Final Project Submission

Please fill out:

- **Student name:**
- **Student pace: self paced / part time / full time**
- **Scheduled project review date/time:**
- **Instructor name:**
- **Blog post URL:**

Background:

The real estate agency aims to provide accurate property valuations to homeowners looking to sell their homes and to buyers interested in purchasing properties. Property valuation is a critical aspect of real estate transactions as it directly impacts pricing decisions, marketing strategies, and negotiation outcomes.

Business Problem Statement:

The real estate agency faces challenges in accurately valuing properties, which can lead to overpricing or underpricing homes, affecting client satisfaction, time on market, and overall business performance

OBJECTIVES:

1.Determine the key factors such as square foot living,the number of bedrooms and bathrooms,the condition of the house and others that significantly influence the house prices.

2.Develop a model that can accurately predict house prices based on these factors.

3.To determine which seasons have the highest sales.

4.To provide valuable insights for real estate agents, property developers, and investors in the company's portfolio to make informed decisions regarding pricing, renovations, and marketing strategies

Hypothesis Testing

1.How do structural characteristics,location and others such as view and grading all work together to affect a house's market price, and how may these interactions be best modeled to produce highly accurate housing price predictions?

Ho: There is no statistically significant interaction effect between sqft_living and grading on the price of a house

H1: There is a statistically significant interaction effect between sqft_living and grading on the price of a house.

Data Understanding

In [1]:

```
#importing libraries
import pandas as pd
import csv
import json
import numpy as np
import matplotlib.pyplot as plt
```

```
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # To ignore all warnings in your script
from pandas.plotting import scatter_matrix
# 3D plotting
from mpl_toolkits.mplot3d import Axes3D
from scipy import stats
from statsmodels.stats.outliers_influence import variance_inflation_factor
import statsmodels.api as sm
```

In [2]:

```
#lets load the data
m = r"C:\Users\Victor Keya\Documents\Flatiron\Phase2_project\dsc-phase-2-project-v2-3\data\kc_house_data.csv"
df = pd.read_csv(m)
df
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	NaN	NONE	...	7 Average
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	NO	NONE	...	7 Average
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	NO	NONE	...	6 Low Average
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	NO	NONE	...	7 Average
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	NO	NONE	...	8 Good
...	...	...	...	...	...	...	...	...	...	...	...	...
21592	263000018	5/21/2014	360000.0	3	2.50	1530	1131	3.0	NO	NONE	...	8 Good
21593	6600060120	2/23/2015	400000.0	4	2.50	2310	5813	2.0	NO	NONE	...	8 Good
21594	1523300141	6/23/2014	402101.0	2	0.75	1020	1350	2.0	NO	NONE	...	7 Average
21595	291310100	1/16/2015	400000.0	3	2.50	1600	2388	2.0	NaN	NONE	...	8 Good
21596	1523300157	10/15/2014	325000.0	2	0.75	1020	1076	2.0	NO	NONE	...	7 Average

21597 rows × 21 columns

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] 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] 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

Limitations

Data Timeliness: The dataset may not reflect the current market conditions or housing trends due to its age. Real estate markets can exhibit temporal dynamics that require up-to-date data for accurate predictions.

Local Market Dynamics: The dataset's focus on King County may limit the model's generalizability to other regions or markets with different characteristics.

Interpretability of Results: Some advanced modeling techniques (e.g., polynomial regression) may produce complex results that are difficult to interpret or explain to stakeholders

In [3]:

```
df.head()
```

Out[3]:

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

5 rows × 21 columns

In [4]:

```
df.tail()
```

Out[4]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	s
21592	263000018	5/21/2014	360000.0	3	2.50	1530	1131	3.0	NO	NONE	...	8 Good	
21593	6600060120	2/23/2015	400000.0	4	2.50	2310	5813	2.0	NO	NONE	...	8 Good	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	s
21594	1523300141	6/23/2014	402101.0	2	0.75	1020	1350	2.0	NO	NONE	...	7 Average	
21595	291310100	1/16/2015	400000.0	3	2.50	1600	2388	2.0	NaN	NONE	...	8 Good	
21596	1523300157	10/15/2014	325000.0	2	0.75	1020	1076	2.0	NO	NONE	...	7 Average	

5 rows × 21 columns

In [5]:

```
df.sample()
```

Out[5]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_a
9432	7215730580	9/2/2014	680000.0	4	3.0	3150	6175	2.0	NO	NONE	...	9 Better	

1 rows × 21 columns

In [6]:

```
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
```

Our dataset has 21597 rows and 21 columns. There are null values on the waterfront, yr_renovated columns. Data types in the data set include floats, integers and objects

In [7]:

```
df.describe()
```

Out[7]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	sqft_above
count	2.159700e+04	2.159700e+04	21597.000000	21597.000000	21597.000000	2.159700e+04	21597.000000	21597.000000

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	sqft_above	
count	2.159700e+04	2.159700e+04	21597.000000	21597.000000	21597.000000	2.159700e+04	21597.000000	21597.000000	21
mean	4.580474e+09	5.402966e+05	3.373200	2.115826	2080.321850	1.509941e+04	1.494096	1788.596842	19
std	2.876736e+09	3.673681e+05	0.926299	0.768984	918.106125	4.141264e+04	0.539683	827.759761	
min	1.000102e+06	7.800000e+04	1.000000	0.500000	370.000000	5.200000e+02	1.000000	370.000000	19
25%	2.123049e+09	3.220000e+05	3.000000	1.750000	1430.000000	5.040000e+03	1.000000	1190.000000	19
50%	3.904930e+09	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	1560.000000	19
75%	7.308900e+09	6.450000e+05	4.000000	2.500000	2550.000000	1.068500e+04	2.000000	2210.000000	19
max	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	9410.000000	20

This gives a summary of the distribution of the numeric data. From the count, we can see which columns have numeric data. Outliers can also be detected using the maximum and minimum values.

DATA CLEANING

In [8]:

```
#lets make a copy of the dataset first
df1 = df.copy()
df1
```

Out[8]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	s
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	NaN	NONE	...	7 Average	
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	NO	NONE	...	7 Average	
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	NO	NONE	...	6 Low Average	
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	NO	NONE	...	7 Average	
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	NO	NONE	...	8 Good	
...	...	...	...	...	...	...	...	...	...	...	...	...	
21592	263000018	5/21/2014	360000.0	3	2.50	1530	1131	3.0	NO	NONE	...	8 Good	
21593	6600060120	2/23/2015	400000.0	4	2.50	2310	5813	2.0	NO	NONE	...	8 Good	
21594	1523300141	6/23/2014	402101.0	2	0.75	1020	1350	2.0	NO	NONE	...	7 Average	
21595	291310100	1/16/2015	400000.0	3	2.50	1600	2388	2.0	NaN	NONE	...	8 Good	
21596	1523300157	10/15/2014	325000.0	2	0.75	1020	1076	2.0	NO	NONE	...	7 Average	

21597 rows × 21 columns

The id columns seems not to be important, drop. Capitalise the column headings. Change the date format to dd/mm/yy. Put comas on the prices. Look at the data types, make sure they are correct.

i) Handling Missing Values

In [9]:

```
#handling missing values.
```

```
#Lets first identify if we have null values
df1.isnull().sum()
```

Out[9]:

```
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
```

In [10]:

```
df1.isnull().mean()
```

Out[10]:

```
id               0.000000
date             0.000000
price            0.000000
bedrooms         0.000000
bathrooms        0.000000
sqft_living      0.000000
sqft_lot         0.000000
floors           0.000000
waterfront       0.110015
view             0.002917
condition        0.000000
grade            0.000000
sqft_above       0.000000
sqft_basement    0.000000
yr_built         0.000000
yr_renovated     0.177895
zipcode          0.000000
lat              0.000000
long             0.000000
sqft_living15    0.000000
sqft_lot15       0.000000
dtype: float64
```

In [11]:

```
df1['sqft_basement'].unique()
```

Out[11]:

```
array(['0.0', '400.0', '910.0', '1530.0', '?', '730.0', '1700.0', '300.0',
       '970.0', '760.0', '720.0', '700.0', '820.0', '780.0', '790.0',
       '330.0', '1620.0', '360.0', '588.0', '1510.0', '410.0', '990.0',
       '600.0', '560.0', '550.0', '1000.0', '1600.0', '500.0', '1040.0',
       '880.0', '1010.0', '240.0', '265.0', '290.0', '800.0', '540.0',
       '710.0', '840.0', '380.0', '770.0', '480.0', '570.0', '1490.0',
       '620.0', '1250.0', '1270.0', '120.0', '650.0', '180.0', '1130.0',
       '450.0', '1640.0', '1460.0', '1020.0', '1030.0', '750.0', '640.0',
       '1070.0', '490.0', '1310.0', '630.0', '2000.0', '390.0', '430.0',
       '850.0', '210.0', '1430.0', '1950.0', '440.0', '220.0', '1160.0',
```

```
       '860.0', '580.0', '2060.0', '1820.0', '1180.0', '200.0', '1150.0',
       '1200.0', '680.0', '530.0', '1450.0', '1170.0', '1080.0', '960.0',
       '280.0', '870.0', '1100.0', '460.0', '1400.0', '660.0', '1220.0',
       '900.0', '420.0', '1580.0', '1380.0', '475.0', '690.0', '270.0',
       '350.0', '935.0', '1370.0', '980.0', '1470.0', '160.0', '950.0',
       '50.0', '740.0', '1780.0', '1900.0', '340.0', '470.0', '370.0',
       '140.0', '1760.0', '130.0', '520.0', '890.0', '1110.0', '150.0',
       '1720.0', '810.0', '190.0', '1290.0', '670.0', '1800.0', '1120.0',
       '1810.0', '60.0', '1050.0', '940.0', '310.0', '930.0', '1390.0',
       '610.0', '1830.0', '1300.0', '510.0', '1330.0', '1590.0', '920.0',
       '1320.0', '1420.0', '1240.0', '1960.0', '1560.0', '2020.0',
       '1190.0', '2110.0', '1280.0', '250.0', '2390.0', '1230.0', '170.0',
       '830.0', '1260.0', '1410.0', '1340.0', '590.0', '1500.0', '1140.0',
       '260.0', '100.0', '320.0', '1480.0', '1060.0', '1284.0', '1670.0',
       '1350.0', '2570.0', '1090.0', '110.0', '2500.0', '90.0', '1940.0',
       '1550.0', '2350.0', '2490.0', '1481.0', '1360.0', '1135.0',
       '1520.0', '1850.0', '1660.0', '2130.0', '2600.0', '1690.0',
       '243.0', '1210.0', '1024.0', '1798.0', '1610.0', '1440.0',
       '1570.0', '1650.0', '704.0', '1910.0', '1630.0', '2360.0',
       '1852.0', '2090.0', '2400.0', '1790.0', '2150.0', '230.0', '70.0',
       '1680.0', '2100.0', '3000.0', '1870.0', '1710.0', '2030.0',
       '875.0', '1540.0', '2850.0', '2170.0', '506.0', '906.0', '145.0',
       '2040.0', '784.0', '1750.0', '374.0', '518.0', '2720.0', '2730.0',
       '1840.0', '3480.0', '2160.0', '1920.0', '2330.0', '1860.0',
       '2050.0', '4820.0', '1913.0', '80.0', '2010.0', '3260.0', '2200.0',
       '415.0', '1730.0', '652.0', '2196.0', '1930.0', '515.0', '40.0',
       '2080.0', '2580.0', '1548.0', '1740.0', '235.0', '861.0', '1890.0',
       '2220.0', '792.0', '2070.0', '4130.0', '2250.0', '2240.0',
       '1990.0', '768.0', '2550.0', '435.0', '1008.0', '2300.0', '2610.0',
       '666.0', '3500.0', '172.0', '1816.0', '2190.0', '1245.0', '1525.0',
       '1880.0', '862.0', '946.0', '1281.0', '414.0', '2180.0', '276.0',
       '1248.0', '602.0', '516.0', '176.0', '225.0', '1275.0', '266.0',
       '283.0', '65.0', '2310.0', '10.0', '1770.0', '2120.0', '295.0',
       '207.0', '915.0', '556.0', '417.0', '143.0', '508.0', '2810.0',
       '20.0', '274.0', '248.0'], dtype=object)
```

In [12]:

```
#we have noticed that our data set has a ? on the column sqft_basement.lets convert into
a null value
df1.replace('?', pd.NA, inplace=True)
```

The null values appear on the waterfront,view and yr_renovated columns. The percentages are small with the highest percentage being 17%. The percentage in View was small,hence dropped.The column waterfront missing values was replaced with NONE.The column yr_renovated missing values was replaced with mode(0)

In [13]:

```
#Drop missing values in the column View
df1 = df1.dropna(subset=['view', 'sqft_basement'],axis=0)
df1
```

Out[13]:

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

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	s
21592	263000018	5/21/2014	360000.0	3	2.50	1530	1131	3.0	NO	NONE	...	8 Good	
21593	6600060120	2/23/2015	400000.0	4	2.50	2310	5813	2.0	NO	NONE	...	8 Good	
21594	1523300141	6/23/2014	402101.0	2	0.75	1020	1350	2.0	NO	NONE	...	7 Average	
21595	291310100	1/16/2015	400000.0	3	2.50	1600	2388	2.0	NaN	NONE	...	8 Good	
21596	1523300157	10/15/2014	325000.0	2	0.75	1020	1076	2.0	NO	NONE	...	7 Average	

21082 rows × 21 columns

In [14]:

```
df1.isnull().sum()
```

Out[14]:

```
id                  0
date                0
price               0
bedrooms            0
bathrooms           0
sqft_living         0
sqft_lot            0
floors              0
waterfront       2333
view                0
condition           0
grade               0
sqft_above          0
sqft_basement       0
yr_built            0
yr_renovated     3742
zipcode             0
lat                 0
long                0
sqft_living15       0
sqft_lot15          0
dtype: int64
```

In [15]:

```
# Replacing null values with NONE
df1['waterfront'].fillna('NONE', inplace=True)
df1.head()
```

Out[15]:

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

5 rows × 21 columns

In [16]:

In [16]:

```
df1.shape
```

Out[16]:

```
(21082, 21)
```

In [17]:

```
mode_year = df1['yr_renovated'].mode()[0]
```

In [18]:

```
#Replace the null values in yr_renovated column with the most repeated year
df1.fillna(mode_year,inplace=True)
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21082 entries, 0 to 21596
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21082 non-null  int64
 1   date           21082 non-null  object
 2   price          21082 non-null  float64
 3   bedrooms       21082 non-null  int64
 4   bathrooms      21082 non-null  float64
 5   sqft_living    21082 non-null  int64
 6   sqft_lot       21082 non-null  int64
 7   floors         21082 non-null  float64
 8   waterfront     21082 non-null  object
 9   view           21082 non-null  object
 10  condition      21082 non-null  object
 11  grade          21082 non-null  object
 12  sqft_above     21082 non-null  int64
 13  sqft_basement  21082 non-null  object
 14  yr_built       21082 non-null  int64
 15  yr_renovated   21082 non-null  float64
 16  zipcode        21082 non-null  int64
 17  lat            21082 non-null  float64
 18  long           21082 non-null  float64
 19  sqft_living15  21082 non-null  int64
 20  sqft_lot15     21082 non-null  int64
dtypes: float64(6), int64(9), object(6)
memory usage: 3.5+ MB
```

Our Dataset has 21 columns, 21534 rows and no missing values.

DROPPING IRRELEVANT VALUES

In [19]:

```
#dropping the ID column as it seems to be irrelevant
df1 = df1.drop(columns=['id'])
df1.head()
```

Out[19]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above
0	10/13/2014	221900.0	3	1.00	1180	5650	1.0	NONE	NONE	Average	7 Average	1180
1	12/9/2014	538000.0	3	2.25	2570	7242	2.0	NO	NONE	Average	7 Average	2170
2	2/25/2015	180000.0	2	1.00	770	10000	1.0	NO	NONE	Average	6 Low Average	770
3	12/9/2014	604000.0	4	3.00	1960	5000	1.0	NO	NONE	Very Good	7 Average	1050

4	2/18/2015 date	510000.0 price	bedrooms 3	bathrooms 2.00	sqft_living 1680	sqft_lot 8080	floors 1.0	waterfront NO	NONE view	condition Average	8 Good grade	sqft_above 1680

DUPLICATED VALUES

In [20]:

```
df1.duplicated().any
```

Out[20]:

```
<bound method Series.any of 0        False
1        False
2        False
3        False
4        False
         ...
21592    False
21593    False
21594    False
21595    False
21596    False
Length: 21082, dtype: bool>
```

We dont have any duplicated values

In [21]:

```
df1.shape
```

Out[21]:

```
(21082, 20)
```

CHANGING THE DATA TYPES

In [22]:

```
#The floors and yr_renovated data types need to be changed.
df1['floors'] = df1['floors'].astype(int)
#df['sqft_basement'] = df['sqft_basement'].astype(int)
df1['yr_renovated'] = df1['yr_renovated'].astype(int)
df1['sqft_basement'] = df1['sqft_basement'].astype(float)
```

In [23]:

```
df1.info()
```

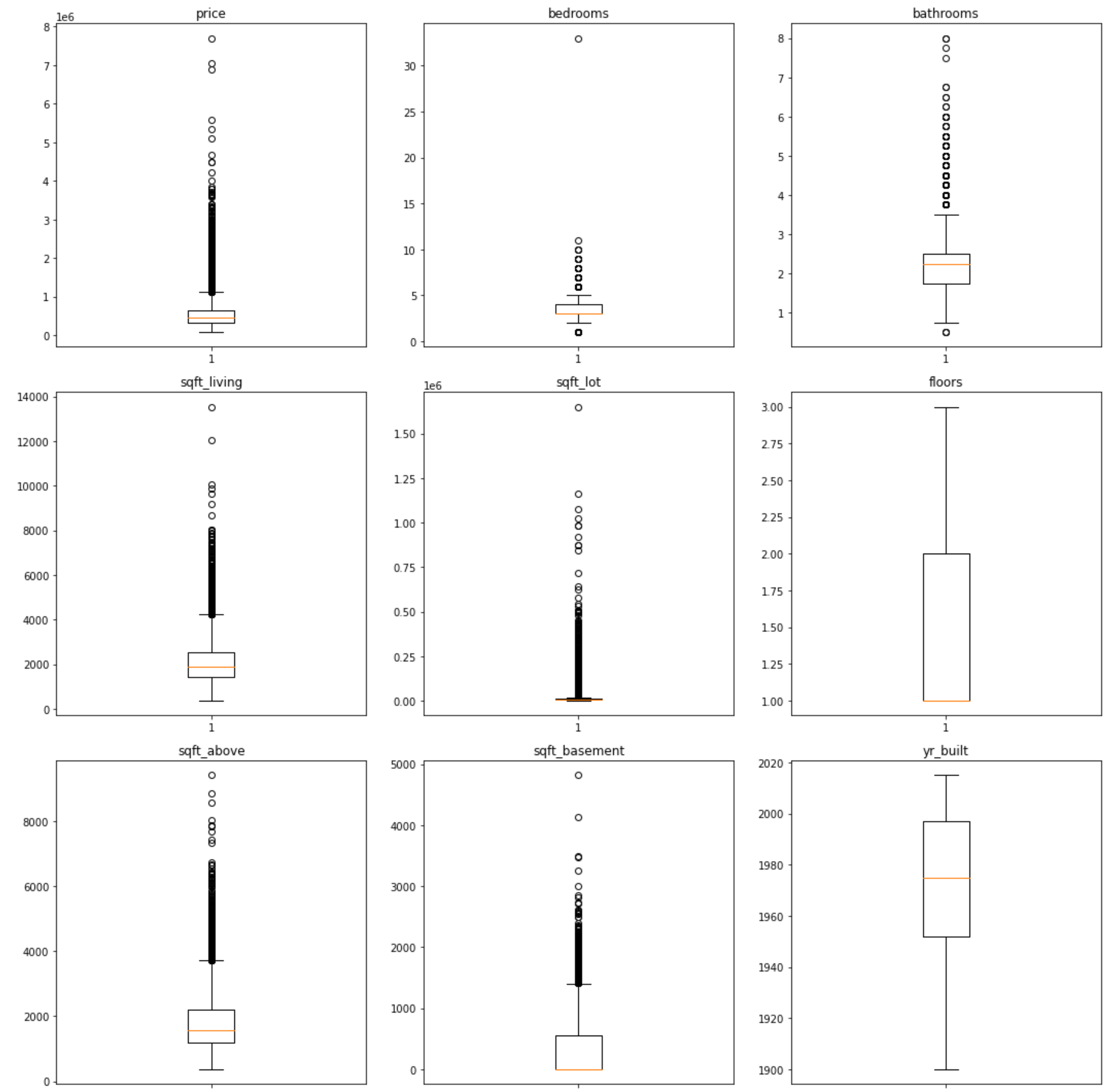
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21082 entries, 0 to 21596
Data columns (total 20 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   date            21082 non-null  object
 1   price           21082 non-null  float64
 2   bedrooms        21082 non-null  int64
 3   bathrooms       21082 non-null  float64
 4   sqft_living     21082 non-null  int64
 5   sqft_lot        21082 non-null  int64
 6   floors          21082 non-null  int32
 7   waterfront      21082 non-null  object
 8   view            21082 non-null  object
 9   condition       21082 non-null  object
 10  grade           21082 non-null  object
 11  sqft_above      21082 non-null  int64
 12  sqft_basement   21082 non-null  float64
 13  yr_built        21082 non-null  int64
 14  yr_renovated    21082 non-null  int32
 15  zipcode         21082 non-null  int64
 16  lat             21082 non-null  float64
```

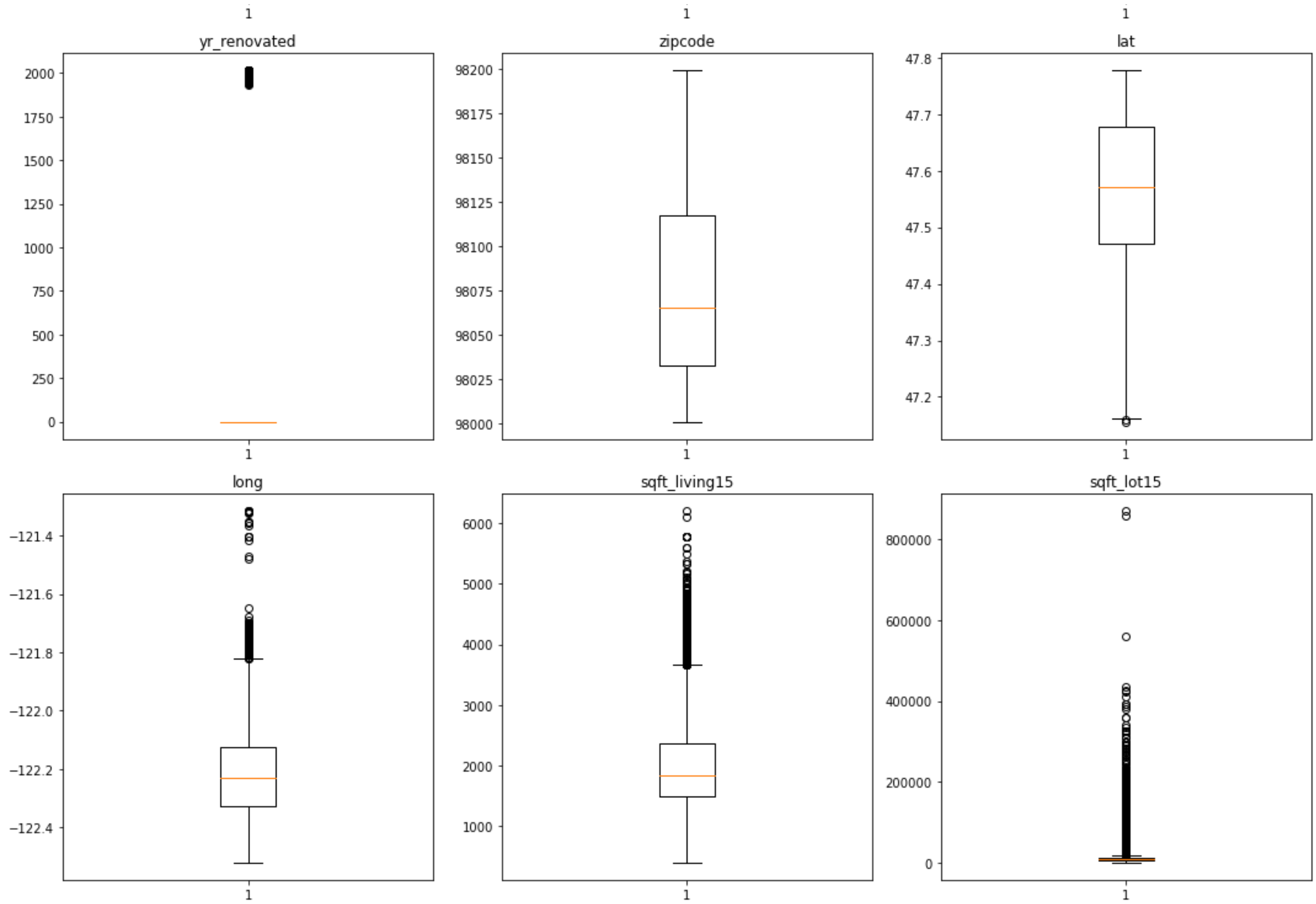
```
 16  lat            21082 non-null  float64
 17  long           21082 non-null  float64
 18  sqft_living15  21082 non-null  int64
 19  sqft_lot15     21082 non-null  int64
dtypes: float64(5), int32(2), int64(8), object(5)
memory usage: 3.2+ MB
```

OUTLIERS

In [24]:

```python
numeric_columns = df1.select_dtypes(include=['float64', 'int64', 'int32'])

# Plot box plots for each numeric column
num_cols = len(numeric_columns.columns)
cols_per_row = 3
num_rows = (num_cols - 1) // cols_per_row + 1

plt.figure(figsize=(15, 5 * num_rows))
for i, col in enumerate(numeric_columns.columns):
    plt.subplot(num_rows, cols_per_row, i+1)
    plt.boxplot(numeric_columns[col])
    plt.title(col)
plt.tight_layout()
plt.show()
```





We have noted that we do not have outliers in the floors, yr_built, zipcode and latitude

SKEWNESS AND KURTOSIS

In [25]:

```
df1.price.skew()
```

Out[25]:

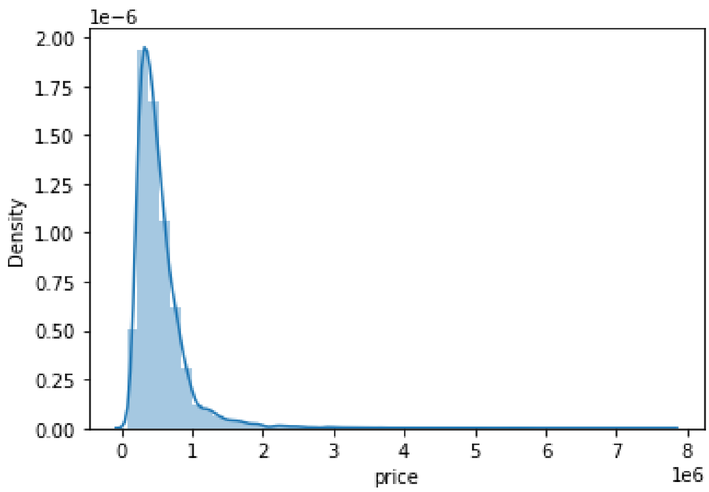
```
3.9864235583473797
```

In [26]:

```
sns.distplot(df1['price'])
```

Out[26]:

```
<AxesSubplot:xlabel='price', ylabel='Density'>
```



A skewness of 4 indicates that the distibution of data in the price column is positively skewed. In regression

analysis, highly skewed target variables can violate assumptions of normality and homoscedasticity hence the need to remove outliers

In [27]:

```
import pandas as pd
from scipy import stats

for column in df1.columns:
    if df1[column].dtype in ['int64', 'float64'] and column not in ['lat', "id", "yr_ren
ovated", "zipcode", 'long']:
        # Calculate skewness including NaN values
        column_skew = stats.skew(df1[column].dropna())
        column_kurtosis = stats.kurtosis(df1[column].dropna())

        print(f"{column}: Skewness = {column_skew}, Kurtosis = {column_kurtosis}")
```

```
price: Skewness = 3.9861399157364543, Kurtosis = 34.06048946333713
bedrooms: Skewness = 2.0676579682064817, Kurtosis = 51.30311750509116
bathrooms: Skewness = 0.5159339157923863, Kurtosis = 1.2700320775132043
sqft_living: Skewness = 1.4751290520336788, Kurtosis = 5.285155141668961
sqft_lot: Skewness = 13.109625412685952, Kurtosis = 289.6613994250277
sqft_above: Skewness = 1.4545738313845544, Kurtosis = 3.4493589341150575
sqft_basement: Skewness = 1.5736432185240308, Kurtosis = 2.6985933943644964
yr_built: Skewness = -0.4707948263068775, Kurtosis = -0.6502440458482863
sqft_living15: Skewness = 1.1098692917019588, Kurtosis = 1.604595686027447
sqft_lot15: Skewness = 9.592239187088412, Kurtosis = 154.46697770724586
```

If the kurtosis value is greater than 3 (positive kurtosis): The distribution has heavier tails and a sharper peak compared to a normal distribution. This suggests the presence of more outliers or extreme values in the dataset.

If the kurtosis value is less than 3 (negative kurtosis): The distribution has lighter tails and is flatter compared to a normal distribution. This suggests fewer outliers and a more dispersed distribution of data.

Kurtosis values closer to 3 indicate a distribution similar to the normal distribution in terms of tail behavior.

