M1 Project 1b

September 14, 2023

1 CS5610 M1: Project 1b

1.1 Introduction to Pandas and Seaborn Python Libraries

By Jacob Buysse, 2023-09-07

In this notebook we will demonstrate how to use `pandas` and `seaborn` to load, analyze, plot, clean, and save a file containing real estate transactions in the Sacramento area.

We will be using the `pandas`, `seaborn`, and `matplotlib` libraries imported as `pd`, `sns`, and `plt`, respectively. We will also be using the `datetime.datetime` class to parse dates.

[1]:
```
import pandas as pd
import seaborn as sns
import matplotlib as plt
from datetime import datetime
```

Configure the default axis label size to 18. Ensure the plots generated have a high resolution (150dpi).

[2]:
```
plt.rc("axes", labelsize=18)
plt.rc("figure", dpi=150)
```

1.2 Load the Data and a First Look

Load the data from the input CSV file and display the head and info.

[3]:
```
df = pd.read_csv("csc5610-m2-Sacramento-real-estate-transactions.csv")
df.head()
```

[3]:
```
           address        city    zip state  beds  baths  sq__ft  \
0      3526 HIGH ST  SACRAMENTO  95838    CA     2      1     836
1       51 OMAHA CT  SACRAMENTO  95823    CA     3      1    1167
2    2796 BRANCH ST  SACRAMENTO  95815    CA     2      1     796
3  2805 JANETTE WAY  SACRAMENTO  95815    CA     2      1     852
4   6001 MCMAHON DR  SACRAMENTO  95824    CA     2      1     797

          type                     sale_date  price   latitude   longitude
0  Residential  Wed May 21 00:00:00 EDT 2008  59222  38.631913 -121.434879
1  Residential  Wed May 21 00:00:00 EDT 2008  68212  38.478902 -121.431028
```

```
2  Residential  Wed May 21 00:00:00 EDT 2008  68880  38.618305 -121.443839
3  Residential  Wed May 21 00:00:00 EDT 2008  69307  38.616835 -121.439146
4  Residential  Wed May 21 00:00:00 EDT 2008  81900  38.519470 -121.435768
```

The file contains columns for: * **address** - street address * **city** - uppercase city name * **zip** - 5 digit, does not include zip+4 * **state** - 2 character abbreviation * **beds** - number of bedrooms * **baths** - number of bathrooms * **sq___ft** - square footage * **type** - appears to be classification of the type of property * **sale_date** - the date the transaction took place * **price** - sale price (assumed to be in USD($)) * **latitude** - degrees north of the equator * **longitude** - degrees west of the meridian

[4]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 985 entries, 0 to 984
Data columns (total 12 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   address    985 non-null    object
 1   city       985 non-null    object
 2   zip        985 non-null    int64
 3   state      985 non-null    object
 4   beds       985 non-null    int64
 5   baths      985 non-null    int64
 6   sq__ft     985 non-null    int64
 7   type       985 non-null    object
 8   sale_date  985 non-null    object
 9   price      985 non-null    int64
 10  latitude   985 non-null    float64
 11  longitude  985 non-null    float64
dtypes: float64(2), int64(5), object(5)
memory usage: 92.5+ KB
```

We see that there are 985 rows and that none of the columns contain any nulls. The Dtype isn't so helpful for `object` columns.

Next, do a deeper analysis on the columns for the underlying python types and also get the count of distinct values to find some candidates for converting to categories.

[5]:
```python
df.columns.to_series().apply(lambda name: {
    "type": df[name].apply(lambda value: type(value).__name__).unique(),
    "count": df[name].nunique()
})
```

[5]:
```
address        {'type': ['str'], 'count': 981}
city            {'type': ['str'], 'count': 39}
zip             {'type': ['int'], 'count': 68}
state            {'type': ['str'], 'count': 1}
beds             {'type': ['int'], 'count': 8}
```

