Project 2: Multiple Linear Regression On Housing Data (King County, WA)

In [1]:
```
from IPython.display import Image
#Image(url= "https://upload.wikimedia.org/wikipedia/commons/thumb/2/23/Space_
Image(url= "images/1200px-Space_Needle_2011-07-04.jpg", width=400, height=400
```

Out[1]:

Final Project Submission

Please fill out:

- Student name: Eric Hansen
- Student pace: self paced / part time / full time --- Self Paced
- Scheduled project review date/time: 9/7/21 6:30 EST
- Instructor name: Jeff Herman
- Blog post URL: https://ericthansen.medium.com/linear-regression-and-assumption-validity-94712f714ea4 (https://ericthansen.medium.com/linear-regression-and-assumption-validity-94712f714ea4)

Introduction and assumptions

In this project, we are given agency to assume our stakeholder (e.g. house sellers, buyers, real estate agents, flippers, etc.)
For most, if not all, of these parties, it seems likely that 'price' will be a desirable target variable.

For this project, we will postulate that our stakeholder is a family looking to sell their house in the Seattle area, and potentially relocate to another house in the area.

Our data source (King's County, WA home sale price) contains price (target variable) for sold houses in 2014-2015, along with a plethora of other potential predictor variables. Therefore, our data source is a viable tool to inform choices about house purchase/sale.

Relevant questions might be:

- What modifications can be made to their home to increase price?
- What characteristics could be desirably low to get a good deal on the future house?
- What is a good time to buy/sell? I.e. are there months or days of the week that work better?

In general, I will aim to use the OSEMN framework, though due to the iterative nature, some steps blend into each other.

- Obtain
- Scrub
- Explore
- Model
- INterpret

In [2]:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.impute import MissingIndicator
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
#from sklearn import metrics
import sklearn.metrics as metrics


from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler

from statsmodels.api import OLS
import statsmodels.api as sm
from statsmodels.formula.api import ols
from statsmodels.stats.outliers_influence import variance_inflation_factor

import statsmodels.stats.api as sms
import scipy.stats as stats


from random import randrange, randint

import warnings
warnings.filterwarnings('ignore')
warnings.simplefilter('ignore')

import numpy as np

import pandas as pd
pd.get_option("display.max_columns")
# settings to display all columns
pd.set_option("display.max_columns", None)
# display the dataframe head


verbose = True
```

In [3]:

```
df = pd.read_csv('data/kc_house_data.csv')
if verbose:
    display(df.head())
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wate
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	

Column Names and descriptions for King County Data Set

- **id** - unique identified for a house
- **dateDate** - house was sold
- **pricePrice** - is prediction target
- **bedroomsNumber** - of Bedrooms/House
- **bathroomsNumber** - of bathrooms/bedrooms
- **sqft_livingsquare** - footage of the home
- **sqft_lotsquare** - footage of the lot
- **floorsTotal** - floors (levels) in house
- **waterfront** - House which has a view to a waterfront
- **view** - Has been viewed
- **condition** - How good the condition is (Overall)
- **grade** - overall grade given to the housing unit, based on King County grading system
- **sqft_above** - square footage of house apart from basement
- **sqft_basement** - square footage of the basement
- **yr_built** - Built Year
- **yr_renovated** - Year when house was renovated
- **zipcode** - zip
- **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

Data understanding - the description for view is especially vague; we'd like more detail on all of these that are relevant to saleprice, but can explore that as additional step.

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

```
<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  float64
 9   view           21534 non-null  float64
 10  condition      21597 non-null  int64  
 11  grade          21597 non-null  int64  
 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(8), int64(11), object(2)
memory usage: 3.5+ MB
```

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
sqft_living15      0
sqft_lot15         0
dtype: int64
```

There are some na values that need to be cleaned.

There are some na values that need to be cleaned.

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

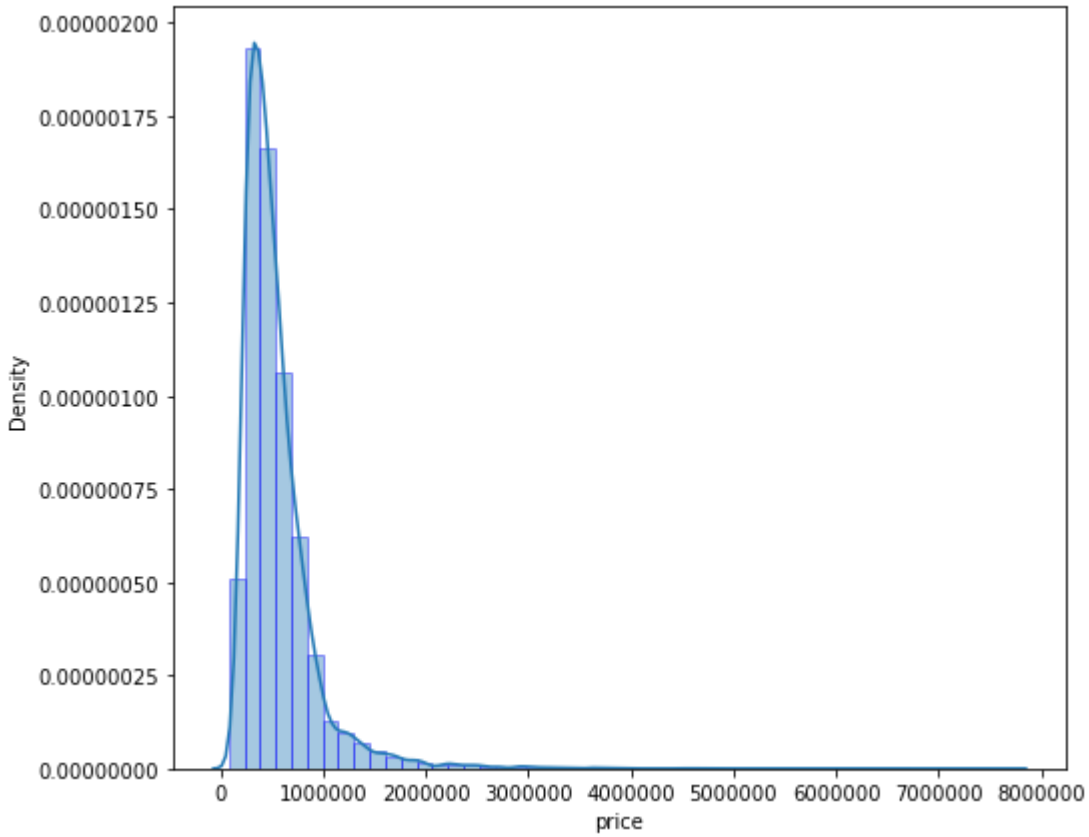
Out[5]:

	count	mean	std	min	25%	50
id	21597.0	4.580474e+09	2.876736e+09	1.000102e+06	2.123049e+09	3.904930e+
price	21597.0	5.402966e+05	3.673681e+05	7.800000e+04	3.220000e+05	4.500000e+
bedrooms	21597.0	3.373200e+00	9.262989e-01	1.000000e+00	3.000000e+00	3.000000e+
bathrooms	21597.0	2.115826e+00	7.689843e-01	5.000000e-01	1.750000e+00	2.250000e+
sqft_living	21597.0	2.080322e+03	9.181061e+02	3.700000e+02	1.430000e+03	1.910000e+
sqft_lot	21597.0	1.509941e+04	4.141264e+04	5.200000e+02	5.040000e+03	7.618000e+
floors	21597.0	1.494096e+00	5.396828e-01	1.000000e+00	1.000000e+00	1.500000e+
waterfront	19221.0	7.595859e-03	8.682485e-02	0.000000e+00	0.000000e+00	0.000000e+
view	21534.0	2.338627e-01	7.656862e-01	0.000000e+00	0.000000e+00	0.000000e+
condition	21597.0	3.409825e+00	6.505456e-01	1.000000e+00	3.000000e+00	3.000000e+
grade	21597.0	7.657915e+00	1.173200e+00	3.000000e+00	7.000000e+00	7.000000e+
sqft_above	21597.0	1.788597e+03	8.277598e+02	3.700000e+02	1.190000e+03	1.560000e+
yr_built	21597.0	1.971000e+03	2.937523e+01	1.900000e+03	1.951000e+03	1.975000e+
yr_renovated	17755.0	8.363678e+01	3.999464e+02	0.000000e+00	0.000000e+00	0.000000e+
zipcode	21597.0	9.807795e+04	5.351307e+01	9.800100e+04	9.803300e+04	9.806500e+
lat	21597.0	4.756009e+01	1.385518e-01	4.715590e+01	4.747110e+01	4.757180e+
long	21597.0	-1.222140e+02	1.407235e-01	-1.225190e+02	-1.223280e+02	-1.222310e+
sqft_living15	21597.0	1.986620e+03	6.852305e+02	3.990000e+02	1.490000e+03	1.840000e+
sqft_lot15	21597.0	1.275828e+04	2.727444e+04	6.510000e+02	5.100000e+03	7.620000e+

Note that "long" is negative and several have min value of 0 - be careful for log transformation.

In [6]:

```
# Price distribution

plt.figure(figsize=(8, 7))
sns.distplot(df['price'], bins=50, hist_kws=dict(edgecolor="blue", linewidth=

plt.ticklabel_format(style='plain')
```



Outliers

For our "client", we aren't interested in exceedingly high-value homes anyway, and these outliers could disrupt other trends.
So, we will just remove them now.

In [7]:

```
for i in range(90, 100):
    q = i / 100
    print('{} percentile: {}'.format(q, df['price'].quantile(q=q)))
outlier_cutoff = 2 * 10**6
#remove the outliers!  This is worth returning to
subset = df[df['price'] < outlier_cutoff]
print('Percent removed for subset:',(len(df) - len(subset))/len(df))

df = subset
```

```
0.9 percentile: 887000.0
0.91 percentile: 919993.6
0.92 percentile: 950000.0
0.93 percentile: 997964.0000000002
0.94 percentile: 1060000.0
0.95 percentile: 1160000.0
0.96 percentile: 1260000.0
0.97 percentile: 1390000.0
0.98 percentile: 1600000.0
0.99 percentile: 1970000.0
Percent removed for subset: 0.009630967263971849
```

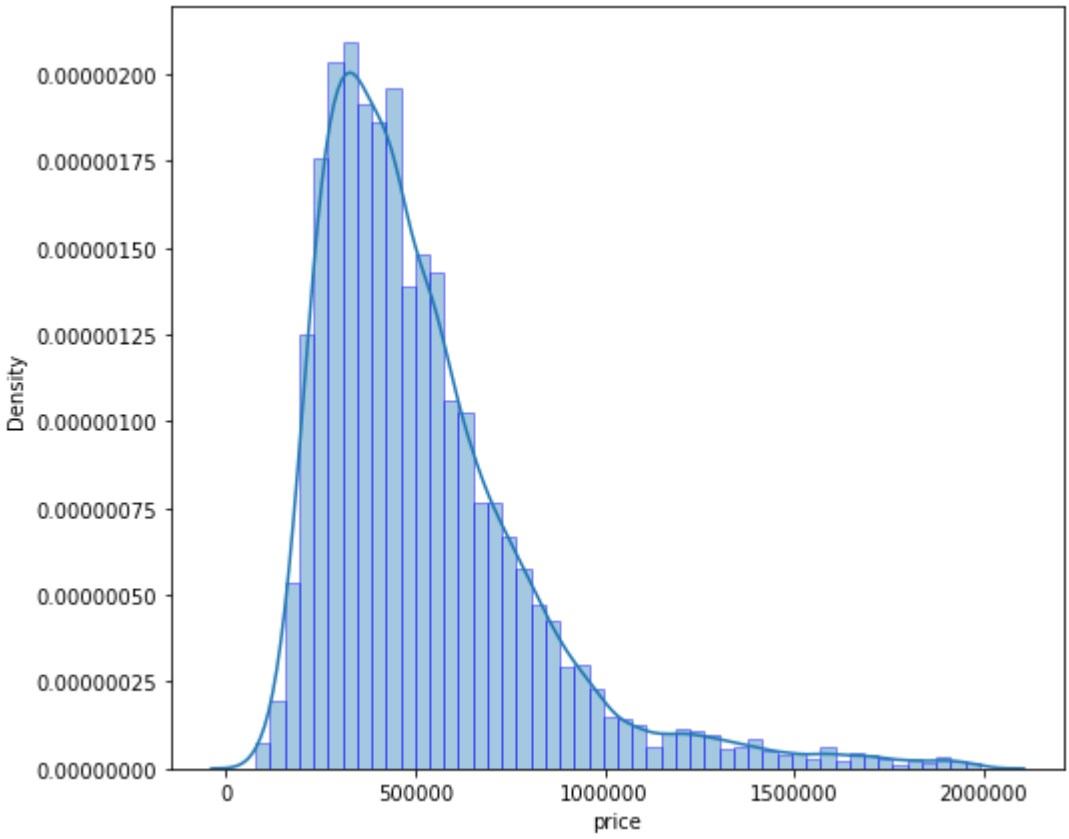
In [8]:

```
#Similarly, let's check on low outliers

for i in range(0, 10):
    q = i / 100
    print('{} percentile: {}'.format(q, df['price'].quantile(q=q)))
#those low value homes are valid, and in significant enough proportion that t
```

```
0.0 percentile: 78000.0
0.01 percentile: 153502.64
0.02 percentile: 175000.0
0.03 percentile: 191000.0
0.04 percentile: 202000.0
0.05 percentile: 210000.0
0.06 percentile: 219900.0
0.07 percentile: 225032.80000000002
0.08 percentile: 233000.0
0.09 percentile: 240000.0
```

In [9]:

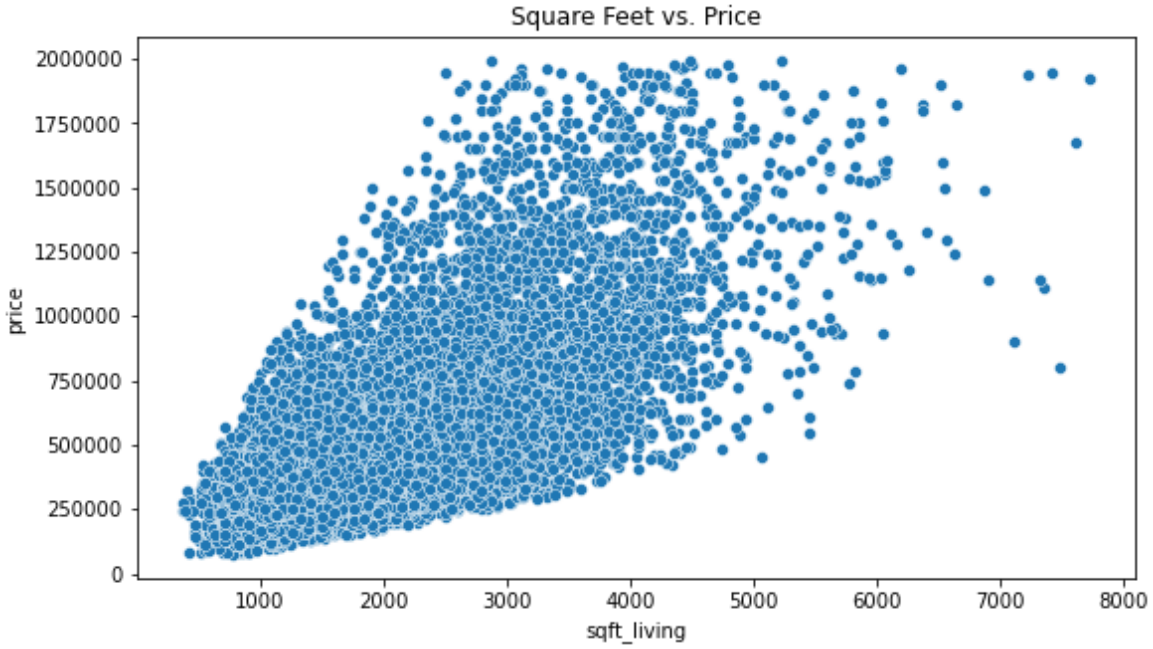
```
#After outliers removed:
# Price distribution

plt.figure(figsize=(8, 7))
sns.distplot(subset['price'], bins=50, hist_kws=dict(edgecolor="blue", linewi

plt.ticklabel_format(style='plain')
```



This price graph looks a lot more normal. We could potentially even remove more.

In [10]:

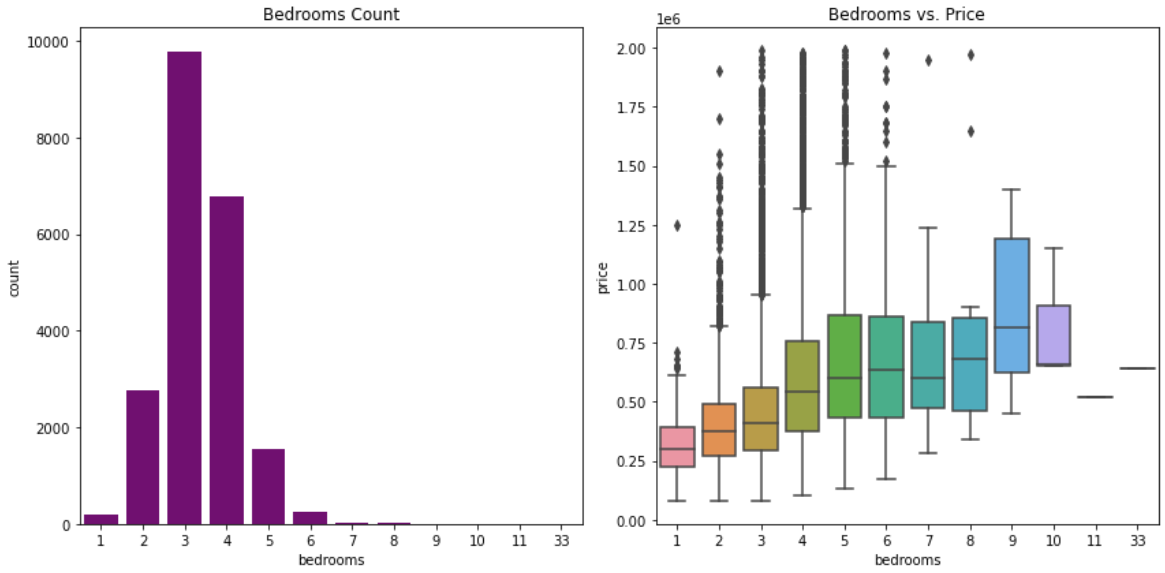
```
#A quick look at sqft vs price
plt.figure(figsize=(9, 5))
sns.scatterplot(x='sqft_living', y='price', data=df).set_title('Square Feet v
plt.ticklabel_format(style='plain')#style='plain', 'sci', 'scientific'
```



Certainly there is a positive correlation.

In [11]:

```
#Investigating bedrooms vs price

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('Bedrooms Count')
sns.countplot(df['bedrooms'], ax=axes[0], color='purple')

axes[1].set_title('Bedrooms vs. Price')
sns.boxplot(x='bedrooms', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



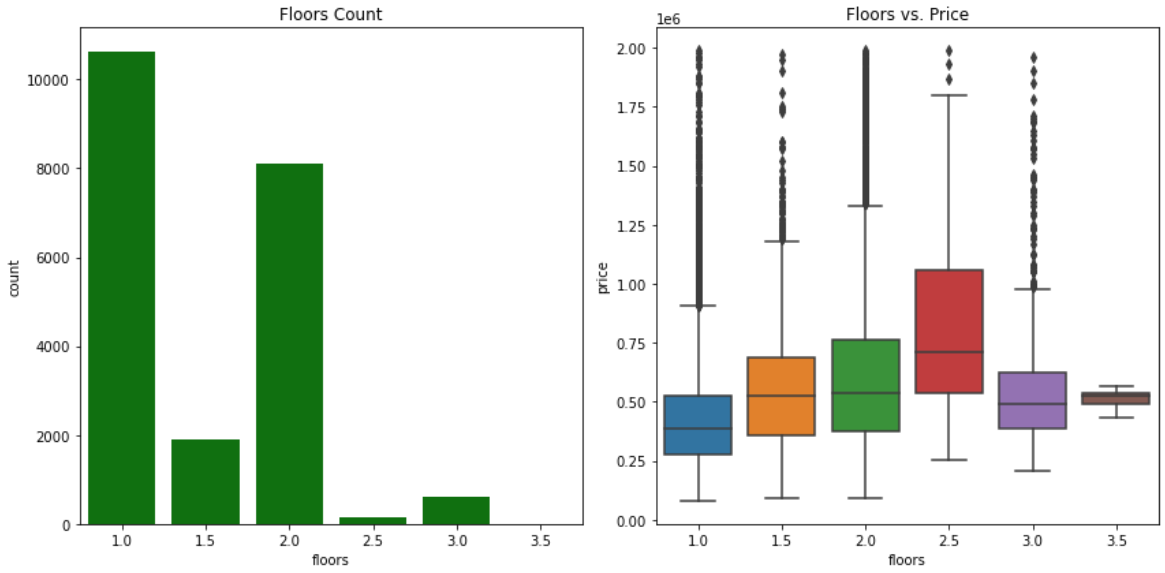
In [12]:

```
# That 33 bedroom house for 750K?  No way.  Probably error.  Let's drop it.
#less egregious, so perhaps leave it.
df = df.drop(df[df['bedrooms'] == 33].index)
```

Positive correlation here as well, though lots of outliers.

