Unit 4 Challenge - Tier 3

November 6, 2025

1 Springboard Data Science Career Track Unit 4 Challenge - Tier 3 Complete

1.1 Objectives

Hey! Great job getting through those challenging DataCamp courses. You're learning a lot in a short span of time.

In this notebook, you're going to apply the skills you've been learning, bridging the gap between the controlled environment of DataCamp and the *slightly* messier work that data scientists do with actual datasets!

Here's the mystery we're going to solve: ***which boroughs of London have seen the greatest increase in housing prices, on average, over the last two decades?***

A borough is just a fancy word for district. You may be familiar with the five boroughs of New York… well, there are 32 boroughs within Greater London (here's some info for the curious). Some of them are more desirable areas to live in, and the data will reflect that with a greater rise in housing prices.

This is the Tier 3 notebook, which means it's not filled in at all: we'll just give you the skeleton of a project, the brief and the data. It's up to you to play around with it and see what you can find out! Good luck! If you struggle, feel free to look at easier tiers for help; but try to dip in and out of them, as the more independent work you do, the better it is for your learning!

This challenge will make use of only what you learned in the following DataCamp courses: - Prework courses (Introduction to Python for Data Science, Intermediate Python for Data Science) - Data Types for Data Science - Python Data Science Toolbox (Part One) - pandas Foundations - Manipulating DataFrames with pandas - Merging DataFrames with pandas

Of the tools, techniques and concepts in the above DataCamp courses, this challenge should require the application of the following: - **pandas** - **data ingestion and inspection** (pandas Foundations, Module One) - **exploratory data analysis** (pandas Foundations, Module Two) - **tidying and cleaning** (Manipulating DataFrames with pandas, Module Three) - **transforming DataFrames** (Manipulating DataFrames with pandas, Module One) - **subsetting DataFrames with lists** (Manipulating DataFrames with pandas, Module One) - **filtering DataFrames** (Manipulating DataFrames with pandas, Module One) - **grouping data** (Manipulating DataFrames with pandas, Module Four) - **melting data** (Manipulating DataFrames with pandas, Module Three) - **advanced indexing** (Manipulating DataFrames with pandas, Module Four) - **matplotlib** (Intermediate Python for Data Science, Module One) - **fundamental data types** (Data Types for

Data Science, Module One) - **dictionaries** (Intermediate Python for Data Science, Module Two) - **handling dates and times** (Data Types for Data Science, Module Four) - **function definition** (Python Data Science Toolbox - Part One, Module One) - **default arguments, variable length, and scope** (Python Data Science Toolbox - Part One, Module Two) - **lambda functions and error handling** (Python Data Science Toolbox - Part One, Module Four)

1.2 The Data Science Pipeline

This is Tier Three, so we'll get you started. But after that, it's all in your hands! When you feel done with your investigations, look back over what you've accomplished, and prepare a quick presentation of your findings for the next mentor meeting.

Data Science is magical. In this case study, you'll get to apply some complex machine learning algorithms. But as David Spiegelhalter reminds us, there is no substitute for simply **taking a really, really good look at the data.** Sometimes, this is all we need to answer our question.

Data Science projects generally adhere to the four stages of Data Science Pipeline: 1. Sourcing and loading 2. Cleaning, transforming, and visualizing 3. Modeling 4. Evaluating and concluding

1.2.1 1. Sourcing and Loading

Any Data Science project kicks off by importing ***pandas***. The documentation of this wonderful library can be found here. As you've seen, pandas is conveniently connected to the Numpy and Matplotlib libraries.

Hint: This part of the data science pipeline will test those skills you acquired in the pandas Foundations course, Module One.

1.1. Importing Libraries

[312]:
```
# Let's import the pandas, numpy libraries as pd, and np respectively.
import numpy as np
import pandas as pd

# Load the pyplot collection of functions from matplotlib, as plt
import matplotlib.pyplot as plt
```

1.2. Loading the data Your data comes from the London Datastore: a free, open-source data-sharing portal for London-oriented datasets.