```
baths               {'type': ['int'], 'count': 6}
sq__ft            {'type': ['int'], 'count': 603}
type                {'type': ['str'], 'count': 4}
sale_date           {'type': ['str'], 'count': 5}
price             {'type': ['int'], 'count': 605}
latitude        {'type': ['float'], 'count': 969}
longitude       {'type': ['float'], 'count': 967}
dtype: object
```

So all of the `object` columns were indeed just `strs`. `type`, `city`, `state`, and `zip` appear to be good candidates for categories. `beds`, `baths`, and `sale_date` have low distinct counts but don't logically represent categorical data. We can see that both `beds` and `baths` are `ints` so we don't have to worry about half-baths or other unexpected floating point data. We can see that sale_date is a `str` so we will likely want to parse this value into a `datetime` for our final clean results.

We won't do any further analysis or cleaning of the `address` column for this project.

1.3 Type Analysis

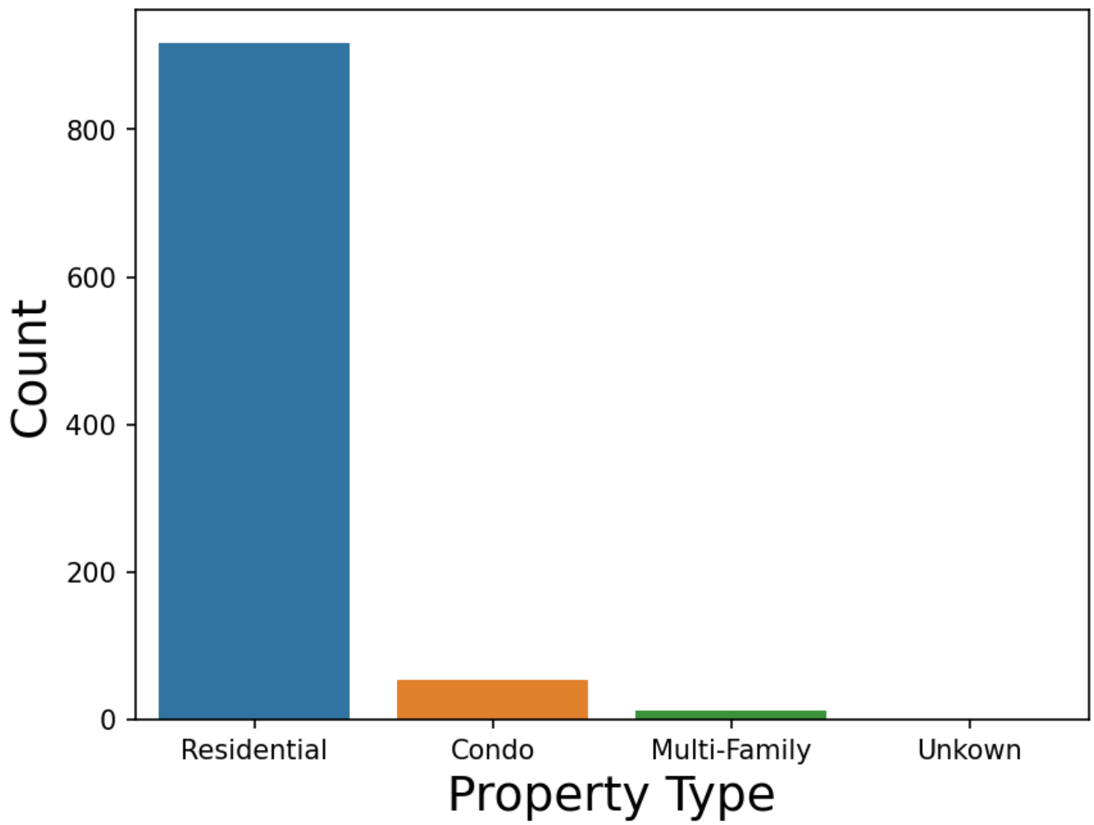
Let us take a look at the data for the type column.

[6]:
```
df.groupby(by="type").size()
```

[6]:
```
type
Condo             54
Multi-Family      13
Residential      917
Unkown             1
dtype: int64
```

We can see that `Unknown` is both odd and spelled incorrectly. We will likely filter that single record out in our final results. Let us look at some plots for this data.

[7]:
```
type_plot = sns.countplot(data=df, x="type")
type_plot.set(xlabel="Property Type", ylabel="Count");
type_plot.figure.savefig("m1p1b.type.png")
```

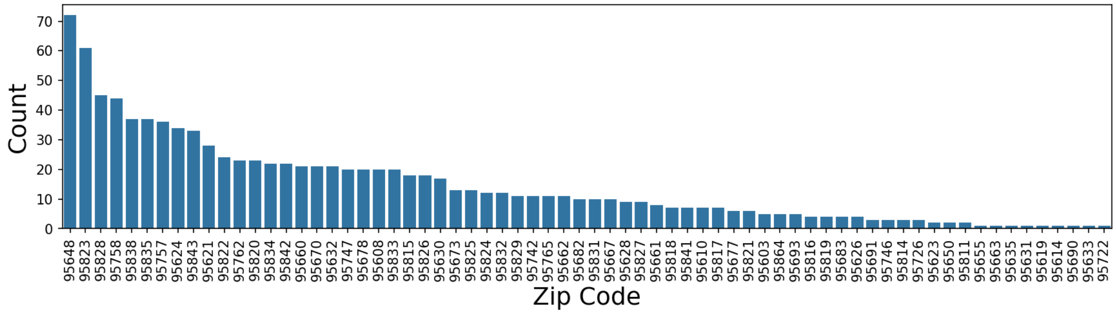


We can see that condos and multi-family properties don't make up a significant proportion of the rows. However, I don't think it is enough to exclude the data at this point.

1.4 Zip Analysis

Let us take a look at the histogram plot for the zip codes (even though there are 68 of them, they do make a good candidate for a category).

[8]:
```
zip_counts = df.zip.value_counts()
_, ax = plt.pyplot.subplots(figsize=(14, 3))
zip_plot = sns.countplot(data=df, x="zip", ax=ax, order=zip_counts.index,␣
↪color=sns.color_palette()[0])
zip_plot.set(xlabel="Zip Code", ylabel="Count")
zip_plot.tick_params(axis='x', rotation=90)
```



None of the zip codes look out of place. A secondary data source could validate that these zip codes are all valid, but that is out of the scope of this analysis. However, some of the trailing items appear to have almost no data.

[9]:
```
zip_counts.apply(lambda count: "More than 1" if count > 1 else "Exactly 1").
↪value_counts()
```

[9]:
```
count
More than 1    59
Exactly 1       9
Name: count, dtype: int64
```

And it turns out that 9 of them only have a single corresponding row. I don't have the exact context for what we will be using this data for, but we will likely want to filter out these records since they don't provide a valid sample size for the given area.

1.5 City Analysis

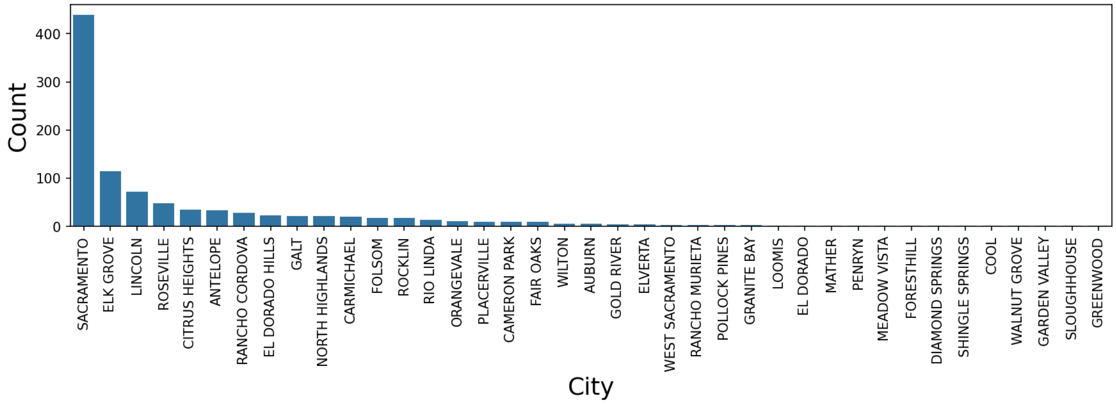
Let's start by making sure the 39 distinct values in the table don't have any duplicates solely on different casing.

[10]:
```
df.city.apply(lambda city: city.lower()).nunique()
```

[10]:
```
39
```

And we can see that they don't. Now let us take a look at a histogram plot.

[11]:
```
city_counts = df.city.value_counts()
_, ax = plt.pyplot.subplots(figsize=(14, 3))
city_plot = sns.countplot(data=df, x="city", ax=ax, order=city_counts.index,␣
↪color=sns.color_palette()[0])
city_plot.set(xlabel="City", ylabel="Count")
city_plot.tick_params(axis='x', rotation=90)
```



The vast majority of rows appear to be for Sacramento proper but there are still 38 other cities being included in the file. We can also see that some of the data doesn't look valid: Cool. But upon further investigation it does appear to be a real suburb of Sacramento. How many of the cities only have one corresponding row in the table?

[12]:
```
city_counts.apply(lambda count: "More than 1" if count > 1 else "Exactly 1").
↪value_counts()
```

[12]:
```
count
More than 1    28
Exactly 1      11
