05_visualization

November 23, 2022

1 Visualizing Data

1.1 Introduction to Python

Data Sciences Institute, University of Toronto

Instructor: Kaylie Lau | TA: Salaar Liaqat

November - December 2022

2 Contents:

1. Setup
2. `matplotlib`
3. `seaborn`
4. `plotly`

2.1 Data

The specific file names are: - bike_thefts_joined.csv - neighbourhoods.csv

2.2 Supporting packages and data

Let's import `numpy` and `pandas` and load up some data to work with.

[ ]:
```
import numpy as np
import pandas as pd
```

[ ]:
```
# load data
thefts_joined = pd.read_csv('/content/data/bike_thefts_joined.csv',
                            dtype={'n_id': str})
neighbourhoods = pd.read_csv('/content/data/neighbourhoods.csv',
                             dtype={'n_id': str})

# fix dates
```

```python
thefts_joined['occurrence_date'] = pd.
 ↪to_datetime(thefts_joined['occurrence_date'])
thefts_joined['report_date'] = pd.to_datetime(thefts_joined['report_date'])
```

```
     ␣
↪---------------------------------------------------------------------------

       FileNotFoundError                         Traceback (most recent call␣
↪last)

       <ipython-input-3-b6bb675e370f> in <module>
         1 # load data
         2 thefts_joined = pd.read_csv('/content/data/bike_thefts_joined.csv',
   ----> 3                            dtype={'n_id': str})
         4 neighbourhoods = pd.read_csv('/content/data/neighbourhoods.csv',
         5                             dtype={'n_id': str})


       /usr/local/lib/python3.7/dist-packages/pandas/util/_decorators.py in␣
↪wrapper(*args, **kwargs)
       309                     stacklevel=stacklevel,
       310                 )
   --> 311             return func(*args, **kwargs)
       312
       313         return wrapper


       /usr/local/lib/python3.7/dist-packages/pandas/io/parsers/readers.py in␣
↪read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col,␣
↪usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine, converters,␣
↪true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows,␣
↪na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates,␣
↪infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_dates,␣
↪iterator, chunksize, compression, thousands, decimal, lineterminator,␣
↪quotechar, quoting, doublequote, escapechar, comment, encoding,␣
↪encoding_errors, dialect, error_bad_lines, warn_bad_lines, on_bad_lines,␣
↪delim_whitespace, low_memory, memory_map, float_precision, storage_options)
       584     kwds.update(kwds_defaults)
       585
   --> 586     return _read(filepath_or_buffer, kwds)
       587
       588


       /usr/local/lib/python3.7/dist-packages/pandas/io/parsers/readers.py in␣
↪_read(filepath_or_buffer, kwds)
```

```
    480
    481     # Create the parser.
--> 482     parser = TextFileReader(filepath_or_buffer, **kwds)
    483
    484     if chunksize or iterator:


    /usr/local/lib/python3.7/dist-packages/pandas/io/parsers/readers.py in␣
↪__init__(self, f, engine, **kwds)
    809             self.options["has_index_names"] = kwds["has_index_names"]
    810
--> 811         self._engine = self._make_engine(self.engine)
    812
    813     def close(self):


    /usr/local/lib/python3.7/dist-packages/pandas/io/parsers/readers.py in␣
↪_make_engine(self, engine)
   1038             )
   1039         # error: Too many arguments for "ParserBase"
-> 1040         return mapping[engine](self.f, **self.options)  # type:␣
↪ignore[call-arg]
   1041
   1042     def _failover_to_python(self):


    /usr/local/lib/python3.7/dist-packages/pandas/io/parsers/
↪c_parser_wrapper.py in __init__(self, src, **kwds)
     49
     50         # open handles
---> 51         self._open_handles(src, kwds)
     52         assert self.handles is not None
     53


    /usr/local/lib/python3.7/dist-packages/pandas/io/parsers/base_parser.py␣
↪in _open_handles(self, src, kwds)
    227             memory_map=kwds.get("memory_map", False),
    228             storage_options=kwds.get("storage_options", None),
--> 229             errors=kwds.get("encoding_errors", "strict"),
    230         )
    231


    /usr/local/lib/python3.7/dist-packages/pandas/io/common.py in␣
↪get_handle(path_or_buf, mode, encoding, compression, memory_map, is_text,␣
↪errors, storage_options)
```

```
      705                     encoding=ioargs.encoding,
      706                     errors=errors,
  --> 707                     newline="",
      708                 )
      709             else:


      FileNotFoundError: [Errno 2] No such file or directory: '/content/data/
↪bike_thefts_joined.csv'
```

[ ]:
```
thefts_joined.head()
```

```
            ␣
↪---------------------------------------------------------------------------

      NameError                                 Traceback (most recent call␣
↪last)

      <ipython-input-1-cd2d292bf923> in <module>
  ----> 1 thefts_joined.head()


      NameError: name 'thefts_joined' is not defined
```

[ ]:
```
# exclude the City of Toronto
neighbourhoods = neighbourhoods.loc[neighbourhoods['neighbourhood'] != 'City of␣
↪Toronto']
neighbourhoods.head()
```

[ ]:
```
# add new columns showing % of commuters for each mode
def calc_pct(mode):
    return round(mode/neighbourhoods['total_commuters'], 3)

# new column names
pct_cols = ['pct_drive', 'pct_cp', 'pct_transit', 'pct_walk']
neighbourhoods[pct_cols] = neighbourhoods.loc[:, 'drive':'walk'].apply(calc_pct)
```

3 Overview

3.1 Data visualization in Python

So far, we have gotten data, wrangled it, and scratched the surface of exploratory analyses. As part of that exploration, we created charts with `pandas`. However, there are dedicated visualization libraries let us customize our charts further.

4 matplotlib

4.1 matplotlib

`matplotlib` is *the* foundational data visualization library in Python. `pandas`'s visualization functions are, at their core, `matplotlib` functions. Other popular libraries like `seaborn` similarly build on `matplotlib`.

For historical reasons, when we import `matplotlib`, we really import `matplotlib.pyplot`. The conventional alias is `plt`.

[ ]:
```
# jupyter-specific "magic" command to render plots in-line
%matplotlib inline

import matplotlib.pyplot as plt
```

4.2 Anatomy of a plot

`matplotlib` visuals consist of one or more Axes in a Figure. An *Axes*, confusingly, is what we would consider a graph, while the *Figure* is a container for those graphs. An Axes has an x-*Axis* and a y-*Axis*.

More details can be found at: https://matplotlib.org/stable/tutorials/introductory/quick_start.html

4.3 Plotting with matplotlib

`matplotlib` provides two ways to create visualizations: * by having **pyplot** automatically create and manage Figures and Axes, keeping track of which Figure and Axes we are currently working on * by taking an **object-oriented approach**, where we explicitly create Figures and Axes and modify them

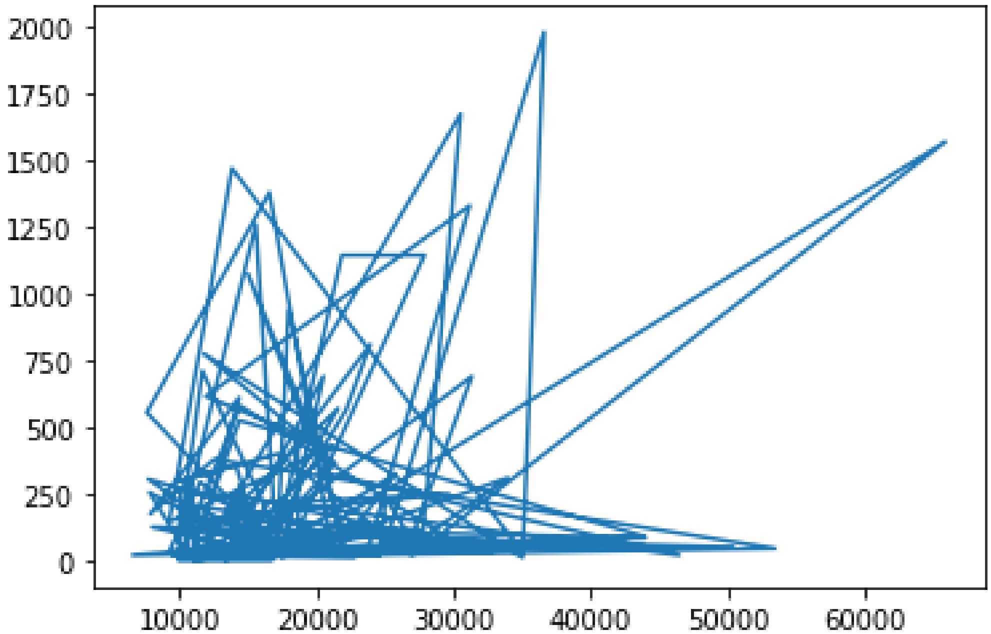
The object-oriented approach is recommended, but the `pyplot` approach is convenient for quick plots.

4.4 pyplot-style plotting

`pyplot`-style plotting is convenient for quick, exploratory plots, where we don't plan on doing a lot of customization. When we plotted data in `pandas`, `pandas` took this approach. Let's plot the neighbourhood data with the `pyplot` approach. `plot()` produces a line plot by default.

[ ]:
```
plt.plot(neighbourhoods['pop_2016'],
         neighbourhoods['bike'])
```

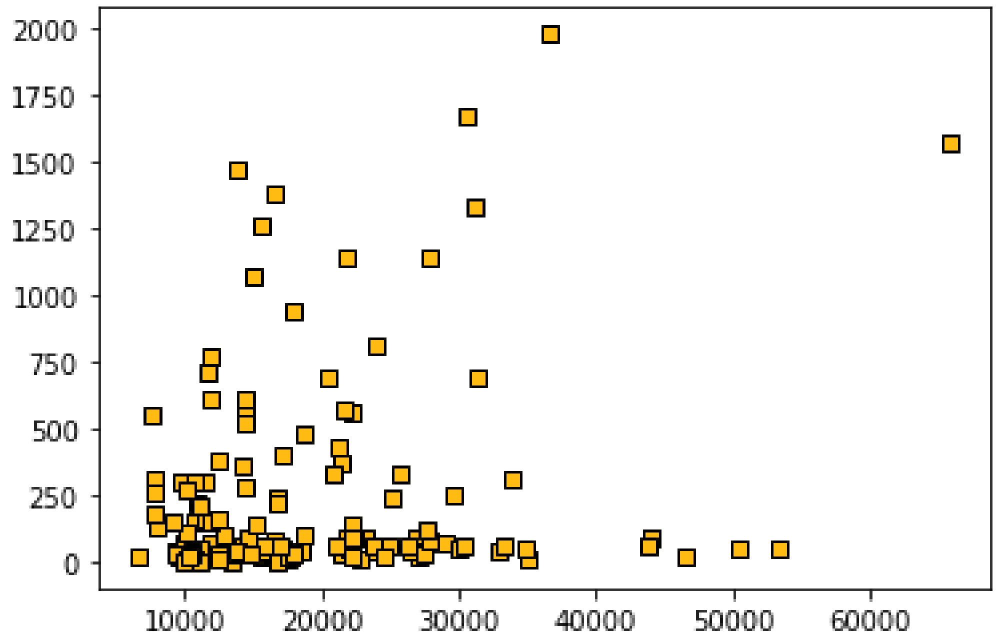
[ ]: [<matplotlib.lines.Line2D at 0x7f88008147d0>]



Let's make it a scatterplot instead with the `scatter()` function. We can use keyword arguments like `facecolor` and `edgecolor` to change the styling. `matplotlib` lets us specify colour with RGB(A) tuples, hexadecimal strings, single-character shortcodes, and even xkcd colours.

[ ]:
```
plt.scatter(neighbourhoods['pop_2016'],
            neighbourhoods['bike'],
            marker='s',  # square marker
            facecolor='#fb1',
            edgecolor='k') # black
```

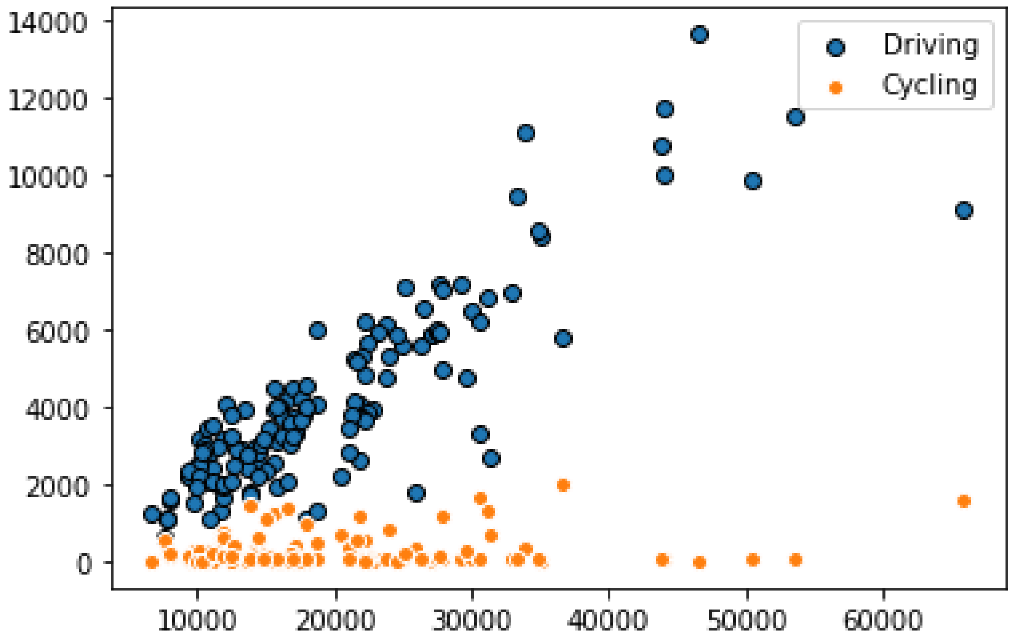
[ ]: <matplotlib.collections.PathCollection at 0x7f88000a0250>



Using the `pyplot` approach, the outputs of successive function calls in the same cell context are layered on. Let's layer driving and biking commuter counts and add a legend.