[313]:
```
# First, make a variable called url_LondonHousePrices, and assign it the␣
↪following link, enclosed in quotation-marks as a string:
# https://data.london.gov.uk/download/uk-house-price-index/
↪70ac0766-8902-4eb5-aab5-01951aaed773/UK%20House%20price%20index.xls

url_LondonHousePrices = "https://data.london.gov.uk/download/
↪uk-house-price-index/70ac0766-8902-4eb5-aab5-01951aaed773/
↪UK%20House%20price%20index.xls"
```

```
# The dataset we're interested in contains the Average prices of the houses,␣
↪and is actually on a particular sheet of the Excel file.
# As a result, we need to specify the sheet name in the read_excel() method.
# Put this data into a variable called properties.
properties = pd.read_excel(url_LondonHousePrices, sheet_name='Average price',␣
↪index_col= None)
```

1.2.2 2. Cleaning, transforming, and visualizing

This second stage is arguably the most important part of any Data Science project. The first thing to do is take a proper look at the data. Cleaning forms the majority of this stage, and can be done both before or after Transformation.

The end goal of data cleaning is to have tidy data. When data is tidy:

1. Each variable has a column.
2. Each observation forms a row.

Keep the end goal in mind as you move through this process, every step will take you closer.

Hint: This part of the data science pipeline should test those skills you acquired in: - Intermediate Python for data science, all modules. - pandas Foundations, all modules. - Manipulating DataFrames with pandas, all modules. - Data Types for Data Science, Module Four. - Python Data Science Toolbox - Part One, all modules

2.1. Exploring your data

Think about your pandas functions for checking out a dataframe.

[314]:
```
print(properties.info())
print(properties.head())
#print(properties.tail())
print(properties.shape)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 369 entries, 0 to 368
Data columns (total 49 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Unnamed: 0            368 non-null    datetime64[ns]
 1   City of London        369 non-null    object
 2   Barking & Dagenham    369 non-null    object
 3   Barnet                369 non-null    object
 4   Bexley                369 non-null    object
 5   Brent                 369 non-null    object
 6   Bromley               369 non-null    object
 7   Camden                369 non-null    object
 8   Croydon               369 non-null    object
 9   Ealing                369 non-null    object
 10  Enfield               369 non-null    object
 11  Greenwich             369 non-null    object
```

```
 12  Hackney                369 non-null    object
 13  Hammersmith & Fulham   369 non-null    object
 14  Haringey               369 non-null    object
 15  Harrow                 369 non-null    object
 16  Havering               369 non-null    object
 17  Hillingdon             369 non-null    object
 18  Hounslow               369 non-null    object
 19  Islington              369 non-null    object
 20  Kensington & Chelsea   369 non-null    object
 21  Kingston upon Thames   369 non-null    object
 22  Lambeth                369 non-null    object
 23  Lewisham               369 non-null    object
 24  Merton                 369 non-null    object
 25  Newham                 369 non-null    object
 26  Redbridge              369 non-null    object
 27  Richmond upon Thames   369 non-null    object
 28  Southwark              369 non-null    object
 29  Sutton                 369 non-null    object
 30  Tower Hamlets           369 non-null    object
 31  Waltham Forest         369 non-null    object
 32  Wandsworth             369 non-null    object
 33  Westminster            369 non-null    object
 34  Unnamed: 34            0 non-null      float64
 35  Inner London           369 non-null    object
 36  Outer London           369 non-null    object
 37  Unnamed: 37            0 non-null      float64
 38  NORTH EAST             369 non-null    object
 39  NORTH WEST             369 non-null    object
 40  YORKS & THE HUMBER     369 non-null    object
 41  EAST MIDLANDS          369 non-null    object
 42  WEST MIDLANDS          369 non-null    object
 43  EAST OF ENGLAND        369 non-null    object
 44  LONDON                 369 non-null    object
 45  SOUTH EAST             369 non-null    object
 46  SOUTH WEST             369 non-null    object
 47  Unnamed: 47            0 non-null      float64
 48  England                369 non-null    object
dtypes: datetime64[ns](1), float64(3), object(45)
memory usage: 141.4+ KB
None
  Unnamed: 0 City of London Barking & Dagenham     Barnet     Bexley  \
0        NaT      E09000001          E09000002  E09000003  E09000004
1 1995-01-01          90347              51870      98948      64956
2 1995-02-01          81213              52513      98848      64786
3 1995-03-01          78168              52701      97848      64366
4 1995-04-01          76172              54618      96273      64276

       Brent    Bromley     Camden    Croydon     Ealing  … NORTH WEST  \
```

```
0  E09000005  E09000006  E09000007  E09000008  E09000009  …  E12000002
1      76880      83082     119775      70118      85469  …      40907
2      77651      83068     118365      69908      86551  …      40877
3      77644      82856     119131      69666      87067  …      41351
4      78668      82525     118948      69562      87933  …      41195

  YORKS & THE HUMBER EAST MIDLANDS WEST MIDLANDS EAST OF ENGLAND     LONDON  \
0          E12000003      E12000004      E12000005       E12000006  E12000007
1              42171          43856          46470           56098      79687
2              41912          44344          47249           55991      77913
3              42544          43701          47345           55574      79110
4              42934          44414          47359           55966      79708

  SOUTH EAST SOUTH WEST Unnamed: 47    England
0  E12000008  E12000009         NaN  E92000001
1      64502      52799         NaN      50231
2      64196      52462         NaN      50130
3      64597      51716         NaN      50229
4      65111      52877         NaN      50597

[5 rows x 49 columns]
(369, 49)
```