Name: count, dtype: int64
```

We can see that 11 cities only have a single row. I don't have the exact context for what we will be using this data for, but we will likely want to filter out these records since they don't provide a valid sample size for the given area.

1.6 Beds Analysis

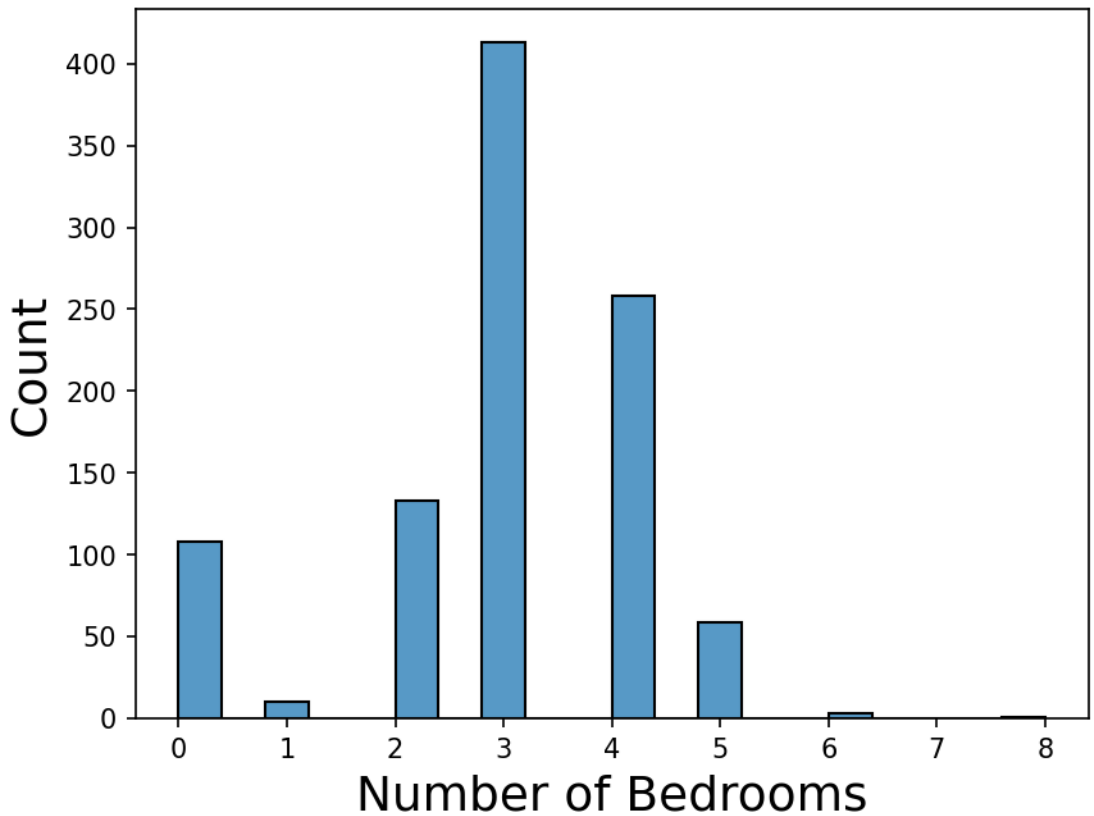
Let describe the beds column to get an idea of the aggregate statistics.

[13]:
```
df.beds.describe()
```

[13]:
```
count    985.000000
mean       2.911675
std        1.307932
min        0.000000
25%        2.000000
50%        3.000000
75%        4.000000
max        8.000000
Name: beds, dtype: float64
```

The min value of 0 looks strange. How does this look when shown on a histogram plot?

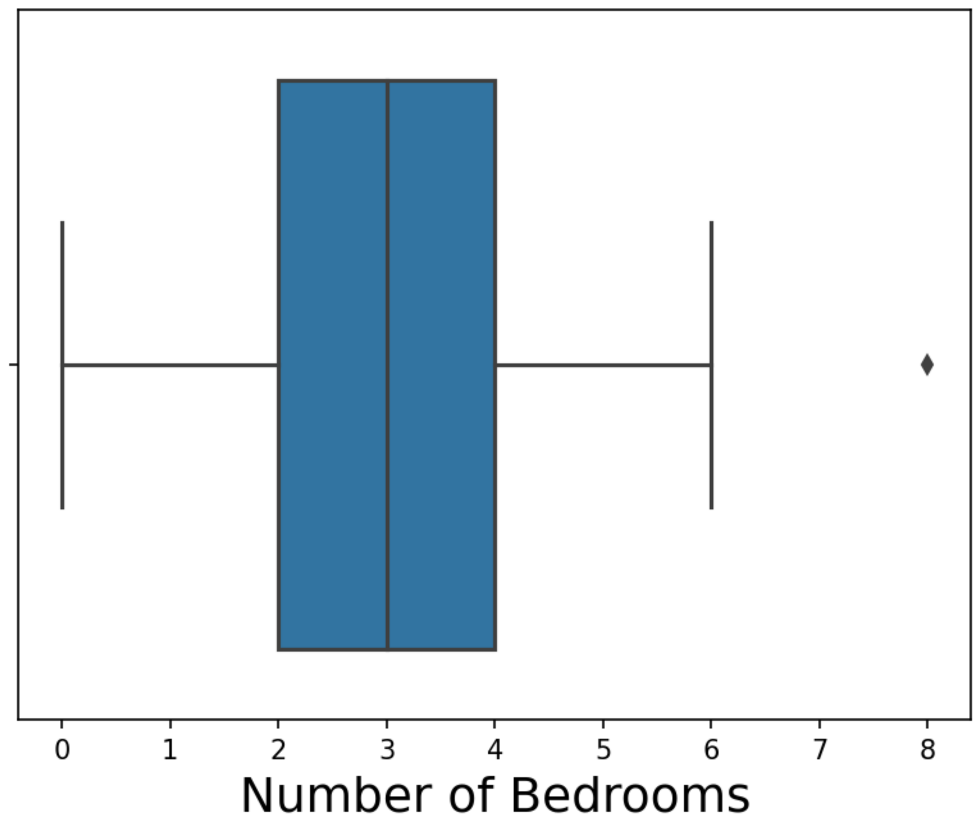
[14]:
```
beds_plot = sns.histplot(df.beds)
beds_plot.set(xlabel="Number of Bedrooms");
```



We have a somewhat power law distribution between 1 and 8. The zero values are definitely wrong (unless these were studio condos). There is an outlier out at 8 but that is a reasonable mansion. It would be worth considering exlucding that house depending on what our final analysis might be.

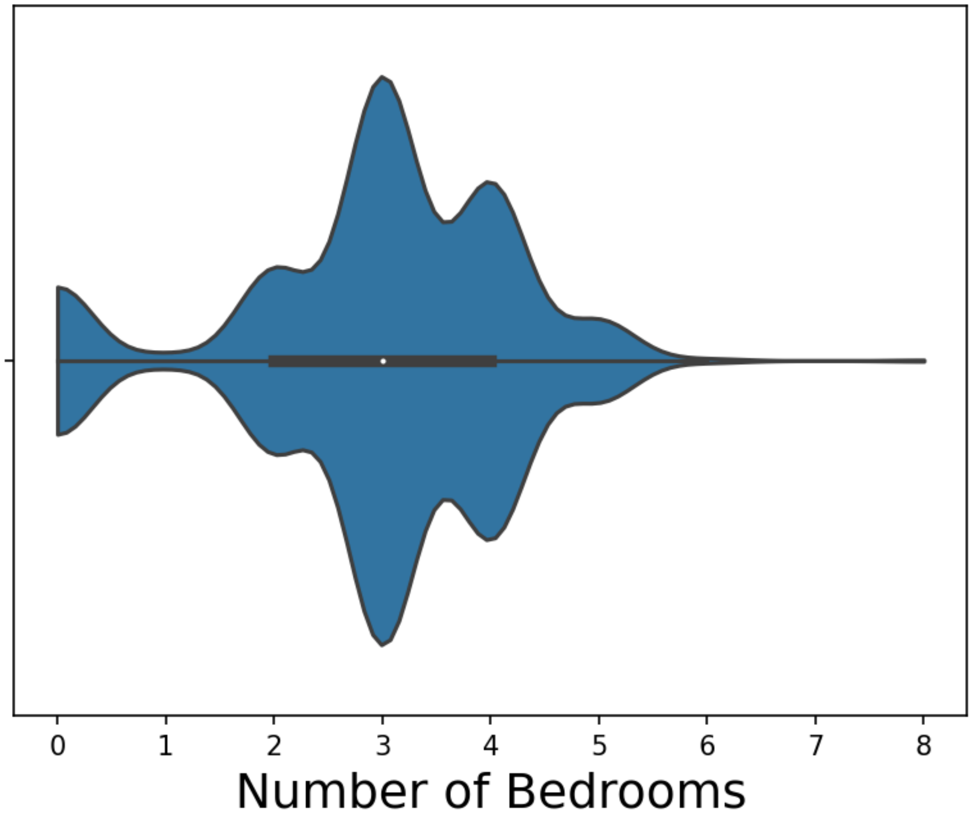
Let's also look at this on a box plot and violin plot.

[15]:
```
beds_plot = sns.boxplot(x=df.beds)
beds_plot.set(xlabel="Number of Bedrooms");
```



Here we can see the 8 bedroom house is definitely an outlier and should be filtered out of the final results.

[16]:
```
beds_plot = sns.violinplot(x=df.beds, cut=0)
beds_plot.set(xlabel="Number of Bedrooms");
```



The violin plot doesn't tell us anything new. It does look very wavy, for whatever that is worth.

Let us do another aggregate analysis on the table but with the bad data and outliers excluded.

[17]:
```
df.beds[lambda beds: (beds > 0) & (beds < 8)].describe()
```

[17]:
```
count    876.000000
mean       3.264840
std        0.850248
min        1.000000
25%        3.000000
50%        3.000000
75%        4.000000
max        6.000000
Name: beds, dtype: float64
```

We can see that the mean and standard deviation are now more reasonable, 3.2 ± 0.9, and a range of 1 to 6.

1.7 Baths Analysis

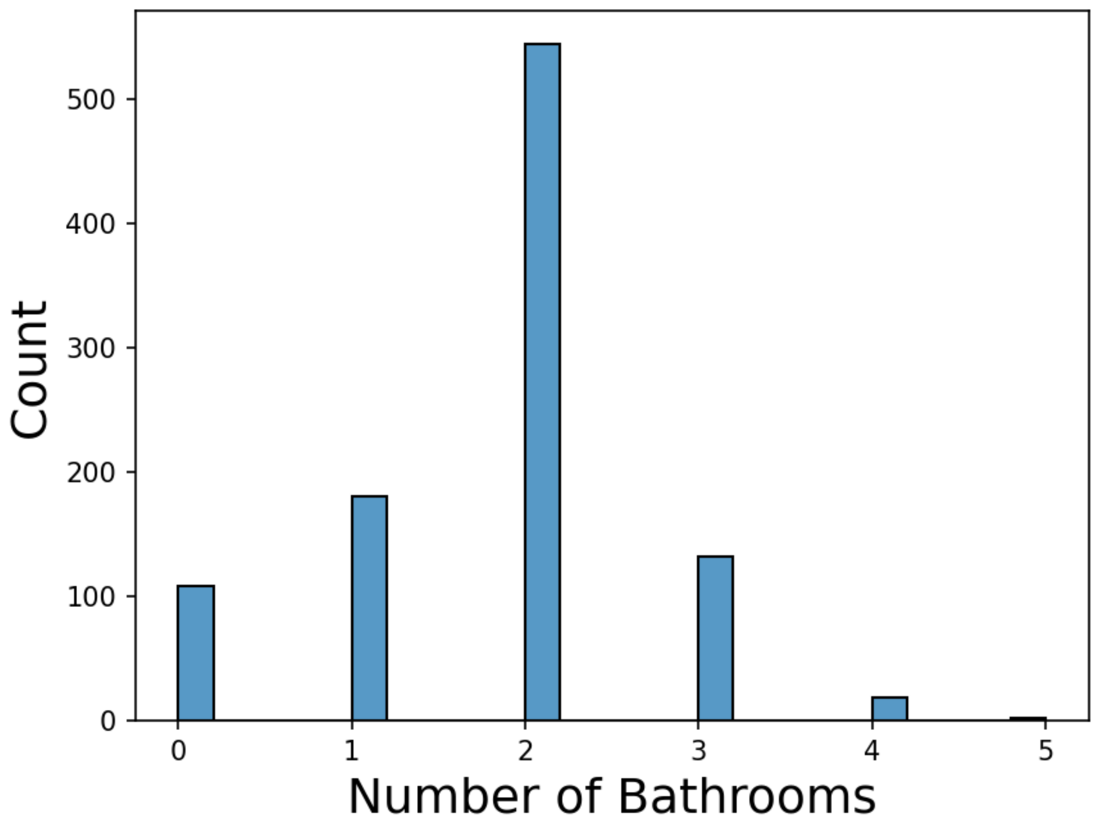
Let describe the baths column to get an idea of the aggregate statistics.

[18]:
```
df.baths.describe()
```

[18]:
```
count    985.000000
mean       1.776650
std        0.895371
min        0.000000
25%        1.000000
50%        2.000000
75%        2.000000
max        5.000000
Name: baths, dtype: float64
```

The min value of 0 looks strange. How does this look when shown on a histogram plot?

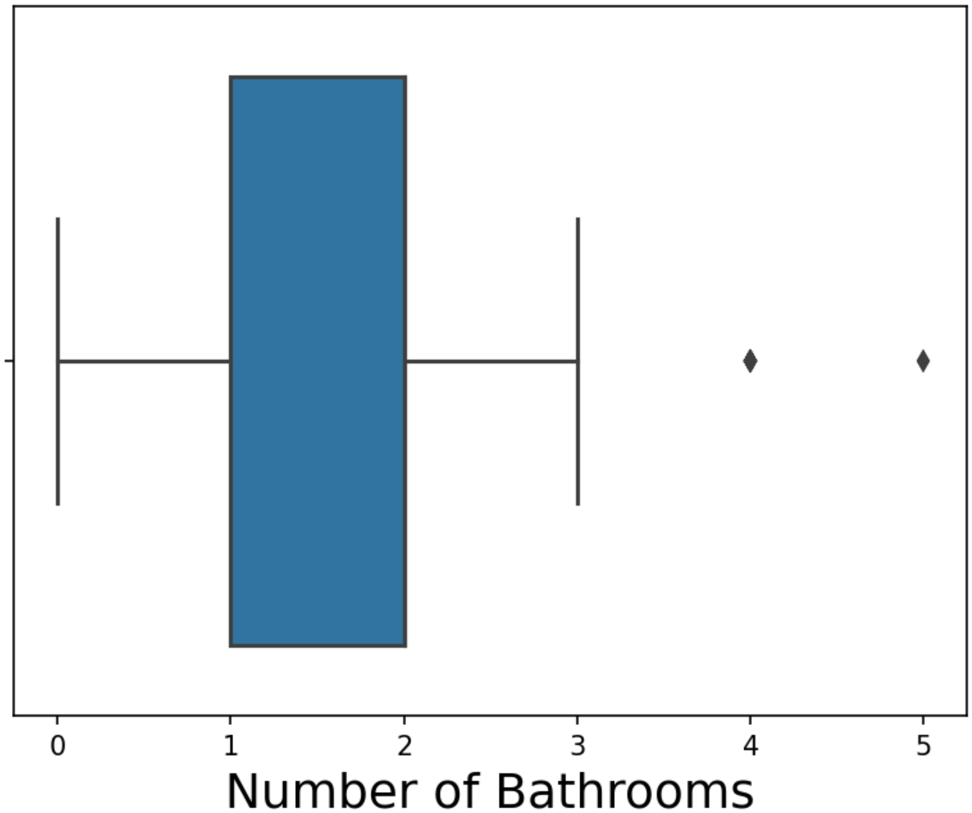
[19]:
```
baths_plot = sns.histplot(df.baths)
baths_plot.set(xlabel="Number of Bathrooms");
```



We have a somewhat power law distribution between 1 and 5. The zero values are definitely wrong.

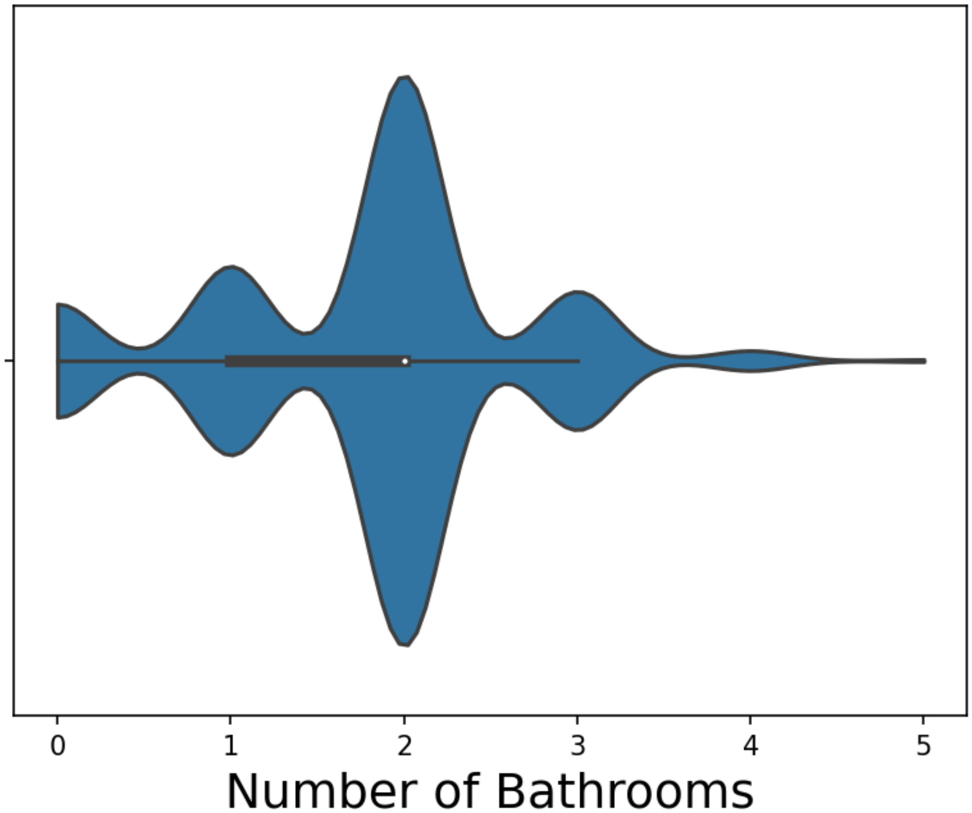
Let's also look at this on a box plot and violin plot.

[20]:
```
baths_plot = sns.boxplot(x=df.baths)
baths_plot.set(xlabel="Number of Bathrooms");
```



This plot makes the 4 and 5 bathroom properties seem like outliers. However, those are pefectly sensible but rare values so we will not be excluding them.

[21]:
```
baths_plot = sns.violinplot(x=df.baths, cut=0)
baths_plot.set(xlabel="Number of Bathrooms");
```



The violin plot doesn't tell us anything new. It does look very wavy, for whatever that is worth.

Let us do another aggregate analysis on the table but with the bad data and outliers excluded.

[22]:
```
df.baths[lambda baths: baths > 0].describe()
```

[22]:
```
count    877.000000
mean       1.995439
std        0.680771
min        1.000000
25%        2.000000
50%        2.000000
75%        2.000000
max        5.000000
Name: baths, dtype: float64
```

We can see that the mean and standard deviation are now more reasonable, 2.0 ± 0.7, and a range of 1 to 5.

1.8 Square Footage Analysis

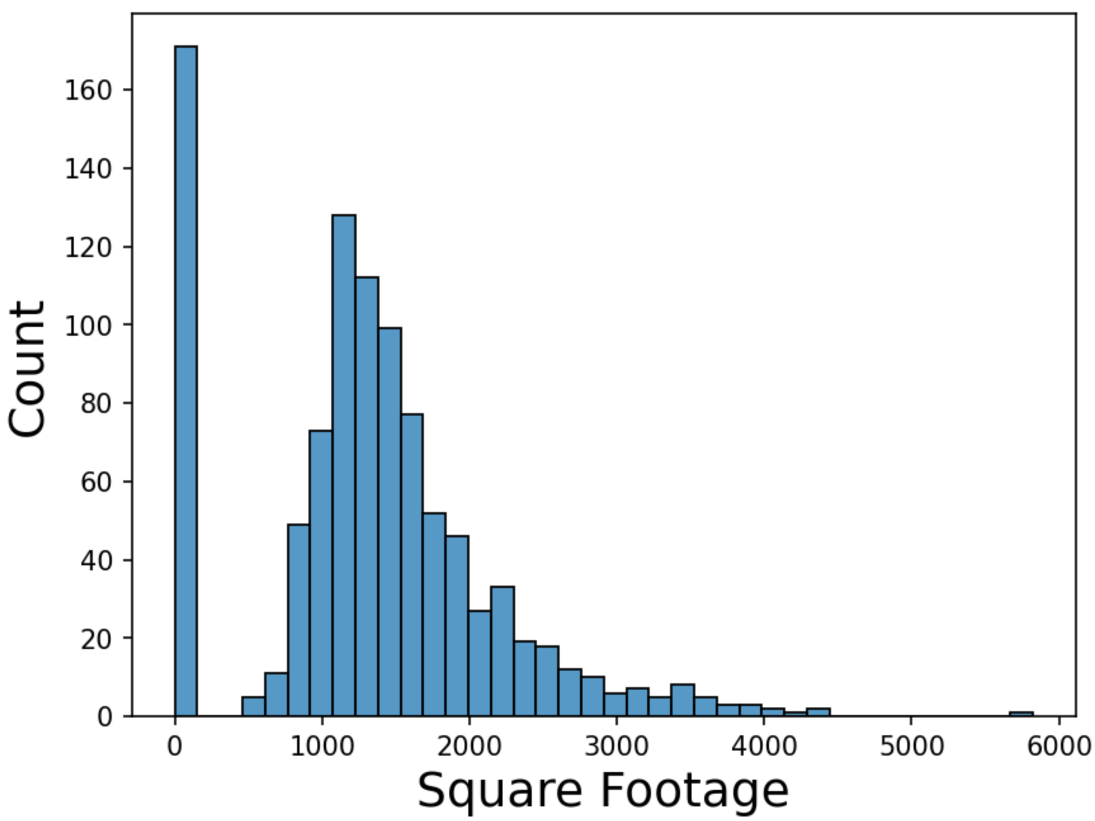
Let us look at the aggregate statistics for the sq__ft column.

[23]:
```
df.sq__ft.describe()
```

[23]:
```
count     985.000000