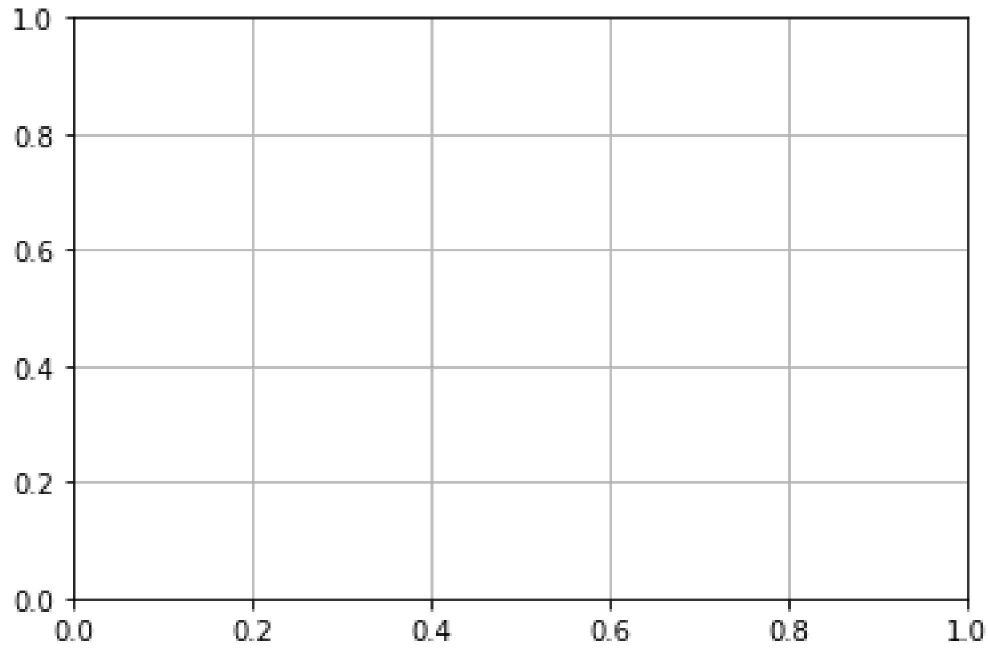
[ ]:
```
plt.scatter(neighbourhoods['pop_2016'],
            neighbourhoods['drive'],
            edgecolor='k',
            label='Driving')
plt.scatter(neighbourhoods['pop_2016'],
            neighbourhoods['bike'],
            edgecolor='w',
            label='Cycling')
plt.legend()
```

[ ]: <matplotlib.legend.Legend at 0x7f8800032850>



Calls in a different cell are treated as a new Axes.

[ ]:
```
plt.grid()
```



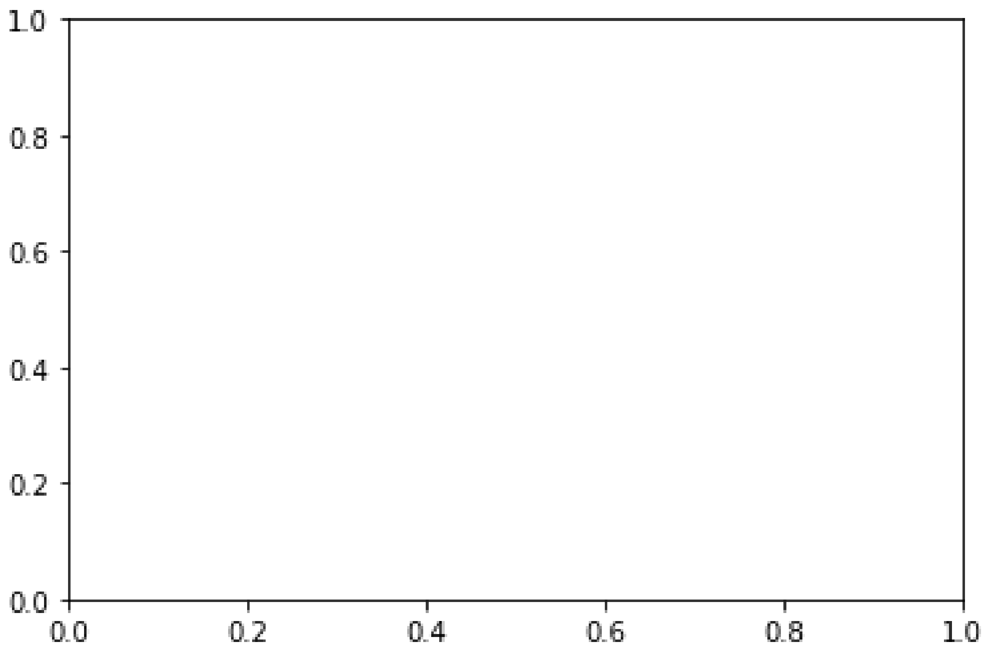
4.5 Object-oriented approach to plotting

The object-oriented approach is the preferred method of plotting with `matplotlib`. In this approach, we use the `subplots()` function to create plot objects, then call methods to modify them.

By default, `subplots()` returns one Figure and one Axes. We can use Python's unpacking syntax to assign the Figure and Axes to their own variables in one line.

[ ]:
```python
fig, ax = plt.subplots()
print(f'{type(fig)}, {type(ax)}')
```

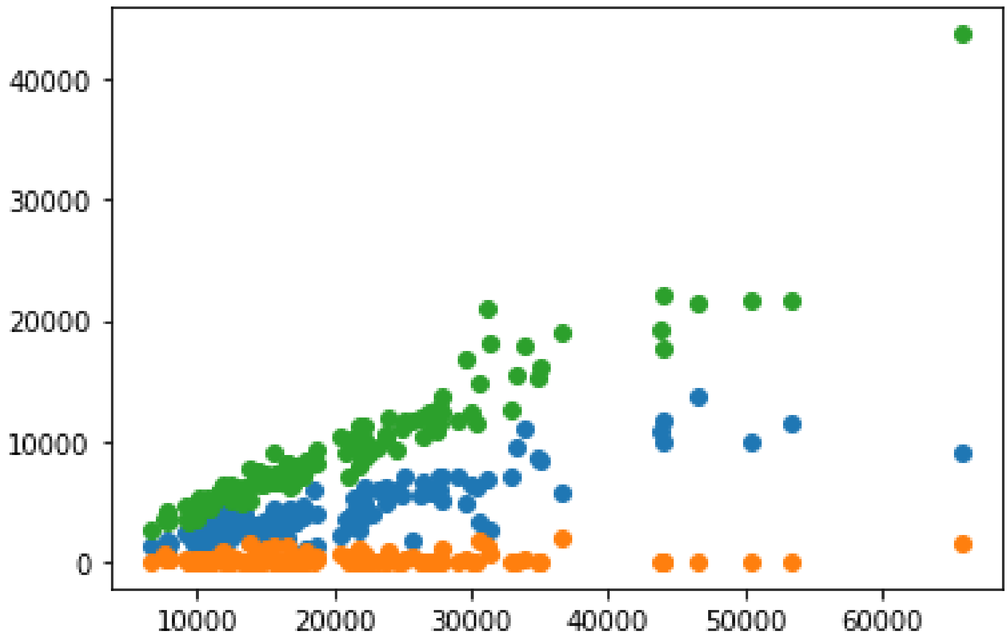
```
<class 'matplotlib.figure.Figure'>, <class
'matplotlib.axes._subplots.AxesSubplot'>
```



The Axes is empty. Let's plot data on it with the Axes `scatter()` method. This method updates `ax` with a scatterplot. To make it easier to refer to each scatterplot later, we assign the outputs to their own variables, `drivers` and `cyclists`.

[ ]:
```python
drivers = ax.scatter(neighbourhoods['pop_2016'],
           neighbourhoods['drive'])
cyclists = ax.scatter(neighbourhoods['pop_2016'],
           neighbourhoods['bike'])
total = ax.scatter(neighbourhoods['pop_2016'],
                  neighbourhoods['total_commuters'])
fig
```

[ ]:

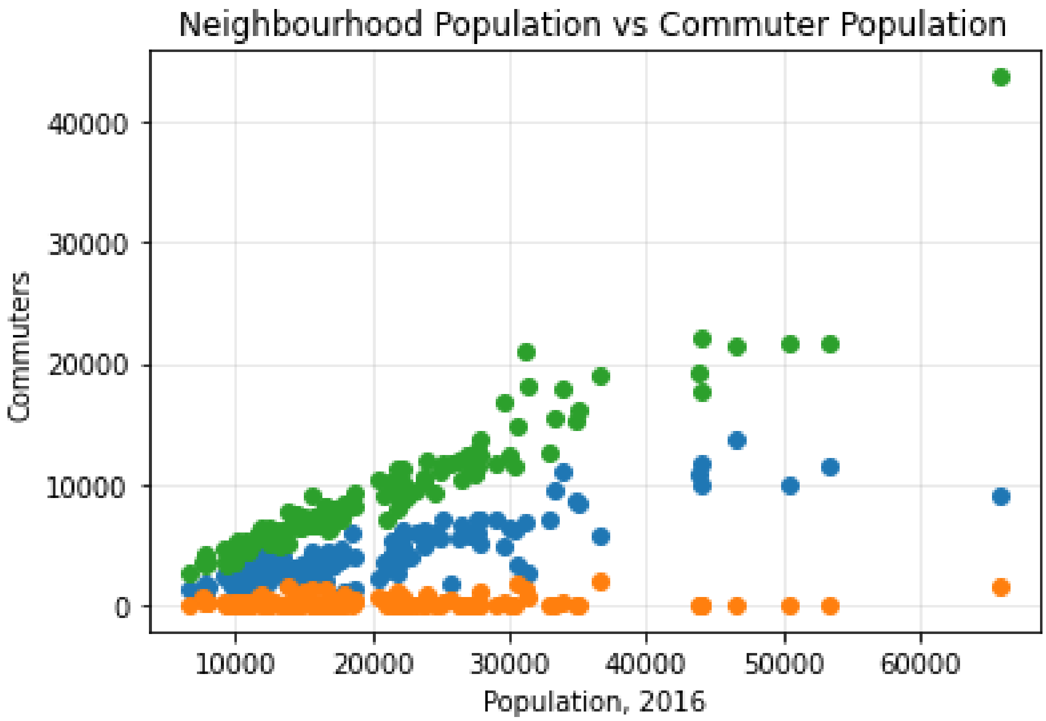


4.6 Adding labels, a title, and grid

This graph doesn't give much context. To add a title, we can use the Axes set_title() method, which takes the title as a string, plus optional arguments like fontsize. Similarly, we can set x and y labels with the set_xlabel() and set_ylabel() methods. Finally, let's add a grid with the Axes grid() method, and use the alpha parameter to make it translucent. We'll also use the set_axisbelow() method to make sure markers draw over the grid.

[ ]:
```
ax.set_title('Neighbourhood Population vs Commuter Population')
ax.set_xlabel('Population, 2016')
ax.set_ylabel('Commuters')
ax.set_axisbelow(True)
ax.grid(alpha=0.3)
fig
```

[ ]:

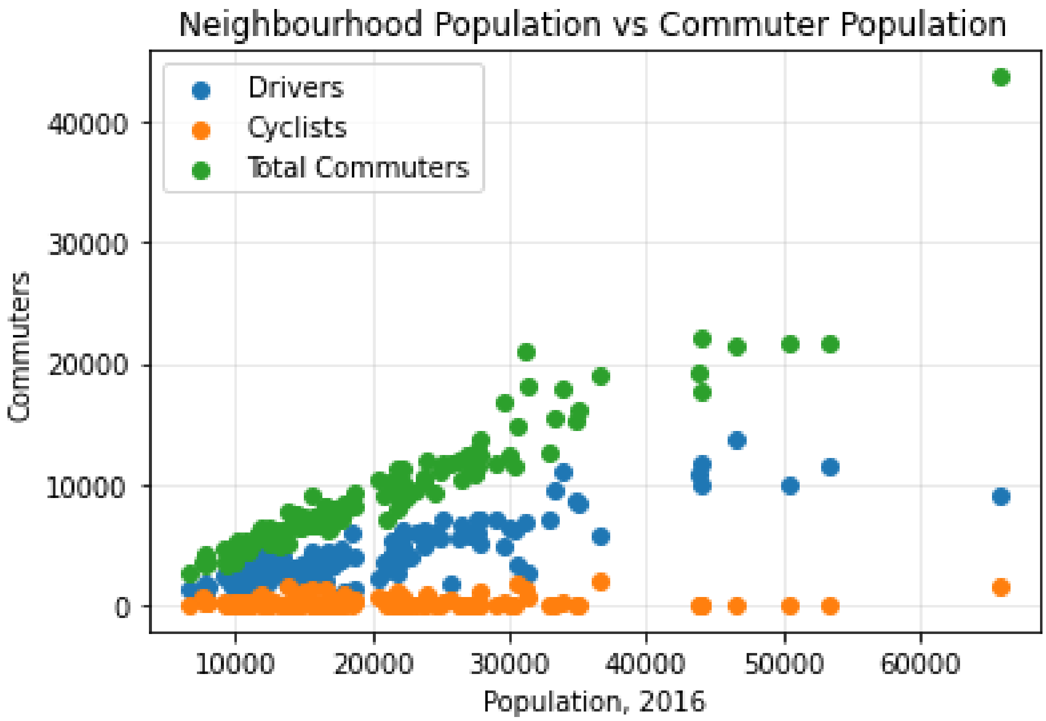


4.7 Adding a legend

This graph could use a legend. To add one, we call the Axes legend() method. If we passed a label argument in the scatter() calls, legend() would use those labels. However, because we did not, we pass a list of the geometries to use in the legend, plus a list of labels to show.

