1 King County House Sales Analysis

1.1 Introduction

Here we will be looking into the King County House Sales dataset to find information on how home renovations might increase the estimated value of homes (and by what amount) for the magazine 'Home Owners Yearly', who wants to put out an article on what renovations will or will not be likely to improve the value of middle class and upper middle class homes.

In order to do this, we will be looking at a data set on houses and housing prices from King County in Washington State (https://en.wikipedia.org/wiki/King_County,_Washington).

The dataset covers alot of information, but the magazine gave us a few questions to focus in on.

- Will increasing the living area size lead to an associated increase in the value of the home?
- Will adding bedrooms or bathrooms lead to an associated increase in the value of the home?
- Is the grade or condition rating of the house associated with the value of the home?

In [1]:

```
# importing required packages
import warnings
import zipfile
import seaborn as sns
import pandas as pd
import numpy as np
import pylab
import scipy.stats as stats
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.formula.api import ols
%matplotlib inline
warnings.filterwarnings("ignore")
```

1.2 Looking at the dataset

1.2.1 Limitations of Dataset

There are some limitations inherent to this dataset. First and foremost, this dataset is all from King County, WA. This is a fairly affluent and densely populated area (Wikipedia page) (https://en.wikipedia.org/wiki/King_County,_Washington), and as such the recommendations and conclusions from this data may not hold true for other areas with different characteristics (e.g. rural areas). More information and analysis is necessary to determine what neighborhoods and counties can use these recommendations.

Additionally, there are many types of renovations that aren't included in the dataset (e.g. renovating the plumbing, new roof, adding a deck, ect.), which limits the specificity of the recommendations.

1.2.2 Why We Used This Dataset

Despite the above limitations, this dataset does represent a middle and upper class neighborhood, which is the demographic that the magazine is trying to appeal to. It does contain the information on bedrooms and bathrooms (which were some of the magazines specific questions that they wanted answers to) and was easily available.

