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

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)

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. 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

In [1]:
```
# Let's import the pandas, numpy libraries as pd, and np respectively.
import pandas as pd
import numpy as np

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

# Load the ticker collection from matplotlib for ticker fine tuning:
from matplotlib.ticker import FormatStrFormatter
```

1.2. Loading the data

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

In [2]:
```
#conda install -c anaconda openpyxl
```

In [3]:
```
# First, make a variable called url_LondonHousePrices, and assign it the fol
# https://data.london.gov.uk/download/uk-house-price-index/70ac0766-8902-4eb

url_LondonHousePrices = "https://data.london.gov.uk/download/uk-house-price-

# The dataset we're interested in contains the Average prices of the houses,
# As a result, we need to specify the sheet name in the read_excel() method.
# Put this data into a variable called properties.

# index_col=None: It means no column is used as the DataFrame's index. Inste
# index_col=1: This specifies that the first column (index 0) in the CSV fil
#properties = pd.read_excel(url_LondonHousePrices, sheet_name='Average price
properties = pd.read_excel(url_LondonHousePrices, sheet_name='Average price'
```

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.

In [4]:
```
properties.head(10)
```

Out[4]:

	City of London	Barking & Dagenham	Barnet	Bexley	Brent	Bror
NaT	E09000001	E09000002	E09000003	E09000004	E09000005	E09000
1995-01-01	91448.98487	50460.2266	93284.51832	64958.09036	71306.56698	81671.47
1995-02-01	82202.77314	51085.77983	93190.16963	64787.92069	72022.26197	81657.55
1995-03-01	79120.70256	51268.96956	92247.52435	64367.49344	72015.76274	81449.3
1995-04-01	77101.20804	53133.50526	90762.87492	64277.66881	72965.63094	81124.4
1995-05-01	84409.14932	53042.24852	90258.00033	63997.13588	73704.04743	81542.6
1995-06-01	94900.51244	53700.34831	90107.23471	64252.32335	74310.48167	82382.83
1995-07-01	110128.0423	52113.12157	91441.24768	63722.70055	74127.03788	82898.52
1995-08-01	112329.4376	52232.19868	92361.31512	64432.60005	73547.0411	82054.37
1995-09-01	104473.1096	51471.61353	93273.12245	64509.54767	73789.54287	81440.43

10 rows × 48 columns

In [5]:
```
properties.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 359 entries, NaT to 2024-10-01
Data columns (total 48 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   City of London        359 non-null    object 
 1   Barking & Dagenham    359 non-null    object 
 2   Barnet                359 non-null    object 
 3   Bexley                359 non-null    object 
 4   Brent                 359 non-null    object 
 5   Bromley               359 non-null    object 
 6   Camden                359 non-null    object 
 7   Croydon               359 non-null    object 
 8   Ealing                359 non-null    object 
 9   Enfield               359 non-null    object 
 10  Greenwich             359 non-null    object 
 11  Hackney               359 non-null    object 
 12  Hammersmith & Fulham  359 non-null    object 
 13  Haringey              359 non-null    object 
 14  Harrow                359 non-null    object 
 15  Havering              359 non-null    object 
 16  Hillingdon            359 non-null    object 
 17  Hounslow              359 non-null    object 
 18  Islington             359 non-null    object 
 19  Kensington & Chelsea  359 non-null    object 
 20  Kingston upon Thames  359 non-null    object 
 21  Lambeth               359 non-null    object 
 22  Lewisham              359 non-null    object 
 23  Merton                359 non-null    object 
 24  Newham                359 non-null    object 
 25  Redbridge             359 non-null    object 
 26  Richmond upon Thames  359 non-null    object 
 27  Southwark             359 non-null    object 
 28  Sutton                359 non-null    object 
 29  Tower Hamlets         359 non-null    object 
 30  Waltham Forest        359 non-null    object 
 31  Wandsworth            359 non-null    object 
 32  Westminster           359 non-null    object 
 33  Unnamed: 34           0 non-null      float64
 34  Inner London          359 non-null    object 
 35  Outer London          359 non-null    object 
 36  Unnamed: 37           0 non-null      float64
 37  NORTH EAST            359 non-null    object 
 38  NORTH WEST            359 non-null    object 
 39  YORKS & THE HUMBER    359 non-null    object 
 40  EAST MIDLANDS         359 non-null    object 
 41  WEST MIDLANDS         359 non-null    object 
 42  EAST OF ENGLAND       359 non-null    object 
 43  LONDON                359 non-null    object 
 44  SOUTH EAST            359 non-null    object 
 45  SOUTH WEST            359 non-null    object 
 46  Unnamed: 47           0 non-null      float64
 47  England               359 non-null    object 
dtypes: float64(3), object(45)
memory usage: 137.4+ KB
```

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.

