# Data Cleaning

Our housing data is provided to us in a file called kc_house_data.csv. First, we bring that data into a pandas dataframe and start examining it.

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
df = pd.read_csv('data/kc_house_data.csv')
df
```

Out[2]:

| | id | date | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | wat |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 7129300520 | 10/13/2014 | 221900.0 | 3 | 1.00 | 1180 | 5650 | 1.0 | |
| **1** | 6414100192 | 12/9/2014 | 538000.0 | 3 | 2.25 | 2570 | 7242 | 2.0 | |
| **2** | 5631500400 | 2/25/2015 | 180000.0 | 2 | 1.00 | 770 | 10000 | 1.0 | |
| **3** | 2487200875 | 12/9/2014 | 604000.0 | 4 | 3.00 | 1960 | 5000 | 1.0 | |
| **4** | 1954400510 | 2/18/2015 | 510000.0 | 3 | 2.00 | 1680 | 8080 | 1.0 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **21592** | 263000018 | 5/21/2014 | 360000.0 | 3 | 2.50 | 1530 | 1131 | 3.0 | |
| **21593** | 6600060120 | 2/23/2015 | 400000.0 | 4 | 2.50 | 2310 | 5813 | 2.0 | |
| **21594** | 1523300141 | 6/23/2014 | 402101.0 | 2 | 0.75 | 1020 | 1350 | 2.0 | |
| **21595** | 291310100 | 1/16/2015 | 400000.0 | 3 | 2.50 | 1600 | 2388 | 2.0 | |
| **21596** | 1523300157 | 10/15/2014 | 325000.0 | 2 | 0.75 | 1020 | 1076 | 2.0 | |

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 21 columns):
id               21597 non-null int64
date             21597 non-null object
price            21597 non-null float64
bedrooms         21597 non-null int64
bathrooms        21597 non-null float64
sqft_living      21597 non-null int64
sqft_lot         21597 non-null int64
floors           21597 non-null float64
waterfront       19221 non-null float64
view             21534 non-null float64
condition        21597 non-null int64
grade            21597 non-null int64
sqft_above       21597 non-null int64
sqft_basement    21597 non-null object
yr_built         21597 non-null int64
yr_renovated     17755 non-null float64
```

In [4]:
```
df.isna().sum()
```

Out[4]:
```
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
```

In [5]:
```
for col in list(df.columns)[1:]:
    print(col)
    print(df[col].value_counts())
```

```
date
6/23/2014    142
6/26/2014    131
6/25/2014    131
7/8/2014     127
4/27/2015    126
            ...
5/15/2015      1
11/2/2014      1
1/10/2015      1
2/15/2015      1
3/8/2015       1
Name: date, Length: 372, dtype: int64
price
350000.0    172
450000.0    172
550000.0    159
500000.0    152
425000.0    150
```

We mostly have numerical objects, but with a few anomolies.

- "sqft_basement" should be a number and yet is a string. We will also need to replace the '?' with 0, to indicate no basement.
- "date" will need to be modified to work with a linear regression. We will break the year out from that column to be its own feature, "yr_sold".
- NaN values can be safely turned into 0, to indicate a lack of waterfront or renovations.
- 0 values in "yr_renovated" won't work with most values being in the 1900's-2000's, so we will replace it with a "yr_since_renovated" column (yr_sold-yr_renovated, or yr_sold-yr_built when yr_renovated=0).
- similarly, we will create a "yr_since_built" column (yr_sold-yr_built).

Our categorical variables are:

- waterfront
- view
- condition
- grade
- zipcode

In [6]:
```
df['yr_sold'] = df.date.map(lambda x: int(x.split('/')[-1]))
df
```

Out[6]:

| | id | date | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | wat |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 7129300520 | 10/13/2014 | 221900.0 | 3 | 1.00 | 1180 | 5650 | 1.0 | |
| **1** | 6414100192 | 12/9/2014 | 538000.0 | 3 | 2.25 | 2570 | 7242 | 2.0 | |
| **2** | 5631500400 | 2/25/2015 | 180000.0 | 2 | 1.00 | 770 | 10000 | 1.0 | |
| **3** | 2487200875 | 12/9/2014 | 604000.0 | 4 | 3.00 | 1960 | 5000 | 1.0 | |
| **4** | 1954400510 | 2/18/2015 | 510000.0 | 3 | 2.00 | 1680 | 8080 | 1.0 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **21592** | 263000018 | 5/21/2014 | 360000.0 | 3 | 2.50 | 1530 | 1131 | 3.0 | |
| **21593** | 6600060120 | 2/23/2015 | 400000.0 | 4 | 2.50 | 2310 | 5813 | 2.0 | |
| **21594** | 1523300141 | 6/23/2014 | 402101.0 | 2 | 0.75 | 1020 | 1350 | 2.0 | |
| **21595** | 291310100 | 1/16/2015 | 400000.0 | 3 | 2.50 | 1600 | 2388 | 2.0 | |
| **21596** | 1523300157 | 10/15/2014 | 325000.0 | 2 | 0.75 | 1020 | 1076 | 2.0 | |

In [7]:
```
df.replace({'sqft_basement': {'?': '0.0'}}, inplace=True)
df.sqft_basement = pd.to_numeric(df.sqft_basement)
```

In [8]:
```
df.fillna(0.0, inplace=True)
```

In [9]:
```
df.isna().sum()
```

Out[9]:
```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
```

In [10]:
```
df['yr_since_renovation'] = np.where(df['yr_renovated']==0.0, df['yr_sold']-df['y
df['yr_since_built'] = df['yr_sold'] - df['yr_built']
df['renovated'] = df.yr_renovated.map(lambda x: 1 if x>0 else 0)
```

In [11]:
```
df
```

Out[11]:

| | id | date | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | water |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 7129300520 | 10/13/2014 | 221900.0 | 3 | 1.00 | 1180 | 5650 | 1.0 | |
| **1** | 6414100192 | 12/9/2014 | 538000.0 | 3 | 2.25 | 2570 | 7242 | 2.0 | |
| **2** | 5631500400 | 2/25/2015 | 180000.0 | 2 | 1.00 | 770 | 10000 | 1.0 | |
| **3** | 2487200875 | 12/9/2014 | 604000.0 | 4 | 3.00 | 1960 | 5000 | 1.0 | |
| **4** | 1954400510 | 2/18/2015 | 510000.0 | 3 | 2.00 | 1680 | 8080 | 1.0 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **21592** | 263000018 | 5/21/2014 | 360000.0 | 3 | 2.50 | 1530 | 1131 | 3.0 | |
| **21593** | 6600060120 | 2/23/2015 | 400000.0 | 4 | 2.50 | 2310 | 5813 | 2.0 | |
| **21594** | 1523300141 | 6/23/2014 | 402101.0 | 2 | 0.75 | 1020 | 1350 | 2.0 | |
| **21595** | 291310100 | 1/16/2015 | 400000.0 | 3 | 2.50 | 1600 | 2388 | 2.0 | |
| **21596** | 1523300157 | 10/15/2014 | 325000.0 | 2 | 0.75 | 1020 | 1076 | 2.0 | |

21597 rows × 25 columns

We have a few features we want to remove:

- id
- date
- view
- sqft_above
- sqft_living15
- sqft_lot15

"sqft_living15", "sqft_lot15" are not about the house itself, and are likely to be highly corellated with our other features anyways. "sqft_above" is directly the difference between "sqft_living" and "sqft_basement", so it is also unnecessary. The description for "view" states "Has been viewed", and yet indicates numbers 1 through 3, with some NaN values. We aren't sure how to interperet this, so we have decided to exclude it. "date" has been incorperated into our "yr_sold" column, and so is now unnecessary. the "id" column has no bearing on our modeling, so we will remove it as well.

In [12]:
```
df.drop(['id', 'view', 'sqft_above', 'sqft_living15', 'sqft_lot15', 'date'], axis
```

In [13]:
```
df
```

Out[13]:

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

In [14]:
```
df.to_csv('data/cleaned_data.csv', index=False)
```

# Exploratory Data Analysis

In order to prepare for our modeling, we will check which features are corolated with "price", check the feature's distribution, and look for collinear features.

