Kaggle Challenge - Exploratory Data Analysis

WE'LL COVER THE FOLLOWING

Our project is based on the Kaggle Housing Prices Competition. In this challenge we are given a dataset with different attributes for houses and their prices. Our goal is to develop a model that can predict the prices of houses based on this data.

At this point we already know two things:

1. We are given labeled training examples
2. We are asked to predict a value

What do these tell us in terms of framing our problem? The first point tells us that this is clearly a typical **supervised learning** task, while the second one tells us that this is a typical **regression** task.

If we look at the file with data description, *data_description.txt*, we can see the kind of attributes we are expected to have for the houses we are working with. Here is a sneak peek into some of the interesting attributes and their description from that file:

- ***SalePrice*** – the property's sale price in dollars. This is the target variable that we are trying to predict.
- ***MSSubClass***: The building class.
- ***LotFrontage***: Linear feet of street connected to property.
- ***LotArea***: Lot size in square feet. Street: Type of road access.
- ***Alley***: Type of alley access.
- ***LotShape***: General shape of property.
- ***LandContour***: Flatness of the property.
- ***LotConfig***: Lot configuration.
- ***LandSlope***: Slope of property. Neighborhood: Physical locations within Ames city limits.
- ***Condition1***: Proximity to main road or railroad.
- ***HouseStyle***: Style of dwelling.
- ***OverallQual***: Overall material and finish quality.
- ***OverallCond***: Overall condition rating.

- ***YearBuilt***: Original construction date.

> 📌 **Note**: Before moving further, download the dataset, *train.csv*, from here. Launch your Jupyter notebook and then follow along! ***It is important to get your hands dirty; don't just read through these lessons!***

> 📌 **You can also find the Juptyter notebook with all the code for this project on my Git profile, here.**

> 📌 **You can see the live execution of code in the Jupyter Notebook at the end of the lesson and can also play with it.**

1. Exploratory Data Analysis

Importing modules and getting the data

Let's start by importing the modules and getting the data. In the code snippet below, it is assumed that you have downloaded the csv file and saved it in the working directory as *'./data/train.csv'*.

```
# Core Modules
import pandas as pd
import numpy as np

# Basic modules for data visualization
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

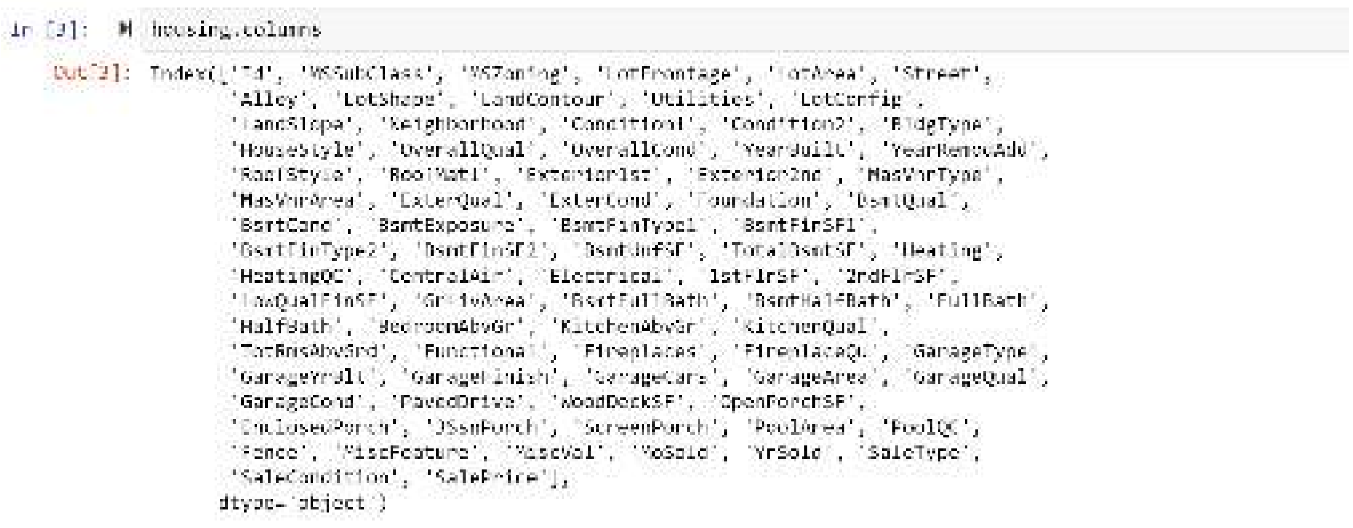
```
# Load data into a pandas DataFrame from given filepath
housing = pd.read_csv('./data/train.csv')
```

Understand the Data Structure

We now have our DataFrame in place, so let's get familiar with it by looking at the columns it contains:

```
# Get column names of the df
```

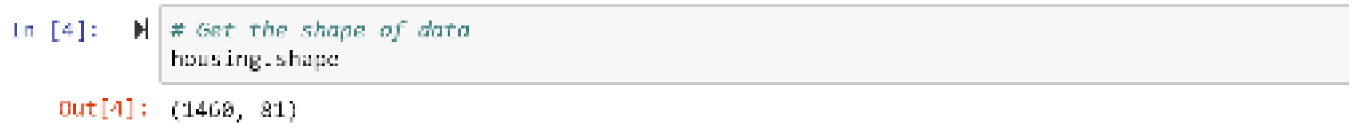
```
housing.columns
```



```
In [3]:  housing.columns
Out[3]: Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
               'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
               'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
               'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
               'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
               'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
               'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
               'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
               'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
               'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
               'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
               'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
               'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
               'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
               'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
               'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
               'SaleCondition', 'SalePrice'],
              dtype='object')
```

How many attributes do we have in total? Of course, we are not going to count them ourselves from the results above! Let's get the number of rows and columns in our DataFrame by calling `shape`.

```
# Get the Shape of data
housing.shape
```

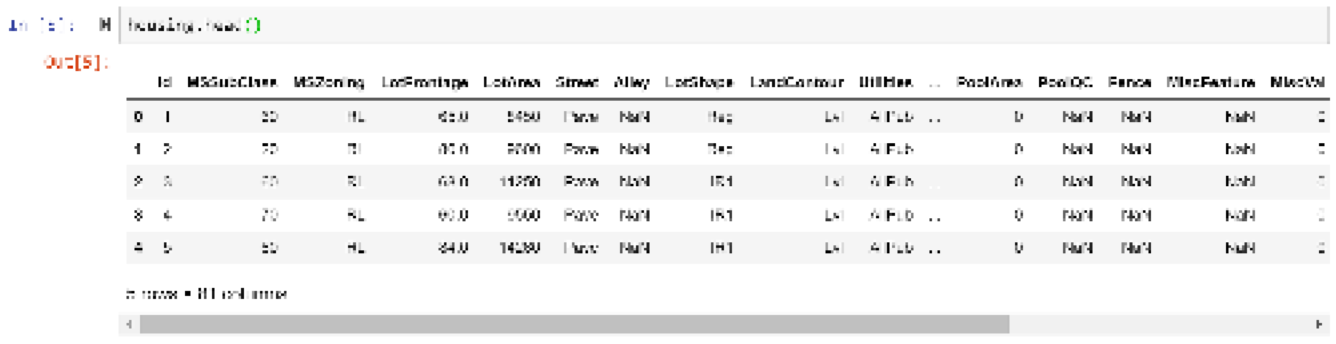


```
In [4]:  # Get the shape of data
         housing.shape
Out[4]: (1460, 81)
```

There are only 1460 training examples in the dataset, which means that it is small by machine learning standards. The shape of the dataset also tells is that we have 81 attributes. Of the 81 attributes, one is the Id for the houses – not useful as a feature – and one is the target variable, SalePrice, that the model should predict. This means that we have 79 attributes that have the potential to be used to train our predictive model.

Now let's take a look at the top five rows using the DataFrame's `head()` method.

```
# Get the top 5 rows
housing.head()
```



```
In [5]:  housing.head()
Out[5]:
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0

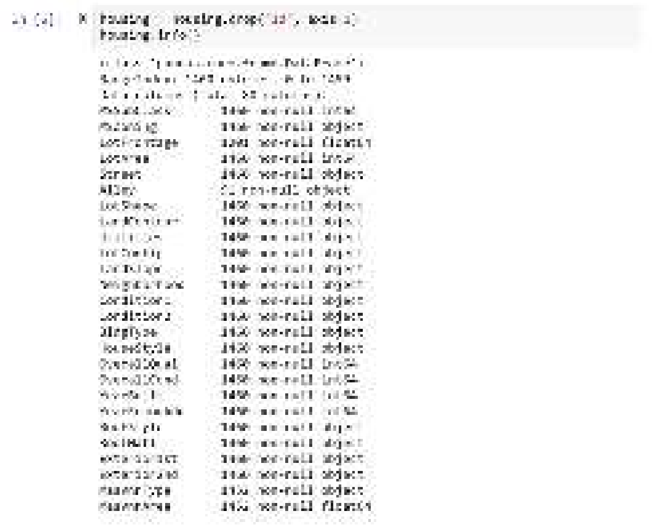
5 rows × 81 columns

Each row represents one house. We can see that we have both numerical (e.g., *LotFrontage*) and categorical attributes (e.g., *LotShape*). We also notice that we have many missing values (NaN) as not all the houses have values set for all the attributes.

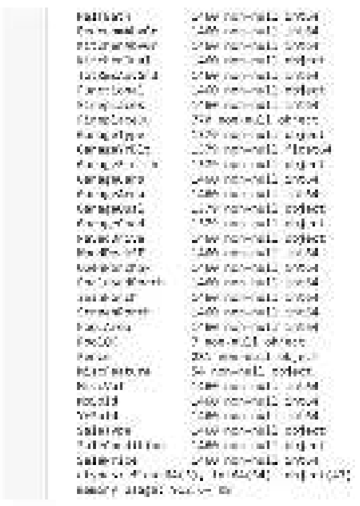
We have a column called *Id* which is not useful as an attribute. We can either omit it or use it as an index for our DataFrame. We are going to drop that column because indexes of houses are not relevant for this problem anyway.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type and number of non-null values. So let's drop the *Id* column and call the *info()* method:

```
housing = housing.drop("Id", axis=1)
housing.info()
```



```
In [6]:  housing = housing.drop("Id", axis=1)
         housing.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 80 columns):
MSSubClass       1460 non-null int64
MSZoning         1460 non-null object
LotFrontage      1201 non-null float64
LotArea          1460 non-null int64
Street           1460 non-null object
Alley            91 non-null object
LotShape         1460 non-null object
LandContour      1460 non-null object
Utilities        1460 non-null object
LotConfig        1460 non-null object
LandSlope        1460 non-null object
Neighborhood     1460 non-null object
Condition1       1460 non-null object
Condition2       1460 non-null object
BldgType         1460 non-null object
HouseStyle       1460 non-null object
OverallQual      1460 non-null int64
OverallCond      1460 non-null int64
YearBuilt        1460 non-null int64
YearRemodAdd     1460 non-null int64
RoofStyle        1460 non-null object
RoofMatl         1460 non-null object
Exterior1st      1460 non-null object
Exterior2nd      1460 non-null object
MasVnrType       1452 non-null object
MasVnrArea       1452 non-null float64
```



```
HalfBath         1460 non-null int64
BedroomAbvGr     1460 non-null int64
KitchenAbvGr     1460 non-null int64
KitchenQual      1460 non-null object
TotRmsAbvGrd     1460 non-null int64
Functional       1460 non-null object
Fireplaces       1460 non-null int64
FireplaceQu      770 non-null object
GarageType       1379 non-null object
GarageYrBlt      1379 non-null float64
GarageFinish     1379 non-null object
GarageCars       1460 non-null int64
GarageArea       1460 non-null int64
GarageQual       1379 non-null object
GarageCond       1379 non-null object
PavedDrive       1460 non-null object
WoodDeckSF       1460 non-null int64
OpenPorchSF      1460 non-null int64
EnclosedPorch    1460 non-null int64
3SsnPorch        1460 non-null int64
ScreenPorch      1460 non-null int64
PoolArea         1460 non-null int64
PoolQC           7 non-null object
Fence            281 non-null object
MiscFeature      54 non-null object
MiscVal          1460 non-null int64
MoSold           1460 non-null int64
YrSold           1460 non-null int64
SaleType         1460 non-null object
SaleCondition    1460 non-null object
SalePrice        1460 non-null int64
dtypes: float64(3), int64(34), object(43)
memory usage: [illegible]
```

