each data entry, and the number of texts sent (usually 1) for each sms entry.
3. **item**: A description of the event occurring – can be one of call, sms, or data.
4. **month**: The billing month that each entry belongs to – of form ‘YYYY-MM’.
5. **network**: The mobile network that was called/texted for each entry.
6. **network_type**: Whether the number being called was a mobile, international (‘world’), voicemail, landline, or other (‘special’) number.

Phone numbers were removed for privacy. The date column can be parsed using the extremely handy `dateutil` library.

```python
import pandas as pd
import dateutil

# Load data from csv file
data = pd.DataFrame.from_csv('phone_data.csv')
# Convert date from string to date times
data['date'] = data['date'].apply(dateutil.parser.parse, dayfirst=True)
```

### Summarising the DataFrame

Once the data has been loaded into Python, Pandas makes the calculation of different statistics very simple. For example, mean, max, min, standard deviations and more for columns are easily calculable:

```python
# How many rows the dataset
data['item'].count()
Out[38]: 830

# What was the longest phone call / data entry?
data['duration'].max()
Out[39]: 10528.0

# How many seconds of phone calls are recorded in total?
data['duration'][data['item'] == 'call'].sum()
Out[40]: 92321.0

# How many entries are there for each month?
data['month'].value_counts()
Out[41]:
2014-11  230
2015-01  205
2014-12  157
2015-02  137
2015-03  101
dtype: int64

# Number of non-null unique network entries
data['network'].nunique()
Out[42]: 9
```

The need for custom functions is minimal unless you have very specific requirements. The full range of [basic statistics](#) that are quickly calculable and built into the base Pandas package are:
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of non-null observations</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>mad</td>
<td>Mean absolute deviation</td>
</tr>
<tr>
<td>median</td>
<td>Arithmetic median of values</td>
</tr>
<tr>
<td>min</td>
<td>Minimum</td>
</tr>
<tr>
<td>max</td>
<td>Maximum</td>
</tr>
<tr>
<td>mode</td>
<td>Mode</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute Value</td>
</tr>
<tr>
<td>prod</td>
<td>Product of values</td>
</tr>
<tr>
<td>std</td>
<td>Unbiased standard deviation</td>
</tr>
<tr>
<td>var</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>sem</td>
<td>Unbiased standard error of the mean</td>
</tr>
<tr>
<td>skew</td>
<td>Unbiased skewness (3rd moment)</td>
</tr>
<tr>
<td>kurt</td>
<td>Unbiased kurtosis (4th moment)</td>
</tr>
<tr>
<td>quantile</td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product</td>
</tr>
<tr>
<td>cummax</td>
<td>Cumulative maximum</td>
</tr>
<tr>
<td>cummin</td>
<td>Cumulative minimum</td>
</tr>
</tbody>
</table>

The `.describe()` function is a useful summarisation tool that will quickly display statistics for any variable or group it is applied to. The describe() output varies depending on whether you apply it to a numeric or character column.

Summarising Groups in the DataFrame

There’s further power put into your hands by mastering the Pandas “groupby()” functionality. Groupby essentially splits the data into different groups depending on a variable of your choice. For example, the expression `data.groupby('month')` will split our current DataFrame by month.

The groupby() function returns a GroupBy object, but essentially describes how the rows of the original data set has been split. the GroupBy object .groups variable is a dictionary whose keys are the computed unique groups and corresponding values being the axis labels belonging to each group. For example:

```
data.groupby(['month']).groups.keys()
```

```
len(data.groupby(['month']).groups['2014-11'])
Out[61]: 230
```
Functions like max(), min(), mean(), first(), last() can be quickly applied to the GroupBy object to obtain summary statistics for each group – an immensely useful function. This functionality is similar to the dplyr and plyr libraries for R. Different variables can be excluded / included from each summary requirement.

```python
# Get the first entry for each month
data.groupby('month').first()
Out[69]:
   date     duration  item  network  network_type
month
2014-11 2014-10-15 06:58:00 34.429     data      data      data
2014-12 2014-11-13 06:58:00 34.429     data      data      data
2015-01 2014-12-13 06:58:00 34.429     data      data      data
2015-02 2015-01-13 06:58:00 34.429     data      data      data
2015-03 2015-02-12 20:15:00 69.000    call     landline  landline
```

```python
# Get the sum of the durations per month
data.groupby('month')['duration'].sum()
Out[70]:
   month  
2014-11 26639.441
2014-12 14641.870
2015-01 18223.299
2015-02 15522.299
2015-03 22750.441
Name: duration, dtype: float64
```

```python
# Get the number of dates / entries in each month
data.groupby('month')['date'].count()
Out[74]:
   month  
2014-11     230
2014-12     157
2015-01     205
2015-02     137
2015-03     101
Name: date, dtype: int64
```

```python
# What is the sum of durations, for calls only, to each network
data[data['item'] == 'call'].groupby('network')['duration'].sum()
Out[78]:
   network
Meteor  7200
Tesco  13828
Three  36464
Vodafone  14621
landline  18433
voicemail  1775
Name: duration, dtype: float64
```

You can also group by more than one variable, allowing more complex queries.
# How many calls, sms, and data entries are in each month?
data.groupby(['month', 'item'])['date'].count()