In [1]:
```
import pandas as pd
```

In [2]:
```
df = pd.read_csv('data/cleaned_data.csv')
df.head()
```

Out[2]:

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | condition | grade | sqft_ |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 221900.0 | 3 | 1.00 | 1180 | 5650 | 1.0 | 0.0 | 3 | 7 | |
| **1** | 538000.0 | 3 | 2.25 | 2570 | 7242 | 2.0 | 0.0 | 3 | 7 | |
| **2** | 180000.0 | 2 | 1.00 | 770 | 10000 | 1.0 | 0.0 | 3 | 6 | |
| **3** | 604000.0 | 4 | 3.00 | 1960 | 5000 | 1.0 | 0.0 | 5 | 7 | |
| **4** | 510000.0 | 3 | 2.00 | 1680 | 8080 | 1.0 | 0.0 | 3 | 8 | |

In [3]:
```
df.describe()
```

Out[3]:

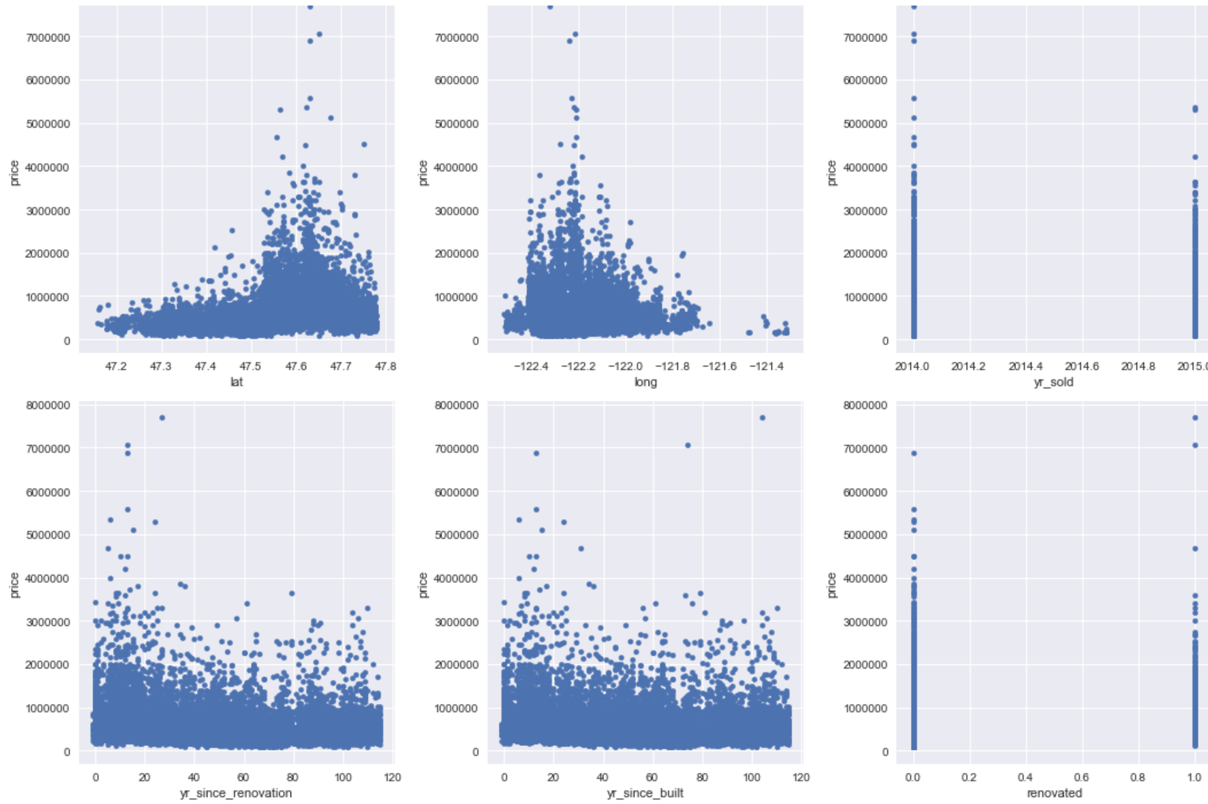
| | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | wa |
|---|---|---|---|---|---|---|---|
| **count** | 2.159700e+04 | 21597.000000 | 21597.000000 | 21597.000000 | 2.159700e+04 | 21597.000000 | 21597 |
| **mean** | 5.402966e+05 | 3.373200 | 2.115826 | 2080.321850 | 1.509941e+04 | 1.494096 | 0 |
| **std** | 3.673681e+05 | 0.926299 | 0.768984 | 918.106125 | 4.141264e+04 | 0.539683 | 0 |
| **min** | 7.800000e+04 | 1.000000 | 0.500000 | 370.000000 | 5.200000e+02 | 1.000000 | 0 |
| **25%** | 3.220000e+05 | 3.000000 | 1.750000 | 1430.000000 | 5.040000e+03 | 1.000000 | 0 |
| **50%** | 4.500000e+05 | 3.000000 | 2.250000 | 1910.000000 | 7.618000e+03 | 1.500000 | 0 |
| **75%** | 6.450000e+05 | 4.000000 | 2.500000 | 2550.000000 | 1.068500e+04 | 2.000000 | 0 |
| **max** | 7.700000e+06 | 33.000000 | 8.000000 | 13540.000000 | 1.651359e+06 | 3.500000 | 1 |

# Which Features are Corolated with the Target

We plot each variable against "price", our target, to see what relationship they have.

In [4]:
```
import matplotlib.pyplot as plt
plt.style.use('seaborn')
%matplotlib inline
```

In [5]:
```
plt.figure(figsize=(15,30))
for i, col in enumerate(df.drop('price', axis=1).columns):
    ax = plt.subplot(6, 3, i+1)
    df.plot.scatter(x=col, y='price', ax=ax, legend=False)
plt.tight_layout()
plt.savefig(f'figures/scatter-plots.png')
plt.show()
```




These columns have a strong obvious corellation with price:

```
- bedrooms
- bathrooms
- sqft_living
- sqft_lot
- sqft_basement
- yr_built
```

These columns are to be treated as categorical data:

```
- Condition
- Grade
- Zip Code
```

These columns do not appear to have a strong linear corellation:

```
- Waterfront
- lat
- long
- yr_sold
- yr_since_renovation
- yr_since_built
- renovated
```

We don't need to remove any data yet, as uncorellated data will appear with high p-values when we run our model.

# Features to Transform

A histogram of each variable will show us their distribution. If they aren't normal, they will need to be log transformed. This will also make clearer which of our variables are continuous.

In [6]:
```
df.hist(figsize = (20,18))
plt.tight_layout()
plt.savefig(f'figures/histogram-plots.png')
plt.show()
```


The only seemingly normal distribution is "grade". We will need to log transform all continuous variables.

continuous variables:

- bedrooms
- bathrooms
- sqft_living
- sqft_lot
- sqft_basement

- lat
- long
- yr_since_built

In [7]:
```
df.yr_sold.value_counts()
```

Out[7]:
```
2014    14622
2015     6975
Name: yr_sold, dtype: int64
```

Data is only for 2 years, so this column is unlikely to help us. We will go ahead and drop this feature.

There are also two variables in particular - price and bedrooms - where the range is far greater than the average (i.e. there are many outliers). For price, this is expected, although it will need to be dealt with in order to get a more accurate model.

In [8]:
```
df[df['bedrooms']>15]
```

Out[8]:

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | condition | grade |
|---|---|---|---|---|---|---|---|---|---|
| **15856** | 640000.0 | 33 | 1.75 | 1620 | 6000 | 1.0 | 0.0 | 5 | 7 |

For bedrooms, this is unexpected. It turns out this whole thing is caused by one data point. Given the square footage and number of bathrooms in the house is not near enough for 33 bedrooms, we believe this to be an typo, and will correct this to 3 bedrooms in our iterative modeling process. Here is what the bedrooms histogram should look like without this one error.

In [9]:
```
df.at[15856, 'bedrooms'] = 3
df.hist('bedrooms')
plt.show()
```

# What Features are Colinear

Here we check which features are colinear. We don't want colinear pairs, and will drop one from each pair.