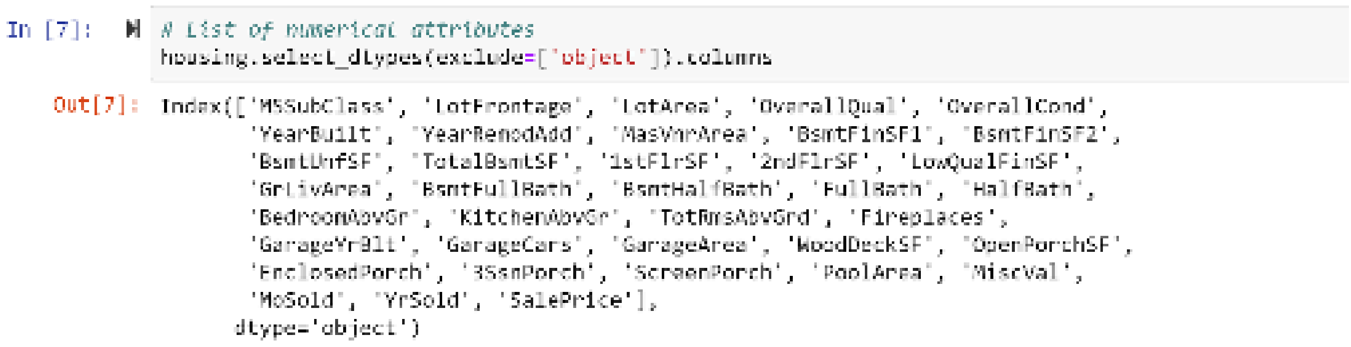
The *info()* method tells us that we have 37 numerical attributes, 3 float64 and 34 int64, and 43 categorical columns. Notice that we have many attributes that are not set for most of the houses. For example, the *Alley* attribute has only 91 non-null values, meaning that all other houses are missing this feature. We will need to take care of this later.

Here we have a mix of numerical and categorical attributes. Let's look at these separately and also use the `describe()` method to get their statistical summary.

Note that we can distinguish between numerical and categorical attributes by filtering based on their `dtypes`.

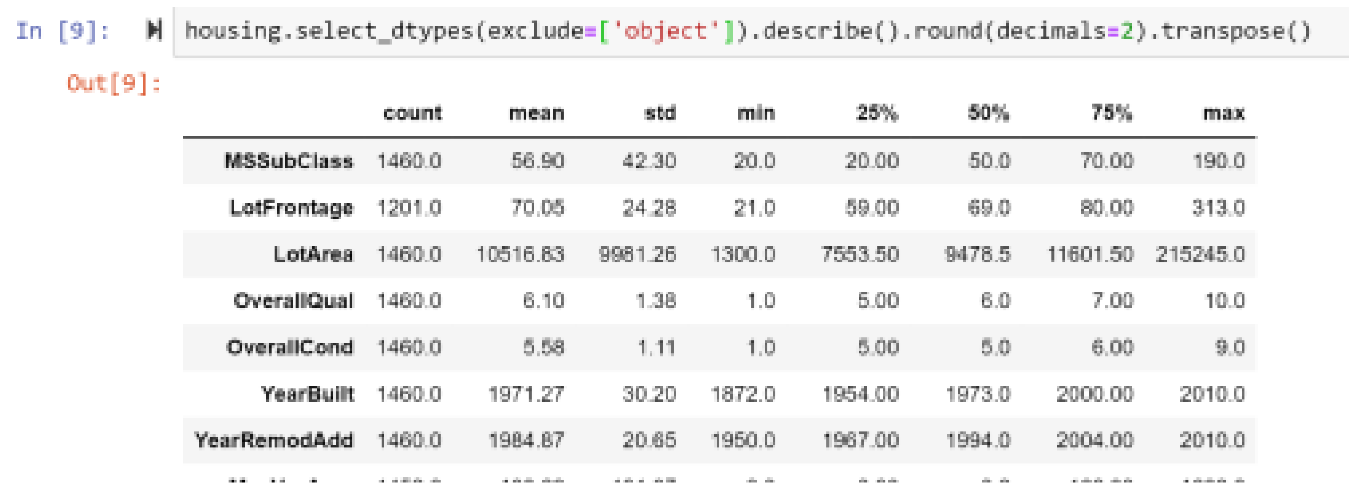
Numerical Attributes

```
# List of numerical attributes
housing.select_dtypes(exclude=['object']).columns
```



```
In [7]:  # List of numerical attributes
          housing.select_dtypes(exclude=['object']).columns

Out[7]: Index(['MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual', 'OverallCond',
               'YearBuilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2',
               'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF',
               'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath', 'HalfBath',
               'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces',
               'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
               'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal',
               'MoSold', 'YrSold', 'SalePrice'],
              dtype='object')
```

```
# Get the data summary with upto 2 decimals and call transpose() for a better view of the res
housing.select_dtypes(exclude=['object']).describe().round(decimals=2).transpose()
```



```
In [9]:  housing.select_dtypes(exclude=['object']).describe().round(decimals=2).transpose()

Out[9]:
```

	count	mean	std	min	25%	50%	75%	max
MSSubClass	1460.0	56.90	42.30	20.0	20.00	50.0	70.00	190.0
LotFrontage	1201.0	70.05	24.28	21.0	59.00	69.0	80.00	313.0
LotArea	1460.0	10516.83	9981.26	1300.0	7553.50	9478.5	11601.50	215245.0
OverallQual	1460.0	6.10	1.38	1.0	5.00	6.0	7.00	10.0
OverallCond	1460.0	5.58	1.11	1.0	5.00	5.0	6.00	9.0
YearBuilt	1460.0	1971.27	30.20	1872.0	1954.00	1973.0	2000.00	2010.0
YearRemodAdd	1460.0	1984.87	20.65	1950.0	1967.00	1994.0	2004.00	2010.0

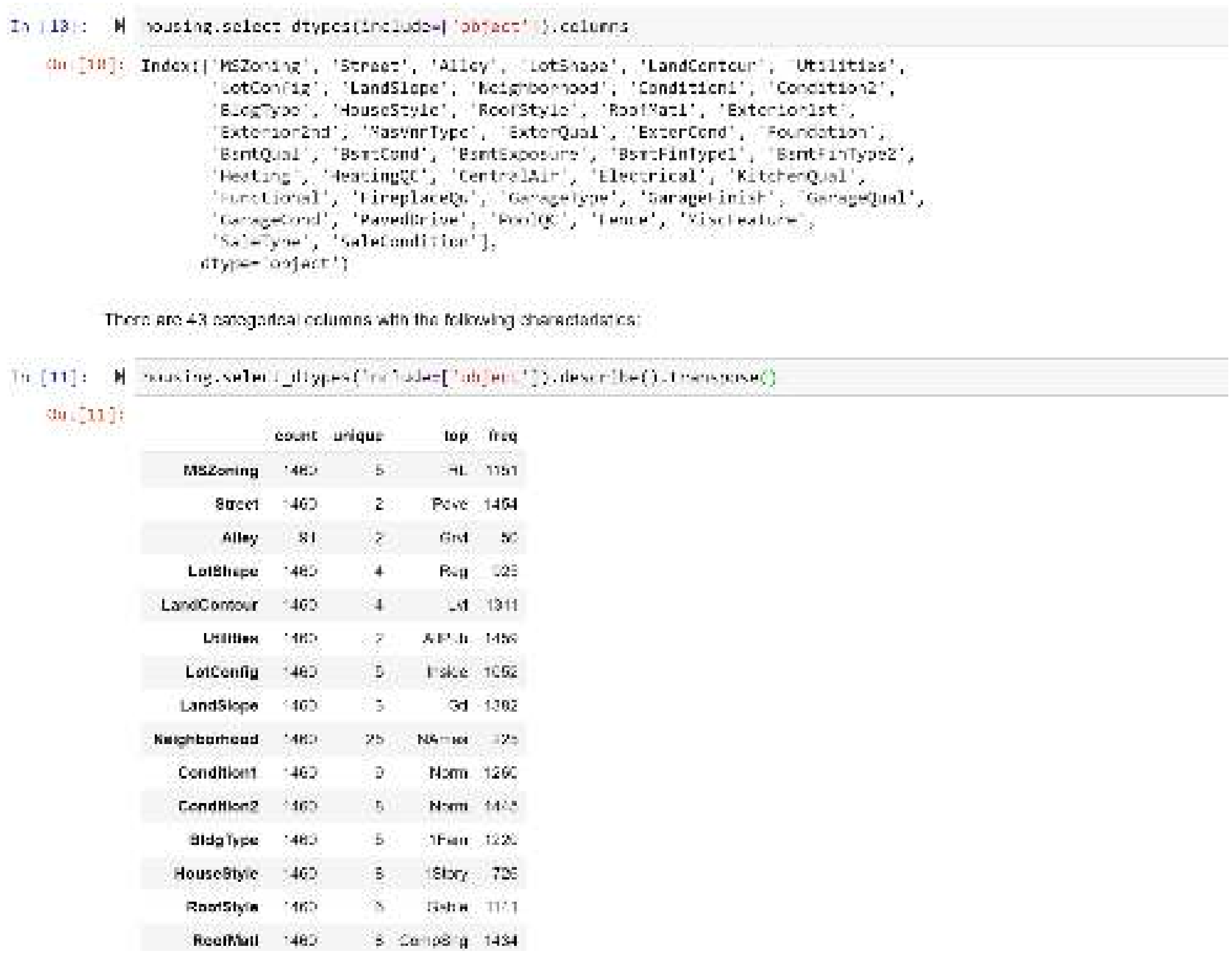
The `count`, `mean`, `min`, and `max` columns are self-explanatory. Note that the null values are ignored; for example, the count of *LotFrontage* is 1201, not 1460. The `std` column shows the standard deviation which measures how dispersed the values are.

The `25%`, `50%`, and `75%` columns show the corresponding **percentiles**: a percentile indicates the value below which a given percentage of observations in a group of observations falls. For example, 25% of the houses have *YearBuilt* lower than 1954, while 50% are lower than 1973 and 75% are lower than 2000.

Recall from the statistics lessons that the 25th percentile is also known as the 1st quartile, the 50th percentile is the median, and the 75th percentile is also known as the 3rd quartile.

