# Forecasting Home Prices in Austin, TX Using Time-Series Data

Austin, TX is the fastest growing city in the country. [From 2019 to 2020, the city saw its population grow by 3%, the largest among metropolitan areas with at least 1 million people. (https://www.bizjournals.com/austin/news/2021/05/04/census-data-austin-metro-population.html)](https://www.bizjournals.com/austin/news/2021/05/04/census-data-austin-metro-population.html). As a consultant for an Austin-based real estate investment firm, I will be advising on what 5 zip codes to invest their money in. For the purposes of accurate forecasting, we'll be forecasting 2-year growth rates, using predicted prices for 2018 and 2019.

What makes a zip code a top 5 zip code? For this firm, it pure growth rate. The higher the growth rate, the better. With such a short time horizon of two years, the firm is willing to take big swings to produce higher returns for their investors. Using monthly data from 2010 - 2017, we'll create a time series model to help us forecast what prices would be in 2018 and 2019. The zip codes with the highest growth from 2017 to 2019 will be the optimal investments for this firm.

We'll also explore the housing market in the broader metro area, and look at how Austin prices and growth compares to other major cities.

## Step 1: Load the Data/Filtering for Chosen Zipcodes

In [1]:
```
#importing libraries
import pandas as pd
import numpy as np
import itertools
import warnings
warnings.filterwarnings('ignore')

import matplotlib
import matplotlib.pyplot as plt
from matplotlib.pylab import rcParams
plt.style.use('ggplot')
import seaborn as sns

import plotly as py
import plotly.express as px
import plotly.graph_objects as go

import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose

import pmdarima as pm
from pmdarima.arima import auto_arima
from pmdarima import model_selection
from pmdarima.utils import decomposed_plot
from pmdarima.arima import decompose
```

In [23]:
```
df = pd.read_csv('zillow_data.csv')

#renaming region name column to zipcode
df.rename({'RegionName': 'Zipcode'}, axis='columns', inplace=True)
df.head()
```

Out[23]:

| | RegionID | Zipcode | City | State | Metro | CountyName | SizeRank | 1996-04 | 1996-05 | 199 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 84654 | 60657 | Chicago | IL | Chicago | Cook | 1 | 334200.0 | 335400.0 | 3365 |
| **1** | 90668 | 75070 | McKinney | TX | Dallas-Fort Worth | Collin | 2 | 235700.0 | 236900.0 | 2367 |
| **2** | 91982 | 77494 | Katy | TX | Houston | Harris | 3 | 210400.0 | 212200.0 | 2122 |
| **3** | 84616 | 60614 | Chicago | IL | Chicago | Cook | 4 | 498100.0 | 500900.0 | 5031 |
| **4** | 93144 | 79936 | El Paso | TX | El Paso | El Paso | 5 | 77300.0 | 77300.0 | 773 |

5 rows × 272 columns

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14723 entries, 0 to 14722
Columns: 272 entries, RegionID to 2018-04
dtypes: float64(219), int64(49), object(4)
memory usage: 30.6+ MB
```

Our Zillow data has nearly 15K zipcodes, accounting for about 35% of the total number of zipcodes in the U.S. The data captures the median home sale price for each month from April 1996 - April 2018. To get a clearer picture and to make data processing easier, I'll drop 1996 and 2018 so I only have complete years in the data set.

In [4]:
```
#creating austin dataframe, also filtering by state because there are some Austir
austin = df.loc[(df['City'] == 'Austin') & (df['State'] == 'TX')]
austin.head()
```

Out[4]:

| | RegionID | Zipcode | City | State | Metro | CountyName | SizeRank | 1996-04 | 1996-05 | 1996-0 |
|---|---|---|---|---|---|---|---|---|---|---|
| **66** | 92617 | 78704 | Austin | TX | Austin | Travis | 67 | 221300.0 | 221100.0 | 221000. |
| **98** | 92654 | 78745 | Austin | TX | Austin | Travis | 99 | 135000.0 | 134200.0 | 133800. |
| **422** | 92667 | 78758 | Austin | TX | Austin | Travis | 423 | 129000.0 | 128300.0 | 127500. |
| **432** | 92651 | 78741 | Austin | TX | Austin | Travis | 433 | 93800.0 | 93600.0 | 93500. |
| **502** | 92662 | 78753 | Austin | TX | Austin | Travis | 503 | 111300.0 | 110600.0 | 109900. |

5 rows × 272 columns

In [5]:
```
#let's see how many zipcodes in Austin there are:

print('# of Zipcodes in Austin, TX: ', austin['Zipcode'].nunique())
```

```
# of Zipcodes in Austin, TX:  38
```

In [6]:
```
def melt_data(df, time_series = None):
    '''
    The melt data method is essentially a pivot table. Since our monthly data is 
    melting the data moves each date into its own row and aggregates the median h
    When time_series is set to 'yes', the data is transformed into strictly a tim
    that it only has dates and the average median home price. Setting the paramet
    retain the other columns, like zipcode
    '''
    melted = pd.melt(df, id_vars=['RegionID','Zipcode', 'City', 'State', 'Metro',
    melted['time'] = pd.to_datetime(melted['time'], infer_datetime_format=True)
    melted = melted.dropna(subset=['value'])
    if time_series == 'yes':
        return melted.groupby('time').aggregate({'value':'mean'})
    else:
        return melted
```

In [7]:
```
#austin_df will retain categorical info
austin_df = melt_data(austin)
austin_df.rename(columns = {'time':'date','value':'median_sale_price'}, inplace =
austin_df.head()
```

Out[7]:

| | RegionID | Zipcode | City | State | Metro | CountyName | SizeRank | date | median_sale_price |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 92617 | 78704 | Austin | TX | Austin | Travis | 67 | 1996-04-01 | 221300.0 |
| **1** | 92654 | 78745 | Austin | TX | Austin | Travis | 99 | 1996-04-01 | 135000.0 |
| **2** | 92667 | 78758 | Austin | TX | Austin | Travis | 423 | 1996-04-01 | 129000.0 |
| **3** | 92651 | 78741 | Austin | TX | Austin | Travis | 433 | 1996-04-01 | 93800.0 |
| **4** | 92662 | 78753 | Austin | TX | Austin | Travis | 503 | 1996-04-01 | 111300.0 |

In [8]:
```
#austin_ts will just be used to look at time series aggregation
austin_ts = melt_data(austin, time_series = 'yes')
austin_ts.rename(columns = {'value':'median_sale_price'}, inplace = True)
austin_ts.head()
```

Out[8]:

| | median_sale_price |
|---|---|
| **time** | |
| **1996-04-01** | 217871.052632 |
| **1996-05-01** | 217673.684211 |
| **1996-06-01** | 217610.526316 |
| **1996-07-01** | 217657.894737 |
| **1996-08-01** | 217792.105263 |

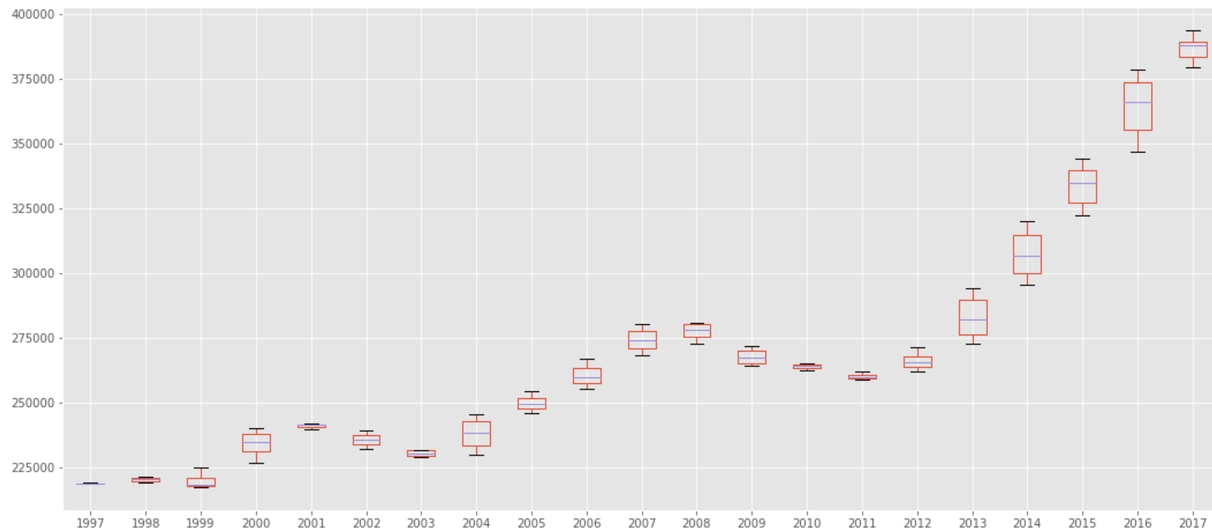
In [9]:
```
#Let's look at the overall trend of the avg. median sale price
fig = px.line(austin_ts, x=austin_ts.index, y = 'median_sale_price',
              title = 'Avg. Median Sale Price in Austin, TX')
fig.update_xaxes(rangeslider_visible=True)
fig.show()
```


Interestingly, from its pre-recession peak, the average median sale price of homes in Austin only fell by 20K at its lowest during the crisis. Although affected, prices were much less impacted here than in many other metropolitan areas. Since hitting a low in June 2011, prices have been rising rapidly, increasing by 56% between 06/2011 and 04/2018.

Now let's look at a box plot of prices from 1997 - 2017:

In [10]:
```
#both 1996 and 2018 only start in April, let's slice those off from our dataset
austin_ts_slice = austin_ts['1997':'2017']