[ ]:
```
ax.legend([drivers, cyclists, total],
          ['Drivers', 'Cyclists', 'Total Commuters'])
fig
```

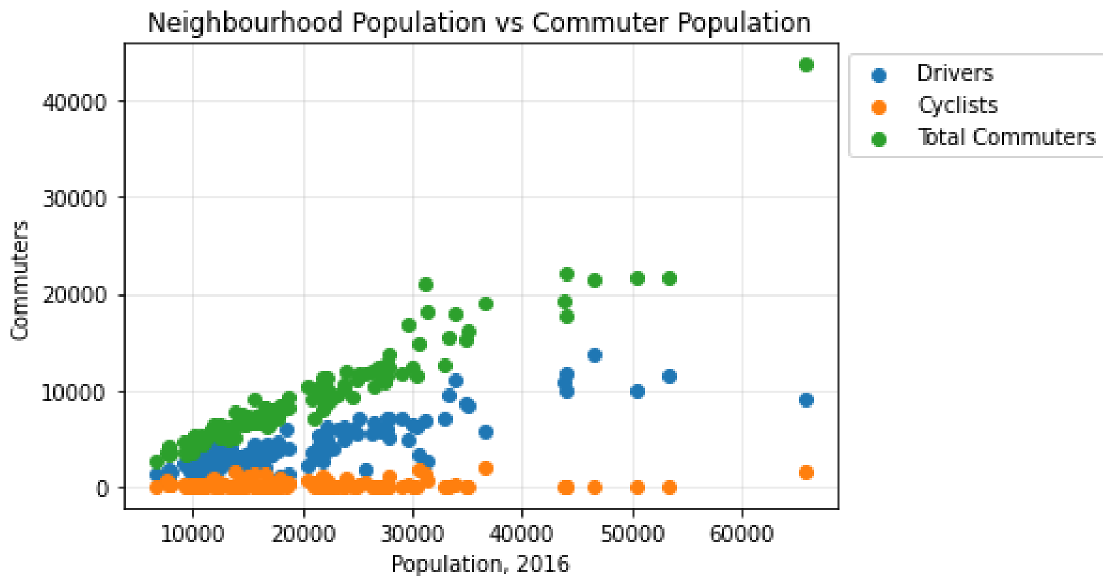
[ ]:



To place the legend outside the Axes, we can pass a tuple with the bbox_to_anchor argument. The legend's loc corner will be placed at the coordinates in the bbox_to_anchor tuple.

[ ]:
```
ax.legend([drivers, cyclists, total],
          ['Drivers', 'Cyclists', 'Total Commuters'],
          bbox_to_anchor=(1, 1),
          loc='upper left')
fig
```

[ ]:



4.8 Modifying axis ticks

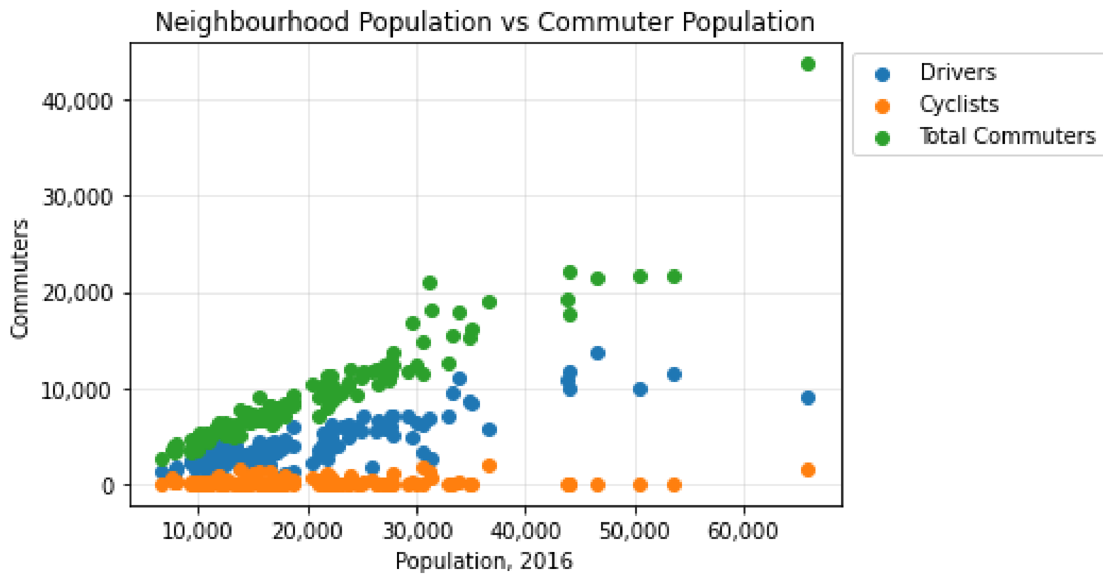
We can change how the x-axis and y-axis are formatted by accessing an Axes xaxis and yaxis attributes and calling methods like set_ticks() or set_major_formatter().

Some configurations of Python and matplotlib allow us to pass a format string by itself to set_major_formatter(). Older versions require that we import matplotlib's ticker submodule and create a StrMethodFormatter with the format string we want to use.

[ ]:
```
import matplotlib.ticker as tick
```

[ ]:
```
# label with a thousands place comma and zero decimal places
ax.xaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
ax.yaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
fig
```

[ ]:



We can also change axis limits.

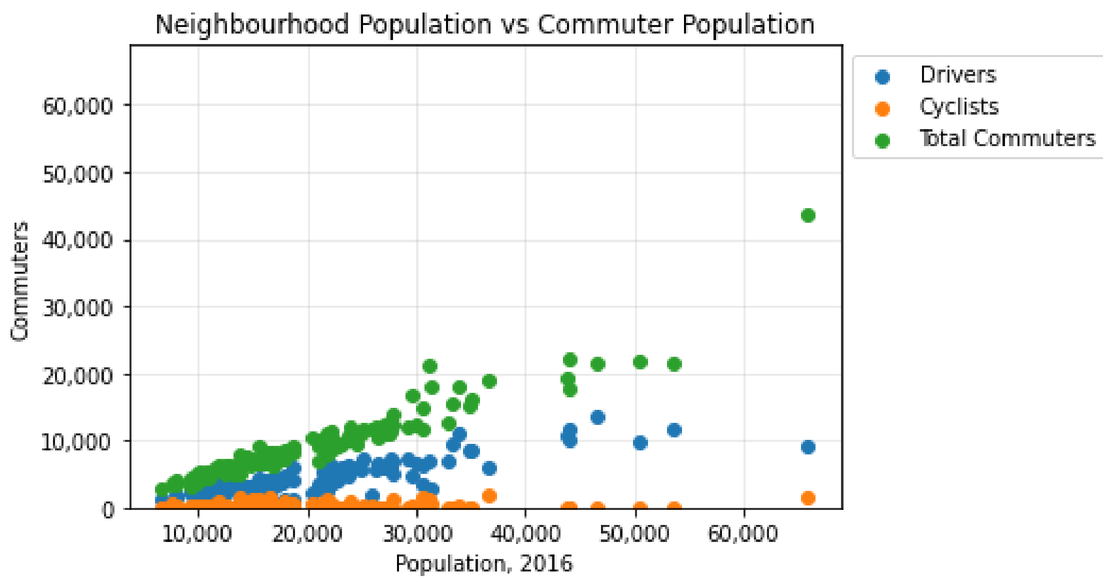
[ ]:
```
#ax.xaxis.set_ticks(np.arange(0, max(neighbourhoods['pop_2016']+10), 10000))
```

[ ]:
```
ax.axis()
```

[ ]: (3610.2, 68879.8, -2189.25, 45974.25)

[ ]:
```
ax.set(ylim=(0, ax.axis()[1])) # make the y-axis match the x-axis
fig
```

[ ]:



4.9 Changing styles

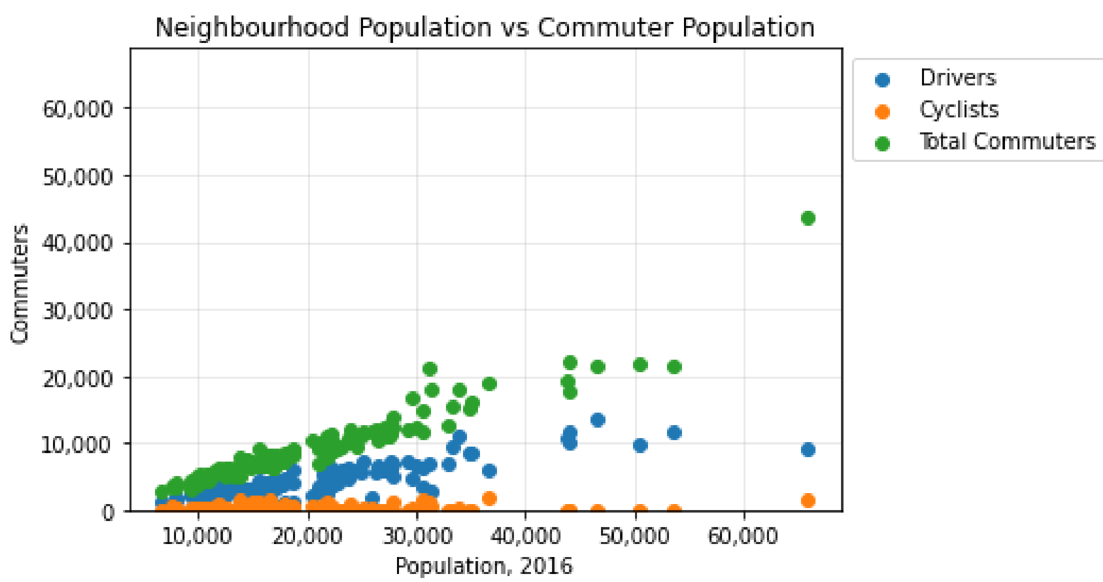
matplotlib comes with a bunch of predefined styles. We can view the available ones with plt.style.available. Passing one of the options to style.use() makes it the aesthetic style for all new plots. **Already created Figures and Axes are not affected.**

```
[ ]: plt.style.available[5:10]  # print a subset
```

```
[ ]: ['fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn']
```

```
[ ]: # set style for new plots
plt.style.use('fivethirtyeight')

# notice that the style of fig did not change
fig
```

[ ]:



4.10 Other plot types

Of course, matplotlib offers more than just line plots and scatterplots. Among the many kinds of plots we can make are bar plots, histograms, and boxplots. To create each the object-oriented way, we call the appropriate Axes method, like Axes.boxplot() or Axes.barh(), for a horizontal bar plot.

[ ]:
```
# review the neighbourhoods data
neighbourhoods.head()
```

[ ]:
```
                  neighbourhood n_id     designation  pop_2016  pop_2011  \
0               Agincourt North  129  No Designation     29113     30279
1  Agincourt South-Malvern West  128  No Designation     23757     21988
2                     Alderwood   20  No Designation     12054     11904
3                         Annex   95  No Designation     30526     29177
4             Banbury-Don Mills   42  No Designation     27695     26918

   pop_change  private_dwellings  occupied_dwllings  pop_dens  area  …  \
0      -0.039               9371               9120      3929  7.41  …
1       0.080               8535               8136      3034  7.83  …
2       0.013               4732               4616      2435  4.95  …
3       0.046              18109              15934     10863  2.81  …
4       0.029              12473              12124      2775  9.98  …

   car_passenger  transit  walk  bike  other  pct_bike  pct_drive  pct_cp  \
0            930     3350   265    70     45     0.006      0.605   0.079
1            665     2985   280    35     65     0.003      0.604   0.065
2            355     1285   195    65     65     0.011      0.677   0.059
3            290     6200  3200  1675    225     0.112      0.221   0.019
4            500     2945   615    65    140     0.006      0.627   0.044

   pct_transit  pct_walk
0        0.283     0.022
1        0.294     0.028
2        0.213     0.032
3        0.416     0.215
4        0.258     0.054

[5 rows x 22 columns]
```

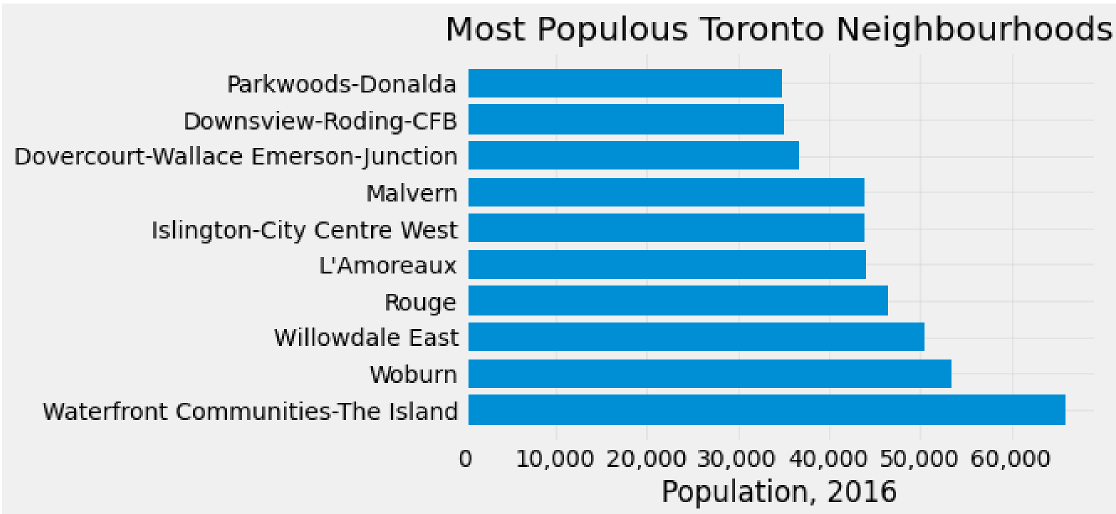
[ ]:
```
# get just the 10 biggest neighbourhoods to plot
top10_pop = neighbourhoods.sort_values('pop_2016', ascending=False).head(10)
top10_pop
```