2.2. Cleaning the data

You might find you need to transpose your dataframe, check out what its row indexes are, and reset the index. You also might find you need to assign the values of the first row to your column headings . (Hint: recall the .columns feature of DataFrames, as well as the iloc[] method).

Don't be afraid to use StackOverflow for help with this.

2.3. Cleaning the data (part 2)

You might we have to **rename** a couple columns. How do you do this? The clue's pretty bold…

[393]:
```python
properties_transposed = properties.T.reset_index()
print(properties_transposed.head(3))
```

```
                index          0                    1                    2  \
0          Unnamed: 0        NaT  1995-01-01 00:00:00  1995-02-01 00:00:00
1      City of London  E09000001                90347                81213
2  Barking & Dagenham  E09000002                51870                52513

                     3                    4                    5  \
0  1995-03-01 00:00:00  1995-04-01 00:00:00  1995-05-01 00:00:00
1                78168                76172                83392
2                52701                54618                54524

                     6                    7                    8  …  \
0  1995-06-01 00:00:00  1995-07-01 00:00:00  1995-08-01 00:00:00  …
1                93757               108801               110976  …
```

```
2            55200            53569            53691  …

                  359                  360                  361  \
0  2024-11-01 00:00:00  2024-12-01 00:00:00  2025-01-01 00:00:00
1               728756               770014               789031
2               350798               354406               356394

                  362                  363                  364  \
0  2025-02-01 00:00:00  2025-03-01 00:00:00  2025-04-01 00:00:00
1               749626               715511               747919
2               361644               369633               371161

                  365                  366                  367  \
0  2025-05-01 00:00:00  2025-06-01 00:00:00  2025-07-01 00:00:00
1               802389               807327               780192
2               367979               362254               364858

                  368
0  2025-08-01 00:00:00
1               711729
2               362062

[3 rows x 370 columns]
```

[316]:

```
properties_col = properties_transposed.iloc[0]
new_prop = pd.DataFrame(properties_transposed.values[1:], columns =
 ↪properties_col)
print(new_prop.head())
new_prop.columns
```

```
0          Unnamed: 0        NaT 1995-01-01 00:00:00 1995-02-01 00:00:00  \
0      City of London  E09000001               90347               81213
1  Barking & Dagenham  E09000002               51870               52513
2              Barnet  E09000003               98948               98848
3              Bexley  E09000004               64956               64786
4               Brent  E09000005               76880               77651

0 1995-03-01 00:00:00 1995-04-01 00:00:00 1995-05-01 00:00:00  \
0               78168               76172               83392
1               52701               54618               54524
2               97848               96273               95737
3               64366               64276               63995
4               77644               78668               79464

0 1995-06-01 00:00:00 1995-07-01 00:00:00 1995-08-01 00:00:00  …  \
0               93757              108801              110976  …
1               55200               53569               53691  …
2               95577               96992               97968  …
```

```
3             64251               63721               64431  …
4             80118               79920               79295  …

0 2024-11-01 00:00:00 2024-12-01 00:00:00 2025-01-01 00:00:00  \
0              728756              770014              789031
1              350798              354406              356394
2              628891              621602              611125
3              405701              403968              403383
4              590969              579308              567742

0 2025-02-01 00:00:00 2025-03-01 00:00:00 2025-04-01 00:00:00  \
0              749626              715511              747919
1              361644              369633              371161
2              599845              597110              591360
3              405699              405705              398795
4              564953              564650              567344

0 2025-05-01 00:00:00 2025-06-01 00:00:00 2025-07-01 00:00:00  \
0              802389              807327              780192
1              367979              362254              364858
2              583161              592221              601498
3              400821              402247              411080
4              557669              543691              536131

0 2025-08-01 00:00:00
0              711729
1              362062
2              615503
3              412151
4              560535

[5 rows x 370 columns]
```