Categorical Attributes

```
# Get the categorical attributes
housing.select_dtypes(include=['object']).columns

#Get the sumamry of categorical attributes
housing.select_dtypes(include=['object']).describe().transpose()
```



```
In [10]:  housing.select_dtypes(include=['object']).columns

Out[10]: Index(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
                'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
                'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
                'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
                'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
                'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
                'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
                'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature',
                'SaleType', 'SaleCondition'],
               dtype='object')
```

There are 43 categorical columns with the following characteristics:

```
In [11]:  housing.select_dtypes(include=['object']).describe().transpose()

Out[11]:
```

	count	unique	top	freq
MSZoning	1460	5	RL	1151
Street	1460	2	Pave	1454
Alley	91	2	Grvl	50
LotShape	1460	4	Reg	925
LandContour	1460	4	Lvl	1311
Utilities	1460	2	AllPub	1459
LotConfig	1460	5	Inside	1052
LandSlope	1460	3	Gtl	1382
Neighborhood	1460	25	NAmes	225
Condition1	1460	9	Norm	1260
Condition2	1460	8	Norm	1445
BldgType	1460	5	1Fam	1220
HouseStyle	1460	8	1Story	726
RoofStyle	1460	6	Gable	1141
RoofMatl	1460	8	CompShg	1434

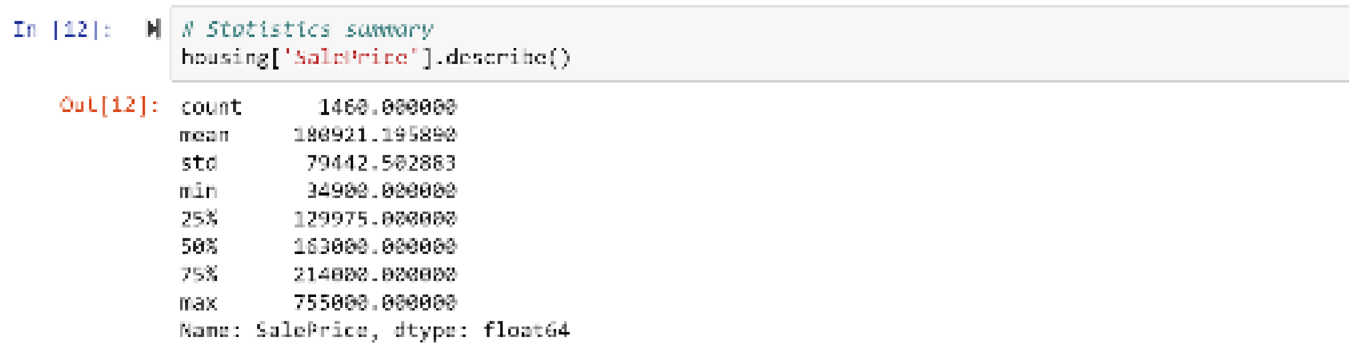
Note that for categorical attributes we do not get a statistical summary. But we can get some important information like number of unique values and top values for each attribute. For example, we can see that we can have 8 types of *HouseStyle*, with *1Story* houses being the most frequent type.

Explore Numerical Attributes

Looking at data distributions

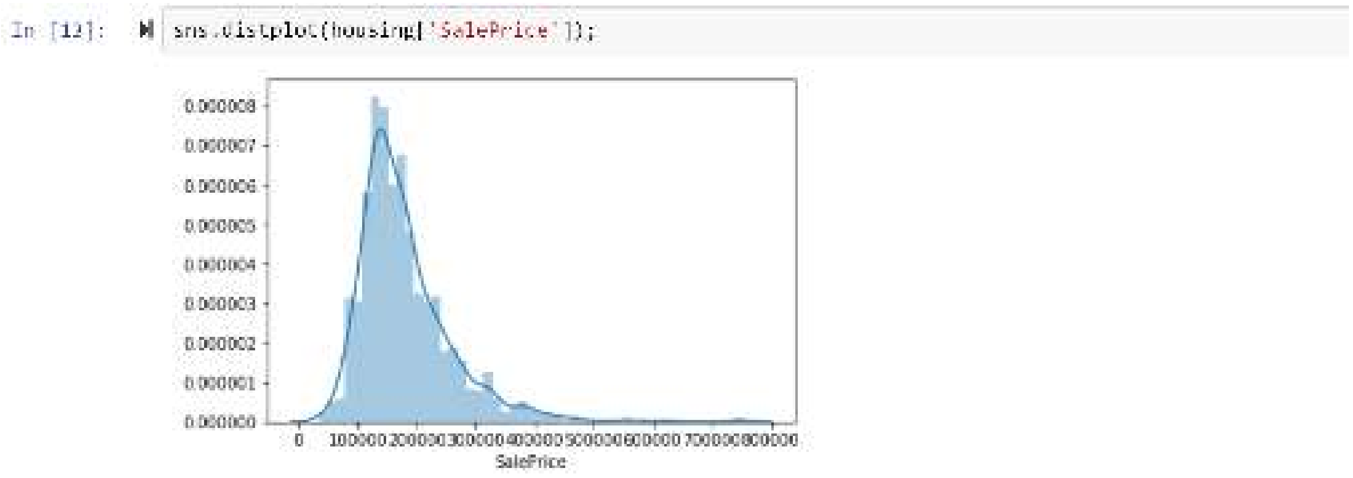
Let’s have a detailed look at our target variable, *SalePrice*.

```
# Descriptive statistics summary
housing['SalePrice'].describe()
```



```
In [12]:  # Statistics summary
          housing['SalePrice'].describe()

Out[12]:  count      1460.000000
          mean     180921.195890
          std       79442.502883
          min       34900.000000
          25%      129975.000000
          50%      163000.000000
          75%      214000.000000
          max      755000.000000
          Name: SalePrice, dtype: float64
```

```
# Get the distribution plot
sns.distplot(housing['SalePrice']);
```



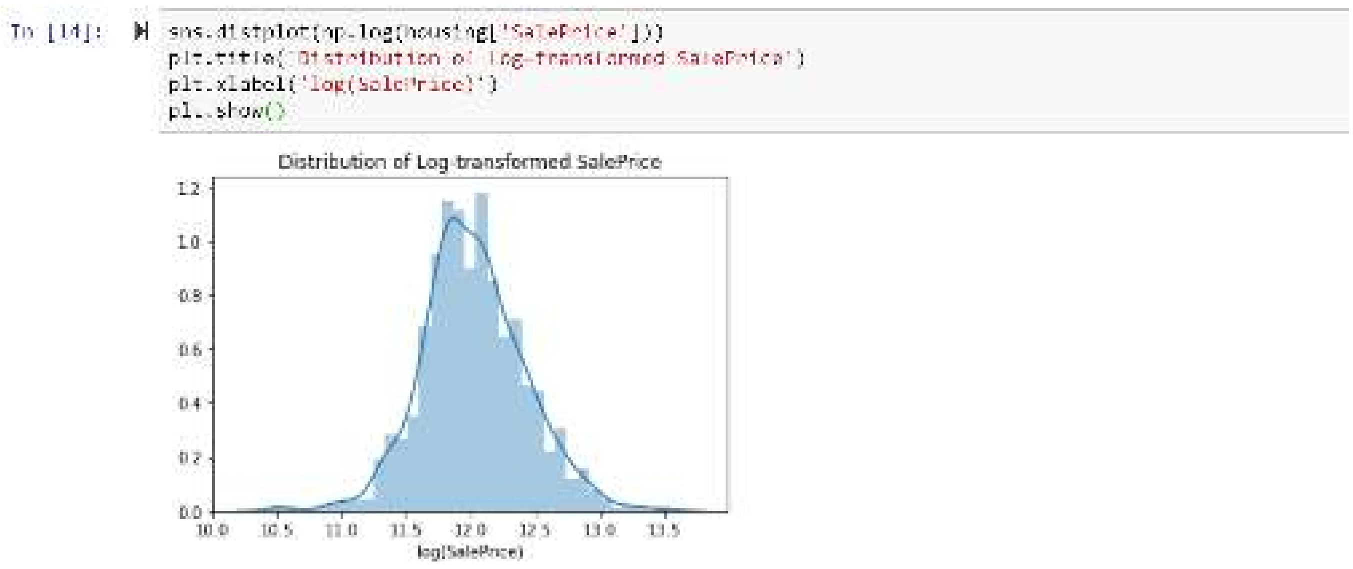
```
In [13]:  sns.distplot(housing['SalePrice']);
```

The distribution plot tells us that we have a skewed variable. In fact from the statistical summary, we already saw that the mean price is about 181K while 50% of the houses were sold for less than 163K.

When dealing with skewed variables, it is a good practice to reduce the skew of the dataset because it can impact the accuracy of the model. This is an important step if we are going to use linear regression modeling; other algorithms, like tree-based Random Forests can handle skewed data. We will understand this in detail later under “Feature Scaling”. For now, let’s look at the updated distribution of our target variable once we apply a log transformation to it. Applying a **log transformation** means to simply take the

log of the skewed variable to improve the fit by altering the scale and making the variable more normally distributed.

```
# Take the log to make the distribution more normal
sns.distplot(np.log(housing['SalePrice']))
plt.title('Distribution of Log-transformed SalePrice')
plt.xlabel('log(SalePrice)')
plt.show()
```

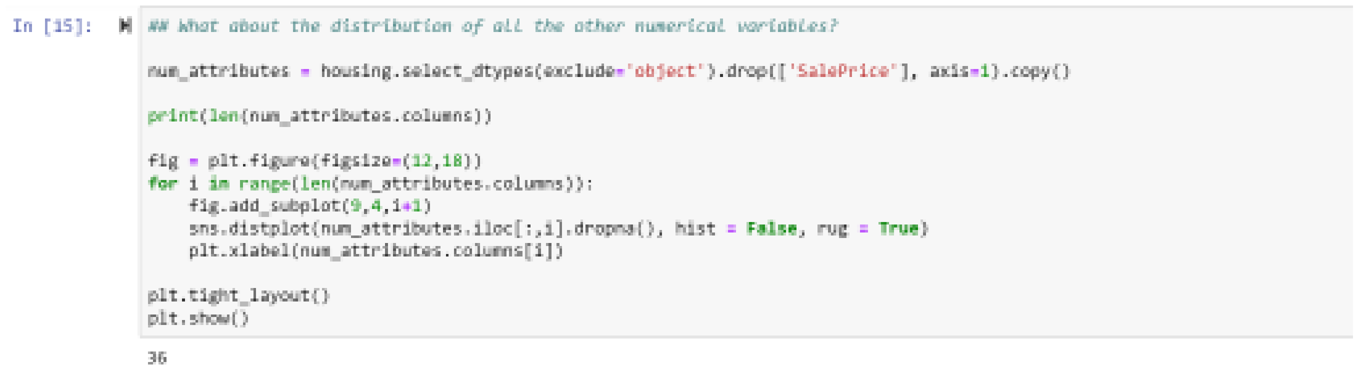


```
In [14]:  sns.distplot(np.log(housing['SalePrice']))
          plt.title('Distribution of Log-transformed SalePrice')
          plt.xlabel('log(SalePrice)')
          plt.show()
```

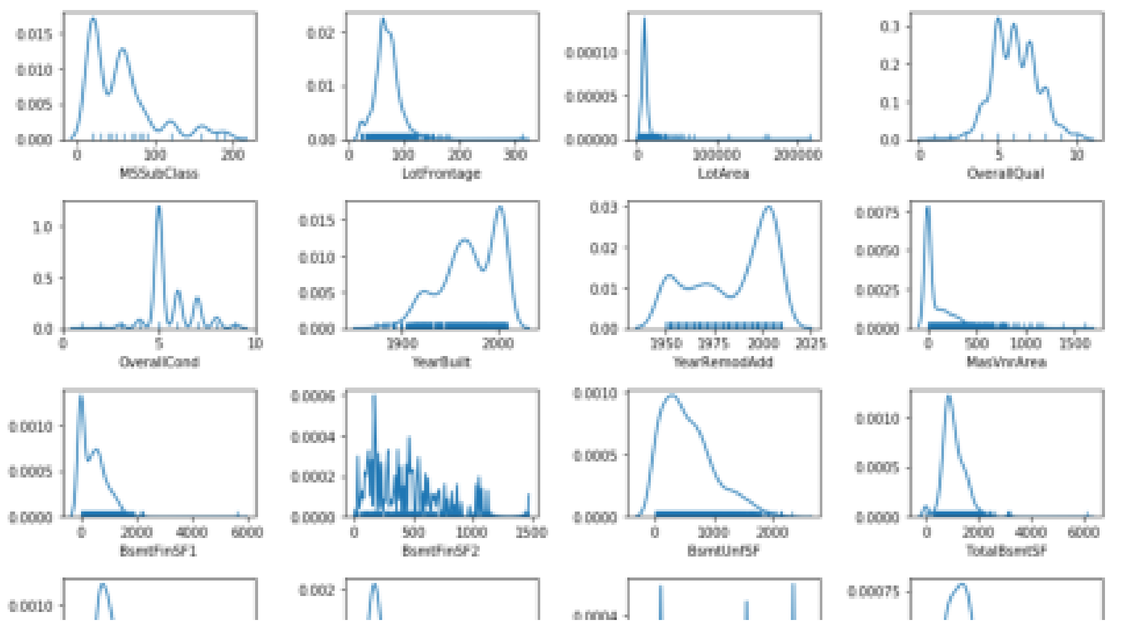
We can clearly see that the log-transformed variable is more normally distributed and we have managed to reduce the skew.