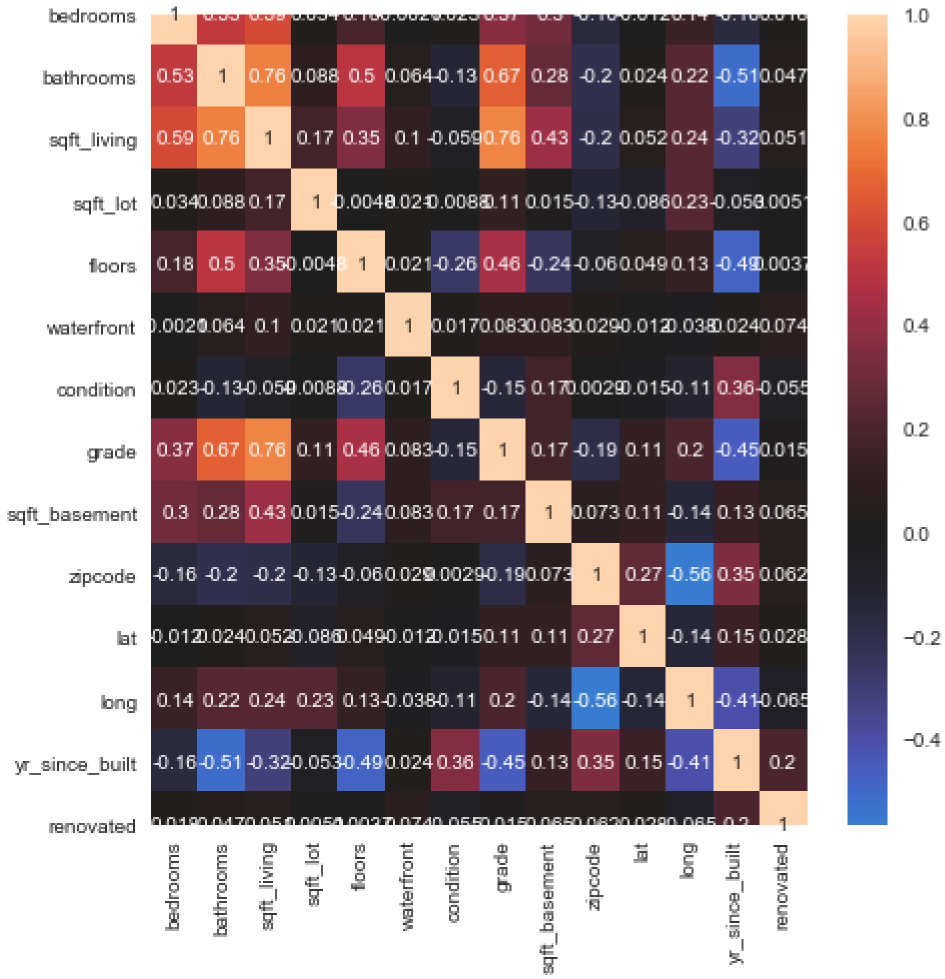
In [10]:
```
import seaborn as sns
```

In [11]:
```
corr = df.drop('price' , axis=1).corr()

df_corr=corr.abs().stack().reset_index().sort_values(0, ascending=False)
df_corr['pairs'] = list(zip(df_corr.level_0, df_corr.level_1))
df_corr.set_index(['pairs'], inplace = True)
df_corr.drop(columns=['level_1', 'level_0'], inplace = True)
df_corr.columns = ['cc']
df_corr.drop_duplicates(inplace=True)
df_corr[(df_corr.cc>.8) & (df_corr.cc <1)]
```

Out[11]:

| | **cc** |
|---|---|
| **pairs** | |
| **(yr_renovated, renovated)** | 0.999968 |
| **(yr_built, yr_since_built)** | 0.999873 |
| **(yr_since_renovation, yr_since_built)** | 0.926424 |
| **(yr_since_renovation, yr_built)** | 0.926173 |

In [17]:
```
plt.figure(figsize=(7, 7))
sns.heatmap(corr, center=0, annot=True);
plt.tight_layout()
plt.savefig(f'figures/heatmap-before.png')
plt.show()
```



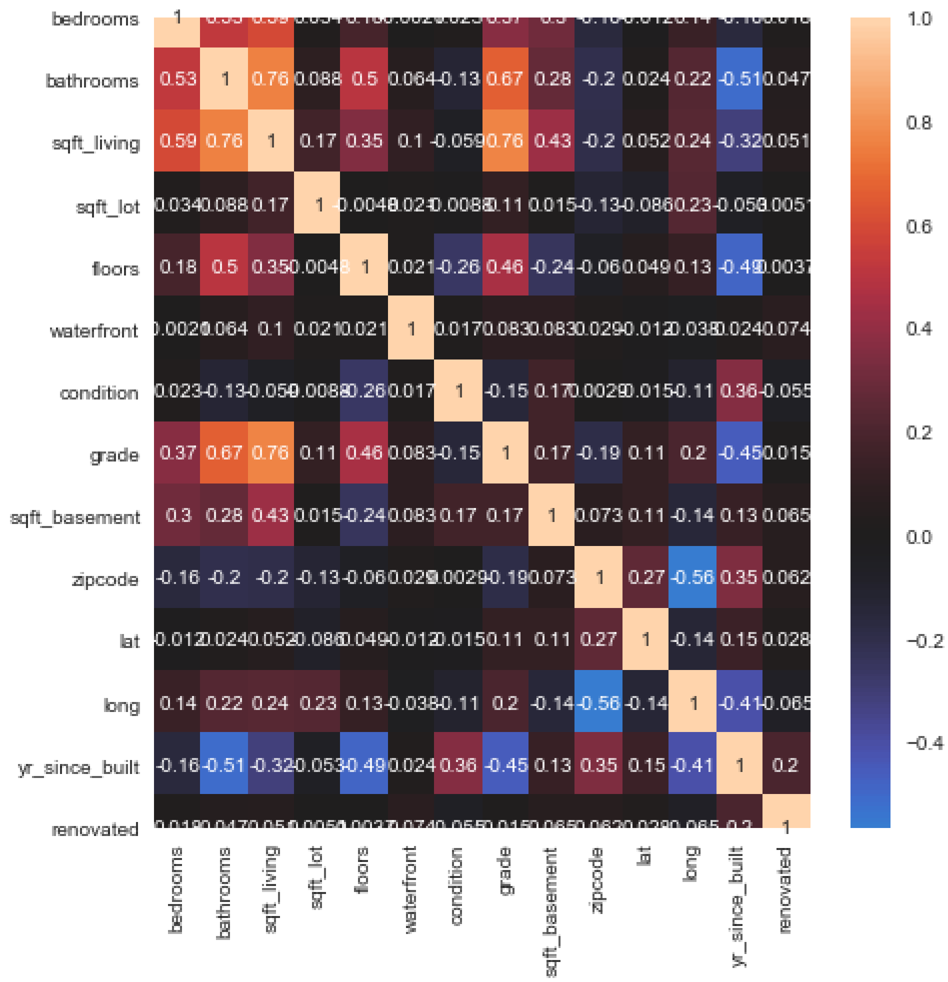
In order to remove collinearity, we will drop the yr_since_renovation, yr_built and the yr_renovated columns. These are also incorporated into other features, renovated and yr_since_built, so they will be safe to remove.

In [13]:
```
corr = df.drop(['price', 'yr_built', 'yr_renovated', 'yr_sold', 'yr_since_renovat

df_corr=corr.abs().stack().reset_index().sort_values(0, ascending=False)
df_corr['pairs'] = list(zip(df_corr.level_0, df_corr.level_1))
df_corr.set_index(['pairs'], inplace = True)
df_corr.drop(columns=['level_1', 'level_0'], inplace = True)
df_corr.columns = ['cc']
df_corr.drop_duplicates(inplace=True)
df_corr[(df_corr.cc>.8) & (df_corr.cc <1)]
```

Out[13]:

| | **cc** |
|---|---|
| **pairs** | |

In [18]:
```
plt.figure(figsize=(7, 7))
sns.heatmap(corr, center=0, annot=True);
plt.tight_layout()
plt.savefig(f'figures/heatmap-after.png')
plt.show()
```




We check for colinearity after removing the stated features to verify we no longer have any strongly colinear features. We will be sure to remove these columns once we begin our iterative modeling process.