In [13]:

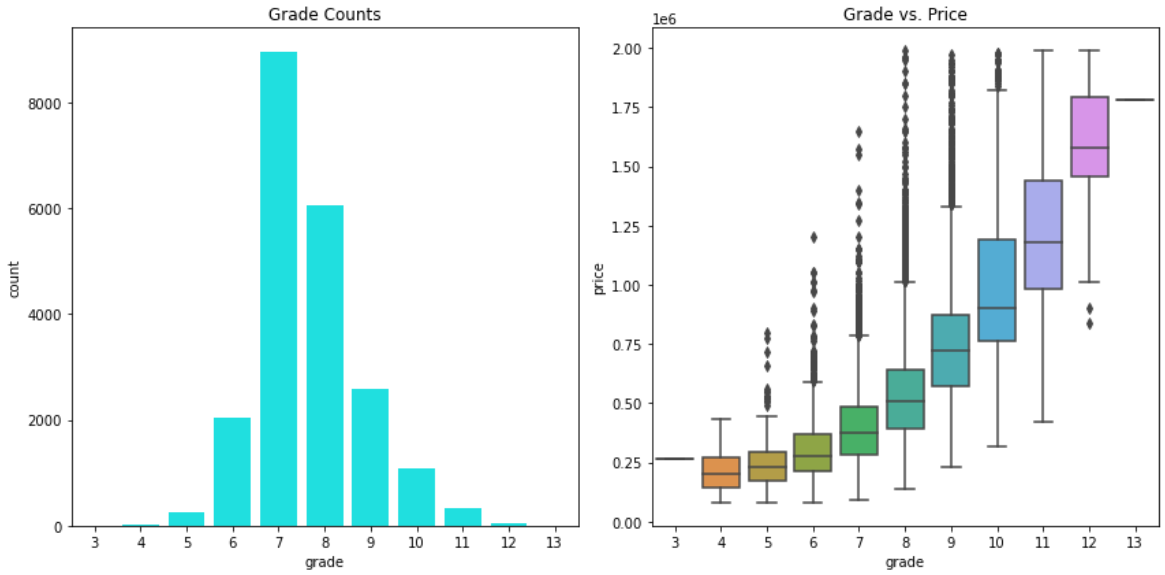
```
# floors vs price

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('Floors Count')
sns.countplot(df['floors'], ax=axes[0], color='green')

axes[1].set_title('Floors vs. Price')
sns.boxplot(x='floors', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



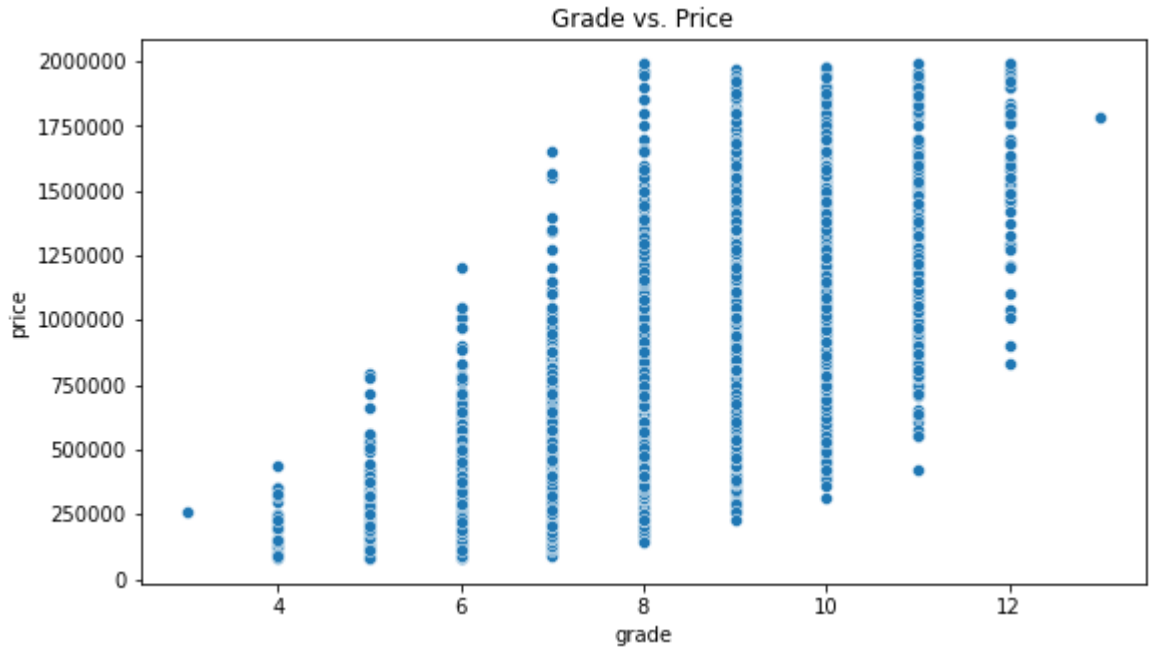
Surprisingly not an overwhelming increase on homes with 3 floors - there are other more important characteristics, it seems. May have linearity issue here.

In [14]:

```
# grade vs price

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('Grade Counts')
sns.countplot(df['grade'], ax=axes[0], color='aqua')

axes[1].set_title('Grade vs. Price')
sns.boxplot(x='grade', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



In [15]:
```
#Let's take another look with scatter
plt.figure(figsize=(9, 5))
sns.scatterplot(x='grade', y='price', data=df).set_title('Grade vs. Price')
plt.ticklabel_format(style='plain')#style='plain', 'sci', 'scientific'
```

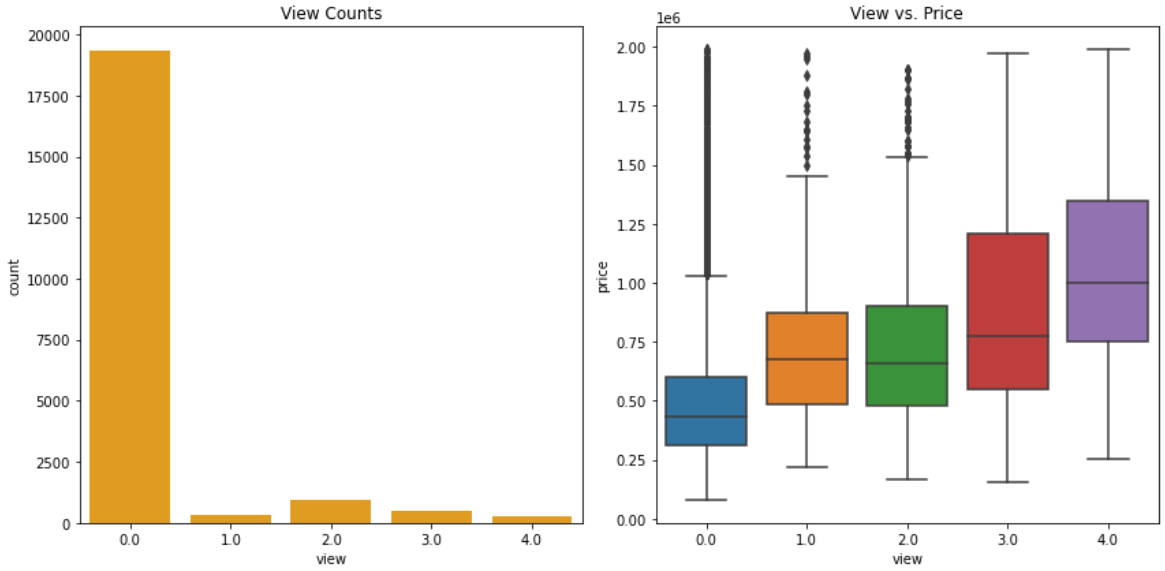


Grade may even increase quadratically - let's make a x^2 column for it. Note, this is a departure for the linearity requirement of our linear regression model - but the grade graph is so compelling, I think it's valid.

In [16]:
```
# Feature engineering!
df['grade_squared'] = df['grade']**2
```

In [17]:

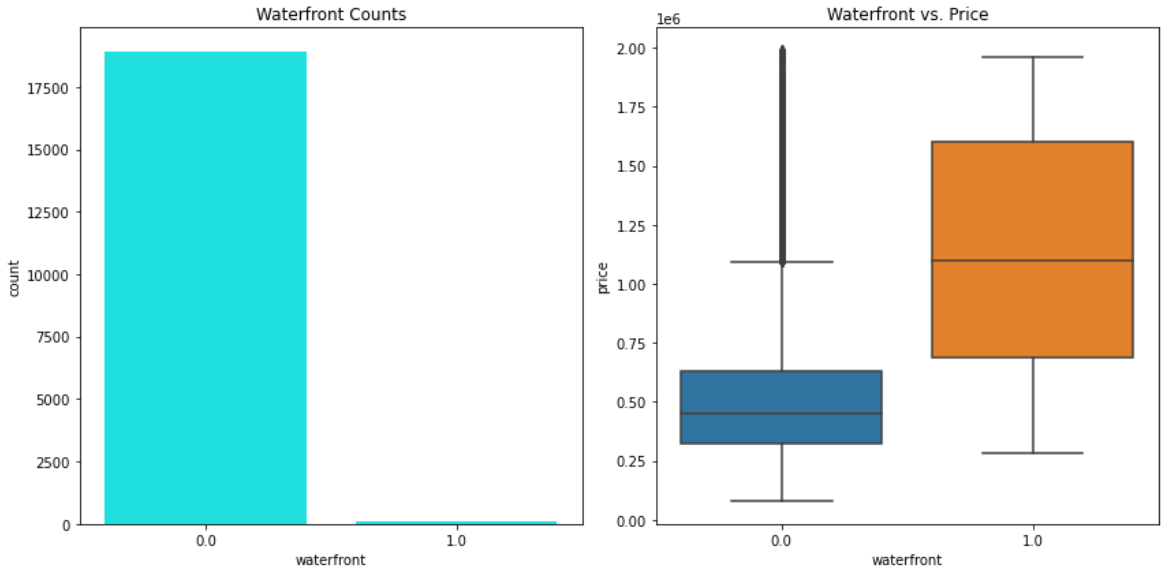
```
# View
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('View Counts')
sns.countplot(df['view'], ax=axes[0], color='orange')

axes[1].set_title('View vs. Price')
sns.boxplot(x='view', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



Documentation is unclear about what View entails. But, it seems like more is better.

In [18]:

```
# Waterfront
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('Waterfront Counts')
sns.countplot(df['waterfront'], ax=axes[0], color='cyan')

axes[1].set_title('Waterfront vs. Price')
sns.boxplot(x='waterfront', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



Homes with waterfront proximity tend to have higher prices.

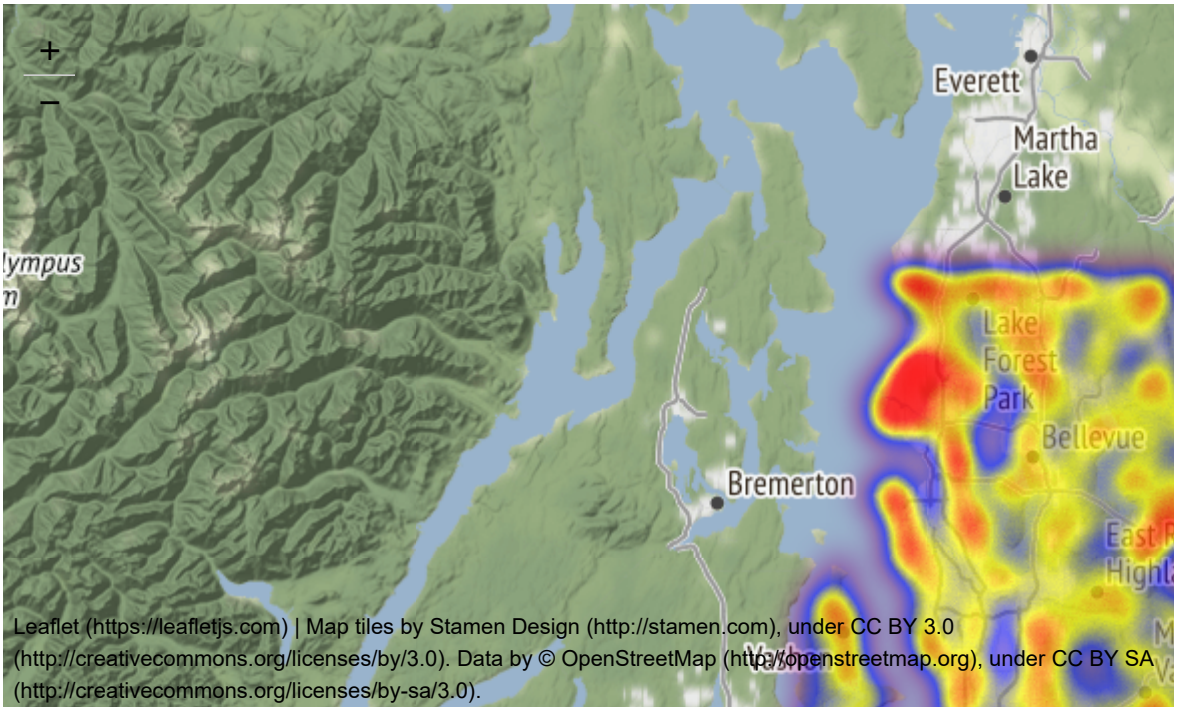
Latitude and longitude

Each of these values alone doesn't tell the story of neighborhood value - we need them together somehow.

In [19]:

```
# Neighborhood has a huge impact in house prices historically.  Though linear
# I found a nice visualization package for this.
import folium
from folium.plugins import HeatMap

price_map = folium.Map(location=[47.5,-122.3],
                       tiles = "Stamen Terrain",
                      zoom_start = 9)
data_heatmap = df[['lat','long','price']]
data_heatmap = df.dropna(axis=0, subset=['lat','long','price'])
data_heatmap = [[row['lat'],row['long']] for index, row in data_heatmap.iterr
HeatMap(data_heatmap, radius=10,
        gradient = {.35: 'purple',.55: 'blue',.68:'yellow',.78:'red'}).add_to
price_map
```

Out[19]:



Other Data Cleaning

In [20]:
```
#sqft_basement is an object.  let's look into that.
display(df.sqft_basement.value_counts())
#looks like there are "?" values here.
df[['sqft_basement', 'floors']][df.sqft_basement == '?']
df[df.sqft_basement == '?']
#Aha!  it appears that the sum of sqft_above + sqft_basement seems to equal s
df.loc[df.sqft_basement=='?','sqft_basement'] = df.sqft_living - df.sqft_abov
df.sqft_basement = df.sqft_basement.astype(float)
#df.loc[df.sqft_basement<0]
#df.loc[df.sqft_basement=='?']

#df.info()
```

```
0.0       12771
?           450
600.0       215
500.0       209
700.0       205
           ...
1880.0        1
518.0         1
1930.0        1
248.0         1
1820.0        1
Name: sqft_basement, Length: 291, dtype: int64
```

In [21]:
```
#I noticed that yr_renovated has 0 values.
#let's overwrite yr_renovated 0 values to yr_built - seems most representativ
df.loc[df.yr_renovated==0,'yr_renovated'] = df.yr_built
#df.loc[df.sqft_basement=='?','sqft_basement'] = df.sqft_living - df.sqft_abo
df.yr_renovated.value_counts()
```

Out[21]:
```
2014.0    524
2005.0    402
2006.0    391
2004.0    379
2007.0    372
          ...
1901.0     22
1902.0     20
1933.0     14
1935.0     12
1934.0     10
Name: yr_renovated, Length: 116, dtype: int64
```

In [22]:

```
#fix date/time all the way down, first.
df.date =  pd.to_datetime(df['date']) #pd.to_datetime(df.date,format= '%Y%B')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21388 entries, 0 to 21596
Data columns (total 22 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21388 non-null  int64
 1   date           21388 non-null  datetime64[ns]
 2   price          21388 non-null  float64
 3   bedrooms       21388 non-null  int64
 4   bathrooms      21388 non-null  float64
 5   sqft_living    21388 non-null  int64
 6   sqft_lot       21388 non-null  int64
 7   floors         21388 non-null  float64
 8   waterfront     19029 non-null  float64
 9   view           21326 non-null  float64
 10  condition      21388 non-null  int64
 11  grade          21388 non-null  int64
 12  sqft_above     21388 non-null  int64
 13  sqft_basement  21388 non-null  float64
```