#try making an annual box plot here as well
year_groups = austin_ts_slice.groupby(pd.Grouper(freq = 'A'))

austin_ts_annual = pd.DataFrame()

for yr, group in year_groups:
    austin_ts_annual[yr.year] = group.values.ravel()

austin_ts_annual.boxplot(figsize = (18,8))
```

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x201495fbcc0>




In [11]:
```
#Let's see how average price has increased in Austin from 2011 to 2017
(austin_ts_annual[2017].mean() - austin_ts_annual[2011].mean())/austin_ts_annual[
```

Out[11]: 0.488396361071837

Over the years, we can see that the range of prices decreases in down years. In 1997 - 1998, 2001 - 2003, and 2009 - 2011, the boxplots are much smaller in size compared to growth years. Prices across the board are clearly affected in recessions, while the top end of home prices take off in years the market is growing. Overall, we see that prices increased by almost 43% from 2011 - 2017

## Data Exploration: Comparing other cities in the Austin

# Metro area

Let's look at other cities in the Austin metro area to see if prices behave similarly.

In [12]:
```
austin_metro_df = df[(df['Metro'] == 'Austin') & (df['State'] == 'TX')]
austin_metro_df.reset_index(inplace = True)
austin_metro_df.drop(columns = 'index', inplace = True)
austin_metro_df.head()
```

Out[12]:

| | RegionID | Zipcode | City | State | Metro | CountyName | SizeRank | 1996-04 | 1996-05 | 1996 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 92593 | 78660 | Pflugerville | TX | Austin | Travis | 19 | 138900.0 | 138600.0 | 13840 |
| **1** | 92551 | 78613 | Cedar Park | TX | Austin | Williamson | 33 | 169600.0 | 169000.0 | 16860 |
| **2** | 92617 | 78704 | Austin | TX | Austin | Travis | 67 | 221300.0 | 221100.0 | 22100 |