[ ]:
```
                          neighbourhood n_id             designation  \
123    Waterfront Communities-The Island   77          No Designation
133                              Woburn  137                     NIA
130                      Willowdale East   51          No Designation
106                                Rouge  131          No Designation
66                           L'Amoreaux  117  Emerging Neighbourhood
59             Islington-City Centre West   14          No Designation
74                              Malvern  132  Emerging Neighbourhood
33   Dovercourt-Wallace Emerson-Junction   93          No Designation
34                   Downsview-Roding-CFB   26                     NIA
```

```
96                    Parkwoods-Donalda   45          No Designation

     pop_2016  pop_2011  pop_change  private_dwellings  occupied_dwllings  \
123     65913     43361       0.520              47209              40756
133     53485     53350       0.003              19098              18436
130     50434     45041       0.120              23901              22304
106     46496     45912       0.013              13730              13389
66      43993     44919      -0.021              15486              15037
59      43965     38084       0.154              19911              19328
74      43794     45086      -0.029              13936              13426
33      36625     34631       0.058              16248              15320
34      35052     34659       0.011              14244              13121
96      34805     34617       0.005              13921              13315

     pop_dens   area  …  car_passenger  transit   walk  bike  other  \
123      8943   7.37  …            760    10915  20855  1570    610
133      4345  12.31  …           1405     7635    780    45    210
130     10087   5.00  …            695     9390   1550    50    215
106      1260  36.89  …           1510     5935    220    20    160
66       6144   7.16  …           1220     5895    370    85    120
59       2712  16.21  …            975     8205    795    90    195
74       4948   8.85  …           1400     6425    425    60    115
33       9819   3.73  …            820     8950   1215  1980    310
34       2337  15.00  …           1060     6085    460    10    145
96       4691   7.42  …            820     5275    420    45    115

     pct_bike  pct_drive  pct_cp  pct_transit  pct_walk
123     0.036      0.208   0.017        0.249     0.476
133     0.002      0.533   0.065        0.354     0.036
130     0.002      0.454   0.032        0.431     0.071
106     0.001      0.636   0.070        0.276     0.010
66      0.005      0.565   0.069        0.333     0.021
59      0.004      0.534   0.044        0.372     0.036
74      0.003      0.561   0.073        0.334     0.022
33      0.104      0.305   0.043        0.469     0.064
34      0.001      0.520   0.066        0.376     0.028
96      0.003      0.562   0.054        0.345     0.028

[10 rows x 22 columns]
```

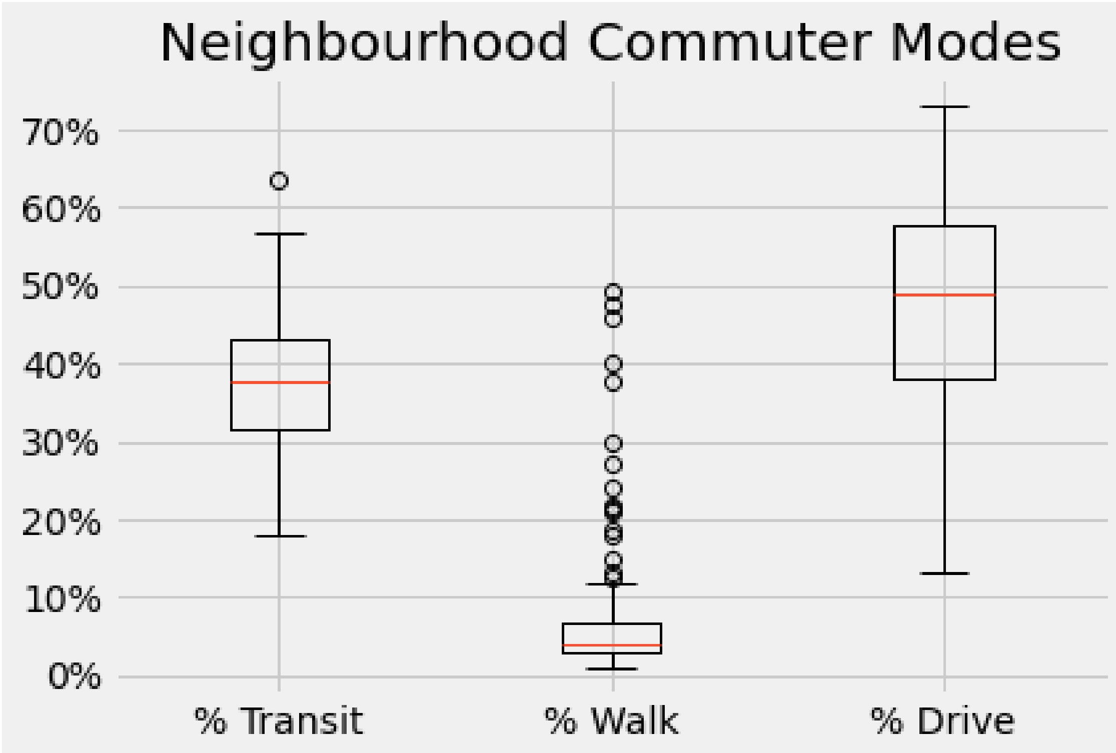
[ ]:
```
bar_fig, bar_ax = plt.subplots()
bar_ax.barh(top10_pop['neighbourhood'], top10_pop['pop_2016'])
bar_ax.xaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
bar_ax.set_axisbelow(True)
bar_ax.grid(alpha=0.3)
bar_ax.set_title('Most Populous Toronto Neighbourhoods')
bar_ax.set_xlabel('Population, 2016')
```

[ ]: Text(0.5, 0, 'Population, 2016')



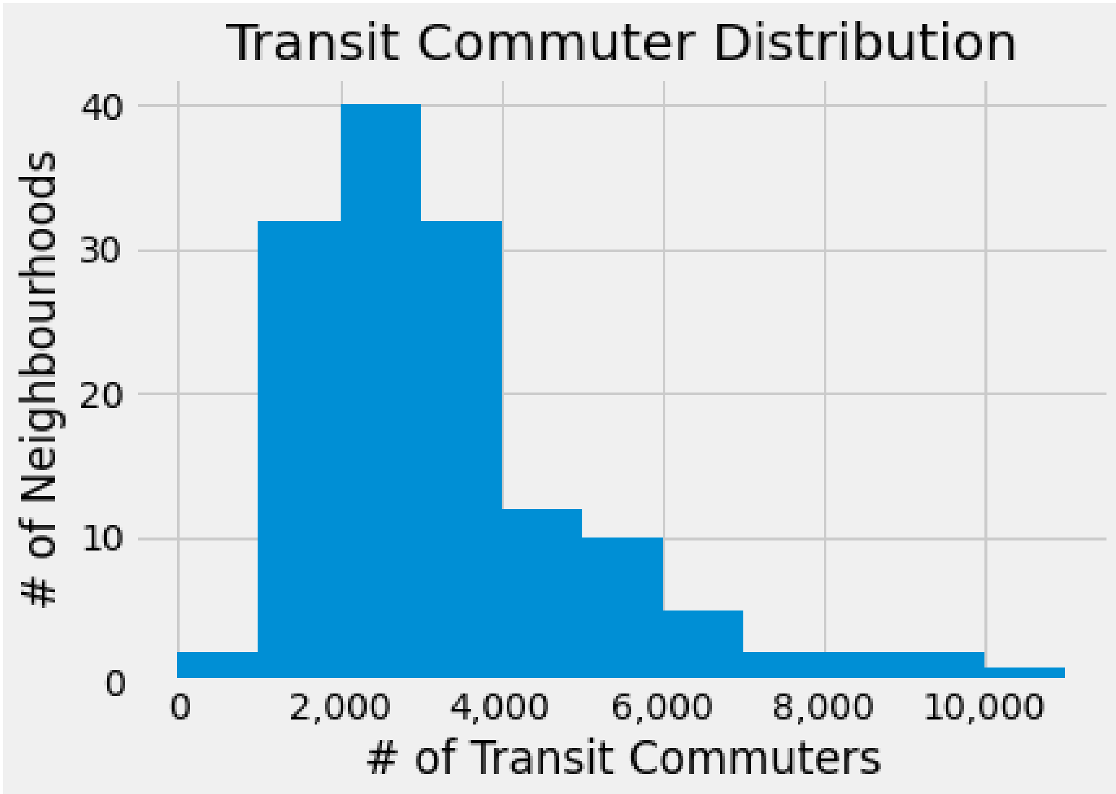
[ ]:
```
# create a box plot
box_fig, box_ax = plt.subplots()
box_ax.boxplot([neighbourhoods['pct_transit'],
                neighbourhoods['pct_walk'],
                neighbourhoods['pct_drive']],
               # add labels so we know which box is which var
               labels=['% Transit', '% Walk', '% Drive'])
box_ax.yaxis.set_major_formatter(tick.StrMethodFormatter('{x:.0%}'))
box_ax.set_title('Neighbourhood Commuter Modes')
```

[ ]: Text(0.5, 1.0, 'Neighbourhood Commuter Modes')



[ ]:
```
# create a histogram
hist_fig, hist_ax = plt.subplots()
hist_ax.hist(neighbourhoods['transit'],
             # count the neighbourhoods with 0-1000 transit commuters,
             # 1001-2000 transit commuters, etc
             bins=range(0, 12000, 1000))
hist_ax.xaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
hist_ax.set_title('Transit Commuter Distribution')
hist_ax.set_xlabel('# of Transit Commuters')
hist_ax.set_ylabel('# of Neighbourhoods')
```

[ ]:
```
Text(0, 0.5, '# of Neighbourhoods')
```

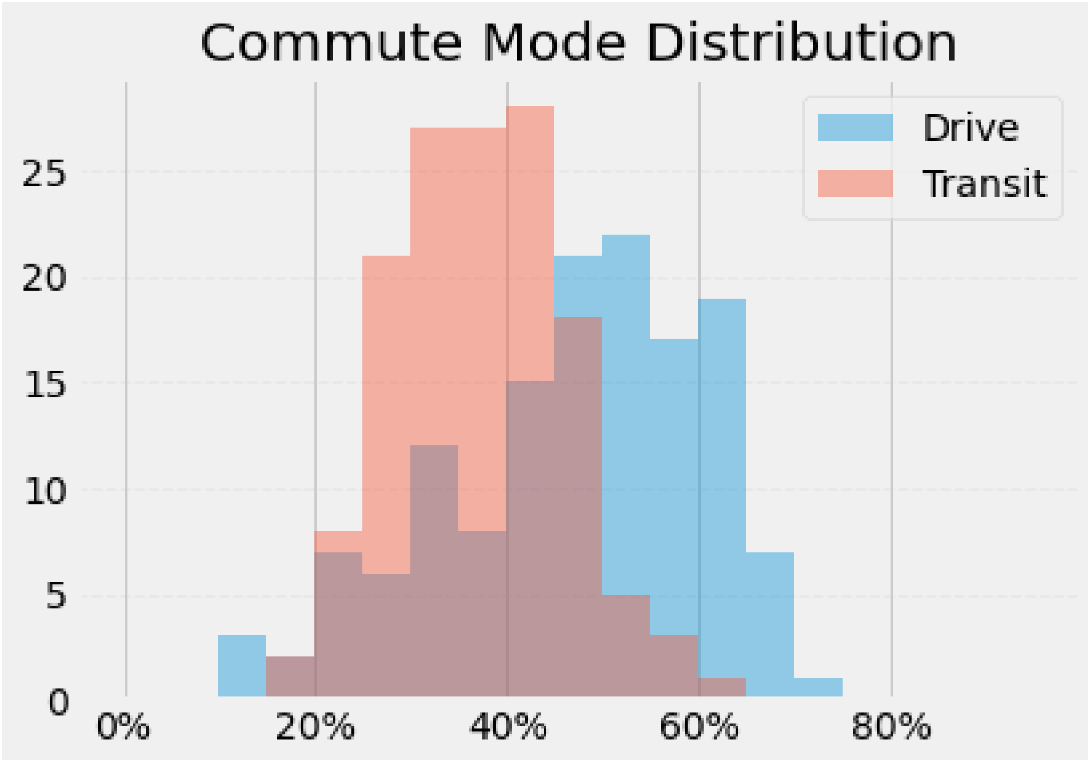


4.11 Layering plots

We've seen that a single Axes can have more than one set of data points plotted on it with our multi-modal scatterplot. We can similarly layer on other graphics, using the alpha argument to set transparency.

[ ]:
```
layer_fig, layer_ax = plt.subplots()

settings = {'alpha': 0.4, 'bins': np.arange(0, 1, .05)}

layer_ax.hist(neighbourhoods['pct_drive'], label='Drive', **settings)
layer_ax.hist(neighbourhoods['pct_transit'], label='Transit', **settings)
layer_ax.xaxis.set_major_formatter(tick.StrMethodFormatter('{x:.0%}'))
layer_ax.set_axisbelow(True)
layer_ax.grid(alpha=0.2, linestyle='--', axis='y')
layer_ax.set_title('Commute Mode Distribution')
layer_ax.legend()
layer_ax
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8800814bd0>
```



4.12 More complex plots

Let's try plotting the number of reported bike thefts each year by whether the bike was recovered or not. We'll need to wrangle the theft data a bit to get counts by year and status. Then, we'll use the data to make a stackplot(). Finally, we'll style it.