What about all the other numerical variables? What do their distributions look like? We can plot the distributions of all the numerical variables by calling the `distplot()` method in a for loop, like so:

```
## What about the distribution of all the other numerical variables?
num_attributes = housing.select_dtypes(exclude='object').drop(['SalePrice'], axis=1).copy()

# Print num of variables to make sure we didn't mess up in the last step
print(len(num_attributes.columns))

fig = plt.figure(figsize=(12,18))
for i in range(len(num_attributes.columns)):
    fig.add_subplot(9,4,i+1)
    sns.distplot(num_attributes.iloc[:,i].dropna(), hist = False, rug = True)
    plt.xlabel(num_attributes.columns[i])

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



```
In [15]:  ## What about the distribution of all the other numerical variables?

          num_attributes = housing.select_dtypes(exclude='object').drop(['SalePrice'], axis=1).copy()

          print(len(num_attributes.columns))

          fig = plt.figure(figsize=(12,18))
          for i in range(len(num_attributes.columns)):
              fig.add_subplot(9,4,i+1)
              sns.distplot(num_attributes.iloc[:,i].dropna(), hist = False, rug = True)
              plt.xlabel(num_attributes.columns[i])

          plt.tight_layout()
          plt.show()

          36
```



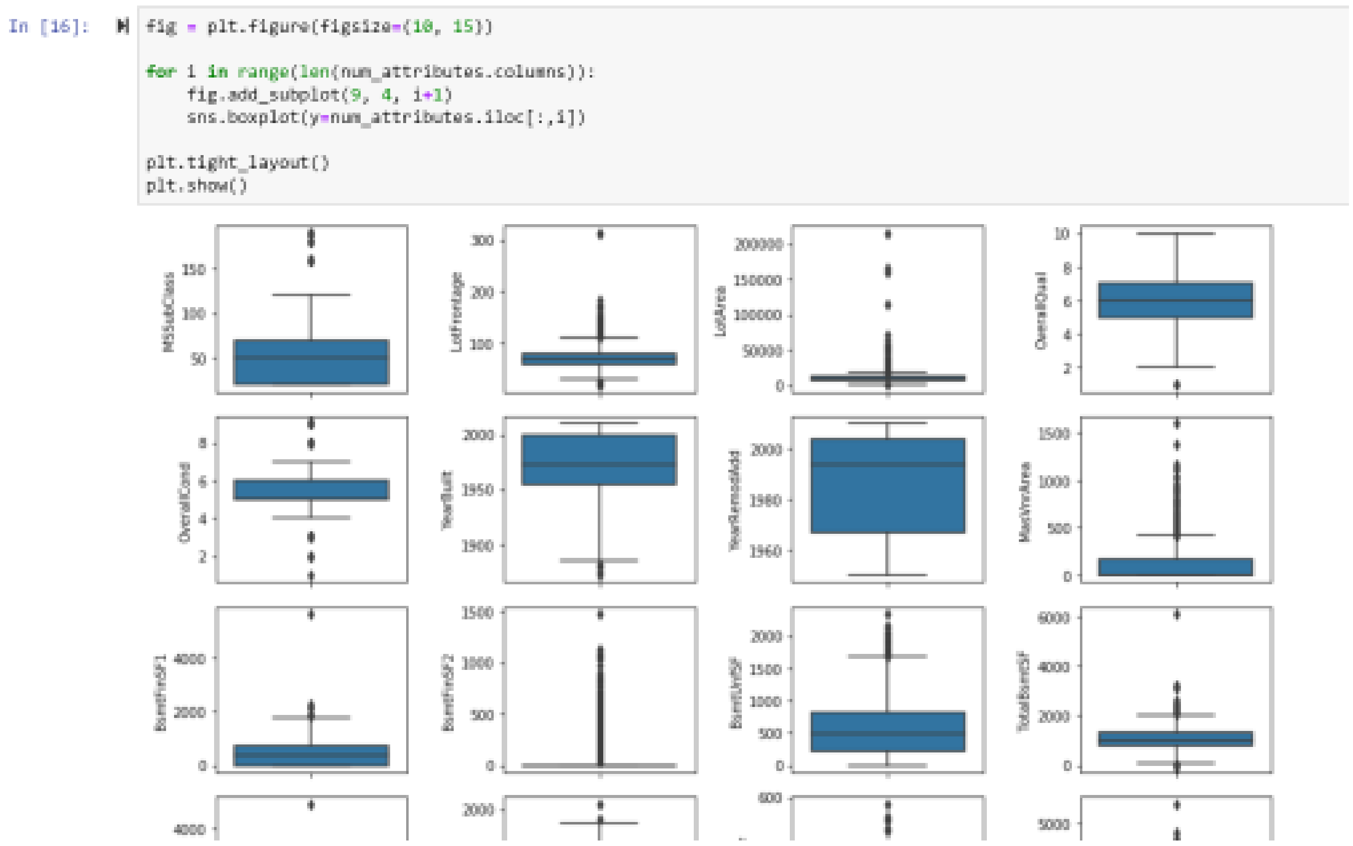
Notice how varying the distributions and scales for the different variables are ,this is the reason we need to do feature scaling before we can use these features for modeling. For example, we can clearly see how skewed *LotArea* is. It is in dire need of some polishing before it can be used for learning.

📝 We will get back to all the needed transformations and “applying the fixes” later. In this exploratory analysis steps, we are just taking notes on what we need to take care of in order to create a good predictive model.

Looking for Outliers

In the statistics lesson, we learned that box plots give us a good overview of our data. From the distribution of observations in relation to the upper and lower quartiles, we can spot outliers. Let’s see this in action with the `boxplot()` method and a for loop to plot all the attributes in one go:

```
fig = plt.figure(figsize=(10, 15))

for i in range(len(num_attributes.columns)):
    fig.add_subplot(9, 4, i+1)
    sns.boxplot(y=num_attributes.iloc[:,i])



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



```
In [16]: fig = plt.figure(figsize=(10, 15))

         for i in range(len(num_attributes.columns)):
             fig.add_subplot(9, 4, i+1)
             sns.boxplot(y=num_attributes.iloc[:,i])

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

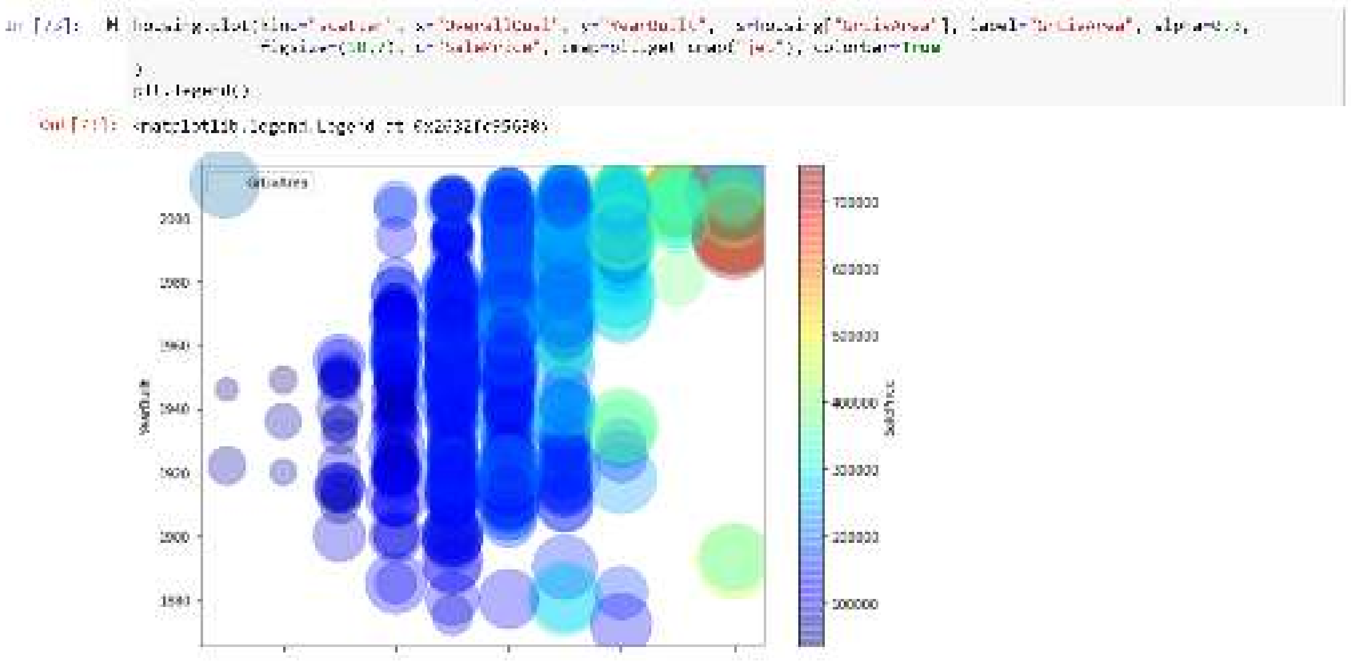
📝 From the boxplots we can see that for instance *LotFrontage* values above 200 and *LotArea* above 150000 can be marked as outliers. However, instead of relying on our own “visual sense” to spot patterns and define the range for outliers, when doing data cleaning, we will use the knowledge of percentiles to be more accurate. For now our takeaway from this analysis is that we need to take care of outliers in the data cleaning phase.

Just-for-fun plot

Our brains are very good at spotting patterns on pictures, but sometimes we need to play around with visualization parameters and try out different kind of plots to make those patterns stand out. Let’s create a fun example plot for learning to play with visualizations, especially when we want to analyze relations among multiple variables at once.

We are going to look at the prices. The radius of each circle represents *GrLivArea* (option ***s***), and the color represents the price (option ***c***). We will use a predefined color map (option ***cmap***) called *jet*, which ranges from blue (low values) to red (high prices).

```
housing.plot(kind="scatter", x="OverallQual", y="YearBuilt",  s=housing["GrLivArea"], label='
             alpha=0.3, figsize=(10,7), c="SalePrice", cmap=plt.get_cmap("jet"), colorbar=Tru

