Introduction

Since the data science world is all about "Big data", I decided to look at the different ways to manipulate datasets that are too large to be loaded into the memory of my laptop.

This is the first of a series where I look at big datasets, and in this case I'll be using the `chunksize` parameter of pandas `read_csv` to manage a large file size.

Dataset description

We'll use the SmartMeter Energy Consumption Data in London Households dataset, which according to the website contains:

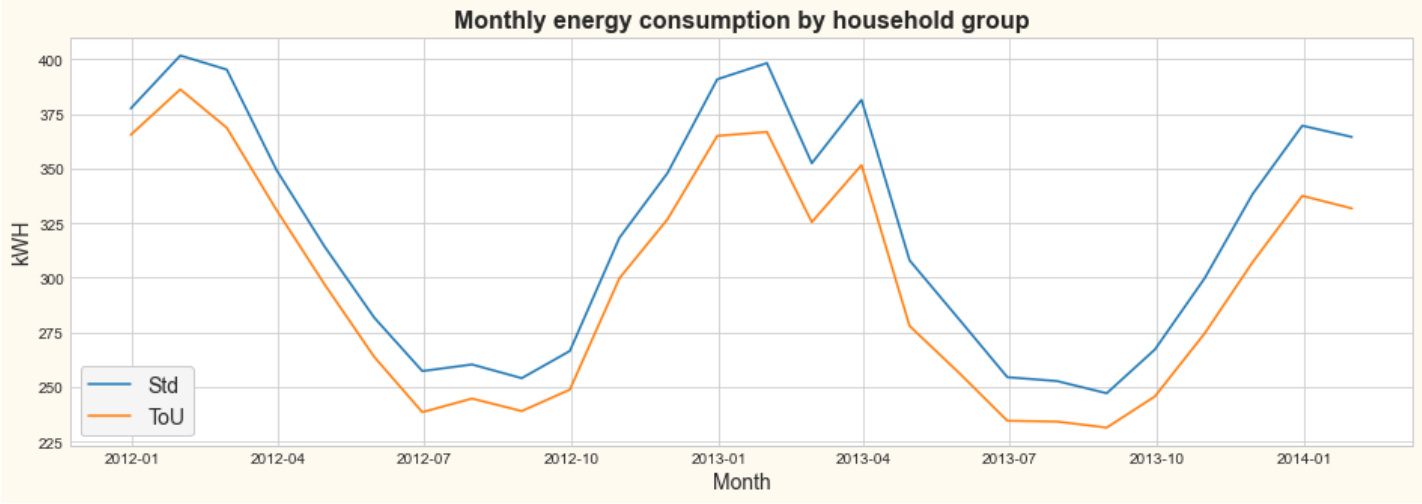
> Energy consumption readings for a sample of 5,567 London Households that took part in the UK Power Networks led Low Carbon London project between November 2011 and February 2014.

The households were divided into two groups:

- Those who were sent Dynamic Time of Use (dToU) energy prices (labelled "High", "Medium", or "Low") a day in advance of the price being applied.
- Those who were subject to the Standard tariff.

One aim of the study was to see if pricing knowledge would affect energy consumption behaviour.

Results



The results show the expected seasonal variation with a clear difference between the two groups, suggesting that energy price knowledge does indeed help reduce energy consumption.

The rest of the notebook shows how this chart was produced from the raw data.

Accessing the data

The data is downloadable as a single zip file which contains a csv file of 167 million rows. If the `curl` command doesn't work (and it will take a while as it's a file of 800MB), you can download the file here and put it in the folder `data` which is in the folder where this notebook is saved.

In [ ]:
```
!curl "https://data.london.gov.uk/download/smartmeter-energy-use-data-in-london-households/3527bf39-d93e-4071-8451-df2ade1ea4f2/LCL-FullData.zip" --location -o "data/LCL-FullData.zip"
```

First we unzip the data. This may take a while! Alternatively you can unzip it manually using whatever unzip utility you have. Just make sure the extracted file is in a folder called `data` within the folder where your notebook is saved.