[ ]:
```
# review the available columns
thefts_joined.columns
```

[ ]:
```
Index(['_id', 'objectid', 'event_unique_id', 'primary_offence',
        'occurrence_date', 'occurrence_year', 'occurrence_month',
        'occurrence_dayofweek', 'occurrence_dayofmonth', 'occurrence_dayofyear',
        'occurrence_hour', 'report_date', 'report_year', 'report_month',
        'report_dayofweek', 'report_dayofmonth', 'report_dayofyear',
        'report_hour', 'division', 'city', 'hood_id', 'neighbourhoodname',
        'location_type', 'premises_type', 'bike_make', 'bike_model',
        'bike_type', 'bike_speed', 'bike_colour', 'bike_cost', 'status',
        'objectid2', 'geometry', 'neighbourhood', 'n_id', 'designation',
        'pop_2016', 'pop_2011', 'pop_change', 'private_dwellings',
        'occupied_dwllings', 'pop_dens', 'area', 'total_commuters', 'drive',
        'car_passenger', 'transit', 'walk', 'bike', 'other', 'pct_bike'],
```

```
      dtype='object')
```

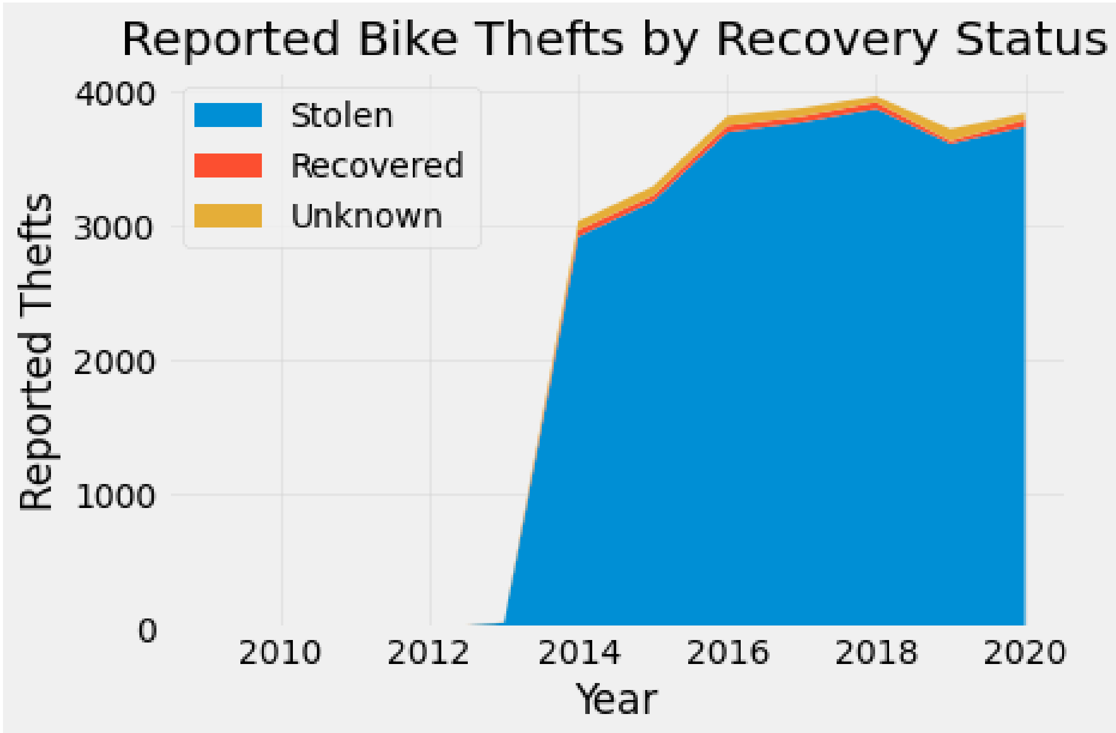
[ ]:
```
thefts_grouped = (thefts_joined
                  .groupby(['occurrence_year', 'status'])
                  .agg(thefts=('_id', 'count'))
                  .reset_index()  # make occurrence year a regular col
                  .pivot(index='occurrence_year', columns='status',␣
 ↪values='thefts')
                  .reset_index()  # ...and again
                  .fillna(0))
thefts_grouped
```

[ ]:

status	occurrence_year	RECOVERED	STOLEN	UNKNOWN
0	2009	0.0	1.0	0.0
1	2010	0.0	2.0	0.0
2	2011	0.0	3.0	0.0
3	2012	0.0	2.0	0.0
4	2013	1.0	43.0	2.0
5	2014	50.0	2916.0	65.0
6	2015	43.0	3177.0	69.0
7	2016	49.0	3692.0	72.0
8	2017	43.0	3766.0	63.0
9	2018	49.0	3865.0	46.0
10	2019	22.0	3606.0	89.0
11	2020	51.0	3734.0	48.0

[ ]:
```
stfig, stax = plt.subplots()

stax.stackplot(thefts_grouped['occurrence_year'], thefts_grouped['STOLEN'],
        thefts_grouped['RECOVERED'], thefts_grouped['UNKNOWN'],
       labels=['Stolen', 'Recovered', 'Unknown'])
stax.set_axisbelow(True)
stax.grid(alpha=0.3)
stax.legend(loc='upper left')
stax.set_title('Reported Bike Thefts by Recovery Status')
stax.set_ylabel('Reported Thefts')
stax.set_xlabel('Year')
```

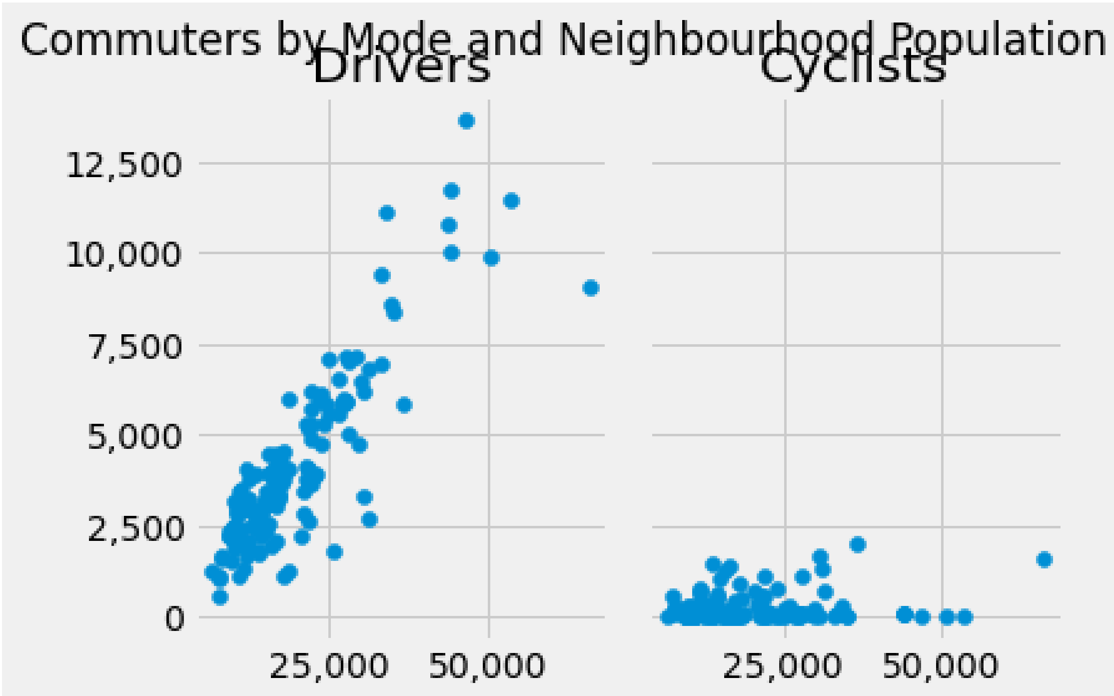
[ ]: Text(0.5, 0, 'Year')



4.13 Subplots

We can create multiple Axes in one Figure by passing nrows and ncols arguments to subplots(). The number of Axes we get equals nrows * ncols. Multiple Axes are returned as a numpy array.

[ ]:
```
modefig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, sharey=True)
ax1.scatter(neighbourhoods['pop_2016'],
            neighbourhoods['drive'])
ax2.scatter(neighbourhoods['pop_2016'],
            neighbourhoods['bike'])
ax1.set_title('Drivers')
ax2.set_title('Cyclists')
ax1.yaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
ax1.xaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
ax2.xaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
modefig.suptitle('Commuters by Mode and Neighbourhood Population')
modefig.tight_layout()
```

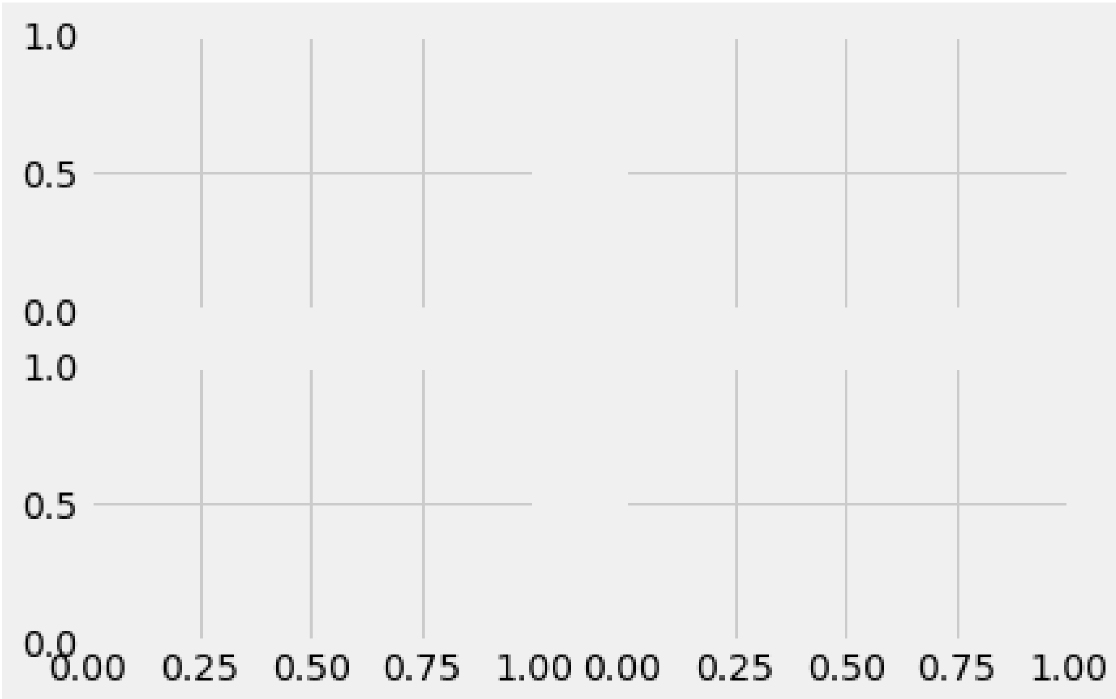


4.13.1 Unpacking subplots

As the number of subplots grows, it gets cumbersome to unpack them in the assignment statement. We can temporarily assign all of them to a single variable.

[ ]:
```
# make a 2x2 grid of subplots
modefig2, mode_ax = plt.subplots(nrows=2, ncols=2, sharey=True, sharex=True)
mode_ax
```

[ ]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd2e6ad0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd28a0d0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd2430d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd1fd110>]],
      dtype=object)
```



The Axes are arranged in a 2x2 array. It would be more straightforward to refer to them if we had a 1x4 array instead.

[ ]:
```
# accessing items in a 2x2 array can be annoying
mode_ax
```

[ ]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd2e6ad0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd28a0d0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd2430d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd1fd110>]],
      dtype=object)
```

[ ]:
```
# example: getting the bottom left Axes
mode_ax[1, 0]
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f87fd2430d0>
```

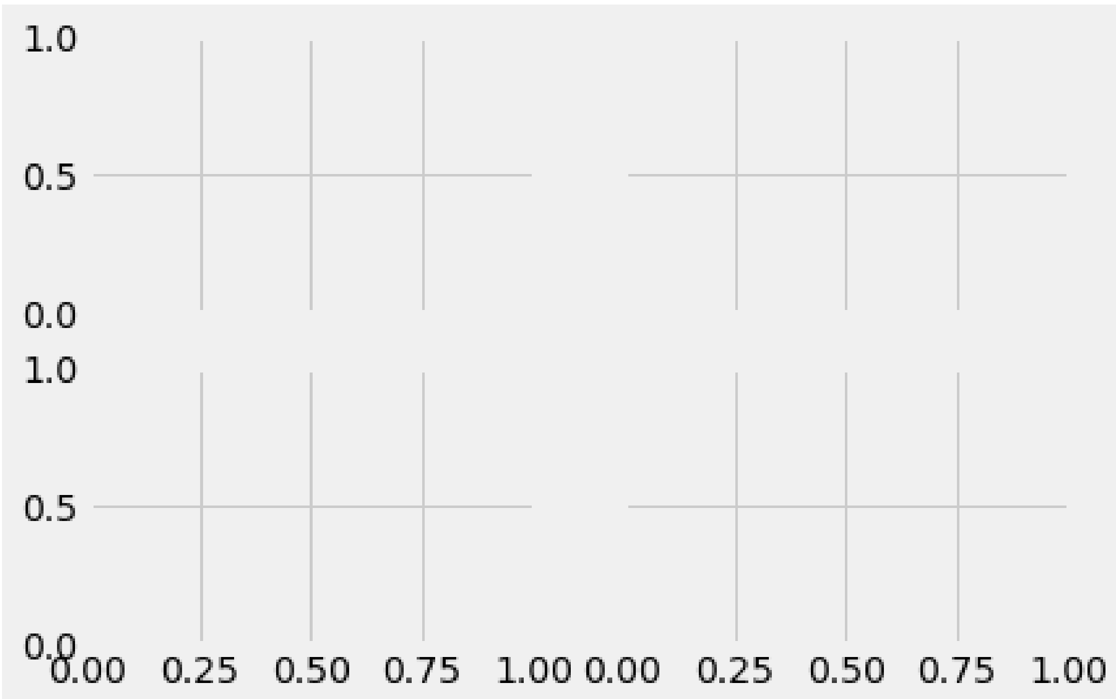
We can take advantage of numpy arrays' flatten() method. Recall that flatten() returns a new array with all the elements arranged in a single row. We can then unpack the elements of that row and assign them to individual variables.