# Feature Engineering

Here we will perform the final steps to get our data ready for modeling, split our data into train and test portions, and create our baseline model.

In [1]:
```
import pandas as pd
```

In [2]:
```
df = pd.read_csv('data/cleaned_data.csv')
df
```

Out[2]:

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

21597 rows × 19 columns

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 19 columns):
price                    21597 non-null float64
bedrooms                 21597 non-null int64
bathrooms                21597 non-null float64
sqft_living              21597 non-null int64
sqft_lot                 21597 non-null int64
floors                   21597 non-null float64
waterfront               21597 non-null float64
condition                21597 non-null int64
grade                    21597 non-null int64
sqft_basement            21597 non-null float64
yr_built                 21597 non-null int64
yr_renovated             21597 non-null float64
zipcode                  21597 non-null int64
lat                      21597 non-null float64
long                     21597 non-null float64
yr_sold                  21597 non-null int64
```

In [4]:
```
df.isna().sum()
```

Out[4]:
```
price                  0
bedrooms               0
bathrooms              0
sqft_living            0
sqft_lot               0
floors                 0
waterfront             0
condition              0
grade                  0
sqft_basement          0
yr_built               0
yr_renovated           0
zipcode                0
lat                    0
long                   0
yr_sold                0
yr_since_renovation    0
yr_since_built         0
renovated              0
dtype: int64
```

We have verified that our data has no nan values, and all data types are integers. However, we want to create dummy variables for those categories we've deemed categorical: 'floors', 'condition', 'grade', and 'zipcode'. The Pandas get_dummies function works on object datatypes, so we turn those columns into strings before running it.

In [5]:
```
categoricals = ['floors', 'condition', 'grade', 'zipcode']
df = df.astype({col: 'str' for col in categoricals})
df = pd.get_dummies(df, drop_first=True)
df
```

Out[5]:

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | waterfront | sqft_basement | yr_built |
|---|---|---|---|---|---|---|---|---|
| **0** | 221900.0 | 3 | 1.00 | 1180 | 5650 | 0.0 | 0.0 | 1955 |
| **1** | 538000.0 | 3 | 2.25 | 2570 | 7242 | 0.0 | 400.0 | 1951 |
| **2** | 180000.0 | 2 | 1.00 | 770 | 10000 | 0.0 | 0.0 | 1933 |
| **3** | 604000.0 | 4 | 3.00 | 1960 | 5000 | 0.0 | 910.0 | 1965 |
| **4** | 510000.0 | 3 | 2.00 | 1680 | 8080 | 0.0 | 0.0 | 1987 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **21592** | 360000.0 | 3 | 2.50 | 1530 | 1131 | 0.0 | 0.0 | 2009 |
| **21593** | 400000.0 | 4 | 2.50 | 2310 | 5813 | 0.0 | 0.0 | 2014 |
| **21594** | 402101.0 | 2 | 0.75 | 1020 | 1350 | 0.0 | 0.0 | 2009 |
| **21595** | 400000.0 | 3 | 2.50 | 1600 | 2388 | 0.0 | 0.0 | 2004 |
| **21596** | 325000.0 | 2 | 0.75 | 1020 | 1076 | 0.0 | 0.0 | 2008 |

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Columns: 103 entries, price to zipcode_98199
dtypes: float64(8), int64(7), uint8(88)
memory usage: 4.3 MB
```

Many punctuation marks won't work as column names with the statsmodel linear regression modeling, so we remove or replace them.

In [7]:
```
def col_formatting(col):
    for old, new in subs:
        col = col.replace(old,new)
    return col
subs = [(' ', '_'),('.',''),(',',''),("'",""),('™', ''), ('®',''),('+','plus'), (
df.columns = [col_formatting(col) for col in df.columns]
```

# Train-Test Split

We declare our train and test sets before running any modeling, using sklearn's train_test_split function. We keep its default of 25% of the data for the test set, and set a random state for repeatability. The data will also be saved to csv so they can be read later.

In [8]:
```
from sklearn.model_selection import train_test_split
```

In [9]:
```
train, test = train_test_split(df, random_state=7)
```

In [10]:
```
train
```

Out[10]:

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | waterfront | sqft_basement | yr_built |
|---|---|---|---|---|---|---|---|---|
| **15200** | 175000.0 | 3 | 1.00 | 1070 | 6164 | 0.0 | 0.0 | 1967 |
| **20737** | 775000.0 | 4 | 2.50 | 2580 | 5787 | 0.0 | 0.0 | 2007 |
| **19361** | 440000.0 | 4 | 2.50 | 2350 | 7203 | 0.0 | 0.0 | 1989 |
| **15578** | 1680000.0 | 5 | 5.25 | 4830 | 18707 | 0.0 | 900.0 | 1952 |
| **8436** | 2140000.0 | 4 | 3.75 | 5150 | 453895 | 0.0 | 790.0 | 1997 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **919** | 250000.0 | 3 | 2.00 | 1440 | 9220 | 0.0 | 0.0 | 1965 |
| **20691** | 380000.0 | 5 | 3.50 | 2420 | 4670 | 0.0 | 0.0 | 2013 |
| **5699** | 276500.0 | 4 | 1.75 | 1400 | 6650 | 0.0 | 0.0 | 1942 |
| **10742** | 440000.0 | 4 | 2.75 | 2340 | 11034 | 0.0 | 620.0 | 1967 |
| **16921** | 335000.0 | 2 | 1.75 | 1060 | 1202 | 0.0 | 300.0 | 2003 |

In [11]:
```
test
```

Out[11]:

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | waterfront | sqft_basement | yr_built |
|---|---|---|---|---|---|---|---|---|
| **5460** | 560000.0 | 4 | 2.75 | 1950 | 6192 | 0.0 | 0.0 | 1992 |
| **7131** | 500000.0 | 5 | 3.00 | 2920 | 11440 | 0.0 | 0.0 | 2003 |
| **8759** | 470000.0 | 2 | 1.00 | 1220 | 4000 | 0.0 | 0.0 | 1908 |
| **14957** | 1020000.0 | 4 | 3.00 | 2720 | 4800 | 0.0 | 930.0 | 1928 |
| **5431** | 375000.0 | 3 | 2.50 | 1930 | 6180 | 0.0 | 600.0 | 1961 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **19665** | 1850000.0 | 4 | 3.25 | 4160 | 10335 | 0.0 | 0.0 | 2014 |
| **7714** | 249900.0 | 3 | 1.00 | 1100 | 5000 | 0.0 | 0.0 | 1960 |
| **2480** | 679000.0 | 4 | 1.50 | 1420 | 4923 | 0.0 | 0.0 | 1928 |
| **16033** | 300000.0 | 3 | 1.75 | 1700 | 8481 | 0.0 | 0.0 | 1993 |
| **6193** | 450000.0 | 4 | 2.50 | 2070 | 3982 | 0.0 | 0.0 | 2004 |

In [12]:
```
train.to_csv('data/train.csv', index=False)
test.to_csv('data/test.csv', index=False)
```

# Baseline Model

To create our baseline model, we will run our train set as is through the statsmodel linear regression function. We will then collect some metrics from the model using the test data.