Feature Engineering - date and day of week

In [23]:

```
#In addition, let's extract day/mo/yr and Feature Engineer ourselves a day of
#df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].apply(lambda date : date.year)
df['month'] = df['date'].apply(lambda date : date.month)
df['day'] = df['date'].apply(lambda date : date.day)

#df['day'].value_counts()
day_map = {0:'Mon', 1:'Tue', 2:'Wed', 3:'Thu', 4:'Fri', 5:'Sat', 6:'Sun'}
df['dayOfWeek'] = df['date'].apply(lambda date : date.dayofweek)
df2 = df.copy()
df2['day_word'] = df['dayOfWeek'].apply(lambda x: day_map[x])
df2['day_name'] = df['date'].apply(lambda date: date.day_name())
print(df2['dayOfWeek'].value_counts())
print(df2['day_name'].value_counts())
df2['day_word'].value_counts()
#Note for day of week, Monday is 0, Sunday is 6
```

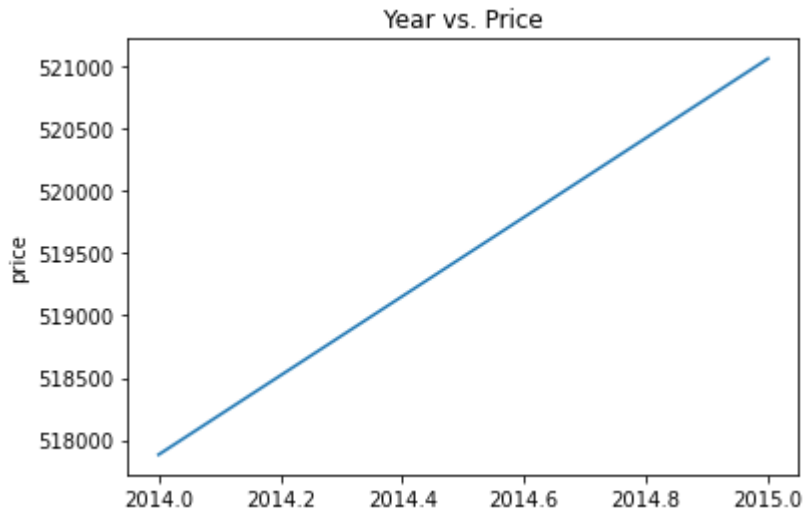
```
1    4659
2    4547
0    4060
3    3960
4    3648
5     285
6     229
Name: dayOfWeek, dtype: int64
Tuesday      4659
Wednesday    4547
Monday       4060
Thursday     3960
Friday       3648
Saturday      285
Sunday        229
Name: day_name, dtype: int64
```

Out[23]:

```
Tue    4659
Wed    4547
Mon    4060
Thu    3960
Fri    3648
Sat     285
Sun     229
Name: day_word, dtype: int64
```

In [24]:
```
#Let's inspect date relationships a little
sns.lineplot(x=df['year'].unique(), y=df.groupby('year').mean()['price']).set
```

Out[24]: Text(0.5, 1.0, 'Year vs. Price')

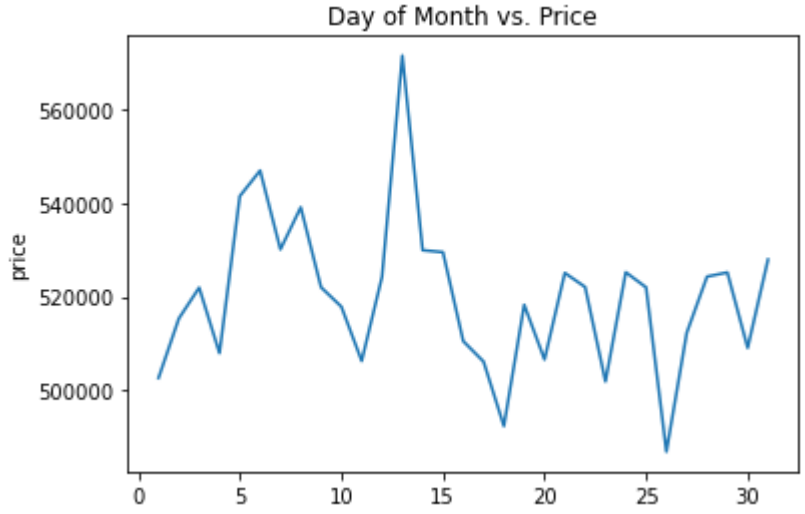


In [25]:
```
sns.lineplot(x=df['month'].unique(), y=df.groupby('month').mean()['price'], )
```

Out[25]: Text(0.5, 1.0, 'Month vs. Price')

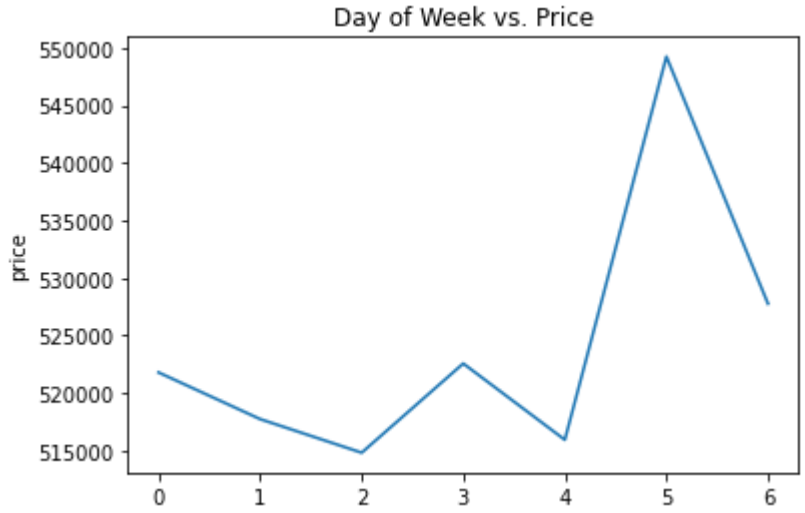
In [26]:
```
sns.lineplot(x=df['day'].unique(), y=df.groupby('day').mean()['price']).set_t
```

Out[26]: Text(0.5, 1.0, 'Day of Month vs. Price')



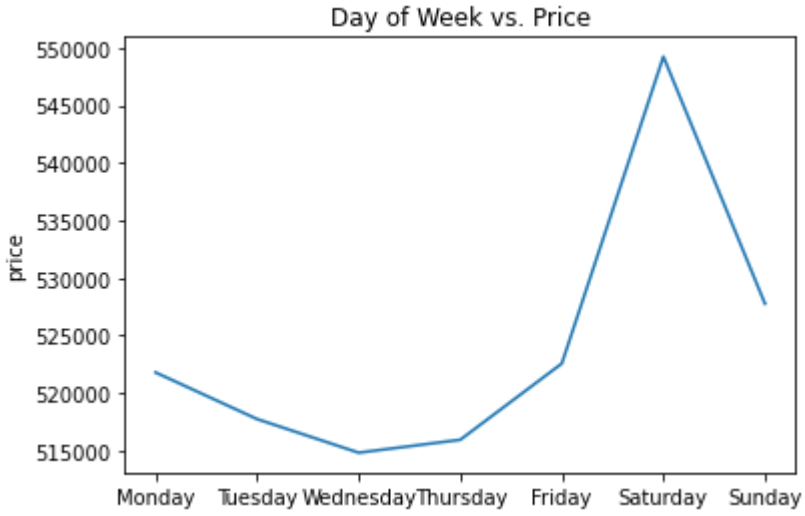
In [27]:
```
sns.lineplot(x=df['dayOfWeek'].unique(), y=df.groupby('dayOfWeek').mean()['pr
```

Out[27]: Text(0.5, 1.0, 'Day of Week vs. Price')



In [28]:
```
l = list(df2['day_name'].unique())
l[3], l[4] = l[4], l[3]
sns.lineplot(x=l, y=df.groupby('dayOfWeek').mean()['price']).set_title('Day c
```

Out[28]: Text(0.5, 1.0, 'Day of Week vs. Price')



Date summary

Prices seem to increase by year; there is some seasonality by month. Day of Month seems unpredictable. Day of week definitely has some notable pattern (especially Friday-Saturday).

In [29]:
```
if verbose:
    df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21388 entries, 0 to 21596
Data columns (total 26 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21388 non-null  int64
 1   date           21388 non-null  datetime64[ns]
 2   price          21388 non-null  float64
 3   bedrooms       21388 non-null  int64
 4   bathrooms      21388 non-null  float64
 5   sqft_living    21388 non-null  int64
 6   sqft_lot       21388 non-null  int64
 7   floors         21388 non-null  float64
 8   waterfront     19029 non-null  float64
 9   view           21326 non-null  float64
 10  condition      21388 non-null  int64
 11  grade          21388 non-null  int64
 12  sqft_above     21388 non-null  int64
 13  sqft_basement  21388 non-null  float64
 14  yr_built       21388 non-null  int64
 15  yr_renovated   17583 non-null  float64
 16  zipcode        21388 non-null  int64
 17  lat            21388 non-null  float64
 18  long           21388 non-null  float64
 19  sqft_living15  21388 non-null  int64
 20  sqft_lot15     21388 non-null  int64
 21  grade_squared  21388 non-null  int64
 22  year           21388 non-null  int64
 23  month          21388 non-null  int64
 24  day            21388 non-null  int64
 25  dayOfWeek      21388 non-null  int64
dtypes: datetime64[ns](1), float64(9), int64(16)
memory usage: 5.0 MB
```

yr_renovated

Overwriting yr_renovated = 0 entries

In [30]:
```
##We should probably remap 0 in yr_renovated to something.  it doesn't work a
###  probably replace with yr_built
df[df['yr_renovated']==0]['yr_renovated']  = df[df['yr_renovated']==0]['yr_bu

display(df[df['yr_renovated']==0]['yr_built'])
#X_train['yr_renovated']
```

```
Series([], Name: yr_built, dtype: int64)
```

Zipcode

There are too many zipcodes to properly one-hot encode them. One approach is to create a new ordinal column based on the mean price for each zipcode

In [31]:
```
zipcode_mean = df.groupby('zipcode').mean()
zipcode_mean.sort_values('price', ascending=False)
df['zipcode_avg'] = df['zipcode'].apply(lambda zipcode : zipcode_mean.loc[zip
```

Okay, that's good so far for course cleaning; let's get a little more nuanced.

Before we can clean those na columns with style, we should train/test split.

Train Test Split!!!!!

In [32]:
```
target = 'price'
X = df.drop(target, axis = 1)
y = df[target]
```

In [33]:
```
SPLIT_IS_RANDOM = False
if SPLIT_IS_RANDOM:
    random_state = randint(1,2**32 - 2)
    #print(random_state)
else:
    random_state = 14
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = rand
#X_train
```

Great, now that we have the split, we can fix/impute some fixed columns

In [34]:

```
#waterfront cleaning
display(df.waterfront.value_counts())
display(df.waterfront.isna().sum())
# https://github.com/ericthansen/dsc-sklearn-preprocessing-lab
#display(df[df['waterfront'].isna()].head())
#I don't see any corresponding column to indicate these are real values; or i
#there are also 0s.  So I'll presume these are missing, make a missing indica
#will use the median, but since this is really more of a categorical value th

col = 'waterfront'

# (1) Identify data to be transformed
# We only want missing indicators for column
col1_train = X_train[[col]]

# (2) Instantiate the transformer object
missing_indicator1 = MissingIndicator()

# (3) Fit the transformer object on col
missing_indicator1.fit(col1_train)

# (4) Transform col and assign the result
# to col_missing_train
col1_missing_train = missing_indicator1.transform(col1_train)

# Visually inspect col_missing_train
#col1_missing_train

#import numpy as np

# col_missing_train should be a NumPy array
assert type(col1_missing_train) == np.ndarray

# We should have the same number of rows as the full X_train
assert col1_missing_train.shape[0] == X_train.shape[0]

# But we should only have 1 column
assert col1_missing_train.shape[1] == 1

X_train["waterfront_Missing"] = col1_missing_train
#X_train


# (1) col1_train was created previously, so we don't
# need to extract the relevant data again

# (2) Instantiate a SimpleImputer with strategy="median"
imputer1 = SimpleImputer(strategy='median')

# (3) Fit the imputer on col_train
imputer1.fit(col1_train)

# (4) Transform frontage_train using the imputer and
# assign the result to col_imputed_train
col1_imputed_train = imputer1.transform(col1_train)
```

```
# Visually inspect col_imputed_train
display(col1_imputed_train)

# Run this cell without changes

# (5) Replace value of col
X_train["waterfront"] = col1_imputed_train

# Visually inspect X_train
# display(X_train)

display(X_train.waterfront.value_counts())
display(X_train.waterfront.isna().sum())
```

```
0.0    18928
1.0      101
Name: waterfront, dtype: int64

2359

array([[0.],
       [0.],
       [0.],
       ...,
       [0.],
       [0.],
       [0.]])

0.0    15961
1.0       80
Name: waterfront, dtype: int64

0
```

In [35]:

```
#view pre-cleaning exploration
display(X_train.view.value_counts())
display('N/A count:', X_train.view.isna().sum())
#display(df[(df.view.isna() & df.view==0)])

# Since there are values of 0 here, I'm inclined to assume that N/A doesn't j
```

```
0.0    14486
2.0      704
3.0      362
1.0      242
4.0      191
Name: view, dtype: int64
```

```
'N/A count:'
```

```
56
```

In [36]:

```
col = 'view'

# (1) Identify data to be transformed
# We only want missing indicators for column
col2_train = X_train[[col]]

# (2) Instantiate the transformer object
missing_indicator2 = MissingIndicator()

# (3) Fit the transformer object on col
missing_indicator2.fit(col2_train)

# (4) Transform col and assign the result
# to col_missing_train
col2_missing_train = missing_indicator2.transform(col2_train)

# Visually inspect col_missing_train
#col2_missing_train

#import numpy as np

# col_missing_train should be a NumPy array
assert type(col2_missing_train) == np.ndarray

# We should have the same number of rows as the full X_train
assert col2_missing_train.shape[0] == X_train.shape[0]

# But we should only have 1 column
assert col2_missing_train.shape[1] == 1

X_train["view_Missing"] = col2_missing_train
#X_train


# (1) col1_train was created previously, so we don't
# need to extract the relevant data again

# (2) Instantiate a SimpleImputer with strategy="median"
imputer2 = SimpleImputer(strategy='median')

# (3) Fit the imputer on col_train
imputer2.fit(col2_train)

# (4) Transform frontage_train using the imputer and
# assign the result to col_imputed_train
col2_imputed_train = imputer2.transform(col2_train)

# Visually inspect col_imputed_train
#display(col2_imputed_train)

# Run this cell without changes

# (5) Replace value of col
X_train[col] = col2_imputed_train

# Visually inspect X_train
```

```
# display(X_train)

display(X_train[col].value_counts())
display('N/A count:', X_train[col].isna().sum())
```

```
0.0    14542
2.0      704
3.0      362
1.0      242
4.0      191
Name: view, dtype: int64
```

```
'N/A count:'
```

```
0
```

In [37]:

```
# yr_renovated cleaning
display(X_train.yr_renovated.value_counts())
display(X_train.yr_renovated.isna().sum())
X_train[X_train.yr_renovated == 0]

col = 'yr_renovated'


# (1) Identify data to be transformed
# We only want missing indicators for column
col3_train = X_train[[col]]

# (2) Instantiate the transformer object
missing_indicator3 = MissingIndicator()

# (3) Fit the transformer object on col
missing_indicator3.fit(col3_train)

# (4) Transform col and assign the result
# to col_missing_train
col3_missing_train = missing_indicator3.transform(col3_train)

# Visually inspect col_missing_train
#col2_missing_train

#import numpy as np

# col_missing_train should be a NumPy array
assert type(col3_missing_train) == np.ndarray

# We should have the same number of rows as the full X_train
assert col3_missing_train.shape[0] == X_train.shape[0]

# But we should only have 1 column
assert col3_missing_train.shape[1] == 1

X_train["yr_renovated_Missing"] = col3_missing_train
#X_train


# (1) col1_train was created previously, so we don't
# need to extract the relevant data again

# (2) Instantiate a SimpleImputer with strategy="median"
imputer3 = SimpleImputer(strategy='median')

# (3) Fit the imputer on col_train
imputer3.fit(col3_train)

# (4) Transform frontage_train using the imputer and
# assign the result to col_imputed_train
col3_imputed_train = imputer3.transform(col3_train)

# Visually inspect col_imputed_train
#display(col2_imputed_train)
```

```
# Run this cell without changes

# (5) Replace value of col
X_train[col] = col3_imputed_train

# Visually inspect X_train
# display(X_train)

display(X_train[col].value_counts())
display('N/A count:', X_train[col].isna().sum())
```

```
2014.0    381
2006.0    292
2005.0    289
2004.0    283
2007.0    280
         ...
1902.0     14
1901.0     14
1935.0     11
1933.0     10
1934.0      6
Name: yr_renovated, Length: 116, dtype: int64
```

```
2848
```

```
1977.0    3110
2014.0     381
2006.0     292
2005.0     289
2004.0     283
          ...
1902.0      14
1901.0      14
1935.0      11
1933.0      10
1934.0       6
Name: yr_renovated, Length: 116, dtype: int64
```

```
'N/A count:'
```

```
0
```

In [38]:

```
if verbose:
    #display(X_train.value_counts())
    display('N/A count:', X_train.isna().sum())
```

```
'N/A count:'

id                      0
date                    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
sqft_lot15              0
grade_squared           0
year                    0
month                   0
day                     0
dayOfWeek               0
zipcode_avg             0
waterfront_Missing      0
view_Missing            0
yr_renovated_Missing    0
dtype: int64
```

In [39]:
```
#More checking for types - numerical or do we need categorical?  We can do a
X_train.info()
X_train.head()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16041 entries, 7688 to 19232
Data columns (total 29 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   id                    16041 non-null  int64
 1   date                  16041 non-null  datetime64[ns]
 2   bedrooms              16041 non-null  int64
 3   bathrooms             16041 non-null  float64
 4   sqft_living           16041 non-null  int64
 5   sqft_lot              16041 non-null  int64
 6   floors                16041 non-null  float64
 7   waterfront            16041 non-null  float64
 8   view                  16041 non-null  float64
 9   condition             16041 non-null  int64
 10  grade                 16041 non-null  int64
 11  sqft_above            16041 non-null  int64
 12  sqft_basement         16041 non-null  float64
 13  yr_built              16041 non-null  int64
 14  yr_renovated          16041 non-null  float64
 15  zipcode               16041 non-null  int64
 16  lat                   16041 non-null  float64
 17  long                  16041 non-null  float64
 18  sqft_living15         16041 non-null  int64
 19  sqft_lot15            16041 non-null  int64
 20  grade_squared         16041 non-null  int64
 21  year                  16041 non-null  int64
 22  month                 16041 non-null  int64
 23  day                   16041 non-null  int64
 24  dayOfWeek             16041 non-null  int64
 25  zipcode_avg           16041 non-null  float64
 26  waterfront_Missing    16041 non-null  bool
 27  view_Missing          16041 non-null  bool
 28  yr_renovated_Missing  16041 non-null  bool
dtypes: bool(3), datetime64[ns](1), float64(9), int64(16)
memory usage: 3.4 MB
```