[316]:
```
Index([       'Unnamed: 0',                 NaT, 1995-01-01 00:00:00,
       1995-02-01 00:00:00, 1995-03-01 00:00:00, 1995-04-01 00:00:00,
       1995-05-01 00:00:00, 1995-06-01 00:00:00, 1995-07-01 00:00:00,
       1995-08-01 00:00:00,
       …
       2024-11-01 00:00:00, 2024-12-01 00:00:00, 2025-01-01 00:00:00,
       2025-02-01 00:00:00, 2025-03-01 00:00:00, 2025-04-01 00:00:00,
       2025-05-01 00:00:00, 2025-06-01 00:00:00, 2025-07-01 00:00:00,
       2025-08-01 00:00:00],
      dtype='object', name=0, length=370)
```

2.4.Transforming the data

Remember what Wes McKinney said about tidy data?

You might need to **melt** your DataFrame here.

[324]:
```python
new_prop.rename(columns = {new_prop.columns[0]: 'London_boroughs', new_prop.
 ↪columns[1]: 'ID'}, inplace = True)
properties_melt = pd.melt(new_prop, id_vars = ['London_boroughs', 'ID'],␣
 ↪var_name = 'Date', value_name = 'Avg_price')
print(properties_melt.head())
```

```
      London_boroughs          ID                Date Avg_price
0      City of London  E09000001  1995-01-01 00:00:00     90347
1  Barking & Dagenham  E09000002  1995-01-01 00:00:00     51870
2              Barnet  E09000003  1995-01-01 00:00:00     98948
3              Bexley  E09000004  1995-01-01 00:00:00     64956
4               Brent  E09000005  1995-01-01 00:00:00     76880
```

Remember to make sure your column data types are all correct. Average prices, for example, should be floating point numbers…

[325]:
```python
properties_melt['Date'] = pd.to_datetime(properties_melt['Date'])
properties_melt['Avg_price'] = properties_melt['Avg_price'].astype(float)
print(properties_melt.info())
print(properties_melt.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17664 entries, 0 to 17663
Data columns (total 4 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   London_boroughs  17664 non-null  object
 1   ID               16560 non-null  object
 2   Date             17664 non-null  datetime64[ns]
 3   Avg_price        16560 non-null  float64
dtypes: datetime64[ns](1), float64(1), object(2)
memory usage: 552.1+ KB
None
      London_boroughs          ID       Date  Avg_price
0      City of London  E09000001 1995-01-01    90347.0
1  Barking & Dagenham  E09000002 1995-01-01    51870.0
2              Barnet  E09000003 1995-01-01    98948.0
3              Bexley  E09000004 1995-01-01    64956.0
4               Brent  E09000005 1995-01-01    76880.0
```

2.5. Cleaning the data (part 3)

Do we have an equal number of observations in the ID, Average Price, Month, and London Borough columns? Remember that there are only 32 London Boroughs. How many entries do you have in that column?

Check out the contents of the London Borough column, and if you find null values, get rid of them however you see fit.

[332]:
```
print(properties_melt.isna().any())
print(properties_melt.isna().sum())
print(properties_melt.shape)
prop_clean = properties_melt.dropna()
print(prop_clean.isna().any())
```

```
London_boroughs    False
ID                  True
Date               False
Avg_price           True
dtype: bool
London_boroughs       0
ID                 1104
Date                  0
Avg_price          1104
dtype: int64
(17664, 4)
London_boroughs    False
ID                 False
Date               False
Avg_price          False
dtype: bool
```

2.6. Visualizing the data

To visualize the data, why not subset on a particular London Borough? Maybe do a line plot of Month against Average Price?