mean     1314.916751
std       853.048243
min         0.000000
25%       952.000000
50%      1304.000000
75%      1718.000000
max      5822.000000
Name: sq__ft, dtype: float64
```

That min value of zero looks odd. Let is look at a histogram plot.

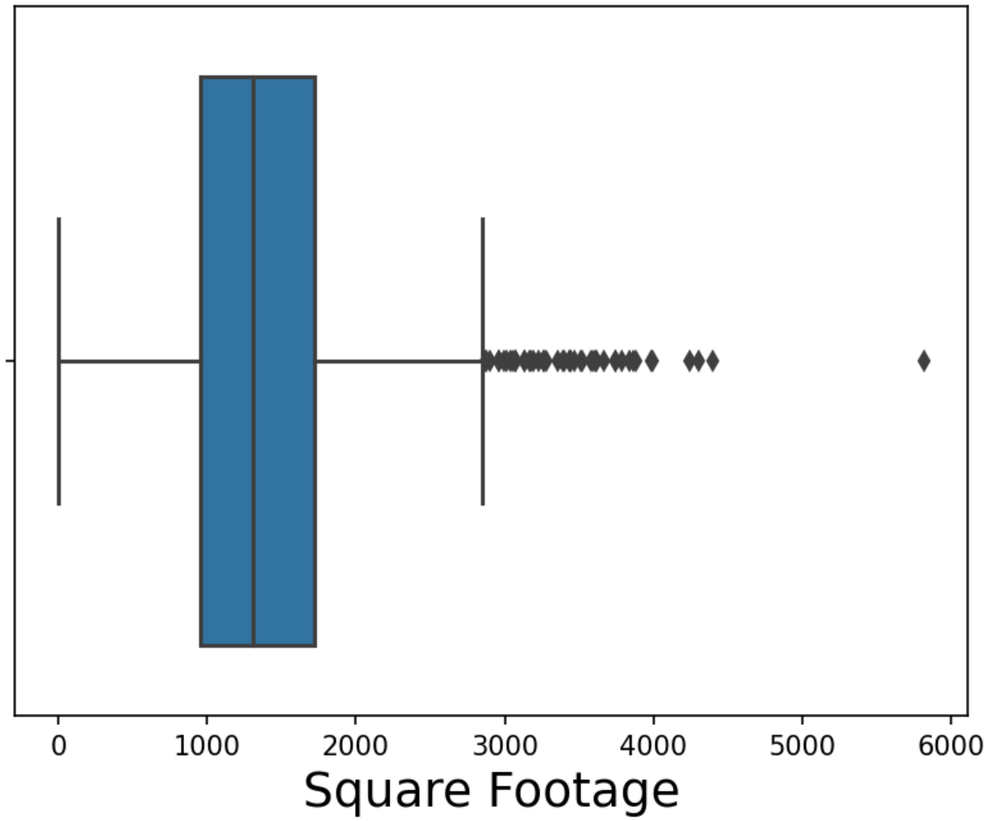
[24]:
```
sq_ft_plot = sns.histplot(df.sq__ft)
sq_ft_plot.set(xlabel="Square Footage");
sq_ft_plot.figure.savefig("m1p1b.sqft.png")
```



We can see there is a large number of properties that are missing the square footage. These values should be excluded in the final results. The rest of the data appears to be a power law distribution.

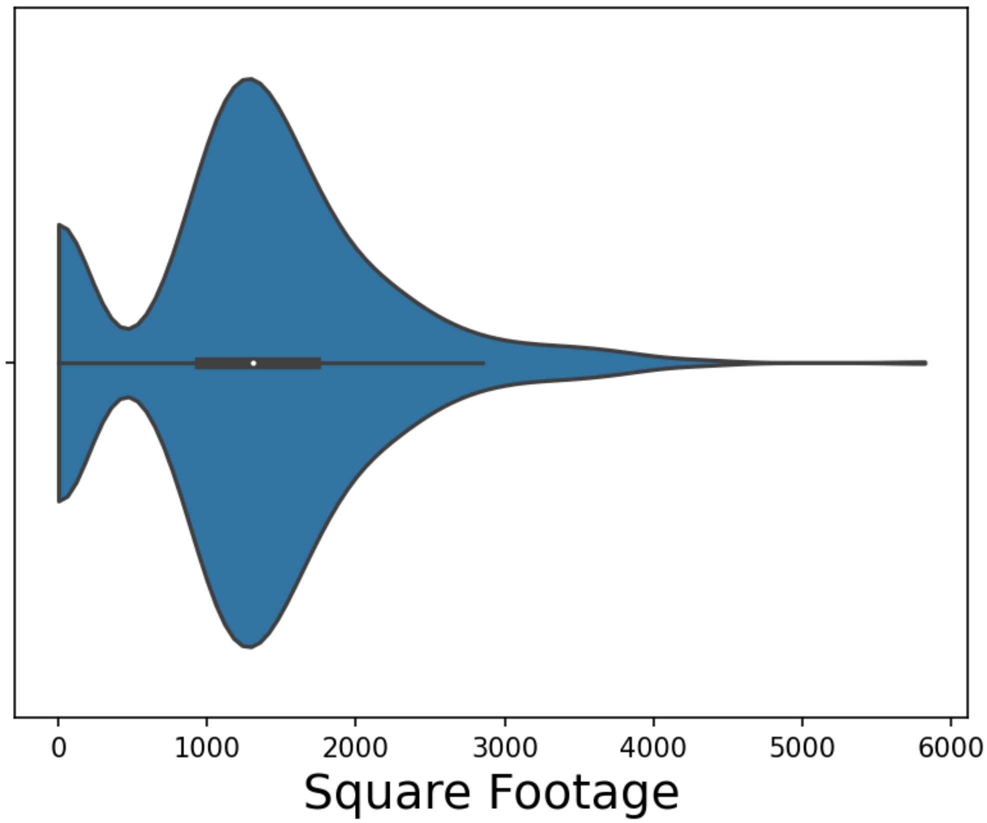
Let us look at a box plot and violin plot too.

[25]:
```python
sq_ft_plot = sns.boxplot(x=df.sq__ft)
sq_ft_plot.set(xlabel="Square Footage");
```



This has quite a few outliers above 3000 square feet. However, only that single outlier around 6000 square feet should probably be ignored.

[26]:
```python
sq_ft_plot = sns.violinplot(x=df.sq__ft, cut=0)
sq_ft_plot.set(xlabel="Square Footage");
```



This shows a pretty smooth violin plot (ignoring the lump around zero).

Let us do a final aggregate analysis on the set of data we plan to keep.

[27]:
```
filtered_sq_ft = df.sq__ft[lambda sq__ft: (sq__ft > 0) & (sq__ft < 5000)]
filtered_sq_ft.describe()
```

[27]:
```
count     813.000000
mean     1585.942189
std       647.423526
min       484.000000
25%      1144.000000
50%      1418.000000
75%      1851.000000
max      4400.000000
Name: sq__ft, dtype: float64
```

We have a range of 1586 ± 647 and a range of 484 to 4400.

Let us look at the smallest remaining values.

[28]:
```
filtered_sq_ft.sort_values().head(10).to_list()
```

[28]: [484, 539, 588, 610, 611, 623, 625, 682, 696, 722]

Those all still seem pretty small. However, we will assume they are correct and leave them in.

Now let us look at the largest remaining values.

[29]:
```
filtered_sq_ft.sort_values().tail(10).to_list()
```

[29]: [3746, 3788, 3838, 3863, 3881, 3984, 3992, 4246, 4303, 4400]

Those all look pretty reasonable (as far as large properties are concerned).

1.9 Price Analysis

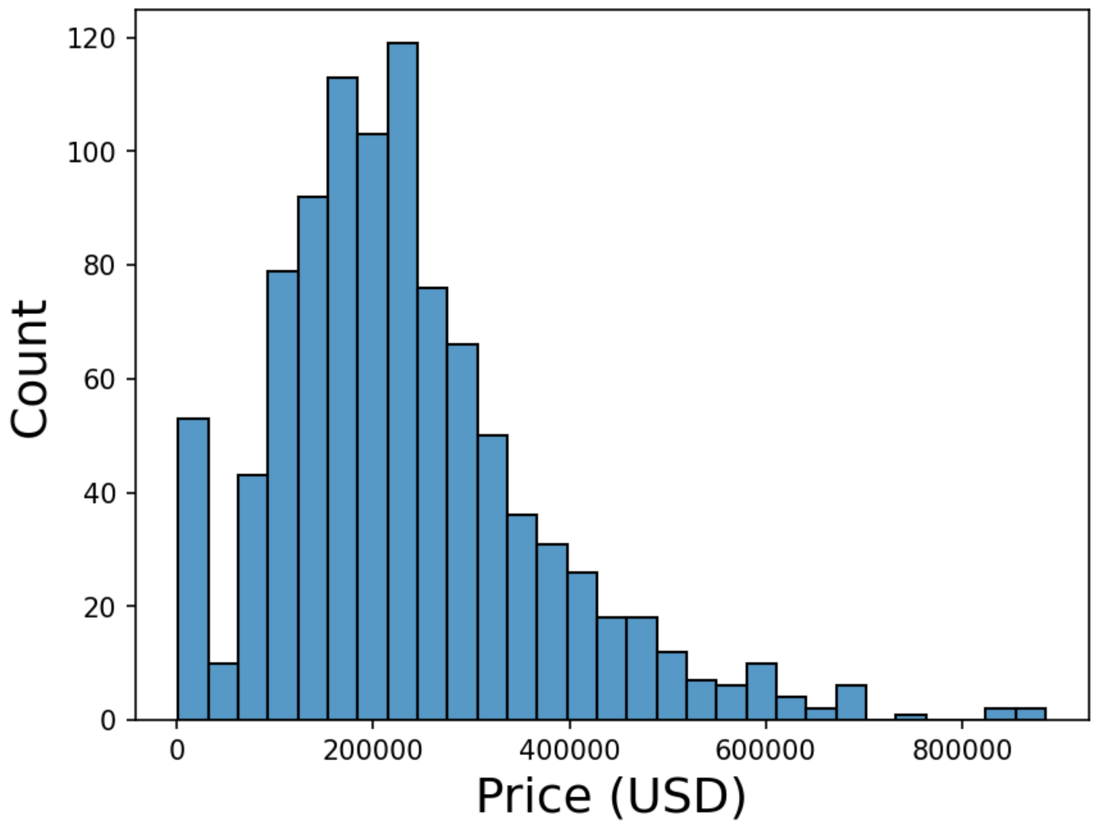
Let us take a look at the aggregate statistics for the price column.

[30]:
```
df.price.describe()
```

[30]:
```
count       985.000000
mean     234144.263959
std      138365.839085
min        1551.000000
25%      145000.000000
50%      213750.000000
75%      300000.000000
max      884790.000000
Name: price, dtype: float64
```

We can see that there is definitely some bad data on the lower end (properties don't cost $1500). The upper end doesn't look incorrect but it does seem like it might be an outlier. Let us look at this data on a histogram.

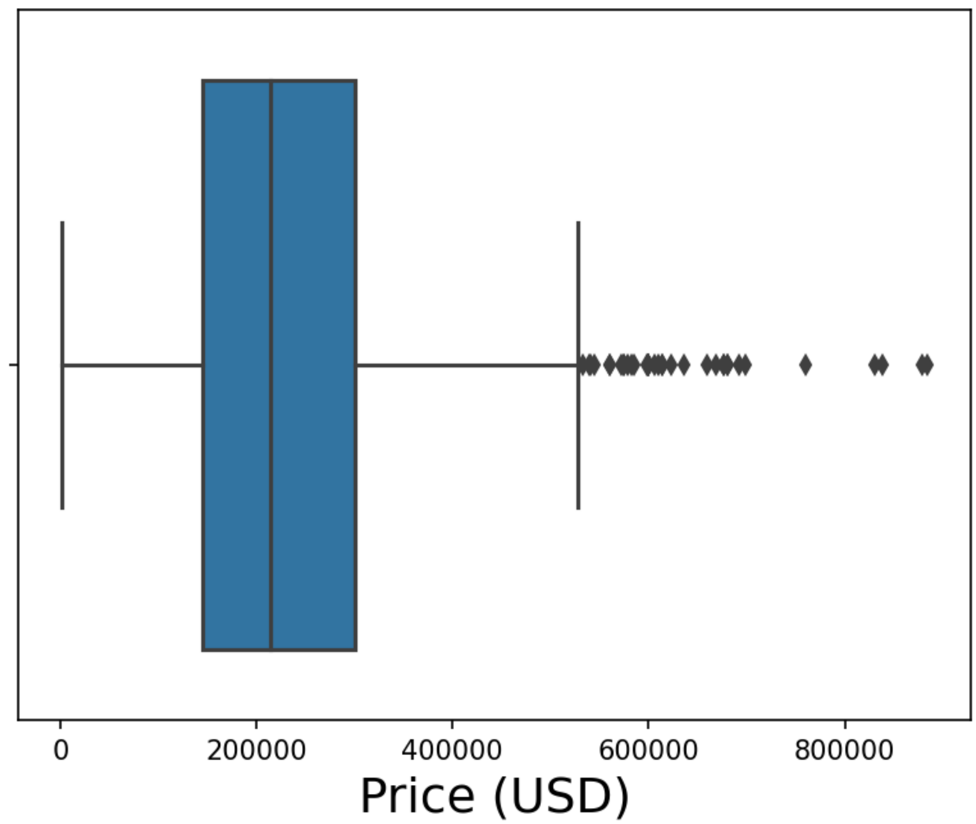
[31]:
```
price_plot = sns.histplot(df.price)
price_plot.set(xlabel="Price (USD)");
price_plot.figure.savefig("m1p1b.price.png")
```



We can see some bad data around zero, but also some data that is likely bad below $50k. The $800k+ properties seem like outliers but the graph also indicates this is a power law distributation.

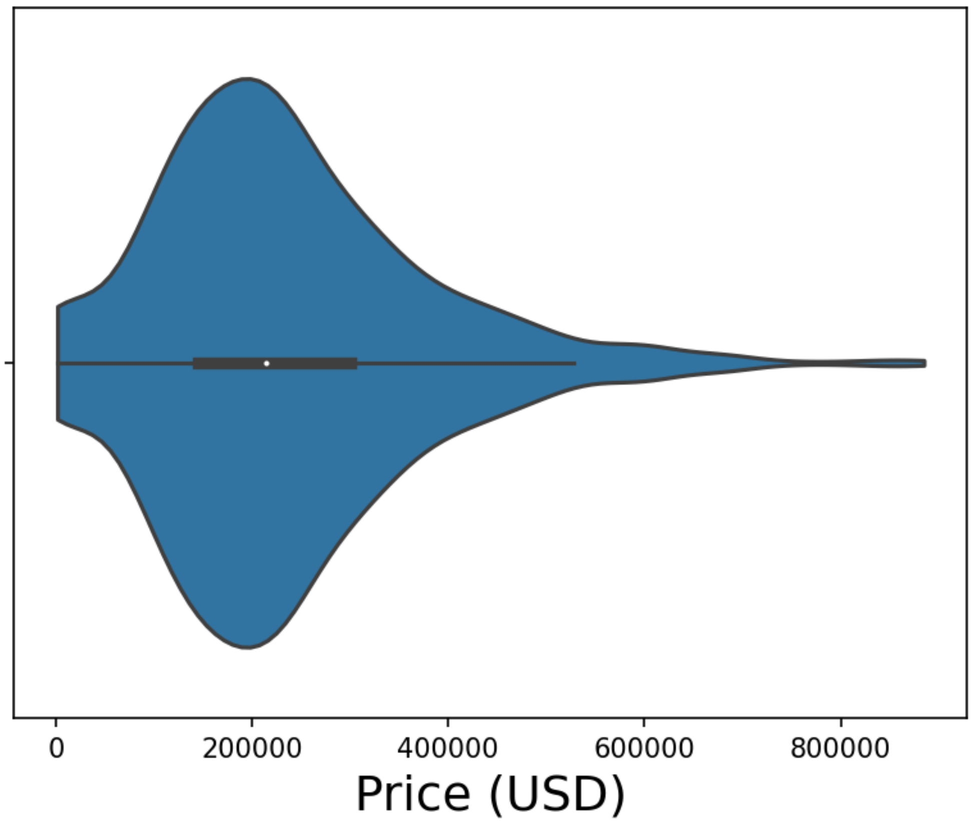
Let us look at a box plot and violin plot now.

[32]:
```
price_plot = sns.boxplot(x=df.price)
price_plot.set(xlabel="Price (USD)");
```



We can see a bunch of what look like outliers above $500k. However, that doesn't look like enough to exclude anything.

[33]:
```
price_plot = sns.violinplot(x=df.price, cut=0)
price_plot.set(xlabel="Price (USD)");
```



The violin plot doesn't give us any further insights.

Let us get a better look at the smallest values.

[34]:
```
df.price.sort_values().head(10).to_list()
```

[34]: [1551, 2000, 4897, 4897, 4897, 4897, 4897, 4897, 4897, 4897]

There are some small values and quite a few 4897 values. Let take a another look after filtering those out.

[35]:
```
df.price[lambda value: value > 5000].sort_values().head(10)
```

[35]:
```
603     30000
604     30000
335     40000
336     48000
605     55422
867     56950
0       59222
868     60000
869     61000
```