REMOVING OUTLIERS

Instead of dropping them, we consider replacing them with upper limit and lower limit values using z score method

Z SCORE METHOD

In [28]:

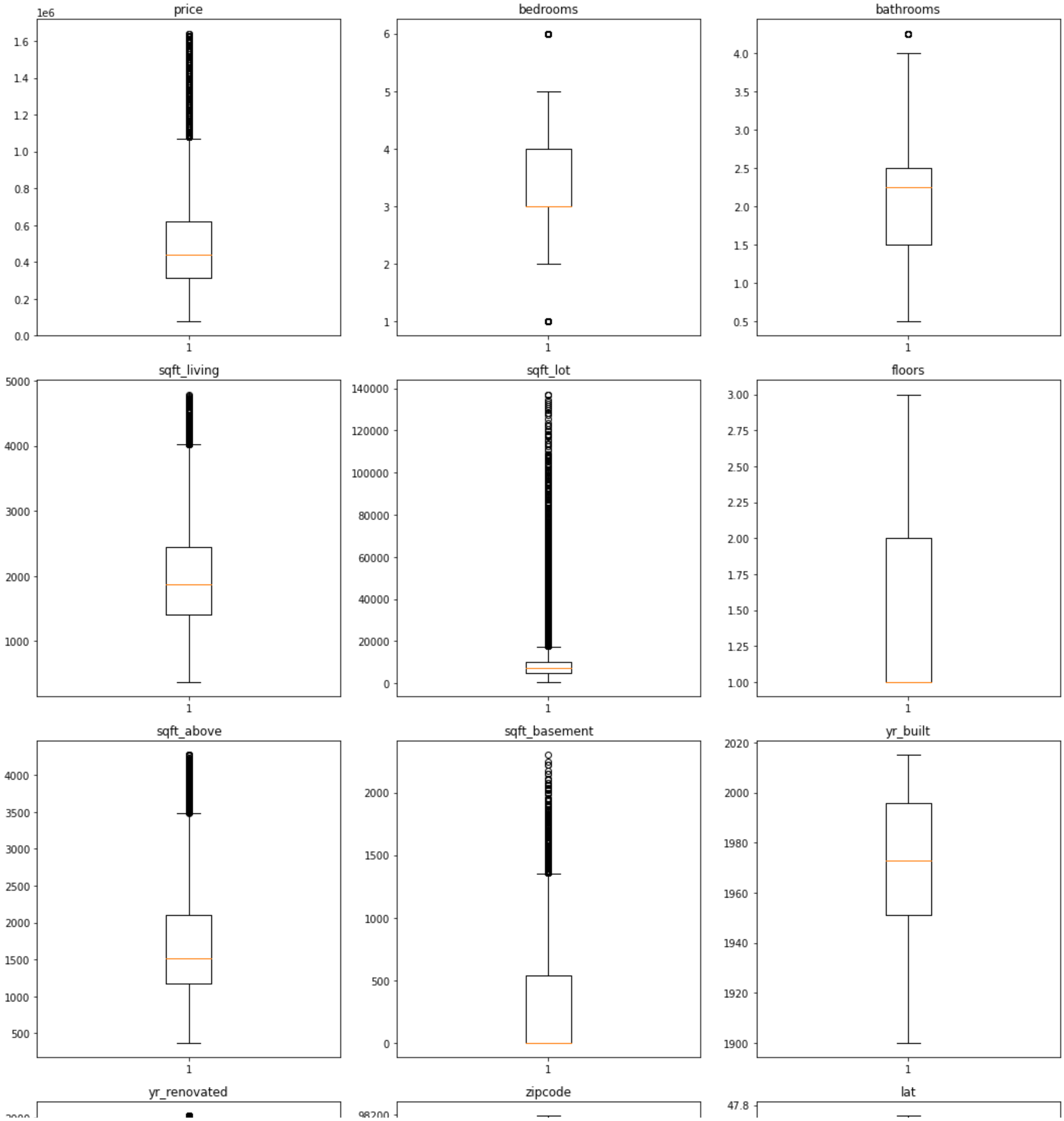
```
# Columns to check for outliers
columns_to_check = ['bathrooms', 'bedrooms', 'sqft_living', 'sqft_above', 'price', 'sqft
_lot', 'sqft_above', 'long', 'sqft_living15', 'sqft_lot15']

# Calculate z-scores for selected columns
z_scores = np.abs((df1[columns_to_check] - df1[columns_to_check].mean()) / df1[columns_to
_check].std())

# Define threshold for outlier detection (e.g., z-score > 3)
threshold = 3

# Identify rows with outliers based on z-scores
outlier_mask = (z_scores > threshold).any(axis=1)

# Filter out rows containing outliers
data_filtered = df1[~outlier_mask]

# Display summary (optional)
print(f"Original DataFrame shape: {df1.shape}")
print(f"Filtered DataFrame shape (after removing outliers): {data_filtered.shape}")
```

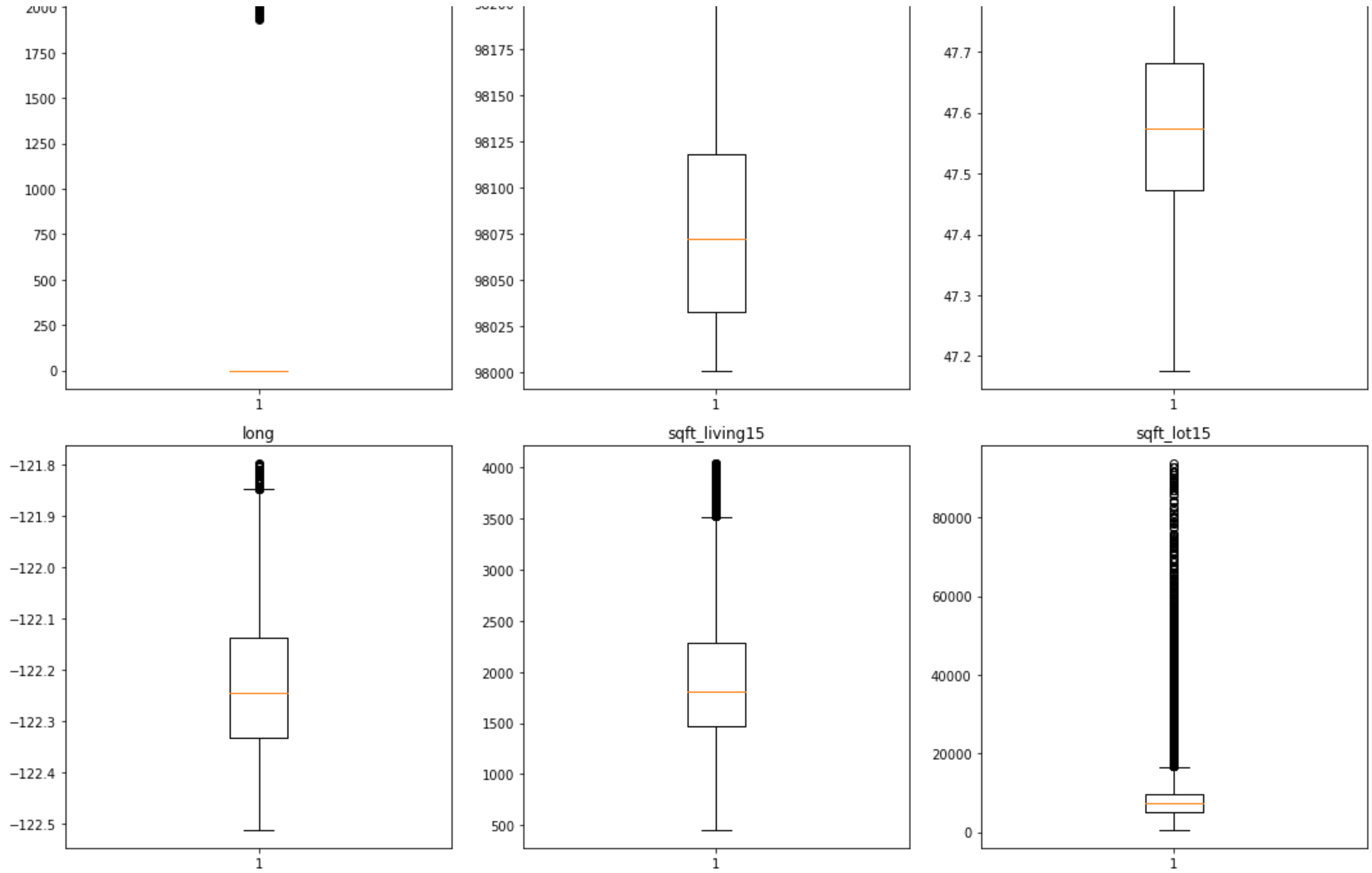
```
# Optionally, you can assign the filtered DataFrame back to 'data' if needed
data = data_filtered
```

```
Original DataFrame shape: (21082, 20)
Filtered DataFrame shape (after removing outliers): (19685, 20)
```

In [29]:

```
#lets check if the data still has outliers.
numeric_columns = data.select_dtypes(include=['float64', 'int64', 'int32'])

# Plot box plots for each numeric column
num_cols = len(numeric_columns.columns)
cols_per_row = 3
num_rows = (num_cols - 1) // cols_per_row + 1

plt.figure(figsize=(15, 5 * num_rows))
for i, col in enumerate(numeric_columns.columns):
    plt.subplot(num_rows, cols_per_row, i+1)
    plt.boxplot(numeric_columns[col])
    plt.title(col)
plt.tight_layout()
plt.show()
```





FEATURE ENGINEERING

In [30]:

```
#Lets capitalise the column titles.
data.columns = [col.capitalize() for col in data.columns]
data.columns
```

Out[30]:

```
Index(['Date', 'Price', 'Bedrooms', 'Bathrooms', 'Sqft_living', 'Sqft_lot',
       'Floors', 'Waterfront', 'View', 'Condition', 'Grade', 'Sqft_above',
       'Sqft_basement', 'Yr_built', 'Yr_renovated', 'Zipcode', 'Lat', 'Long',
       'Sqft_living15', 'Sqft_lot15'],
      dtype='object')
```

In [31]:

```
data.head()
```

Out[31]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	Grade	Sqft_abo
0	10/13/2014	221900.0	3	1.00	1180	5650	1	NONE	NONE	Average	7 Average	11
1	12/9/2014	538000.0	3	2.25	2570	7242	2	NO	NONE	Average	7 Average	21
2	2/25/2015	180000.0	2	1.00	770	10000	1	NO	NONE	Average	6 Low Average	7
3	12/9/2014	604000.0	4	3.00	1960	5000	1	NO	NONE	Very Good	7 Average	10
4	2/18/2015	510000.0	3	2.00	1680	8080	1	NO	NONE	Average	8 Good	16

In [32]:

```
#checking for leading & trailing whitespaces and removing them in our datasets
```

```
#checking for leading & trailing whitespaces and removing them in our datasets
[col.strip() for col in data.columns]
```

Out[32]:

```
['Date',
 'Price',
 'Bedrooms',
 'Bathrooms',
 'Sqft_living',
 'Sqft_lot',
 'Floors',
 'Waterfront',
 'View',
 'Condition',
 'Grade',
 'Sqft_above',
 'Sqft_basement',
 'Yr_built',
 'Yr_renovated',
 'Zipcode',
 'Lat',
 'Long',
 'Sqft_living15',
 'Sqft_lot15']
```

In [33]:

```
# Lets make sure the date column is in dd/mm/yy format
data['Date'] = pd.to_datetime(data['Date'])

# Format the date column to dd/mm/yy format
data['Date'] = data['Date'].dt.strftime('%d/%m/%y')
data.head()
```

Out[33]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	Grade	Sqft_above
0	13/10/14	221900.0	3	1.00	1180	5650	1	NONE	NONE	Average	7 Average	1180
1	09/12/14	538000.0	3	2.25	2570	7242	2	NO	NONE	Average	7 Average	2170
2	25/02/15	180000.0	2	1.00	770	10000	1	NO	NONE	Average	6 Low Average	770
3	09/12/14	604000.0	4	3.00	1960	5000	1	NO	NONE	Very Good	7 Average	1050
4	18/02/15	510000.0	3	2.00	1680	8080	1	NO	NONE	Average	8 Good	1680

Now the date column is in date/month/year format.

Using the Date Column to come up with a new column called seasons

In [34]:

```
# Convert 'Date' column to datetime format
data['Date'] = pd.to_datetime(data['Date'])

# Extract month to determine the season
data['Month'] = data['Date'].dt.month

# Define a function to map month to season
def get_season(month):
    if month in [12, 1, 2]:  # Winter: December, January, February
        return 'Winter'
    elif month in [3, 4, 5]:  # Spring: March, April, May
```

```
        return 'Spring'
    elif month in [6, 7, 8]:  # Summer: June, July, August
        return 'Summer'
    else:  # Fall: September, October, November
        return 'Fall'

# Apply the function to create the 'seasons' column
data['Seasons'] = data['Month'].apply(get_season)

# Drop the 'Month' column if not needed
data.drop('Month', axis=1, inplace=True)
```

In [35]:

```
data.head()
```

Out[35]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	...	Sqft_above	Sqft_ba
0	2014-10-13	221900.0	3	1.00	1180	5650	1	NONE	NONE	Average	...	1180	
1	2014-09-12	538000.0	3	2.25	2570	7242	2	NO	NONE	Average	...	2170	
2	2015-02-25	180000.0	2	1.00	770	10000	1	NO	NONE	Average	...	770	
3	2014-09-12	604000.0	4	3.00	1960	5000	1	NO	NONE	Very Good	...	1050	
4	2015-02-18	510000.0	3	2.00	1680	8080	1	NO	NONE	Average	...	1680	

5 rows × 21 columns

In [36]:

```
#LETS SPLIT THE  COLUMN GRADE INTO TWO, COLUMN GRADING
data['Grading'] = data["Grade"].str.split().apply(lambda x: x[0])
data
```

Out[36]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	...	Sqft_basement
0	2014-10-13	221900.0	3	1.00	1180	5650	1	NONE	NONE	Average	...	0.0
1	2014-09-12	538000.0	3	2.25	2570	7242	2	NO	NONE	Average	...	400.0
2	2015-02-25	180000.0	2	1.00	770	10000	1	NO	NONE	Average	...	0.0
3	2014-09-12	604000.0	4	3.00	1960	5000	1	NO	NONE	Very Good	...	910.0
4	2015-02-18	510000.0	3	2.00	1680	8080	1	NO	NONE	Average	...	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...
21592	2014-05-21	360000.0	3	2.50	1530	1131	3	NO	NONE	Average	...	0.0
21593	2015-02-23	400000.0	4	2.50	2310	5813	2	NO	NONE	Average	...	0.0
21594	2014-06-23	402101.0	2	0.75	1020	1350	2	NO	NONE	Average	...	0.0
21595	2015-01-16	400000.0	3	2.50	1600	2388	2	NONE	NONE	Average	...	0.0
	2014-											

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	...	Sqft_basement
21596	10-15	325000.0	2	0.75	1020	1076	2	NO	NONE	Average	...	0.0

19685 rows × 22 columns

In [37]:

```
#lets change the data type of grading into an int
data['Grading'] = data['Grading'].astype(int)
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19685 entries, 0 to 21596
Data columns (total 22 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Date           19685 non-null  datetime64[ns]
 1   Price          19685 non-null  float64
 2   Bedrooms       19685 non-null  int64
 3   Bathrooms      19685 non-null  float64
 4   Sqft_living    19685 non-null  int64
 5   Sqft_lot       19685 non-null  int64
 6   Floors         19685 non-null  int32
 7   Waterfront     19685 non-null  object
 8   View           19685 non-null  object
 9   Condition      19685 non-null  object
 10  Grade          19685 non-null  object
 11  Sqft_above     19685 non-null  int64
 12  Sqft_basement  19685 non-null  float64
 13  Yr_built       19685 non-null  int64
 14  Yr_renovated   19685 non-null  int32
 15  Zipcode        19685 non-null  int64
 16  Lat            19685 non-null  float64
 17  Long           19685 non-null  float64
 18  Sqft_living15  19685 non-null  int64
 19  Sqft_lot15     19685 non-null  int64
 20  Seasons        19685 non-null  object
 21  Grading        19685 non-null  int32
dtypes: datetime64[ns](1), float64(5), int32(3), int64(8), object(5)
memory usage: 3.2+ MB
```