In [13]:
```
from statsmodels.formula.api import ols
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn')
%matplotlib inline
```

In [14]:
```
predictors = '+'.join(df.columns[1:])
formula = 'price' + '~' + predictors
model = ols(formula=formula, data=train).fit()
model.summary()
```

Out[14]:

OLS Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | price | **R-squared:** | 0.821 |
| **Model:** | OLS | **Adj. R-squared:** | 0.820 |
| **Method:** | Least Squares | **F-statistic:** | 730.0 |
| **Date:** | Sun, 29 Nov 2020 | **Prob (F-statistic):** | 0.00 |
| **Time:** | 10:30:04 | **Log-Likelihood:** | -2.1678e+05 |
| **No. Observations:** | 16197 | **AIC:** | 4.338e+05 |
| **Df Residuals:** | 16095 | **BIC:** | 4.345e+05 |
| **Df Model:** | 101 | | |
| **Covariance Type:** | nonrobust | | |

| | **coef** | **std err** | **t** | **P>\|t\|** | **[0.025** | **0.975]** |
|---|---|---|---|---|---|---|
| **Intercept** | -9.703e+07 | 8.78e+06 | -11.047 | 0.000 | -1.14e+08 | -7.98e+07 |

In [15]:
```
train_r2, train_r2_adj = model.rsquared, model.rsquared_adj
train_r2, train_r2_adj
```

Out[15]: (0.8208200222124922, 0.8196956247128626)

In [16]:
```
y_hat_train = model.predict(train.drop('price', axis=1))
y_train = train['price']
train_rmse = np.sqrt(mean_squared_error(y_train, y_hat_train))
train_rmse
```

Out[16]: 157156.21796974784

In [17]:
```
y_hat_test = model.predict(test.drop('price', axis=1))
y_test = test['price']
test_rmse = np.sqrt(mean_squared_error(y_test, y_hat_test))
test_rmse
```

Out[17]: 139700.57671817578

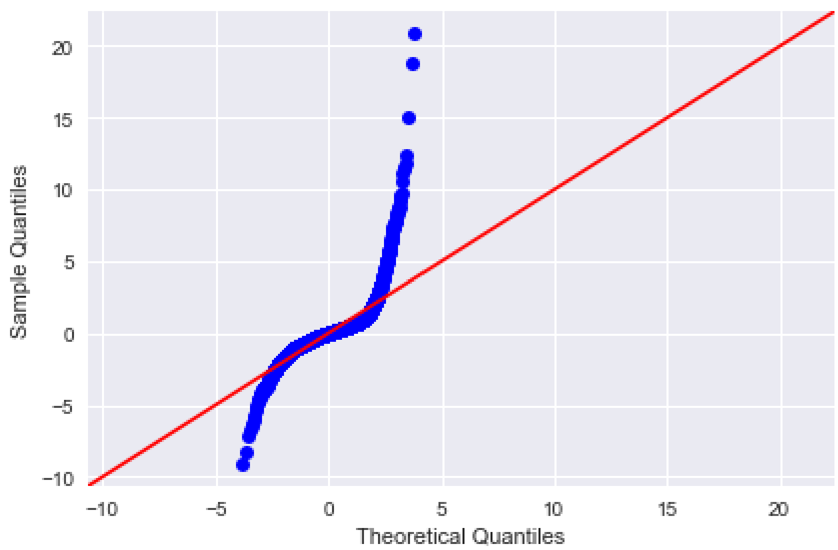
In [18]:
```
pvalues = model.pvalues.to_dict()
significant_items = {}
for key, value in pvalues.items():
    if value < 0.05:
        significant_items[key] = value
len(significant_items), len(pvalues)
```

Out[18]: (81, 103)

Baseline model:

- R2 of 0.821
- adjusted R2 of 0.820
- Train RMSE of 157156
- Test RMSE of 139700
- 81 significant features (p-value < 0.05)
- 103 features total

In [19]:
```
import statsmodels.api as sm
import scipy.stats as stats
sm.graphics.qqplot(model.resid, dist=stats.norm, line='45', fit=True)
plt.tight_layout()
plt.savefig('figures/baseline-qq-plot.png')
plt.show()
```




Our metrics of our baseline model shows we have some problems. Our R2 isn't too bad, but with such a huge number of features, it doesn't mean much. Our RMSE values are high, and our QQ plot shows our data seems to have fat tails, although it doesn't seem to have much skew.

# Iterative Modeling Process

To start our iterative modeling process, we will create a function to print out our metrics and create our qq plot. We will have to un-log-transform before calculating RMSE in order to have a comparable metric, so that will have to be included as an argument, as well as the model itself.

In [1]:
```
import pandas as pd
import statsmodels.api as sm
import scipy.stats as stats
from statsmodels.formula.api import ols
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-darkgrid')
%matplotlib inline
```

In [2]:
```
train = pd.read_csv('data/train.csv')
test = pd.read_csv('data/test.csv')
```

In [3]:
```
def get_model_data(model, is_log_transformed=False):

    train_r2, train_r2_adj = model.rsquared, model.rsquared_adj

    y_hat_train = model.predict(train.drop('price', axis=1))
    y_train = train['price']
    if is_log_transformed:
        train_rmse = np.sqrt(mean_squared_error(np.exp(y_train), np.exp(y_hat_tra
    else:
        train_rmse = np.sqrt(mean_squared_error(y_train, y_hat_train))

    y_hat_test = model.predict(test.drop('price', axis=1))
    y_test = test['price']
    if is_log_transformed:
        test_rmse = np.sqrt(mean_squared_error(np.exp(y_test), np.exp(y_hat_test)
    else:
        test_rmse = np.sqrt(mean_squared_error(y_test, y_hat_test))

    pvalues = model.pvalues.to_dict()
    significant_items = {}
    for key, value in pvalues.items():
        if value < 0.05:
            significant_items[key] = value

    print('R2 =', train_r2)
    print('R2 adjusted =', train_r2_adj)
    print('RMSE (train) =', train_rmse)
    print('RMSE (test) =', test_rmse)
    print('number of significant features =', len(significant_items))

    sm.graphics.qqplot(model.resid, dist=stats.norm, line='45', fit=True)
    plt.title('Q-Q Plot')
```

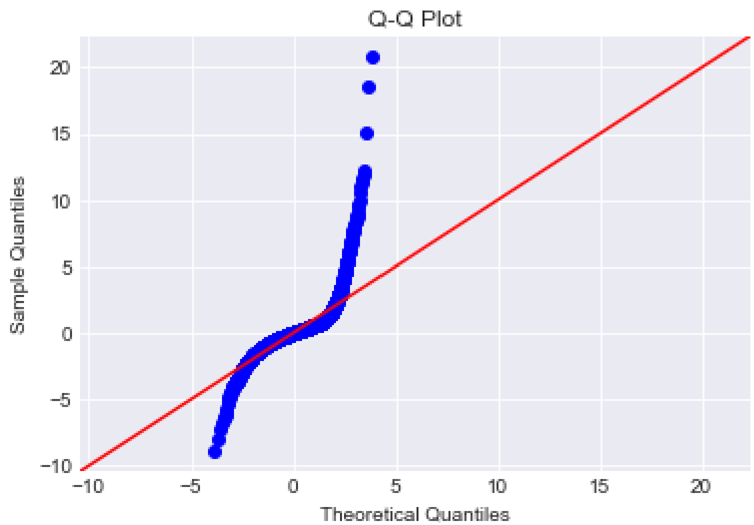
## Dropping Colinear Features

Here we drop the colinear features, as established in our exploratory data analysis.

In [4]:
```
train.drop(['yr_built', 'yr_renovated', 'yr_sold', 'yr_since_renovation'] , axis=
test.drop(['yr_built', 'yr_renovated', 'yr_sold', 'yr_since_renovation'] , axis=1
```

In [5]:
```
predictors = '+'.join(train.columns[1:])
formula = 'price' + '~' + predictors
model = ols(formula=formula, data=train).fit()
get_model_data(model)
```

```
R2 = 0.8189229383789512
R2 adjusted = 0.8178205932404954
RMSE (train) = 157985.98012359045
RMSE (test) = 140609.21109533816
number of significant features = 78
```




## Removing Outliers

Our data has a lot of high outliers in the price variable, and one very high bedroom count that is likely a typo. We will fix the one error, then remove the outlighers higher than three times the standard deviation away from the mean. This will limit the range of house prices we can predict for, but will make for a much more accurate model. The mean and standard deviation our taken from our exploratory data analysis.