In [6]:
```
properties_T = properties.transpose()
```

In [7]:
```
properties_T.head()
```

Out[7]:

	NaT	1995-01-01	1995-02-01	1995-03-01	1995-04-01	1995
City of London	E09000001	91448.98487	82202.77314	79120.70256	77101.20804	84409
Barking & Dagenham	E09000002	50460.2266	51085.77983	51268.96956	53133.50526	53042
Barnet	E09000003	93284.51832	93190.16963	92247.52435	90762.87492	90258
Bexley	E09000004	64958.09036	64787.92069	64367.49344	64277.66881	63997
Brent	E09000005	71306.56698	72022.26197	72015.76274	72965.63094	73704

5 rows × 359 columns

In [8]:
```
properties_T.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 48 entries, City of London to England
Columns: 359 entries, NaT to 2024-10-01
dtypes: object(359)
memory usage: 136.0+ KB
```

In [9]:
```
properties_T.shape
```

Out[9]:
```
(48, 359)
```

In [10]:
```
properties_T.columns.unique()
```

Out[10]:
```
DatetimeIndex([       'NaT', '1995-01-01', '1995-02-01', '1995-03-01',
               '1995-04-01', '1995-05-01', '1995-06-01', '1995-07-01',
               '1995-08-01', '1995-09-01',
               ...
               '2024-01-01', '2024-02-01', '2024-03-01', '2024-04-01',
               '2024-05-01', '2024-06-01', '2024-07-01', '2024-08-01',
               '2024-09-01', '2024-10-01'],
              dtype='datetime64[ns]', length=359, freq=None)
```

In [11]:
```
# Give Column[0] a new label name called "ID":
# properties_T.columns[0]: Gets the name of the first column.
# rename(columns={...}, inplace=True): Renames the first column to 'ID' in p

properties_T.rename(columns = {list(properties_T)[0]: 'ID'}, inplace = True)
```

In [12]:
```
# Check the top 5 rows of properties_T df:
properties_T.head()
```

Out[12]:

	ID	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 00
City of London	E09000001	91448.98487	82202.77314	79120.70256	77101.20804	84409
Barking & Dagenham	E09000002	50460.2266	51085.77983	51268.96956	53133.50526	53042
Barnet	E09000003	93284.51832	93190.16963	92247.52435	90762.87492	90258
Bexley	E09000004	64958.09036	64787.92069	64367.49344	64277.66881	63997
Brent	E09000005	71306.56698	72022.26197	72015.76274	72965.63094	73704

5 rows × 359 columns

In [13]:
```
# reset the index by calling .reset_index():
# The "Districts" - "City of London" column now becomes the index, making th
properties_T_1 = properties_T.reset_index()
```

In [14]:
```
properties_T_1.head()
```

Out[14]:

	index	ID	1995-01-01 00:00:00	1995-02-01 00:00:00	1995-03-01 00:00:00	1995-04-01 00:00:00	19
0	City of London	E09000001	91448.98487	82202.77314	79120.70256	77101.20804	844
1	Barking & Dagenham	E09000002	50460.2266	51085.77983	51268.96956	53133.50526	530
2	Barnet	E09000003	93284.51832	93190.16963	92247.52435	90762.87492	902
3	Bexley	E09000004	64958.09036	64787.92069	64367.49344	64277.66881	639
4	Brent	E09000005	71306.56698	72022.26197	72015.76274	72965.63094	737

5 rows × 360 columns

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...

In [15]:
```
# Give column[0] a new label name called "District":
# properties_T_1.rename(columns={properties_T_1.columns[0]: 'Districts'}, ir
properties_T_1.rename(columns = {list(properties_T_1)[0]: 'Districts'}, inpl
```

In [16]:
```
properties_T_1.head()
```

Out[16]:

	Districts	ID	1995-01-01 00:00:00	1995-02-01 00:00:00	1995-03-01 00:00:00	1995-04-01 00:00:00	19
0	City of London	E09000001	91448.98487	82202.77314	79120.70256	77101.20804	844
1	Barking & Dagenham	E09000002	50460.2266	51085.77983	51268.96956	53133.50526	530
2	Barnet	E09000003	93284.51832	93190.16963	92247.52435	90762.87492	902
3	Bexley	E09000004	64958.09036	64787.92069	64367.49344	64277.66881	63!
4	Brent	E09000005	71306.56698	72022.26197	72015.76274	72965.63094	737

5 rows × 360 columns

In [17]:
```
properties_T_1.tail()
```

Out[17]:

	Districts	ID	1995-01-01 00:00:00	1995-02-01 00:00:00	1995-03-01 00:00:00	1995-04-01 00:00:00	19
43	LONDON	E12000007	74435.76052	72777.93709	73896.84204	74455.28754	754
44	SOUTH EAST	E12000008	64018.87894	63715.02399	64113.60858	64623.22395	645
45	SOUTH WEST	E12000009	54705.1579	54356.14843	53583.07667	54786.01938	540
46	Unnamed: 47	NaN	NaN	NaN	NaN	NaN	
47	England	E92000001	53202.77128	53096.1549	53201.2843	53590.8548	53

5 rows × 360 columns

2.4.Transforming the data

Remember what Wes McKinney said about tidy data?

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

In [18]:
```
# Pivot table from longitude to latitude
clean_properties = pd.melt(properties_T_1, id_vars=['Districts', 'ID'], var_
```

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

In [19]:
```
clean_properties.head()
```

Out[19]:

	Districts	ID	Date	Price
0	City of London	E09000001	1995-01-01 00:00:00	91448.98487
1	Barking & Dagenham	E09000002	1995-01-01 00:00:00	50460.2266
2	Barnet	E09000003	1995-01-01 00:00:00	93284.51832
3	Bexley	E09000004	1995-01-01 00:00:00	64958.09036
4	Brent	E09000005	1995-01-01 00:00:00	71306.56698

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.

