In [1]:
```
# imports and setup
from os import listdir
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn
%matplotlib inline
```

In [2]:
```
#import data
d16 = pd.read_csv('data/NYCgov_Poverty_Measure_Data__2016_.csv')
d16_orig = d16.copy()
```

Data Overview - 2016 Dataset

The data in the 2016 dataset is **mostly clean of null values**. It has **79** columns, and only **9** columns containing nulls (see below).

The data dictionary shows that for most of these columns, a null value actually carries information rather than showing a lack of information. For example, the 'SCHL' column (year in school) is null for anyone under age 3. So, it's easy to clean these up by assigning a new value to cells of that sort. I'll do that for the columns below where possible.

In [3]:
```
print('Rows in the 2016 Dataset: ' + str(len(d16)) + '\n')

nulls = d16.isnull().sum()
nulls = nulls[nulls > 0].sort_values(ascending=False)
print('Number of Nulls by Column in the 2016 Dataset:' + '\n' + str(nulls))
```

```
Rows in the 2016 Dataset: 68644

Number of Nulls by Column in the 2016 Dataset:
ENG              38015
JWTR             36009
WKW              32406
ESR              12198
MSP              11419
LANX              3738
EducAttain        2240
SCHL              2240
Off_Threshold        2
dtype: int64
```

Cleaning up the ENG Column

For the 'ENG' column, a null value means the person is less than five years old or only speaks English. The pre-set values for this field are 1 ('Very Well') to 4 ('Not at all').

Let's make 0 mean 'less than five years old'. Some other columns are similar, so even though this seems to be in the wrong place relative to the pre-set values, it will be consistent with the other columns. That way mistakes are less likely.

On the other end of the scale, let's make 5 mean 'speaks only English'.

In [4]:
```
# Create temporary column 'FivePlus' for easy manipulation
d16['FivePlus'] = d16.AGEP >= 5

# Show the situation before changes
print('Nulls in column \'ENG\' by age group: ' + str(d16.loc[d16.ENG.isnull(), 'Fiv

# Change the rows where ENG is null and over age five to be 5 (they only speak Engl
# Change the rows where ENG is null and under age five to be 0
d16.loc[((d16.ENG.isnull()) & (d16.FivePlus == True)), 'ENG'] = 5
d16.loc[((d16.ENG.isnull()) & (d16.FivePlus == False)), 'ENG'] = 0

# Show the situation after changes and remove the temporary column
d16.drop('FivePlus', axis=1, inplace=True)
print('Nulls in column \'ENG\' now: ' + str(len(d16[d16.ENG.isnull()])))
```

```
Nulls in column 'ENG' by age group: FivePlus
False     3738
True     34277
Name: FivePlus, dtype: int64

Nulls in column 'ENG' now: 0
```

Cleaning up a bunch of columns with zeros:

For any nulls in the columns below, I will replace with zeros all the way down:

- 'JWTR' is means of transportation to work. A null means the person is not a worker for any reason (too young, unemployed, armed forces, etc.).
- 'WKW' is weeks worked; null means they are not a worker.
- 'ESR' is employment status; null means they are less than 16 years old.
- 'MSP' means 'married, spouse present'; any null means less than 15 years old.
- 'LANX' means 'language other than English spoken at home'; any null means less than 5 years old.
- 'EducAttain' means educational attainment; any null means less than 3 years old.
- 'SCHL' means educational attainment (the 'EducAttain' column referenced above was taken from this column); any null means less than 3 years old.

In [5]:
```
d16.loc[d16.JWTR.isnull(), 'JWTR'] = 0
d16.loc[d16.WKW.isnull(), 'WKW'] = 0
d16.loc[d16.ESR.isnull(), 'ESR'] = 0
d16.loc[d16.MSP.isnull(), 'MSP'] = 0
d16.loc[d16.LANX.isnull(), 'LANX'] = 0
d16.loc[d16.EducAttain.isnull(), 'EducAttain'] = 0
d16.loc[d16.SCHL.isnull(), 'SCHL'] = 0
```

How do we look now?

We should only have two NaN rows left. Let's check that we only have two:

In [6]:
```
new_nulls = d16.isnull().sum()
new_nulls = new_nulls[new_nulls > 0].sort_values(ascending=False)
print(new_nulls)
```

```
Off_Threshold    2
dtype: int64
```

Double-check the edits

We only changed cells that had NaNs in them; every other value didn't change. So we subtract a copy of our original dataset from the updated dataset, and count the non-zero rows by column. If it's the same as the NaNs by column in our original dataset, we know we didn't change anything else.