In [6]:
```
train[train['bedrooms']>15]
```

Out[6]:

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | waterfront | sqft_basement | lat | |
|---|---|---|---|---|---|---|---|---|---|
| **4196** | 640000.0 | 33 | 1.75 | 1620 | 6000 | 0.0 | 580.0 | 47.6878 | -1: |

1 rows × 99 columns

In [7]:
```
train.at[4196, 'bedrooms'] = 3
train[train['bedrooms']>15]
```

Out[7]:

| price | bedrooms | bathrooms | sqft_living | sqft_lot | waterfront | sqft_basement | lat | long | yr_since_ |
|---|---|---|---|---|---|---|---|---|---|

0 rows × 99 columns

In [8]:
```
mean = 5.402966e+05
std = 3.673681e+05
lower_cutoff = mean - (3*std)
upper_cutoff = mean + (3*std)
lower_cutoff, upper_cutoff
```

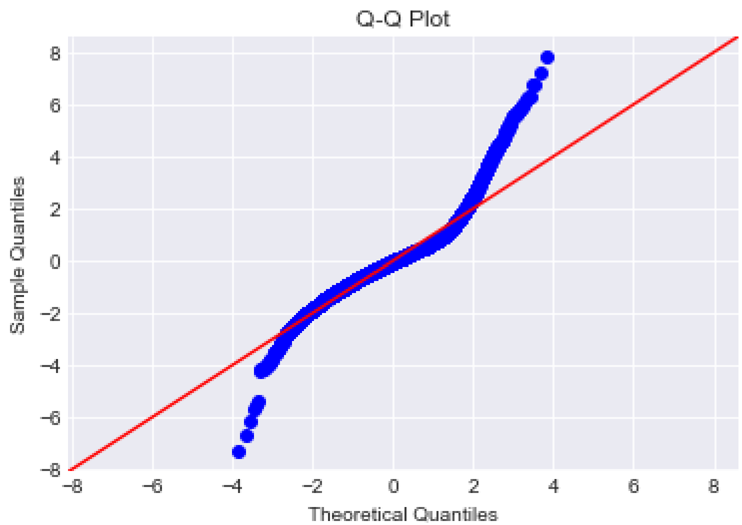
Out[8]:
```
(-561807.6999999998, 1642400.9)
```

Our lower cutoff is negative. Since we have no negative prices (our minimum from our exploratory data analysis was $78,000, and negative prices just don't make sense), we don't have to worry about our lower cutoff.

In [9]:
```
train = train[train['price'] < upper_cutoff]
test = test[test['price'] < upper_cutoff]
```

In [10]:
```
predictors = '+'.join(train.columns[1:])
formula = 'price' + '~' + predictors
model = ols(formula=formula, data=train).fit()
get_model_data(model)
```

```
R2 = 0.8264414496035375
R2 adjusted = 0.8253755271400415
RMSE (train) = 107801.0153531246
RMSE (test) = 105098.35236656363
number of significant features = 82
```




The model looks a lot better after dropping values higher than three times the standard deviation. However, now we only have data within three times the standard deviation to work with. Our model will now only accurately predict house prices up to $1,642,400.

# Log Transforming Continuous Variables

We will log transform most of our continuous variables in order to make them more normally distributed. We won't transform longitude and latitude, as they can have negative values. yr_since_built and sqft_basement, as well, can be zero. Log transformations do not work on negative and zero values, so we will skip these columns.

The price column, our dependent variable, will be log transformed as well. This will get us better results, but it means we need to do the inverse transformation when calculating rmse values in order to get a comparable result.

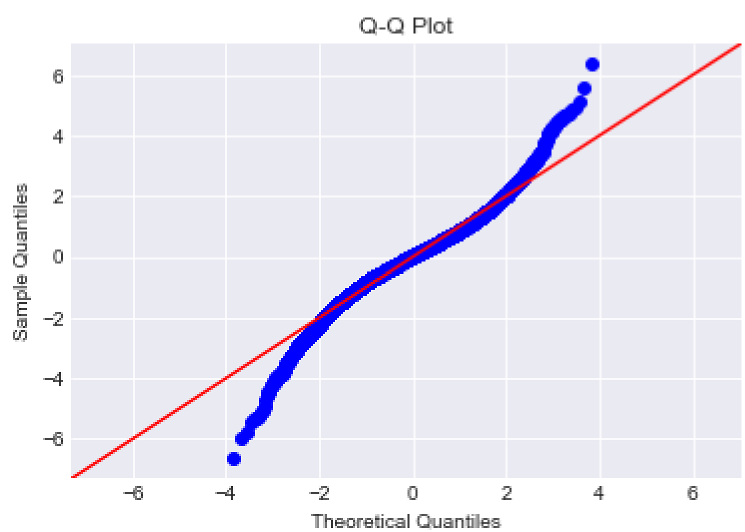
In [11]:
```
train.columns
```

Out[11]:
```
Index(['price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot',
       'waterfront', 'sqft_basement', 'lat', 'long', 'yr_since_built',
       'renovated', 'floors_15', 'floors_20', 'floors_25', 'floors_30',
       'floors_35', 'condition_2', 'condition_3', 'condition_4', 'condition_
5',
       'grade_11', 'grade_12', 'grade_13', 'grade_3', 'grade_4', 'grade_5',
       'grade_6', 'grade_7', 'grade_8', 'grade_9', 'zipcode_98002',
       'zipcode_98003', 'zipcode_98004', 'zipcode_98005', 'zipcode_98006',
       'zipcode_98007', 'zipcode_98008', 'zipcode_98010', 'zipcode_98011',
       'zipcode_98014', 'zipcode_98019', 'zipcode_98022', 'zipcode_98023',
       'zipcode_98024', 'zipcode_98027', 'zipcode_98028', 'zipcode_98029',
       'zipcode_98030', 'zipcode_98031', 'zipcode_98032', 'zipcode_98033',
       'zipcode_98034', 'zipcode_98038', 'zipcode_98039', 'zipcode_98040',
       'zipcode_98042', 'zipcode_98045', 'zipcode_98052', 'zipcode_98053',
       'zipcode_98055', 'zipcode_98056', 'zipcode_98058', 'zipcode_98059',
       'zipcode_98065', 'zipcode_98070', 'zipcode_98072', 'zipcode_98074',
       'zipcode_98075', 'zipcode_98077', 'zipcode_98092', 'zipcode_98102',
       'zipcode_98103', 'zipcode_98105', 'zipcode_98106', 'zipcode_98107',
       'zipcode_98108', 'zipcode_98109', 'zipcode_98112', 'zipcode_98115',
```

In [12]:
```
continuous = ['price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot']
for col in continuous:
    train[col] = train[col].map(np.log)
    test[col] = test[col].map(np.log)
```

In [13]:
```
predictors = '+'.join(train.columns[1:])
formula = 'price' + '~' + predictors
model = ols(formula=formula, data=train).fit()
get_model_data(model, True)
```

```
R2 = 0.8566667276430895
R2 adjusted = 0.8557864359235365
RMSE (train) = 103013.87299746794
RMSE (test) = 99717.16956285348
number of significant features = 90
```



# Removing Insignificant Features

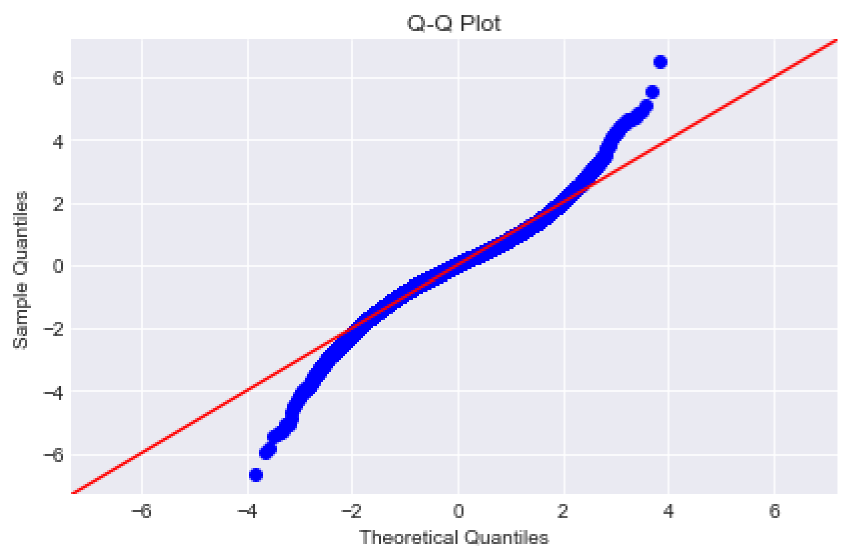
Here we loop through our model multiple times. Each time we find the feature with the highest p-value and remove it if it is higher than 0.05. By doing this, we get rid of 9 insignificant features. By reviewing the model summary, we can verify that we are left with only significant ones.