Out[39]:

	id	date	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
7688	2781250100	2015-04-15	3	2.5	2550	5349	2.0	0.0	0.0
12456	2386000240	2014-09-29	5	3.5	3870	65556	2.0	0.0	0.0
17230	5406500440	2014-08-12	4	2.5	2780	6235	2.0	0.0	0.0
15395	1723049008	2014-08-22	2	1.0	930	8665	1.0	0.0	0.0
12903	6352600650	2015-04-02	4	2.5	3330	8897	2.0	0.0	0.0

In [40]:

```
#checking into unique values for each column
for col in X_train.columns:
    print(col)
    display(X_train[col].value_counts())

#notes:  Id has some repeats.  what's up with that? ------- let's drop it any
#bedrooms - a house with 33 bedrooms?
#bathrooms .75 up, with .25 increments
#floors is 1-3.5 by .5, categorical
#waterfront is 0/1 cat
#view is 0-4 cat
#condition 1-5 categorical
#grade 3-13 ordinals - categorical? that's a lot for one-hot
#yr_built...ordinal... probably too many for onehot
#yr_renovated, same ------------------------------------should alter the on
#zipcode categorical but 70 options...
#lat and longitude - i expect this to matter, but not in a linear way
#waterfront_missing boolean
#view_missing boolean
#yr_renovated missing boolean
```

```
id

5417600130    2
2892700041    2
6308000010    2
722039087     2
6021500970    2
             ..
2626119062    1
7436400020    1
104530240     1
2976800145    1
1777500160    1
Name: id, Length: 15950, dtype: int64

date

2015-04-27    105
2014-06-23    104
2014-06-26    104
```

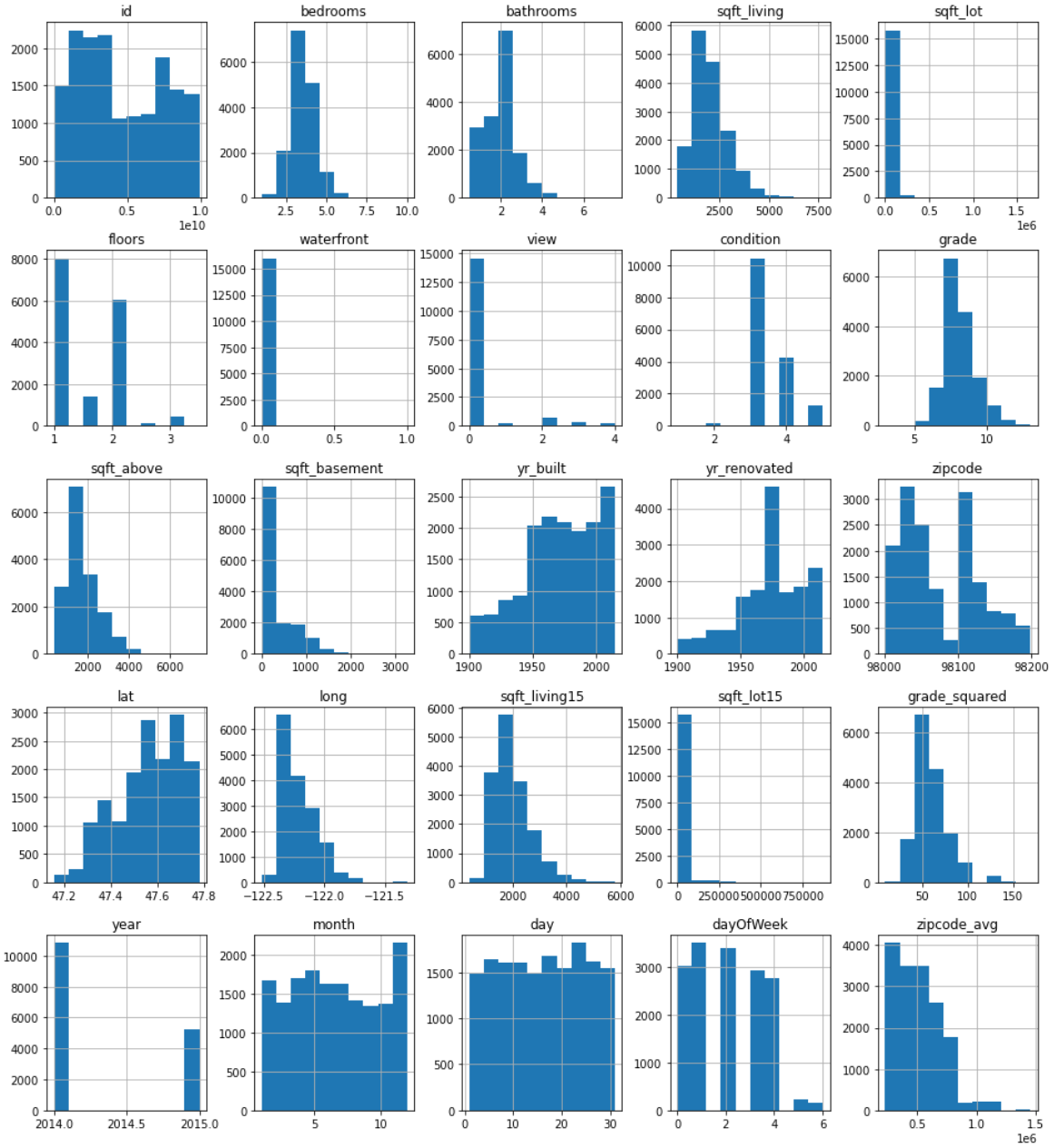
First pass at splitting numeric vs categorical

At the same time, we can inspect for normality

In [41]:

```
plt.tight_layout()
exclude = ['waterfront_Missing','view_Missing','yr_renovated_Missing']
X_hist = X_train.drop(exclude, axis=1)
if verbose:
    X_hist.hist(figsize  = [16, 18],);
```

```
<Figure size 432x288 with 0 Axes>
```



Virtually all the numerical columns exhibit some non-standardness (tails, bimodal, etc). Could log tranform them all. Also, we'll want to use standard normalization.

Column allocation

In [42]:

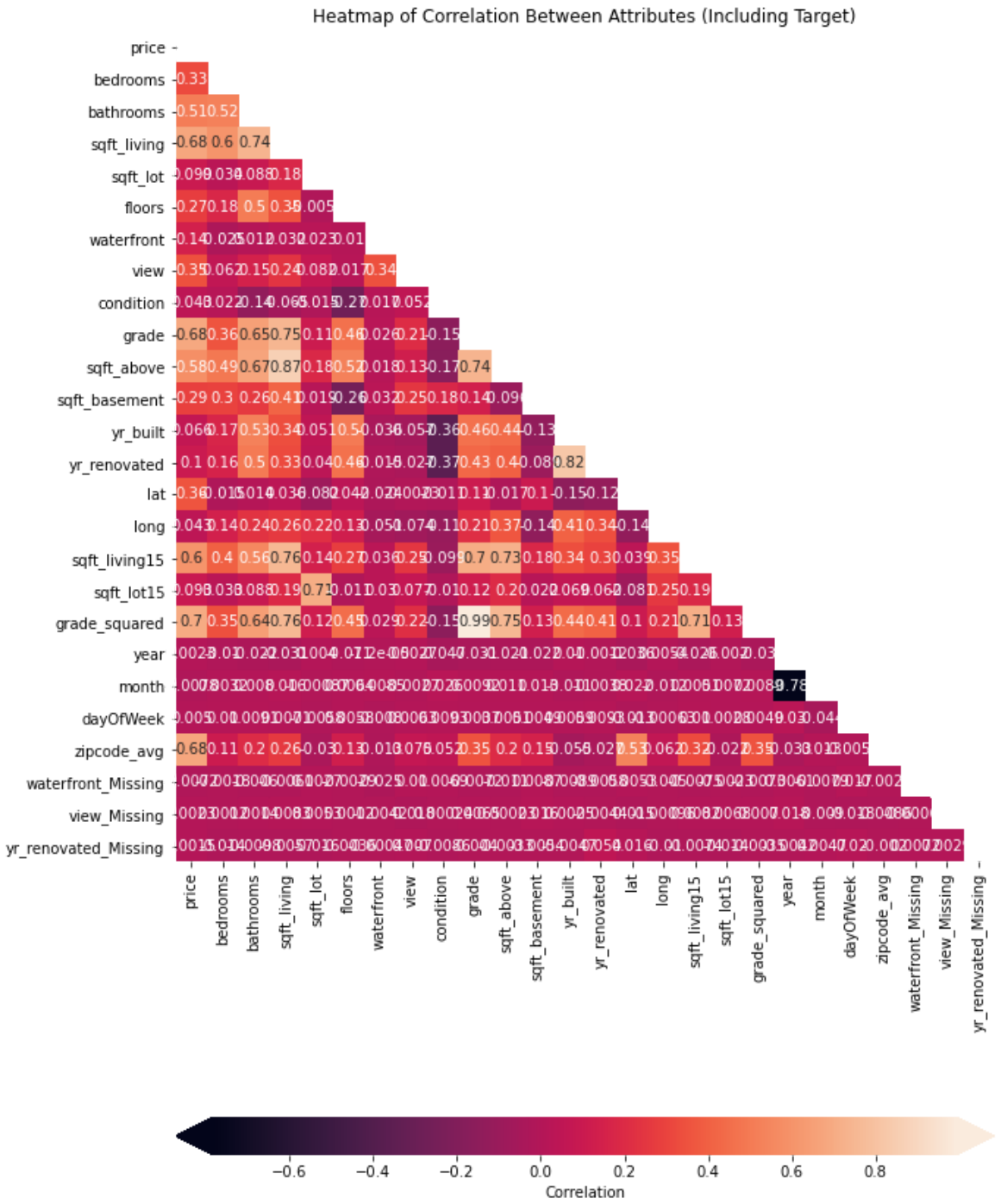
```
#I would want to only keep relevant columns.  I.e, an id number shouldn't hav
#But, who knows, if the id numbers are not assigned at random, there could be
#We will do some iterative model building.  For round 1, let's go for maximiz
dropped_cols = ['id','day', 'date', 'zipcode']
num_cols = [  'sqft_living', 'sqft_lot',
         'grade', 'grade_squared','sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated',  'lat', 'long',
       'sqft_living15', 'sqft_lot15',
            'bedrooms', 'bathrooms',
             'zipcode_avg',

       ]#'date',
#non_log_cols = ['sqft_basement, ']
#plan to move month/dayOfWeek to categorical eventually
#originally put bedrooms/bathrooms as categoricals...but moved back up to num
cat_cols = ['year', 'month', 'dayOfWeek','condition','floors', 'waterfront',
            'waterfront_Missing', 'view_Missing','yr_renovated_Missing']
#display(sorted(X_train.columns))
#display(sorted(dropped_cols+num_cols+cat_cols))
print(len(X_train.columns),'+', len(dropped_cols), len(num_cols) , len(cat_cc
#assert len(X_train.columns) == len(dropped_cols) + len(num_cols) + len(cat_c
```

```
29 + 4 15 10
```

Introductory heatmap - before any transformations or one-hot encoding

In [43]:

```
#Exploring correlation and multicollinearity in the data

#X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = ran

heatmap_data = pd.concat([y_train, X_train.drop(dropped_cols, axis=1)], axis=
# https://seaborn.pydata.org/generated/seaborn.jointplot.html
#did corr matrix
heatmap_data.corr()
#used heatmap
import seaborn as sns
import numpy as np

# Create a df with the target as the first column,
# then compute the correlation matrix
##heatmap_data = pd.concat([y_train, X_train], axis=1)
##corr = heatmap_data.corr()
corr = heatmap_data.corr()

# Set up figure and axes
fig, ax = plt.subplots(figsize=(10, 16))

# Plot a heatmap of the correlation matrix, with both
# numbers and colors indicating the correlations

sns.heatmap(
    # Specifies the data to be plotted
    data=corr,
    # The mask means we only show half the values,
    # instead of showing duplicates. It's optional.
    mask=np.triu(np.ones_like(corr, dtype=bool)),
    # Specifies that we should use the existing axes
    ax=ax,
    # Specifies that we want labels, not just colors
    annot=True,
    # Customizes colorbar appearance
    cbar_kws={"label": "Correlation", "orientation": "horizontal", "pad": .2,
)

# Customize the plot appearance
ax.set_title("Heatmap of Correlation Between Attributes (Including Target)");
```



Just from this inspection, we may start to target sqft_living15, sqft_basement, sqft_above, grade, sqft_living, grade_sq, zipcode_avg. Also note the correlation between grade and grade_squared - not surprising, probably should remove grade. Will do VIF to verify, then likely remove.

In [44]:
```
#Temporary reload point
#backup = X_train.copy()
#X_train = backup.copy()
```

One-hot Encoding of Categoricals

In [45]:

```
#Next steps - onehot encode categoricals
#from sklearn.preprocessing import OneHotEncoder
# (1) Create a variable fireplace_qu_train
# extracted from X_train
# (double brackets due to shape expected by OHE)
##fireplace_qu_train = X_train[["FireplaceQu"]]

# (2) Instantiate a OneHotEncoder with categories="auto",
# sparse=False, and handle_unknown="ignore"
##ohe = OneHotEncoder(categories='auto', sparse=False, handle_unknown='ignore

# (3) Fit the encoder on fireplace_qu_train
##ohe.fit(fireplace_qu_train)

# Inspect the categories of the fitted encoder
##ohe.categories_
cat_dict = {}
for col in cat_cols:
    col_train = X_train[[col]]
    ohe = OneHotEncoder(categories='auto', sparse=False, drop='first', handle
    ohe.fit(col_train)
    cat_dict[col] = ohe
    if verbose:
        display(col, ohe.categories_)
    # (4) Transform fireplace_qu_train using the encoder and
    # assign the result to fireplace_qu_encoded_train
    col_train_encoded = ohe.transform(col_train)

    if verbose:
        pass
        #display('featurenames:', ohe.get_feature_names([col]))


    # Visually inspect fireplace_qu_encoded_train
    #fireplace_qu_encoded_train

    col_train_encoded = pd.DataFrame(
        # Pass in NumPy array
        col_train_encoded,
        # Set the column names to the categories found by OHE
        #columns=ohe.categories_[0][1:], #old and busted; instead use this co
        columns = ohe.get_feature_names([col]),
        # Set the index to match X_train's index
        index=X_train.index
    )
    # (5b) Drop original FireplaceQu column
    X_train.drop(col, axis=1, inplace=True)

    # (5c) Concatenate the new dataframe with current X_train
    X_train = pd.concat([X_train, col_train_encoded], axis=1)

#X_train.info()
```

```
'year'

[array([2014, 2015], dtype=int64)]
```

```
'month'

[array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12], dtype=int64)]

'dayOfWeek'

[array([0, 1, 2, 3, 4, 5, 6], dtype=int64)]

'condition'

[array([1, 2, 3, 4, 5], dtype=int64)]

'floors'

[array([1. , 1.5, 2. , 2.5, 3. , 3.5])]

'waterfront'

[array([0., 1.])]

'view'

[array([0., 1., 2., 3., 4.])]

'waterfront_Missing'

[array([False,  True])]

'view_Missing'

[array([False,  True])]

'yr_renovated_Missing'

[array([False,  True])]
```

Drop dropped_cols

In [46]:

```
#date seems to be causing trouble.  drop it if it hasn't already been droppea
X_train.head()
#X_train.info()
#y_train
X_train.date.value_counts()
X_train = X_train.drop(dropped_cols, axis = 1)
#X_test = X_test.drop('date', axis = 1)
```

Prepare log and scaler functions

In [47]:

```
#Earlier, we checked numericals for desirable properties; if problematic, log
#******
#pd.plotting.scatter_matrix(X_train[num_cols], figsize=(10,12)); #This comman
#see also normalization function below
def log_col(col):
    '''input a pd series
    return a log-ed series'''
    return np.log(col)
#testing
#display(log_col(X['sqft_living']))
#display((X['sqft_living']))
```

In [48]:

```
#Here, define a streamlined scaling function using the StandardScaler from sc
def scale(col):
    '''input a pd series
    return a scaled series'''
    #from sklearn.preprocessing import StandardScaler
    #this was a pain to reshape and such.  Do it the manual way.
#     l = len(col)
#     a = np.array(col).reshape(-1,1)
#     scaler = StandardScaler()
#     scaler.fit(a)
#     trans = scaler.transform(a)
#     trans = trans.reshape(l,)
    trans = (col-np.mean(col))/np.sqrt(np.var(col))

    return pd.Series(trans)
#testing
#scaled = scale(np.array(X_train['sqft_living']).reshape(-1,1))
scaled = scale(X_train['sqft_living'])
display('scaled',scaled, scaled.shape)
display('orig',X_train.sqft_living)
```

```
'scaled'

7688     0.578587
12456    2.108049
17230    0.845084
15395   -1.298479
12903    1.482360
           ...
12132   -0.128209
19016    0.787150
9576     1.737270
13779   -1.402760
19232   -0.985635
Name: sqft_living, Length: 16041, dtype: float64

(16041,)

'orig'

7688     2550
12456    3870
17230    2780
15395     930
12903    3330
           ...
12132    1940
19016    2730
9576     3550
13779     840
19232    1200
Name: sqft_living, Length: 16041, dtype: int64
```

In [49]:
```
#Check for 0 values - don't want to log those
for col in num_cols:
    if verbose:
        if df[col].min() <= 0:
            display(col, df[col].min())

#aha, of course, sqft_basement has 0 values.  As it turns out, we'll be remov
#but for now let's just not log it.
#Also, longitude is negative...so we can't log that either. - let's take its
```

```
'sqft_basement'
```

```
0.0
```

```
'long'
```

```
-122.51899999999999
```

Apply log and scaler where applicable

In [50]:
```
non_log_cols=['sqft_basement']
X_train.long = abs(X_train.long)

to_log = True
to_scale = True

for col in num_cols:
    currcol = X_train[col]
    if to_log:
        if col not in non_log_cols:
            currcol = log_col(currcol)
    if to_scale:
        currcol = scale(currcol)
    X_train[col] = currcol
```

In [51]:
```
X_train.describe()
```

Out[51]:

	bedrooms	bathrooms	sqft_living	sqft_lot	grade	sqft_abo
count	1.604100e+04	1.604100e+04	1.604100e+04	1.604100e+04	1.604100e+04	1.604100e+
mean	9.827187e-14	-1.248860e-13	4.696497e-14	2.363019e-14	2.350018e-12	1.188162e-
std	1.000031e+00	1.000031e+00	1.000031e+00	1.000031e+00	1.000031e+00	1.000031e+
min	-4.160715e+00	-3.520416e+00	-3.843249e+00	-3.015150e+00	-6.296804e+00	-3.446658e+
25%	-2.660120e-01	-6.860452e-01	-6.778783e-01	-5.085502e-01	-5.107804e-01	-7.259823e-
50%	-2.660120e-01	3.600367e-01	2.134638e-02	-5.414580e-02	-5.107804e-01	-9.606063e-
75%	7.538530e-01	6.318621e-01	6.994596e-01	3.109389e-01	4.010782e-01	7.277341e-
max	4.002205e+00	3.466233e+00	3.390854e+00	5.888002e+00	3.716512e+00	3.636075e+

In [52]:
```
#check again for negative infinities
#Check for 0 values - don't want to log those
for col in X_train.columns:
    if verbose:
        display(col, X_train[col].max(), 'na:'+str(X_train[col].isna().sum()))
#display(X_train.bedrooms)
#aha, of course, sqft_basement has 0 values.  As it turns out, we'll be remov
#but for now let's just not log it.
```

```
'bathrooms'

3.4662327170935625

'na:0'

'sqft_living'

3.390853714682894

'na:0'

'sqft_lot'

5.888002417131413

'na:0'

'grade'


3.716511879780761
```

First pass at model fit!