In [7]:
```
# create a check dataframe - the original minus the edited version.
check = d16_orig - d16

# Since we only edited the NaNs, the number of changes by column should be exactly
# of NaNs per column.  Count the number of changes by column
check1 = check.ne(0).sum()
check1 = check1[check1 > 0].sort_values(ascending=False)

# Count the original number of nulls by column
d16_orig_nulls = d16_orig.isnull().sum()
d16_orig_nulls = d16_orig_nulls[d16_orig_nulls > 0].sort_values(ascending=False)

print('Did we change only the NaNs?  By column: \n' + str(d16_orig_nulls == check1)
```

```
Did we change only the NaNs?  By column:
ENG              True
JWTR             True
WKW              True
ESR              True
MSP              True
LANX             True
EducAttain       True
SCHL             True
Off_Threshold    True
dtype: bool
```

Drop the two remaining NaN rows

By inspection, the two remaining NaN rows are two rows without an official poverty threshold, that are unrelated to other cells. As much as I'd like to go fill in the missing data, we have a lot of other work to do (10 more datasets like this!). Plus, these are two observations out of 68,000+. Let's drop them and move on. We'll check afterward that we now have 68,642 rows.

In [8]:
```
mask = d16.isna().any(1)
print('Number of rows before dropping: ' + str(len(d16)))
d16.dropna(inplace=True)
print('Number of rows after dropping: ' + str(len(d16)))
```

```
Number of rows before dropping: 68644
Number of rows after dropping: 68642
```

Check for outliers, part 1

Each of the columns can take a range of values. I'll create a dictionary where the key is the column name, and the values are tuples of the min and max values. Then I'll check that each column is within the expected min and max values.

In [9]:
```
COLS_LIMS_DICT = {'SERIALNO': (0, 9999999), 'SPORDER': (0,20), 'PWGTP': (0, 9999),

def check_col_limits(series, min, max):
    '''Check the limits of a Pandas Series against expected values.
    Input: series name, column name, expected minimum, and expected maximum.
    Output: No return value. If a value in the column is outside the limits, print

    if series.min() < min:
        print(str(series.name) + ' has stated min of ' + "{:,}".format(min) + ' but
    if series.max() > max:
        print(str(series.name) + ' has stated max of ' + "{:,}".format(max) + ' but

def check_df_limits(df, limits):
    '''Check the limits of a DataFrame against expected values.
    Input: dataframe name and a dictionary with keys of column names, values of a t
    Output: No return value. Print a message for any columns with values outside it

    for col, lims in COLS_LIMS_DICT.items():
        if col in df:
            check_col_limits(df[col], lims[0], lims[1])
        else:
            print(str(col) + ' is not in dataframe.')

check_df_limits(d16, COLS_LIMS_DICT)
```

```
DS is not in dataframe.
HousingStatus has stated max of 9 but actual max of 10.00.
NYCgov_Income has stated min of -25,000 but actual min of -423,810.62.
NYCgov_Income has stated max of 999,999 but actual max of 1,662,673.94.
NYCgov_IncomeTax has stated max of 99,999 but actual max of 1,341,696.07.
TaxUnit_FILETYPE has stated max of 3 but actual max of 4.00.
```

After changing some of the expected limits to include very-high or very-low values relative to taxes, we have a few classification columns (above) where the coded values are outside the values that the data dictionary allows.

I've already made some changes via my code, but I wanted to highlight a few key changes/assumptions:

- HousingStatus of 10 must mean an errant or unknown value, so below we'll make it 0 meaning unknown/not available.
- Originally, the column 'EducAttain' had some unexpected zero values. It was created from the column 'SCHL', and in that column, a zero value means that the person is less than 3 years old. Each column had exactly 2239 zero values, so it's a match; I updated the allowable values for the column 'EducAttain' to include zero values.
- Originally, the 'NYCgov_SFR' column had some unexpected zeor values. It was built from the ACS data, and 0 means not in a subfamily. Turns out, 90%+ of the column was zeros. So I left it as-is and updated the allowable values.
- For the TaxUnit_FILESTAT and TaxUnit_FILETYPE columns, let's leave the zeros as an unknown/NA value.
- The TaxUnit_FILESTAT and TaxUnit_FILETYPE columns had some unexpected zeros. I updated the check to allow those as unknown. But for TaxUnit_FILETYPE values of 4, there's no reason for this; there are about 100 of these out of 60,000+ rows, so below we'll change those '4' values to 0.
- Note that the 'DS' column is not in this dataframe; it existed in some previous years' data, but not in this one. We'll update that below.
- Also note that NYCgov_Income has a huge negative min. We'll clean that up later.