plt.legend()
```



```
In [illegible]:  housing.plot(kind="scatter", x="OverallQual", y="YearBuilt",  s=housing["GrLivArea"], label="GrLivArea", alpha=0.3,
                         figsize=(10,7), c="SalePrice", cmap=plt.get_cmap("jet"), colorbar=True
             )
             plt.legend()
Out[illegible]: <matplotlib.legend.Legend at 0x2d32fc95630>
```

The plot above tells us that the housing prices are very much related to the YearBuilt (*y-axis*) and OverallQual (*x-axis*). Newer and higher quality houses mean more expensive prices. This is shown increasing red going towards upper-right end of the plot and vice versa. Prices are also related to the *GrLivArea*, radius of the circle.

Correlations Among Numerical Attributes

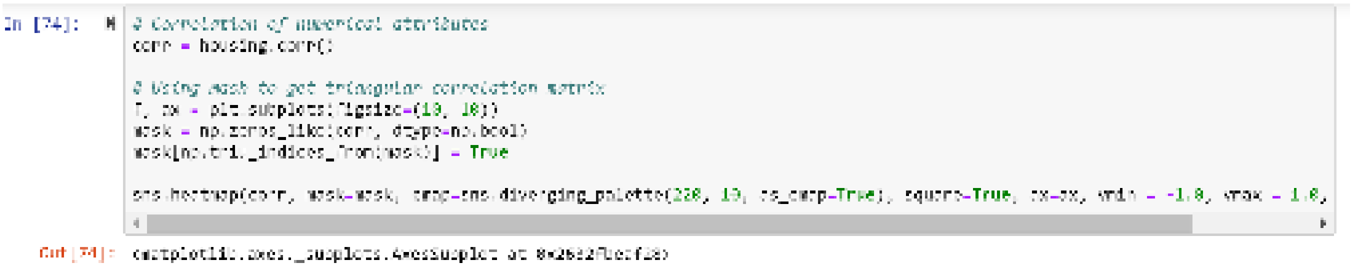
Correlation tells us the strength of the relationship between pairs of attributes. In an ideal situation, we would have an independent set of features/attributes, but real data is not ideal. It is useful to know whether some pairs of attributes are correlated and by how much because it is a good practice to **remove highly correlated features**.

We can use the `corr()` method to easily get the correlations and then visualize them using the `heatmap()` method – Python does feel like magic often, isn't it?!

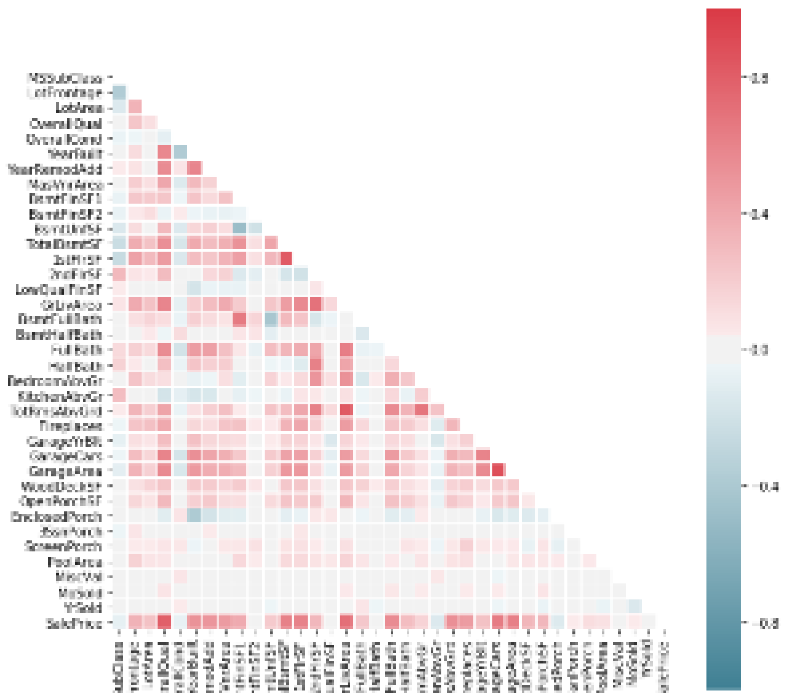
The *corr()* method returns pairs of all attributes and their correlation coefficients in range [-1; 1], where 1 indicates positive correlation, -1 negative correlation and 0 means no relationship between variables at all.

correlation and 0 means no relationship between variables at all.

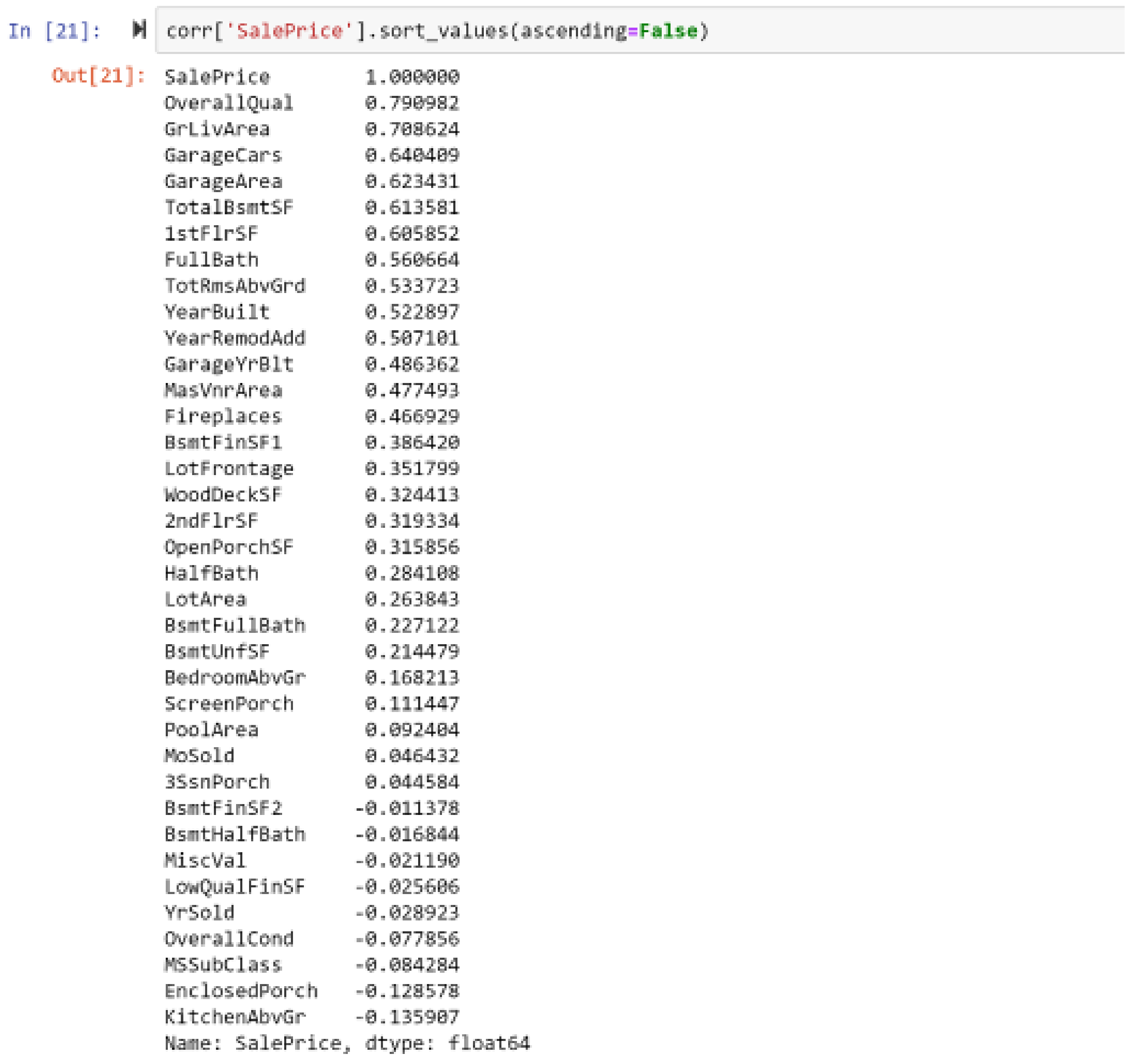
```
# Correlation of numerical attributes
corr = housing.corr()

# Using mask to get triangular correlation matrix
f, ax = plt.subplots(figsize=(15, 15))
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

sns.heatmap(corr, mask=mask, cmap=sns.diverging_palette(220, 10, as_cmap=True), square=True,
```



```
In [74]:  # Correlation of numerical attributes
          corr = housing.corr()

          # Using mask to get triangular correlation matrix
          f, ax = plt.subplots(figsize=(15, 15))
          mask = np.zeros_like(corr, dtype=np.bool)
          mask[np.triu_indices_from(mask)] = True

          sns.heatmap(corr, mask=mask, cmap=sns.diverging_palette(220, 10, as_cmap=True), square=True, ax=ax, vmin = -1.0, vmax = 1.0,

Out[74]:  <matplotlib.axes._subplots.AxesSubplot at 0x2632fbecf28>
```



From the heatmap, we can easily see that we have some variables that are highly correlated with price (darker red) and that there are variables highly correlated among themselves as well. The heatmap is useful for a first high-level overview. Let's get a sorted list of correlations among all the attributes and the target variable, *SalePrice*, for a deeper understanding of what's going on.



```
In [21]:  corr['SalePrice'].sort_values(ascending=False)
```

```
Out[21]: SalePrice        1.000000
         OverallQual      0.790982
         GrLivArea        0.708624
         GarageCars       0.640409
         GarageArea       0.623431
         TotalBsmtSF      0.613581
         1stFlrSF         0.605852
         FullBath         0.560664
         TotRmsAbvGrd     0.533723
         YearBuilt        0.522897
         YearRemodAdd     0.507101
         GarageYrBlt      0.486362
         MasVnrArea       0.477493
         Fireplaces       0.466929
         BsmtFinSF1       0.386420
         LotFrontage      0.351799
         WoodDeckSF       0.324413
         2ndFlrSF         0.319334
         OpenPorchSF      0.315856
         HalfBath         0.284108
         LotArea          0.263843
         BsmtFullBath     0.227122
         BsmtUnfSF        0.214479
         BedroomAbvGr     0.168213
         ScreenPorch      0.111447
         PoolArea         0.092404
         MoSold           0.046432
         3SsnPorch        0.044584
         BsmtFinSF2      -0.011378
         BsmtHalfBath    -0.016844
         MiscVal         -0.021190
         LowQualFinSF    -0.025606
         YrSold          -0.028923
         OverallCond     -0.077856
         MSSubClass      -0.084284
         EnclosedPorch   -0.128578
         KitchenAbvGr    -0.135907
         Name: SalePrice, dtype: float64
```

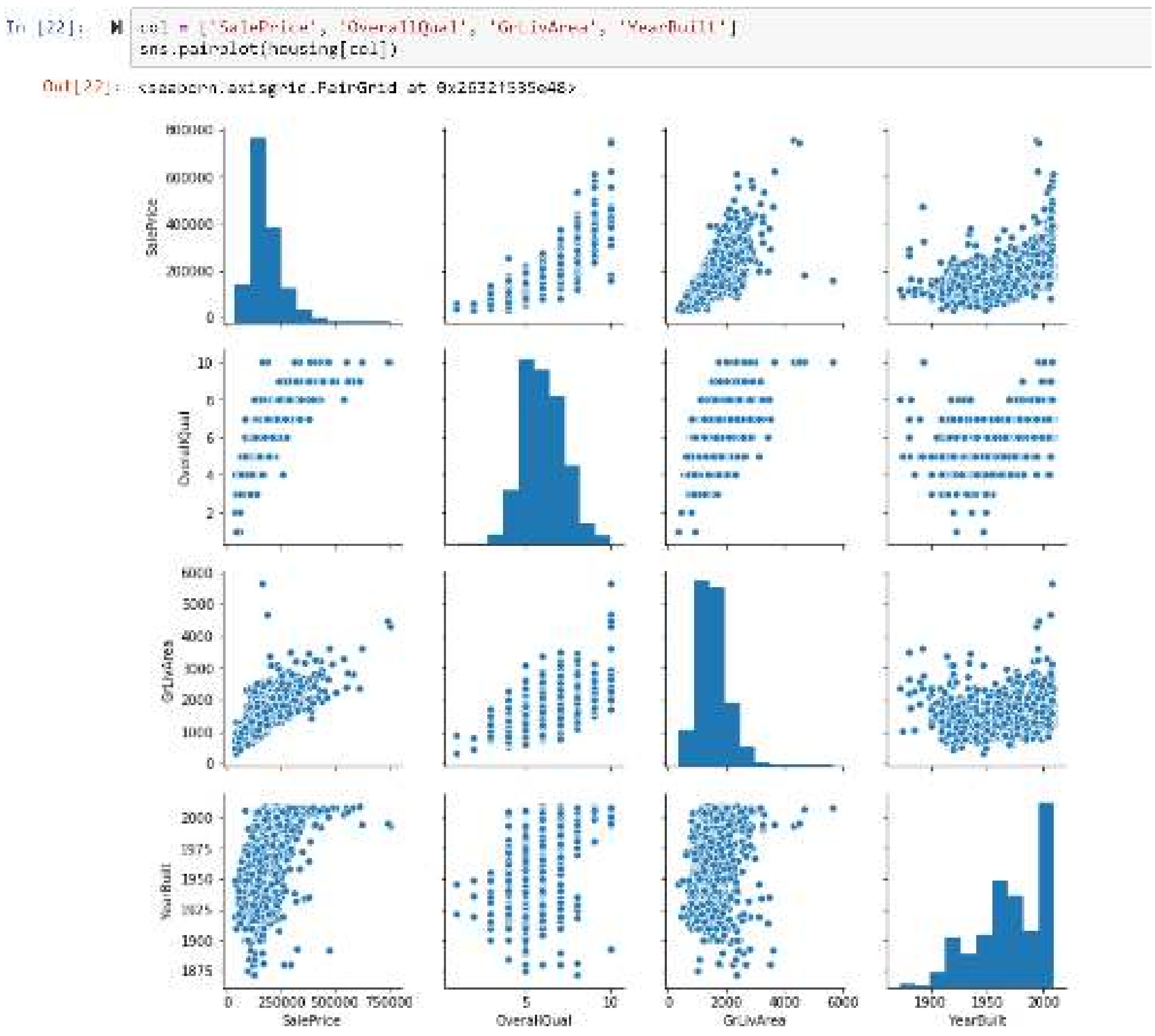
From these values, we can see that *OverallQual* and *GrLivArea* have the most impact on price, while attributes like *PoolArea* and *MoSold* are not related to it.

Pair-wise scatter matrix

We have a lot of unique pairs of variables i.e. *N(N - 1)/2*. **Joint distribution** can be used to look for a relationship between all of the possible pairs, two at a time.

For the sake of completeness, we might want to display a rough joint distribution plot for each pair of variables. This can be done by using `pairplot()` from *sns*. Since we have a fairly big *N*, so we are going to create scatter plots for only some of the interesting attributes to get a visual feel for these correlations.