In [2]:
```python
# lets take an initial look at the data in the `kc_house_data.csv` dataset
df = pd.read_csv('Data/kc_house_data.csv')
df.head()
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	NaN	NONE	...	7 Average	1180
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	NO	NONE	...	7 Average	2170
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	NO	NONE	...	6 Low Average	770
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	NO	NONE	...	7 Average	1050
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	NO	NONE	...	8 Good	1680

5 rows × 21 columns

In [3]:
```python
# So above we see there are 21 columns, and it looks like 'price'
# may be a good contender for our dependant variable, as we
# want to know what improvements will increase the selling price
# of a home.
```

In [4]:
```python
# looking into the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             21597 non-null  int64  
 1   date           21597 non-null  object 
 2   price          21597 non-null  float64
 3   bedrooms       21597 non-null  int64  
 4   bathrooms      21597 non-null  float64
 5   sqft_living    21597 non-null  int64  
 6   sqft_lot       21597 non-null  int64  
 7   floors         21597 non-null  float64
 8   waterfront     19221 non-null  object 
 9   view           21534 non-null  object 
 10  condition      21597 non-null  object 
 11  grade          21597 non-null  object 
 12  sqft_above     21597 non-null  int64  
 13  sqft_basement  21597 non-null  object 
 14  yr_built       21597 non-null  int64  
 15  yr_renovated   17755 non-null  float64
 16  zipcode        21597 non-null  int64  
 17  lat            21597 non-null  float64
 18  long           21597 non-null  float64
 19  sqft_living15  21597 non-null  int64  
 20  sqft_lot15     21597 non-null  int64  
dtypes: float64(6), int64(9), object(6)
memory usage: 3.5+ MB
```

In [5]:
```
df.describe()
```

Out[5]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	sqft_above	yr_
count	2.159700e+04	2.159700e+04	21597.000000	21597.000000	21597.000000	2.159700e+04	21597.000000	21597.000000	21597.00
mean	4.580474e+09	5.402966e+05	3.373200	2.115826	2080.321850	1.509941e+04	1.494096	1788.596842	1970.99
std	2.876736e+09	3.673681e+05	0.926299	0.768984	918.106125	4.141264e+04	0.539683	827.759761	29.37
min	1.000102e+06	7.800000e+04	1.000000	0.500000	370.000000	5.200000e+02	1.000000	370.000000	1900.00
25%	2.123049e+09	3.220000e+05	3.000000	1.750000	1430.000000	5.040000e+03	1.000000	1190.000000	1951.00
50%	3.904930e+09	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	1560.000000	1975.00
75%	7.308900e+09	6.450000e+05	4.000000	2.500000	2550.000000	1.068500e+04	2.000000	2210.000000	1997.00
max	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	9410.000000	2015.00

In [6]:
```
df.shape
```

Out[6]:
```
(21597, 21)
```

1.2.2.1 Dataset Size

So we see above that starting off we have 21 columns, and 21,597 rows (each representing a different home) in total.

1.3 Preprocessing

1.3.1 Check for missing values

In [7]:
```
df.isnull().sum()
```

Out[7]:
```
id                  0
date                0
price               0
bedrooms            0
bathrooms           0
sqft_living         0
sqft_lot            0
floors              0
waterfront       2376
view               63
condition           0
grade               0
sqft_above          0
sqft_basement       0
yr_built            0
yr_renovated     3842
zipcode             0
lat                 0
long                0
sqft_living15       0
sqft_lot15          0
dtype: int64
```

Lets peek into the three columns with NaN data, starting with the `waterfront` data:

In [8]:
```
df['waterfront'].value_counts()
```

Out[8]:
```
NO     19075
YES      146
Name: waterfront, dtype: int64
```

Seems like we could recode these NaN's as NO - it could be that the NaN's are in areas where being on the waterfront isn't possible? Regardless, it's improbable that a homeowner or could change the location of a home to improve the homes value, but having a waterfront propety could affect the value of the renovations, so we'll replace all the NaN's with `NO` .

In [9]:
```
df.waterfront.replace({np.nan: 'NO'}, inplace=True)
df['waterfront'].value_counts()
# df.head()
```

Out[9]:
```
NO     21451
YES      146
Name: waterfront, dtype: int64
```

Now let's deal with the NaN's in `view`. Here we are not missing so many, so we could just drop those rows completely from the dataset, but let's peak into the data and see if we can convert the NaN's into another option instead.

In [10]:
```
df['view'].value_counts()
```

Out[10]:
```
NONE         19422
AVERAGE        957
GOOD           508
FAIR           330
EXCELLENT      317
Name: view, dtype: int64
```

As we have the NONE value category (which makes up most of the data) we can just convert the NaN's into NONE's.

In [11]:
```
df.view.replace({np.nan: 'NONE'}, inplace=True)
df['view'].value_counts()
```

Out[11]:
```
NONE         19485
AVERAGE        957
GOOD           508
FAIR           330
EXCELLENT      317
Name: view, dtype: int64
```

Finally, lets look at `yr_renovated`.

In [12]:
```
df['yr_renovated'].value_counts()
```

Out[12]:
```
0.0       17011
2014.0       73
2003.0       31
2013.0       31
2007.0       30
          ...
1946.0        1
1959.0        1
1971.0        1
1951.0        1
1954.0        1
Name: yr_renovated, Length: 70, dtype: int64
```

While we are looking at renovations, we are less interested in past renovations, and more concerned with future improvements we can do, but knowing when the last renovations were may still be usefull data. There already seems to be a missing data value (0.0) so we'll replace all our NaN's with that.

In [13]:
```python
df.yr_renovated.replace({np.nan: 0.0}, inplace=True)
df['yr_renovated'].value_counts()
# df.head()
```

Out[13]:
```
0.0       20853
2014.0       73
2003.0       31
2013.0       31
2007.0       30
          ...
1946.0        1
1959.0        1
1971.0        1
1951.0        1
1954.0        1
Name: yr_renovated, Length: 70, dtype: int64
```

1.3.2 Dropping Columns

When we looked into the NaN values, we saw that there are some columns that are irrelevant to the buisness question we are trying to answer.

- (Just a reminder of the question: How could home renovations possibly increase the estimated value of homes?)

As such, we'll quickly peek into the `column_names.md.txt` file, to see what the column names mean and see if there are any more we can drop.

Here's a copy-paste of the information in `column_names.md.txt` :

1.3.2.1 Column Names and Descriptions for King County Data Set

- `id` - Unique identifier for a house
- `date` - Date house was sold
- `price` - Sale price (prediction target)
- `bedrooms` - Number of bedrooms
- `bathrooms` - Number of bathrooms
- `sqft_living` - Square footage of living space in the home
- `sqft_lot` - Square footage of the lot
- `floors` - Number of floors (levels) in house
- `waterfront` - Whether the house is on a waterfront
  - Includes Duwamish, Elliott Bay, Puget Sound, Lake Union, Ship Canal, Lake Washington, Lake Sammamish, other lake, and river/slough waterfronts
- `view` - Quality of view from house
  - Includes views of Mt. Rainier, Olympics, Cascades, Territorial, Seattle Skyline, Puget Sound, Lake Washington, Lake Sammamish, small lake / river / creek, and other
- `condition` - How good the overall condition of the house is. Related to maintenance of house.
  - See the King County Assessor Website (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r) for further explanation of each condition code
- `grade` - Overall grade of the house. Related to the construction and design of the house.
  - See the King County Assessor Website (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r) for further explanation of each building grade code
- `sqft_above` - Square footage of house apart from basement
- `sqft_basement` - Square footage of the basement
- `yr_built` - Year when house was built
- `yr_renovated` - Year when house was renovated
- `zipcode` - ZIP Code used by the United States Postal Service
- `lat` - Latitude coordinate
- `long` - Longitude coordinate
- `sqft_living15` - The square footage of interior housing living space for the nearest 15 neighbors
- `sqft_lot15` - The square footage of the land lots of the nearest 15 neighbors

- We should also probably drop `id` (as we don't need to know specific houses identifiers), `date` (as it's not important when the house was last sold), `lat` , `long` , and `zipcode` (as we can't change where the house is located).
- The rest of the categories are things that could possibly be changed in the suggested renovations (e.g. you could add on another bedroom, which would change the value in `bedrooms`) or may have implications for the renovations (e.g. knowing when the house was built could affect the renovations.)
- The `yr_renovated` and `yr_built` aren't directly related to our questions, and will likely add a lot of bulk/noise to our dataset. Additionally, we aren't asking questions about the lot sizes (or basement sizes), so lets drop `sqft_lot` , `sqft_above` , `sqft_basement` , and `sqft_15 too` .

Lets drop those variables now.

In [14]:
```
df.drop(columns=['id', 'date', 'lat', 'long', 'zipcode'], axis=1, inplace=True)
df.head()
```

Out[14]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement
0	221900.0	3	1.00	1180	5650	1.0	NO	NONE	Average	7 Average	1180	0.0
1	538000.0	3	2.25	2570	7242	2.0	NO	NONE	Average	7 Average	2170	400.0
2	180000.0	2	1.00	770	10000	1.0	NO	NONE	Average	6 Low Average	770	0.0
3	604000.0	4	3.00	1960	5000	1.0	NO	NONE	Very Good	7 Average	1050	910.0
4	510000.0	3	2.00	1680	8080	1.0	NO	NONE	Average	8 Good	1680	0.0

Great! Now our dataframe only includes variables that will (hopefully) allow us to answer our buisness question. Dealing with fewer variables will simplfy the analysis process.

1.3.3 Handeling non-numeric values

Lets look into the types of data in our dataframe again, now that we've altered it a little.

In [15]:
```
# Looking into the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 16 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   price          21597 non-null  float64
 1   bedrooms       21597 non-null  int64  
 2   bathrooms      21597 non-null  float64
 3   sqft_living    21597 non-null  int64  
 4   sqft_lot       21597 non-null  int64  
 5   floors         21597 non-null  float64
 6   waterfront     21597 non-null  object 
 7   view           21597 non-null  object 
 8   condition      21597 non-null  object 
 9   grade          21597 non-null  object 
 10  sqft_above     21597 non-null  int64  
 11  sqft_basement  21597 non-null  object 
 12  yr_built       21597 non-null  int64  
 13  yr_renovated   21597 non-null  float64
 14  sqft_living15  21597 non-null  int64  
 15  sqft_lot15     21597 non-null  int64  
dtypes: float64(4), int64(7), object(5)
memory usage: 2.6+ MB
```

We see we have 5 columns listed as objects - `waterfront` , `view` , `condition` , `grade` , and `sqft_basement` .

- We've already looked into `waterfront` and seen that this category is binary - either `YES` or `NO` . As such, we can recode these as 1 and 0 respectivly.
- We also already looked in `view` , so we can also recode these values on a scale of 0-4 (0 = `NONE` , ... 4 = `EXCELLENT`).

- We'll have to look into `condition` , `grade` , and `sqft_basement` to better know how to handle them.

Let's start by recoding `waterfront` :

In [16]:
```
df['waterfront'] = df['waterfront'].replace(
    to_replace=['YES', 'NO'],
    value=[1, 0])
```

In [17]:
```
# checking that the recode worked
df['waterfront'].value_counts()
```

Out[17]:
```
0    21451
1      146
Name: waterfront, dtype: int64
```

Now we'll recode `view` :

In [18]:
```
df['view'] = df['view'].replace(
    to_replace=['NONE', 'FAIR', 'AVERAGE', 'GOOD', 'EXCELLENT'],
    value=[0, 1, 2, 3, 4])
```

In [19]:
```
# checking that the recode worked
df['view'].value_counts()
```

Out[19]:
```
0    19485
2      957
3      508
1      330
4      317
Name: view, dtype: int64
```

Lets look into `condition` :

In [20]:
```
df['condition'].value_counts()
```

Out[20]:
```
Average      14020
Good          5677
Very Good     1701
Fair           170
Poor            29
Name: condition, dtype: int64
```

This is category is a little less intuitive to know how to classify, so I looked into the dictonary and went to the [link (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r)](https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r) mentioned there, searched for BUILDING CONDITION and found this scale to work with:

1 = Poor, 2 = Fair, 3 = Average, 4 = Good, 5= Very Good

As such, we will recode the condition column as they reccomended!

In [21]:
```
df['condition'] = df['condition'].replace(
    to_replace=['Poor', 'Fair', 'Average', 'Good', 'Very Good'],
    value=[1, 2, 3, 4, 5])
```

In [22]:
```
# check the changes we made
df['condition'].value_counts()
```

Out[22]:
```
3    14020
4     5677
5     1701
2      170
1       29
Name: condition, dtype: int64
```

Looking into `grade` :

In [23]:
```
df['grade'].value_counts()
```

Out[23]:
```
7 Average        8974
8 Good           6065
9 Better         2615
6 Low Average    2038
10 Very Good     1134
11 Excellent      399
5 Fair            242
12 Luxury          89
4 Low              27
13 Mansion         13
3 Poor              1
Name: grade, dtype: int64
```

So this is messier than the previous variables. Once again I looked in the `column_names.md.txt` dictionary and found this under the heading BUILDING GRADE:

Represents the construction quality of improvements. Grades run from grade 1 to 13. Generally defined as:

- 1-3 Falls short of minimum building standards. Normally cabin or inferior structure.
- 4 Generally older, low quality construction. Does not meet code.
- 5 Low construction costs and workmanship. Small, simple design.
- 6 Lowest grade currently meeting building code. Low quality materials and simple designs.
- 7 Average grade of construction and design. Commonly seen in plats and older sub-divisions.
- 8 Just above average in construction and design. Usually better materials in both the exterior and interior finish work.
- 9 Better architectural design with extra interior and exterior design and quality.
- 10 Homes of this quality generally have high quality features. Finish work is better and more design quality is seen in the floor plans. Generally have a larger square footage.
- 11 Custom design and higher quality finish work with added amenities of solid woods, bathroom fixtures and more luxurious options.
- 12 Custom design and excellent builders. All materials are of the highest quality and all conveniences are present.
- 13 Generally custom designed and built. Mansion level. Large amount of highest quality cabinet work, wood trim, marble, entry ways etc.

From this we see that the values do have a scale, indicated by the numbers at the prefix of the values shown above, but the scale starts at 3 (as 1-3 seem to all be lumped into one category). As such, let's recode these values from 1 (Poor) to 11 (Mansion) according to the above scale.

In [24]:
```
df['grade'].value_counts()
```

Out[24]:
```
7 Average        8974
8 Good           6065
9 Better         2615
6 Low Average    2038
10 Very Good     1134
11 Excellent      399
5 Fair            242
12 Luxury          89
4 Low              27
13 Mansion         13
3 Poor              1
Name: grade, dtype: int64
```

In [25]:
```
df['grade'] = df['grade'].replace(
    to_replace=['3 Poor', '4 Low', '5 Fair', '6 Low Average',
                '7 Average', '8 Good', '9 Better', '10 Very Good',
                '11 Excellent', '12 Luxury', '13 Mansion'],
    value=[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
```

In [26]:
```
# checking on the changes we made above
df['grade'].value_counts()
```

Out[26]:
```
7     8974
8     6065
9     2615
6     2038
10    1134
11     399
5      242
12      89
4       27
13      13
3        1
Name: grade, dtype: int64
```

Because the magazine is focused on middle class homes, let's use this category to subset the dataset by removing high grade homes *(12 and above)* and the low grade homes *(5 and below)*

In [27]:
```
drop_grade = df[(df['grade'] >= 12) | (df['grade'] < 6)].index
df.drop(drop_grade, inplace=True)
df  # we see we dropped around 300 rows
```

Out[27]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement
0	221900.0	3	1.00	1180	5650	1.0	0	0	3	7	1180	0.0
1	538000.0	3	2.25	2570	7242	2.0	0	0	3	7	2170	400.0
2	180000.0	2	1.00	770	10000	1.0	0	0	3	6	770	0.0
3	604000.0	4	3.00	1960	5000	1.0	0	0	5	7	1050	910.0
4	510000.0	3	2.00	1680	8080	1.0	0	0	3	8	1680	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...
21592	360000.0	3	2.50	1530	1131	3.0	0	0	3	8	1530	0.0
21593	400000.0	4	2.50	2310	5813	2.0	0	0	3	8	2310	0.0
21594	402101.0	2	0.75	1020	1350	2.0	0	0	3	7	1020	0.0
21595	400000.0	3	2.50	1600	2388	2.0	0	0	3	8	1600	0.0
21596	325000.0	2	0.75	1020	1076	2.0	0	0	3	7	1020	0.0

21225 rows × 16 columns

Finally, lets look at our last object category `sqft_basement` :

In [28]:
```
df['sqft_basement'].value_counts()
```

Out[28]:
```
0.0       12535
?           447
600.0       216
500.0       209
700.0       207
          ...  
935.0         1
274.0         1
2180.0        1
3260.0        1
2610.0        1
Name: sqft_basement, Length: 291, dtype: int64
```

So it seems like the only string variable that we have here is `?` - so lets turn all of those into zeros, and recast this category as a float.

In [29]:
```
df['sqft_basement'] = df['sqft_basement'].replace(
    to_replace=['?'],
    value=[0.0]).astype(float)
```

In [30]:
```
# Once again, checking what we did
df['sqft_basement'].value_counts()
```

Out[30]:
```
0.0       12982
600.0       216
500.0       209
700.0       207
800.0       200
          ...
2130.0        1
1548.0        1
915.0         1
508.0         1
906.0         1
Name: sqft_basement, Length: 290, dtype: int64
```

In [31]:
```
# Now we see that there are no more `object` Dtypes - woohoo!
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21225 entries, 0 to 21596
Data columns (total 16 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   price          21225 non-null  float64
 1   bedrooms       21225 non-null  int64
 2   bathrooms      21225 non-null  float64
 3   sqft_living    21225 non-null  int64
 4   sqft_lot       21225 non-null  int64
 5   floors         21225 non-null  float64
 6   waterfront     21225 non-null  int64
 7   view           21225 non-null  int64
 8   condition      21225 non-null  int64
 9   grade          21225 non-null  int64
 10  sqft_above     21225 non-null  int64
 11  sqft_basement  21225 non-null  float64
 12  yr_built       21225 non-null  int64
 13  yr_renovated   21225 non-null  float64
 14  sqft_living15  21225 non-null  int64
 15  sqft_lot15     21225 non-null  int64
dtypes: float64(5), int64(11)
memory usage: 2.8 MB
```

Now that we have all our data as integers or floats, we still have to deal with the categorical variables, otherwise when we try to build our models it will interprete the information provided incorrectly. We will use one hot encoding (OHE) to do this.

The columns we will use this on are as they are categorical are: `view` , `condition` , and `grade` .

In [32]:
```
# OHE the above categories
cat_var = ['view', 'condition', 'grade']
preprocessed_df = pd.get_dummies(
    df, prefix=cat_var, columns=cat_var, drop_first=True)
```

In [33]:
```
# Look at the pretty preprocessed data!
preprocessed_df.head()
```

Out[33]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	sqft_above	sqft_basement	yr_built	...	view_4	condit
0	221900.0	3	1.00	1180	5650	1.0	0	1180	0.0	1955	...	0	
1	538000.0	3	2.25	2570	7242	2.0	0	2170	400.0	1951	...	0	
2	180000.0	2	1.00	770	10000	1.0	0	770	0.0	1933	...	0	
3	604000.0	4	3.00	1960	5000	1.0	0	1050	910.0	1965	...	0	
4	510000.0	3	2.00	1680	8080	1.0	0	1680	0.0	1987	...	0	

5 rows × 26 columns

1.3.4 Check for Multicollinearity

In [34]:

```
# Now lets drop our dependant variable 'price' to look at
# the relationships between our predictors
predict = preprocessed_df.drop('price', axis=1)
# and then we will look at the correlations between the predictors
corr_predictors = predict.corr().abs().stack(
).reset_index().sort_values(0, ascending=False)
corr_predictors['pairs'] = list(
    zip(corr_predictors.level_0, corr_predictors.level_1))
corr_predictors.set_index(['pairs'], inplace=True)
corr_predictors.drop(columns=['level_1', 'level_0'], inplace=True)
corr_predictors.columns = ['correlations']
corr_predictors[(corr_predictors.correlations > .75)
                & (corr_predictors.correlations < 1)]
```

Out[34]:

pairs	correlations
(sqft_above, sqft_living)	0.866763
(sqft_living, sqft_above)	0.866763
(condition_3, condition_4)	0.815018
(condition_4, condition_3)	0.815018
(sqft_living, sqft_living15)	0.753515
(sqft_living15, sqft_living)	0.753515

For 3 out out of 4 of the above high correlations, we see that `sqft_living` is one of the variables, along with some of our OHE variables - `condition_3` and `condition_4` . For now, we will leave them in, but it's something to keep in mind as we build our model later.

In [35]:

```
preprocessed_df.head()
```

Out[35]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	sqft_above	sqft_basement	yr_built	...	view_4	condit
0	221900.0	3	1.00	1180	5650	1.0	0	1180	0.0	1955	...	0	
1	538000.0	3	2.25	2570	7242	2.0	0	2170	400.0	1951	...	0	
2	180000.0	2	1.00	770	10000	1.0	0	770	0.0	1933	...	0	
3	604000.0	4	3.00	1960	5000	1.0	0	1050	910.0	1965	...	0	
4	510000.0	3	2.00	1680	8080	1.0	0	1680	0.0	1987	...	0	

5 rows × 26 columns

1.3.5 Checking Variable Distributions

In [36]:
```
""" Just by looking at the `price` column, we see we have
21,1225 data points, with a mean sale price of 535,000$ (rounded
up)"""
preprocessed_df.describe()
```

Out[36]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	sqft_above	sqft_base
count	2.122500e+04	21225.000000	21225.000000	21225.000000	2.122500e+04	21225.000000	21225.000000	21225.00000	21225.00
mean	5.351421e+05	3.382144	2.119022	2077.122874	1.482257e+04	1.497150	0.006219	1785.09371	285.91
std	3.345604e+05	0.915107	0.748204	872.486658	3.998511e+04	0.540138	0.078617	795.28664	433.94
min	8.200000e+04	1.000000	0.500000	390.000000	5.200000e+02	1.000000	0.000000	390.00000	0.00
25%	3.250000e+05	3.000000	1.750000	1440.000000	5.033000e+03	1.000000	0.000000	1200.00000	0.00
50%	4.520000e+05	3.000000	2.250000	1920.000000	7.600000e+03	1.500000	0.000000	1570.00000	0.00
75%	6.440000e+05	4.000000	2.500000	2550.000000	1.057400e+04	2.000000	0.000000	2210.00000	550.00
max	7.060000e+06	33.000000	7.500000	10040.000000	1.651359e+06	3.500000	1.000000	8020.00000	3260.00

8 rows × 26 columns

1.4 Basic Model

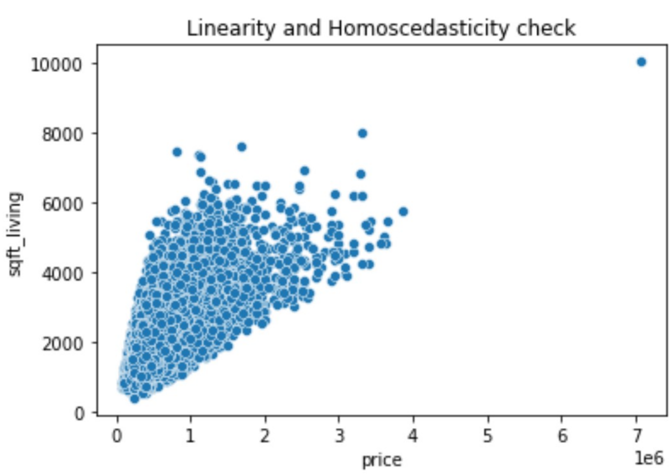
1.4.1 Squarefoot Living Space

To start off we will pick an independent variable that should be important (`sqft_living`) and create a simple linear model with our dependent variable `price` as our baseline model. (Remember, one of our initial questions was if increasing the living space of a home increased the home's value!)

After we've looked into this, we will add more variables to see if we can improve on the model.

Before we start on this, let's check if the relationship between `price` and `sqft_living` meets the criteria for linear regression.

In [37]:
```
# check for linearity and Homoscedasticity
sns.scatterplot(x=preprocessed_df['price'], y=preprocessed_df['sqft_living'])
plt.title("Linearity and Homoscedasticity check");
```

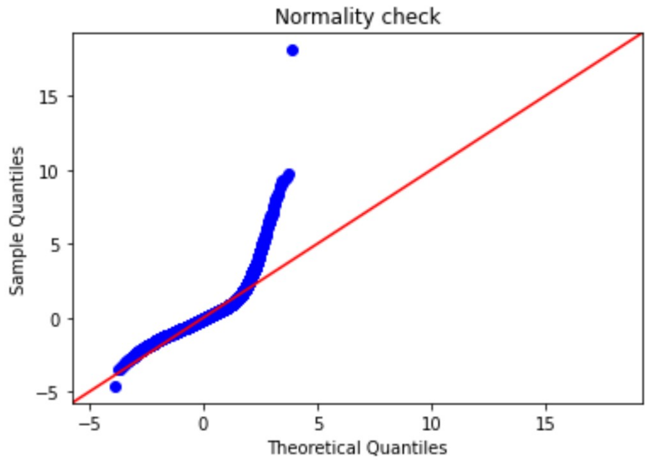


In [38]:
```
# create predictors
predictors = preprocessed_df['sqft_living']
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
baseline_model = sm.OLS(preprocessed_df['price'], predictors_int).fit()

# check model
baseline_model.params
```

Out[38]:
```
const          -3174.361664
sqft_living      259.164480
dtype: float64
```

In [39]:

```
# check normality assumption

residuals = baseline_model.resid
fig = sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True)
plt.title("Normality check")
fig.show()
```



So we see that 2/3 of the assumptions of linearity are violated here - the residuals aren't normally distributed, and the data isn't homoscedastic. We'll get a summary of the model as is, see if performing a log transformation on `price` and `sqft_living` will help with these conditions, and then see if adding in some other variables to our model will improve our R^2 .

In [40]:

```
baseline_model.summary()
```

Out[40]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.457
Model:	OLS	**Adj. R-squared:**	0.457
Method:	Least Squares	**F-statistic:**	1.785e+04
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:44	**Log-Likelihood:**	-2.9363e+05
No. Observations:	21225	**AIC:**	5.873e+05
Df Residuals:	21223	**BIC:**	5.873e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-3174.3617	4370.593	-0.726	0.468	-1.17e+04	5392.331

In [41]:
```
# apply logarithmic function to independant variable
preprocessed_df['log_sqft_living'] = np.log(preprocessed_df['sqft_living'])


# re-create the model with `log_sqft_living`
# create predictors
predictors = preprocessed_df['log_sqft_living']
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
log_model1 = sm.OLS(preprocessed_df['price'], predictors_int).fit()

# check model
print(log_model1.params)
log_model1.summary()
```

```
const              -3.218143e+06
log_sqft_living     4.967772e+05
dtype: float64
```

Out[41]:

OLS Regression Results

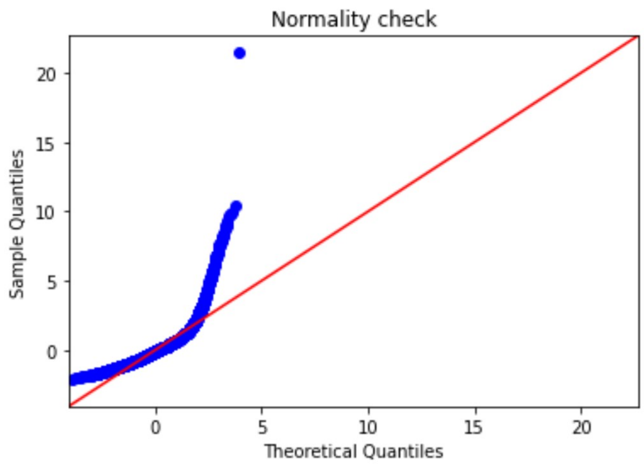
Dep. Variable:	price	**R-squared:**	0.371
Model:	OLS	**Adj. R-squared:**	0.371
Method:	Least Squares	**F-statistic:**	1.251e+04
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:44	**Log-Likelihood:**	-2.9519e+05
No. Observations:	21225	**AIC:**	5.904e+05
Df Residuals:	21223	**BIC:**	5.904e+05
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-3.218e+06	3.36e+04	-95.778	0.000	-3.28e+06	-3.15e+06
log_sqft_living	4.968e+05	4440.693	111.869	0.000	4.88e+05	5.05e+05

Omnibus:	14777.401	**Durbin-Watson:**	1.977
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	505627.536
Skew:	2.914	**Prob(JB):**	0.00
Kurtosis:	26.190	**Cond. No.**	142.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [42]:
```
residuals = log_model1.resid
fig = sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True)
plt.title("Normality check")
fig.show()
```



Ooph! So that lowered our R^2 signifigantly. Lets see what happens if we perform a log function just on `price`.

In [43]:

```
# apply logarithmic function to dependant variable
preprocessed_df['log_price'] = np.log(preprocessed_df['price'])


# re-create the model with `sqft_living`
# create predictors
predictors = preprocessed_df['sqft_living']
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
log_model2 = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
print(log_model2.params)
log_model2.summary()
```

```
const          12.226847
sqft_living     0.000396
dtype: float64
```

Out[43]:

OLS Regression Results

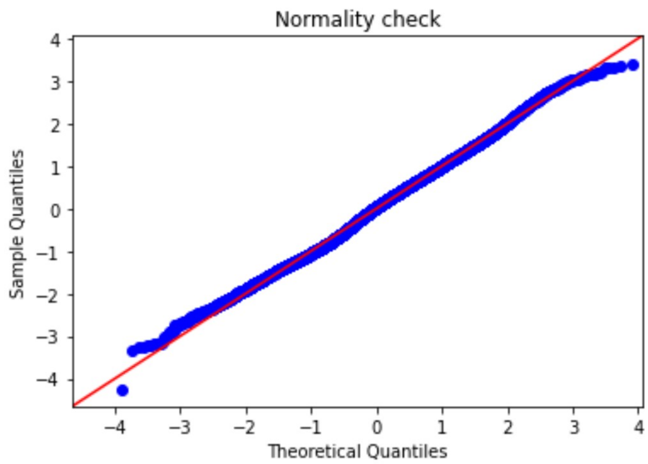
Dep. Variable:	log_price	**R-squared:**	0.460
Model:	OLS	**Adj. R-squared:**	0.460
Method:	Least Squares	**F-statistic:**	1.805e+04
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:45	**Log-Likelihood:**	-9303.0
No. Observations:	21225	**AIC:**	1.861e+04
Df Residuals:	21223	**BIC:**	1.863e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	12.2268	0.007	1839.098	0.000	12.214	12.240
sqft_living	0.0004	2.95e-06	134.337	0.000	0.000	0.000

Omnibus:	66.633	**Durbin-Watson:**	1.976
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	56.717
Skew:	0.069	**Prob(JB):**	4.83e-13
Kurtosis:	2.788	**Cond. No.**	5.82e+03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 5.82e+03. This might indicate that there are
strong multicollinearity or other numerical problems.

In [44]:
```
residuals = log_model2.resid
fig = sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True)
plt.title("Normality check")
fig.show()
```



Wow! Thats looking way better, and our R^2 is slighly higher than our baseline model. Lets check what happens if we apply the log function to both and throw them in our model.

In [45]:

```
# re-create the model with `sqft_living`
# create predictors
predictors = preprocessed_df['log_sqft_living']
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
log_model3 = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
print(log_model3.params)
log_model3.summary()
```

```
const              6.871903
log_sqft_living    0.817756
dtype: float64
```

Out[45]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.432
Model:	OLS	**Adj. R-squared:**	0.432
Method:	Least Squares	**F-statistic:**	1.615e+04
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:45	**Log-Likelihood:**	-9827.2
No. Observations:	21225	**AIC:**	1.966e+04
Df Residuals:	21223	**BIC:**	1.967e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	6.8719	0.049	141.158	0.000	6.776	6.967
log_sqft_living	0.8178	0.006	127.099	0.000	0.805	0.830

Omnibus:	115.753	**Durbin-Watson:**	1.975
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	105.001
Skew:	0.132	**Prob(JB):**	1.58e-23
Kurtosis:	2.780	**Cond. No.**	142.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

So takign the log of both seems to have lowered our R^2 a bit, and will make interpretation a bit more challenging, so lets stick with `log_model2` with only `price` being transformed.

1.5 Adding Features

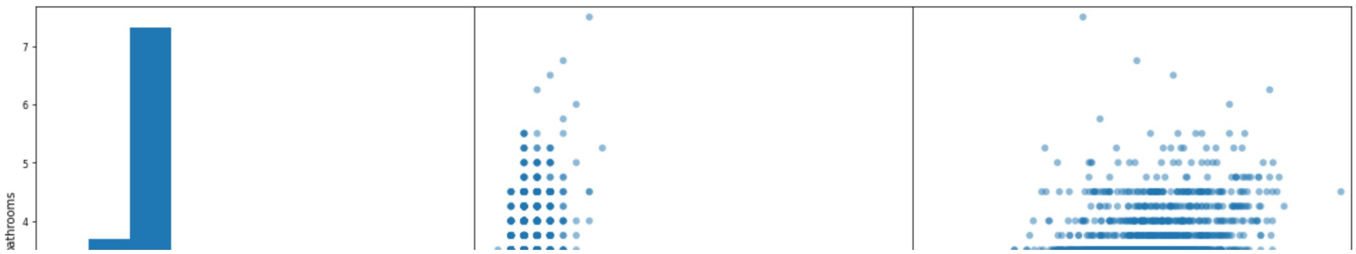
1.5.1 Bathrooms and Bedrooms

Lets start by adding `bathrooms` to the model.

First we'll check for linearity and homoscedasticity in `bathrooms` and `bedrooms` compared to `log_price` , as these are the only continuous variables we are looking at.

In [46]:
```python
col = ['bathrooms', 'bedrooms', 'log_price']
pd.plotting.scatter_matrix(
    preprocessed_df[col], figsize=(20, 20), grid=True, marker='o')
```

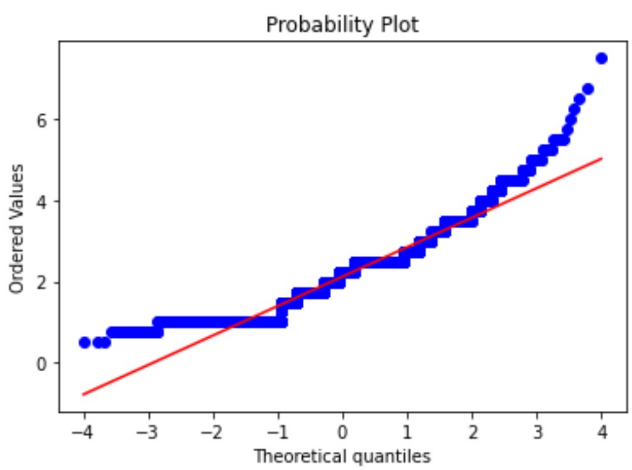
Out[46]:
```
array([[<AxesSubplot:xlabel='bathrooms', ylabel='bathrooms'>,
        <AxesSubplot:xlabel='bedrooms', ylabel='bathrooms'>,
        <AxesSubplot:xlabel='log_price', ylabel='bathrooms'>],
       [<AxesSubplot:xlabel='bathrooms', ylabel='bedrooms'>,
        <AxesSubplot:xlabel='bedrooms', ylabel='bedrooms'>,
        <AxesSubplot:xlabel='log_price', ylabel='bedrooms'>],
       [<AxesSubplot:xlabel='bathrooms', ylabel='log_price'>,
        <AxesSubplot:xlabel='bedrooms', ylabel='log_price'>,
        <AxesSubplot:xlabel='log_price', ylabel='log_price'>]],
      dtype=object)
```



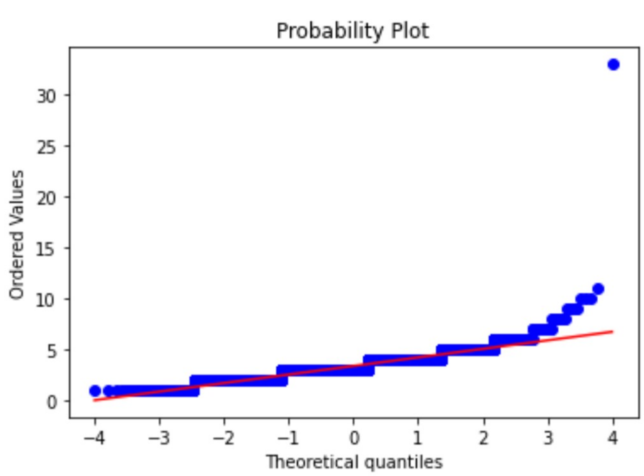
We see that there is linearity and homoscedasticity in `bathrooms` and `log_price` - lets check for normality in `bathrooms` and `bedrooms`.

Type *Markdown* and LaTeX: α^2

In [47]:
```python
stats.probplot(preprocessed_df['bathrooms'], dist="norm", plot=pylab)
pylab.show()
```



In [48]:
```python
stats.probplot(preprocessed_df['bedrooms'], dist="norm", plot=pylab)
pylab.show()
```



Looks like it's not perfectly normal, but it's better than it was for `sqft_living`.

In [49]:

```
# create predictors
predictors = preprocessed_df[['sqft_living', 'bathrooms']]
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
second_model = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
second_model.summary()
```

Out[49]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.461
Model:	OLS	**Adj. R-squared:**	0.461
Method:	Least Squares	**F-statistic:**	9070.
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-9277.5
No. Observations:	21225	**AIC:**	1.856e+04
Df Residuals:	21222	**BIC:**	1.858e+04
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.1976	0.008	1563.860	0.000	12.182	12.213
sqft_living	0.0004	4.39e-06	85.085	0.000	0.000	0.000
bathrooms	0.0366	0.005	7.152	0.000	0.027	0.047

Omnibus:	74.997	**Durbin-Watson:**	1.976
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	64.878
Skew:	0.083	**Prob(JB):**	8.16e-15
Kurtosis:	2.786	**Cond. No.**	7.36e+03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 7.36e+03. This might indicate that there are
strong multicollinearity or other numerical problems.

So we see here that in this model, our R^2 has dropped a little. That may be due to high multicollinearity between `sqft_living` and `bathrooms`. If we try building off of this model, we get some really weird results (the bathrooms coefficient becomes negative). Lets build another model with `log_price` with `bedrooms` and `bathrooms` as predictors, without `sqft_living`.

In [50]:

```
# create predictors
predictors = preprocessed_df[['bedrooms', 'bathrooms']]
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
third_model = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
third_model.summary()
```

Out[50]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.282
Model:	OLS	**Adj. R-squared:**	0.282
Method:	Least Squares	**F-statistic:**	4164.
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-12320.
No. Observations:	21225	**AIC:**	2.465e+04
Df Residuals:	21222	**BIC:**	2.467e+04
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.1959	0.012	1029.522	0.000	12.173	12.219

So, here we see a lower R^2 as we took out `sqft_living` , but we can see that adding bathrooms seems to be associated with more significant price increases in homes, compared to bedrooms.

1.5.2 Grade and Condition

Before dealing with our final question, lets glance at our current dataframe.

In [51]:
```
preprocessed_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21225 entries, 0 to 21596
Data columns (total 28 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   price            21225 non-null  float64
 1   bedrooms         21225 non-null  int64  
 2   bathrooms        21225 non-null  float64
 3   sqft_living      21225 non-null  int64  
 4   sqft_lot         21225 non-null  int64  
 5   floors           21225 non-null  float64
 6   waterfront       21225 non-null  int64  
 7   sqft_above       21225 non-null  int64  
 8   sqft_basement    21225 non-null  float64
 9   yr_built         21225 non-null  int64  
 10  yr_renovated     21225 non-null  float64
 11  sqft_living15    21225 non-null  int64  
 12  sqft_lot15       21225 non-null  int64  
 13  view_1           21225 non-null  uint8  
 14  view_2           21225 non-null  uint8  
 15  view_3           21225 non-null  uint8  
 16  view_4           21225 non-null  uint8  
 17  condition_2      21225 non-null  uint8  
 18  condition_3      21225 non-null  uint8  
 19  condition_4      21225 non-null  uint8  
 20  condition_5      21225 non-null  uint8  
 21  grade_7          21225 non-null  uint8  
 22  grade_8          21225 non-null  uint8  
 23  grade_9          21225 non-null  uint8  
 24  grade_10         21225 non-null  uint8  
 25  grade_11         21225 non-null  uint8  
 26  log_sqft_living  21225 non-null  float64
 27  log_price        21225 non-null  float64
dtypes: float64(7), int64(8), uint8(13)
memory usage: 2.9 MB
```

So we have 4 categories in condition, and another 4 in grade. Lets add in the grade categories to our `sqft_living` model and see how that goes. Because these are binary columns, we do not have to check for assumptions of linearity.

In [52]:

```
# create predictors
predictors = preprocessed_df[['sqft_living',
                              'grade_7','grade_8', 'grade_9', 'grade_10', 'grade_11']]
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
fourth_model = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
fourth_model.summary()
```

Out[52]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.531
Model:	OLS	**Adj. R-squared:**	0.531
Method:	Least Squares	**F-statistic:**	3998.
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-7806.3
No. Observations:	21225	**AIC:**	1.563e+04
Df Residuals:	21218	**BIC:**	1.568e+04
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.2790	0.009	1333.674	0.000	12.261	12.297
sqft_living	0.0002	4.18e-06	53.165	0.000	0.000	0.000
grade_7	0.1822	0.009	20.637	0.000	0.165	0.199
grade_8	0.3717	0.010	37.676	0.000	0.352	0.391
grade_9	0.5711	0.012	45.754	0.000	0.547	0.596
grade_10	0.7418	0.016	45.788	0.000	0.710	0.774
grade_11	0.8762	0.023	37.512	0.000	0.830	0.922

Omnibus:	68.914	**Durbin-Watson:**	1.969
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	67.147
Skew:	0.121	**Prob(JB):**	2.63e-15
Kurtosis:	2.868	**Cond. No.**	2.63e+04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.63e+04. This might indicate that there are
strong multicollinearity or other numerical problems.

In [53]:
```
predictors = preprocessed_df[['sqft_living','grade_7', 'grade_8', 'grade_9', 'grade_10', 'grade_11', 'conc
                                'condition_4', 'condition_5']]
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
fifth_model = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
fifth_model.summary()
```

Out[53]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.548
Model:	OLS	**Adj. R-squared:**	0.548
Method:	Least Squares	**F-statistic:**	2572.
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-7406.3
No. Observations:	21225	**AIC:**	1.483e+04
Df Residuals:	21214	**BIC:**	1.492e+04
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.2608	0.079	155.328	0.000	12.106	12.416

Looking at the confidence interval of condition 2-4 (along with their low coefficients) lets see what happens to our R^2 when we remove them from the model

In [54]:
```
predictors = preprocessed_df[['sqft_living', 'grade_7','grade_8',
                               'grade_9', 'grade_10', 'grade_11', 'condition_5']]
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
sixth_model = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
sixth_model.summary()
```

Out[54]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.542
Model:	OLS	**Adj. R-squared:**	0.542
Method:	Least Squares	**F-statistic:**	3590.
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-7542.2
No. Observations:	21225	**AIC:**	1.510e+04
Df Residuals:	21217	**BIC:**	1.516e+04
Df Model:	7		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.2622	0.009	1344.156	0.000	12.244	12.280
sqft_living	0.0002	4.14e-06	52.054	0.000	0.000	0.000
grade_7	0.1915	0.009	21.938	0.000	0.174	0.209
grade_8	0.3902	0.010	39.912	0.000	0.371	0.409
grade_9	0.5976	0.012	48.264	0.000	0.573	0.622
grade_10	0.7726	0.016	48.120	0.000	0.741	0.804
grade_11	0.9172	0.023	39.643	0.000	0.872	0.963
condition_5	0.2052	0.009	23.120	0.000	0.188	0.223

Omnibus:	55.293	**Durbin-Watson:**	1.970
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	55.276
Skew:	0.119	**Prob(JB):**	9.93e-13
Kurtosis:	2.922	**Cond. No.**	2.65e+04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.65e+04. This might indicate that there are
strong multicollinearity or other numerical problems.

Okay, so that lowered our R^2 slightly, and left us with less information. Going forward lets focus in on the fifth and third models, along with one of our earlier models (`log_model2`) as these will be the most usefull in answering our initial questions.

2 Results and Conclusions

Before we get into our results, lets just remember our initial questions:

2.0.0.1 Initial Questions

1. Will increasing the living area size lead to an associated increase in the value of the home?
2. Will adding bedrooms or bathrooms lead to an associated increase in the value of the home?
3. Is the grade or condition rating of the house associated with the value of the home?

2.1 1. Will increasing the living area size lead to an associated increase in the value of the home?

In [55]:
```
#divinding by 100 to get
preprocessed_df['per100_sqft_living'] = (preprocessed_df['sqft_living']/100)
```

In [56]:
```