In [10]:
```
d16.loc[d16.HousingStatus == 10, 'HousingStatus'] = 0
d16.loc[d16.TaxUnit_FILETYPE == 4, 'TaxUnit_FILETYPE'] = 0

check_df_limits(d16, COLS_LIMS_DICT)
```

```
DS is not in dataframe.
NYCgov_Income has stated min of -25,000 but actual min of -423,810.62.
NYCgov_Income has stated max of 999,999 but actual max of 1,662,673.94.
NYCgov_IncomeTax has stated max of 99,999 but actual max of 1,341,696.07.
```

Making a function to do it all

Most of what I've done above can be rolled into a function to hit all of the other annual datasets as well. I'll put it all together here:

In [11]:
```
def clean_set(df, name):
    '''Cleans an annual dataset in the NYC Poverty Measure data.
    Input: Dataframe from pd.read_csv on an annual dataset, and name of the dataset
    Output: no return value. Prints out results of an attempted automated cleanup.'

    print('###################################################################
    print('Starting output for dataset ' + name + '\n')

    # Create a copy for comparison later, then do initial cleanup
    df_orig = df.copy()

    # Create temporary column 'FivePlus' to make it easier to distribute the ENG Na
    df['FivePlus'] = df.AGEP >= 5

    # Change the rows where ENG is null and over age five to be 5 (they only speak 
    # Change the rows where ENG is null and under age five to be 0
    df.loc[((df.ENG.isnull()) & (df.FivePlus == True)), 'ENG'] = 5
    df.loc[((df.ENG.isnull()) & (df.FivePlus == False)), 'ENG'] = 0

    # Remove the temporary column created above
    df.drop('FivePlus', axis=1, inplace=True)

   # Change null values to zeros where the person is under age or the question does
    # (e.g. a five-year-old is neither married nor unmarried, they're just a five-y
    df.loc[df.JWTR.isnull(), 'JWTR'] = 0
    df.loc[df.WKW.isnull(), 'WKW'] = 0
    df.loc[df.ESR.isnull(), 'ESR'] = 0
    df.loc[df.MSP.isnull(), 'MSP'] = 0
    df.loc[df.LANX.isnull(), 'LANX'] = 0
    df.loc[df.EducAttain.isnull(), 'EducAttain'] = 0
    df.loc[df.SCHL.isnull(), 'SCHL'] = 0
    if 'DS' in df.columns:
        df.loc[df.DS.isnull(), 'DS'] = 0

    # Compare nulls between the original and new datasets
    check = df_orig - df
    check1 = check.ne(0).sum()
    check1 = check1[check1 > 0].sort_values(ascending=False)
    df_orig_nulls = df_orig.isnull().sum()
    df_orig_nulls = df_orig_nulls[df_orig_nulls > 0].sort_values(ascending=False)
    print('Number of changes by column equals original columns: \n' + str(df_orig_n

    # Drop any remaining rows with any NaNs and check we haven't dropped too many
    mask = df.isna().any(1)
    rows_before = len(df)
    print('Number of rows before dropping the remaining NaNs: ' + str(len(df)))
    df.dropna(inplace=True)
    print('Number of rows after dropping the remaining NaNs: ' + str(len(df)))
    rows_after = len(df)
    print('% of original rows remaining: ' + str(round(100 * rows_after/rows_before

    # Update HousingStatus and TaxUnit_FILETYPE columns so that any values outside 
    df.loc[df.HousingStatus == 10, 'HousingStatus'] = 0
    df.loc[df.TaxUnit_FILETYPE == 4, 'TaxUnit_FILETYPE'] = 0

    check_df_limits(df, COLS_LIMS_DICT)

    print('\n End of output for dataset ' + name)
    print('###################################################################
```

Pulling each year's data into dataframes

Let's create a dict of dataframes, one per year, so we can clean and review them individually before dumping them into one big dataframe.

Below I'll list out the files and filenames, then loop over the list to load each file as a dataframe and run our cleaning function on it.