In [20]:
```
# Check the summary of clean_properties df:
clean_properties.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17184 entries, 0 to 17183
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   Districts  17184 non-null  object
 1   ID         16110 non-null  object
 2   Date       17184 non-null  object
 3   Price      16110 non-null  object
dtypes: object(4)
memory usage: 537.1+ KB
```

In [21]:
```
clean_properties.shape
```

Out[21]:
```
(17184, 4)
```

In [22]:
```
# As shown in the summary table, Price column are all object so this needs t
clean_properties_index = clean_properties.set_index(['Date', 'Districts', 'I
```

In [23]:
```
clean_properties_float = clean_properties_index.apply(pd.to_numeric)
```

In [24]:
```
clean_properties_float.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 17184 entries, (Timestamp('1995-01-01 00:00:00'), 'City of Londo
n', 'E09000001') to (Timestamp('2024-10-01 00:00:00'), 'England', 'E9200000
1')
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Price   16110 non-null  float64
dtypes: float64(1)
memory usage: 215.3+ KB
```

In [25]:
```
clean_properties_index.head()
```

Out[25]:

Date	Districts	ID	Price
1995-01-01	**City of London**	**E09000001**	91448.98487
	Barking & Dagenham	**E09000002**	50460.2266
	Barnet	**E09000003**	93284.51832
	Bexley	**E09000004**	64958.09036
	Brent	**E09000005**	71306.56698

In [27]:
```
# Reset the index:
clean_properties_f_1 = clean_properties_float.reset_index()
```

In [28]:
```
# Check the top 5 rows of clean_properties_f_1 df:
clean_properties_f_1.head()
```

Out[28]:

	Date	Districts	ID	Price
0	1995-01-01	City of London	E09000001	91448.98487
1	1995-01-01	Barking & Dagenham	E09000002	50460.22660
2	1995-01-01	Barnet	E09000003	93284.51832
3	1995-01-01	Bexley	E09000004	64958.09036
4	1995-01-01	Brent	E09000005	71306.56698

In [29]:
```
# Check the unique columns labels to see which columns are part of London Bo
clean_properties_f_1['Districts'].unique()
```

Out[29]:
```
array(['City of London', 'Barking & Dagenham', 'Barnet', 'Bexley',
       'Brent', 'Bromley', 'Camden', 'Croydon', 'Ealing', 'Enfield',
       'Greenwich', 'Hackney', 'Hammersmith & Fulham', 'Haringey',
       'Harrow', 'Havering', 'Hillingdon', 'Hounslow', 'Islington',
       'Kensington & Chelsea', 'Kingston upon Thames', 'Lambeth',
       'Lewisham', 'Merton', 'Newham', 'Redbridge',
       'Richmond upon Thames', 'Southwark', 'Sutton', 'Tower Hamlets',
       'Waltham Forest', 'Wandsworth', 'Westminster', 'Unnamed: 34',
       'Inner London', 'Outer London', 'Unnamed: 37', 'NORTH EAST',
       'NORTH WEST', 'YORKS & THE HUMBER', 'EAST MIDLANDS',
       'WEST MIDLANDS', 'EAST OF ENGLAND', 'LONDON', 'SOUTH EAST',
       'SOUTH WEST', 'Unnamed: 47', 'England'], dtype=object)
```

In [30]:
```python
# Assign Non-London-Borough to the Boroughs which are not part of london Bor
Non_London_Borough = ['City of London', 'Inner London', 'Outer London', 'NOR
       'NORTH WEST', 'YORKS & THE HUMBER', 'EAST MIDLANDS',
       'WEST MIDLANDS', 'EAST OF ENGLAND', 'LONDON', 'SOUTH EAST',
       'SOUTH WEST', 'England']
```

In [31]:
```python
# Drop Non_London_Borough from clean_properties_f_1 using .isin() and keepir
clean_properties_f_2 = clean_properties_f_1[~clean_properties_f_1.Districts.
```

In [32]:
```python
# Confirm that we have now only the London Borough = 32
clean_properties_f_2['Districts'].unique()
```

Out[32]:
```
array(['Barking & Dagenham', 'Barnet', 'Bexley', 'Brent', 'Bromley',
       'Camden', 'Croydon', 'Ealing', 'Enfield', 'Greenwich', 'Hackney',
       'Hammersmith & Fulham', 'Haringey', 'Harrow', 'Havering',
       'Hillingdon', 'Hounslow', 'Islington', 'Kensington & Chelsea',
       'Kingston upon Thames', 'Lambeth', 'Lewisham', 'Merton', 'Newham',
       'Redbridge', 'Richmond upon Thames', 'Southwark', 'Sutton',
       'Tower Hamlets', 'Waltham Forest', 'Wandsworth', 'Westminster',
       'Unnamed: 34', 'Unnamed: 37', 'Unnamed: 47'], dtype=object)
```