[ ]:
```
# recall what flatten() does
mode_ax.flatten()
```

```
[ ]: array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd2e6ad0>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd28a0d0>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd2430d0>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f87fd1fd110>],
           dtype=object)
```

```
[ ]: a1, a2, a3, a4 = mode_ax.flatten()
     modefig2  # we haven't changed the Figure
```

[ ]:



4.13.2 Plotting with helper functions

Plotting commute mode against total population four times will be tedious. To reuse code, we can write a helper function that takes an Axes, the mode we're plotting, and a dictionary of style parameters and updates the Axes. **param_dict unpacks the dictionary of parameters and arguments passed to plot_modes() and passes them on to scatter().

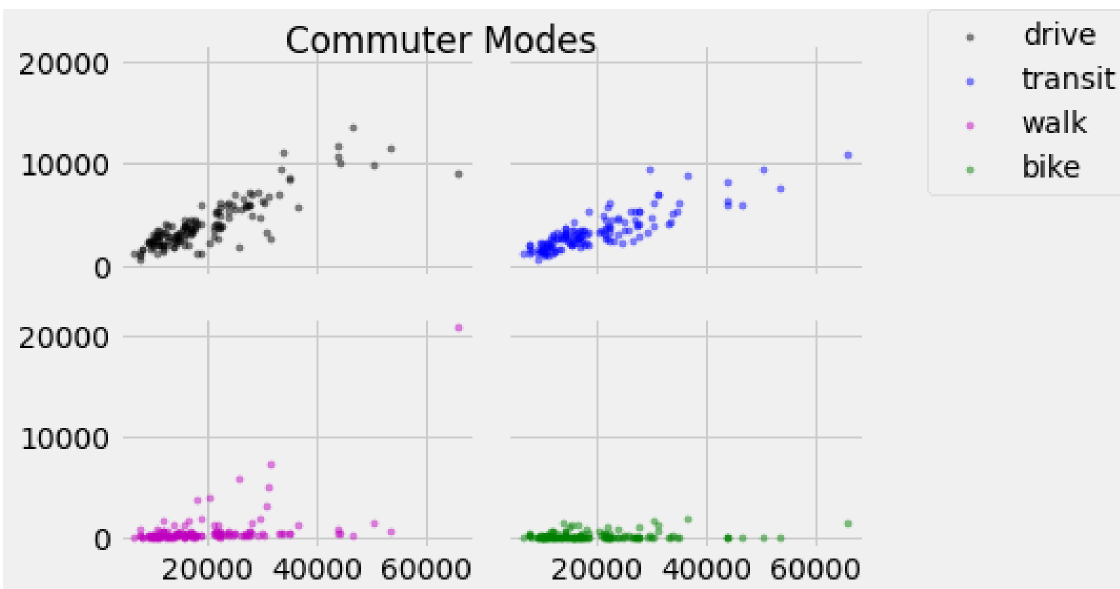
```
[ ]: def plot_modes(ax, mode, param_dict):
         '''
         Helper function to plot neighbourhood pop
         against commuting mode.
         '''
         defaults = {'alpha': 0.45, 's': 10}
         defaults.update(param_dict)
```

```
    out = ax.scatter(neighbourhoods['pop_2016'],
                     neighbourhoods[mode],
                     **defaults)
    return out
```

Then, we can call plot_modes to plot each of the subplots.

```
[ ]: # add data to each axes
plot_modes(a1, 'drive', {'label': 'drive', 'facecolor': 'k'})
plot_modes(a2, 'transit', {'label': 'transit', 'facecolor': 'b'})
plot_modes(a3, 'walk', {'label': 'walk', 'facecolor': 'm'})
plot_modes(a4, 'bike', {'label': 'bike', 'facecolor': 'g'})
modefig2.legend(bbox_to_anchor=(1, 1), loc='upper left')
modefig2.tight_layout()
modefig2.suptitle('Commuter Modes')
modefig2
```

[ ]:



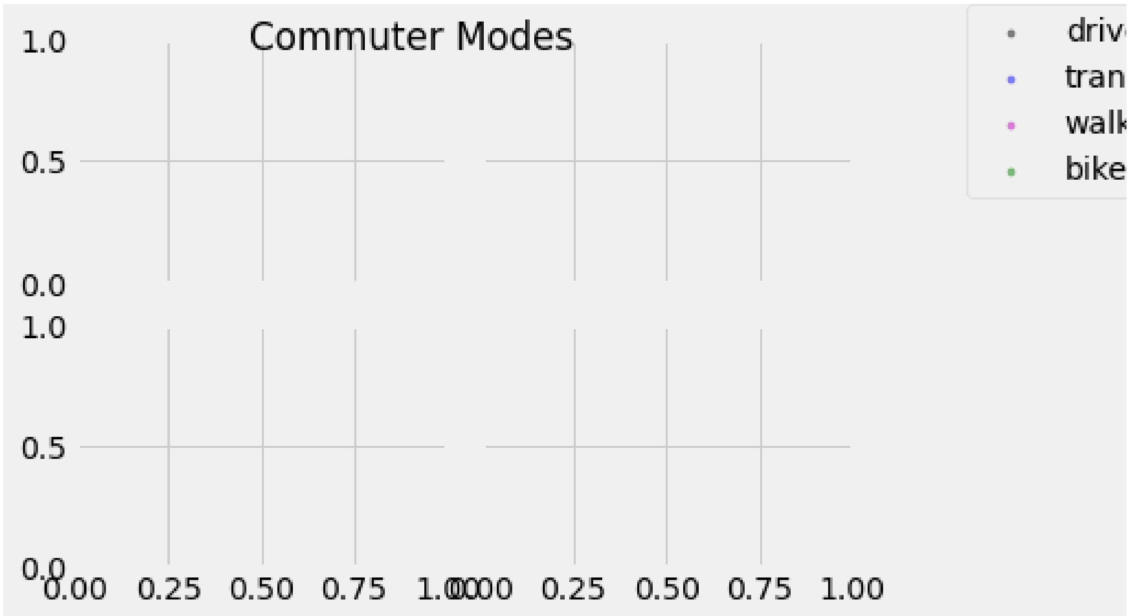
4.13.3 Clearing plots

Successive method calls on an Axes object layer on graphics. To clear everything from an Axes, we can use its clear() method. To clear every subplot in a Figure, we can loop through the flattened array of Axes and clear() each Axes in turn.

```
[ ]: for axes in mode_ax.flatten():
    axes.clear()
```

```
[ ]: modefig2
```

[ ]:



[ ]:
```
# let's reset our style before moving on
plt.style.use('default')
```

5 seaborn

5.1 Easier plotting with seaborn

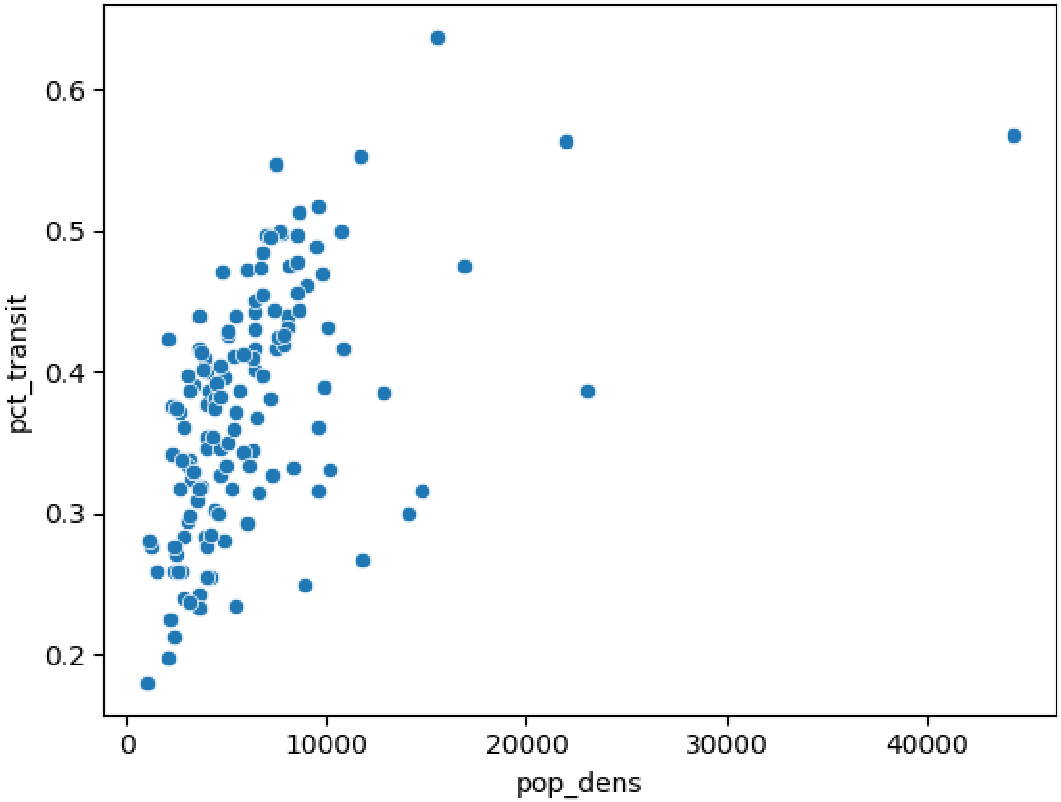
seaborn builds upon and complements matplotlib, producing nicer-looking Axes with less code, and giving us a few more convenient plot types. seaborn is typically given the alias sns, after a pop culture reference.

[ ]:
```
import seaborn as sns
```

With seaborn, we have two ways of structuring arguments to plotting functions: * specifying the x and y axis columns * specifying the data we are visualizing, then the x and y axis columns