The results below for all of the files look pretty good! There are three areas of concern:

- In the original run of this, over 5% of the 2005 dataset was getting thrown out because of NaNs. After investigating, I discovered that the DS column was coded as NaN when the person was less than five years old - so I edited our cleaning function to change these to 0, and the problem was solved.
- The 'WKW' column was coded differently from 2005-2007; below I'll create a function to manually recode those.
- The datasets from 2008 and prior have a column called 'DS' but not one called 'DIS' -- vice-versa for the later datasets. After reviewing the dataset, I'm comfortable changing the name of the column to 'DIS' for all of them. I'll do that below along with fixing the 'WKW' column.
- The other warnings are not of concern - the made-up boundaries that I included for checking purposes were occasionally too tight. No big deal.

In [12]:

```
files = sorted(list(filter(lambda files: files.endswith('.csv'), listdir('./data'))
names = ['d' + str(num).zfill(2) for num in range(5,17)]
data = {}
for file, name in zip(files, names):
    data[name] = pd.read_csv('./data/' + file)
    clean set(data[name], name)
```

```
################################################################################
Starting output for dataset d05

Number of changes by column equals original columns:
JWTR          True
ENG           True
WKW           True
ESR           True
MSP           True
DS            True
LANX          True
EducAttain    True
SCHL          True
dtype: bool

Number of rows before dropping the remaining NaNs: 60512
Number of rows after dropping the remaining NaNs: 60512
% of original rows remaining: 100.0

WKW has stated max of 6 but actual max of 52.00
```

Cleaning up the 2005-2007 dataframes

The 2005-2007 editions of the data had two major differences that we need to clean up:

- The DIS column was named DS, but was otherwise the same.
- The WKW (weeks worked) column was expressed as a number from 1-52, rather than coded into 6 values as the rest of the data.

Let's clean that up.

In [13]:
```
# Change the column name 'DS' to 'DIS' just to be consistent.
data['d05'].rename(columns={'DS': 'DIS'}, inplace=True)
data['d06'].rename(columns={'DS': 'DIS'}, inplace=True)
data['d07'].rename(columns={'DS': 'DIS'}, inplace=True)

def recode_WKW(df):
    '''Recodes the WKW column, in-place.
    Input: a dataframe with WKW coded by actual weeks worked, rather than in the 20
    Output: no return value. Just recodes by 2016 values, and shows the value_count

    df.loc[(df.WKW >= 1) & (df.WKW <= 13), 'WKW'] = 6
    df.loc[(df.WKW >= 14) & (df.WKW <= 26), 'WKW'] = 5
    df.loc[(df.WKW >= 27) & (df.WKW <= 39), 'WKW'] = 4
    df.loc[(df.WKW >= 40) & (df.WKW <= 47), 'WKW'] = 3
    df.loc[(df.WKW == 48) | (df.WKW == 49), 'WKW'] = 2
    df.loc[df.WKW >= 50, 'WKW'] = 1
    print(df.WKW.value_counts())
    print('Non-Nulls: ' + str(len(df[df.WKW.notnull()])))
    print('Nulls: ' + str(len(df[df.WKW.isnull()])))
    print('Total: ' + str(len(df[df.WKW.isnull()]) + len(df[df.WKW.notnull()])))

for year in ['d05', 'd06', 'd07']:
    print('Cleaning up the data for ' + year)
    recode_WKW(data[year])
    print('\n')
```

```
Cleaning up the data for d05
0.0    29957
1.0    20455
3.0     2382
6.0     2357
5.0     2030
4.0     1736
2.0     1595
Name: WKW, dtype: int64
Non-Nulls: 60512
Nulls: 0
Total: 60512


Cleaning up the data for d06
0.0    30218
1.0    21415
3.0     2466
6.0     2381
5.0     1995
4.0     1728
2.0     1687
Name: WKW, dtype: int64
Non-Nulls: 61890
Nulls: 0
Total: 61890


Cleaning up the data for d07
0.0    29681
1.0    21665
6.0     2377
3.0     2321
5.0     2042
4.0     1784
2.0     1730
Name: WKW, dtype: int64
Non-Nulls: 61600
```

Merging all the datasets into one

Now that our annual datasets are mostly consistent, it makes sense to merge them into one dataframe for ease of use. (I say 'mostly consistent' because the earlier datasets still don't have some data, such as NYCgov_MedSpending). In order to do that, we have to add a year column for each dataframe and then merge them together.