In [1]:
```
!unzip "data/LCL-FullData.zip" -d "data"
```

```
Archive:  data/LCL-FullData.zip
  inflating: data/CC_LCL-FullData.csv
```

Examining the data

In [2]:
```
import pandas as pd
```

The standard way to load the csv file would be `data = pd.read_csv('data/CC_LCL-FullData.csv')`, but at 8GB that isn't feasible on my computer. So instead we use the `chunksize` parameter which loads data in "chunks", each containing the specified number of rows (apart from the last chunk). The chunks are then identified as separate dataframes that are referenced within a `TextFileReader` object.

We'll load our file in chunks of 1 million rows.

In [3]:
```
chunks = pd.read_csv('data/CC_LCL-FullData.csv', chunksize=1000000)
type(chunks)
```

Out[3]:
```
pandas.io.parsers.readers.TextFileReader
```

Note that the data has not yet been loaded. Using the `TextFileReader` means that the data is processed "lazily" i.e. only when needed. We can iterate through each chunk and act on each one at a time.

Let's have a look at just the first chunk, using the standard methods of `describe()`, `head()` and `info()`.

In [4]:
```
from IPython.display import HTML

table_style = [{
    'selector' : 'caption',
    'props' : [
        ('font-size', '16px'),
        ('color', 'black'),
        ('font-weight', 'bold'),
        ('text-align', 'left')
    ]
}]

for chunk in chunks:

    display(
        chunk.describe(include='all')
        .style.set_caption('Describe')
```

```
        .set_table_styles(table_style)
    )

    display(
        chunk.head()
        .style.set_caption('Head')
        .set_table_styles(table_style)
    )

    display(HTML('<br><span style="font-weight: bold; font-size: 16px">Info</span>'))
    display(chunk.info())

    break # Just the first chunk
```

Describe

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)
count	1000000	1000000	1000000	1000000
unique	30	1	39102	4801
top	MAC000018	Std	2012-11-20 00:00:00.0000000	0
freq	39082	1000000	58	45538

Head

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)
0	MAC000002	Std	2012-10-12 00:30:00.0000000	0
1	MAC000002	Std	2012-10-12 01:00:00.0000000	0
2	MAC000002	Std	2012-10-12 01:30:00.0000000	0
3	MAC000002	Std	2012-10-12 02:00:00.0000000	0
4	MAC000002	Std	2012-10-12 02:30:00.0000000	0

Info

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 4 columns):
 #   Column                  Non-Null Count    Dtype
---  ------                  --------------    -----
 0   LCLid                   1000000 non-null  object
 1   stdorToU                1000000 non-null  object
 2   DateTime                1000000 non-null  object
 3   KWH/hh (per half hour)  1000000 non-null  object
dtypes: object(4)
memory usage: 30.5+ MB
None
```

The column `KWH/hh (per half hour)` is of type `object` and not `float` which is surprising, so that probably means there are some non-numeric values we'll need to deal with.

Pre-processing

We'll end up renaming the columns to make them more readable, so we'll define the names now.

In [5]:
```
col_renaming = {
    'LCLid' : 'Household ID',
    'stdorToU' : 'Tariff Type',
    'KWH/hh (per half hour) ' : 'kWh'
}
```

Let's try converting the kWh data column to numeric.

In [6]:
```
chunks = pd.read_csv('data/CC_LCL-FullData.csv', chunksize=1000000)
for chunk in chunks:
    chunk.rename(
        columns = col_renaming,
        inplace=True
    )
    chunk.loc[:, 'kWh'] = pd.to_numeric(chunk['kWh'])
    break
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
File ~\miniconda3\envs\datascience\lib\site-packages\pandas\_libs\lib.pyx:2315, in
pandas._libs.lib.maybe_convert_numeric()

ValueError: Unable to parse string "Null"

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
Input In [6], in <cell line: 2>()
      2 for chunk in chunks:
      3     chunk.rename(
      4         columns = col_renaming,
      5         inplace=True
      6     )
----> 7     chunk.loc[:, 'kWh'] = pd.to_numeric(chunk['kWh'])
      8     break