```
col = ['SalePrice', 'OverallQual', 'GrLivArea', 'YearBuilt']
sns.pairplot(housing[col])
```



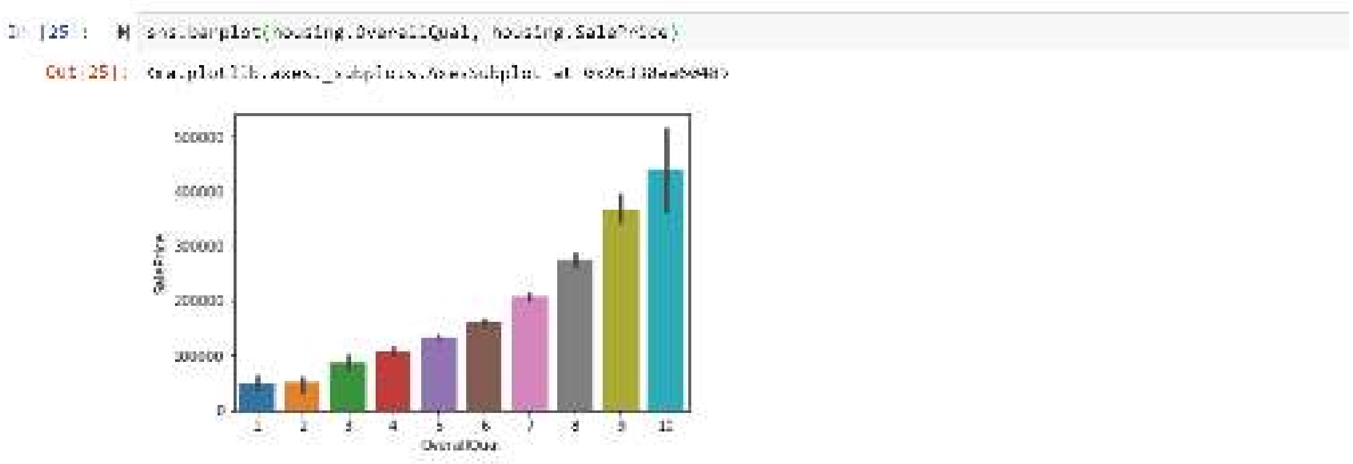
```
In [22]:  col = ['SalePrice', 'OverallQual', 'GrLivArea', 'YearBuilt']
          sns.pairplot(housing[col])

Out[22]: <seaborn.axisgrid.PairGrid at 0x2632f535e48>
```

From the pairplots, we can clearly see how with an increase in *GrLivArea* the price increases as well. Play around with other attributes as well.

In order to train our “creating plots muscle”, let’s look at other types of plots that can make the relationship for the highest correlated variables, *OverallQual*, with the target variable, *SalePrice*, really standout.

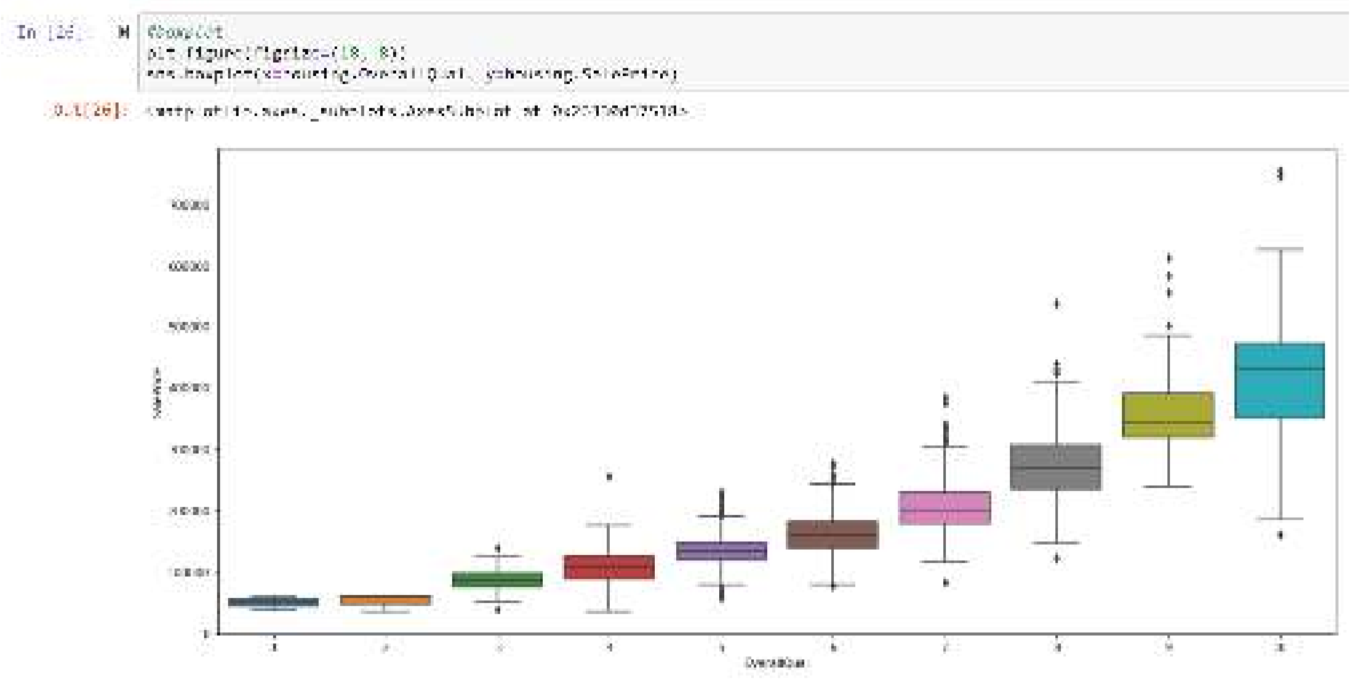
Let’s see what the `barplot()` and `boxplot()` methods give us.

```
sns.barplot(housing.OverallQual, housing.SalePrice)
```



```
In [25]:  sns.barplot(housing.OverallQual, housing.SalePrice)
Out[25]: <matplotlib.axes._subplots.AxesSubplot at [illegible]>
```

```
# Boxplot
plt.figure(figsize=(18, 8))
sns.boxplot(x=housing.OverallQual, y=housing.SalePrice)
```



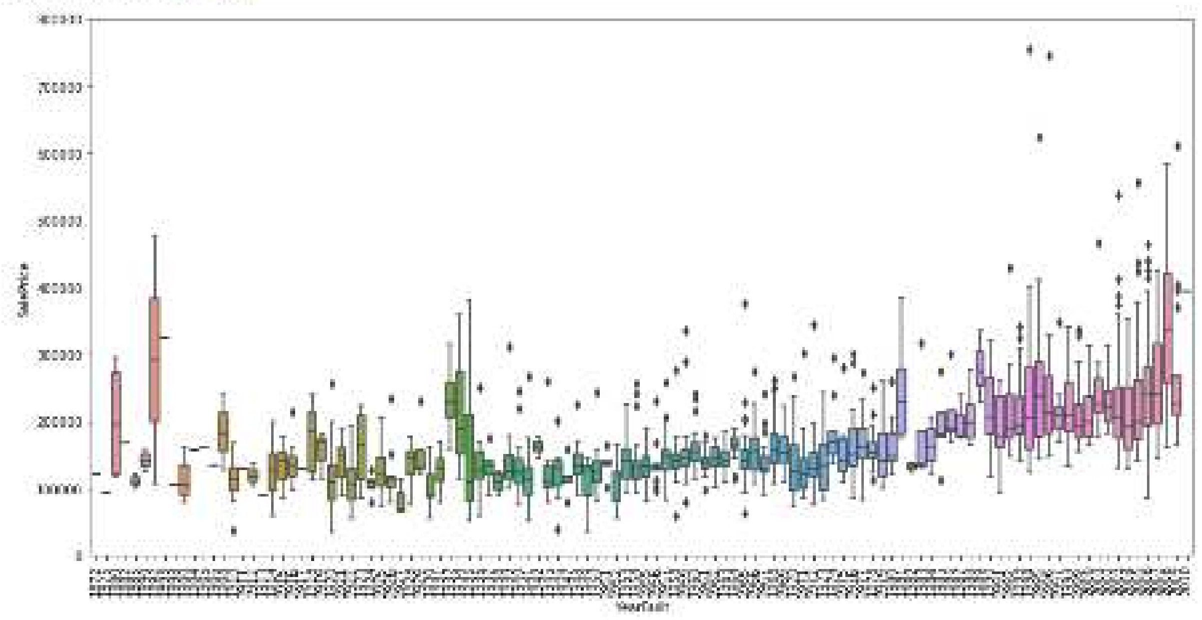
```
In [26]:  #boxplot
          plt.figure(figsize=(18, 8))
          sns.boxplot(x=housing.OverallQual, y=housing.SalePrice)
Out[26]: <matplotlib.axes._subplots.AxesSubplot at [illegible]>
```

Now we can clearly see how prices change with quality, higher quality, higher price is so obvious. But we can also notice that at higher qualities there is much higher variability in prices.

What about the age of the house? Let's look at that as well.