```
337    61500
Name: price, dtype: int64
```

These still seem pretty inexpensive but not enough to exclude them.

Now let us look at the largest values.

[36]:
```
df.price.sort_values().tail(10)
```

[36]:
```
551    677048
862    680000
332    680000
552    691659
333    699000
553    760000
157    830000
334    839000
863    879000
864    884790
Name: price, dtype: int64
```

We don't see any obvious errors. These large house prices don't necessarily need to be removed.

Let us look at the aggregate statistics after filtering out the small values.

[37]:
```
df.price[lambda price: price > 5000].describe()
```

[37]:
```
count       934.000000
mean     246668.732334
std      130991.448400
min       30000.000000
25%      156000.000000
50%      220000.000000
75%      305000.000000
max      884790.000000
Name: price, dtype: float64
```

The min is still pretty small (30000) and the standard deviation is still pretty large ($130k). Let us see what those numbers would look like if we excluded the most expensive properties.

[38]:
```
df.price[lambda price: (price > 5000) & (price < 800_000)].describe()
```

[38]:
```
count       930.000000
mean     244038.501075
std      124952.156754
min       30000.000000
25%      156000.000000
50%      220000.000000
75%      303750.000000
max      760000.000000
```

```
Name: price, dtype: float64
```

They did not significantly change. We will only filter out the smaller values and leave in those expensive properties.

1.10 Latitude Analysis

Let us look at the aggregate statistics for latitude.

[39]:
```
df.latitude.describe()
```

[39]:
```
count    985.000000
mean      38.607732
std        0.145433
min       38.241514
25%       38.482717
50%       38.626582
75%       38.695589
max       39.020808
Name: latitude, dtype: float64
```

We can see we have a tight clustering of latitude values. This seems like good data.

[40]:
```
df.latitude.sort_values().head(10)
```

[40]:
```
174    38.241514
372    38.242270
820    38.247659
63     38.251808
508    38.253500
761    38.258976
957    38.259708
61     38.260443
409    38.260467
189    38.270617
Name: latitude, dtype: float64
```

The smallest values all look good.

[41]:
```
df.latitude.sort_values().tail(10)
```

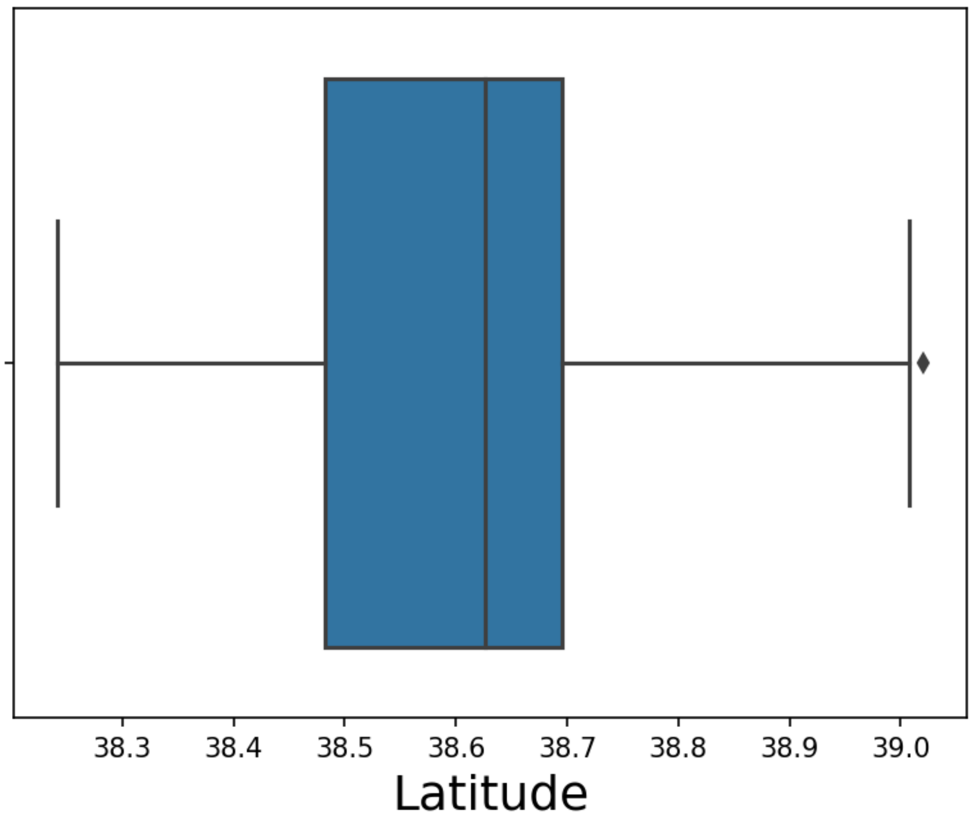
[41]:
```
976    38.897814
750    38.899180
828    38.904869
778    38.905927
468    38.931671
833    38.935579
484    38.939802
142    38.945357
```