File ~\miniconda3\envs\datascience\lib\site-packages\pandas\core\tools\numeric.py:184, in
to_numeric(arg, errors, downcast)
    182 coerce_numeric = errors not in ("ignore", "raise")
    183 try:
--> 184     values, _ = lib.maybe_convert_numeric(
    185         values, set(), coerce_numeric=coerce_numeric
    186     )
    187 except (ValueError, TypeError):
    188     if errors == "raise":

File ~\miniconda3\envs\datascience\lib\site-packages\pandas\_libs\lib.pyx:2357, in
pandas._libs.lib.maybe_convert_numeric()

ValueError: Unable to parse string "Null" at position 3240
```

The error message tells us that there are string entries of "Null". Let's have a test of removing all those from the first chunk.

In [7]:
```
test_chunks = pd.read_csv('data/CC_LCL-FullData.csv', chunksize=1000000)

for chunk in test_chunks:
    chunk.rename(columns=col_renaming, inplace=True)
    chunk.loc[:, 'kWh'] = pd.to_numeric(chunk['kWh'], errors='coerce')
    chunk.dropna(subset=['kWh'], inplace=True)
    print(f"{chunk.shape[0]} rows remaining after dropping rows where kWH is 'Null'")
    break
```

```
999971 rows remaining after dropping rows where kWH is 'Null'
```

That seems to work as we now have fewer than 1,000,000 rows. Now we know how to do that, let's consider how we can create some useful data.

Reducing the data

The goal here is to ***reduce*** the data by aggregating it in some way. Since we know that we have data in half-hour intervals, we'll aggregate it to daily data by summing over each 24-hour period. That should reduce the number of rows by a factor of about 48.

When I did this, I tested on just 2 chunks to start with to test that everything works. That way I could build up step-by-step without waiting to process the whole dataset, each time chaining another method once I saw a satisfactory result for the previous transformation. When we iterate through, for each chunk the various transformations are:

1. Convert the timestamp data to date format, ready for grouping
2. Rename the columns
3. Convert the kWh data to numeric, converting any errors to `NaN`
4. Drop any rows where the kWh value is `NaN`
5. Aggregate the data from half-hourly to daily by grouping by `Household ID`, `Tariff Type`, and `Date` and summing the `kWh` data
6. Add the aggregated chunk data to the running total of aggregated data

In [8]:
```
def process_chunks(chunks, max_chunks=None, display_each_trunk=False):

    count = 1

    output = None

    for chunk in chunks:

        # Convert timestamp to date
        chunk['Date'] = pd.to_datetime(chunk['DateTime']).dt.date

        # Rename columns
        chunk.rename(columns=col_renaming, inplace=True)

        # Remove rows where kWh value is 'Null'
        chunk.loc[:, 'kWh'] = pd.to_numeric(chunk['kWh'], errors='coerce')
        chunk.dropna(subset=['kWh'], inplace=True)

        # Aggregate from half-hourly to daily kWh data
        daily_data = (
            chunk.groupby(
                by = ['Household ID', 'Tariff Type', 'Date']
            )
            .agg({
                'kWh' : 'sum'
            })
        )

        # Display for checking purposes
        if display_each_trunk:
            display(HTML(f'<span style="font-weight: bold; font-size: 16px">Chunk {count}
</span>'))
            display(daily_data.head(10))
```

```
            print()

        # Add to running total
        if output is None:
            output = daily_data
        else:
            output = output.add(daily_data, fill_value=0)

        # Display progress every 10 chunks.
        if count % 10 == 0:
            print(count, end=', ')

        if max_chunks and count == max_chunks:
            break
        count += 1

    return output
```

In [9]:

```
chunks = pd.read_csv('data/CC_LCL-FullData.csv', chunksize=1000000)
test_daily_summary = process_chunks(chunks, 2, True)