```
var = 'YearBuilt'
data = pd.concat([housing['SalePrice'], housing[var]], axis=1)
f, ax = plt.subplots(figsize=(16, 8))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=800000);
plt.xticks(rotation=90);
```

```
In [29]:  var = 'YearBuilt'
          data = pd.concat([housing['SalePrice'], housing[var]], axis=1)
          f, ax = plt.subplots(figsize=(16, 8))
          fig = sns.boxplot(x=var, y="SalePrice", data=data)
          fig.axis(ymin=0, ymax=800000);
          plt.xticks(rotation=90);
```



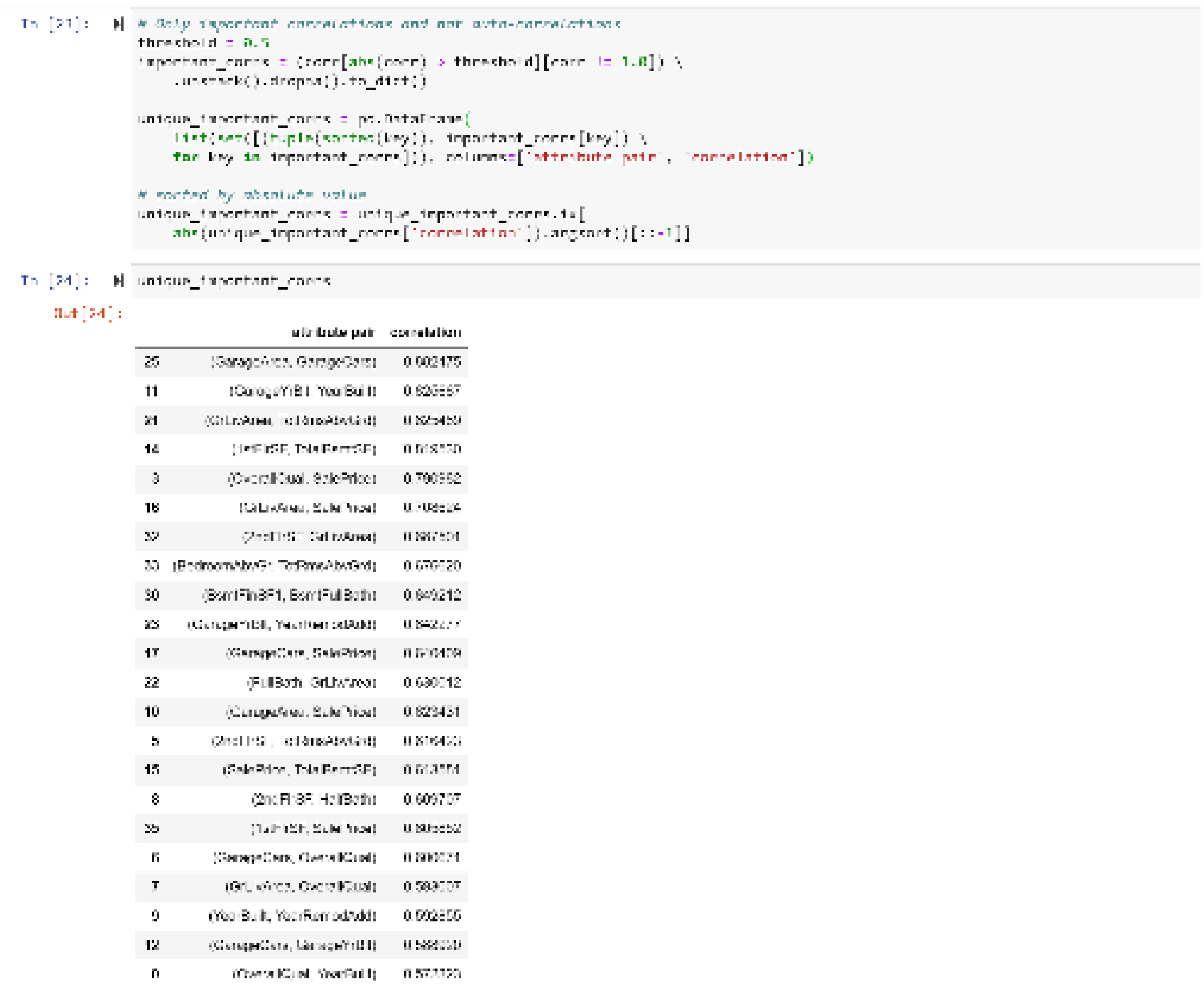
We can see that some of the very old houses are expensive, but overall the newest houses tend to have higher prices.

From our correlations list, we also know that *GarageArea* and *GarageCars* both also have a high correlation with price, but do we need both? Both common sense and the heat map suggest that these two variables are highly correlated among themselves, so only one should be sufficient for modeling purposes. In order to talk with numbers and data, let's also look at the values for correlations among attributes with regard to each other as well, not just with the target variable.

To avoid getting correlations for a variable with its own self, we can use the knowledge that a perfect correlation of 1.0 indicates that we are looking at self-correlations. We will drop these from our results. We are also going to sort our results from the most correlated pair to the least.

```
# Only important correlations and not auto-correlations
threshold = 0.5
important_corrs = (corr[abs(corr) > threshold][corr != 1.0]) \
    .unstack().dropna().to_dict()

unique_important_corrs = pd.DataFrame(
    list(set([(tuple(sorted(key)), important_corrs[key]) \
    for key in important_corrs])), columns=['attribute pair', 'correlation'])
```

```
# Sorted by absolute value
unique_important_corrs = unique_important_corrs.ix[
    abs(unique_important_corrs['correlation']).argsort()[::-1]]


unique_important_corrs
```



```
In [21]:  # Only important correlations and not auto-correlations
          threshold = 0.5
          important_corrs = (corr[abs(corr) > threshold][corr != 1.0]) \
              .unstack().dropna().to_dict()

          unique_important_corrs = pd.DataFrame(
              list(set([(tuple(sorted(key)), important_corrs[key]) \
              for key in important_corrs])), columns=['attribute pair', 'correlation'])

          # sorted by absolute value
          unique_important_corrs = unique_important_corrs.ix[
              abs(unique_important_corrs['correlation']).argsort()[::-1]]

In [24]:  unique_important_corrs
Out[24]:
```

	attribute pair	correlation
25	(GarageArea, GarageCars)	0.882475
11	(GarageYrBlt, YearBuilt)	0.825667
21	(GrLivArea, TotRmsAbvGrd)	0.825489
14	(1stFlrSF, TotalBsmtSF)	0.819530
3	(OverallQual, SalePrice)	0.790982
18	(GrLivArea, SalePrice)	0.708624
32	(2ndFlrSF, GrLivArea)	0.687501
33	(BedroomAbvGr, TotRmsAbvGrd)	0.676620
30	(BsmtFinSF1, BsmtFullBath)	0.649212
23	(GarageYrBlt, YearRemodAdd)	0.642277
17	(GarageCars, SalePrice)	0.640409
22	(FullBath, GrLivArea)	0.630012
10	(GarageArea, SalePrice)	0.623431
5	(2ndFlrSF, TotRmsAbvGrd)	0.616423
15	(SalePrice, TotalBsmtSF)	0.613581
8	(2ndFlrSF, HalfBath)	0.609707
35	(1stFlrSF, SalePrice)	0.605852
6	(GarageCars, OverallQual)	0.600671
7	(GrLivArea, OverallQual)	0.593007
9	(YearBuilt, YearRemodAdd)	0.592855
12	(GarageCars, GarageYrBlt)	0.588920
0	(OverallQual, YearBuilt)	0.572323

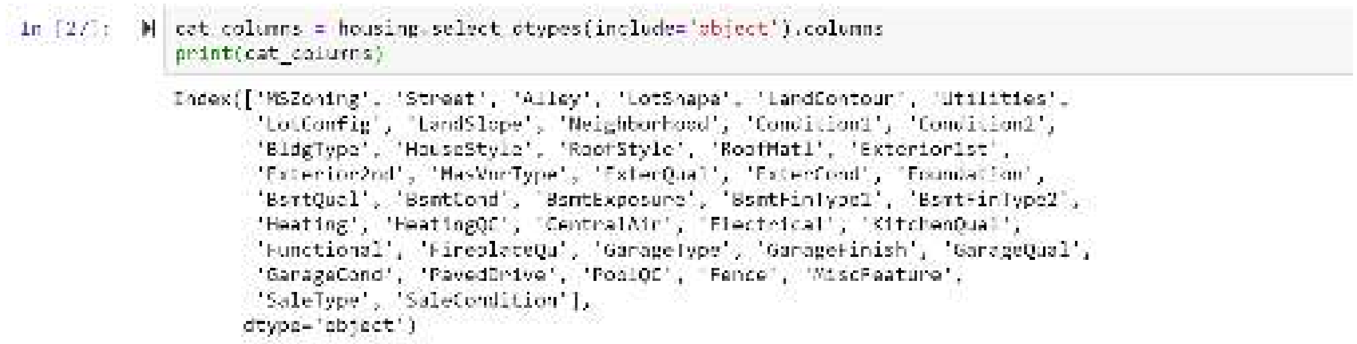
We can see that we have many highly correlated attributes and these results confirm our common sense analysis.

📝 We can take some notes here for the feature selection phase where we are going to drop the highly correlated variables. For example, *GarageCars* and *GarageArea* are highly correlated but since *GarageCars* has a higher correlation with the target variable, *SalePrice*, we are going to keep *GarageCars* and drop *GarageArea*. We will also drop the attributes that have almost no correlation with price, like *MoSold*, *3SsnPorch* and *BsmtFinSF2*.

Explore Categorical Attributes

Let's print again the names of the categorical columns again and then handpick some of the interesting ones for visual analysis.

```
cat_columns = housing.select_dtypes(include='object').columns
print(cat_columns)
```

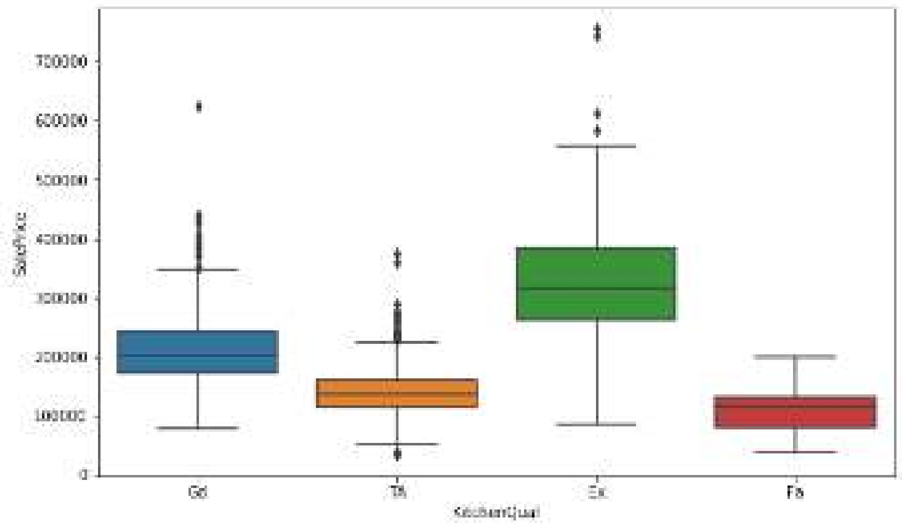


```
In [27]:  cat_columns = housing.select_dtypes(include='object').columns
          print(cat_columns)

          Index(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
                 'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
                 'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
                 'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
                 'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
                 'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
                 'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
                 'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature',
                 'SaleType', 'SaleCondition'],
                dtype='object')
```

Say we want to look at the impact of *KitchQual* on price:

```
var = housing['KitchenQual']
f, ax = plt.subplots(figsize=(10,6))
sns.boxplot(y=housing.SalePrice, x=var)
plt.show()
```

```
In [28]:  var = housing['KitchenQual']
          f, ax = plt.subplots(figsize=(10,6))
          sns.boxplot(y=housing.SalePrice, x=var)
          plt.show()
```



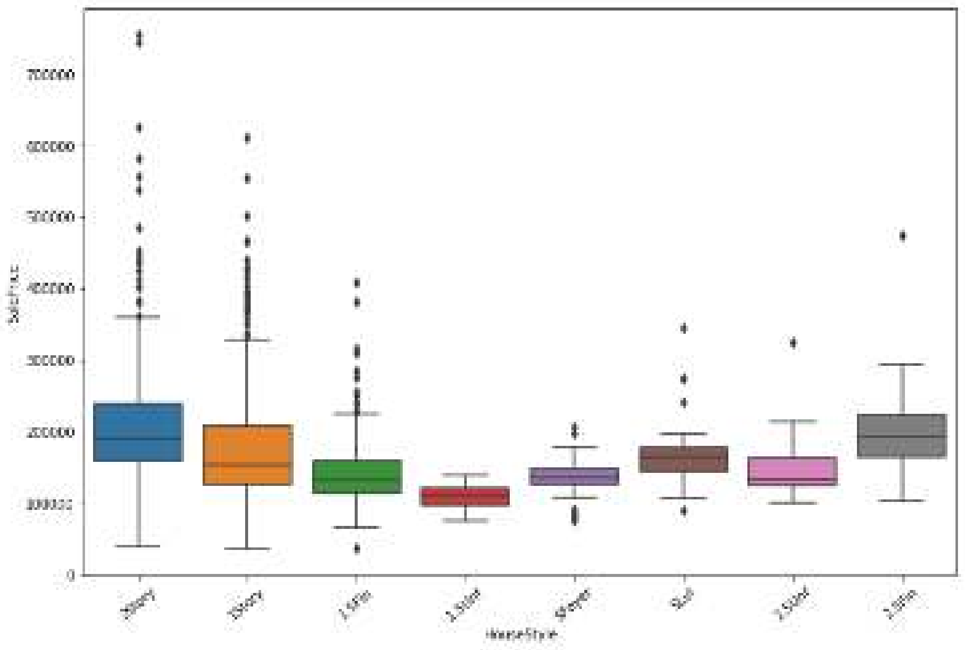
We can now see that *Ex* seems to be the more expensive option while *Fa* brings the prices down.

What about the style of the houses? Which styles do we have and how do they impact prices?