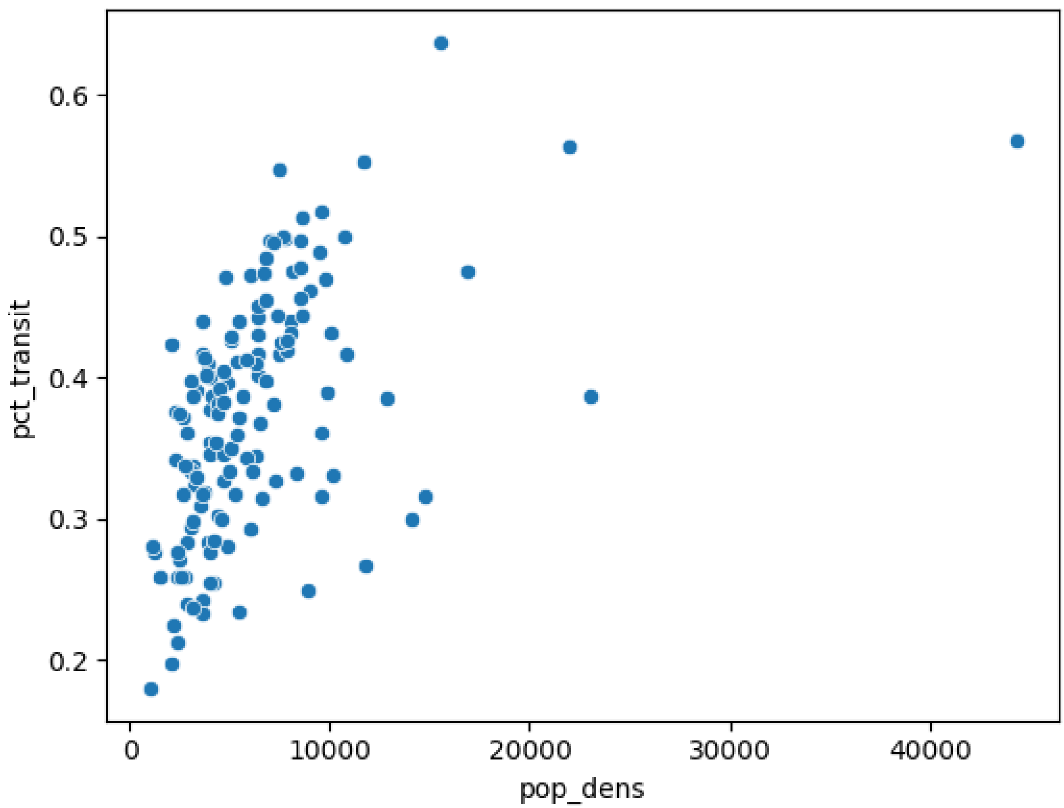
[ ]:
```
# use x and y axis columns
sns.scatterplot(x=neighbourhoods['pop_dens'],
                y=neighbourhoods['pct_transit'])
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f87f171df50>



[ ]:
```
# use the dataframe and column names
sns.scatterplot(data=neighbourhoods,
                x='pop_dens',
                y='pct_transit')
```

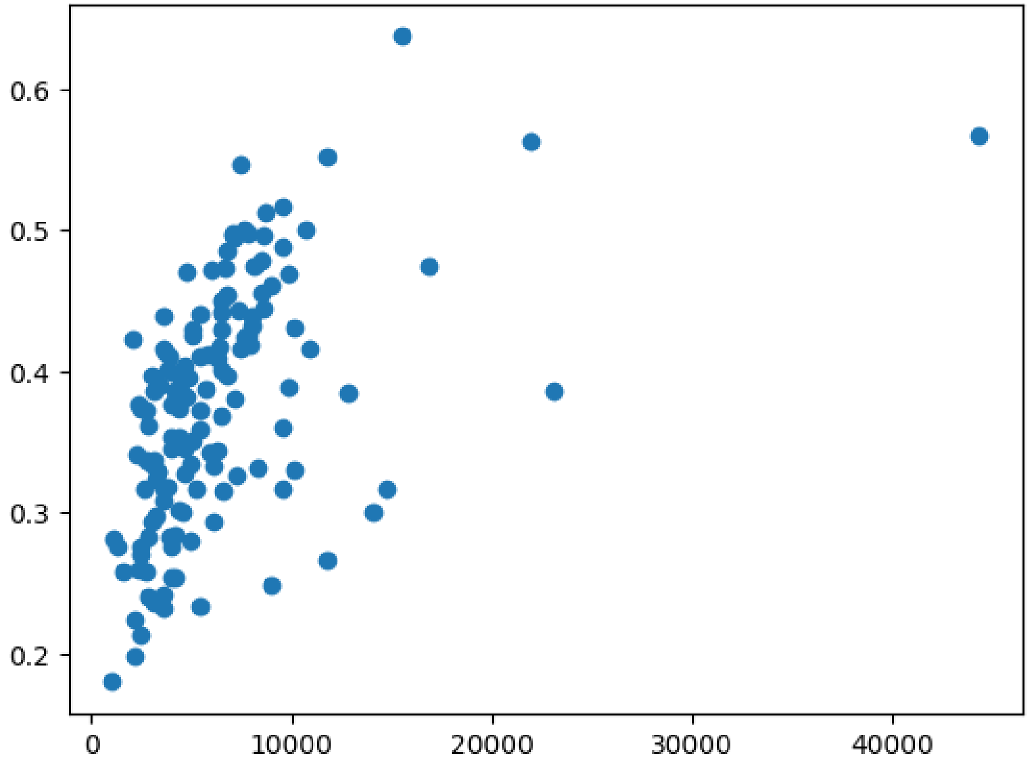
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f87f0e8c110>



For comparison, we can create the same plot using matplotlib's pyplot approach.

[ ]:
```
plt.scatter(neighbourhoods['pop_dens'],
            neighbourhoods['pct_transit'])
```

[ ]: <matplotlib.collections.PathCollection at 0x7f87f0df4b10>



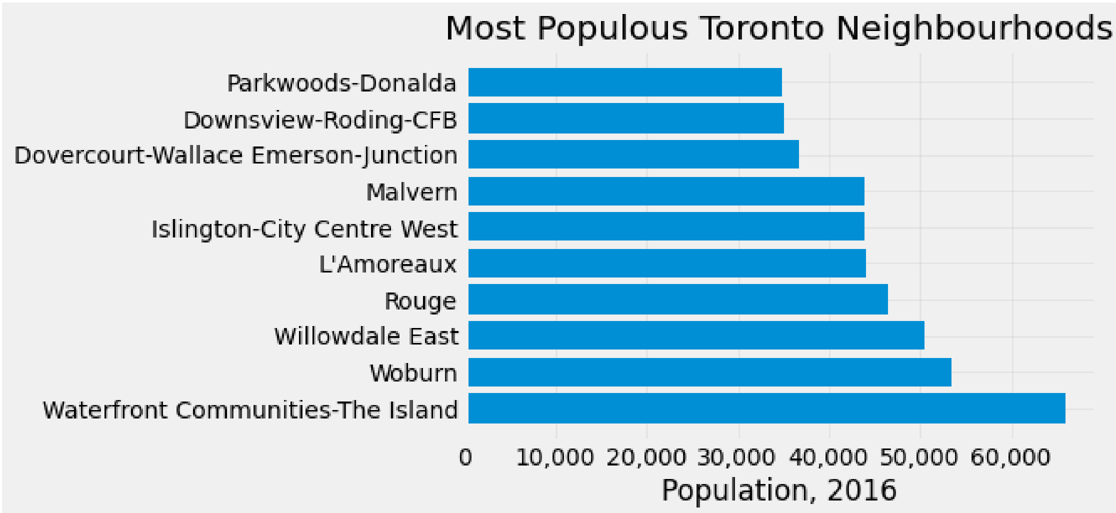
5.2 seaborn and object-oriented matplotlib

We can use seaborn as a complement to matplotlib's object-oriented approach. seaborn functions that work in individual plots have an optional keyword argument that lets us pass in an existing Axes to update. As a bonus, they return the Axes we're working with, making it easy to chain methods together.

Let's revisit our 10 biggest Toronto neighbourhoods chart.

[ ]:
```
bar_fig
```

[ ]:

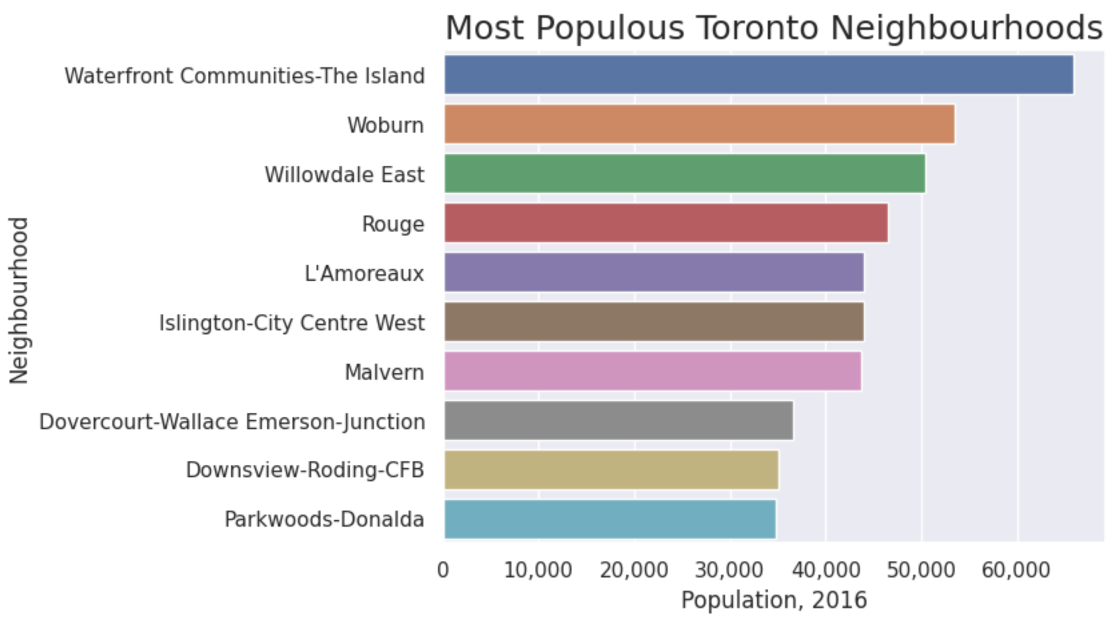


This was the code to create that plot. We'll recreate it with seaborn.

```
bar_fig, bar_ax = plt.subplots()
bar_ax.barh(top10_pop['neighbourhood'], top10_pop['pop_2016'])
bar_ax.xaxis.set_major_formatter('{x:,.0f}')
bar_ax.set_axisbelow(True)
bar_ax.grid(alpha=0.3)
bar_ax.set_title('Most Populous Toronto Neighbourhoods')
bar_ax.set_xlabel('Population, 2016')
```

And with seaborn:

[ ]:
```
sns.set_theme()  # use seaborn's default style settings going forward

sns_fig, sns_ax = plt.subplots()  # create a Figure and Axes
(sns.barplot(data=top10_pop,  # set datasource
            x='pop_2016',  # for a horizontal bar graph
            y='neighbourhood',
            ax=sns_ax)  # plot on an existing Axes
 .set(xlabel='Population, 2016',
      ylabel='Neighbourhood'))

# .set() returns text, so we can't chain .set_title()
sns_ax.set_title('Most Populous Toronto Neighbourhoods',
                fontdict={'fontsize': 18})
sns_ax.xaxis.set_major_formatter(tick.StrMethodFormatter('{x:,.0f}'))
```



5.3 Facets

With matplotlib, we created individual subplots and updated them with a helper function to visualize data for different categories. With seaborn, we can create a FacetGrid and then use its map() method to visualize data by category. map() takes the name of the plotting function to use, then the needed arguments, such as the columns to use for the x-axis and y-axis.

[ ]:
```
# reshape neighbourhood data to support faceting
neighbourhoods_reshaped = (neighbourhoods[['neighbourhood',
                                           'pct_transit',
                                           'pct_drive',
                                           'pct_walk',
                                           'pct_bike']]
                           .melt(id_vars='neighbourhood'))
neighbourhoods_reshaped.head()
```

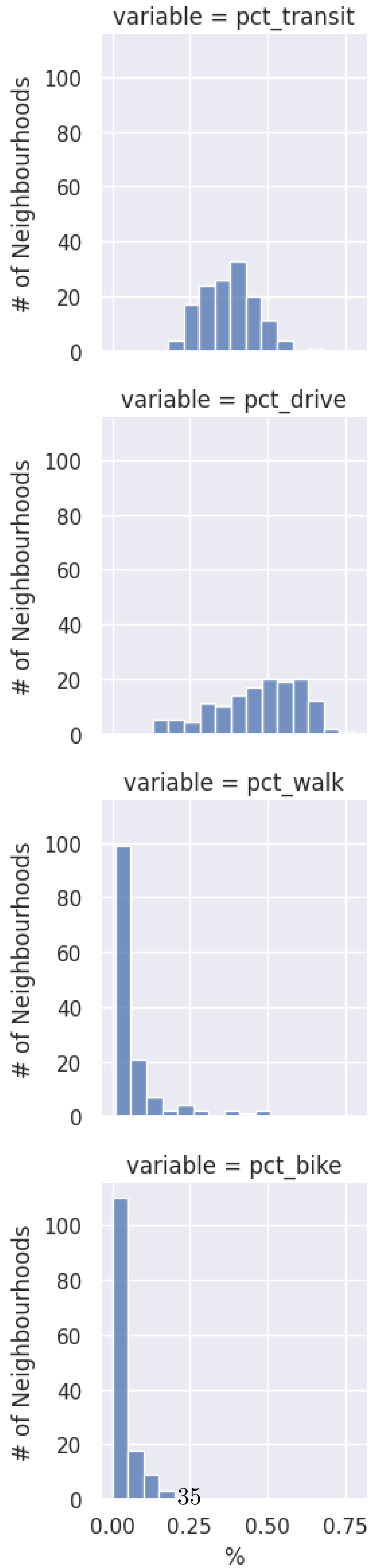
[ ]:

	neighbourhood	variable	value
0	Agincourt North	pct_transit	0.283
1	Agincourt South-Malvern West	pct_transit	0.294
2	Alderwood	pct_transit	0.213
3	Annex	pct_transit	0.416
4	Banbury-Don Mills	pct_transit	0.258

[ ]:
```
# specify the data to use and the column to facet by
# we'll give each variable its own row
```

```python
facets = sns.FacetGrid(data=neighbourhoods_reshaped,
                       row='variable')

# create a histogram for each mode
facets.map(sns.histplot, 'value', binwidth=0.05)
facets.set_axis_labels('%', '# of Neighbourhoods')
```

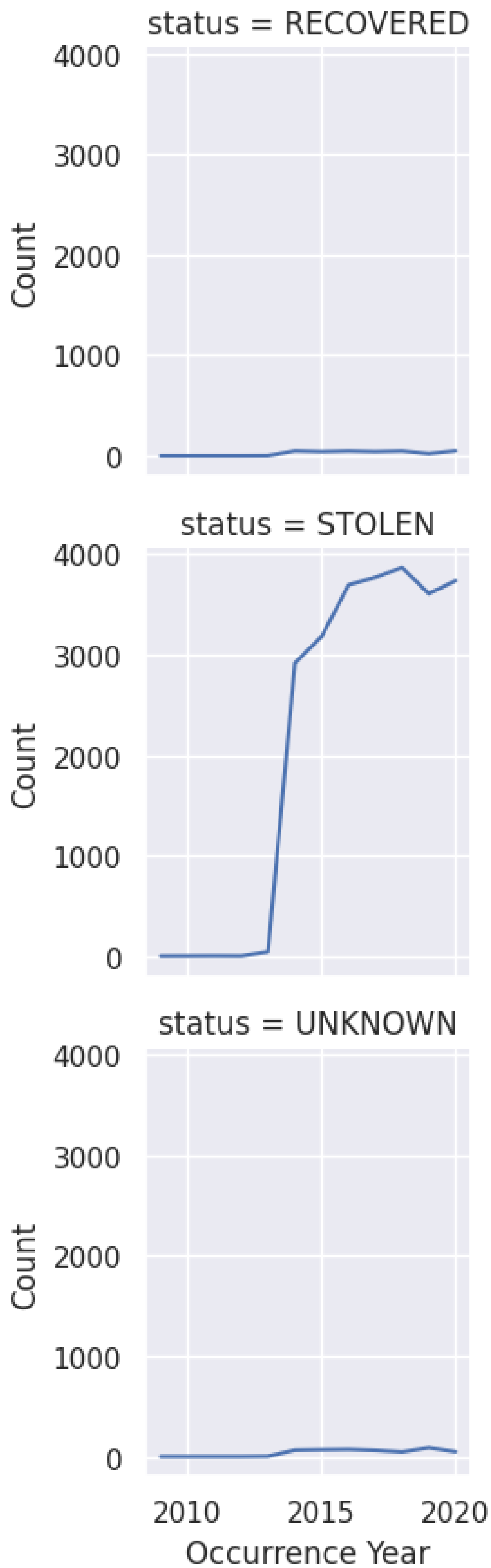
[ ]: <seaborn.axisgrid.FacetGrid at 0x7f87f0c55490>


variable = pct_transit
variable = pct_drive
variable = pct_walk
variable = pct_bike
# of Neighbourhoods
0
20
40
60
80
100
0.00
0.25
0.50
0.75
%

For another example, we can plot reported bike thefts by year, faceted by status.

[ ]:
```
# reshape the theft counts to support faceting
theft_counts_long = thefts_grouped.melt(id_vars='occurrence_year',
                                        value_name='Count')