In [14]:
```
model_dict = list(dict(model.pvalues).items())
model_dict.sort(key = lambda x: x[1], reverse=True)
highest_pvalue = model_dict[0]

while highest_pvalue[1] > 0.05:
    print(f'Dropping "{highest_pvalue[0]}" with p-value {highest_pvalue[1]}')
    train.drop(highest_pvalue[0], inplace = True, axis = 1)
    test.drop(highest_pvalue[0], inplace = True, axis = 1)

    predictors = '+'.join(train.columns[1:])
    formula = 'price' + '~' + predictors
    model = ols(formula=formula, data=train).fit()

    model_dict = list(dict(model.pvalues).items())
    model_dict.sort(key = lambda x: x[1], reverse=True)
    highest_pvalue = model_dict[0]
```

```
Dropping "zipcode_98003" with p-value 0.9081859817590578
Dropping "zipcode_98188" with p-value 0.780713820714147
Dropping "floors_20" with p-value 0.6356104612914132
Dropping "grade_3" with p-value 0.5856472881210619
Dropping "zipcode_98002" with p-value 0.5406474710298468
Dropping "floors_35" with p-value 0.349191640439133
Dropping "floors_25" with p-value 0.32369742178652816
Dropping "zipcode_98148" with p-value 0.15467038496210414
Dropping "zipcode_98198" with p-value 0.21086855197031598
```

In [15]:
```
get_model_data(model, True)
plt.tight_layout()
plt.savefig('figures/final-qq-plot.png')
plt.show()
```

```
R2 = 0.8566083994927416
R2 adjusted = 0.8558099143415273
RMSE (train) = 103086.41637514034
RMSE (test) = 99701.98273002672
number of significant features = 90
```




In [16]:
```
model.summary()
```

Out[16]:

OLS Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | price | **R-squared:** | 0.857 |
| **Model:** | OLS | **Adj. R-squared:** | 0.856 |
| **Method:** | Least Squares | **F-statistic:** | 1073. |
| **Date:** | Sun, 29 Nov 2020 | **Prob (F-statistic):** | 0.00 |
| **Time:** | 10:31:48 | **Log-Likelihood:** | 4484.5 |
| **No. Observations:** | 15892 | **AIC:** | -8791. |
| **Df Residuals:** | 15803 | **BIC:** | -8108. |
| **Df Model:** | 88 | | |
| **Covariance Type:** | nonrobust | | |

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **Intercept** | -86.9633 | 7.303 | -11.908 | 0.000 | -101.278 | -72.649 |

## Adding Interaction Features

We use the combinations function to create every possible combination of our continuous variables. Looping over them, we create a new interaction variable in our training set, dropping the two originals. We then make a new model with this train set, and check its R2 and adjusted R2.

In [17]:
```
from itertools import combinations
```

In [18]:
```
interactions = pd.DataFrame(columns = ['interaction', 'r2', 'r2_adj'])
interactions = interactions.append({'interaction':'baseline', 'r2':0.856608399492
continuous = ['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'sqft_basement'
feat_combinations = combinations(continuous, 2)
for i, (a, b) in enumerate(feat_combinations):
    train_interactions = train.copy()
    train_interactions['interaction'] = train_interactions[a] * train_interaction
    train_interactions.drop([a, b], axis=1, inplace=True)

    predictors_interactions = '+'.join(train_interactions.columns[1:])
    formula_interactions = 'price' + '~' + predictors_interactions
    model_interactions = ols(formula=formula_interactions, data=train_interaction
    train_r2, train_r2_adj = model_interactions.rsquared, model_interactions.rsqu

    interactions = interactions.append({'interaction':f'{a}*{b}', 'r2':train_r2,
```

In [19]:
```
interactions.sort_values(by=['r2'], ascending=False).head()
```

Out[19]:

| | interaction | r2 | r2_adj |
|---|---|---|---|
| **0** | baseline | 0.856608 | 0.855810 |
| **26** | lat*long | 0.856450 | 0.855660 |
| **4** | bedrooms*sqft_basement | 0.856086 | 0.855293 |
| **7** | bedrooms*yr_since_built | 0.856004 | 0.855212 |
| **16** | sqft_living*lat | 0.855948 | 0.855155 |

In [20]:
```
interactions.sort_values(by=['r2_adj'], ascending=False).head()
```

Out[20]:

| | interaction | r2 | r2_adj |
|---|---|---|---|
| **0** | baseline | 0.856608 | 0.855810 |
| **26** | lat*long | 0.856450 | 0.855660 |
| **4** | bedrooms*sqft_basement | 0.856086 | 0.855293 |
| **7** | bedrooms*yr_since_built | 0.856004 | 0.855212 |
| **16** | sqft_living*lat | 0.855948 | 0.855155 |

It looks like every interaction decreases both our R2 and adjusted R2. So, we will not include any interaction variables in our model.

# Cross Validation

We cross validate our data in order to verify that we aren't overfitting to our test set, and that there aren't outliers in our test set skewing the results. statsmodel does not have its own cross validation function, so we split the total data into parts manually, and loop over the resulting folds and calculate the rmse of each fold individually. The resulting values are near enough to each other that we can call this test a success.

In [21]:
```
def kfolds(data, k):
    # Force data as pandas DataFrame
    data = pd.DataFrame(data)
    num_observations = len(data)
    fold_size = num_observations//k
    leftovers = num_observations%k
    folds = []
    start_obs = 0
    for fold_n in range(k):
        if fold_n < leftovers:
            #Fold Size will be 1 larger to account for leftovers
            fold = data.iloc[start_obs : start_obs+fold_size+1]
            start_obs += 1
        else:
            fold = data.iloc[start_obs : start_obs+fold_size]
        start_obs += fold_size
        folds.append(fold)

    return folds
```

In [22]:
```
test_errs = []
k=5
folds = kfolds(pd.concat([train, test]).sample(frac=1, random_state=100), k)
for n in range(k):
    # Split in train and test for the fold
    fold_train = pd.concat([fold for i, fold in enumerate(folds) if i!=n])
    fold_test = folds[n]
    # Fit a linear regression model
    predictors = '+'.join(fold_train.columns[1:])
    formula = 'price' + '~' + predictors
    fold_model = ols(formula=formula, data=fold_train).fit()
    #Evaluate Test Errors
    y_hat_fold_test = fold_model.predict(fold_test.drop('price', axis=1))
    y_fold_test = fold_test['price']
    test_errs.append(np.sqrt(mean_squared_error(np.exp(y_fold_test), np.exp(y_hat_
print(test_errs)
```

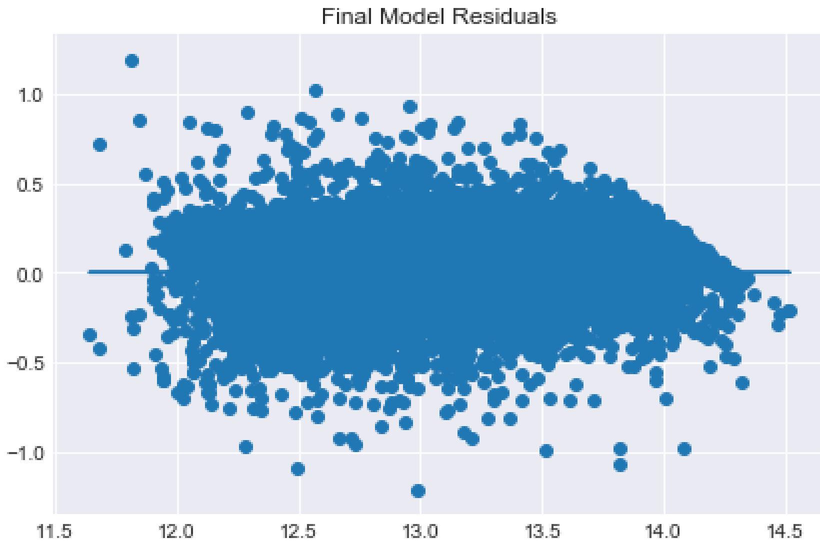
```
[102263.4216279941, 103702.93643726473, 105491.1945506598, 100820.10288828705,
102073.61737834036]
```

Our cross validation scores are not far off from each other, so we do not have a problem.