```
242    39.008159
686    39.020808
Name: latitude, dtype: float64
```

The largest values all look good.

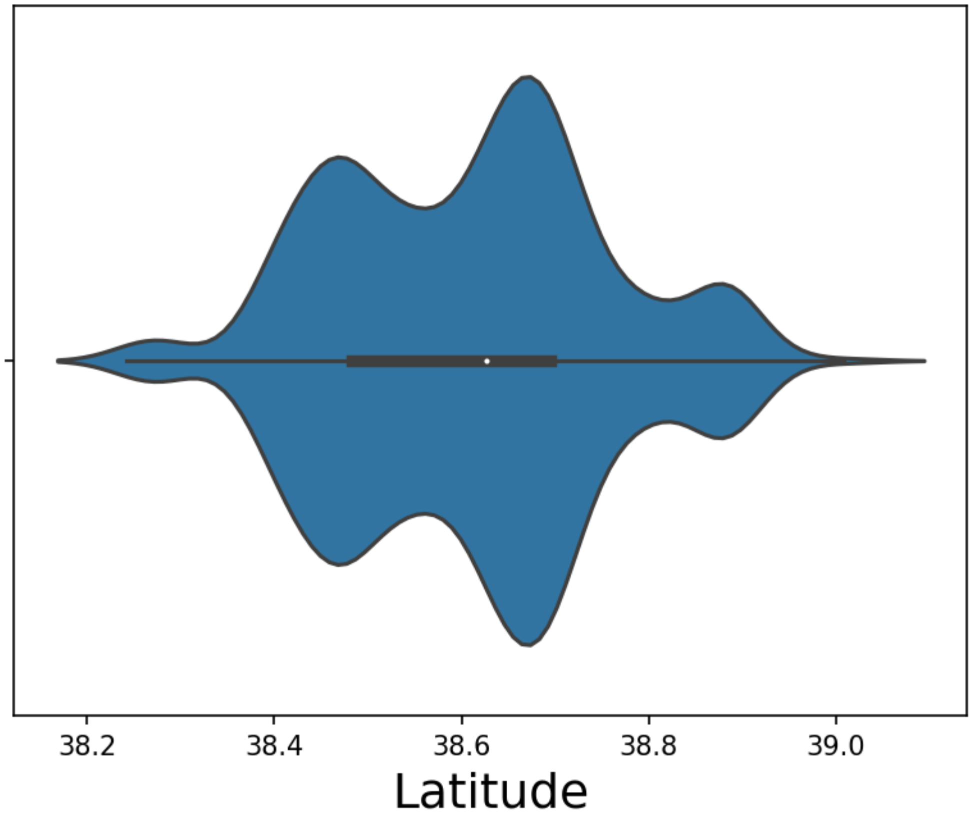
Let us look at a box plot and a violin plot.

[42]:
```
lat_plot = sns.boxplot(x=df.latitude)
lat_plot.set(xlabel="Latitude");
```



We see there might be one outlier around 39. We will leave this unless we see something else that catches our eye.

[43]:
```
lat_plot = sns.violinplot(x=df.latitude)
lat_plot.set(xlabel="Latitude");
```



The violin plot doesn't give us any more information.

1.11 Longitude Analysis

Let us look at the aggregate statistics for the longitude column.

[44]:
```
df.longitude.describe()
```

[44]:
```
count     985.000000
mean     -121.355982
std         0.138278
min      -121.551704
25%      -121.446127
50%      -121.376220
75%      -121.295778
max      -120.597599
Name: longitude, dtype: float64
```

We can see we have a tight clustering of values. This all seems good.

[45]:
```
df.longitude.sort_values().head(10)
```

[45]:
```
310   -121.551704
445   -121.550527
117   -121.549521
318   -121.549437
446   -121.549049
787   -121.547664
98    -121.547572
144   -121.545947
286   -121.545490
101   -121.544023
Name: longitude, dtype: float64
```

The smallest values look good.

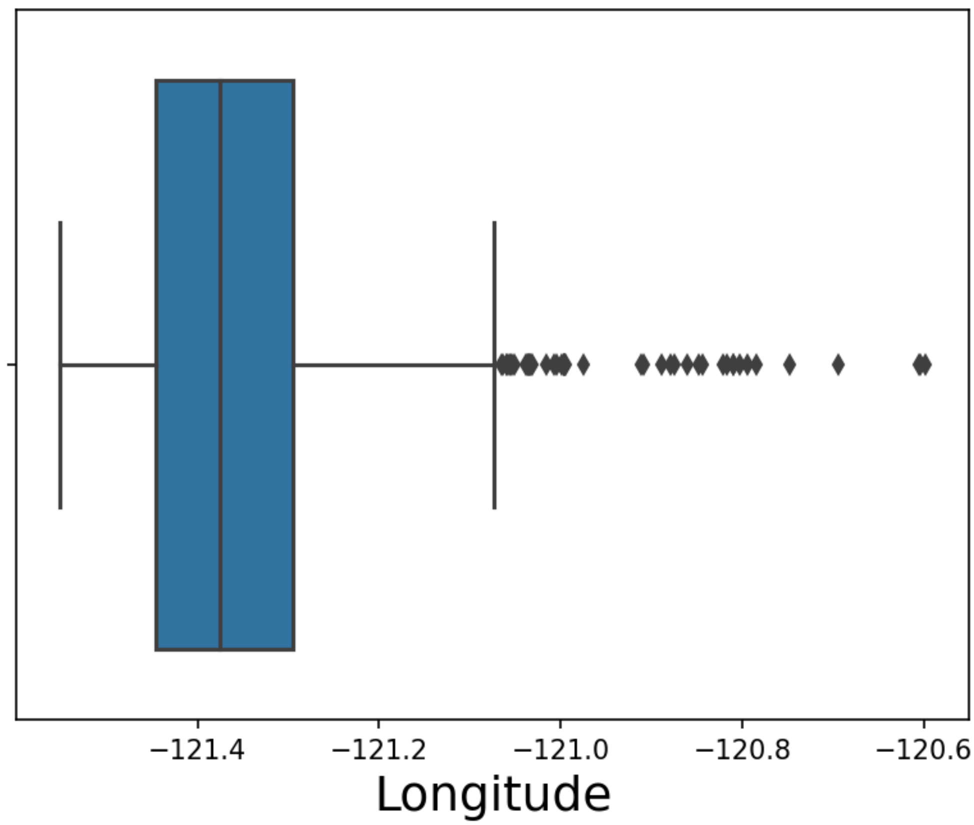
[46]:
```
df.longitude.sort_values().tail(10)
```

[46]:
```
709   -120.810235
754   -120.809254
771   -120.802458
227   -120.794254
297   -120.784145
518   -120.748039
844   -120.693641
106   -120.604760
102   -120.603872
663   -120.597599
Name: longitude, dtype: float64
```

The largest values look good.

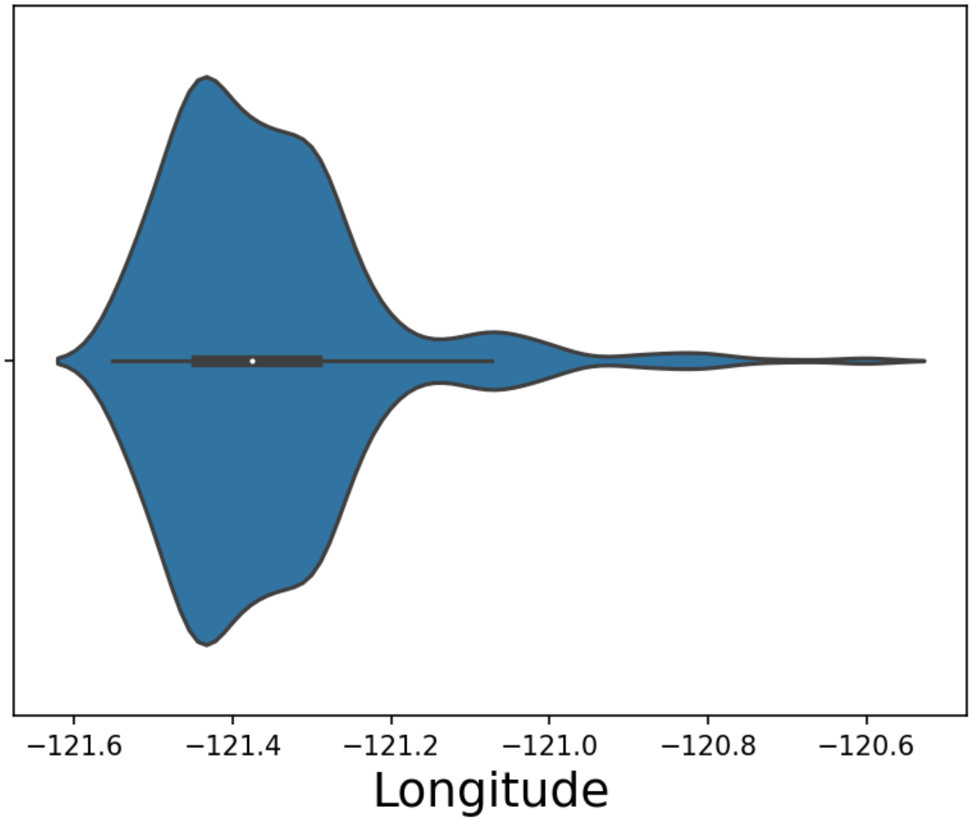
Let us look at a box plot and violin plot.

[47]:
```
long_plot = sns.boxplot(x=df.longitude)
long_plot.set(xlabel="Longitude");
```



We can see this make it look like a lot of the longitude values are outliers.

[48]:
```
long_plot = sns.violinplot(x=df.longitude)
long_plot.set(xlabel="Longitude");
```

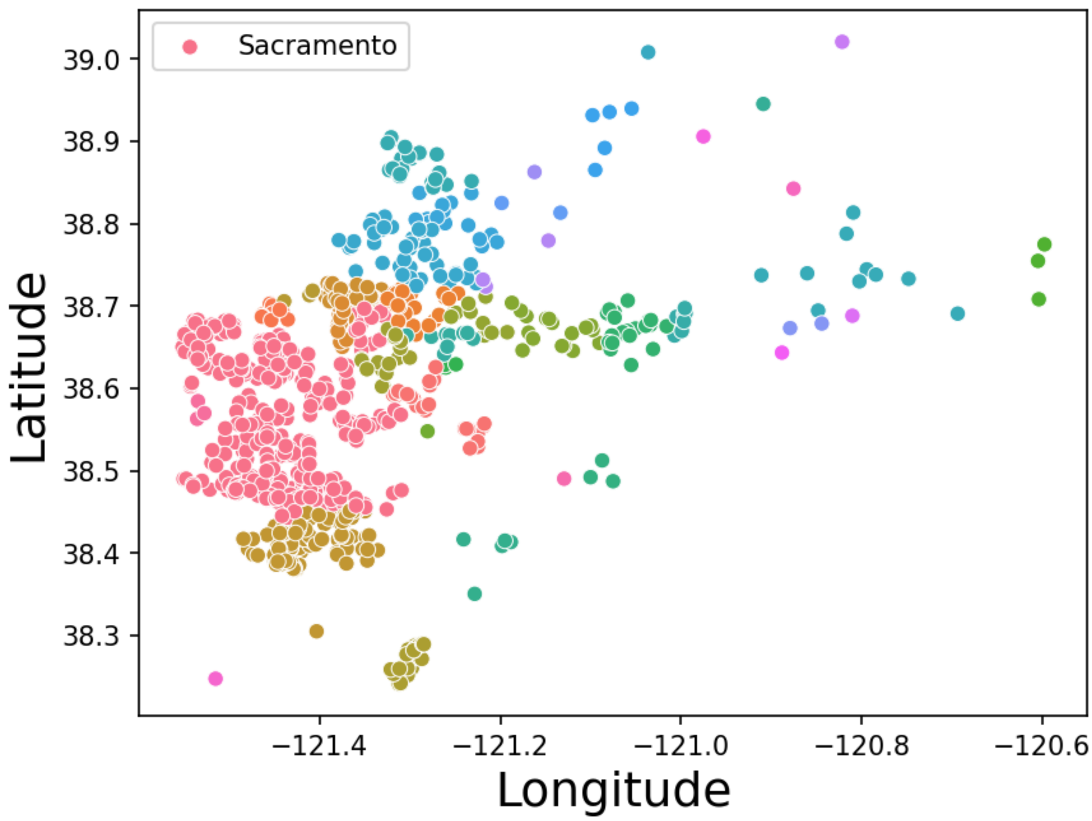


The violin plot also gives us a sense there might be tail of data.

1.12 Latitude and Longitude Combined Analysis

Since points of latitude and longitude are two components of one piece of information, the location of a property, let us look at a scatter plot of them (colored by City).

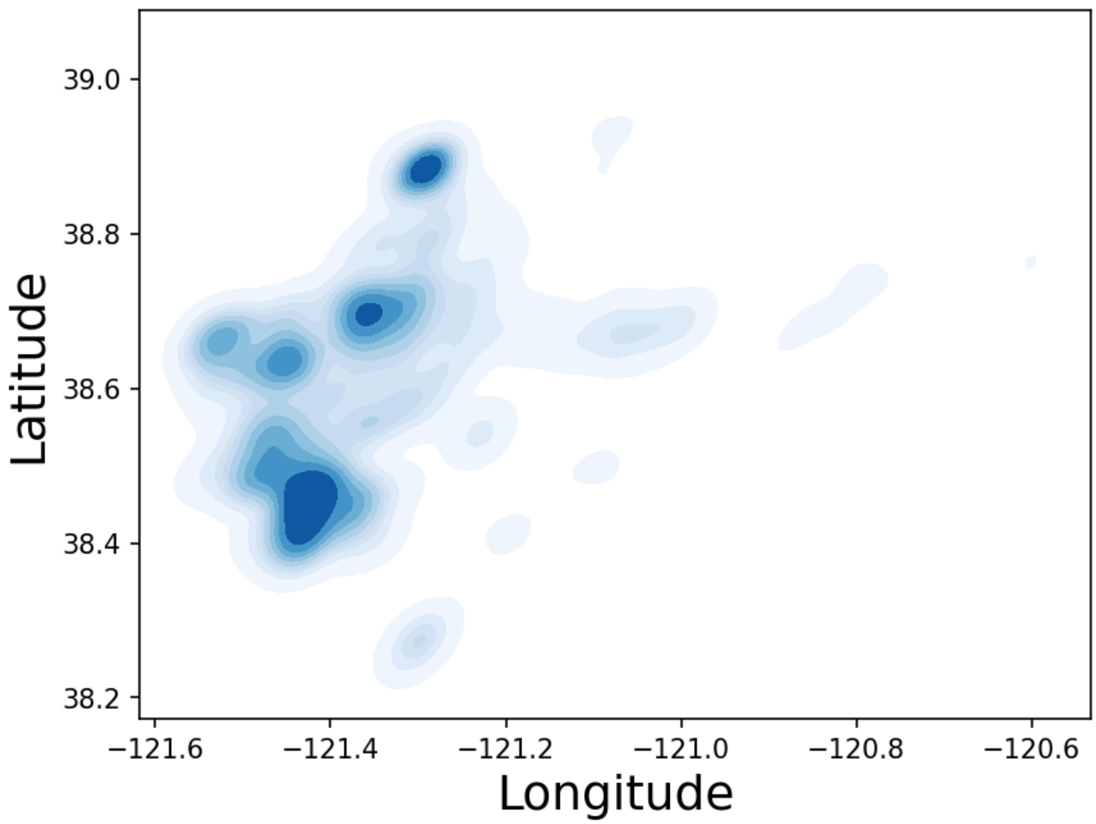
[49]:
```
lat_long_plot = sns.scatterplot(data=df, x="longitude", y="latitude",␣
  ↪hue="city")
lat_long_plot.legend(labels=["Sacramento"])
lat_long_plot.set(xlabel="Longitude", ylabel="Latitude");
lat_long_plot.figure.savefig("m1p1b.latlonscatter.png")
```



We can see that we have some tight clusters and what look like a lot of less centralized locations.

Let us see this as a kernel density estimate plot instead.

[50]:
```
lat_long_plot = sns.kdeplot(x=df.longitude, y=df.latitude, cmap="Blues",
    fill=True, bw_adjust=0.5)
lat_long_plot.set(xlabel="Longitude", ylabel="Latitude");
lat_long_plot.figure.savefig("m1p1b.latlonkde.png")
```



We can see there are three dense regions of properties. These are relatively closely clustered and a few more remote regions included. Some of these more remote region could be excluded, but without a supplementary data source for the bounding latitude longitude of Sacramento proper there isn't a clean way to exclude them. It does make for an intersting plot though.

1.13 Price per Square Foot Analysis

We have analyze the price and square footage independently, but let us now analyze them together to make sure they make sense with respect to each other.

[51]:
```
price_per_sq_ft = df.price / df.sq__ft
price_per_sq_ft.describe()
```

[51]:
```
count    985.000000
mean            inf
std             NaN
min        0.343525
25%      114.142628
50%      149.253731
75%      213.178295
max             inf
```