ONE HOT ENCODING

In [38]:

```
#Setting up our feature and target variable
x = data.iloc[:, 2:]  # Selecting features (all columns except the first)
y = data.iloc[:, 1]  # Selecting the target variable (first column)
print(x)
print(y)
```

```
       Bedrooms  Bathrooms  Sqft_living  Sqft_lot  Floors Waterfront  View  \
0             3       1.00         1180      5650       1       NONE  NONE
1             3       2.25         2570      7242       2         NO  NONE
2             2       1.00          770     10000       1         NO  NONE
3             4       3.00         1960      5000       1         NO  NONE
4             3       2.00         1680      8080       1         NO  NONE
...         ...        ...          ...       ...     ...        ...   ...
21592         3       2.50         1530      1131       3         NO  NONE
21593         4       2.50         2310      5813       2         NO  NONE
21594         2       0.75         1020      1350       2         NO  NONE
21595         3       2.50         1600      2388       2       NONE  NONE
21596         2       0.75         1020      1076       2         NO  NONE

       Condition          Grade  Sqft_above  Sqft_basement  Yr_built  \
0        Average      7 Average        1180            0.0      1955
1        Average      7 Average        2170          400.0      1951
2        Average  6 Low Average         770            0.0      1933
3      Very Good      7 Average        1050          910.0      1965
4        Average         8 Good        1680            0.0      1987
...          ...            ...         ...            ...       ...
```

```
21592    Average         8  Good        1530            0.0      2009
21593    Average         8  Good        2310            0.0      2014
21594    Average      7  Average        1020            0.0      2009
21595    Average         8  Good        1600            0.0      2004
21596    Average      7  Average        1020            0.0      2008

       Yr_renovated  Zipcode      Lat     Long  Sqft_living15  Sqft_lot15  \
0                 0    98178  47.5112 -122.257           1340        5650
1              1991    98125  47.7210 -122.319           1690        7639
2                 0    98028  47.7379 -122.233           2720        8062
3                 0    98136  47.5208 -122.393           1360        5000
4                 0    98074  47.6168 -122.045           1800        7503
...             ...      ...      ...      ...            ...         ...
21592             0    98103  47.6993 -122.346           1530        1509
21593             0    98146  47.5107 -122.362           1830        7200
21594             0    98144  47.5944 -122.299           1020        2007
21595             0    98027  47.5345 -122.069           1410        1287
21596             0    98144  47.5941 -122.299           1020        1357

      Seasons  Grading
0        Fall        7
1        Fall        7
2      Winter        6
3        Fall        7
4      Winter        8
...       ...      ...
21592  Spring        8
21593  Winter        8
21594  Summer        7
21595  Winter        8
21596    Fall        7

[19685 rows x 20 columns]
0        221900.0
1        538000.0
2        180000.0
3        604000.0
4        510000.0
           ...
21592    360000.0
21593    400000.0
21594    402101.0
21595    400000.0
21596    325000.0
Name: Price, Length: 19685, dtype: float64
```

In [39]:

```
#converting the column into categorical columns
house_cond = pd.get_dummies(x['Condition'], drop_first=True, dtype=int)
view_1 = pd.get_dummies(x['View'], drop_first=True, dtype=int)
water_front = pd.get_dummies(x['Waterfront'], drop_first=True, dtype=int)
seasons1 = pd.get_dummies(x['Seasons'], drop_first=True, dtype=int)
```

In [40]:

```
#drop the condition, waterfront, View and Grade column
x = x.drop('Condition', axis=1)
x= x.drop('Waterfront', axis=1)
x = x.drop('View', axis=1)
x = x.drop('Grade', axis=1)
x = x.drop('Seasons', axis=1)
print(x)
```

```
       Bedrooms  Bathrooms  Sqft_living  Sqft_lot  Floors  Sqft_above  \
0             3       1.00         1180      5650       1        1180
1             3       2.25         2570      7242       2        2170
2             2       1.00          770     10000       1         770
3             4       3.00         1960      5000       1        1050
4             3       2.00         1680      8080       1        1680
...         ...        ...          ...       ...     ...         ...
21592         3       2.50         1530      1131       3        1530
```

```
21593        4       2.50       2310     5813      2       2310
21594        2       0.75       1020     1350      2       1020
21595        3       2.50       1600     2388      2       1600
21596        2       0.75       1020     1076      2       1020

       Sqft_basement  Yr_built  Yr_renovated  Zipcode      Lat     Long  \
0                0.0      1955             0    98178  47.5112 -122.257
1              400.0      1951          1991    98125  47.7210 -122.319
2                0.0      1933             0    98028  47.7379 -122.233
3              910.0      1965             0    98136  47.5208 -122.393
4                0.0      1987             0    98074  47.6168 -122.045
...              ...       ...           ...      ...      ...      ...
21592            0.0      2009             0    98103  47.6993 -122.346
21593            0.0      2014             0    98146  47.5107 -122.362
21594            0.0      2009             0    98144  47.5944 -122.299
21595            0.0      2004             0    98027  47.5345 -122.069
21596            0.0      2008             0    98144  47.5941 -122.299

       Sqft_living15  Sqft_lot15  Grading
0               1340        5650        7
1               1690        7639        7
2               2720        8062        6
3               1360        5000        7
4               1800        7503        8
...              ...         ...      ...
21592           1530        1509        8
21593           1830        7200        8
21594           1020        2007        7
21595           1410        1287        8
21596           1020        1357        7

[19685 rows x 15 columns]
```

In [41]:

```
#concertinate our dummy variables to our data
x = pd.concat([data,house_cond, view_1, water_front],axis=1)
x
```

Out[41]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	...	Fair	Good	Po
0	2014-10-13	221900.0	3	1.00	1180	5650	1	NONE	NONE	Average	...	0	0	
1	2014-09-12	538000.0	3	2.25	2570	7242	2	NO	NONE	Average	...	0	0	
2	2015-02-25	180000.0	2	1.00	770	10000	1	NO	NONE	Average	...	0	0	
3	2014-09-12	604000.0	4	3.00	1960	5000	1	NO	NONE	Very Good	...	0	0	
4	2015-02-18	510000.0	3	2.00	1680	8080	1	NO	NONE	Average	...	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
21592	2014-05-21	360000.0	3	2.50	1530	1131	3	NO	NONE	Average	...	0	0	
21593	2015-02-23	400000.0	4	2.50	2310	5813	2	NO	NONE	Average	...	0	0	
21594	2014-06-23	402101.0	2	0.75	1020	1350	2	NO	NONE	Average	...	0	0	
21595	2015-01-16	400000.0	3	2.50	1600	2388	2	NONE	NONE	Average	...	0	0	
21596	2014-10-15	325000.0	2	0.75	1020	1076	2	NO	NONE	Average	...	0	0	

19685 rows × 32 columns

In [42]:

```
print(x.dtypes)
```

```
Date                datetime64[ns]
Price                      float64
Bedrooms                     int64
Bathrooms                  float64
Sqft_living                  int64
Sqft_lot                     int64
Floors                       int32
Waterfront                  object
View                        object
Condition                   object
Grade                       object
Sqft_above                   int64
Sqft_basement              float64
Yr_built                     int64
Yr_renovated                 int32
Zipcode                      int64
Lat                        float64
Long                       float64
Sqft_living15                int64
Sqft_lot15                   int64
Seasons                     object
Grading                      int32
Fair                         int32
Good                         int32
Poor                         int32
Very Good                    int32
EXCELLENT                    int32
FAIR                         int32
GOOD                         int32
NONE                         int32
NONE                         int32
YES                          int32
dtype: object
```

COLINEARITY AND MULTICOLINEARITY

CHECKING FOR COLLINEARITY BETWEEN THE DEPENDENT(PRICE) AND INDEPENDENT VARIABLES

In [43]:

```
data_corr = data.corr()['Price'].map(abs).sort_values(ascending=False)
data_corr
```

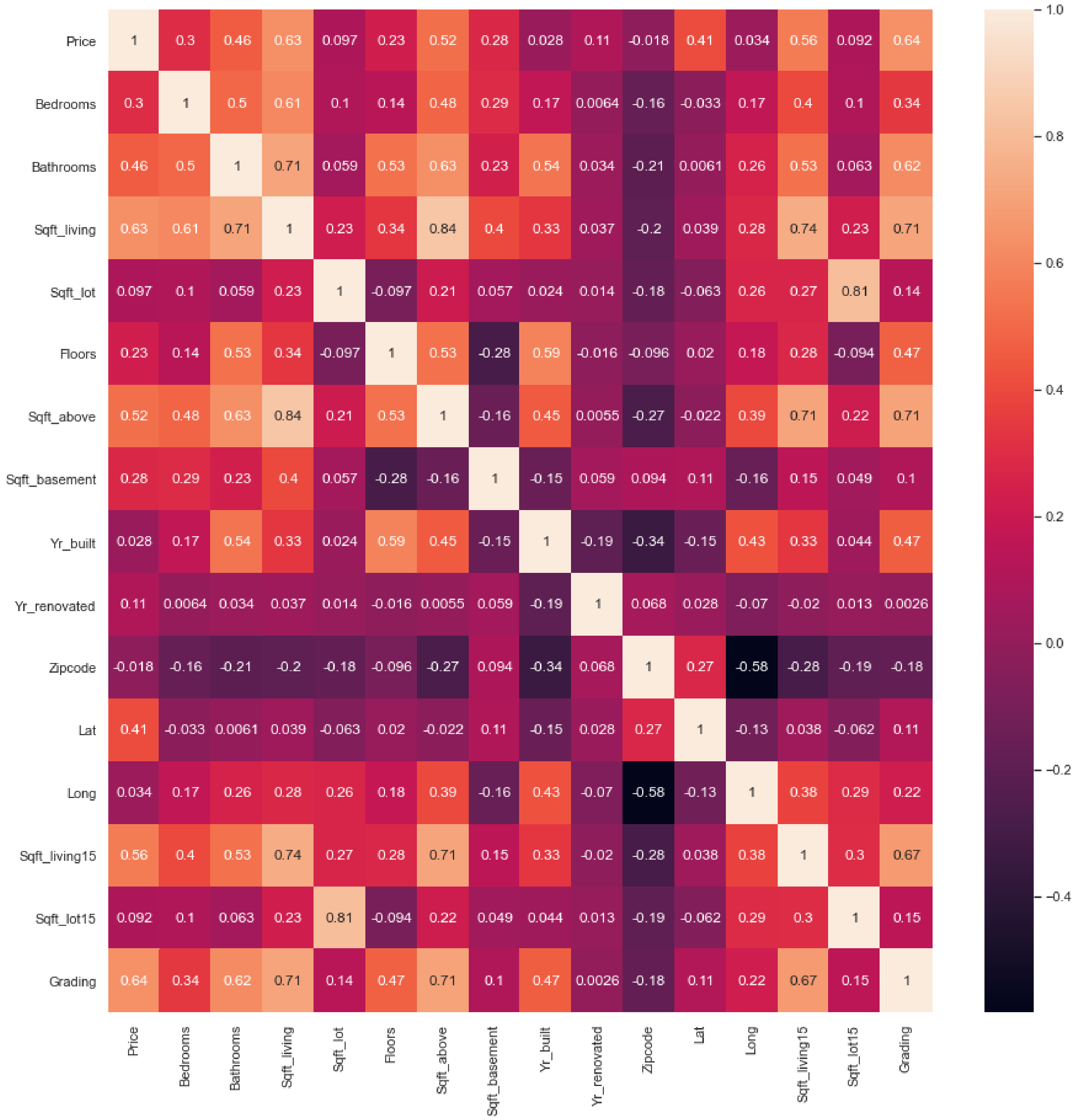
Out[43]:

```
Price            1.000000
Grading          0.640072
Sqft_living      0.631970
Sqft_living15    0.561656
Sqft_above       0.518802
Bathrooms        0.455976
Lat              0.407917
Bedrooms         0.298772
Sqft_basement    0.276792
Floors           0.231544
Yr_renovated     0.106911
Sqft_lot         0.097231
Sqft_lot15       0.092238
Long             0.034362
Yr_built         0.028031
Zipcode          0.017773
Name: Price, dtype: float64
```

In [44]:

In [44]:

```
sns.set(rc={'figure.figsize':(15, 15)})
#Use the .heatmap method to depict the relationship visually

sns.heatmap(data.corr(), annot=True, annot_kws={"size": 12})
plt.show()
```



Step 1: Visualize Relationships Between Features and Target For each feature in the subset, create a scatter plot that shows the feature on the x-axis and SalePrice on the y-axis to identify linear relationships. We will first split our columns into 3 subsets for better visualization.

In [45]:

```
#Visulaizing linear relationship between features and target variables
data_subset = data[['Sqft_living', 'Sqft_lot', 'Bedrooms', 'Bathrooms','Price']].copy()
data_subset
```

Out[45]:

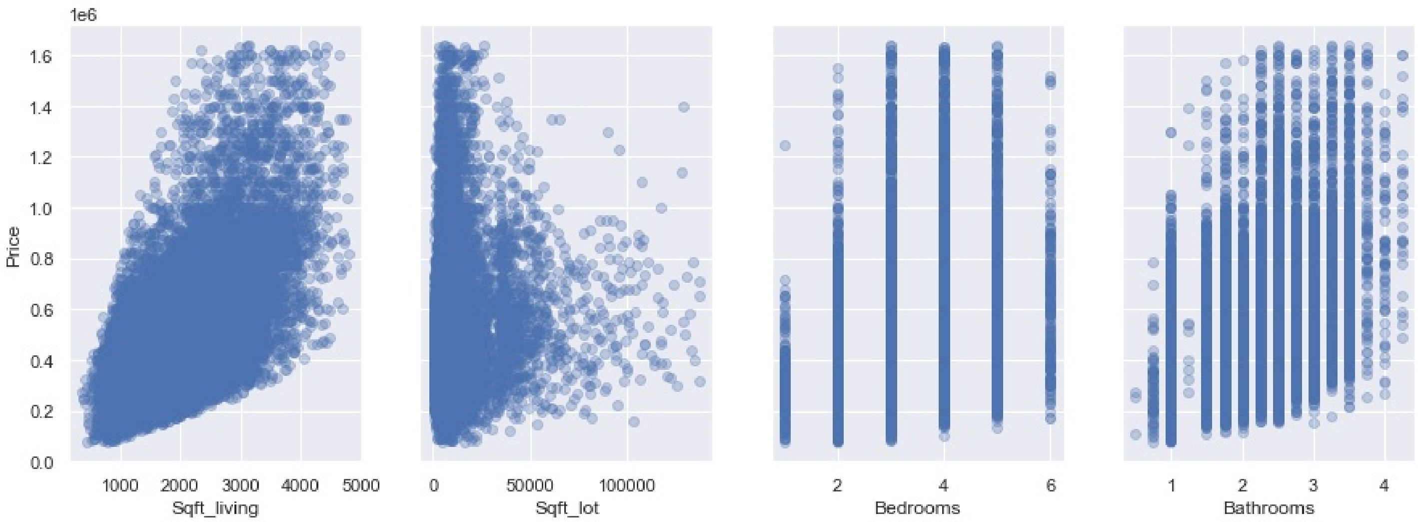
	Sqft_living	Sqft_lot	Bedrooms	Bathrooms	Price

	Sqft_living	Sqft_lot	Bedrooms	Bathrooms	Price
0	1180	5650	3	1.00	221900.0
1	2570	7242	3	2.25	538000.0
2	770	10000	2	1.00	180000.0
3	1960	5000	4	3.00	604000.0
4	1680	8080	3	2.00	510000.0
...	...	...	...	...	...
21592	1530	1131	3	2.50	360000.0
21593	2310	5813	4	2.50	400000.0
21594	1020	1350	2	0.75	402101.0
21595	1600	2388	3	2.50	400000.0
21596	1020	1076	2	0.75	325000.0