In [14]:
```
all_years = pd.DataFrame()
for name, df in data.items():
    df['Year'] = int('20' + (name[1:]))

all_years = pd.concat((df for df in data.values()), sort=True)
all_years.reset_index(drop=True, inplace=True)
len(all_years.index.unique())
```

Out[14]: 779254

Making columns consistent

In putting all the dataframes together, we discover a few issues:

- The 'Ol_adj' column is mis-titled 'Ol_Adj' in one year, and 'Ol_adj' in another.
- The 'PA_adj' column is mis-titled 'PA_Adj' in 2005.
- The 'NYCgov_Income' column is missing entirely from the data dictionary! No change needed in our code, but I had to add an entry in our COLS_LIMS_DICT to make sure I was checking it.

In [15]:
```
# Put all the OI_adj columns into one column
all_years.loc[all_years.Year == 2005, 'OI_adj'] = all_years.loc[all_years.Year == 2
all_years.loc[all_years.Year == 2006, 'OI_adj'] = all_years.loc[all_years.Year == 2

# Delete the mislabeled columns
del(all_years['Ol_Adj'])
del(all_years['Ol_adj'])

# Check that it worked
print(all_years[['Year', 'OI_adj']].groupby('Year').count())

# Put all the PA_adj columns into one column
all_years.loc[all_years.Year == 2005, 'PA_adj'] = all_years.loc[all_years.Year == 2
del(all_years['PA_Adj'])
print(all_years[['Year', 'PA_adj']].groupby('Year').count())
```

```
      OI_adj
Year
2005   60512
2006   61890
2007   61600
2008   61560
2009   63066
2010   65018
2011   66246
2012   66899
2013   66942
2014   67778
2015   69101
2016   68642
      PA_adj
Year
2005   60512
2006   61890
2007   61600
2008   61560
2009   63066
2010   65018
2011   66246
2012   66899
2013   66942
2014   67778
2015   69101
2016   68642
```

In [16]:
```
financial_columns = ['INTP_adj', 'MRGP_adj', 'OI_adj', 'PA_adj', 'PreTaxIncome_PU',
NYC_columns = ['NYCgov_Childcare', 'NYCgov_Commuting', 'NYCgov_EITC', 'NYCgov_FICAt
poverty_columns = ['NYCgov_PovGap', 'NYCgov_PovGapIndex', 'NYCgov_Pov_Stat', 'Off_P
all_years.columns
```

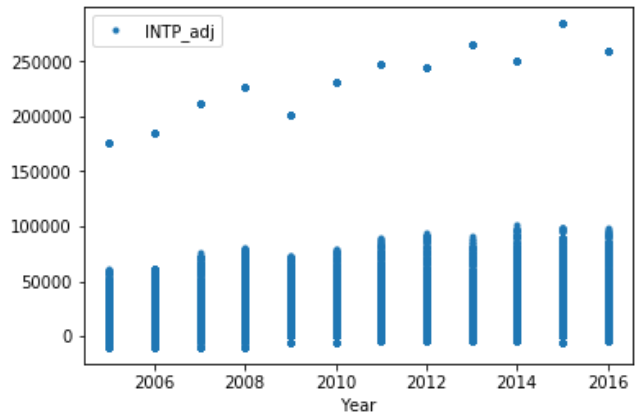
Out[16]:
```
Index(['AGEP', 'AgeCateg', 'Boro', 'CIT', 'CitizenStatus', 'DIS', 'ENG', 'ESR',
       'EducAttain', 'Ethnicity', 'FTPTWork', 'FamType_PU', 'HHT',
       'HIUnit_Head', 'HIUnit_ID', 'HousingStatus', 'INTP_adj', 'JWTR', 'LANX',
       'MAR', 'MRGP_adj', 'MSP', 'NP', 'NYCgov_Childcare', 'NYCgov_Commuting',
       'NYCgov_EITC', 'NYCgov_FICAtax', 'NYCgov_HEAP', 'NYCgov_Housing',
       'NYCgov_Income', 'NYCgov_IncomeTax', 'NYCgov_MOOP',
       'NYCgov_MedPremiums', 'NYCgov_MedSpending', 'NYCgov_Nutrition',
       'NYCgov_PovGap', 'NYCgov_PovGapIndex', 'NYCgov_Pov_Stat', 'NYCgov_REL',
       'NYCgov_SFN', 'NYCgov_SFR', 'NYCgov_SNAP', 'NYCgov_SchoolBreakfast',
       'NYCgov_SchoolLunch', 'NYCgov_Threshold', 'NYCgov_WIC', 'OI_adj',
       'Off_Pov_Stat', 'Off_Threshold', 'PA_adj', 'PWGTP', 'Povunit_ID',
       'Povunit_Rel', 'PreTaxIncome_PU', 'REL', 'RETP_adj', 'RNTP_adj', 'SCH',
       'SCHG', 'SCHL', 'SEMP_adj', 'SERIALNO', 'SEX', 'SNAPUnit_ID',
       'SNAPUnit_Rel', 'SPORDER', 'SSIP_adj', 'SSP_adj', 'TEN',
       'TaxUnit_FILER', 'TaxUnit_FILESTAT', 'TaxUnit_FILETYPE', 'TaxUnit_ID',
       'TaxUnit_Rel', 'TotalWorkHrs_PU', 'WAGP_adj', 'WGTP', 'WKHP', 'WKW',
       'Year'],
      dtype='object')
```