# re-create the model with `sqft_living`
# create predictors
predictors = preprocessed_df['per100_sqft_living']
# create model intercept
predictors_int = sm.add_constant(predictors)
# fit model
log_model4 = sm.OLS(preprocessed_df['log_price'], predictors_int).fit()

# check model
print(log_model4.params)
log_model4.summary()
```

```
const                 12.226847
per100_sqft_living     0.039643
dtype: float64
```

Out[56]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.460
Model:	OLS	**Adj. R-squared:**	0.460
Method:	Least Squares	**F-statistic:**	1.805e+04
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-9303.0
No. Observations:	21225	**AIC:**	1.861e+04
Df Residuals:	21223	**BIC:**	1.863e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	12.2268	0.007	1839.098	0.000	12.214	12.240
per100_sqft_living	0.0396	0.000	134.337	0.000	0.039	0.040

Omnibus:	66.633	**Durbin-Watson:**	1.976
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	56.717
Skew:	0.069	**Prob(JB):**	4.83e-13
Kurtosis:	2.788	**Cond. No.**	58.3

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

So we see here that there is a fairly large association between the log price of a home and its square footage of living space. This also has a small standard error, and confidence interval, making it as a very accurate metric! As such, we can say that for every 100 square foot increased of living space in a home there is an association of an increase of .0396 of the log price. While this can be difficult to interpret in lay terms, it means all in all that based on what we see above there is a strong association between an increase in a home's square footage of living area and its price. Additionally, when we look at the R^2 we see that `sqft_living` can explain 43.2% of the `log_price` - a hefty chunk!

Lets try using the antilog to translate the `log_price` into something more relevant for our visualization.

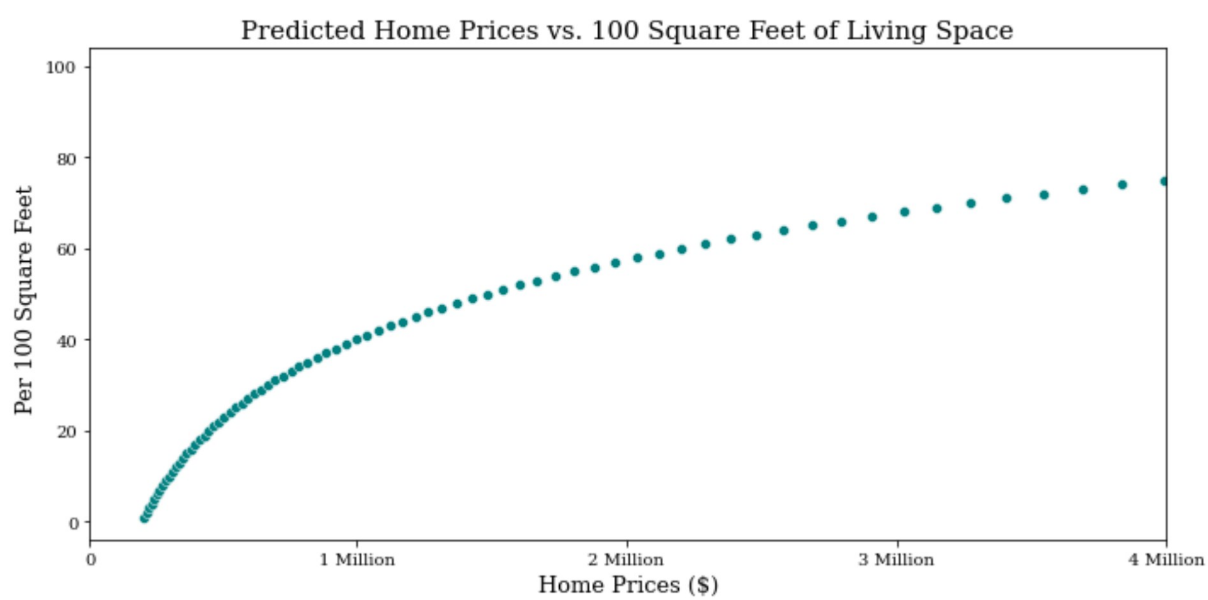
In [57]:
```
# code taken from: https://github.com/fbenamy/tutoring/blob/main/Create%20Simulated%20Data%20for%20Multipl

#defining the upper and lower bound of price, and the spacing interval
target_variable_vector = np.arange(1, 100, 1)
target_variable_matrix = sm.add_constant(target_variable_vector)
```

In [58]:
```
log_results = log_model4.predict(target_variable_matrix)
price_results = np.exp(log_results)
```

2.1.1 Squarefoot Living Visualization

While using the log price was very useful for our linear model, lets use the predicted house prices in `price_results` and compare them to `sqft_living`.

In [59]:
```
# setting universal font type for this and future graphs - from :https://datascienceparichay.com/article/c
plt.rcParams.update({'font.family': 'serif'})

# specify size of plot
fig, ax = plt.subplots(figsize=(10, 5))

# set plot limits and tick labels
ax.set_xlim(10000, 4000000)
plt.xticks([10000,1000000,2000000,3000000,4000000],['0', '1 Million', '2 Million',
                   '3 Million', '4 Million'])
#set up scatterplot
sns.scatterplot(x=price_results, y=target_variable_vector, color ='teal')
#change axis titles and heading
plt.title('Predicted Home Prices vs. 100 Square Feet of Living Space', fontsize=15)
plt.xlabel('Home Prices ($)', fontsize=13)
plt.ylabel('Per 100 Square Feet ', fontsize=13)

plt.tight_layout()
plt.show();
```



2.2 2. Will adding bedrooms or bathrooms lead to an associated increase in the value of the home?

In order to answer this question, lets re-examine the `third_model`.

In [60]:

```
third_model.summary()
```

Out[60]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.282
Model:	OLS	**Adj. R-squared:**	0.282
Method:	Least Squares	**F-statistic:**	4164.
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-12320.
No. Observations:	21225	**AIC:**	2.465e+04
Df Residuals:	21222	**BIC:**	2.467e+04
Df Model:	2		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	12.1959	0.012	1029.522	0.000	12.173	12.219
bedrooms	0.0452	0.004	12.044	0.000	0.038	0.053
bathrooms	0.3311	0.005	72.153	0.000	0.322	0.340

Omnibus:	182.178	**Durbin-Watson:**	1.958
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	186.702
Skew:	0.229	**Prob(JB):**	2.87e-41
Kurtosis:	3.036	**Cond. No.**	17.4

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In examining this model, we see that adding bedrooms and bathrooms are both associated with an increase in the log price. The R^2 is lower than in our previous model (28.2%), which indicates that the number of bedrooms and bathrooms explains less of the log price than `sqft_living` . It's important to remember that there is likely collinearity between `sqft_living` and `bedrooms` and `bathrooms` , which could have led to the wonky results we saw in the analysis. That being said, we see that adding one bedroom is associated with a .05 (rounded) increase in log price, while adding one bathroom is associated with a .3 increase in log price - indicating that if you have to choose between adding a bedroom or a bathroom, adding a bathroom is indicated as the better fiscal choice.

2.3 3. Is the grade or condition rating of the house associated with the value of the home?

Finally, lets look at our final model - the `fifth_model` , to look at `grade` and `condition` . Just a reminder, `grade` indicates the construction/building quality of the house, while `condition` refers to the maintenance level.

The `fifth_model` builds upon the `third_model` used to answer our first question.

In [61]:
```
fifth_model.summary()
```

Out[61]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.548
Model:	OLS	**Adj. R-squared:**	0.548
Method:	Least Squares	**F-statistic:**	2572.
Date:	Mon, 03 Oct 2022	**Prob (F-statistic):**	0.00
Time:	08:30:47	**Log-Likelihood:**	-7406.3
No. Observations:	21225	**AIC:**	1.483e+04
Df Residuals:	21214	**BIC:**	1.492e+04
Df Model:	10		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	12.2608	0.079	155.328	0.000	12.106	12.416
sqft_living	0.0002	4.12e-06	51.555	0.000	0.000	0.000
grade_7	0.1936	0.009	22.236	0.000	0.176	0.211
grade_8	0.4015	0.010	40.984	0.000	0.382	0.421
grade_9	0.6165	0.012	49.655	0.000	0.592	0.641
grade_10	0.7966	0.016	49.574	0.000	0.765	0.828
grade_11	0.9441	0.023	40.895	0.000	0.899	0.989
condition_2	-0.1302	0.084	-1.559	0.119	-0.294	0.034
condition_3	-0.0246	0.079	-0.311	0.756	-0.179	0.130
condition_4	0.0629	0.079	0.797	0.425	-0.092	0.218
condition_5	0.2069	0.079	2.611	0.009	0.052	0.362

Omnibus:	33.626	**Durbin-Watson:**	1.976
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	33.640
Skew:	0.093	**Prob(JB):**	4.96e-08
Kurtosis:	2.942	**Cond. No.**	1.69e+05

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.69e+05. This might indicate that there are
strong multicollinearity or other numerical problems.

At first glance, we see that the p-values of all of the conditions, except `condition_5` , indicate that these are not valuble contributers to the log price. From this, we can conclude that home maintenance only affects the sale price of a home if it is at the highest level. This makes sense, as it's usually assumed when one buys a home that some aspects will be run down and repairs will need to be made.

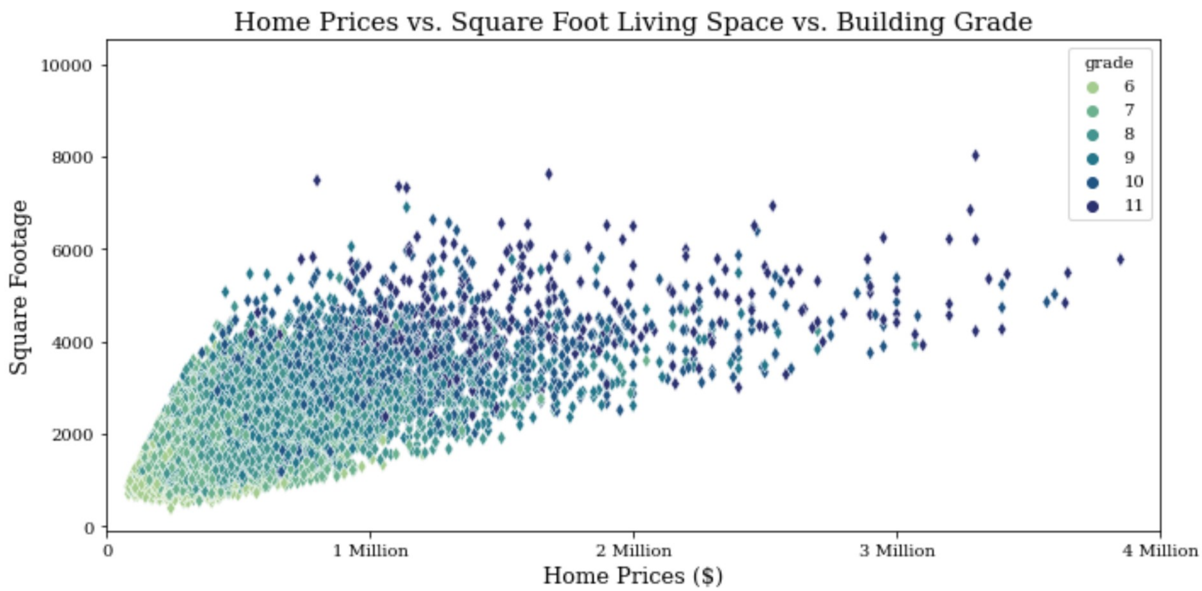
If one does maintain their home to this extent, ("All items well maintained, many having been overhauled and repaired as they have shown signs of wear, increasing the life expectancy and lowering the effective age with little deterioration or obsolescence evident with a high degree of utility") then there is an associated increase in log price of .2652.

Looking at the grade categories, we see that all of these categories are shown to be statistically significant. The coefficients of the grades increase as the grade increases, meaning that buildings with higher building grades are associated with higher log sale prices.

Finally, lets look at how `sqft_living` , `price` , and the grade categories interact prior to all our transformations .

In [62]:

```
# specify size of plot
fig, ax = plt.subplots(figsize=(10, 5))

# set plot limits and tick labels
ax.set_xlim(0, 4000000)
plt.xticks([0,1000000,2000000,3000000,4000000],['0', '1 Million', '2 Million',
                   '3 Million', '4 Million'])
#set up scatterplot
sns.scatterplot(x=df['price'], y=df['sqft_living'], hue = df['grade'], marker='d', palette = 'crest')

#change axis titles and heading
plt.title('Home Prices vs. Square Foot Living Space vs. Building Grade', fontsize=15)
plt.xlabel('Home Prices ($)', fontsize=13)
plt.ylabel('Square Footage', fontsize=13)

plt.tight_layout()
plt.show();
```



Just by eyeballing the above graph, we see that there does seem to be a trend of higher grade homes with larger square footage going for higher prices.

3 Possible Next Steps

- Look at data from other counties
- Look further into disentangling the collinearity between living space and bedrooms/bathrooms
- Investigate datasets with information on other renovations (plumbing, electric, ect.)

Type *Markdown* and LaTeX: α^2