# Display the aggregated totals
display(HTML(f'<span style="font-weight: bold; font-size: 16px">Total</span>'))
display(test_daily_summary.head(10))
```

Chunk 1

			kWh
Household ID	**Tariff Type**	**Date**	
MAC000002	**Std**	**2012-10-12**	7.098
		2012-10-13	11.087
		2012-10-14	13.223
		2012-10-15	10.257
		2012-10-16	9.769
		2012-10-17	10.885
		2012-10-18	10.751
		2012-10-19	8.431
		2012-10-20	17.578
		2012-10-21	24.490

Chunk 2

			kWh
Household ID	**Tariff Type**	**Date**	
MAC000036	**Std**	**2012-11-08**	1.359
		2012-11-09	2.659
		2012-11-10	2.593
		2012-11-11	2.542
		2012-11-12	2.701
		2012-11-13	2.364
		2012-11-14	2.785
		2012-11-15	2.769
		2012-11-16	2.542
		2012-11-17	2.971

Total

			kWh
Household ID	**Tariff Type**	**Date**	
MAC000002	**Std**	**2012-10-12**	7.098
		2012-10-13	11.087
		2012-10-14	13.223
		2012-10-15	10.257
		2012-10-16	9.769
		2012-10-17	10.885
		2012-10-18	10.751
		2012-10-19	8.431
		2012-10-20	17.578
		2012-10-21	24.490

Now we can summarize the full data. This will take some time! The counter shows progress by updating every time another 10 chunks are processed.

In [10]:
```
chunks = pd.read_csv('data/CC_LCL-FullData.csv', chunksize=1000000)
daily_summary = process_chunks(chunks)

# Display the aggregated totals
display(HTML(f'<span style="font-weight: bold; font-size: 16px">Total</span>'))
display(daily_summary.head(10))
```