# specify the data to use and the column to facet by
# we'll give each status its own row
facets = sns.FacetGrid(data=theft_counts_long, row='status')

# for each status, create a lineplot of counts by year
facets.map(sns.lineplot, 'occurrence_year', 'Count')
facets.set_axis_labels('Occurrence Year')
```

[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7f87f0c92890>
```


status = RECOVERED
status = STOLEN
status = UNKNOWN
Count
0
1000
2000
3000
4000
2010
2015
2020
Occurrence Year

5.3.1 Visualization for EDA

`seaborn`'s pair plots are particularly useful for exploratory analyses. `pairplot()` takes a DataFrame or series of columns and creates a Figure containing grid of scatterplots, allowing us to visually look for relationships between variables.

```
[ ]: # review the columns available
     neighbourhoods.columns
```

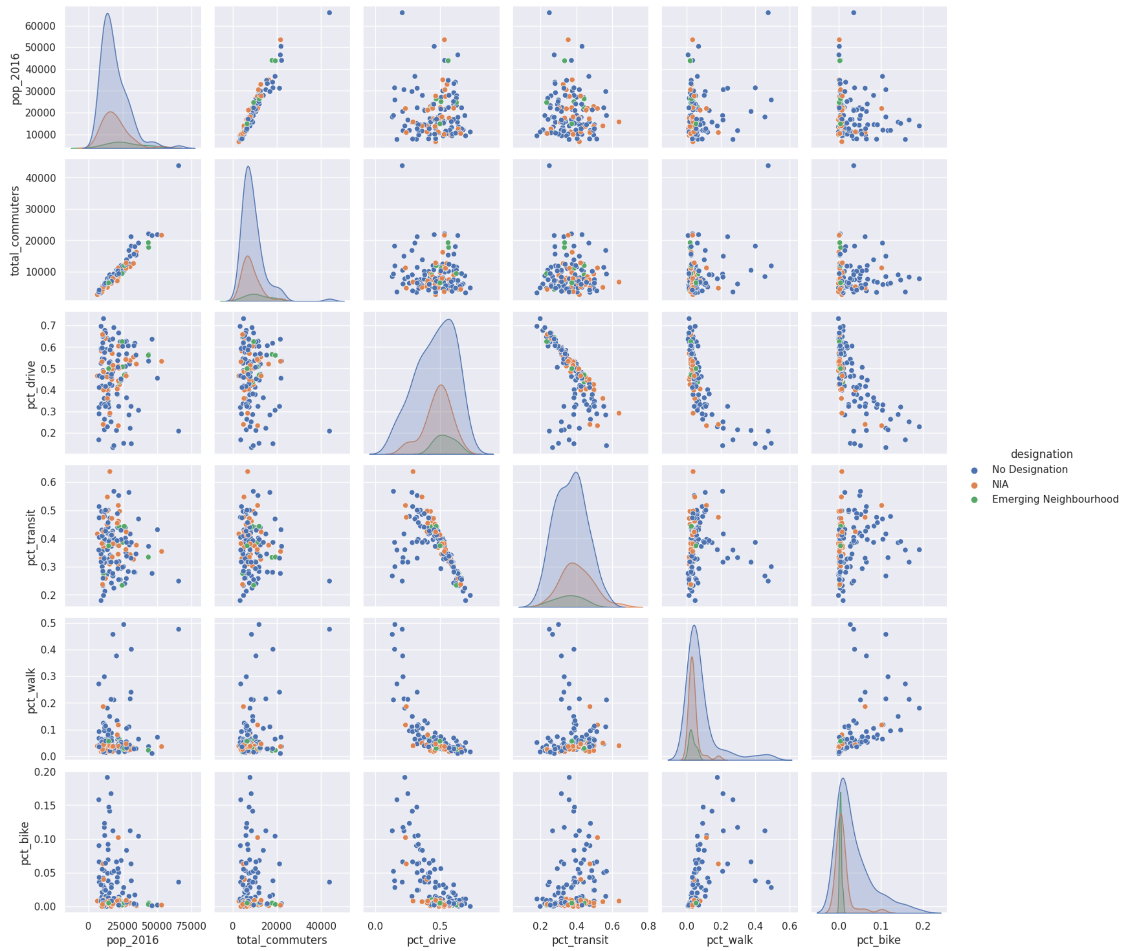
```
[ ]: Index(['neighbourhood', 'n_id', 'designation', 'pop_2016', 'pop_2011',
            'pop_change', 'private_dwellings', 'occupied_dwllings', 'pop_dens',
            'area', 'total_commuters', 'drive', 'car_passenger', 'transit', 'walk',
            'bike', 'other', 'pct_bike', 'pct_drive', 'pct_cp', 'pct_transit',
            'pct_walk'],
           dtype='object')
```

```
[ ]: # review just the numeric columns
     neighbourhoods.select_dtypes('number').columns
```

```
[ ]: Index(['pop_2016', 'pop_2011', 'pop_change', 'private_dwellings',
            'occupied_dwllings', 'pop_dens', 'area', 'total_commuters', 'drive',
            'car_passenger', 'transit', 'walk', 'bike', 'other', 'pct_bike',
            'pct_drive', 'pct_cp', 'pct_transit', 'pct_walk'],
           dtype='object')
```

```
[ ]: # select some columns to use in the pair plot
     cols = ['pop_2016', 'total_commuters', 'pct_drive', 'pct_transit', 'pct_walk',␣
      ↪'pct_bike']
     simple_pairs = sns.pairplot(neighbourhoods[cols])
```

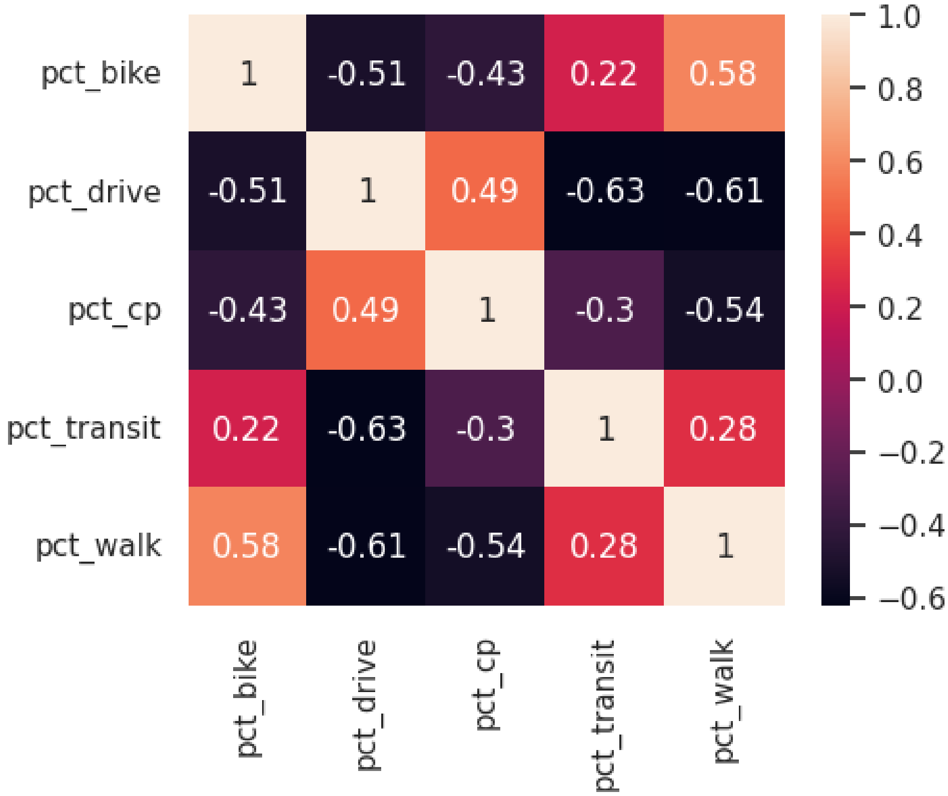
[ ]:
```
# if we include non-numeric variables, they won't be plotted, but we can use␣
↪them for hue
cols = ['pop_2016', 'designation', 'total_commuters', 'pct_drive',␣
↪'pct_transit', 'pct_walk', 'pct_bike']
pairwise_fig = sns.pairplot(neighbourhoods[cols], hue='designation')
```



We can combine seaborn's heatmap() function with the pandas Dataframe corr() method to explore correlations in our data.

[ ]:
```
# calculate correlations with pandas
correlations = neighbourhoods.loc[:, 'pct_bike':].corr('kendall')

# create a figure and axes
corr_fig, corr_ax = plt.subplots()
corr_fig.set_size_inches(5, 4)
sns.heatmap(correlations, ax=corr_ax, annot=True)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f87ee362610>



5.4 Saving Plots

To save a plot, use the Figure savefig() method, which supports exporting figure in common formats like PNG, PDF, and SVG. Setting bbox_inches='tight' will make matplotlib try to figure out the dimensions of the plot and crop the image appropriately. Note that seaborn does not have a plot saving function of its own.

[ ]:
```
pairwise_fig.savefig('pairs.svg', bbox_inches='tight')
corr_fig.savefig('correlations.png', bbox_inches='tight')
```

6 plotly

6.1 Interactive visualizations with plotly

plotly gives us a way to create interactive graphics within Python, building on the plotly.js library rather than matplotlib. Plotly Express provides an entry point to making data visualizations with the package. Let's re-create the drivers vs cyclists scatterplot to start.

[ ]:
```python
import plotly.express as px
```

[ ]:
```python
plotly_fig = px.scatter(neighbourhoods,
                        x='drive',
                        y='bike',
                        title='Commute Modes')
plotly_fig.show(renderer='notebook')  # ensure plot renders nicely in notebook␣
 ↪mode
```

Output hidden; open in https://colab.research.google.com to view.

[ ]:
```python
# add hover data
plotly_fig = px.scatter(neighbourhoods,
                        x='drive',
                        y='bike',
                        hover_name='neighbourhood',  # show neighbourhood on␣
 ↪hover
                        labels={'bike': 'Bike', 'drive':'Drive'},
                        title='Commute Modes')
plotly_fig.show(renderer='notebook')  # ensure plot renders nicely in notebook␣
 ↪mode
```

[ ]:
```python
print(top10_pop.columns)

hist_fig = px.bar(top10_pop,
                  x=['pct_drive', 'pct_cp', 'pct_transit', 'pct_walk',␣
 ↪'pct_bike'],
                  y='neighbourhood',
                  hover_name='neighbourhood',
                  hover_data=['drive', 'car_passenger', 'transit', 'walk',␣
 ↪'bike'],
                  labels={'variable': 'Mode',
                          'value': '%'}
                 )
hist_fig.show(renderer='notebook')
```

```
Index(['neighbourhood', 'n_id', 'designation', 'pop_2016', 'pop_2011',
       'pop_change', 'private_dwellings', 'occupied_dwllings', 'pop_dens',
       'area', 'total_commuters', 'drive', 'car_passenger', 'transit', 'walk',
       'bike', 'other', 'pct_bike', 'pct_drive', 'pct_cp', 'pct_transit',
       'pct_walk'],
      dtype='object')
```

6.1.1 Re-create the population bar chart

[ ]:
```
# view available themes
import plotly.io as pio
pio.templates
```

[ ]:
```
Templates configuration
-----------------------
    Default template: 'plotly'
    Available templates:
        ['ggplot2', 'seaborn', 'simple_white', 'plotly',
         'plotly_white', 'plotly_dark', 'presentation', 'xgridoff',
         'ygridoff', 'gridon', 'none']
```

[ ]:
```
bar_fig = px.bar(top10_pop,
                 x='pop_2016',
                 y='neighbourhood',
                 text='pop_2016',
                 labels={'pop_2016': 'Population, 2016',
                         'neighbourhood': 'Neighbourhood'},
                 hover_data={'neighbourhood': False,
                             'pop_2016':False,
                             'pop_change': ':.2p'},  # add pop change,␣
 ↪formatted as %
                 title='Top Toronto Neighbourhoods by Population',
                 template='seaborn'
                )
bar_fig.show(renderer='notebook')
```

6.2 Futher customizing plotly graphs

For added control over visualizations, we can import plotly's graph_objects submodule.

[ ]:
```
import plotly.graph_objects as go
```

[ ]:
```
transit_hist = go.Histogram(x=neighbourhoods['pct_transit'], name='Transit')
drive_hist = go.Histogram(x=neighbourhoods['pct_drive'], name='Drive')

data = [drive_hist, transit_hist]

layout = go.Layout(template='seaborn',
                   title='Commute Mode Distribution',
                   xaxis={'title': 'Mode %'},
                   yaxis={'title': 'Neighbourhoods'}
                   )
```

```
fig = go.Figure(data=data, layout=layout)
fig.update_layout(hovermode='x')
fig.show(renderer='notebook')
```

6.3 Saving plotly visualizations

We can save visualizatons created in plotly to image or PDF with the write_image() Figure method. Note that write_image() needs the kaleido package to work.

[ ]:
```
!pip install -U kaleido
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: kaleido in /usr/local/lib/python3.7/dist-packages
(0.2.1)
```

[ ]:
```
import kaleido
```

[ ]:
```
fig.write_image('fig.pdf', format='pdf')
```

7 References

- Matplotlib development team. *Basic usage*. https://matplotlib.org/stable/tutorials/introdu
- Matplotlib development team. *The lifecycle of a plot*. https://matplotlib.org/stable/tutorials/introductory/lifecycle.html#sphx-glr-tutorials-int
- Matplotlib development team. *API reference*. https://matplotlib.org/stable/api/index.html
- Plotly. *Getting started*. https://plotly.com/python/getting-started/
- Plotly. *Fundamentals*. https://plotly.com/python/plotly-fundamentals/
- Waskom, M. *An introduction to seaborn*. https://seaborn.pydata.org/introduction.html
- Waskom, M. *API reference*. https://seaborn.pydata.org/api.html