Checking for outliers, part 2

Now that we have the columns named appropriately and we've checked the columns for outliers that are outside the allowable values, let's take a look at the columns visually. Since we already checked the classification columns (e.g. whether or not the person speaks English), I'm really interested in the columns that contain an actual number rather than a classification coded as a number.

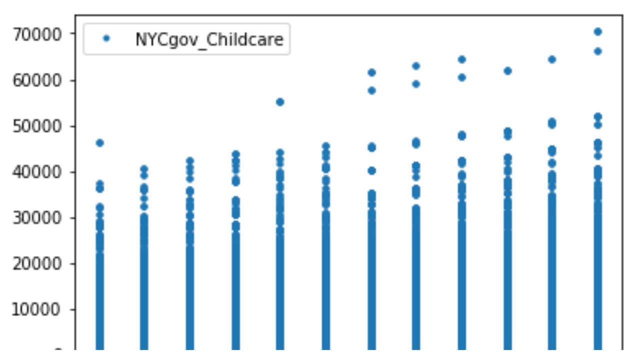
I'll create a couple of lists - one of the financial columns, one containing the columns starting with 'NYCgov_', and one with the poverty indicators. We can look at charts by each. (I've commented out each of the three lines below that actually create the graphs, since each of those lines takes 30 seconds or so to run.)

In [17]:
```
financial_columns = ['INTP_adj', 'MRGP_adj', 'OI_adj', 'PA_adj', 'PreTaxIncome_PU',
NYC_columns = ['NYCgov_Childcare', 'NYCgov_Commuting', 'NYCgov_EITC', 'NYCgov_FICAt
poverty_columns = ['NYCgov_PovGap', 'NYCgov_PovGapIndex', 'NYCgov_Pov_Stat', 'Off_P

def quick_charts_by_year(df, columns):
    '''Displays a series of quick charts by year.
    Input: a dataframe and a list of columns in the dataframe.
    Output: no return value. Prints charts of each column in columns, each chart gr

    for column in columns:
        chart = df[['Year',column]]
        chart.plot(kind='line', x='Year',  y=column, linestyle="none", marker='.')

quick_charts_by_year(all_years, financial_columns)
```

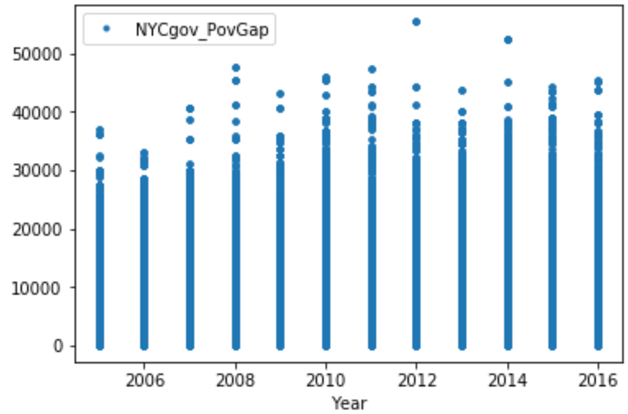


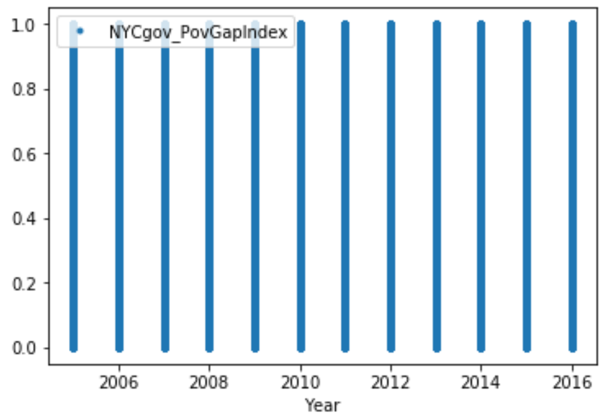
In [18]:
```
quick_charts_by_year(all_years, NYC_columns)
```