```
 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160,
```

Total

			kWh
Household ID	**Tariff Type**	**Date**	
MAC000002	**Std**	**2012-10-12**	7.098
		2012-10-13	11.087
		2012-10-14	13.223
		2012-10-15	10.257
		2012-10-16	9.769
		2012-10-17	10.885
		2012-10-18	10.751
		2012-10-19	8.431
		2012-10-20	17.578
		2012-10-21	24.490

Saving aggregated data

Now that we have reduced the data down to about 3 million rows it should be managable in a single dataframe. It's useful to save the data so that we don't have to re-run the aggregation every time we want to work on the aggregated data.

(We can also use the `chunksize` parameter to write to a csv in chunks, but it shouldn't be necessary now that we've reduced the size.)

In [11]:
```
daily_summary.to_csv("data/daily-summary-data.gz")
```

Analysing the data

We should be able to load the aggregated data without using chunks.

In [12]:
```
saved_daily_summary = pd.read_csv("data/daily-summary-data.gz")
```

Let's check whether there are any households that switched from one group to another, as it would be interesting to see if we could see a behaviour change following the switch.

First, let's aggregate over all time, grouping by `Household ID` and `Tariff Type`.

In [13]:
```
kwh_by_household_and_tariff_type = saved_daily_summary.groupby(by=['Household ID', 'Tariff Type']).agg({
    'kWh' : 'sum'
})
kwh_by_household_and_tariff_type.head()
```

Out[13]:

Household ID	Tariff Type	kWh
MAC000002	Std	6101.138001
MAC000003	Std	14104.433003
MAC000004	Std	1120.788000
MAC000005	ToU	2911.808000
MAC000006	Std	2168.325000

Now we can count the number of tariff types by household. We expect it to be either 1 or 2 for each. Then we can get the unique values...

In [14]:
```
tariff_types_by_household = kwh_by_household_and_tariff_type.groupby('Household ID').count()
tariff_types_by_household['kWh'].unique()