This first model was a decent try - there are still some issues like multicollinearity, heteroscedasticity to sort out. But let's fit it and cross validate to see how it looks.

In [53]:
```
#Model fit!
#from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)
```

Out[53]:
```
LinearRegression()
```

In [54]:
```
#crossvalidation!
#from sklearn.model_selection import cross_val_score

cross_val_score(model, X_train, y_train, cv=5)
```

Out[54]:
```
array([0.78384469, 0.78214365, 0.77675559, 0.78463302, 0.78612281])
```

First look at fit: At first glance, these scores are pretty good.

Should definitely try different approaches, like different combinations of logging/scaling, changing up categoricals, etc.

In [55]:

```
#Preprocess Test data
#waterfront, view, yr_renovated


wf_test = X_test[["waterfront"]]
wf_missing_test = missing_indicator1.transform(wf_test)
X_test["waterfront_Missing"] = wf_missing_test
# Impute missing lot frontage values
wf_imputed_test = imputer1.transform(wf_test)
X_test["waterfront"] = wf_imputed_test

view_test = X_test[["view"]]
view_missing_test = missing_indicator2.transform(view_test)
X_test["view_Missing"] = view_missing_test
# Impute missing lot frontage values
view_imputed_test = imputer2.transform(view_test)
X_test["view"] = view_imputed_test

yr_test = X_test[["yr_renovated"]]
yr_missing_test = missing_indicator3.transform(yr_test)
X_test["yr_renovated_Missing"] = yr_missing_test
# Impute missing lot frontage values
yr_imputed_test = imputer3.transform(yr_test)
X_test["yr_renovated"] = yr_imputed_test

# Check that there are no more missing values
X_test.isna().sum()

X_test[X_test['yr_renovated']==0][col]  = X_test[X_test['yr_renovated']==0]['

for col in cat_cols:
        col_test = X_test[[col]]
        curr_ohe = cat_dict[col]
        col_encoded_test = curr_ohe.transform(col_test)
        col_encoded_test = pd.DataFrame(
            col_encoded_test,
            columns = curr_ohe.get_feature_names([col]),
            index=X_test.index
        )
        X_test.drop(col, axis=1, inplace=True)
        X_test = pd.concat([X_test, col_encoded_test], axis=1)
'''ohe_dict = {}
for col in cat_cols:
    col_train = X_test[[col]]
    ohe = OneHotEncoder(categories='auto', sparse=False, drop='first', handle
    ohe.fit(col_train)
    ohe_dict[col] = ohe
    if verbose:
        display(col, ohe.categories_)
    # (4) Transform fireplace_qu_train using the encoder and
    # assign the result to fireplace_qu_encoded_train
    col_train_encoded = ohe.transform(col_train)

    if verbose:
        pass
        #display('featurenames:', ohe.get_feature_names([col]))
```

```
    # Visually inspect fireplace_qu_encoded_train
    #fireplace_qu_encoded_train

    col_train_encoded = pd.DataFrame(
        # Pass in NumPy array
        col_train_encoded,
        # Set the column names to the categories found by OHE
        #columns=ohe.categories_[0][1:], #old and busted; instead use this co
        columns = ohe.get_feature_names([col]),
        # Set the index to match X_train's index
        index=X_test.index
    )
    # (5b) Drop original FireplaceQu column
    X_test.drop(col, axis=1, inplace=True)

    # (5c) Concatenate the new dataframe with current X_train
    X_test = pd.concat([X_test, col_train_encoded], axis=1)
    '''

#log and scale

X_test.long = abs(X_test.long)

for col in num_cols:
    currcol = X_test[col]
    if to_log:
        if col not in non_log_cols:
            currcol = log_col(currcol)
    if to_scale:
        currcol = scale(currcol)
    X_test[col] = currcol

#drop cols
X_test = X_test.drop(dropped_cols, axis = 1)
```

In [56]:
```
X_train.describe()
```

Out[56]:

	bedrooms	bathrooms	sqft_living	sqft_lot	grade	sqft_abo
count	1.604100e+04	1.604100e+04	1.604100e+04	1.604100e+04	1.604100e+04	1.604100e+
mean	9.827187e-14	-1.248860e-13	4.696497e-14	2.363019e-14	2.350018e-12	1.188162e-
std	1.000031e+00	1.000031e+00	1.000031e+00	1.000031e+00	1.000031e+00	1.000031e+
min	-4.160715e+00	-3.520416e+00	-3.843249e+00	-3.015150e+00	-6.296804e+00	-3.446658e+
25%	-2.660120e-01	-6.860452e-01	-6.778783e-01	-5.085502e-01	-5.107804e-01	-7.259823e-
50%	-2.660120e-01	3.600367e-01	2.134638e-02	-5.414580e-02	-5.107804e-01	-9.606063e-
75%	7.538530e-01	6.318621e-01	6.994596e-01	3.109389e-01	4.010782e-01	7.277341e-
max	4.002205e+00	3.466233e+00	3.390854e+00	5.888002e+00	3.716512e+00	3.636075e+

In [57]:
```
X_train.describe()-X_test.describe()
```

Out[57]:

	bedrooms	bathrooms	sqft_living	sqft_lot	grade	sqft_above
count	1.069400e+04	1.069400e+04	1.069400e+04	1.069400e+04	1.069400e+04	1.069400e+04
mean	9.242976e-14	-8.307256e-14	-3.444010e-15	-2.337778e-14	2.667415e-12	6.667378e-14
std	-6.235191e-05	-6.235191e-05	-6.235191e-05	-6.235191e-05	-6.235191e-05	-6.235191e-05
min	2.494779e-02	-1.067558e+00	1.162144e-01	-4.731841e-02	-1.880994e+00	6.301672e-02
25%	2.176506e-02	-4.960520e-03	2.292150e-02	1.130991e-02	2.078709e-02	4.371657e-03
50%	2.176506e-02	4.700789e-03	9.290338e-03	6.355849e-04	2.078709e-02	-9.816417e-03
75%	2.093164e-02	7.211289e-03	-1.730168e-02	-6.323685e-03	5.816207e-03	6.937661e-03
max	-3.198842e-01	7.125599e-01	7.297145e-03	8.046624e-01	5.069527e-01	2.763151e-01

In [58]:
```
X_test.describe()
```

Out[58]:

	bedrooms	bathrooms	sqft_living	sqft_lot	grade	sqft_abov
count	5.347000e+03	5.347000e+03	5.347000e+03	5.347000e+03	5.347000e+03	5.347000e+(
mean	5.842115e-15	-4.181340e-14	5.040898e-14	4.700797e-14	-3.173965e-13	5.214244e-1
std	1.000094e+00	1.000094e+00	1.000094e+00	1.000094e+00	1.000094e+00	1.000094e+(
min	-4.185662e+00	-2.452857e+00	-3.959463e+00	-2.967831e+00	-4.415811e+00	-3.509675e+(
25%	-2.877770e-01	-6.810846e-01	-7.007998e-01	-5.198601e-01	-5.315675e-01	-7.303540e-(
50%	-2.877770e-01	3.553359e-01	1.205605e-02	-5.478138e-02	-5.315675e-01	-8.624422e-(
75%	7.329213e-01	6.246508e-01	7.167613e-01	3.172626e-01	3.952620e-01	7.207965e-(
max	4.322090e+00	2.753673e+00	3.383557e+00	5.083340e+00	3.209559e+00	3.359760e+(

Checking shapes of test/train sets for consistency

In [59]:

```
if verbose:
    display(X_train.shape)
    display(y_train.shape)

    display(X_test.shape)
    display(y_test.shape)
    display(X_train.columns)
    display(X_test.columns)
```

```
(16041, 50)

(16041,)

(5347, 50)

(5347,)

Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'grade',
       'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'lat',
       'long', 'sqft_living15', 'sqft_lot15', 'grade_squared', 'zipcode_av
g',
       'year_2015', 'month_2', 'month_3', 'month_4', 'month_5', 'month_6',
       'month_7', 'month_8', 'month_9', 'month_10', 'month_11', 'month_12',
       'dayOfWeek_1', 'dayOfWeek_2', 'dayOfWeek_3', 'dayOfWeek_4',
       'dayOfWeek_5', 'dayOfWeek_6', 'condition_2', 'condition_3',
       'condition_4', 'condition_5', 'floors_1.5', 'floors_2.0', 'floors_2.
5',
       'floors_3.0', 'floors_3.5', 'waterfront_1.0', 'view_1.0', 'view_2.
0',
       'view_3.0', 'view_4.0', 'waterfront_Missing_True', 'view_Missing_Tru
e',
       'yr_renovated_Missing_True'],
      dtype='object')

Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'grade',
       'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'lat',
       'long', 'sqft_living15', 'sqft_lot15', 'grade_squared', 'zipcode_av
g',
       'year_2015', 'month_2', 'month_3', 'month_4', 'month_5', 'month_6',
       'month_7', 'month_8', 'month_9', 'month_10', 'month_11', 'month_12',
       'dayOfWeek_1', 'dayOfWeek_2', 'dayOfWeek_3', 'dayOfWeek_4',
       'dayOfWeek_5', 'dayOfWeek_6', 'condition_2', 'condition_3',
       'condition_4', 'condition_5', 'floors_1.5', 'floors_2.0', 'floors_2.
5',
       'floors_3.0', 'floors_3.5', 'waterfront_1.0', 'view_1.0', 'view_2.
0',
       'view_3.0', 'view_4.0', 'waterfront_Missing_True', 'view_Missing_Tru
e',
       'yr_renovated_Missing_True'],
      dtype='object')
```

In [60]:
```
#model test
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

Out[60]:
```
0.7820187911527865
```

In [61]:
```
import sklearn.metrics as metrics
#from sklearn import metrics
def regression_results(y_true, y_pred):

    # Regression metrics
    explained_variance=metrics.explained_variance_score(y_true, y_pred)
    mean_absolute_error=metrics.mean_absolute_error(y_true, y_pred)
    mse=metrics.mean_squared_error(y_true, y_pred)
    #mean_squared_log_error=metrics.mean_squared_log_error(y_true, y_pred)
    median_absolute_error=metrics.median_absolute_error(y_true, y_pred)
    r2=metrics.r2_score(y_true, y_pred)

    print('explained_variance: ', round(explained_variance,4))
    #print('mean_squared_log_error: ', round(mean_squared_log_error,4))
    print('r2: ', round(r2,4))
    print('MAE: ', round(mean_absolute_error,4))
    print('MSE: ', round(mse,4))
    print('RMSE: ', round(np.sqrt(mse),4))
```

In [62]:

```
y_pred = model.predict(X_test)
regression_results(y_test, y_pred)



###########
# Make predictions
expected = y_test
predicted = model.predict(X_test)

# Summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print (model.intercept_, model.coef_, mse)
print(model.score(X_test, y_test))

#This summary isn't great, so let's use statsmodels summary instead
#from statsmodels.api import OLS
#results = OLS(y_test,X_test).fit()#.summary()

#first, necessary to clean column names bc OLS gets confused by decimal point
for col in X_train.columns:
    X_train.rename(columns = {col:col.replace(".", "Point")}, inplace = True)
    X_test.rename(columns = {col:col.replace(".", "Point")}, inplace = True)


#import statsmodels.api as sm
#from statsmodels.formula.api import ols
data_ols = pd.concat([X_train, y_train], axis=1)
outcome = target
predictors = data_ols.drop(target, axis=1)
pred_sum = '+'.join(predictors.columns)
formula = outcome + '~' + pred_sum
model = ols(formula=formula, data=data_ols).fit()
model.summary()
#to avoid making the summation string,  use the ols function
'''The advantage is that you don't have to create the summation string. Impor
#import statsmodels.api as sm
predictors_int = sm.add_constant(predictors)
#model = sm.OLS(data[target],predictors_int).fit()
display(model.summary())
results = model
```

```
explained_variance:  0.7821
r2:  0.782
MAE:  90455.0771
MSE:  17814786296.8965
RMSE:  133472.0431
440712.4727957136 [-1.21172693e+04  7.89598375e+03 -5.44615106e+04  2.0
1899743e+04
  7.97351101e+17  1.45048773e+05  6.52785841e+04 -4.20017687e+04
  8.32688400e+03  3.58400000e+03  3.33440000e+04  1.00800000e+04
 -8.30400000e+03 -7.97351101e+17  1.33504000e+05  3.45580000e+04
  4.66431250e+03  1.88080000e+04  2.72720000e+04  2.78477500e+04
  2.37470000e+04  2.49120000e+04  2.17000000e+04  1.52360000e+04
```

```
 3.42480000e+04  2.66120000e+04  2.57160000e+04 -1.17000000e+03
-1.03250000e+03  4.03200000e+03  3.81800000e+03  2.77167500e+04
-9.34950000e+03  2.23970000e+04  2.54560000e+04  3.00360000e+04
 7.94000000e+04 -9.98400000e+03  5.07200000e+03  7.62710000e+04
-4.20000000e+03 -6.84538750e+04  2.30312000e+05  5.27200000e+04
 6.52960000e+04  1.65312000e+05  2.62592000e+05 -5.51900000e+03
 1.31095000e+04 -1.30000000e+03] 17814786296.89646
```

In [63]:

```
#those missing value columns were handled manually.  Let's put that process i
def handle_missing_train_col(col, handler_dict, X_train):
    #col = 'waterfront'

    # (1) Identify data to be transformed
    # We only want missing indicators for column
    col_train = X_train[[col]]

    # (2) Instantiate the transformer object
    missing_indicator = MissingIndicator()

    # (3) Fit the transformer object on col
    missing_indicator.fit(col_train)

    # (4) Transform col and assign the result
    # to col_missing_train
    col_missing_train = missing_indicator.transform(col_train)

    # col_missing_train should be a NumPy array
    assert type(col_missing_train) == np.ndarray

    # We should have the same number of rows as the full X_train
    assert col_missing_train.shape[0] == X_train.shape[0]

    # But we should only have 1 column
    assert col_missing_train.shape[1] == 1

    X_train[col+"_Missing"] = col_missing_train
    #X_train

    # (2) Instantiate a SimpleImputer with strategy="median"
    imputer = SimpleImputer(strategy='median')

    # (3) Fit the imputer on col_train
    imputer.fit(col_train)

    # (4) Transform frontage_train using the imputer and
    # assign the result to col_imputed_train
    col_imputed_train = imputer.transform(col_train)

    # Visually inspect col_imputed_train
    #display(col_imputed_train)

    # (5) Replace value of col
    X_train[col] = col_imputed_train

#    display(X_train.waterfront.value_counts())
#    display(X_train.waterfront.isna().sum())
    hd_entry = [missing_indicator, imputer]
    handler_dict[col] = hd_entry
    return handler_dict, X_train
```

```
def handle_missing_test_col(col, handler_dict, X_test):
    mi = handler_dict[col][0]
    imp = handler_dict[col][1]

    wf_test = X_test[[col]]
    wf_missing_test = mi.transform(wf_test)
    X_test[col+"_Missing"] = wf_missing_test
    # Impute missing lot frontage values
    wf_imputed_test = imp.transform(wf_test)
    X_test[col] = wf_imputed_test

    return X_test

def handle_categorical_train_col():
    '''input categorical column, ohe_dict, X_train set,
    output updated X_train set, ohe_dict'''
    return None

###this is useful, but I ended up doing it all within the process function be
# def handle_categorical_test_col(col, ohe_dict, X_test):
#     '''input a column name, the dictionary of ohes, and the X_test set
#     output the updated X_test'''
#     c_test = X_test[[col]]
#     ohe = ohe_dict[col]
#     c_enc_test = ohe.transform(c_test)
#     c_enc_test = = pd.DataFrame(
#         c_end_test,
#         columns = ohe.get_feature_names([col]),
#         index=X_test.index
#     )
#     X_test.drop(col, axis=1, inplace=True)
#     X_test = pd.concat([X_test, c_enc_test], axis=1)
#     return X_test
```

VIF

Let's setup the Variance Inflation Factor investigation.

In [64]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor
X = X_test#df[x_cols]
x_cols = X_test.columns
vif = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
list(zip(x_cols, vif)) #variables with vif of 5 or greater display multicolli
```

Out[64]:
```
[('bedrooms', 1.8682394117519536),
 ('bathrooms', 3.3775310246827908),
 ('sqft_living', 38.49770881243451),
 ('sqft_lot', 7.116782506203766),
 ('grade', inf),
 ('sqft_above', 31.497959011803264),
 ('sqft_basement', 9.88606537513876),
 ('yr_built', 4.498236154227225),
 ('yr_renovated', 3.4358102368793477),
 ('lat', 1.8935446215366047),
 ('long', 1.7588925152042543),
 ('sqft_living15', 2.8877112884421416),
 ('sqft_lot15', 6.682819882473186),
 ('grade_squared', inf),
 ('zipcode_avg', 2.277024180930188),
 ('year_2015', 13.877097523257408),
 ('month_2', 2.216577182102263),
 ('month_3', 2.9135050188319576),
 ('month_4', 3.099069558832152),
 ('month_5', 5.815156317705912),
 ('month_6', 7.416197257584875),
 ('month_7', 7.882092163423542),
 ('month_8', 6.989723226766014),
 ('month_9', 6.013514064869438),
 ('month_10', 6.916556736932073),
 ('month_11', 4.854292233075201),
 ('month_12', 5.53619960922814),
 ('dayOfWeek_1', 2.1265589260677857),
 ('dayOfWeek_2', 2.121043582473731),
 ('dayOfWeek_3', 2.013874763093475),
 ('dayOfWeek_4', 1.8736244608426358),
 ('dayOfWeek_5', 1.0710950123914686),
 ('dayOfWeek_6', 1.0655556534989823),
 ('condition_2', 1.386956052598655),
 ('condition_3', 47.19331286406095),
 ('condition_4', 19.344221791146694),
 ('condition_5', 6.524896519053788),
 ('floors_1Point5', 1.505412600377967),
 ('floors_2Point0', 4.552148767035201),
 ('floors_2Point5', 1.1245003891757834),
 ('floors_3Point0', 1.5057916549089743),
 ('floors_3Point5', 1.014453222739764),
 ('waterfront_1Point0', 1.3296178611165383),
 ('view_1Point0', 1.0540565984326438),
 ('view_2Point0', 1.1265399663499618),
 ('view_3Point0', 1.1205796935283519),
 ('view_4Point0', 1.397338635031401),
 ('waterfront_Missing_True', 1.1295809069196145),
```

```
('view_Missing_True', 1.0123832245627502),
('yr_renovated_Missing_True', 1.2474118838226689)]
```

Surprisingly, only a few have a VIF over 10. (Yes, 5 is one rule-of-thumb cutoff but I'm taking care of the big offenders first).
Unsurprisingly, grade and grade_squared are among the culprits. Let's see if dropping grade would fix this. Also, sqft_living, sqft_above, and sqft_basement have very high values. We should drop one or two of those.

year_2015 is weirdly high.