In [33]:
```python
# Check the NaN columns:
clean_properties_f_2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 12530 entries, 1 to 17182
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       12530 non-null  datetime64[ns]
 1   Districts  12530 non-null  object
 2   ID         11456 non-null  object
 3   Price      11456 non-null  float64
dtypes: datetime64[ns](1), float64(1), object(2)
memory usage: 489.5+ KB
```

In [34]:
```python
# Drops all NaN columns using .dropna() method:
clean_properties_drop_NaN = clean_properties_f_2.dropna(how = 'any')
```

In [35]:
```python
# Confirm that we have no NaN columns & we have only 32 boroughs:
clean_properties_drop_NaN['Districts'].unique()
```

Out[35]:
```
array(['Barking & Dagenham', 'Barnet', 'Bexley', 'Brent', 'Bromley',
       'Camden', 'Croydon', 'Ealing', 'Enfield', 'Greenwich', 'Hackney',
       'Hammersmith & Fulham', 'Haringey', 'Harrow', 'Havering',
       'Hillingdon', 'Hounslow', 'Islington', 'Kensington & Chelsea',
       'Kingston upon Thames', 'Lambeth', 'Lewisham', 'Merton', 'Newham',
       'Redbridge', 'Richmond upon Thames', 'Southwark', 'Sutton',
       'Tower Hamlets', 'Waltham Forest', 'Wandsworth', 'Westminster'],
      dtype=object)
```

In [36]:
```
# Assign clean_properties_drop_NaN to df:
df = clean_properties_drop_NaN
df.head()
```

Out[36]:

	Date	Districts	ID	Price
1	1995-01-01	Barking & Dagenham	E09000002	50460.22660
2	1995-01-01	Barnet	E09000003	93284.51832
3	1995-01-01	Bexley	E09000004	64958.09036
4	1995-01-01	Brent	E09000005	71306.56698
5	1995-01-01	Bromley	E09000006	81671.47692

In [37]:
```
# Let's rename 'Date' as 'Month' and 'Price' as 'Average_Price'
df.rename(columns = {list(df)[0]: 'Month', list(df)[3]:'Average_Price'}, inp
```

```
/var/folders/qn/7_ssbz356pz3h_klfys0777m0000gn/T/ipykernel_1665/601508168.p
y:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df.rename(columns = {list(df)[0]: 'Month', list(df)[3]:'Average_Price'}, i
nplace = True)
```

In [38]:
```
df.head()
```

Out[38]:

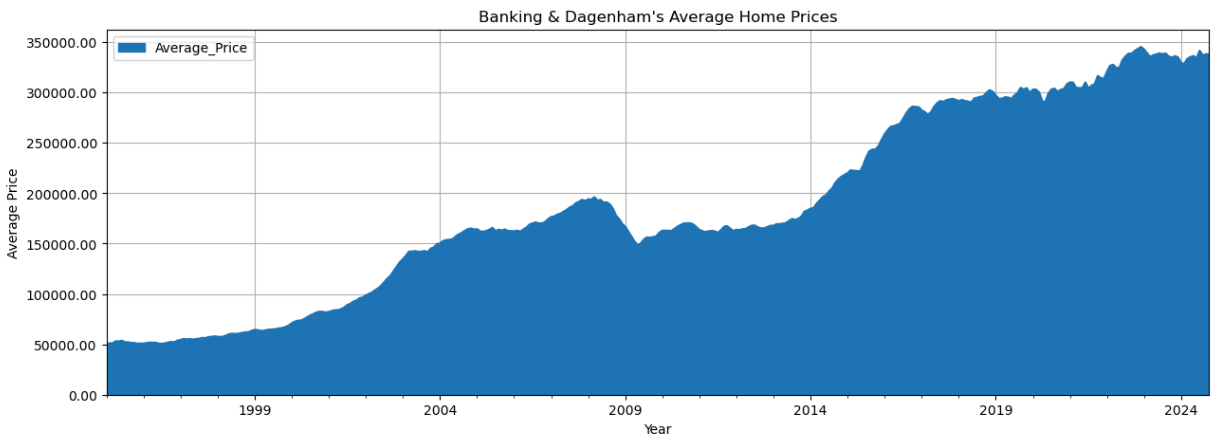
	Month	Districts	ID	Average_Price
1	1995-01-01	Barking & Dagenham	E09000002	50460.22660
2	1995-01-01	Barnet	E09000003	93284.51832
3	1995-01-01	Bexley	E09000004	64958.09036
4	1995-01-01	Brent	E09000005	71306.56698
5	1995-01-01	Bromley	E09000006	81671.47692

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?

In [39]:
```
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter

ax1 = df[df['Districts'] == "Barking & Dagenham"].plot(kind='area', x='Month
plt.title("Banking & Dagenham's Average Home Prices")
plt.ylabel("Average Price")
plt.xlabel("Year")
# Ensures that the grid lines appear below the plot elements for better clar
ax1.set_axisbelow(True)
ax1.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))
plt.show()
```



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?