```

Out[14]:
```
array([1], dtype=int64)
```

... and since we have only 1s we know that no household changed tariff type.

So instead we will compare the two groups side by side. Out of interest let's see what sort of data coverage we have. First we re-organize so that we have households as columns and dates as rows.

In [15]:
```
summary_table = daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Household ID',
    aggfunc='sum'
)
```

In [16]:
```
summary_table.head().dropna(axis=1)
```

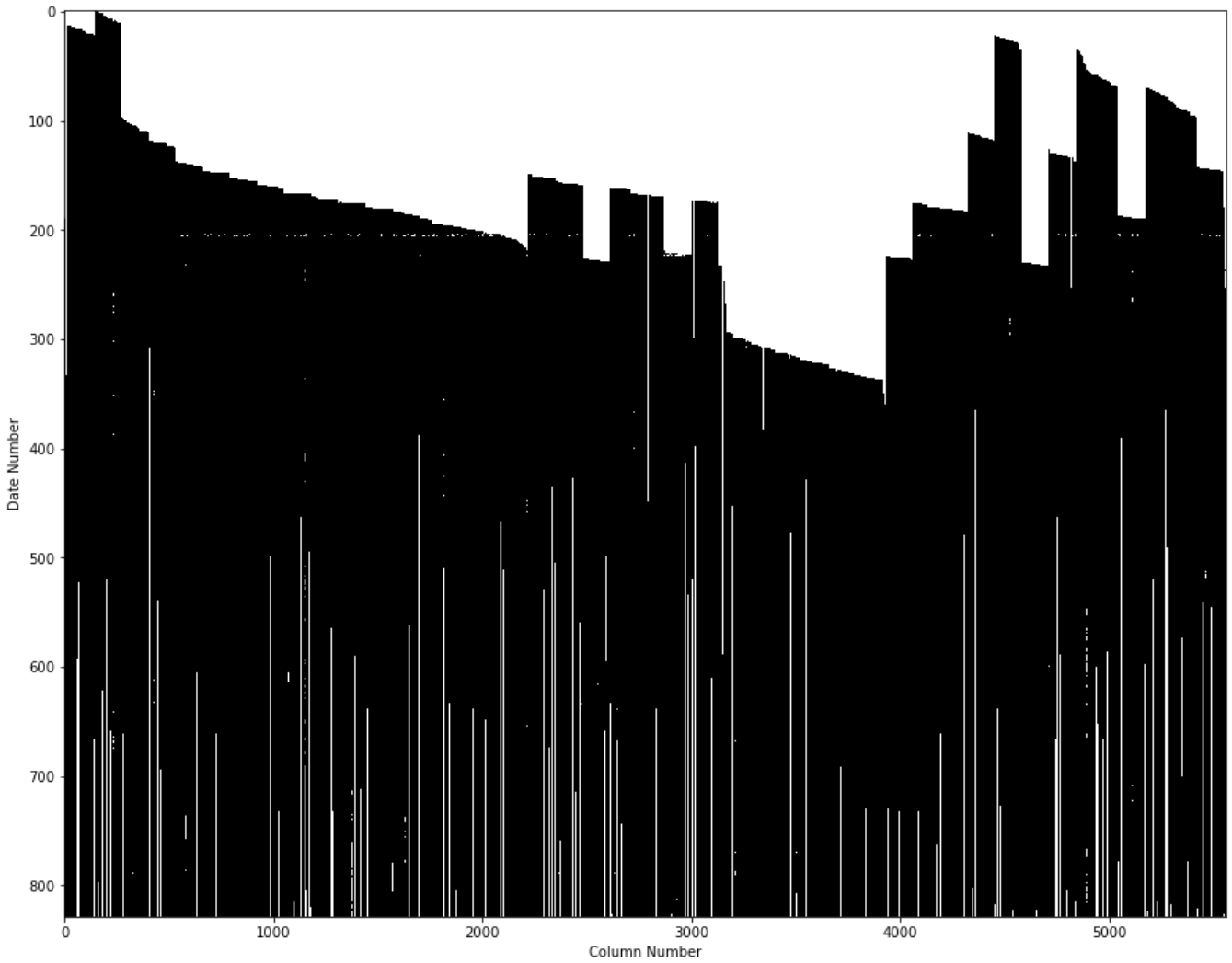
Out[16]:

Household ID	MAC000145	MAC000146	MAC000147	MAC000148	MAC000149	MAC000150	MAC000151	MAC
Date								
2011-11-23	8.952	5.619	3.036	1.283	2.287	9.254	3.273	
2011-11-24	12.135	8.564	7.489	2.304	4.331	9.606	4.620	
2011-11-25	13.720	6.743	6.185	2.324	4.373	12.685	4.783	
2011-11-26	15.234	6.029	6.964	2.295	4.335	9.606	4.935	
2011-11-27	13.189	5.740	7.912	2.302	4.375	14.387	4.553	

Then we can plot where we have data (black) and where we don't (white).

In [17]:
```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(15, 12))
plt.imshow(summary_table.isna(), aspect="auto", interpolation="nearest", cmap="gray")
plt.xlabel("Column Number")
plt.ylabel("Date Number");
```



Despite a slightly patchy data coverage, averaging by tariff type across all households for each day should give us a useful comparison.

In [18]:
```
daily_mean_by_tariff_type = daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Tariff Type',
    aggfunc='mean'
)
daily_mean_by_tariff_type.head()
```

Out[18]:

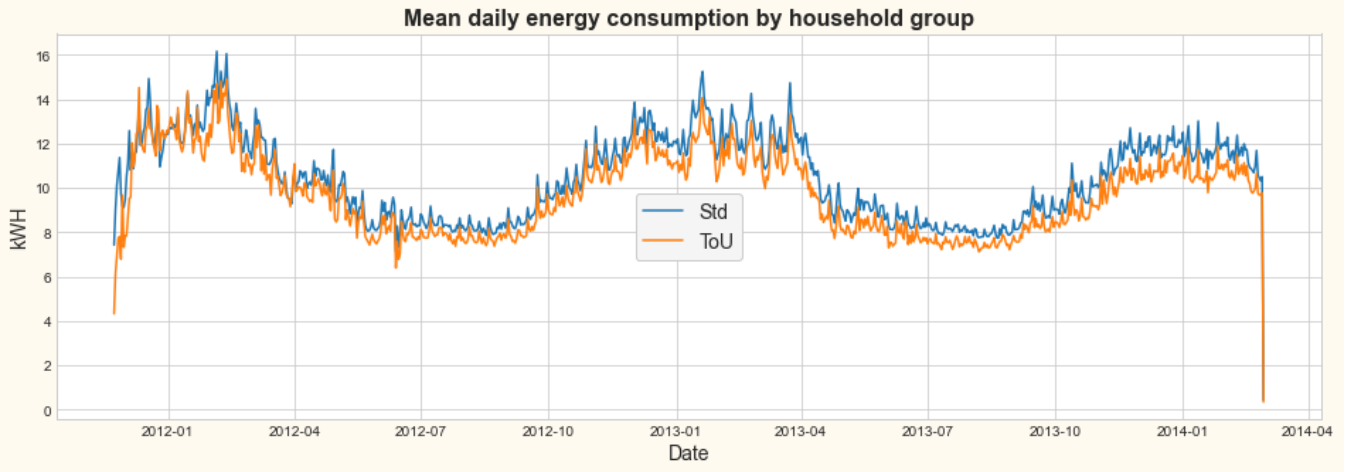
Tariff Type	Std	ToU
Date		
2011-11-23	7.430000	4.327500
2011-11-24	8.998333	6.111750
2011-11-25	10.102885	6.886333
2011-11-26	10.706257	7.709500
2011-11-27	11.371486	7.813500

Finally we can plot the two sets of data.

In [19]:
```
plt.style.use('seaborn-whitegrid')