19685 rows × 5 columns

In [46]:

```
#create scatter plots
fig, axes = plt.subplots(ncols=4, figsize=(15,5), sharey=True)
axes[0].set_ylabel("Price")

for i, col in enumerate(data_subset.drop("Price", axis=1).columns):
    ax=axes[i]
    ax.scatter(data_subset[col], data_subset["Price"], alpha=0.3)
    ax.set_xlabel(col)
```



All four features seem to have a linear relationship with Price

Sqft_living seems to have the most variance vs. Price Bedrooms and Bathrooms have very weak variance Vs Price

In [47]:

```
data_subset2 = data[['Sqft_above', 'Sqft_basement', 'Sqft_living15', 'Sqft_lot15','Price
']].copy()
data_subset2
```

Out[47]:

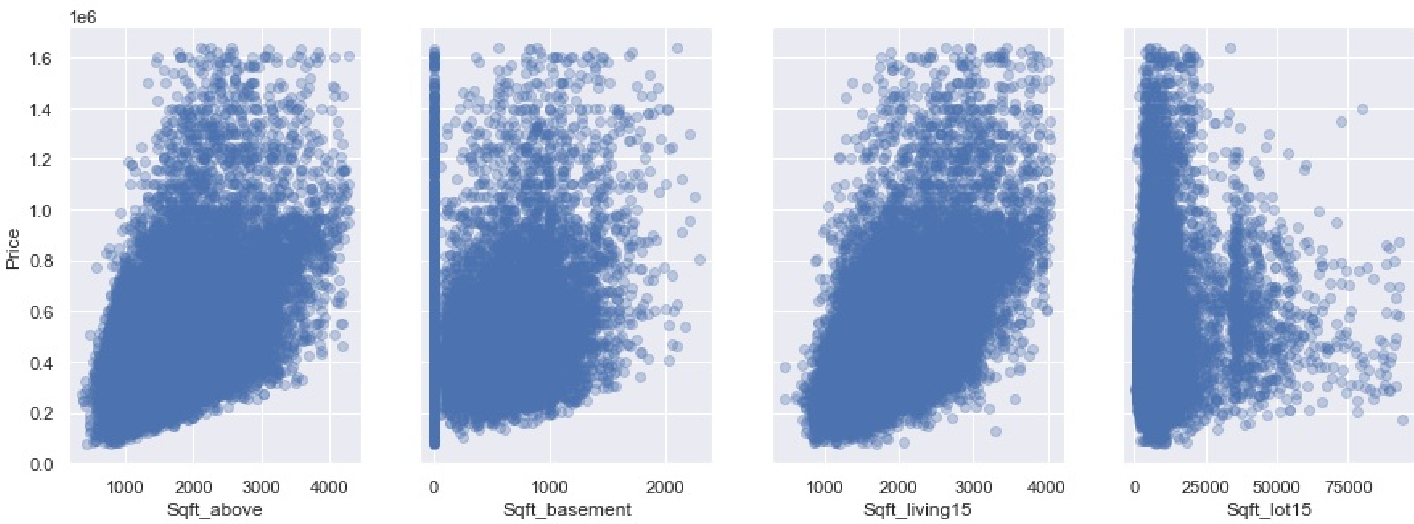
	Sqft_above	Sqft_basement	Sqft_living15	Sqft_lot15	Price
0	1180	0.0	1340	5650	221900.0
1	2170	400.0	1690	7639	538000.0
2	770	0.0	2720	8062	180000.0
3	1050	910.0	1360	5000	604000.0

	Sqft_above	Sqft_basement	Sqft_living15	Sqft_lot15	Price
4	1680	0.0	1800	7503	510000.0
...	...	...	...	...	...
21592	1530	0.0	1530	1509	360000.0
21593	2310	0.0	1830	7200	400000.0
21594	1020	0.0	1020	2007	402101.0
21595	1600	0.0	1410	1287	400000.0
21596	1020	0.0	1020	1357	325000.0

19685 rows × 5 columns

In [48]:

```
# Your code here - import relevant library, create scatter plots
fig, axes = plt.subplots(ncols=4, figsize=(15,5), sharey=True)
axes[0].set_ylabel("Price")

for i, col in enumerate(data_subset2.drop("Price", axis=1).columns):
    ax=axes[i]
    ax.scatter(data_subset2[col], data_subset2["Price"], alpha=0.3)
    ax.set_xlabel(col)
```



All of these four features seem to have a linear relationship with Price

Sqft_living15 column seems to have the most variance vs. Price followed closely by Sqft_lot15

In [49]:

```
data_subset3 = data[['Lat', 'Floors', 'Zipcode', 'Yr_built', 'Price']].copy()
data_subset3,
```

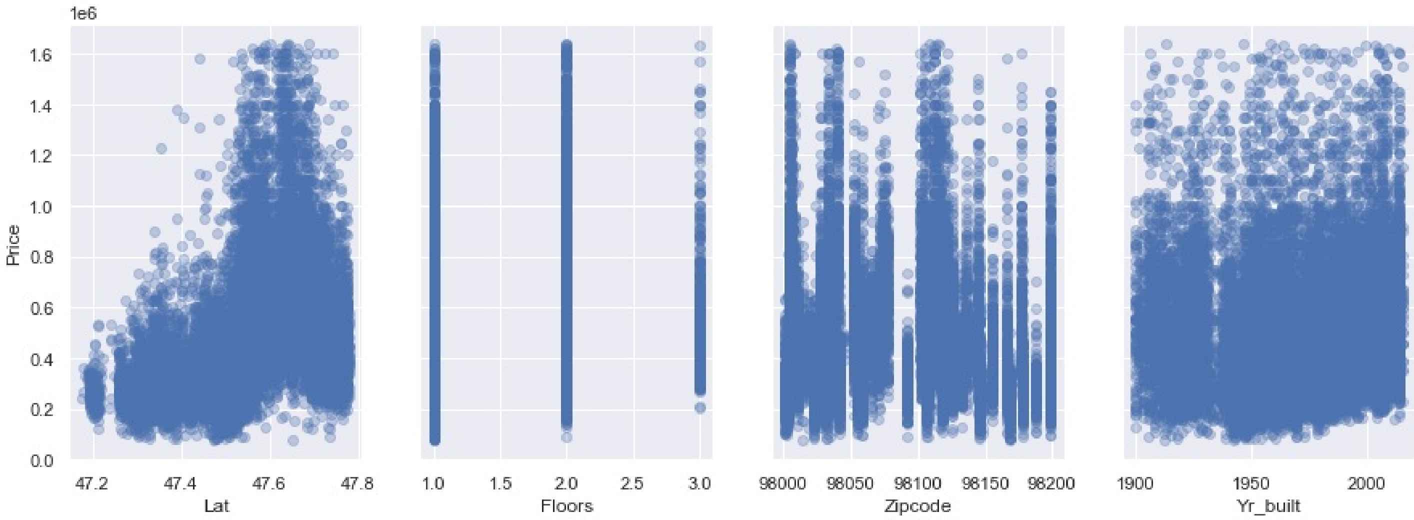
Out[49]:

```
(            Lat  Floors  Zipcode  Yr_built     Price
 0      47.5112       1    98178      1955  221900.0
 1      47.7210       2    98125      1951  538000.0
 2      47.7379       1    98028      1933  180000.0
 3      47.5208       1    98136      1965  604000.0
 4      47.6168       1    98074      1987  510000.0
 ...        ...     ...      ...       ...       ...
 21592  47.6993       3    98103      2009  360000.0
 21593  47.5107       2    98146      2014  400000.0
 21594  47.5944       2    98144      2009  402101.0
 21595  47.5345       2    98027      2004  400000.0
 21596  47.5941       2    98144      2008  325000.0

 [19685 rows x 5 columns],)
```

In [50]:

```
# Your code here - import relevant library, create scatter plots
```

```
fig, axes = plt.subplots(ncols=4, figsize=(15,5), sharey=True)
axes[0].set_ylabel("Price")

for i, col in enumerate(data_subset3.drop("Price", axis=1).columns):
    ax=axes[i]
    ax.scatter(data_subset3[col], data_subset3["Price"], alpha=0.3)
    ax.set_xlabel(col)
```



All of these four features seem to have a linear relationship with Price

Lat column seems to have the most variance vs. Price

CHECKING FOR MULTICOLLINEARITY

In [51]:

```
independent_vars = data[['Bathrooms', 'Bedrooms', 'Sqft_living','Grading', 'Sqft_above',
'Sqft_living15', 'Lat', 'Floors', 'Yr_renovated', 'Sqft_lot', 'Sqft_lot15','Long', 'Yr_b
uilt', 'Zipcode']]

# Calculate correlation matrix
correlation_matrix = independent_vars.corr()

# Display correlation matrix
print(correlation_matrix)
#2. Variance Inflation Factor (VIF)
#Calculate the Variance Inflation Factor (VIF) for each independent variable. VIF measure
s how much the variance of an estimated regression coefficient increases if your predicto
rs are correlated.

# Calculate VIF for each independent variable
vif_data = pd.DataFrame()
vif_data["Feature"] = independent_vars.columns
vif_data["VIF"] = [variance_inflation_factor(independent_vars.values, i) for i in range(
len(independent_vars.columns))]
print(vif_data)
```

```
               Bathrooms  Bedrooms  Sqft_living   Grading  Sqft_above  \
Bathrooms       1.000000  0.496519     0.712976  0.621718    0.630923
Bedrooms        0.496519  1.000000     0.605821  0.341800    0.483000
Sqft_living     0.712976  0.605821     1.000000  0.714065    0.843143
Grading         0.621718  0.341800     0.714065  1.000000    0.708186
Sqft_above      0.630923  0.483000     0.843143  0.708186    1.000000
Sqft_living15   0.529788  0.395662     0.740939  0.673105    0.710484
Lat             0.006097 -0.032532     0.038606  0.109614   -0.021854
Floors          0.526388  0.137811     0.337780  0.470591    0.529241
Yr_renovated    0.034426  0.006353     0.037267  0.002597    0.005483
Sqft_lot        0.059252  0.102367     0.225068  0.135724    0.209207
Sqft_lot15      0.063438  0.102743     0.228282  0.151002    0.217345
Long            0.257063  0.170344     0.277599  0.221835    0.394315
Yr_built        0.538480  0.168718     0.334372  0.466052    0.450352
Zipcode        -0.205413 -0.163142    -0.197157 -0.177999   -0.267725
```

```
                 Sqft_living15       Lat    Floors  Yr_renovated  Sqft_lot  \
Bathrooms             0.529788  0.006097  0.526388      0.034426  0.059252
Bedrooms              0.395662 -0.032532  0.137811      0.006353  0.102367
Sqft_living           0.740939  0.038606  0.337780      0.037267  0.225068
Grading               0.673105  0.109614  0.470591      0.002597  0.135724
Sqft_above            0.710484 -0.021854  0.529241      0.005483  0.209207
Sqft_living15         1.000000  0.038245  0.276098     -0.019999  0.268220
Lat                   0.038245  1.000000  0.019597      0.027616 -0.062716
Floors                0.276098  0.019597  1.000000     -0.015763 -0.097265
Yr_renovated         -0.019999  0.027616 -0.015763      1.000000  0.013946
Sqft_lot              0.268220 -0.062716 -0.097265      0.013946  1.000000
Sqft_lot15            0.295851 -0.062436 -0.094464      0.012742  0.813587
Long                  0.381006 -0.130564  0.178326     -0.070311  0.264670
Yr_built              0.333088 -0.152746  0.590668     -0.194751  0.023701
Zipcode              -0.282029  0.271737 -0.096346      0.068337 -0.179550

                 Sqft_lot15      Long  Yr_built   Zipcode
Bathrooms          0.063438  0.257063  0.538480 -0.205413
Bedrooms           0.102743  0.170344  0.168718 -0.163142
Sqft_living        0.228282  0.277599  0.334372 -0.197157
Grading            0.151002  0.221835  0.466052 -0.177999
Sqft_above         0.217345  0.394315  0.450352 -0.267725
Sqft_living15      0.295851  0.381006  0.333088 -0.282029
Lat               -0.062436 -0.130564 -0.152746  0.271737
Floors            -0.094464  0.178326  0.590668 -0.096346
Yr_renovated       0.012742 -0.070311 -0.194751  0.068337
Sqft_lot           0.813587  0.264670  0.023701 -0.179550
Sqft_lot15         1.000000  0.294880  0.043846 -0.192665
Long               0.294880  1.000000  0.426155 -0.582936
Yr_built           0.043846  0.426155  1.000000 -0.341316
Zipcode           -0.192665 -0.582936 -0.341316  1.000000
          Feature           VIF
0       Bathrooms  2.924803e+01
1        Bedrooms  2.725006e+01
2     Sqft_living  5.241568e+01
3         Grading  1.507519e+02
4      Sqft_above  3.644921e+01
5   Sqft_living15  3.011027e+01
6             Lat  1.363032e+05
7          Floors  1.760715e+01
8    Yr_renovated  1.119356e+00
9        Sqft_lot  5.194917e+00
10     Sqft_lot15  6.240169e+00
11           Long  1.720040e+06
12       Yr_built  9.401162e+03
13        Zipcode  2.020593e+06
```

High correlations(eg above 0.7 ot 0.8), between pairs of variables suggest potential multicollinearity

If the VIF is greater than 5 or 10,it indicates multicolinearity. Higher VIF values signify stronger correlation with other predictors.

FEATURE SELECTION

Its a method used to calculate pairwise correlations among features and remove one of each pair of highly correlated features.