In [40]:
```
df['Year'] = df['Month'].apply(lambda t: t.year)
```

```
/var/folders/qn/7_ssbz356pz3h_klfys0777m0000gn/T/ipykernel_1665/3140778947.p
y: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
  df['Year'] = df['Month'].apply(lambda t: t.year)
```

In [41]:
```
df.head()
```

Out[41]:

	Month	Districts	ID	Average_Price	Year
1	1995-01-01	Barking & Dagenham	E09000002	50460.22660	1995
2	1995-01-01	Barnet	E09000003	93284.51832	1995
3	1995-01-01	Bexley	E09000004	64958.09036	1995
4	1995-01-01	Brent	E09000005	71306.56698	1995
5	1995-01-01	Bromley	E09000006	81671.47692	1995

In [42]:

```
# Using the function 'groupby' will help you to calculate the mean for each
## The variables Borough and Year are now indexs
dfg = df.groupby(by=['Districts', 'Year'])['Average_Price'].mean()
dfg.sample(10)
```

Out[42]:

```
Districts              Year
Kensington & Chelsea   1995    192857.260633
Camden                 2010    513221.129450
Haringey               2012    339685.912767
Sutton                 2011    233026.678975
Camden                 1996    133810.487933
Redbridge              1996     75358.658939
Newham                 1997     60971.380317
Enfield                2008    245839.333108
Westminster            1995    133689.233033
Croydon                2008    235998.602192
Name: Average_Price, dtype: float64
```

In [43]:

```
dfg.head()
```

Out[43]:

```
Districts           Year
Barking & Dagenham  1995    51817.969390
                    1996    51718.192690
                    1997    55974.262309
                    1998    60285.821083
                    1999    65320.934441
Name: Average_Price, dtype: float64
```

In [44]:

```
dfg = dfg.reset_index()
dfg.head()
```

Out[44]:

	Districts	Year	Average_Price
0	Barking & Dagenham	1995	51817.969390
1	Barking & Dagenham	1996	51718.192690
2	Barking & Dagenham	1997	55974.262309
3	Barking & Dagenham	1998	60285.821083
4	Barking & Dagenham	1999	65320.934441

In [45]:
```
# Slicing dfg to have more understanding of the df:
dfg[(dfg['Districts'] == 'Kensington & Chelsea') & (dfg['Year'] == 1995)]
```

Out[45]:

	Districts	Year	Average_Price
540	Kensington & Chelsea	1995	192857.260633

In [46]:
```
dfg[(dfg['Districts'] == 'Kensington & Chelsea') & (dfg['Year'] == 2018)]
```

Out[46]:

	Districts	Year	Average_Price
563	Kensington & Chelsea	2018	1.363802e+06

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.
3. Calculate the ratio of the Average Price for 1998 divided by the Average Price for 2018.
4. 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

In [47]:
```
# Here's where you should write your function:
def create_price_ratio(d):
    y1998 = float(d['Average_Price'][d['Year'] == 1998])
    y2018 = float(d['Average_Price'][d['Year'] == 2018])
    ratio = [y1998/y2018]

    # ratio: The entire list.
```

```
    # ratio[0]: The first element of the list.
    return ratio[0]
```

In [48]:
```
# Test out the function by calling it with the following argument:
# dfg[dfg['London_Borough'] == 'Barking & Dagenham']
ratio = create_price_ratio(dfg[dfg['Districts'] == 'Barking & Dagenham']) *
rounded_perc = round(ratio, 2)
print(rounded_perc, "%")
```

```
20.42 %
```

```
/var/folders/qn/7_ssbz356pz3h_klfys0777m0000gn/T/ipykernel_1665/2958057950.p
y:3: FutureWarning: Calling float on a single element Series is deprecated a
nd will raise a TypeError in the future. Use float(ser.iloc[0]) instead
  y1998 = float(d['Average_Price'][d['Year'] == 1998])
/var/folders/qn/7_ssbz356pz3h_klfys0777m0000gn/T/ipykernel_1665/2958057950.p
y:4: FutureWarning: Calling float on a single element Series is deprecated a
nd will raise a TypeError in the future. Use float(ser.iloc[0]) instead
  y2018 = float(d['Average_Price'][d['Year'] == 2018])
```

In [49]:
```
dfg[dfg['Districts']=='Barking & Dagenham']
```

Out[49]:

	Districts	Year	Average_Price
0	Barking & Dagenham	1995	51817.969390
1	Barking & Dagenham	1996	51718.192690
2	Barking & Dagenham	1997	55974.262309
3	Barking & Dagenham	1998	60285.821083
4	Barking & Dagenham	1999	65320.934441
5	Barking & Dagenham	2000	77549.513290
6	Barking & Dagenham	2001	88664.058223
7	Barking & Dagenham	2002	112221.912482
8	Barking & Dagenham	2003	142498.927800
9	Barking & Dagenham	2004	158175.982483
10	Barking & Dagenham	2005	163360.782017
11	Barking & Dagenham	2006	167853.342558
12	Barking & Dagenham	2007	184909.807383
13	Barking & Dagenham	2008	187356.865783
14	Barking & Dagenham	2009	156446.896358
15	Barking & Dagenham	2010	166560.705275
16	Barking & Dagenham	2011	163465.144225
17	Barking & Dagenham	2012	165863.911600
18	Barking & Dagenham	2013	173733.624933
19	Barking & Dagenham	2014	201172.229417
20	Barking & Dagenham	2015	233460.107425
21	Barking & Dagenham	2016	273919.636042
22	Barking & Dagenham	2017	287734.717358
23	Barking & Dagenham	2018	295185.125625
24	Barking & Dagenham	2019	298207.102333
25	Barking & Dagenham	2020	300836.133975
26	Barking & Dagenham	2021	309046.083333
27	Barking & Dagenham	2022	333405.083333
28	Barking & Dagenham	2023	337535.333333
29	Barking & Dagenham	2024	335233.200000