[333]:
```
prop_clean.loc[:,'Year'] = prop_clean['Date'].dt.year
prop_clean.loc[:,'Month'] = prop_clean['Date'].dt.month
print(prop_clean.head())
```

```
      London_boroughs          ID       Date  Avg_price  Year  Month
0      City of London  E09000001 1995-01-01    90347.0  1995      1
1  Barking & Dagenham  E09000002 1995-01-01    51870.0  1995      1
2              Barnet  E09000003 1995-01-01    98948.0  1995      1
3              Bexley  E09000004 1995-01-01    64956.0  1995      1
4               Brent  E09000005 1995-01-01    76880.0  1995      1
```

```
/var/folders/hq/kk6s209d61dcscgjxy9bp4tw0000gn/T/ipykernel_57889/997687871.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  prop_clean.loc[:,'Year'] = prop_clean['Date'].dt.year
/var/folders/hq/kk6s209d61dcscgjxy9bp4tw0000gn/T/ipykernel_57889/997687871.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
```

```
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  prop_clean.loc[:,'Month'] = prop_clean['Date'].dt.month
```

To limit the number of data points you have, you might want to extract the year from every month value your *Month* column.

To this end, you *could* apply a ***lambda function***. Your logic could work as follows: 1. look through the `Month` column 2. extract the year from each individual value in that column 3. store that corresponding year as separate column.

Whether you go ahead with this is up to you. Just so long as you answer our initial brief: which boroughs of London have seen the greatest house price increase, on average, over the past two decades?

[380]:
```python
from scipy import stats
desc_stat = prop_clean.groupby(['London_boroughs','Year'])['Avg_price'].
 ↪agg(mean_price = 'mean', std_error =
                                  lambda x: stats.sem(x, ddof = 1)).reset_index()
print(desc_stat)
```

```
         London_boroughs  Year     mean_price    std_error
0     Barking & Dagenham  1995   53265.416667   305.060040
1     Barking & Dagenham  1996   53162.666667   266.963462
2     Barking & Dagenham  1997   57537.583333   338.416017
3     Barking & Dagenham  1998   61969.666667   577.836250
4     Barking & Dagenham  1999   67145.333333   467.053552
…                    …   …            …          …
1390  YORKS & THE HUMBER  2021  175033.833333  1590.090939
1391  YORKS & THE HUMBER  2022  191155.833333  1810.666497
1392  YORKS & THE HUMBER  2023  192742.500000   835.833140
1393  YORKS & THE HUMBER  2024  196979.000000  1291.157057
1394  YORKS & THE HUMBER  2025  203766.375000  1369.595377

[1395 rows x 4 columns]
```

[ ]:

[ ]:

[384]:
```python
bark_dag = desc_stat[(desc_stat['London_boroughs'] == 'Barking & Dagenham') &␣
 ↪(desc_stat['Year'].between(2000, 2020))]
print(bark_dag.head())
hamm_ful = desc_stat[(desc_stat['London_boroughs'] == 'Hammersmith & Fulham') &␣
 ↪(desc_stat['Year'].between(2000, 2020))]
print(hamm_ful.head())
```

```
enfield = desc_stat[(desc_stat['London_boroughs'] == 'Enfield') &␣
 ↪(desc_stat['Year'].between(2000, 2020))]
print(enfield.head())
sutt = desc_stat[(desc_stat['London_boroughs'] == 'Sutton') &␣
 ↪(desc_stat['Year'].between(2000, 2020))]
print(sutt.head())
```

```
      London_boroughs  Year     mean_price    std_error
5   Barking & Dagenham  2000   79715.500000  1244.078109
6   Barking & Dagenham  2001   91140.250000  1521.845516
7   Barking & Dagenham  2002  115356.250000  3267.367108
8   Barking & Dagenham  2003  146478.750000  1108.238283
9   Barking & Dagenham  2004  162593.666667  1553.686483
          London_boroughs  Year     mean_price    std_error
470  Hammersmith & Fulham  2000  256096.333333  3490.371241
471  Hammersmith & Fulham  2001  281629.416667  2424.467528
472  Hammersmith & Fulham  2002  313260.333333  5366.348799
473  Hammersmith & Fulham  2003  330190.500000  2635.816530
474  Hammersmith & Fulham  2004  355643.500000  3361.576654
    London_boroughs  Year     mean_price    std_error
346         Enfield  2000  129405.916667  1769.780659
347         Enfield  2001  144141.333333  2112.194415
348         Enfield  2002  169714.500000  4034.296254
349         Enfield  2003  202408.000000  1321.375781
350         Enfield  2004  218631.750000  2097.893298
     London_boroughs  Year     mean_price    std_error
1183          Sutton  2000  127332.250000  1991.120117
1184          Sutton  2001  139421.583333  1943.514122
1185          Sutton  2002  164307.750000  3150.456776
1186          Sutton  2003  189874.083333  1031.409322
1187          Sutton  2004  204041.083333  2415.975073
```