Finally, condition 3 and condition 4 are very high. May need to address those.

In [65]:
```
xtraintemp = X_train.copy()
xtesttemp = X_test.copy()
xtraintemp = xtraintemp.drop('grade', axis=1)
xtesttemp = xtesttemp.drop('grade', axis=1)
model2 = LinearRegression()
model2.fit(xtraintemp, y_train)
display(model2.score(xtesttemp, y_test))

# X = xtesttemp#df[x_cols]
# x_cols = xtesttemp.columns
# vif = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
# list(zip(x_cols, vif)) #variables with vif of 5 or greater display multicol
```

```
0.7820177127019501
```

In [66]:
```
xtraintemp = X_train.copy()
xtesttemp = X_test.copy()
xtraintemp = xtraintemp.drop('grade_squared', axis=1)
xtesttemp = xtesttemp.drop('grade_squared', axis=1)
model2b = LinearRegression()
model2b.fit(xtraintemp, y_train)
display(model2b.score(xtesttemp, y_test))

# X = xtesttemp#df[x_cols]
# x_cols = xtesttemp.columns
# vif = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
# list(zip(x_cols, vif)) #variables with vif of 5 or greater display multicol
```

```
0.7820177127019501
```

I've run this model with different approaches; in most of them, it appears that grade_squared is giving us the better results. Let's keep it.

In [67]:
```
#First, drop sqft_basement
xtraintemp = xtraintemp.drop('sqft_basement', axis=1)
xtesttemp = xtesttemp.drop('sqft_basement', axis=1)
X = xtesttemp
x_cols = xtesttemp.columns
vif = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
list(zip(x_cols, vif)) #variables with vif of 5 or greater display multicolli
```

Out[67]:
```
[('bedrooms', 1.8682092200913765),
 ('bathrooms', 3.3774575790965344),
 ('sqft_living', 7.85408550336147),
 ('sqft_lot', 7.116154648574768),
 ('grade', 3.2410509957720355),
 ('sqft_above', 7.040267219147559),
 ('yr_built', 4.49642804022899),
 ('yr_renovated', 3.434769343041934),
 ('lat', 1.8912600621108995),
 ('long', 1.757397995987772),
 ('sqft_living15', 2.884310304873881),
 ('sqft_lot15', 6.682279439883683),
 ('zipcode_avg', 2.2697945433457387),
 ('year_2015', 13.876012353666344),
 ('month_2', 2.216200876032463),
 ('month_3', 2.912881908540962),
 ('month_4', 3.0987424925656266),
 ('month_5', 5.814135126694981),
 ('month_6', 7.415627836065108),
 ('month_7', 7.881929148661041),
 ('month_8', 6.989690711110525),
 ('month_9', 6.013195388552327),
 ('month_10', 6.915861129343817),
 ('month_11', 4.85379607707307),
 ('month_12', 5.5361152142778085),
 ('dayOfWeek_1', 2.1265105573357537),
 ('dayOfWeek_2', 2.1207867034696),
 ('dayOfWeek_3', 2.0138745314338187),
 ('dayOfWeek_4', 1.8734719211690976),
 ('dayOfWeek_5', 1.0707452991397401),
 ('dayOfWeek_6', 1.0650798798549197),
 ('condition_2', 1.386347196654371),
 ('condition_3', 47.1867757012632),
 ('condition_4', 19.34106007628182),
 ('condition_5', 6.522181854932984),
 ('floors_1Point5', 1.505363083051446),
 ('floors_2Point0', 4.4884819327015455),
 ('floors_2Point5', 1.123484910139548),
 ('floors_3Point0', 1.505085581950366),
 ('floors_3Point5', 1.0144335636501596),
 ('waterfront_1Point0', 1.3265969541241787),
 ('view_1Point0', 1.0467487606243677),
 ('view_2Point0', 1.1102557426454938),
 ('view_3Point0', 1.0931443616449306),
 ('view_4Point0', 1.365230700750758),
 ('waterfront_Missing_True', 1.1295260183094553),
```

```
('view_Missing_True', 1.012307349717223),
('yr_renovated_Missing_True', 1.2473376221048398)]
```

In [68]:
```
# VIF for sqft_living and sqft_above are still > 5.  Let's drop sqft_above as

xtraintemp = xtraintemp.drop('sqft_above', axis=1)
xtesttemp = xtesttemp.drop('sqft_above', axis=1)
X = xtesttemp
x_cols = xtesttemp.columns
vif = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
list(zip(x_cols, vif)) #variables with vif of 5 or greater display multicolli
```

Out[68]:
```
[('bedrooms', 1.866956321377717),
 ('bathrooms', 3.3066748994718678),
 ('sqft_living', 5.341106563651732),
 ('sqft_lot', 7.032096178456876),
 ('grade', 3.1477570901978256),
 ('yr_built', 4.49164654837734),
 ('yr_renovated', 3.434294612819535),
 ('lat', 1.8853315255973848),
 ('long', 1.718481898939104),
 ('sqft_living15', 2.8486094167831983),
 ('sqft_lot15', 6.682027373820476),
 ('zipcode_avg', 2.2670059708884476),
 ('year_2015', 13.874777094373025),
 ('month_2', 2.2153984136290363),
 ('month_3', 2.912868958495572),
 ('month_4', 3.097809933669276),
 ('month_5', 5.81338999649374),
 ('month_6', 7.415421848130023),
 ('month_7', 7.881921094086506),
 ('month_8', 6.9896899195810605),
 ('month_9', 6.012863240128575),
 ('month_10', 6.915838768289588),
 ('month_11', 4.853787628888569),
 ('month_12', 5.536104092797703),
 ('dayOfWeek_1', 2.1260055125184656),
 ('dayOfWeek_2', 2.120783039311522),
 ('dayOfWeek_3', 2.0135839828749784),
 ('dayOfWeek_4', 1.872870857061772),
 ('dayOfWeek_5', 1.0705045637538506),
 ('dayOfWeek_6', 1.0649962116754148),
 ('condition_2', 1.383483850887202),
 ('condition_3', 46.71894678913694),
 ('condition_4', 19.145707301497097),
 ('condition_5', 6.4438695034615545),
 ('floors_1Point5', 1.3858794023417034),
 ('floors_2Point0', 3.2789764391304246),
 ('floors_2Point5', 1.0760191125270966),
 ('floors_3Point0', 1.3725940240019447),
 ('floors_3Point5', 1.01261308204752),
 ('waterfront_1Point0', 1.326152504014816),
 ('view_1Point0', 1.0455156174397775),
 ('view_2Point0', 1.1084375608307921),
 ('view_3Point0', 1.0904424549002356),
 ('view_4Point0', 1.3619450043498689),
 ('waterfront_Missing_True', 1.1294639426380622),
 ('view_Missing_True', 1.0122007277198728),
 ('yr_renovated_Missing_True', 1.2473137613135388)]
```

It's not perfect, but now the only real offenders are condition3 and 4... perhaps can address that in future.

In [69]:
```
X_train.drop(['grade', 'sqft_basement', 'sqft_above'], axis=1, inplace=True)
#X_train.columns
X_test.drop(['grade', 'sqft_basement', 'sqft_above'], axis=1, inplace=True)
display(X_train.columns)
display(X_test.columns)
```

```
Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'yr_built',
       'yr_renovated', 'lat', 'long', 'sqft_living15', 'sqft_lot15',
       'grade_squared', 'zipcode_avg', 'year_2015', 'month_2', 'month_3',
       'month_4', 'month_5', 'month_6', 'month_7', 'month_8', 'month_9',
       'month_10', 'month_11', 'month_12', 'dayOfWeek_1', 'dayOfWeek_2',
       'dayOfWeek_3', 'dayOfWeek_4', 'dayOfWeek_5', 'dayOfWeek_6',
       'condition_2', 'condition_3', 'condition_4', 'condition_5',
       'floors_1Point5', 'floors_2Point0', 'floors_2Point5', 'floors_3Point
0',
       'floors_3Point5', 'waterfront_1Point0', 'view_1Point0', 'view_2Point
0',
       'view_3Point0', 'view_4Point0', 'waterfront_Missing_True',
       'view_Missing_True', 'yr_renovated_Missing_True'],
      dtype='object')

Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'yr_built',
       'yr_renovated', 'lat', 'long', 'sqft_living15', 'sqft_lot15',
       'grade_squared', 'zipcode_avg', 'year_2015', 'month_2', 'month_3',
       'month_4', 'month_5', 'month_6', 'month_7', 'month_8', 'month_9',
       'month_10', 'month_11', 'month_12', 'dayOfWeek_1', 'dayOfWeek_2',
       'dayOfWeek_3', 'dayOfWeek_4', 'dayOfWeek_5', 'dayOfWeek_6',
       'condition_2', 'condition_3', 'condition_4', 'condition_5',
       'floors_1Point5', 'floors_2Point0', 'floors_2Point5', 'floors_3Point
0',
       'floors_3Point5', 'waterfront_1Point0', 'view_1Point0', 'view_2Point
0',
       'view_3Point0', 'view_4Point0', 'waterfront_Missing_True',
       'view_Missing_True', 'yr_renovated_Missing_True'],
      dtype='object')
```

Recursive Feature Elimination function

In [70]:

```
#Recursive feature elim, just to see what it comes up with
'''RFE, input number of features, training set, target variable
    outputs report on RFE'''
#X_train, X_test, y_train, y_test = train_test_split(Xrfe, yrfe, random_state

# from sklearn.feature_selection import RFE
# from sklearn.linear_model import LinearRegression

def rfe( X_train, y_train, target, num_features=5):
    print("Recursive Feature Elimination")
    data_ols = pd.concat([X_train, y_train], axis=1)
    outcome = target
    predictors = data_ols.drop(target, axis=1)
    pred_sum = '+'.join(predictors.columns)
    formula = outcome + '~' + pred_sum
    #model = ols(formula=formula, data=data_ols).fit()
    #model.summary()
    linreg = LinearRegression()
    selector = RFE(linreg, n_features_to_select=num_features)
    selector = selector.fit(predictors, data_ols[outcome])

    #this tells you which variables are selected
    display(selector.support_ )
    display(data_ols.columns)
    display(np.array(data_ols.drop(outcome, axis=1).columns) * np.array(selec
    #and the ranking
    display(selector.ranking_)
    print('\nSCORE:',selector.score(X_train, y_train))

    #you can get access to the parameter estimates
    estimators = selector.estimator_
    print(estimators.coef_)
    print(estimators.intercept_)
```

In [71]:

```
# model.fit(X_train, y_train)
# model.score(X_test, y_test)
rfe(X_train, y_train, target)
```

```
Recursive Feature Elimination

array([False, False,  True, False, False, False, False, False, False,
       False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False,  True, False, False,  True,  True, False,
       False, False])

Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'yr_built',
       'yr_renovated', 'lat', 'long', 'sqft_living15', 'sqft_lot15',
       'grade_squared', 'zipcode_avg', 'year_2015', 'month_2', 'month_3',
       'month_4', 'month_5', 'month_6', 'month_7', 'month_8', 'month_9',
       'month_10', 'month_11', 'month_12', 'dayOfWeek_1', 'dayOfWeek_2',
       'dayOfWeek_3', 'dayOfWeek_4', 'dayOfWeek_5', 'dayOfWeek_6',
       'condition_2', 'condition_3', 'condition_4', 'condition_5',
       'floors_1Point5', 'floors_2Point0', 'floors_2Point5', 'floors_3Point
0',
       'floors_3Point5', 'waterfront_1Point0', 'view_1Point0', 'view_2Point
0',
       'view_3Point0', 'view_4Point0', 'waterfront_Missing_True',
       'view_Missing_True', 'yr_renovated_Missing_True', 'price'],
      dtype='object')

array(['', '', 'sqft_living', '', '', '', '', '', '', '', '',
       'zipcode_avg', '', '', '', '', '', '', '', '', '', '', '', '', '',
       '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',
       'waterfront_1Point0', '', '', 'view_3Point0', 'view_4Point0', '',
       '', ''], dtype=object)

array([31, 36,  1, 15,  6, 35, 40, 12, 30, 28,  5,  1, 11, 34, 25, 24, 20,
       18, 17, 22, 23, 16, 19, 21, 42, 39, 43, 38, 10, 27,  9,  8,  7, 29,
       32, 13,  4, 14, 33,  1,  3,  2,  1,  1, 37, 26, 41])


SCORE: 0.7129496134760305
[135463.69321942 155654.11052375 256143.44361638 204817.41191781
 328199.09484982]
508792.72085770796
```

RFE doesn't produce a compellingly better model than what we picked, so we can put this aside.

In [72]:

```
#Let's see if we can put that whole mess into a function.
def process(isRand, X, y, missing_cols, dropped_cols, num_cols, cat_cols, to_
    '''input - X,y,column lists, booleans for whether to log/scale numerics
        ouput - sklearn model
    '''
    SPLIT_IS_RANDOM = isRand
    if SPLIT_IS_RANDOM:
        random_state = randint(1,2**32 - 2)
        #print(random_state)
    else:
        random_state = 14
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state =

    #clean the cols with NA values
    #missingcols = ['waterfront', 'view','yr_renovated'] #generally should pa

    handler_dict = {}
    for col in missing_cols:
        #handle missing cols
        handler_dict, X_train = handle_missing_train_col(col, handler_dict, X

    for col in dropped_cols:
        try:
            X_train = X_train.drop(col, axis=1)
        except:
            print('Exception:', col,'already dropped.')

    #one-hot encode categoricals
    cat_dict = {}
    for col in cat_cols:
        col_train = X_train[[col]]
        ohe = OneHotEncoder(categories='auto', sparse=False, drop='first', ha
        ohe.fit(col_train)
        # (4) Transform fireplace_qu_train using the encoder and
        # assign the result to fireplace_qu_encoded_train
        col_train_encoded = ohe.transform(col_train)
        col_train_encoded = pd.DataFrame(
            # Pass in NumPy array
            col_train_encoded,
            # Set the column names to the categories found by OHE
            columns = ohe.get_feature_names([col]),
            # Set the index to match X_train's index
            index=X_train.index
        )
        # (5b) Drop original column
        X_train.drop(col, axis=1, inplace=True)
        # (5c) Concatenate the new dataframe with current X_train
        X_train = pd.concat([X_train, col_train_encoded], axis=1)
        cat_dict[col] = ohe

    #check numericals for desirable properties, if problematic, then normaliz
    X_train.long = abs(X_train.long)

    non_log_cols = ['sqft_basement']
    for col in num_cols:
        currcol = X_train[col]
```

```
    if to_log:
        if col not in non_log_cols:
            currcol = log_col(currcol)
    if to_scale:
        currcol = scale(currcol)
    X_train[col] = currcol



#Model fit!
#from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)

cvscore = cross_val_score(model, X_train, y_train, cv=10)

#handle testset mappings
for col in missing_cols:
    X_test = handle_missing_test_col(col, handler_dict, X_test)

#handle testset drops
for col in dropped_cols:
    X_test = X_test.drop(col, axis = 1)

#handle testset categoricals
for col in cat_cols:
    col_test = X_test[[col]]
    curr_ohe = cat_dict[col]
    col_encoded_test = curr_ohe.transform(col_test)
    col_encoded_test = pd.DataFrame(
        col_encoded_test,
        columns = curr_ohe.get_feature_names([col]),
        index=X_test.index
    )
    X_test.drop(col, axis=1, inplace=True)
    X_test = pd.concat([X_test, col_encoded_test], axis=1)
#handle testset log/scale
X_test.long = abs(X_test.long)

non_log_cols = ['sqft_basement']
for col in num_cols:
    currcol = X_test[col]
    if to_log:
        if col not in non_log_cols:
            currcol = log_col(currcol)
    if to_scale:
        currcol = scale(currcol)
    X_test[col] = currcol



###Eval score on test set!
#model.fit(X_train, y_train)
score = model.score(X_test, y_test)
#display(model.summary)
```

```
    #print(model)
    ##############
    #regression_results(y_true, y_pred)
    y_pred = model.predict(X_test)
    regression_results(y_test, y_pred)



    ############
    # Make predictions
    expected = y_test
    predicted = model.predict(X_test)

    # Summarize the fit of the model
    mse = np.mean((predicted-expected)**2)
    print (model.intercept_, model.coef_, mse)
    print(model.score(X_test, y_test))

    #This summary isn't great, so let's use statsmodels summary instead
    #from statsmodels.api import OLS
    #results = OLS(y_test,X_test).fit()#.summary()

    #first, necessary to clean column names bc OLS gets confused by decimal p
    for col in X_train.columns:
        X_train.rename(columns = {col:col.replace(".", "Point")}, inplace = T
        X_test.rename(columns = {col:col.replace(".", "Point")}, inplace = Tr


    #import statsmodels.api as sm
    #from statsmodels.formula.api import ols
    data_ols = pd.concat([X_train, y_train], axis=1)
    outcome = target
    predictors = data_ols.drop(target, axis=1)
    pred_sum = '+'.join(predictors.columns)
    formula = outcome + '~' + pred_sum
    model = ols(formula=formula, data=data_ols).fit()
    model.summary()
    #to avoid making the summation string,  use the ols function
    '''The advantage is that you don't have to create the summation string. I
    #import statsmodels.api as sm
    predictors_int = sm.add_constant(predictors)
    #model = sm.OLS(data[target],predictors_int).fit()
    display(model.summary())
    results = model


    #Give RFE report
    rfe( X_train, y_train, target='price', num_features=5)


    return results

#df = pd.read_csv('data/kc_house_data.csv')
```