In [50]:
```
# We want to do this for all the London Boroughs.
# First, let's make an empty dictionary, called final, where we'll store our
final = {}
```

In [51]:
```
# Now let's declare a for loop that will iterate through each of the unique
# Call the iterator variable 'b'

for b in dfg['Districts'].unique():
    # subset dfg on 'London_Borough' == b
    borough = dfg[dfg['Districts'] == b]
    # Make a new entry in the final dictionary whose value's the result of c
    final[b] = create_price_ratio(borough) * 100

print(final)
```

```
{'Barking & Dagenham': 20.42305517789531, 'Barnet': 22.947444714139664, 'Bex
ley': 23.530326956746084, 'Brent': 20.427076722400113, 'Bromley': 24.4209451
18598183, 'Camden': 20.267350201601634, 'Croydon': 23.803943067256167, 'Eali
ng': 23.19231929467808, 'Enfield': 23.459043566245953, 'Greenwich': 20.99267
632074863, 'Hackney': 16.133361584589345, 'Hammersmith & Fulham': 24.1607774
1787779, 'Haringey': 19.475892515252134, 'Harrow': 24.635696011145082, 'Have
ring': 23.116742430256252, 'Hillingdon': 23.80791298036054, 'Hounslow': 25.1
44117659399683, 'Islington': 20.653143718863255, 'Kensington & Chelsea': 19.
676616601314997, 'Kingston upon Thames': 23.419000399105975, 'Lambeth': 20.1
68631676941153, 'Lewisham': 18.35565611520636, 'Merton': 21.074143727135688,
'Newham': 18.840708833606374, 'Redbridge': 22.370414627844575, 'Richmond upo
n Thames': 24.982762338287166, 'Southwark': 18.127858000786993, 'Sutton': 2
4.278518539792014, 'Tower Hamlets': 21.61351135145911, 'Waltham Forest': 17.
13758725975514, 'Wandsworth': 21.019112262781604, 'Westminster': 18.68205056
782318}
```

```
/var/folders/qn/7_ssbz356pz3h_klfys0777m0000gn/T/ipykernel_1665/2958057950.p
y:3: FutureWarning: Calling float on a single element Series is deprecated a
nd will raise a TypeError in the future. Use float(ser.iloc[0]) instead
  y1998 = float(d['Average_Price'][d['Year'] == 1998])
/var/folders/qn/7_ssbz356pz3h_klfys0777m0000gn/T/ipykernel_1665/2958057950.p
y:4: FutureWarning: Calling float on a single element Series is deprecated a
nd will raise a TypeError in the future. Use float(ser.iloc[0]) instead
  y2018 = float(d['Average_Price'][d['Year'] == 2018])
```

In [52]:
```
# series = pd.Series(final, name = 'Ratio')
# df_ratios = series.to_frame()

df_ratios = pd.DataFrame.from_dict(final, orient='index', columns=['Ratio'])
df_ratios.index.name = 'District'

# Calculate the ratio
df_ratios['Ratio'] = df_ratios['Ratio'].apply(lambda x: f"{x: .2f}%")

df_ratios.sort_values(by='Ratio', ascending=False, inplace=True)

print(df_ratios)
```

```
                         Ratio
District
Hounslow                25.14%
Richmond upon Thames    24.98%
Harrow                  24.64%
Bromley                 24.42%
Sutton                  24.28%
Hammersmith & Fulham    24.16%
Hillingdon              23.81%
Croydon                 23.80%
Bexley                  23.53%
Enfield                 23.46%
Kingston upon Thames    23.42%
Ealing                  23.19%
Havering                23.12%
Barnet                  22.95%
Redbridge               22.37%
Tower Hamlets           21.61%
Merton                  21.07%
Wandsworth              21.02%
Greenwich               20.99%
Islington               20.65%
Brent                   20.43%
Barking & Dagenham      20.42%
Camden                  20.27%
Lambeth                 20.17%
Kensington & Chelsea    19.68%
Haringey                19.48%
Newham                  18.84%
Westminster             18.68%
Lewisham                18.36%
Southwark               18.13%
Waltham Forest          17.14%
Hackney                 16.13%
```

In [53]:

```
df_ratios = pd.DataFrame(df_ratios)
df_ratios
```

Out[53]:

District	Ratio
Hounslow	25.14%
Richmond upon Thames	24.98%
Harrow	24.64%
Bromley	24.42%
Sutton	24.28%
Hammersmith & Fulham	24.16%
Hillingdon	23.81%
Croydon	23.80%
Bexley	23.53%
Enfield	23.46%
Kingston upon Thames	23.42%
Ealing	23.19%
Havering	23.12%
Barnet	22.95%
Redbridge	22.37%
Tower Hamlets	21.61%
Merton	21.07%
Wandsworth	21.02%
Greenwich	20.99%
Islington	20.65%
Brent	20.43%
Barking & Dagenham	20.42%
Camden	20.27%
Lambeth	20.17%
Kensington & Chelsea	19.68%
Haringey	19.48%
Newham	18.84%
Westminster	18.68%
Lewisham	18.36%
Southwark	18.13%
Waltham Forest	17.14%