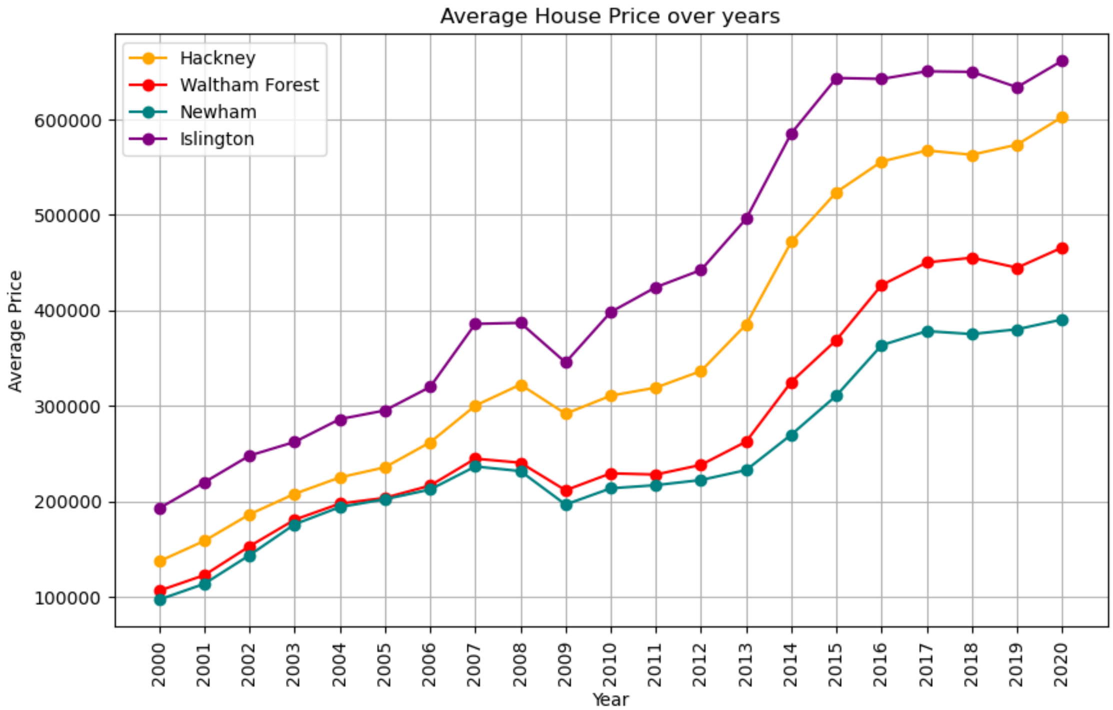
[391]:

```
hackney  = desc_stat[(desc_stat['London_boroughs'] == 'Hackney') &␣
 ↪(desc_stat['Year'].between(2000, 2020))]
print(hackney.head())
walth = desc_stat[(desc_stat['London_boroughs'] == 'Waltham Forest') &␣
 ↪(desc_stat['Year'].between(2000, 2020))]
print(walth.head())
new = desc_stat[(desc_stat['London_boroughs'] == 'Newham') & (desc_stat['Year'].
 ↪between(2000, 2020))]
print(new.head())
isl = desc_stat[(desc_stat['London_boroughs'] == 'Islington') &␣
 ↪(desc_stat['Year'].between(2000, 2020))]
print(isl.head())

ax = hackney.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o',␣
 ↪color = "orange",figsize = (10,6), label = "Hackney")
```