plt.figure(figsize=(16, 5), facecolor='floralwhite')
for tariff in daily_mean_by_tariff_type.columns.to_list():
    plt.plot(
        daily_mean_by_tariff_type.index.values,
        daily_mean_by_tariff_type[tariff],
        label = tariff
    )

plt.legend(loc='center', frameon=True, facecolor='whitesmoke', framealpha=1, fontsize=14)
plt.title(
    'Mean daily energy consumption by household group',
    fontdict = {'fontsize' : 16, 'fontweight' : 'bold'}
)
plt.xlabel('Date', fontsize = 14)
plt.ylabel('kWH', fontsize = 14)
plt.show()
```



The pattern looks seasonal which makes sense given heating energy demand.

It also looks like there's a difference between the two groups with the ToU group tending to consume less, but the display is too granular. Let's aggregate again into months.

In [20]:
```
daily_mean_by_tariff_type.index = pd.to_datetime(daily_mean_by_tariff_type.index)
daily_mean_by_tariff_type.head()
```

Out[20]:

Tariff Type	Std	ToU
Date		
2011-11-23	7.430000	4.327500
2011-11-24	8.998333	6.111750
2011-11-25	10.102885	6.886333
2011-11-26	10.706257	7.709500
2011-11-27	11.371486	7.813500

We can see that the data starts partway through November 2011, so we'll start from 1 December. It looks like the data finishes perfectly at the end of February, but the last value looks suspiciously low compared to the others. It seems likely the data finished part way through the last day. This may be a problem elsewhere in the data too, but

it shouldn't have an enormous effect as at worst it will reduce the month's energy consumption for that household by two days (one at the beginning and one at the end).

In [21]:
```
monthly_mean_by_tariff_type = daily_mean_by_tariff_type['2011-12-01' : '2014-01-31'].resample('M').sum()
monthly_mean_by_tariff_type.head()
```

Out[21]:

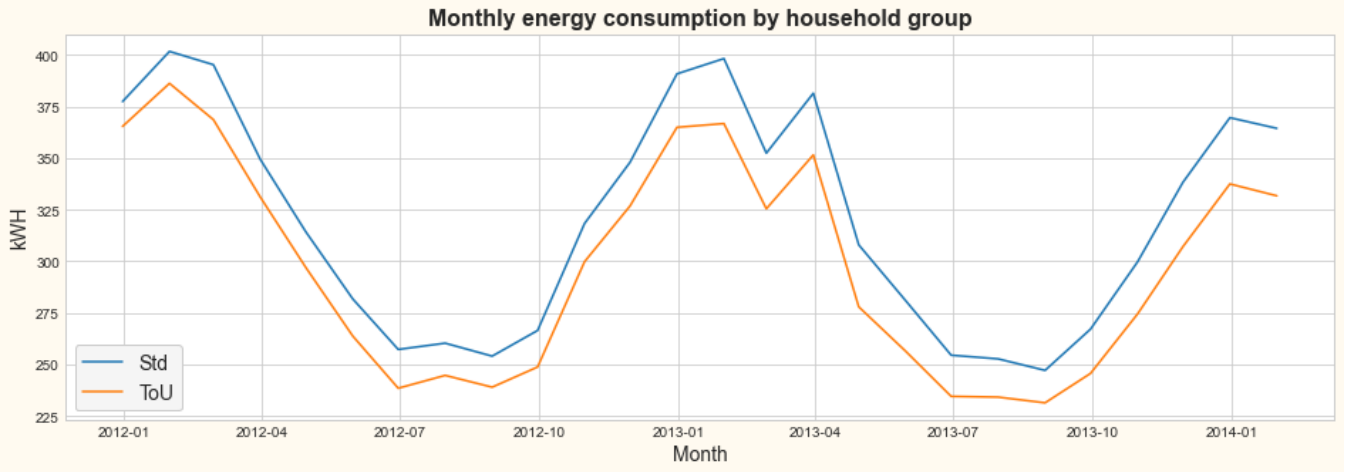
Tariff Type	Std	ToU
Date		
2011-12-31	377.443042	365.391597
2012-01-31	401.744672	386.253703
2012-02-29	395.294296	368.663764
2012-03-31	349.367317	331.095386
2012-04-30	314.323216	297.032370

In [22]:
```
plt.figure(figsize=(16, 5), facecolor='floralwhite')
for tariff in daily_mean_by_tariff_type.columns.to_list():
    plt.plot(
        monthly_mean_by_tariff_type.index.values,
        monthly_mean_by_tariff_type[tariff],
        label = tariff
    )

plt.legend(loc='lower left', frameon=True, facecolor='whitesmoke', framealpha=1, fontsize=14)
plt.title(
    'Monthly energy consumption by household group',
    fontdict = {'fontsize' : 16, 'fontweight' : 'bold'}
)
plt.xlabel('Month', fontsize = 14)
plt.ylabel('kWH', fontsize = 14)

# Uncomment for a copy to display in results
plt.savefig(fname='images/result1.png', bbox_inches='tight')

plt.show()
```



The pattern is much clearer and there is an obvious difference between the two groups of consumers.

Note that the chart does not show mean monthly energy consumption, but the sum over each month of the daily means. To calculate true monthly means we would need to drop the daily data for each household where the data was incomplete for a month. Our method should give a reasonable approximation.