	Ratio
District	
Hackney	16.13%

In [54]:
```
# # Remove the '%' symbol
# df_ratios['Ratio'] = df_ratios['Ratio'].str.replace('%', '')

# # Convert the 'Ratio' column to numeric
# df_ratios['Ratio'] = pd.to_numeric(df_ratios['Ratio'])

# # Now you can plot the bar chart
# df_ratios.plot.bar(figsize=(12, 6), rot=45)
```

In [55]:
```
print(df_ratios['Ratio'].dtype)
```

```
object
```

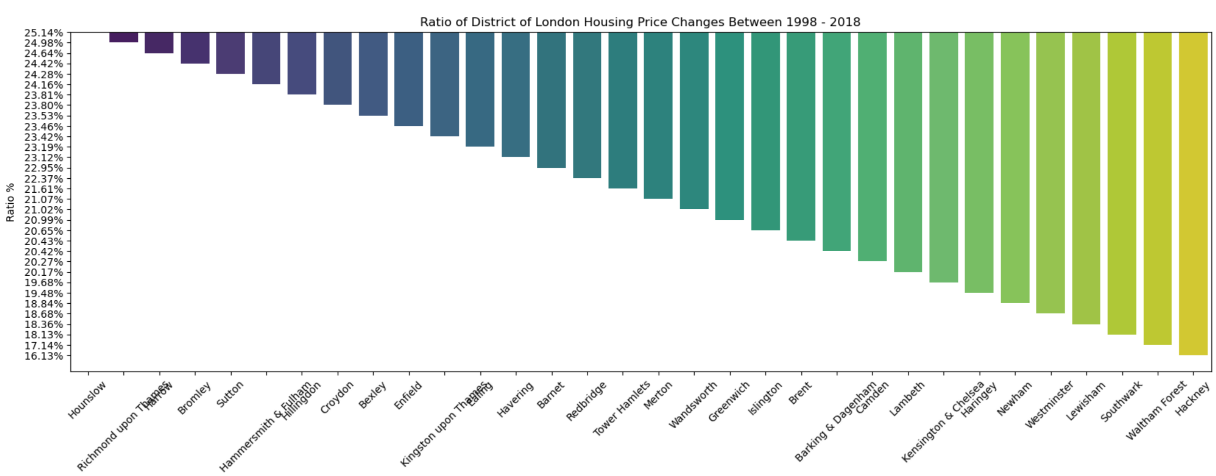
In [125…
```
# df_ratios.plot.bar(figsize=(20, 6), rot=45, cmap='viridis')
# plt.title('Ratio by District')
# plt.xlabel('District')
# plt.ylabel('Ratio %')
# plt.show()
```

In [56]:
```
import warnings
import seaborn as sns

warnings.simplefilter(action='ignore', category=FutureWarning)

plt.figure(figsize=(20, 6))

sns.barplot(x=df_ratios.index, y='Ratio', data=df_ratios, palette='viridis')
plt.xticks(rotation=45)
plt.title('Ratio of District of London Housing Price Changes Between 1998 -
plt.ylabel('Ratio %')
plt.show()
```



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!

The bar plot effectively visualizes the relative changes in housing prices across different London districts. The height of each bar represents the ratio of the average house price in 2018 to the average price in 1998. Districts with higher bars experienced greater price increases.

Key Insights:

- Hounslow and Richmond upon Thames had the highest price increases, indicating significant growth in these areas.
- Hackney and Waltham Forest had the lowest price increases, suggesting relatively slower growth.

Historical data indicates a consistent upward trend in average housing prices across most London boroughs, including Barking & Dagenham.

In [57]:
```
dfg[dfg['Districts'] == "Barking & Dagenham"]
```

Out[57]:

	Districts	Year	Average_Price
0	Barking & Dagenham	1995	51817.969390
1	Barking & Dagenham	1996	51718.192690
2	Barking & Dagenham	1997	55974.262309
3	Barking & Dagenham	1998	60285.821083
4	Barking & Dagenham	1999	65320.934441
5	Barking & Dagenham	2000	77549.513290
6	Barking & Dagenham	2001	88664.058223
7	Barking & Dagenham	2002	112221.912482
8	Barking & Dagenham	2003	142498.927800
9	Barking & Dagenham	2004	158175.982483
10	Barking & Dagenham	2005	163360.782017
11	Barking & Dagenham	2006	167853.342558
12	Barking & Dagenham	2007	184909.807383
13	Barking & Dagenham	2008	187356.865783
14	Barking & Dagenham	2009	156446.896358
15	Barking & Dagenham	2010	166560.705275
16	Barking & Dagenham	2011	163465.144225
17	Barking & Dagenham	2012	165863.911600
18	Barking & Dagenham	2013	173733.624933
19	Barking & Dagenham	2014	201172.229417
20	Barking & Dagenham	2015	233460.107425
21	Barking & Dagenham	2016	273919.636042
22	Barking & Dagenham	2017	287734.717358
23	Barking & Dagenham	2018	295185.125625
24	Barking & Dagenham	2019	298207.102333
25	Barking & Dagenham	2020	300836.133975
26	Barking & Dagenham	2021	309046.083333
27	Barking & Dagenham	2022	333405.083333
28	Barking & Dagenham	2023	337535.333333
29	Barking & Dagenham	2024	335233.200000