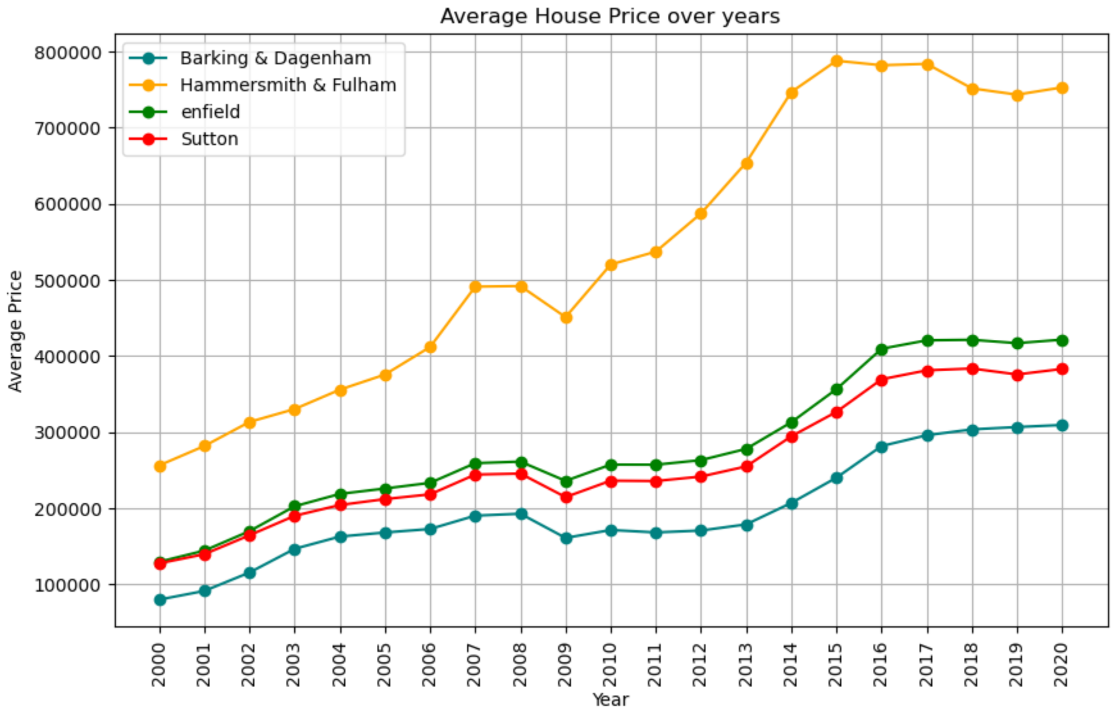
```
walth.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o', color =
    "red",ax = ax ,figsize = (10,6), label = "Waltham Forest")
new.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o', color =
    "teal",ax = ax ,figsize = (10,6), label = "Newham")
isl.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o', color =
    "purple",ax = ax ,figsize = (10,6), label = "Islington")
plt.xlabel('Year')
plt.ylabel('Average Price')
plt.title(f'Average House Price over years')
plt.xticks(range(2000, 2021, 1), rotation = 90)
plt.grid(True)
```

```
    London_boroughs  Year     mean_price    std_error
439         Hackney  2000  137228.583333  2633.154032
440         Hackney  2001  158608.583333  2244.104655
441         Hackney  2002  186382.916667  4015.592651
442         Hackney  2003  207890.250000   752.405185
443         Hackney  2004  225058.833333  2504.723035
     London_boroughs  Year     mean_price    std_error
1276  Waltham Forest  2000  106384.166667  1675.302772
1277  Waltham Forest  2001  122711.583333  2701.200725
1278  Waltham Forest  2002  152959.083333  3813.146113
1279  Waltham Forest  2003  180774.250000   876.902876
1280  Waltham Forest  2004  197606.750000  1512.158377
    London_boroughs  Year     mean_price    std_error
966          Newham  2000   97050.916667  1672.353914
967          Newham  2001  113638.250000  2076.717231
968          Newham  2002  143616.916667  3772.803241
969          Newham  2003  175987.416667  1590.765184
970          Newham  2004  194003.416667  1923.681924
    London_boroughs  Year     mean_price    std_error
687       Islington  2000  192760.000000  2986.830508
688       Islington  2001  219962.166667  2945.134866
689       Islington  2002  247923.333333  3196.264143
690       Islington  2003  262256.750000  1902.011727
691       Islington  2004  286134.250000  2145.673802
```



[383]:
```
ax = bark_dag.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o',␣
    ↪color = "teal", figsize = (10,6), label = "Barking & Dagenham")
hamm_ful.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o', color␣
    ↪= "orange",ax = ax ,figsize = (10,6), label = "Hammersmith & Fulham")
enfield.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o', color␣
    ↪= "green",ax = ax ,figsize = (10,6), label = "enfield")
sutt.plot(x = 'Year', y = 'mean_price' , kind = "line", marker = 'o', color =␣
    ↪"red",ax = ax ,figsize = (10,6), label = "Sutton")
plt.xlabel('Year')
plt.ylabel('Average Price')
plt.title(f'Average House Price over years')
plt.xticks(range(2000, 2021, 1), rotation = 90)
plt.grid(True)

plt.show()
```



3. Modeling

Consider creating a function that will calculate a ratio of house prices, comparing the price of a house in 2018 to the price in 1998.