```
dtype: float64
```

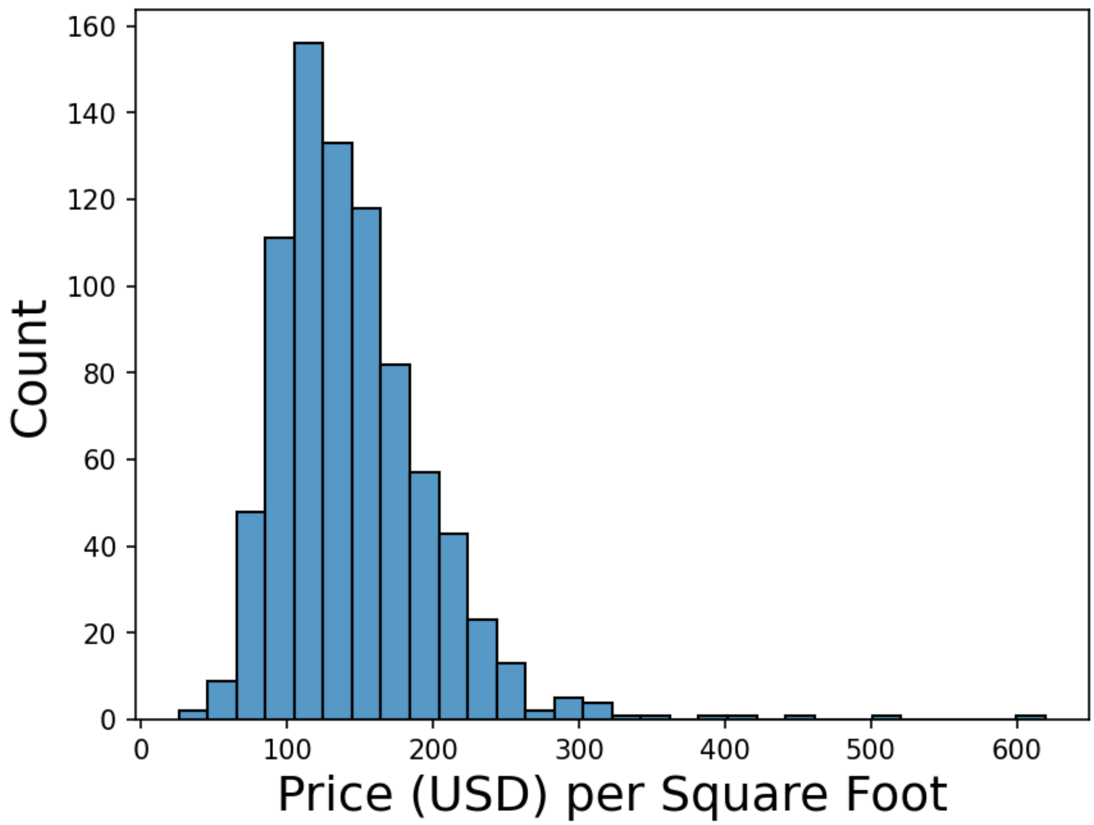
We can see that we have some obvious bad data (due to divide by zero for the square footage and extremely low prices). Let us filter out those values and look at the aggregate statistics one more time.

[52]:
```
filtered_price_per_sq_ft = price_per_sq_ft[lambda value: (value > 1) & (value <␣
↪1_000)]
filtered_price_per_sq_ft.describe()
```

[52]:
```
count    813.000000
mean     145.852010
std       54.636882
min       25.728988
25%      109.176748
50%      137.152778
75%      170.438670
max      619.666048
dtype: float64
```

We can see a much more reasaonable range, 146 ± 55. However, the max of 619 looks a bit high. Let us visualize this on a histogram plot.

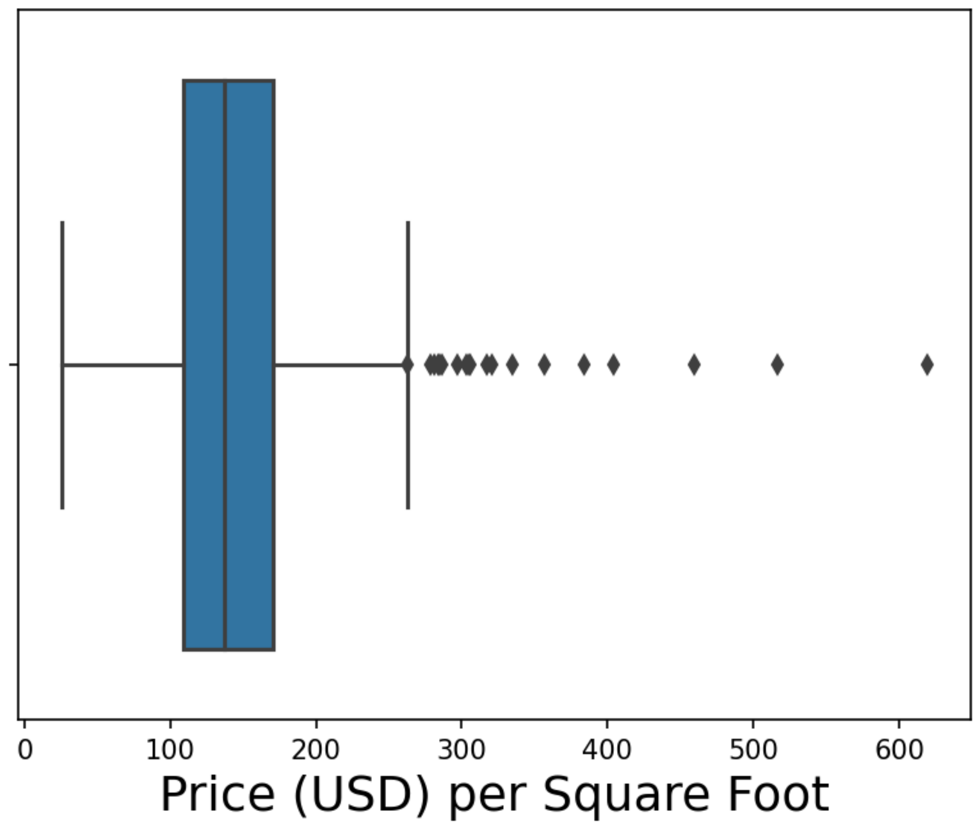
[53]:
```
per_sq_ft_plot = sns.histplot(filtered_price_per_sq_ft, bins=30)
per_sq_ft_plot.set(xlabel="Price (USD) per Square Foot");
per_sq_ft_plot.figure.savefig("m1p1b.ppsqft.png")
```



This looks like a power law distribution. Though there might be a lot of outliers towards the more expensive side.

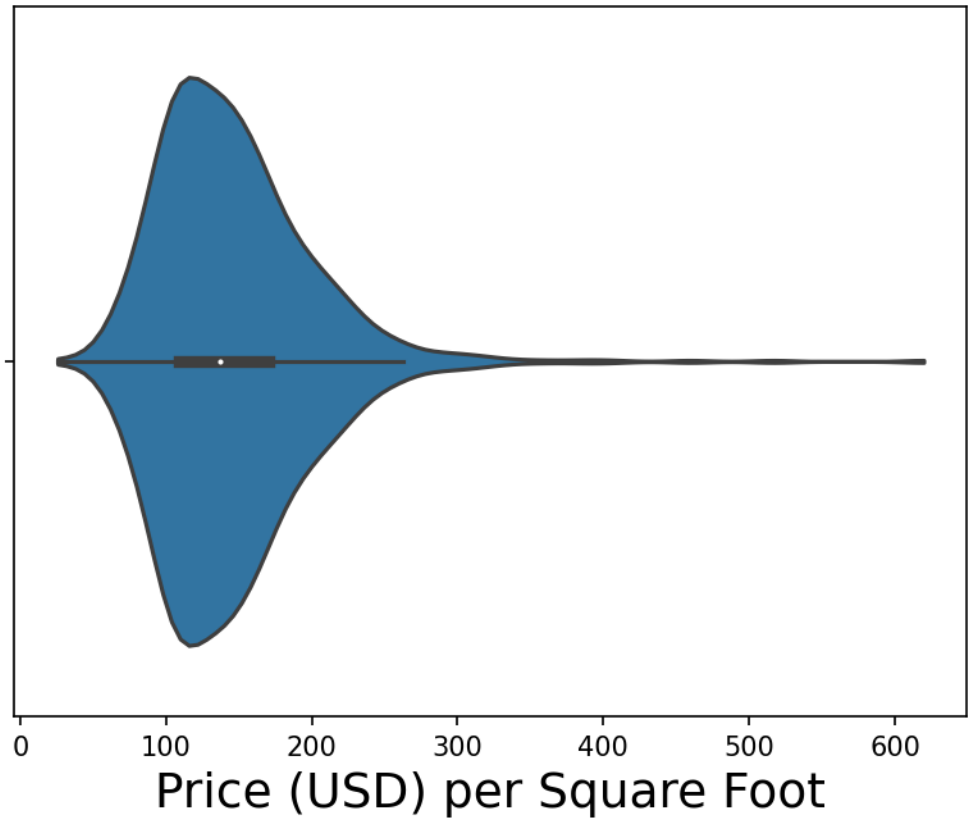
Let us look at a box plot and violin plot.

[54]:
```
per_sq_ft_plot = sns.boxplot(x=filtered_price_per_sq_ft)
per_sq_ft_plot.set(xlabel="Price (USD) per Square Foot");
```



There appear to be quite a few outliers past the 300 mark. It seems fairly arbitrary but we will exclude these values from our final analysis.

[55]:
```
per_sq_ft_plot = sns.violinplot(x=filtered_price_per_sq_ft, cut=0)
per_sq_ft_plot.set(xlabel="Price (USD) per Square Foot");
```



The violin plot agree with the outliers past 300.

1.14 Sale Date Analysis

Let us take a look at the sale_date column. It only had 5 distinct values from the original analysis.

[56]:
```
df.sale_date.unique()
```

[56]:
```
array(['Wed May 21 00:00:00 EDT 2008', 'Tue May 20 00:00:00 EDT 2008',
       'Mon May 19 00:00:00 EDT 2008', 'Fri May 16 00:00:00 EDT 2008',
       'Thu May 15 00:00:00 EDT 2008'], dtype=object)
```

We can see a consistent formatting for the dates so let us clean those up for our final table. Also, all of the values look correct: between 5/15 and 5/21 in 2008. Hopefully we don't mind that all of our transactions are 15 years old.

1.15 Cleaning and Saving

Now that we have finished our analysis of all of the columns in the table we can finally filter out the data we don't want and then convert our columns over to categories. Note that I am doing the category conversion after the filtering so the filtered out values don't show up as categories.

[57]:

```python
clean_df = df\
    .merge(zip_counts.rename("zip_count"), left_on="zip", right_index=True)\
    .merge(city_counts.rename("city_count"), left_on="city", right_index=True)\
    .merge(price_per_sq_ft.rename("price_per_sq_ft"), left_index=True,␣
↪right_index=True)\
    [lambda row:
        (row.type != "Unkown") & # Ignore a listing with type Unkown (both a␣
↪typo and indicates we don't know the type)
        (row.zip_count > 1) & # Ignore a listing if it is the only one in a zip␣
↪code
        (row.city_count > 1) & # Ignore a listing if it is the only one in a␣
↪city
        (row.beds > 0) & # Ignore listings with no beds
        (row.beds < 8) & # Ignore the outlier with 8 beds
        (row.baths > 0) & # Ignore listings with no baths
        (row.sq__ft > 0) & # Ignore listings missing the square footage
        (row.sq__ft < 5000) & # Ignore listings over 5k sq. ft. (only one␣
↪listing)
        (row.price >= 5000) & # Ignore listings below $5k (unreasonable price)
        (row.price_per_sq_ft > 1) & # Ignore prices per sq. ft. below $1
        (row.price_per_sq_ft < 300) # Ignore prices per sq. ft. over $300 (not␣
↪enough samples)
    ]\
    [df.columns]\
    .reset_index()\
    .copy()
clean_df.city = clean_df.city.apply(lambda name: name.upper()).
↪astype("category")
clean_df.zip = clean_df.zip.apply(lambda zip: f"{zip:05d}").astype("category")
clean_df.state = clean_df.state.astype("category")
clean_df.type = clean_df.type.astype("category")
clean_df.sale_date = clean_df.sale_date.apply(lambda value: datetime.
↪strptime(value, "%a %B %d %H:%M:%S EDT %Y"))
clean_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 794 entries, 0 to 793
Data columns (total 13 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   index      794 non-null    int64
 1   address    794 non-null    object
 2   city       794 non-null    category
 3   zip        794 non-null    category
 4   state      794 non-null    category
 5   beds       794 non-null    int64
 6   baths      794 non-null    int64
```

```
 7   sq__ft     794 non-null    int64
 8   type       794 non-null    category
 9   sale_date  794 non-null    datetime64[ns]
 10  price      794 non-null    int64
 11  latitude   794 non-null    float64
 12  longitude  794 non-null    float64
dtypes: category(4), datetime64[ns](1), float64(2), int64(5), object(1)
memory usage: 63.1+ KB
```

We can see that our four columns are now category dtypes. Our final row count after all of our filters is 794 out of the original 985 (191 records being exclude for bad data, sample size too small, or outside reasonable ranges). This might have been a bit aggressive, but again, without knowing the final application of this data it is hard to tell.

Let us finally save the cleaned table back to a feather file.

[58]:
```
clean_df.to_feather("csc5610-m2-Sacramento-real-estate-transactions-cleaned.
 ↪feather")
```