In [58]:
```
avg_price = dfg.set_index('Year')
pivot = avg_price.pivot_table(index='Year', columns='Districts', values='Ave
pivot.head()
```

Out[58]:

Districts	Barking & Dagenham	Barnet	Bexley	Brent	Bromley
Year					
1995	51817.969390	91792.537433	64291.532845	73029.841840	81967.316732
1996	51718.192690	94000.445448	65490.417234	75235.918367	83547.483633
1997	55974.262309	106883.185546	70789.406602	86749.070663	94224.688035
1998	60285.821083	122359.468033	80632.020822	100692.590417	108286.520467
1999	65320.934441	136004.512067	86777.715903	112157.469808	120874.179567

5 rows × 32 columns

In [86]:
```
# df
df_avgP = df.groupby(by=['Districts'])['Average_Price'].mean().sort_values(a
top15_avgP = df_avgP.head(15)
top15_avgP

top15 = top15_avgP.index.tolist()

df_top15 = df[df['Districts'].isin(top15)]
df_top15
```

Out[86]:

	Month	Districts	ID	Average_Price	Year
2	1995-01-01	Barnet	E09000003	93284.51832	1995
6	1995-01-01	Camden	E09000007	120932.88810	1995
8	1995-01-01	Ealing	E09000009	79885.89069	1995
11	1995-01-01	Hackney	E09000012	61296.52637	1995
12	1995-01-01	Hammersmith & Fulham	E09000013	124902.86020	1995
...	...	...	...	...	...
17159	2024-10-01	Merton	E09000024	579347.00000	2024
17162	2024-10-01	Richmond upon Thames	E09000027	738612.00000	2024
17163	2024-10-01	Southwark	E09000028	504782.00000	2024
17167	2024-10-01	Wandsworth	E09000032	606527.00000	2024
17168	2024-10-01	Westminster	E09000033	959769.00000	2024

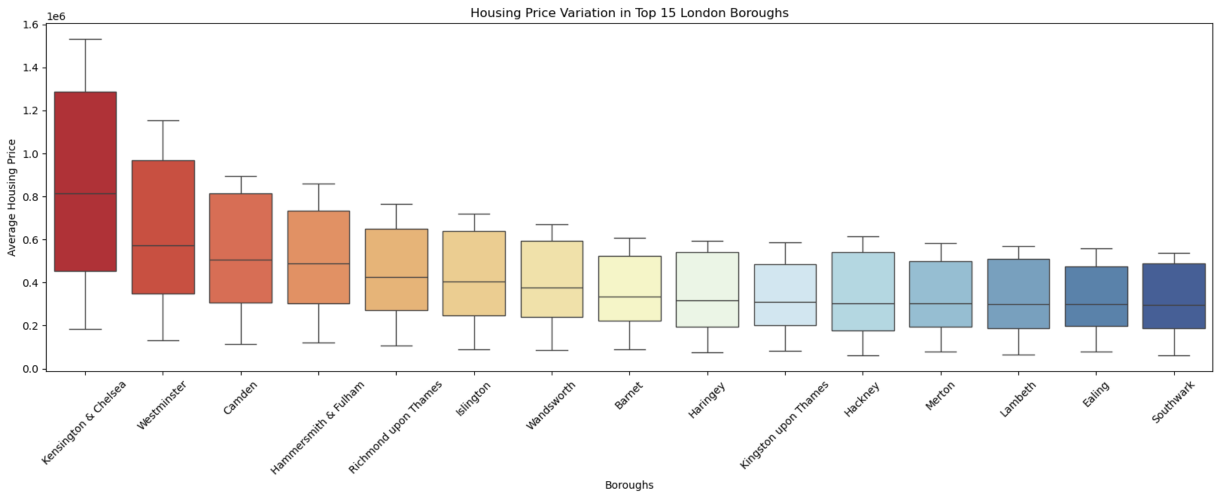
5370 rows × 5 columns

In [93]:
```
medianP = df_top15.groupby('Districts')['Average_Price'].median().sort_value
medianP_districts = medianP.index.tolist()
```

In [103…
```
plt.figure(figsize=(20, 6))

sns.boxplot(x='Districts', y='Average_Price', data=df_top15, palette='RdYlBu
plt.xticks(rotation = 45)
plt.xlabel('Boroughs')
plt.ylabel('Average Housing Price')
plt.title("Housing Price Variation in Top 15 London Boroughs")

plt.show()
```



Out of all 32 London boroughs total, these boroughs are the top 15 ones. The boxplot visually confirms that there is a wide vairation in average housing prices across the top 15 London boroughs. The boroughs are ordered from left to right based on their median housing prices, with Kensington & Chelsea having the highest median and Southwark having the lowest.