Out[76]:

<table>
<thead>
<tr>
<th>month</th>
<th>item</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-11</td>
<td>call</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>sms</td>
<td>94</td>
</tr>
<tr>
<td>2014-12</td>
<td>call</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>sms</td>
<td>48</td>
</tr>
<tr>
<td>2015-01</td>
<td>call</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>sms</td>
<td>86</td>
</tr>
<tr>
<td>2015-02</td>
<td>call</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>sms</td>
<td>39</td>
</tr>
<tr>
<td>2015-03</td>
<td>call</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>sms</td>
<td>25</td>
</tr>
</tbody>
</table>

Name: date, dtype: int64

# How many calls, texts, and data are sent per month, split by network_type?
data.groupby(['month', 'network_type'])['date'].count()

Out[82]:

<table>
<thead>
<tr>
<th>month</th>
<th>network_type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-11</td>
<td>data</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>landline</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>mobile</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>special</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>voicemail</td>
<td>6</td>
</tr>
<tr>
<td>2014-12</td>
<td>data</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>landline</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>mobile</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>voicemail</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>world</td>
<td>4</td>
</tr>
<tr>
<td>2015-01</td>
<td>data</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>landline</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>mobile</td>
<td>160</td>
</tr>
</tbody>
</table>

Groupby output format – Series or DataFrame?
The output from a groupby and aggregation operation varies between Pandas Series and Pandas Dataframes, which can be confusing for new users. As a rule of thumb, if you calculate more than one column of results, your result will be a Dataframe. For a single column of results, the agg function, by default, will produce a Series.

You can change this by selecting your operation column differently:

data.groupby('month')['duration'].sum() # produces Pandas Series
data.groupby('month')[['duration']].sum() # Produces Pandas DataFrame

The groupby output will have an index or multi-index on rows corresponding to your chosen grouping variables. To avoid setting this index, pass “as_index=False” to the groupby operation.
data.groupby('month', as_index=False).agg({'duration': "sum"})

Using the as_index parameter while Grouping data in pandas prevents setting a row index on the result.

Multiple Statistics per Group

The final piece of syntax that we’ll examine is the “agg()” function for Pandas. The aggregation functionality provided by the agg() function allows multiple statistics to be calculated per group in one calculation. The syntax is simple, and is similar to that of MongoDB’s aggregation framework.

There were substantial changes to the Pandas aggregation function in May of 2017. Renaming of variables within the agg() function no longer functions as in the diagram below – see notes.

Aggregation of variables in a Pandas Dataframe using the agg() function. Note that in Pandas versions 0.20.1 onwards, the renaming of results needs to be done separately.

Applying a single function to columns in groups

Instructions for aggregation are provided in the form of a python dictionary or list. The dictionary keys are used to specify the columns upon which you’d like to perform operations, and the dictionary values to specify the function to run.

For example:

# Group the data frame by month and item and extract a number of stats from each group
data.groupby(['month', 'item']).agg({'duration': 'sum',
    'network_type': "count",
    'date': 'first'})

# find the sum of the durations for each group
# find the number of network type entries
# get the first date per group
The aggregation dictionary syntax is flexible and can be defined before the operation. You can also define functions inline using "lambda" functions to extract statistics that are not provided by the built-in options.

```python
# Define the aggregation procedure outside of the groupby operation
aggregations = {
    'duration': 'sum',
    'date': lambda x: max(x) - 1
}
data.groupby('month').agg(aggregations)
```

Applying multiple functions to columns in groups

To apply multiple functions to a single column in your grouped data, expand the syntax above to pass in a list of functions as the value in your aggregation dataframe. See below:

```python
# Group the data frame by month and item and extract a number of stats from each group
data.groupby(['month', 'item']).agg({'duration': [min, max, sum],
                                  'network_type': 'count',
                                  'date': [min, 'first', 'nunique']})
```

The `agg()` syntax is flexible and simple to use. Remember that you can pass in custom and lambda functions to your list of aggregated calculations, and each will be passed the values from the column in your grouped data.

Renaming grouped statistics from groupby operations

When multiple statistics are calculated on columns, the resulting dataframe will have a multi-index set on the column axis. This can be difficult to work with, and I typically have to rename columns after a groupby operation.

One option is to drop the top level (using `.droplevel`) of the newly created multi-index on columns using:

```python
grouped = data.groupby('month').agg("duration": [min, max, mean])
grouped.columns = grouped.columns.droplevel(level=0)
grouped.rename(columns={"min": "min_duration", "max": "max_duration", "mean": "mean_duration"})
grouped.head()
```

However, this approach loses the original column names, leaving only the function names as column headers. A neater approach, as suggested to me by a reader, is using the `ravel()` method on the grouped columns. Ravel() turns a Pandas multi-index into a simpler array, which we can combine into sensible column names:

```python
grouped = data.groupby('month').agg("duration": [min, max, mean])
# Using ravel, and a string join, we can create better names for the columns:
grouped.columns = ["_".join(x) for x in grouped.columns.ravel()]
```

Quick renaming of grouped columns from the groupby() multi-index can be achieved using the ravel() function.