```
f, ax = plt.subplots(figsize=(12,8))
sns.boxplot(y=housing.SalePrice, x=housing.HouseStyle)
```

```
plt.xticks(rotation=40)
plt.show()
```

```
In [11]:  f, ax = plt.subplots(figsize=(17,8))
          sns.boxplot(y=housing.SalePrice, x=housing.HouseStyle)
          plt.xticks(rotation=40)
          plt.show()
```

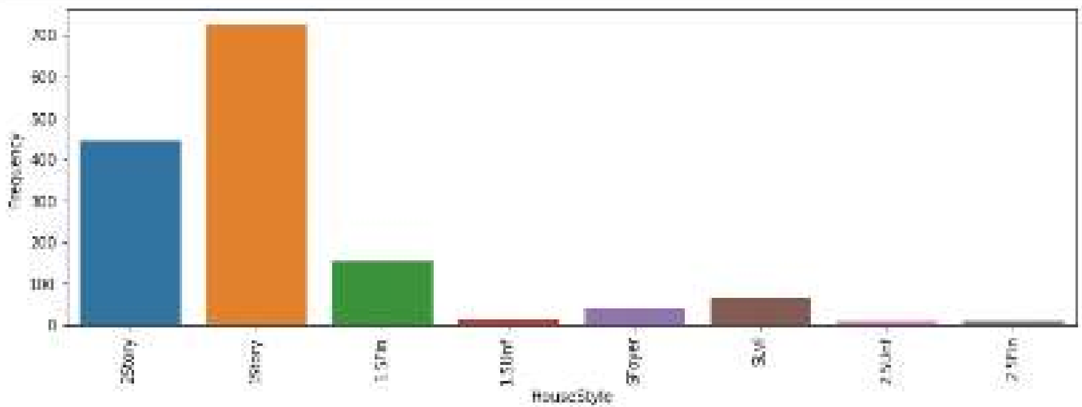


We can see that *2Story* houses have the highest variability in prices and they also tend to be more expensive, while *1.5Unf* are the cheapest option.

Say we want to get the frequency for each of these types, we can use the `countplot()` method from *sns* like so:

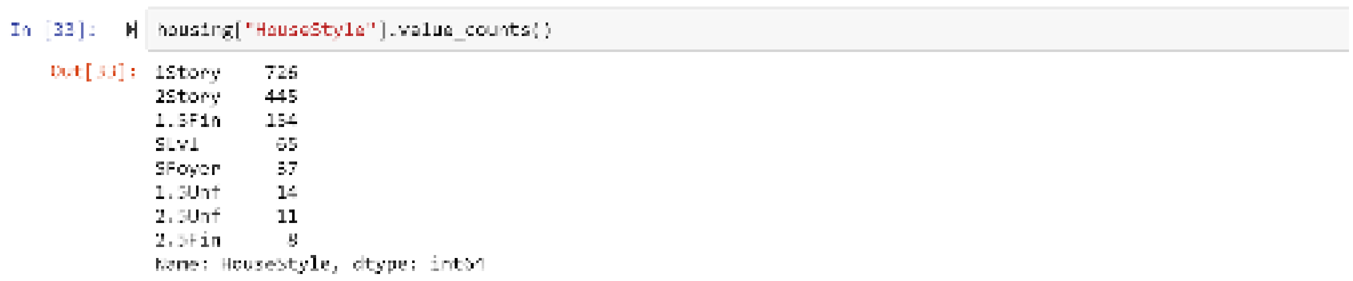
```
# Count of categories within HouseStyle attribute
fig = plt.figure(figsize=(12, 4))
sns.countplot(x='HouseStyle', data=housing)
plt.xticks(rotation=90)
plt.show()
```

```
In [32]:  ## Count of categories within HouseStyle attribute
          fig = plt.figure(figsize=(12.5,4))
          sns.countplot(x='HouseStyle', data=housing)
          plt.xticks(rotation=90)
          plt.ylabel('Frequency')
          plt.show()
```



Now we know that most of the houses are *1Story* type houses. Say we do not want a frequency distribution plot, but only the exact count for each category, we can get that easily from the DataFrame directly:

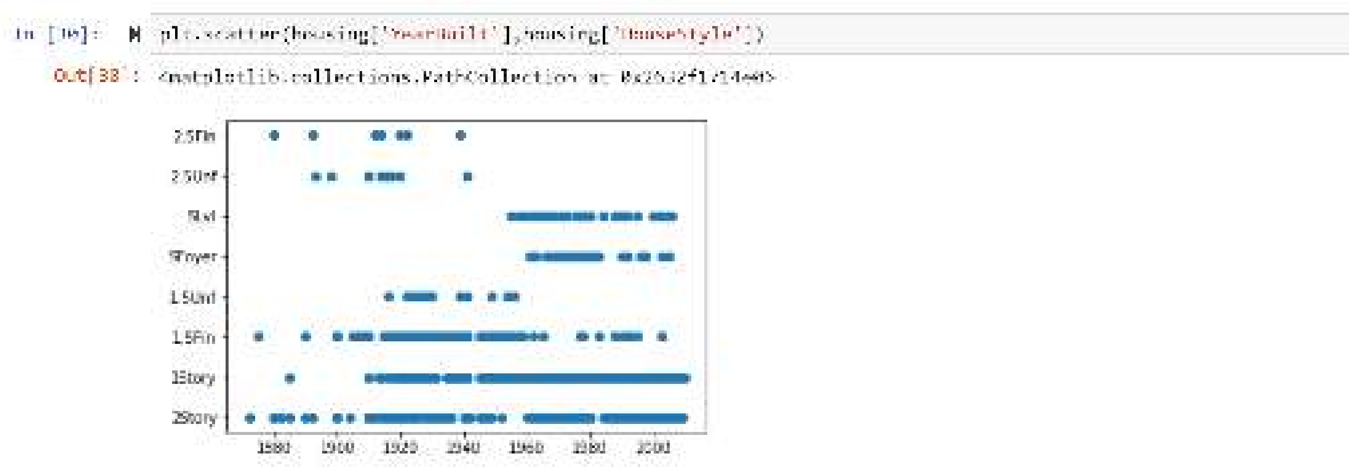
```
housing["HouseStyle"].value_counts()
```



```
In [33]:  housing['HouseStyle'].value_counts()
Out[33]: 1Story    726
         2Story    445
         1.5Fin    154
         SLvl       65
         SFoyer     37
         1.5Unf     14
         2.5Unf     11
         2.5Fin      8
         Name: HouseStyle, dtype: int64
```

We are also curious to see if the style of the houses has changed over the years, so let's plot the two variables against each other.

```
plt.scatter(housing['YearBuilt'],housing['HouseStyle'])
```



```
In [38]:  plt.scatter(housing['YearBuilt'],housing['HouseStyle'])
Out[38]: <matplotlib.collections.PathCollection at 0x2532f1714e0>
```

Now we know that *2Story* and *1Story* have been there for ages and they continue to be built while *SFoyer* and *SLvl* are relatively newer styles. We can also notice that *2.5Fin*, *2.5Unf* and *1.5Unf* are deprecated styles.

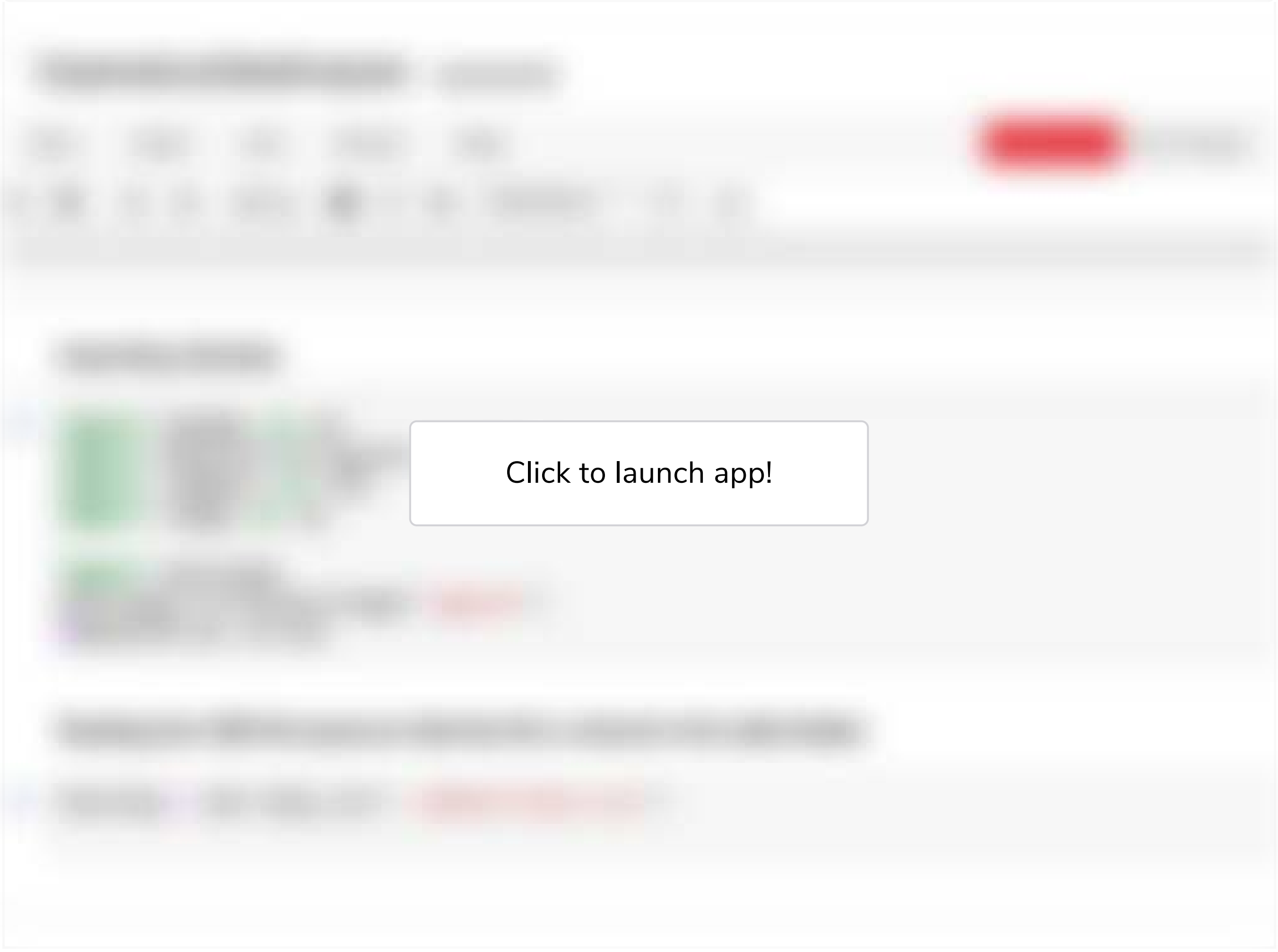
Jupyter Notebook

You can see the instructions running in the Jupyter Notebook below:

How to Use a Jupyter NoteBook?

- Click on **“Click to Launch”** button to work and see the code running live in the notebook.
- You can click to open the **Jupyter Notebook in a new tab**.
- Go to files and click *Download as* and then choose the format of the file to **download** . You can choose Notebook(.ipynb) to download the file and work locally or on your personal Jupyter Notebook.
- ⚠ The notebook **session expires after 30 minutes of inactivity**. It will reset if there is no interaction with the notebook for 30 consecutive minutes.

Your app can be found at: https://1dgnrwwoynk5m-live-app.educative.run/notebooks/ExploratoryDataAnalysis.ipynb



Aaand we are done with the initial *Exploratory Analysis*! We will move onto *Data Preprocessing* in the next lesson.