In [52]:

```
# Calculate correlation matrix
correlation_matrix = data.corr().abs()

# Create a mask for selecting upper triangle of correlation matrix
upper_triangle_mask = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape),
k=1).astype(bool))

# Find features with correlation above threshold
highly_correlated_features = [column for column in upper_triangle_mask.columns if any(up
```

```python
per_triangle_mask[column] > 0.8)]
# Drop highly correlated features
selected_features = data.drop(highly_correlated_features, axis=1)
selected_features
```

Out[52]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	Grade	Sqft_base
0	2014-10-13	221900.0	3	1.00	1180	5650	1	NONE	NONE	Average	7 Average	
1	2014-09-12	538000.0	3	2.25	2570	7242	2	NO	NONE	Average	7 Average	
2	2015-02-25	180000.0	2	1.00	770	10000	1	NO	NONE	Average	6 Low Average	
3	2014-09-12	604000.0	4	3.00	1960	5000	1	NO	NONE	Very Good	7 Average	
4	2015-02-18	510000.0	3	2.00	1680	8080	1	NO	NONE	Average	8 Good	
...	...	...	...	...	...	...	...	...	...	...	...	
21592	2014-05-21	360000.0	3	2.50	1530	1131	3	NO	NONE	Average	8 Good	
21593	2015-02-23	400000.0	4	2.50	2310	5813	2	NO	NONE	Average	8 Good	
21594	2014-06-23	402101.0	2	0.75	1020	1350	2	NO	NONE	Average	7 Average	
21595	2015-01-16	400000.0	3	2.50	1600	2388	2	NONE	NONE	Average	8 Good	
21596	2014-10-15	325000.0	2	0.75	1020	1076	2	NO	NONE	Average	7 Average	

19685 rows × 20 columns

SPLITTING THE DATA INTO TRAINING AND TESTING SETS

In [53]:

```python
from sklearn.model_selection import train_test_split
data2 = selected_features
# Set a random seed for reproducibility
np.random.seed(0)

# Split the DataFrame into training and testing datasets
data2_train, data2_test = train_test_split(data2, train_size=0.7, test_size=0.3, random_s
tate=500)

# Display the shapes of the resulting training and testing datasets
print("Shape of data_train:", data2_train.shape)
print("Shape of data_test:", data2_test.shape)
```

```
Shape of data_train: (13779, 20)
Shape of data_test: (5906, 20)
```

In [54]:

```python
data2_train.head()
```

Out[54]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	Grade	S
20394	2014-10-14	271115.0	2	1.50	830	1325	2	NO	NONE	Average	7 Average	
10871	2014-	1610000.0	4	2.75	4070	25807	2	NO	NONE	Average	11	

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	Grade	S
12871	11-26	1610000.0	4	2.75	4270	25807	2	NO	NONE	Average	Excellent	
13963	2015-03-04	800000.0	4	2.25	2120	9921	2	NO	NONE	Average	8 Good	
15214	2015-03-20	715000.0	3	1.75	1650	7276	1	NO	EXCELLENT	Good	7 Average	
20944	2015-04-15	580000.0	3	1.50	1320	1250	3	NO	NONE	Average	8 Good	

BASELINE MODEL

In [55]:

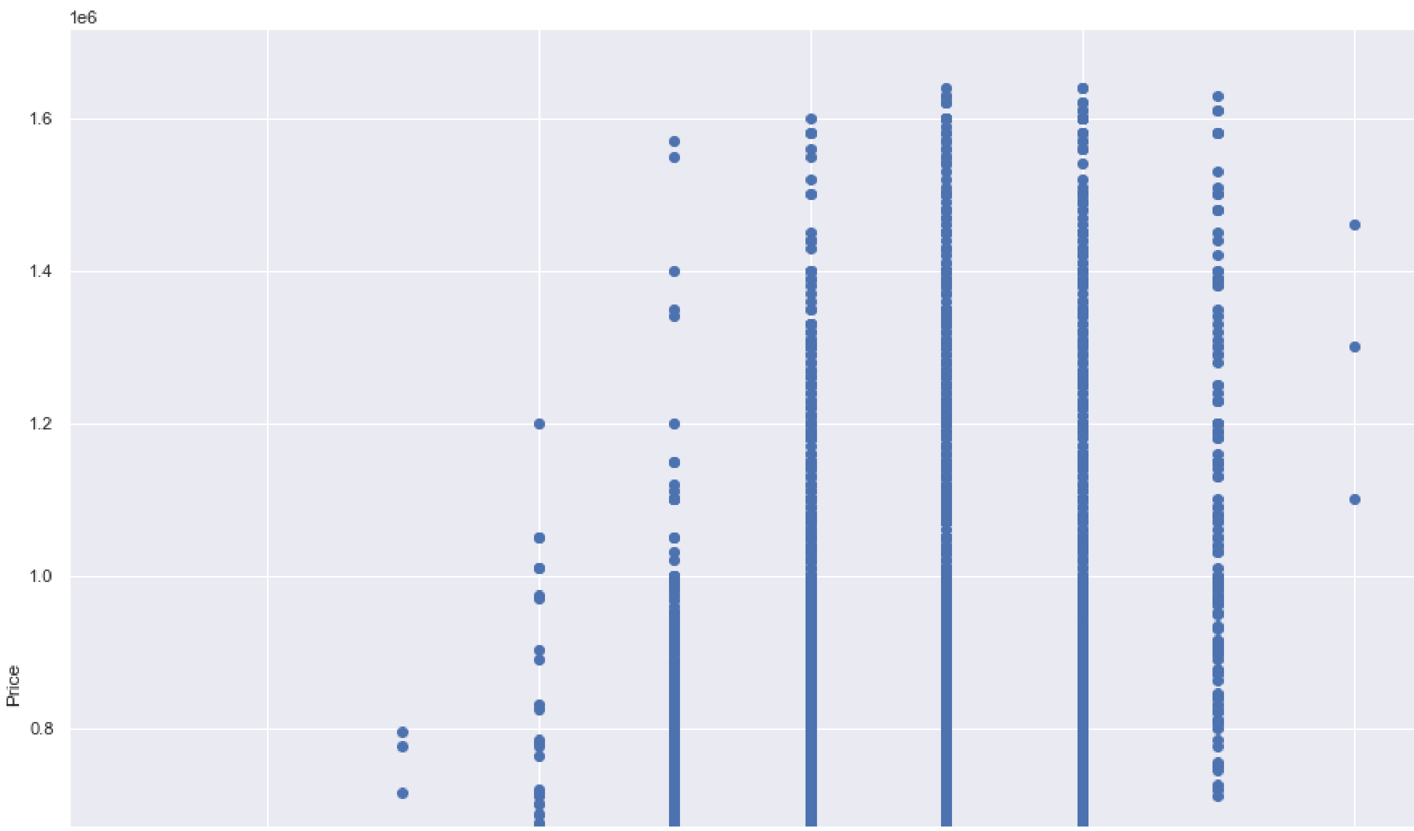
```
data2.head()
```

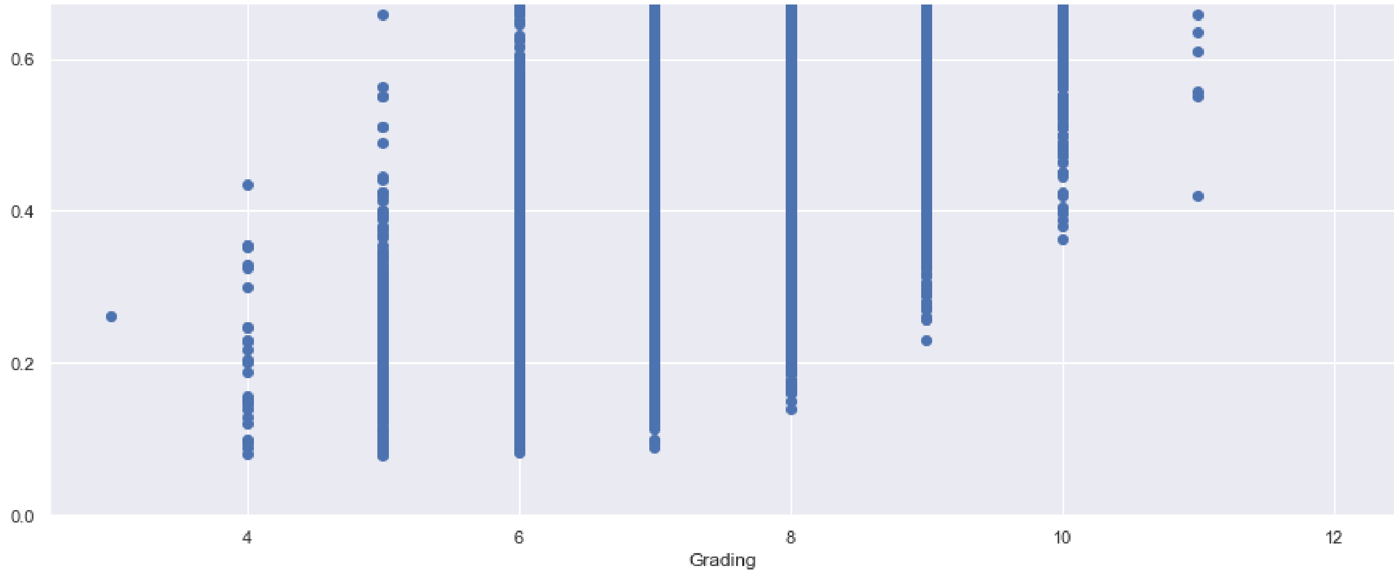
Out[55]:

	Date	Price	Bedrooms	Bathrooms	Sqft_living	Sqft_lot	Floors	Waterfront	View	Condition	Grade	Sqft_basement
0	2014-10-13	221900.0	3	1.00	1180	5650	1	NONE	NONE	Average	7 Average	0.0
1	2014-09-12	538000.0	3	2.25	2570	7242	2	NO	NONE	Average	7 Average	400.0
2	2015-02-25	180000.0	2	1.00	770	10000	1	NO	NONE	Average	6 Low Average	0.0
3	2014-09-12	604000.0	4	3.00	1960	5000	1	NO	NONE	Very Good	7 Average	910.0
4	2015-02-18	510000.0	3	2.00	1680	8080	1	NO	NONE	Average	8 Good	0.0

In [56]:

```
Y = data['Price']
X = data['Grading']
plt.scatter(X,Y)
plt.xlabel("Grading")
plt.ylabel("Price")
plt.show;
```





In [57]:

```
X_baseline = data2_train['Grading']  # Predictor variable
y = np.log(data2_train["Price"])  # Log-transformed target variable

# Add a constant term to the predictor variable
X_baseline = sm.add_constant(X_baseline)

# Fit the Ordinary Least Squares (OLS) model
model = sm.OLS(y, X_baseline).fit()

# Display the summary of the fitted model
result = model.summary()
print(result)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Price   R-squared:                       0.418
Model:                            OLS   Adj. R-squared:                  0.418
Method:                 Least Squares   F-statistic:                     9907.
Date:                Tue, 09 Apr 2024   Prob (F-statistic):               0.00
Time:                        22:31:33   Log-Likelihood:                -5610.7
No. Observations:               13779   AIC:                         1.123e+04
Df Residuals:                   13777   BIC:                         1.124e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         10.7957      0.022    481.624      0.000      10.752      10.840
Grading        0.2923      0.003     99.533      0.000       0.287       0.298
==============================================================================
Omnibus:                       10.074   Durbin-Watson:                   2.032
Prob(Omnibus):                  0.006   Jarque-Bera (JB):                9.597
Skew:                           0.041   Prob(JB):                      0.00824
Kurtosis:                       2.900   Cond. No.                         56.2
==============================================================================

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

Our linear regression model is Price = 0.2923 * Grading + 10.7957.

With a significance level of 0.05, our linear regression model is significant with a p value F statistic of 0.0.

The model explains about 42% of Price as indicated by the R squared.

Both our intercept and our coefficient for Grading are statistically significant.

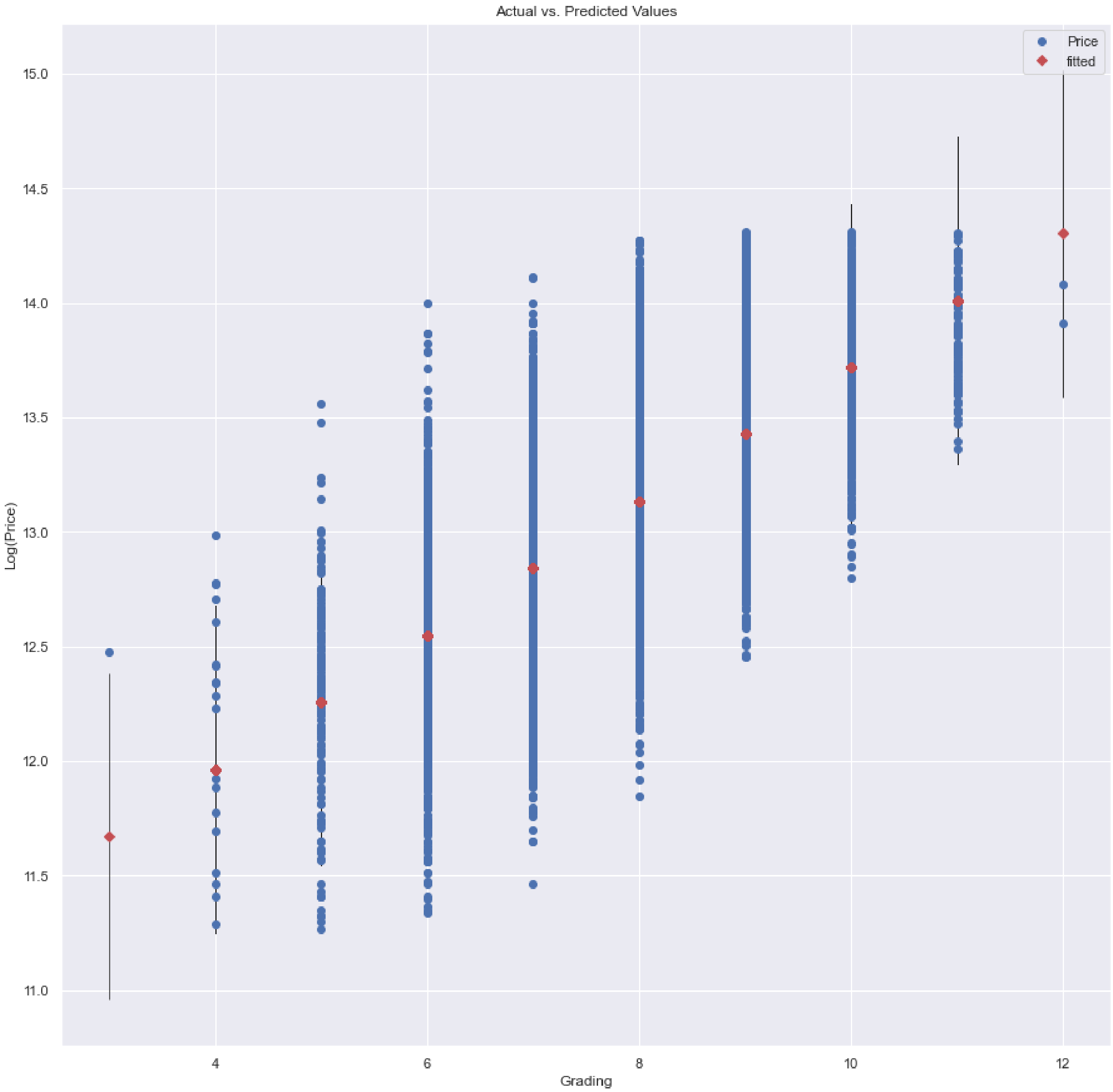
Our intercept is about 10.7957, meaning that a home with 0 Grading area would cost about 10.7957

Our coefficient for Grading is about 10.8, which means that for each additional Grading, we expect the price to increase about $0.2923.

BASELINE MODEL VISUALIZATION

In [58]:

```
# Plotting actual vs. predicted values
sm.graphics.plot_fit(model, "Grading")
plt.title("Actual vs. Predicted Values")
plt.xlabel("Grading")
plt.ylabel("Log(Price)")  # Assuming you are using log-transformed Price
plt.show()
```

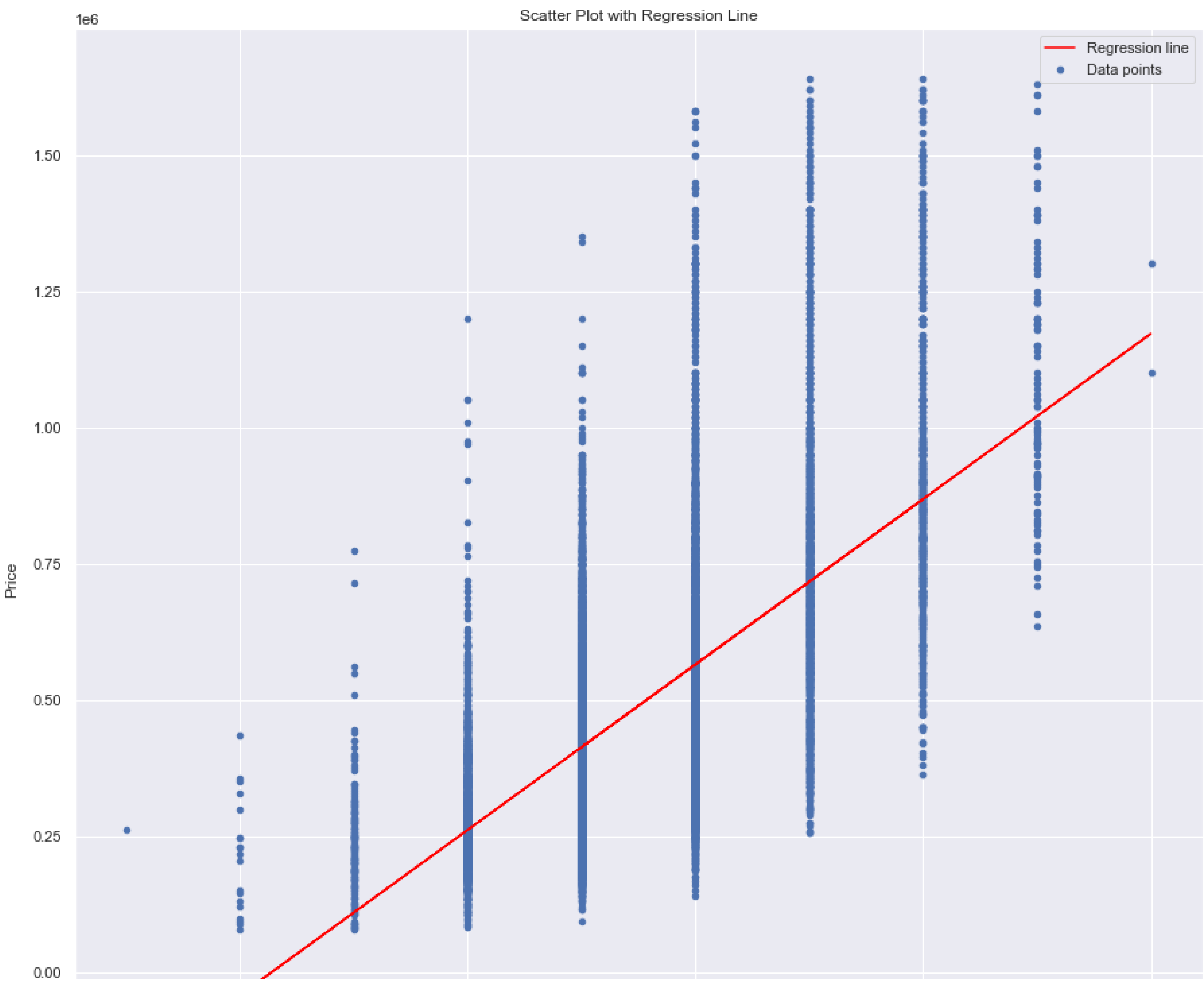


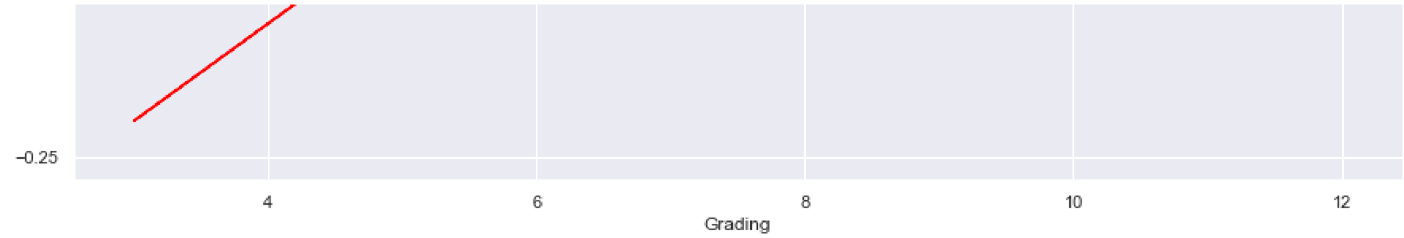
This visualization provides insights into how well the model fits the training data and how the predicted values compare to the actual target values. Our model indicates a good fit between the predicted values and the actual values because the fitted line closely follows the diagonal (a 45-degree line from the bottom-left to the top-right of the plot).

In [59]:

```python
# Define predictor (X) and target (y) variables
X = data2_train['Grading']
y = data2_train['Price']

# Add constant to predictor variable (for intercept)
X = sm.add_constant(X)

# Fit the Ordinary Least Squares (OLS) model
model = sm.OLS(y, X).fit()

# Retrieve the model parameters (coefficients)
intercept, slope = model.params['const'], model.params['Grading']

# Create scatter plot of data points and regression line
fig, ax = plt.subplots()
data2_train.plot.scatter(x='Grading', y='Price', label="Data points", ax=ax)

# Plot the regression line using model parameters
ax.plot(X['Grading'], intercept + slope * X['Grading'], color='red', label="Regression l
ine")

# Add labels, title, and legend
ax.set_xlabel("Grading")
ax.set_ylabel("Price")
ax.set_title("Scatter Plot with Regression Line")
ax.legend()

# Show the plot
plt.show()
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as
value-mapping will have precedence in case its length matches with *x* & *y*.  Please use
the *color* keyword-argument or provide a 2-D array with a single row if you intend to sp
ecify the same RGB or RGBA value for all points.
```

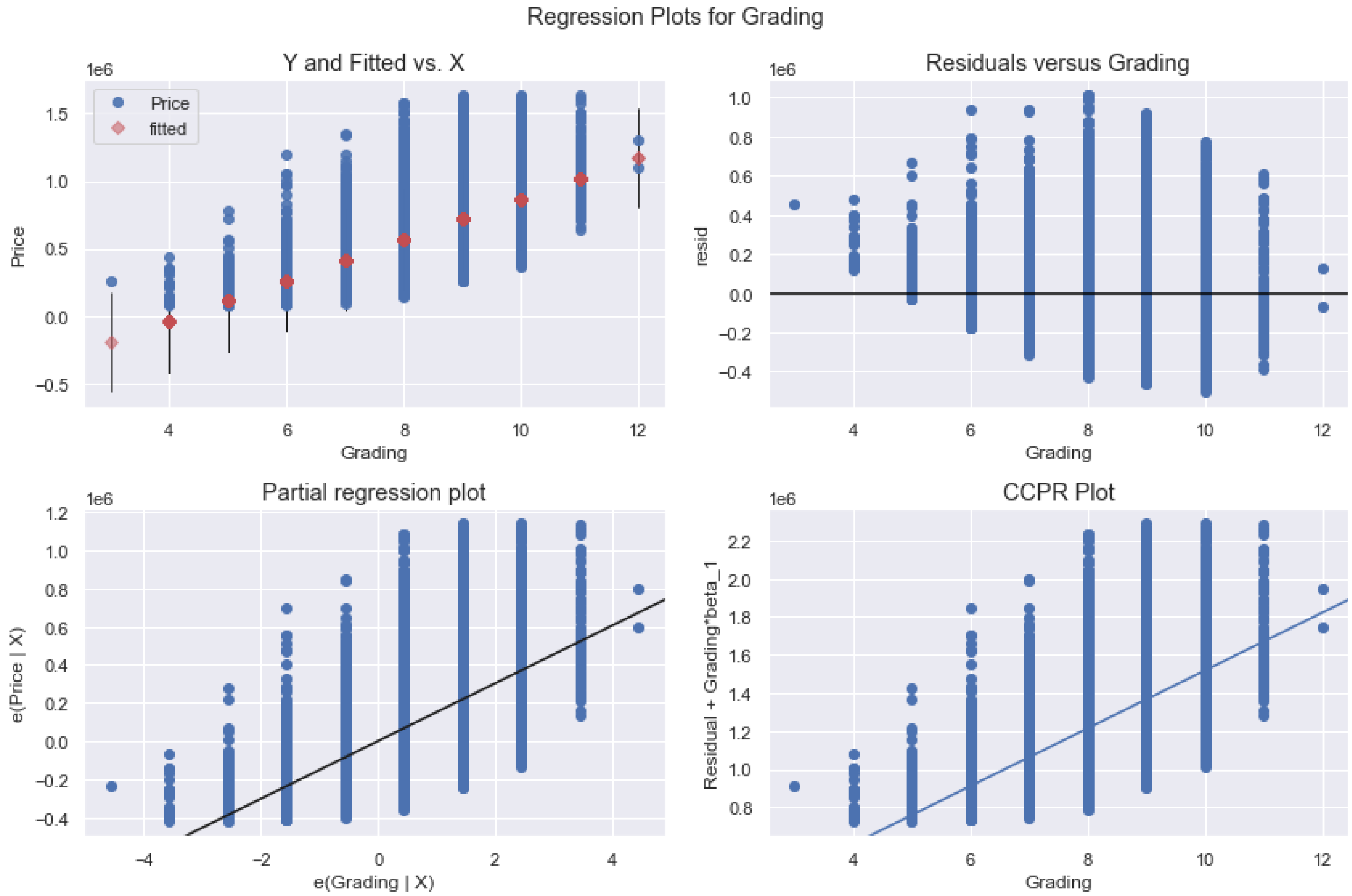


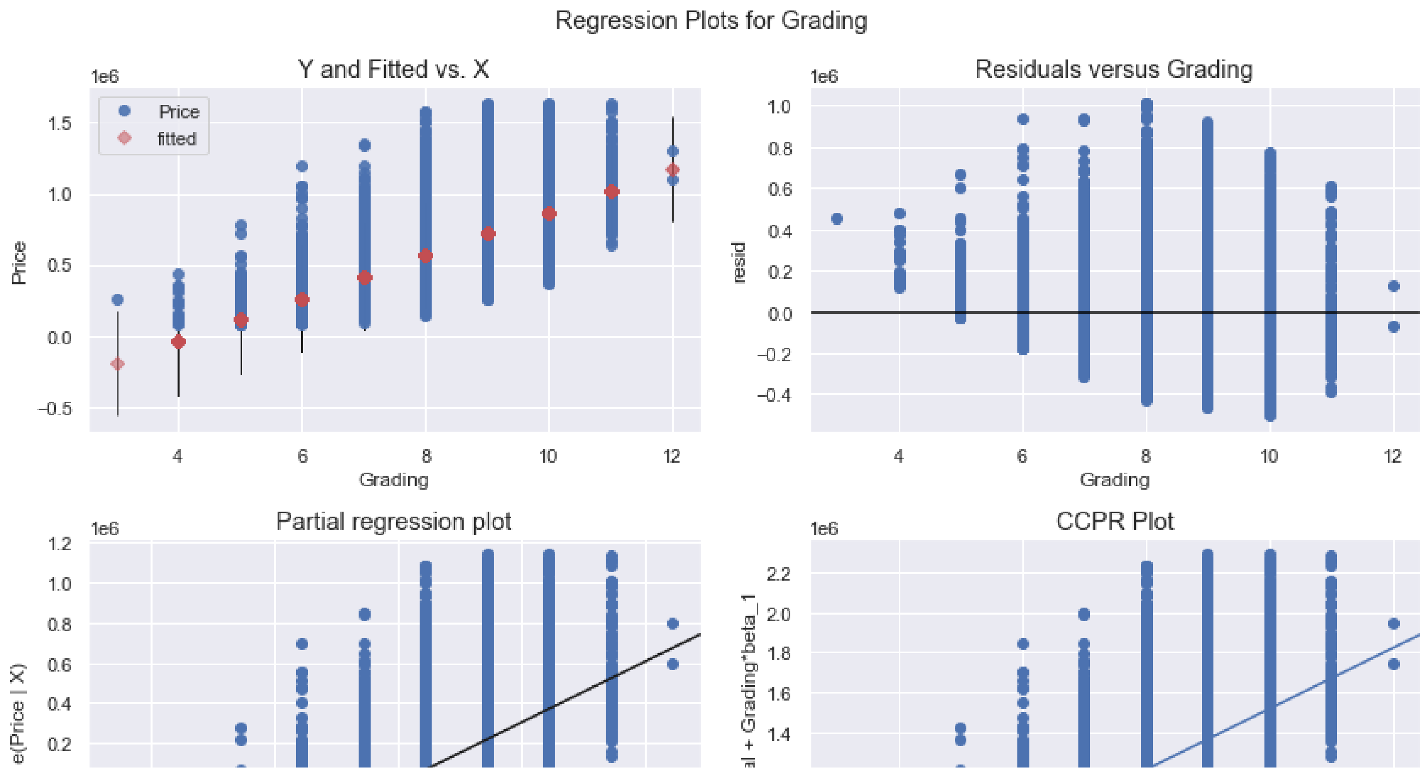


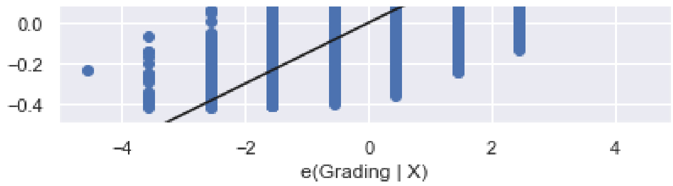
In [60]:

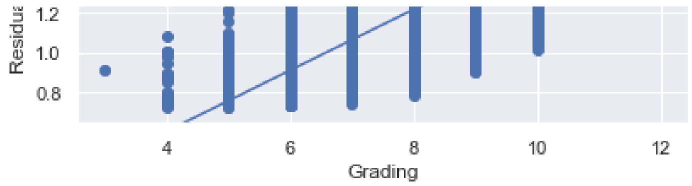
```
#checking residual in model0
sm.graphics.plot_regress_exog(model, "Grading", fig=plt.figure(figsize=(12,8)))
```

Out[60]:









This function displays a set of diagnostic plots that help in assessing the model's assumptions and the relationship between the predictor variable and the target variable

In [61]:

```
import numpy as np
import statsmodels.api as sm

# Assuming you have predictor variable(s) in X_baseline and log-transformed target variab
le in y
# Add a constant term to the predictor variable
X_baseline = sm.add_constant(X_baseline)

# Fit the Ordinary Least Squares (OLS) model
model = sm.OLS(y, X_baseline).fit()

# Calculate the residuals
residuals = y - model.predict(X_baseline)

# Calculate the Root Mean Squared Error (RMSE)
mse = np.mean(residuals**2)
rmse = np.sqrt(mse)

print("Root Mean Squared Error (RMSE):", rmse)
```

```
Root Mean Squared Error (RMSE): 190283.19633348894
```

MULTI LINEAR REGRESSION

Adding more predictors to our baseline model can enhance its predictive power by capturing additional complexity and potential relationships within our data. This can lead to improved model accuracy and potentially reducing bias by considering more factors in the prediction process.

In [62]:

```
data2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19685 entries, 0 to 21596
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Date           19685 non-null  datetime64[ns]
 1   Price          19685 non-null  float64
 2   Bedrooms       19685 non-null  int64
 3   Bathrooms      19685 non-null  float64
 4   Sqft_living    19685 non-null  int64
 5   Sqft_lot       19685 non-null  int64
 6   Floors         19685 non-null  int32
 7   Waterfront     19685 non-null  object
 8   View           19685 non-null  object
 9   Condition      19685 non-null  object
 10  Grade          19685 non-null  object
 11  Sqft_basement  19685 non-null  float64
 12  Yr_built       19685 non-null  int64
 13  Yr_renovated   19685 non-null  int32
 14  Zipcode        19685 non-null  int64
 15  Lat            19685 non-null  float64
 16  Long           19685 non-null  float64
 17  Sqft_living15  19685 non-null  int64
 18  Seasons        19685 non-null  object
 19  Grading        19685 non-null  int32
dtypes: datetime64[ns](1), float64(5), int32(3), int64(6), object(5)
```