# Residual Plots

Our residuals plot tells us where our errors lie. The distribution seems mostly evenly distributed around the horizontal axis, and not skewed left or right, so we have no problems here. We can save the parameters of our model to a csv to analyze later.

In [23]:
```
plt.scatter(model.predict(train.drop('price', axis = 1)), model.resid)
plt.plot(model.predict(train.drop('price', axis = 1)), [0 for i in range(len(trai
plt.title('Final Model Residuals')
plt.tight_layout()
plt.savefig('figures/final-residuals-plot.png')
plt.show()
```




In [24]:
```
model.summary()
```

Out[24]:

OLS Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | price | **R-squared:** | 0.857 |
| **Model:** | OLS | **Adj. R-squared:** | 0.856 |
| **Method:** | Least Squares | **F-statistic:** | 1073. |
| **Date:** | Sun, 29 Nov 2020 | **Prob (F-statistic):** | 0.00 |
| **Time:** | 10:32:11 | **Log-Likelihood:** | 4484.5 |
| **No. Observations:** | 15892 | **AIC:** | -8791. |
| **Df Residuals:** | 15803 | **BIC:** | -8108. |
| **Df Model:** | 88 | | |
| **Covariance Type:** | nonrobust | | |

| | **coef** | **std err** | **t** | **P>\|t\|** | **[0.025** | **0.975]** |
|---|---|---|---|---|---|---|
| **Intercept** | -86.9633 | 7.303 | -11.908 | 0.000 | -101.278 | -72.649 |

In [25]:
```
model.params.to_csv('data/final_model.csv', header=False)
```

# Final Model Analysis

Finally, it is time to analyze our final model. We will greate a prediction function, examine our coefficients, and draw conclusions.

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
df = pd.read_csv('data/final_model.csv', header = None, names = ['names', 'coeffi
coef = df['coefficients'].to_dict()
```

## Prediction Function

To create our prediction function, we will start off with the data in the format it is in the dataset we were originally given, king_house_data.csv. Since we are using it with our coefficients dictionary, we don't actually need to remove the extra data: we can just only use data that is in both dictionaries. We do, however, need to turn all possible data into numeric data types, log transform the necessary data, and inverse log transform (np.exp) the price at the end. Some data will need to be kept as a string in order to create dummy variables from it. Finally, we replace all nan values and turn the data into a dictionary. We can check this on a single row, but we also read in the first 5 lines from king_house_data.csv and try those.

In [3]:
```
def predict(data):
    data = data.split(',')
    columns = 'id,date,price,bedrooms,bathrooms,sqft_living,sqft_lot,floors,water
    df = pd.DataFrame(dict(zip(columns, data)), index=[0])
    df['yr_sold'] = df.date.map(lambda x: int(x.split('/')[-1]))
    df.drop('date', axis=1, inplace=True)
    for col in df.columns:
        df[col] = pd.to_numeric(df[col])
    df['yr_since_renovation'] = np.where(df['yr_renovated']==0.0, df['yr_sold']-d
    df['yr_since_built'] = df['yr_sold'] - df['yr_built']
    categoricals = ['floors', 'condition', 'grade', 'zipcode']
    df = df.astype({col: 'str' for col in categoricals})
    df = pd.get_dummies(df)
    df['renovated'] = df.yr_renovated.map(lambda x: 1 if x>0 else 0)
    continuous = ['price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot']
    for col in continuous:
        df[col] = df[col].map(np.log)
    df.replace(np.nan, 0, inplace=True)
    data_dict = df.iloc[0].to_dict()

    prediction = coef['Intercept']
    for key, value in coef.items():
        prediction += value * data_dict.get(key, 0)
    prediction = np.exp(prediction)
    return round(prediction, 2)
```

In [4]:
```
data = '7129300520,10/13/2014,221900.0,3,1.0,1180,5650,1.0,,0.0,3,7,1180,0.0,1955
predict(data)
# real price value: 221900.0
```

Out[4]: 239005.82

In [5]:
```
# todo: import data from kc_house_data.csv and run the prediction function
with open('data/kc_house_data.csv') as f:
    f.readline()
    for i in range(5):
        print(predict(f.readline()))
# the real price values are: 221900.0, 538000.0, 180000.0, 604000.0, and 510000.0
```

```
239005.82
576006.35
239994.68
556154.46
471701.59
```

# Coefficient Analysis

In [6]:
```
coef
```

Out[6]:
```
{'Intercept': -86.96327587506237,
 'bedrooms': -0.05790908383811544,
 'bathrooms': 0.06216451675867673б,
 'sqft_living': 0.4837753401719039,
 'sqft_lot': 0.06663803910671842,
 'waterfront': 0.6027962254517414,
 'sqft_basement': -5.32076652954763e-05,
 'lat': 0.6472578402367359,
 'long': -0.5282742969088292,
 'yr_since_built': 0.00040867761814662815,
 'renovated': 0.06219396721553915,
 'floors_15': 0.015254015266802584,
 'floors_30': -0.0635517536782565,
 'condition_2': 0.18927766124862488,
 'condition_3': 0.3134553949417471,
 'condition_4': 0.3483349587318392,
 'condition_5': 0.3986555695222103,
 'grade_11': 0.13444823043057247,
 'grade_12': 0.17757036224366418,
```

bedrooms: -0.057155141850407175

- each bedroom decreases the sale price of a house by 5%

bathrooms: 0.062049951410309764

- each bathroom increases the sale price of a house by 6%

sqft_living: 0.4834287268804242

- a 1% change in square footage living area increases the sale price of a house by .48%

sqft_lot: 0.06665523673039399

- a 1% change in square footage lot area increases the sale price of a house by .07%

waterfront: 0.6029579655205117

- if the house is on the waterfront, the sale price of a house increases by 60%

sqft_basement: -5.318890782739171e-05

- a 1% change in square footage basement area decreases the sale price of a house by .00005%

lat: 0.647575198704021

- if you move north, a 1 degree increase in latitude increases the sale price of a house by 65%

long: -0.5281545373156602

- if you move east, a 1 degree increase in longitude decreases the sale price of a house by 53%

yr_since_built: 0.0004074831112883181

- a 1-year increase in the age of a house increases its sale price by .04%

renovated: 0.062242978234706765

- a house that has been renovated has its sale price increased by 6%

floors_15: 0.015254015266802584 and floors_30: -0.0635517536782565

- using a one-floor house as a baseline, a 1.5-floor house has its price increased by 1.5%, while a 3-floor house has its price decreased by 6.4%. Other numbers of floors are approximately equal in price to a 1-floor house.

condition_2: 0.18927766124862488, condition_3: 0.3134553949417471, condition_4: 0.3483349587318392, and condition_5: 0.3986555695222103

- using a condition of 1 as a baseline, a condition of 2 increases the price by 19%, a condition of 3 increases the price of a house by 31%, a condition of 4 increases the price by 35%, and a condition of 5 increases the price by 40%

Using zipcode 98001 as a baseline, the listed zipcodes increase or decrease the price of a house by 100 times the number listed as a percentage. The unlisted zipcodes are all approximately the same in price. (I'm not planning on listing them all out.)

# Recommendations and Future Work

We attempted to add interation features to our model, but our results indicated that they only decreased the accuracy of our model. With more time, we could take a deeoer look at these and find out why that is the case, and see if other interactions could help our model.

Similarly, adding polynomial features could make our predictions more accurate. Trial-and-error would be needed to determine which features could be changed in this way to improve our model.

Using a mapping library could turn the longitude and latitude into more directly beneficial information, like distance to a school or grocery store. With more time, we could create new features using this information to add to our model.

# Conclusions

Our final model will be useful in predicting sale prices of houses in King county. We can use these predictions to help our clients set the prices for their houses, and find houses that are currently underpriced.