Consider calling this function create_price_ratio.

You'd want this function to: 1. Take a filter of dfg, specifically where this filter constrains the London_Borough, as an argument. For example, one admissible argument should be: dfg[dfg['London_Borough']=='Camden']. 2. Get the Average Price for that Borough, for the years 1998 and 2018. 4. Calculate the ratio of the Average Price for 1998 divided by the Average Price for 2018. 5. Return that ratio.

Once you've written this function, you ultimately want to use it to iterate through all the unique London_Boroughs and work out the ratio capturing the difference of house prices between 1998 and 2018.

Bear in mind: you don't have to write a function like this if you don't want to. If you can solve the brief otherwise, then great!

Hint: This section should test the skills you acquired in: - Python Data Science Toolbox - Part One, all modules

[291]:
```
def cal_price_ratio(desc_stat, London_boroughs, start_year, end_year):
    ratios = []
    for borough, group in desc_stat.groupby('London_boroughs'):
     start_price = group.loc[group['Year'] == start_year, 'Avg_price'].values
```

```
        end_price = group.loc[group['Year'] == end_year, 'Avg_price'].values
        price_ratio = start_price[0]/end_price[0]
        ratios.append((borough, price_ratio))
    return pd.DataFrame(ratios, columns = ['London_boroughs', 'price_ratio'])
```

[293]:

```
London_boroughs = desc_stat['London_boroughs']
ratio_df = cal_price_ratio(desc_stat, London_boroughs, start_year = 2000,␣
 ↪end_year = 2020)
#print(ratio_df)
filter_ratio_df = ratio_df.query("London_boroughs not in [ 'City of␣
 ↪London','SOUTH EAST','SOUTH WEST', 'WEST MIDLANDS', \
'YORKS & THE HUMBER',  'EAST MIDLANDS', 'EAST OF ENGLAND', 'NORTH EAST', 'NORTH␣
 ↪WEST','LONDON','Inner London',  'Outer London','England']")
filter_df = filter_ratio_df.sort_values(by = 'price_ratio', ascending = True)
print(filter_df.reset_index(drop = True))
```

```
          London_boroughs  price_ratio
0                 Hackney     0.227705
1          Waltham Forest     0.228332
2                  Newham     0.248508
3                Lewisham     0.253564
4      Barking & Dagenham     0.257780
5                Haringey     0.261997
6             Westminster     0.270863
7               Greenwich     0.277400
8               Southwark     0.277964
9                 Lambeth     0.281007
10   Kensington & Chelsea     0.285334
11                  Brent     0.289047
12                 Merton     0.289254
13              Islington     0.291334
14              Redbridge     0.294050
15               Havering     0.296365
16                 Camden     0.296453
17                 Bexley     0.300549
18             Wandsworth     0.302018
19          Tower Hamlets     0.303156
20                Enfield     0.307079
21                Croydon     0.317038
22                 Barnet     0.319449
23                 Ealing     0.319566
24             Hillingdon     0.320650
25               Hounslow     0.330958
26   Kingston upon Thames     0.331777
27                 Sutton     0.332547
28   Richmond upon Thames     0.334519
29                 Harrow     0.334861
```

```
30                Bromley     0.339794
31  Hammersmith & Fulham     0.340128
```

[306]:
```
filter_df.plot(x = "London_boroughs" , y = "price_ratio" , kind = "barh",␣
    ↪figsize = (10,6))
plt.title("Price ratio of london boroughs in 2000 and 2020")
plt.xlabel("Price ratio")
plt.ylabel("London boroughs")
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

1.2.3 4. Conclusion

What can you conclude? Type out your conclusion below.

Look back at your notebook. Think about how you might summarize what you have done, and prepare a quick presentation on it to your mentor at your next meeting.

We hope you enjoyed this practical project. It should have consolidated your data hygiene and pandas skills by looking at a real-world problem involving just the kind of dataset you might encounter as a budding data scientist. Congratulations, and looking forward to seeing you at the next step in the course!

[ ]:
```
"""Based on the price ratio calculated between the years 2000 and 2020, Hackney␣
    ↪and Waltham Forest have the lowest ratios,
```

```
0.2277 and 0.2283 respectively. Since a smaller price ratio indicates a greater␣
  ↪increase in average house price, these two boroughs
have experienced the largest growth in house prices over the past two decades.␣
  ↪"""
```