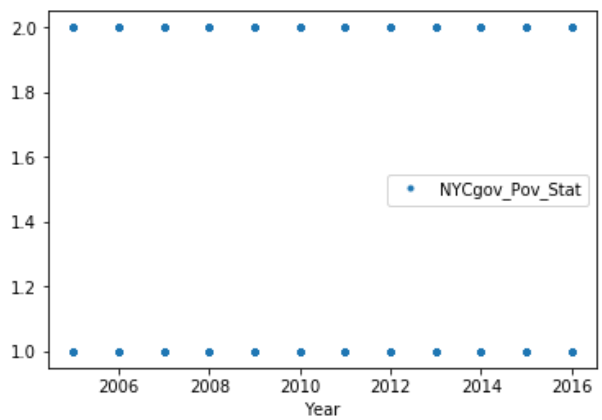
```
/home/chachi/miniconda3/envs/pandas-tutorial/lib/python3.7/site-packages/matplot
lib/pyplot.py:513: RuntimeWarning: More than 20 figures have been opened. Figure
s created through the pyplot interface (`matplotlib.pyplot.figure`) are retained
until explicitly closed and may consume too much memory. (To control this warnin
g, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```

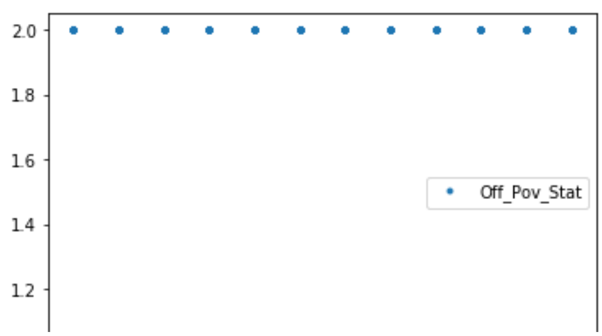


In [19]:
```
quick_charts_by_year(all_years, poverty_columns)
```









Fixing the outliers in NYCgov_Income

The financial columns look mostly fine. There are definitely outliers, but checking against the data dictionary for their meaning, nothing seems amiss.

The charts by poverty measure also indicate no worrisome outliers.

Same with *most* of the NYC columns, except for NYCgov_Income; suddenly in 2016, there were a bunch of well-below-zero values. After investigating, the issue is 23 rows where the PreTaxIncome_PU was multiple hundreds of thousands of dollars, the NYCgov_IncomeTax was multiple hundreds of thousands of dollars, and the resulting NYCgov_Income was less than -50,000. In all other years, only 8 rows had NYCgov_Income of less than $50,000.

As a result, below I'll update the dataframe to only include rows with NYCgov_Income of more than -50,000 (negative fifty-thousand dollars).

In [20]:
```
all_years.drop(all_years[all_years.NYCgov_Income < -50000].index, inplace=True)
print(len(all_years))
check_df_limits(all_years, COLS_LIMS_DICT)
```

```
779223
DIS has stated min of 1 but actual min of 0.00.
DS is not in dataframe.
INTP_adj has stated min of -9,999 but actual min of -10,190.88.
SEMP_adj has stated min of -10,000 but actual min of -10,190.88.
PreTaxIncome_PU has stated min of -10,000 but actual min of -20,311.47.
NYCgov_EITC has stated min of 0 but actual min of -2,586.13.
NYCgov_MOOP has stated max of 99,999 but actual max of 116,359.00.
NYCgov_Income has stated min of -25,000 but actual min of -49,869.92.
NYCgov_Income has stated max of 999,999 but actual max of 1,662,673.94.
NYCgov_IncomeTax has stated max of 99,999 but actual max of 1,341,696.07.
NYCgov_PovGap has stated max of 50,000 but actual max of 55,405.07.
```

In [21]:
```
all_years.to_csv('all_years.csv')
```