Try it on existing model

In [73]:

```
target = 'price'
X = df.drop(target, axis = 1)
y = df[target]

num_cols = [  'sqft_living', 'sqft_lot',
          'grade_squared',
       'yr_built', 'yr_renovated',  'lat', 'long',
       'sqft_living15', 'sqft_lot15',
            'bedrooms', 'bathrooms',
             'zipcode_avg',

       ]#'date',
#non_log_cols = ['sqft_basement, ']
#plan to move month/dayOfWeek to categorical eventually
#originally put bedrooms/bathrooms as categoricals...but moved back up to num
cat_cols = ['year', 'month', 'dayOfWeek','condition','floors', 'waterfront',
            'waterfront_Missing', 'view_Missing','yr_renovated_Missing']


dropped_cols = ['id','day', 'date', 'zipcode', 'grade', 'sqft_basement', 'sqf
missing_cols = ['waterfront', 'view','yr_renovated']



model_results = process(isRand=False, X=X, y=y, missing_cols=missing_cols, dr
```

```
explained_variance:  0.7758
r2:  0.7758
MAE:  91917.644
MSE:  18326091534.7588
RMSE:  135373.8953
459434.5171677111 [-1.15290955e+04  4.09435947e+03  8.61245043e+04  3.2
2914373e+04
 -4.01991388e+04  8.05207371e+03  2.03672162e+03  3.18991093e+04
  1.23332167e+04 -1.39565788e+04  7.93994937e+04  1.34119168e+05
  4.67782024e+04  9.58082684e+03  2.59968566e+04  3.31238609e+04
  3.88816519e+04  4.58065565e+04  4.62430509e+04  4.02127802e+04
  3.38538564e+04  5.08553779e+04  4.43525911e+04  4.28533657e+04
  7.22203513e+02  2.27802925e+03 -1.86201790e+02  3.03887309e+03
  3.06716778e+04 -1.22845641e+04 -2.95171011e+04 -2.42965395e+04
 -1.95720975e+04  2.91265053e+04  1.19520311e+04  2.79590433e+04
  8.13231121e+04  3.29723415e+04  9.39505156e+03  2.28682248e+05
  5.71920092e+04  7.16223487e+04  1.59115572e+05  2.70295340e+05
 -3.60644293e+03  2.00204677e+04  1.47091344e+03] 18326091534.758793
0.7757624750801912
```

Note, this is slightly different due to dropping some columns from multicollinearity above

In [74]:

```
#Try same columns, but now take off logging of numeric cols
target = 'price'
X = df.drop(target, axis = 1)
y = df[target]

num_cols = [  'sqft_living', 'sqft_lot',
           'grade_squared',
       'yr_built', 'yr_renovated',  'lat', 'long',
       'sqft_living15', 'sqft_lot15',
            'bedrooms', 'bathrooms',
             'zipcode_avg',

       ]#'date',
#non_log_cols = ['sqft_basement, ']
#plan to move month/dayOfWeek to categorical eventually
#originally put bedrooms/bathrooms as categoricals...but moved back up to num
cat_cols = ['year', 'month', 'dayOfWeek','condition','floors', 'waterfront',
            'waterfront_Missing', 'view_Missing','yr_renovated_Missing']


dropped_cols = ['id','day', 'date', 'zipcode', 'grade', 'sqft_basement', 'sqf
missing_cols = ['waterfront', 'view','yr_renovated']



model_results2 = process(isRand=False, X=X, y=y, missing_cols=missing_cols, d
```

```
explained_variance:  0.8241
r2:  0.824
MAE:  81882.0645
MSE:  14379832121.6121
RMSE:  119915.9377
428458.55306424026 [-10450.14973074  13778.56893779  97851.89526244  11
018.89617204
 -40405.4579766    8045.16310028  18486.04351414  23392.33923676
   5654.28224009  -2882.85568851  81722.72855334 122928.9652504
  47756.10245567   7106.4298074   25093.34488072  33936.75522297
  37665.58258932  47390.33416723  47443.84679667  41648.36911746
  37055.86665661  47939.52160684  44759.09449526  43370.53204184
    886.9713091    2542.29796811  -1285.11125607   3061.90229554
  26188.51883668  -9783.96002712  -1451.36673982  12769.23888114
  21936.56284142  67699.07795265  17543.25799823  11303.00007513
  49560.51774718  10287.17979253    901.92305999 260014.43173278
  59447.7952848   62640.64248205 135295.25350382 248575.95860931
  -3252.32864242  19463.36677613   1247.02799796] 14379832121.612127
0.8240487908948387
```

In [75]:
```
model_results2.summary()
sorted(model_results2.params)
large_params = [ 81722.72855334177,
97851.89526243522, 122928.9652504016, 135295.2535038157,
 248575.9586093065,
 260014.43173277803,
 428458.55306424014, -40405.457976600475]
model_results2.params
#zipcode_avg                  122928.965250
#view_3Point0                 135295.253504
#view_4Point0                 248575.958609
#waterfront_1Point0           260014.431733
#sqft_living                   97851.895262
#grade_squared                 81722.728553

#yr_built                     -40405.457977
'''The top 6 coefficients in descending order are: zipcode_avg, view_3.0, vie
```

Out[75]:
```
'The top 6 coefficients in descending order are: zipcode_avg, view_3.0, vie
w4.0, waterfront_1.0, sqft_living, and grade_squared.'
```

Wow, that's great! By not logging, we improved the model. Scaling will still allow us to get good interpretable coefficients. These metrics also should be considered in accuracy of the model.
explained_variance: 0.8241
r2: 0.824
MAE: 81882.0645
MSE: 14379832121.6121
RMSE: 119915.9377
But of course, we are using linear regression for interpretability, so looking at the coefficients is of interest too.
sqft_living, grade_squared, zipcode_avg, waterfront_1, view3 and view4 all have on the order of 10**5 coefficient.
Day of week and month are also contenders.

In [76]:
```
X.columns
target
display(X_train.columns)
X.columns
```

```
Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'yr_built',
       'yr_renovated', 'lat', 'long', 'sqft_living15', 'sqft_lot15',
       'grade_squared', 'zipcode_avg', 'year_2015', 'month_2', 'month_3',
       'month_4', 'month_5', 'month_6', 'month_7', 'month_8', 'month_9',
       'month_10', 'month_11', 'month_12', 'dayOfWeek_1', 'dayOfWeek_2',
       'dayOfWeek_3', 'dayOfWeek_4', 'dayOfWeek_5', 'dayOfWeek_6',
       'condition_2', 'condition_3', 'condition_4', 'condition_5',
       'floors_1Point5', 'floors_2Point0', 'floors_2Point5', 'floors_3Point
0',
       'floors_3Point5', 'waterfront_1Point0', 'view_1Point0', 'view_2Point
0',
       'view_3Point0', 'view_4Point0', 'waterfront_Missing_True',
       'view_Missing_True', 'yr_renovated_Missing_True'],
      dtype='object')
```

Out[76]:
```
Index(['id', 'date', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot',
       'floors', 'waterfront', 'view', 'condition', 'grade', 'sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'lon
g',
       'sqft_living15', 'sqft_lot15', 'grade_squared', 'year', 'month', 'da
y',
       'dayOfWeek', 'zipcode_avg'],
      dtype='object')
```

In [77]:

```
#Set up for round 3
#Here, I moved bathrooms to categorical; month/dayofweek to numeric
target = 'price'
X = df.drop(target, axis = 1)
y = df[target]

num_cols = [  'sqft_living', 'sqft_lot',
           'grade_squared',
       'yr_built', 'yr_renovated',  'lat', 'long',
       'sqft_living15', 'sqft_lot15',
            'bedrooms',
             'zipcode_avg',
            'month', 'dayOfWeek'

       ]#'date',
#non_log_cols = ['sqft_basement, ']
#plan to move month/dayOfWeek to categorical eventually
#originally put bedrooms/bathrooms as categoricals...but moved back up to num
cat_cols = ['year','condition','floors', 'waterfront', 'view',
            'waterfront_Missing', 'view_Missing','yr_renovated_Missing', 'bat


dropped_cols = ['id','day', 'date', 'zipcode', 'grade', 'sqft_basement', 'sqf
missing_cols = ['waterfront', 'view','yr_renovated']
model_results3 = process(isRand=False, X=X, y=y, missing_cols=missing_cols, d
```

```
explained_variance:  0.8275
r2:  0.8275
MAE:  81098.8746
MSE:  14100594354.6776
RMSE:  118745.9235
414007.32667407196 [  -8708.8432328    94677.29597361   10074.50650632
-37683.60314099
    9326.45978183   18259.19790288   22532.95518904    7154.25295181
   -2586.35289208   80124.82660546    4969.21154544     687.40193698
  122915.05585101   34739.15827232    3835.31543821   23398.78765805
   35614.86905676   82413.07805872   19298.10324381   13803.24909625
   48422.11842594   10791.66338388   11863.19779786  259302.94758798
   60834.96170479   62691.58195443  133581.23177861  249040.04474069
   -3363.92793353   16106.31304622    1091.83604958   59311.08128885
   42688.47873894   33641.23127454   29601.62850457   28959.29990142
   36109.17487524   36527.2762548    38082.19761558   50704.82527805
   60662.38751246  105606.31374267   92738.32120339  182180.41780844
  115034.10384429  146949.00632601  126107.98899595  135870.69362112
  172514.65025467   60156.81657282  235965.37574029  255109.01336006
```

In [78]:

```
#model.score(X_test, y_test)
#model_results.summary()
```

In [79]:

```
#Set up for round4 - here, move all categoricals to numeric
target = 'price'
X = df.drop(target, axis = 1)
y = df[target]

dropped_cols = ['id','day', 'date', 'zipcode', 'grade', 'sqft_basement', 'sqf
missing_cols = ['waterfront', 'view','yr_renovated']

num_cols = ['sqft_living', 'sqft_lot',
        'condition', 'grade_squared',
         'yr_renovated', 'lat', 'long',
       'sqft_living15', 'sqft_lot15',
            'zipcode_avg',

           'floors', 'waterfront', 'view', 'waterfront_Missing', 'view_Missi
          'bedrooms', 'bathrooms',
       ]
#originally put bedrooms/bathrooms as categoricals...but moved back up to num
cat_cols = []

model_results4 = process(isRand=False, X=X, y=y, missing_cols=missing_cols, d
```

```
explained_variance:  0.8237
r2:  0.8237
MAE:  81820.8908
MSE:  14410798107.0492
RMSE:  120044.9837
-68247956.54679595 [-10297.91823104  14167.17989998  98444.67444765  10
670.43140777
   7098.61238162  22226.55055455  32901.88897375  13329.92659071
 -42934.46412158   8188.06747798  18981.69889461  22853.79687184
   5819.65936397  -2872.90925725  81549.25391604  34133.15803916
   1515.58854594    655.30462162 122284.9874657    -924.55546866
   1087.79871247    457.10779963] 14410798107.049238
0.8236698919944405
```

This has a good R-squared, but high condition, indicating multicollinearity.

Model_results2 has the best balance of Score, while avoiding a large Condition number and other issues. Let's double check it for requirements of linear regression models

Certain assumptions must hold true for a least squares linear regression to be useful - linearity of response-predictor variables, normality of residuals, and homoscedasticity of

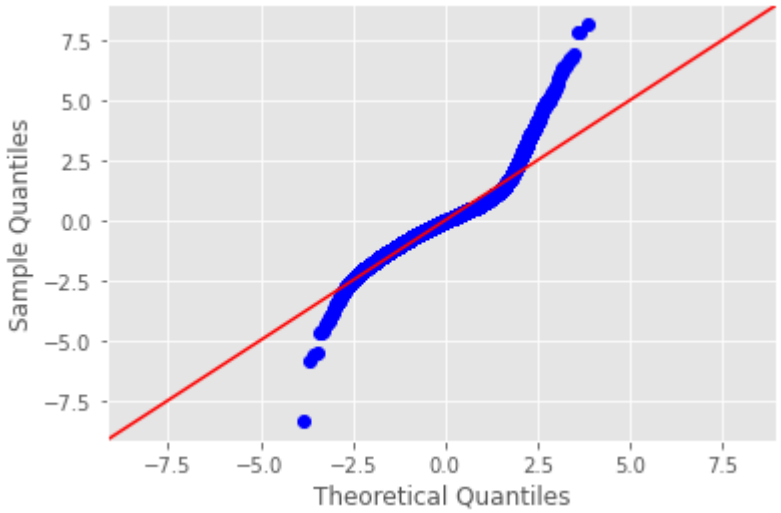
We checked linearity of the predictor variables early on - with notable exception of grade, which we addressed by adding a quadratic term.

We see below that there are some normality problems with the residuals, but at the tails. The main bulk of the values follow the desired line.

Finally, for the requirement of homoscedasticity, I see below that there isn't clear heteroscedasticity for any numeric predictor variable. However, when I also perform the Lagrange Multiplier Heteroscedasticity Test by Breusch-Pagan, the p-value is close to 0. So, there is some underlying heteroscedasticity in the model.

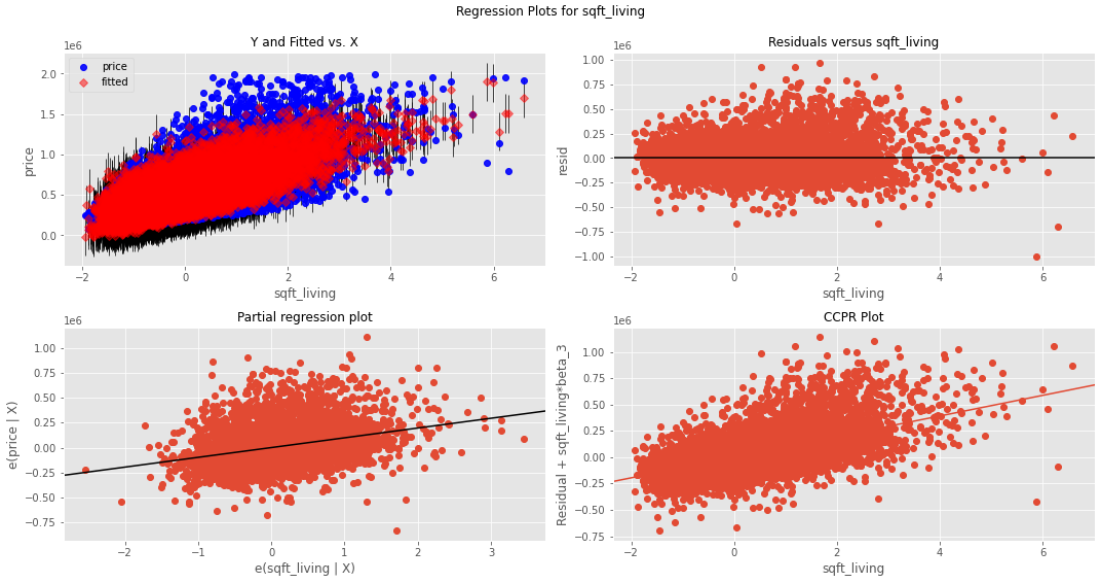
In [80]:

```
import statsmodels.stats.api as sms
import scipy.stats as stats
plt.style.use('ggplot')

model = model_results2
resid1 = model.resid
#resid2 = model2.resid
fig = sm.graphics.qqplot(resid1, dist=stats.norm, line='45', fit=True)
#fig = sm.graphics.qqplot(resid2, dist=stats.norm, line='45', fit=True)
```



In [81]:

```
num_cols = [  'sqft_living', 'sqft_lot',
           'grade_squared',
       'yr_built', 'yr_renovated',  'lat', 'long',
       'sqft_living15', 'sqft_lot15',
            'bedrooms', 'bathrooms',
             'zipcode_avg',

       ]#'date',
#non_log_cols = ['sqft_basement, ']
#plan to move month/dayOfWeek to categorical eventually
#originally put bedrooms/bathrooms as categoricals...but moved back up to num
cat_cols = ['year', 'month', 'dayOfWeek','condition','floors', 'waterfront',
            'waterfront_Missing', 'view_Missing','yr_renovated_Missing']


dropped_cols = ['id','day', 'date', 'zipcode', 'grade', 'sqft_basement', 'sqf
missing_cols = ['waterfront', 'view','yr_renovated']

for col in num_cols:
    fig = plt.figure(figsize=(15,8))
    fig = sm.graphics.plot_regress_exog(model, col, fig=fig)
    plt.show()
```



In [82]:

```
# for col in X_train.columns:
#     plt.plot(model.resid)
#X_train.info()
```

In [83]:

```
#Lagrange Multiplier Heteroscedasticity Test by Breusch-Pagan
'''The tests the hypothesis that the residual variance does not depend on the
num_cols = [  'sqft_living', 'sqft_lot',
       'yr_built', 'yr_renovated',  'lat', 'long',
       'sqft_living15', 'sqft_lot15',
            'bedrooms', 'bathrooms',
       ]
from statsmodels.stats.diagnostic import het_breuschpagan
'''lmfloat
lagrange multiplier statistic

lm_pvaluefloat
p-value of lagrange multiplier test

fvaluefloat
f-statistic of the hypothesis that the error variance does not depend on x

f_pvaluefloat
p-value for the f-statistic'''

(v1, v2, v3, v4) = het_breuschpagan(model.resid, model.model.exog, robust=Tru
#model.resid
(v1, v2, v3, v4)
```

Out[83]: (2707.2585924507707, 0.0, 69.0891408107041, 0.0)

In [84]:

```
# import statsmodels.stats.api as sms
# num_cols = [  'sqft_living', 'sqft_lot',
#         'yr_built', 'yr_renovated',  'lat', 'long',
#         'sqft_living15', 'sqft_lot15',
#              'bedrooms', 'bathrooms',

#         ]

# #x_t1, y_t1

# for col in num_cols:
#     display(col)
#     lwr_thresh = X_train[col].quantile(q=.45)
#     upr_thresh = X_train[col].quantile(q=.55)
#     middle_10percent_indices = X_train[(X_train[col] >= lwr_thresh) & (X_tr
#     display(len(middle_10percent_indices))

#     indices = [x-1 for x in X_train.index if x not in middle_10percent_indi
#     plt.scatter(X_train[col].iloc[indices], model.resid.iloc[indices])
#     plt.xlabel(col)
#     plt.ylabel('Model Residuals')
#     plt.title("Residuals versus {} Feature".format(col))
#     plt.vlines(lwr_thresh, ymax=8, ymin=-8, linestyles='dashed',linewidth=2
#     plt.vlines(upr_thresh, ymax=8, ymin=-8, linestyles='dashed',linewidth=2
#     name = ['F statistic', 'p-value']
#     test = sms.het_goldfeldquandt(model.resid.iloc[indices], model.model.ex
#     list(zip(name, test))
# '''lwr_thresh = data.TV.quantile(q=.45)
# upr_thresh = data.TV.quantile(q=.55)
# middle_10percent_indices = data[(data.TV >= lwr_thresh) & (data.TV<=upr_thr
# # len(middle_10percent_indices)

# indices = [x-1 for x in data.index if x not in middle_10percent_indices]
# plt.scatter(data.TV.iloc[indices], model.resid.iloc[indices])
# plt.xlabel('TV')
# plt.ylabel('Model Residuals')
# plt.title("Residuals versus TV Feature")
# plt.vlines(lwr_thresh, ymax=8, ymin=-8, linestyles='dashed',linewidth=2)
# plt.vlines(upr_thresh, ymax=8, ymin=-8, linestyles='dashed',linewidth=2);''
# # Run Goldfeld Quandt test
# # import statsmodels.stats.api as sms
# # name = ['F statistic', 'p-value']
# # test = sms.het_goldfeldquandt(model2.resid.iloc[indices], model2.model.ex
# # list(zip(name, test))
```

In [ ]:

Takeaways:

the categories that seem to be of consistently highest impact are sqft_living, grade_squared, zipcode_avg, waterfront, View, day of week (namely Saturday vs Sunday)

In [ ]:

```
```

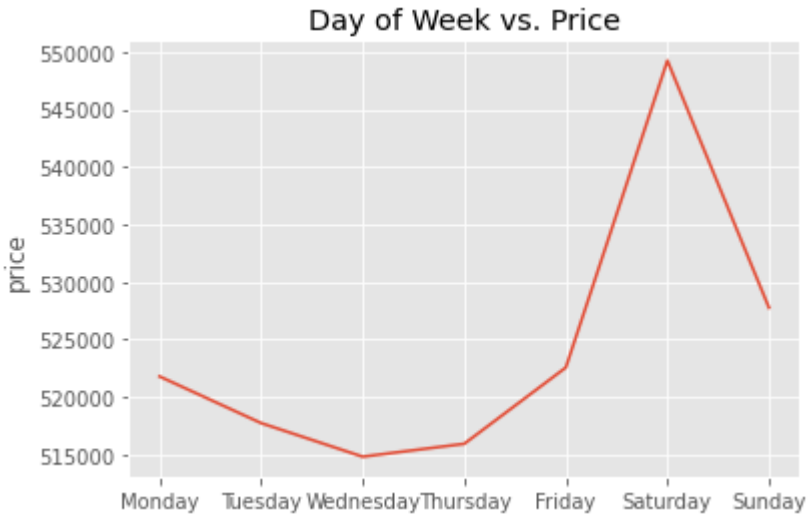
1) I performed Exploratory Data Analysis to identify characteristics of the data
2) I created several new features including grade squared, day of the week, month.
2) I performed multiple linear regression (with one polynomial regression term) under several different permutations (e.g. logging numericals, scaling numericals, and different ways of classifying categoricals/numeric values
4) I identified a model that had a good Score as well as minimal multicollinearity.
5) I examined the necessary conditions of linear regression
6) The business takeaways: sqft_living, grade_squared, zipcode_avg, waterfront, View, day of week (namely Saturday vs Sunday) all have a (normalized) significant impact on price. Furthermore, the model is ready to accept any new home predictors for comparison to current prices, in order to establish if a home is over- or under-valued.