```
dtypes: datetime64[ns](1), float64(5), int32(3), int64(6), object(5)
memory usage: 2.9+ MB
```

In [63]:

```python
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Assuming you have multiple predictor variables in X_data
X_data = data2_train[['Grading', 'Yr_renovated', 'Sqft_living15','Bedrooms', 'Bathrooms'
, 'Sqft_living', 'Sqft_lot', 'Floors', 'Sqft_basement', 'Yr_renovated','Zipcode', 'Lat',
'Long']]  # Add other predictor variables as needed
y = np.log(data2_train["Price"])  # Log-transformed target variable

# Add a constant term to the predictor variables
X_data = sm.add_constant(X_data)

# Fit the Ordinary Least Squares (OLS) model
model = sm.OLS(y, X_data).fit()

# Predict the y values
y_pred = model.predict(X_data)
print(model.summary())
```

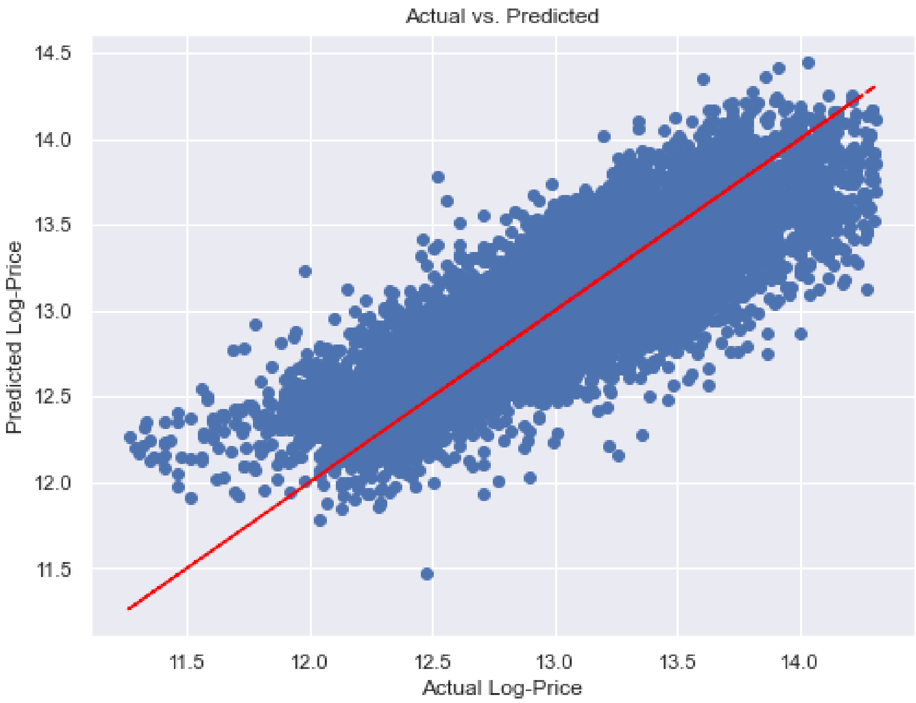
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Price   R-squared:                       0.687
Model:                            OLS   Adj. R-squared:                  0.687
Method:                 Least Squares   F-statistic:                     2516.
Date:                Tue, 09 Apr 2024   Prob (F-statistic):               0.00
Time:                        22:31:45   Log-Likelihood:                -1344.6
No. Observations:               13779   AIC:                             2715.
Df Residuals:                   13766   BIC:                             2813.
Df Model:                          12
Covariance Type:            nonrobust
=================================================================================
                    coef    std err          t      P>|t|      [0.025      0.975]
---------------------------------------------------------------------------------
const           -66.2959      4.484    -14.784      0.000     -75.086     -57.506
Grading           0.1349      0.004     37.743      0.000       0.128       0.142
Yr_renovated   4.338e-05   3.26e-06     13.316      0.000     3.7e-05    4.98e-05
Sqft_living15     0.0001   6.19e-06     19.835      0.000       0.000       0.000
Bedrooms         -0.0164      0.003     -4.767      0.000      -0.023      -0.010
Bathrooms         0.0156      0.005      2.856      0.004       0.005       0.026
Sqft_living       0.0002   6.86e-06     26.297      0.000       0.000       0.000
Sqft_lot      -2.195e-07   2.07e-07     -1.061      0.289   -6.25e-07    1.86e-07
Floors           -0.0125      0.006     -2.140      0.032      -0.024      -0.001
Sqft_basement  4.501e-05   7.51e-06      5.991      0.000    3.03e-05    5.97e-05
Yr_renovated   4.342e-05   3.26e-06     13.328      0.000     3.7e-05    4.98e-05
Zipcode          -0.0004   5.43e-05     -7.164      0.000      -0.000      -0.000
Lat               1.4871      0.017     85.171      0.000       1.453       1.521
Long             -0.3690      0.024    -15.674      0.000      -0.415      -0.323
==============================================================================
Omnibus:                      120.901   Durbin-Watson:                   2.019
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              186.115
Skew:                           0.067   Prob(JB):                     3.85e-41
Kurtosis:                       3.553   Cond. No.                     1.77e+16
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specifie
d.
[2] The smallest eigenvalue is 4.27e-19. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.
```

The model explains 69% of the variance in price as indicated by R squared

Therefore the multiple linear regression is better than our baseline regression

In [64]:

```
# Plot the actual vs. predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y, y_pred)
plt.plot(y, y, color='red', linestyle='--')  # Plotting the perfect fit line
plt.title('Actual vs. Predicted')
plt.xlabel('Actual Log-Price')
plt.ylabel('Predicted Log-Price')
plt.show()
```



In [65]:

```
from sklearn.metrics import mean_squared_error

# Compute Mean Squared Error (MSE)
mse = mean_squared_error(y, y_pred)

# Compute Root Mean Squared Error (RMSE)
rmse = np.sqrt(mse)

print("Root Mean Squared Error (RMSE):", rmse)
```

```
Root Mean Squared Error (RMSE): 0.266772625666969
```

POLYNOMIAL REGRESSION

The independent variable(s) and the dependent variable is modeled as an nth-degree polynomial. It extends linear regression by allowing the relationship between variables to be modeled as a curve rather than a straight line. In thic case we use it to determine whether its a better fit compared to the multiple linear regression

In [66]:

```
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler, PolynomialFeatures, MinM
axScaler

poly= PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(X_data)
X_train, X_test, y_train, y_test = train_test_split(poly_features, y, test_size=0.3, ran
dom_state=10)
# Let's fit our mode
poly_reg_model = LinearRegression()
```

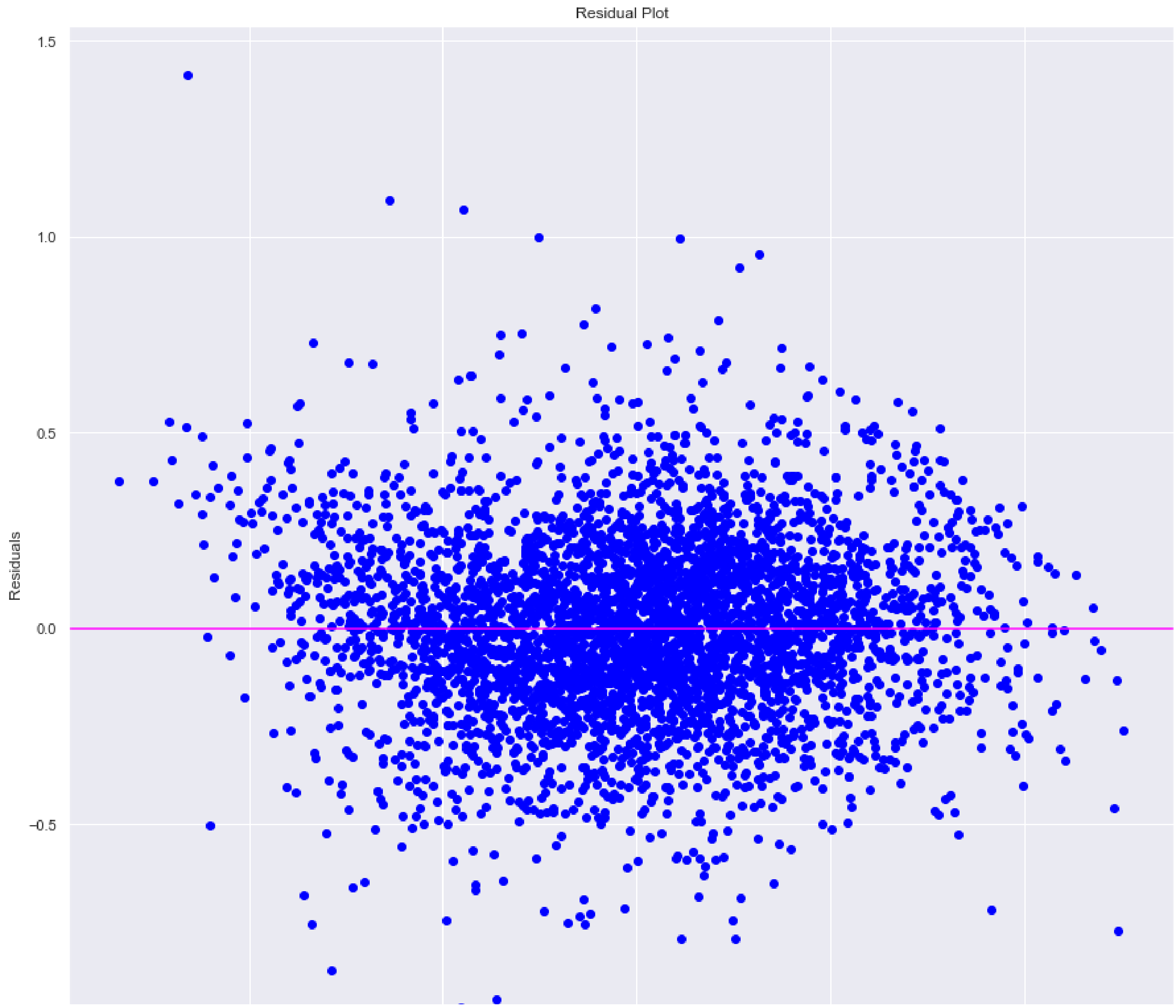
```
poly_reg_model.fit(X_train, y_train)
# Let's train our model
poly_reg_y_predicted1 = poly_reg_model.predict(X_test)
from sklearn.metrics import mean_squared_error
poly_reg_rmse = np.sqrt(mean_squared_error(y_test, poly_reg_y_predicted1))
print("Polynomial RMSE: ",poly_reg_rmse)
r_squared = r2_score(y_test, poly_reg_y_predicted1)
print("R_squared: ", r_squared)
```

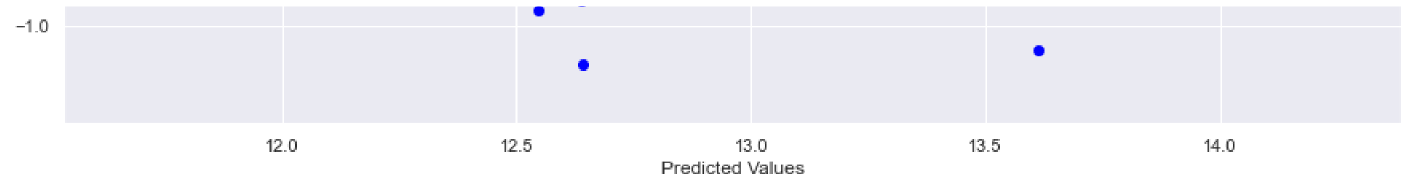
```
Polynomial RMSE:  0.23061511620028652
R_squared:  0.7655407239791847
```

Given that polynomial regression accounts for roughly 76% of the volatility in house prices—a percentage somewhat greater than that explained by multiple linear regression—it is clear that polynomial regression is a superior model in this particular situation. Polynomial regression also has a less Root Squared Mean Error than Multiple Linear Regression Model.This suggests that the polynomial regression yielded superior results. The RMSE of the polynomial regression also suggests the same since it is also lower than the multiple linear regression model.

In [67]:

```
residuals = y_test - poly_reg_y_predicted1

# Plotting the residuals
plt.scatter(poly_reg_y_predicted1, residuals, color='blue')
plt.title('Residual Plot')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.axhline(y=0, color='magenta', linestyle='-')   # Adding a horizontal line at y=0
plt.show()
```





ANOVA ANALYSIS

Ho: There is no statistically significant interaction effect between sqft_living and grading on the price of a house

H1: There is a statistically significant interaction effect between sqft_living and grading on the price of a house.

In [68]:

```
data2.columns
```

Out[68]:

```
Index(['Date', 'Price', 'Bedrooms', 'Bathrooms', 'Sqft_living', 'Sqft_lot',
       'Floors', 'Waterfront', 'View', 'Condition', 'Grade', 'Sqft_basement',
       'Yr_built', 'Yr_renovated', 'Zipcode', 'Lat', 'Long', 'Sqft_living15',
       'Seasons', 'Grading'],
      dtype='object')
```

ANOVA Test: Price vs. Interaction of Sqft_living and Grading To what extent does the combined influence of grading and sqft_living affect the price of a house?

In [69]:

```
#Creating interaction term
# Statistical modeling
import statsmodels.api as sm
from statsmodels.formula.api import ols

data2['Sqft_Grading'] = data2['Grading'] * data2['Sqft_living']
formula = 'Price ~ Sqft_living + Grading'
anova_model = ols(formula, data=data2).fit()
anova_table = sm.stats.anova_lm(anova_model, typ=2)
print(anova_table)
```

```
                   sum_sq        df            F  PR(>F)
Sqft_living  7.595825e+13       1.0  2327.57958     0.0
Grading      8.849776e+13       1.0  2711.82653     0.0
Residual     6.423025e+14   19682.0         NaN     NaN
```

At 0.05 significance level, we reject the Null hypothesis and conclude that there is a statistically significant interaction effect between grading and Sqft_living of a house on Price.

Summary

We embarked on a iterative statistical modelling proccess where we started with a simple linear regression, multilinear regression modelling and finally Polynomial regression where we noted that for each regression, the model became better with each iteration.

1. **The baseline model had an R Squared of 0.42**
2. **The multilinear regression had an R squared of 0.69 and an RMSE of 0.27**
3. **The polynomial regression model had an Rsquared of 0.77 and an RMSE of 0.23**

For the final polynomial regression RMSE value , our model is off by 0.23 dollars in a given prediction

We also conducted an Anova test to test the significance of how various sets of variables(including categorical variables) affected price. we concluded at at a significance level of 0.05,we reject the Null hypothesis and conclude that there is a statistically significant interaction effect between grading and Sqft_living of a house on

conclude that there is a statistically significant interaction effect between grading and sqft_living of a house on Price.

Recommendation

The square footage of living space has a significant influence on house pricing as well. This information can be used by the Real Estate Agency to support higher listing prices for homes with larger square footage.

There are several benefits to using a polynomial regression model instead of a typical linear model when predicting property prices. By taking into consideration the non-linear effects of important features like square footage, location, and on-site facilities, this method enables us to capture intricate correlations and fluctuations in home pricing. The polynomial regression's high R-squared value indicates that the chosen features account for a significant amount of the variability in home prices, which translates into more accurate and consistent price projections. Using this cutting-edge modeling technique gives us, as real estate market participants, more decision-making power and improves pricing tactics, investment analyses, and market competitiveness overall.

In [ ]:

```

```