Visualizations again

In [85]:

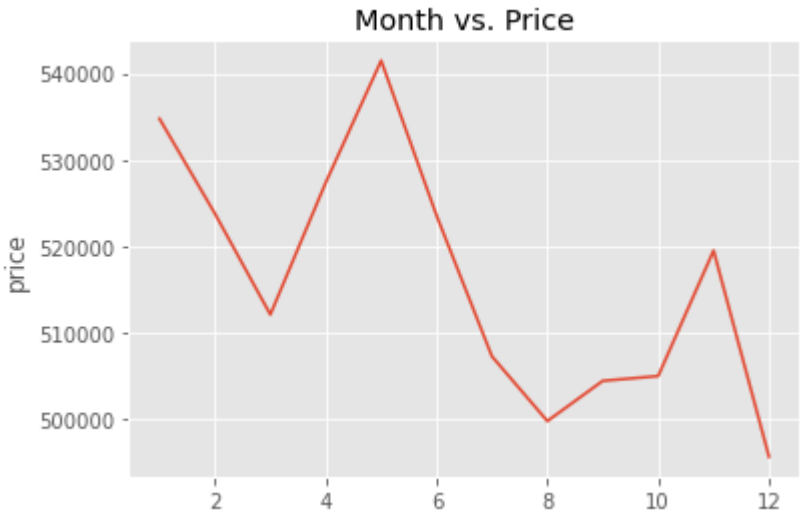
```
#sns.lineplot(x=df['dayOfWeek'].unique(), y=df.groupby('dayOfWeek').mean()['p
sns.lineplot(x=l, y=df.groupby('dayOfWeek').mean()['price']).set_title('Day o
```

Out[85]: Text(0.5, 1.0, 'Day of Week vs. Price')



In [101]:
```
m = df['month'].unique()
#m = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', '
sns.lineplot(x=df['month'].unique(), y=df.groupby('month').mean()['price'], s
```

Out[101]: Text(0.5, 1.0, 'Month vs. Price')

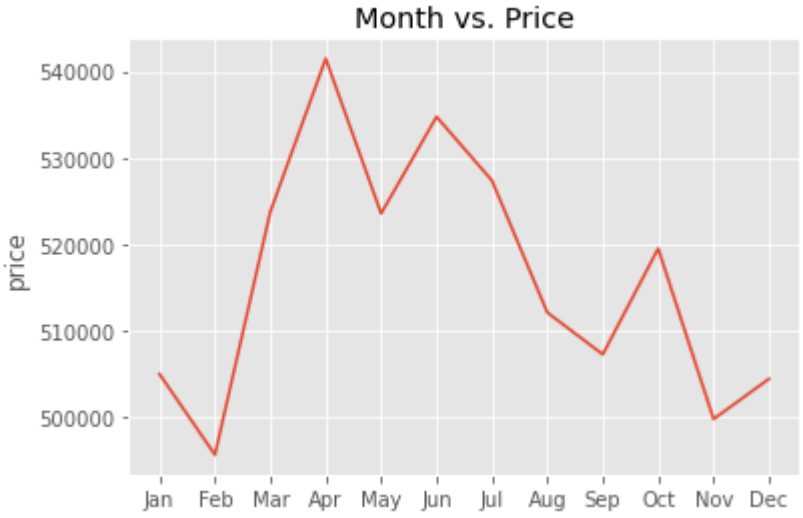


In [102]:

```
m = df['month'].unique()
display(m)
m = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'N
display(m)
sns.lineplot(x=m, y=df.groupby('month').mean()['price'], sort=True).set_title
```

```
array([10, 12,  2,  5,  6,  1,  4,  3,  7, 11,  8,  9], dtype=int64)

['Jan',
 'Feb',
 'Mar',
 'Apr',
 'May',
 'Jun',
 'Jul',
 'Aug',
 'Sep',
 'Oct',
 'Nov',
 'Dec']
```

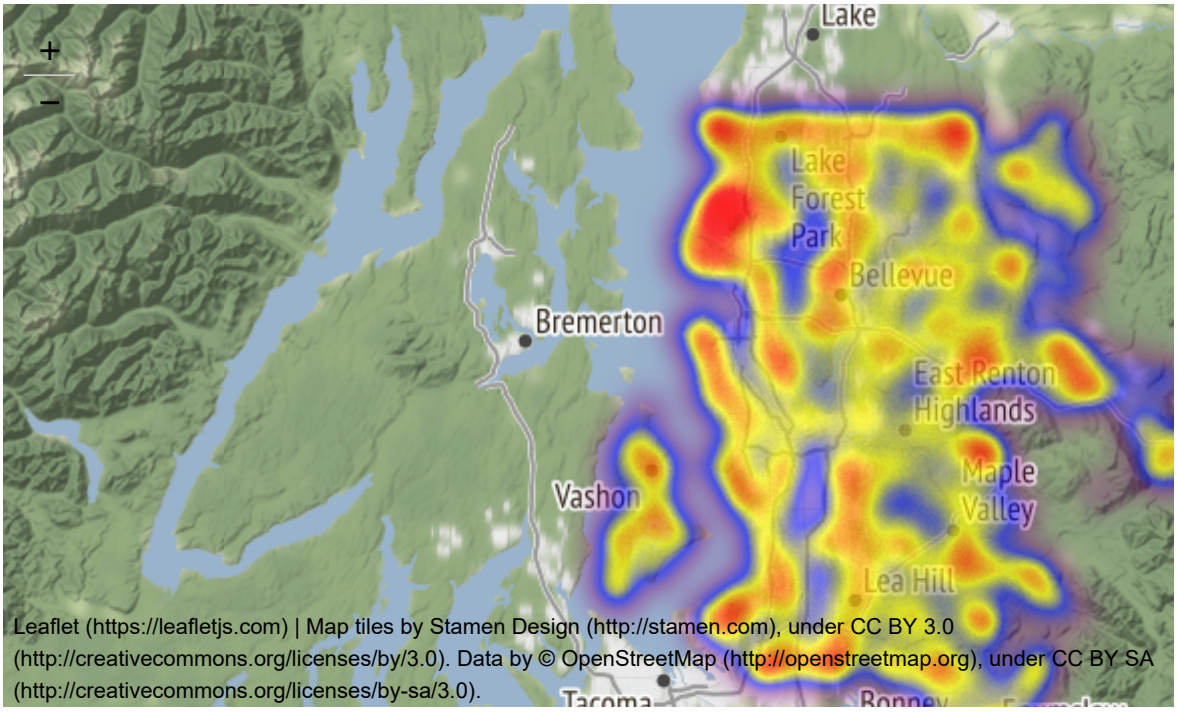
Out[102]: Text(0.5, 1.0, 'Month vs. Price')



In [87]:

```
# Neighborhood has a huge impact in house prices historically.  Though linear
# I found a nice visualization package for this.
import folium
from folium.plugins import HeatMap

price_map = folium.Map(location=[47.5,-122.3],
                       tiles = "Stamen Terrain",
                      zoom_start = 9)
data_heatmap = df[['lat','long','price']]
data_heatmap = df.dropna(axis=0, subset=['lat','long','price'])
data_heatmap = [[row['lat'],row['long']] for index, row in data_heatmap.iterr
HeatMap(data_heatmap, radius=10,
        gradient = {.35: 'purple',.55: 'blue',.68:'yellow',.78:'red'}).add_to
price_map
```

Out[87]:

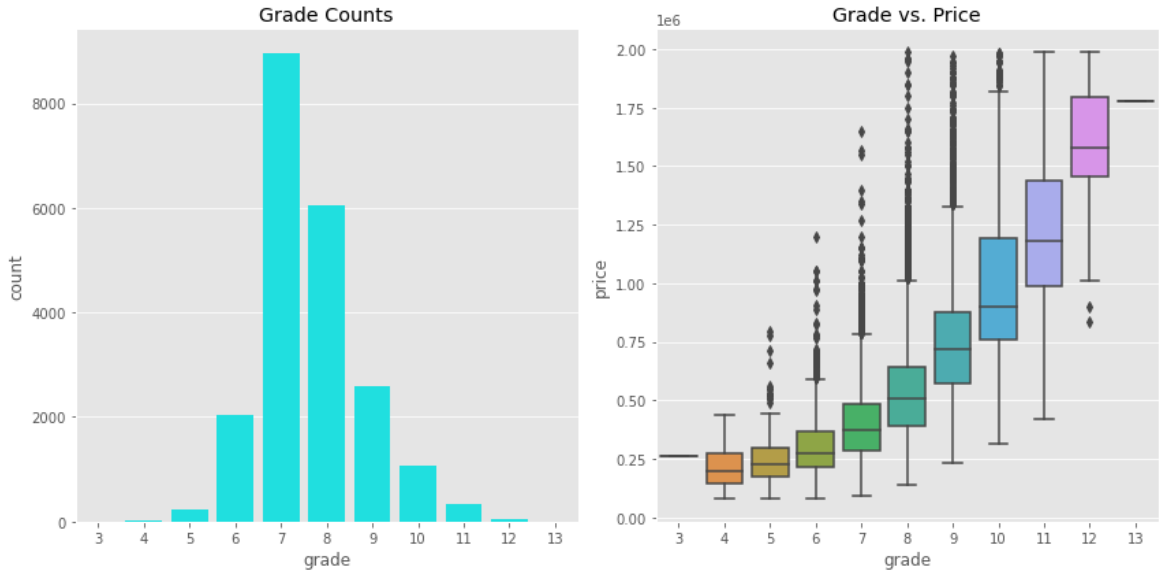


In [88]:

```
#A quick look at sqft vs price
plt.figure(figsize=(9, 5))
sns.scatterplot(x='sqft_living', y='price', data=df).set_title('Square Feet v
plt.ticklabel_format(style='plain')#style='plain', 'sci', 'scientific'
```

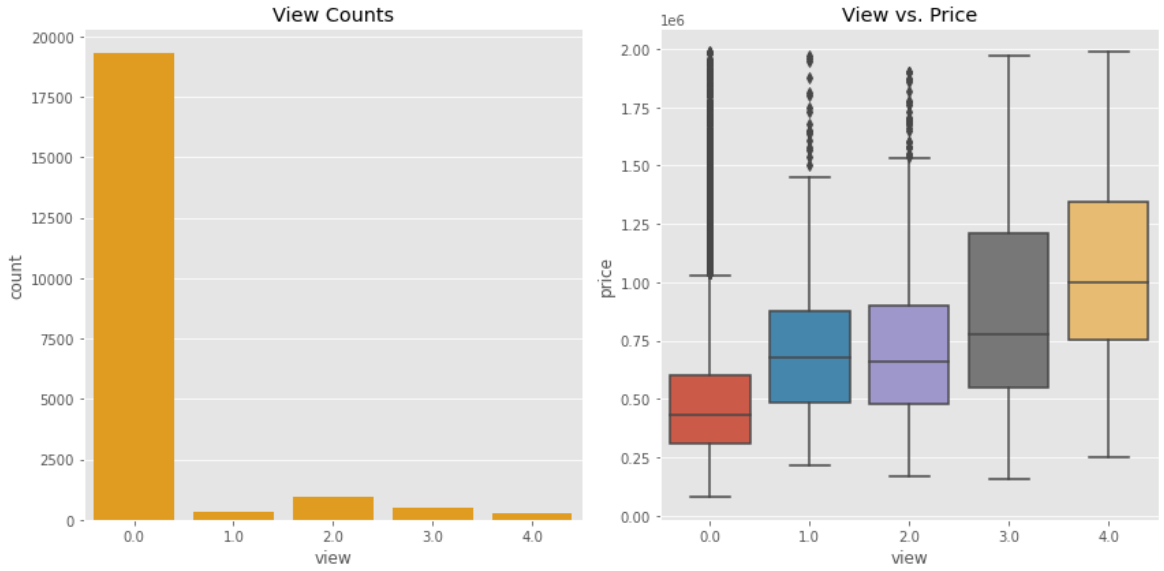
In [89]:

```
# grade vs price

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('Grade Counts')
sns.countplot(df['grade'], ax=axes[0], color='cyan')

axes[1].set_title('Grade vs. Price')
sns.boxplot(x='grade', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



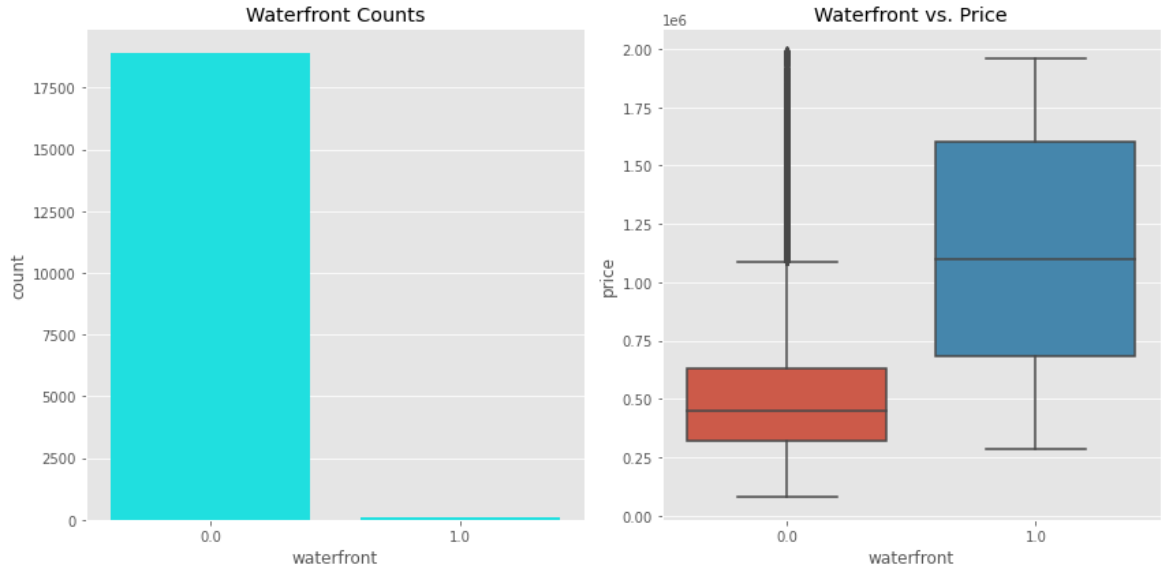
In [90]:

```
# View
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('View Counts')
sns.countplot(df['view'], ax=axes[0], color='orange')

axes[1].set_title('View vs. Price')
sns.boxplot(x='view', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



In [91]:

```
# Waterfront
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].set_title('Waterfront Counts')
sns.countplot(df['waterfront'], ax=axes[0], color='cyan')

axes[1].set_title('Waterfront vs. Price')
sns.boxplot(x='waterfront', y='price', data=df, ax=axes[1])

plt.tight_layout()
```



In [92]:

```
#I wound up not using this for presentation
# #Exploring correlation and multicollinearity in the data

# #X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = r

# heatmap_data = pd.concat([y_train, X_train], axis=1)
# # https://seaborn.pydata.org/generated/seaborn.jointplot.html
# #did corr matrix
# heatmap_data.corr()
# #used heatmap
# import seaborn as sns
# import numpy as np

# # Create a df with the target as the first column,
# # then compute the correlation matrix
# ##heatmap_data = pd.concat([y_train, X_train], axis=1)
# ##corr = heatmap_data.corr()
# corr = heatmap_data.corr()

# # Set up figure and axes
# fig, ax = plt.subplots(figsize=(10, 16))

# # Plot a heatmap of the correlation matrix, with both
# # numbers and colors indicating the correlations

# sns.heatmap(
#     # Specifies the data to be plotted
#     data=corr,
#     # The mask means we only show half the values,
#     # instead of showing duplicates. It's optional.
#     mask=np.triu(np.ones_like(corr, dtype=bool)),
#     # Specifies that we should use the existing axes
#     ax=ax,
#     # Specifies that we want labels, not just colors
#     annot=True,
#     # Customizes colorbar appearance
#     cbar_kws={"label": "Correlation", "orientation": "horizontal", "pad": .
# )

# # Customize the plot appearance
# ax.set_title("Heatmap of Correlation Between Attributes (Including Target)"
```

In [93]:

```
#to update - add something about model to non-tech presentation -give R^2 ana
#top 5 most important features
#Done!
```