| **3** | 92598 | 78666 | San Marcos | TX | Austin | Hays | 78 | 103100.0 | 103000.0 | 10300 |
| **4** | 92654 | 78745 | Austin | TX | Austin | Travis | 99 | 135000.0 | 134200.0 | 13380 |

5 rows × 272 columns

In [13]:
```
#let's compare to 5 largest cities in metro area, excluding Austin
austin_metro_sorted = austin_metro_df.sort_values('SizeRank')
austin_metro_sorted = austin_metro_sorted[austin_metro_sorted['City'] != 'Austin'
```

In [14]:
```
austin_metro_sorted.head()
```

Out[14]:

| | RegionID | Zipcode | City | State | Metro | CountyName | SizeRank | 1996-04 | 1996-05 | 199 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 92593 | 78660 | Pflugerville | TX | Austin | Travis | 19 | 138900.0 | 138600.0 | 1384 |
| **1** | 92551 | 78613 | Cedar Park | TX | Austin | Williamson | 33 | 169600.0 | 169000.0 | 1686 |
| **3** | 92598 | 78666 | San Marcos | TX | Austin | Hays | 78 | 103100.0 | 103000.0 | 1030 |
| **5** | 92576 | 78641 | Leander | TX | Austin | Williamson | 284 | 153600.0 | 152900.0 | 1524 |
| **10** | 92597 | 78664 | Round Rock | TX | Austin | Williamson | 515 | 133100.0 | 132700.0 | 1324 |

5 rows × 272 columns

So our top 5 cities ranked by size and excluding Austin are:

- 1) Pflugerville
- 2) Cedar Park
- 3) San Marcos
- 4) Leander

- 5) Round Rock

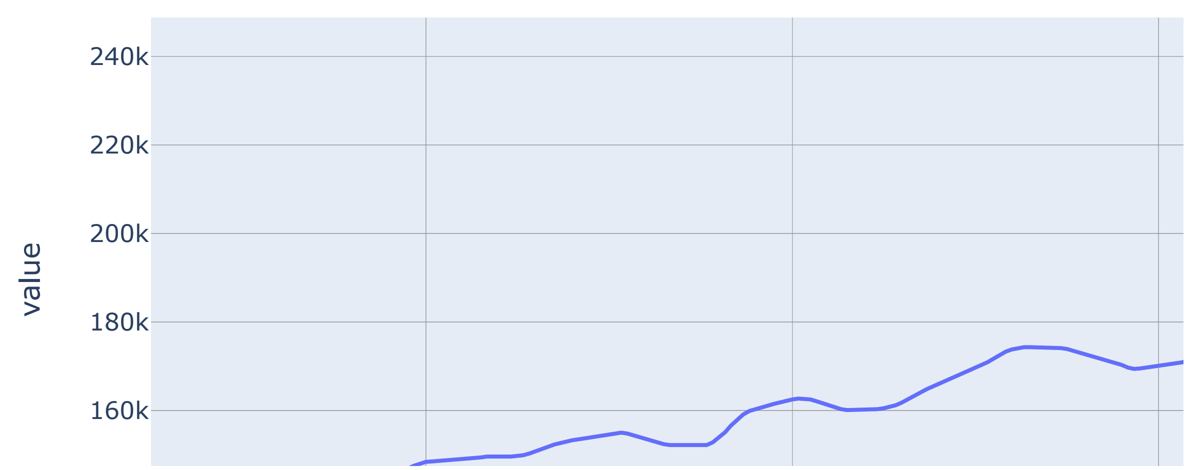
In [15]:
```
top_5 = ['Pflugerville', 'Cedar Park', 'San Marcos', 'Leander', 'Round Rock']

austin_metro_sorted_top5 = austin_metro_sorted[austin_metro_sorted['City'].isin(t
```

In [16]:
```
Pflugerville = austin_metro_sorted_top5.loc[austin_metro_sorted_top5['City'] == '
Cedar_Park = austin_metro_sorted_top5.loc[austin_metro_sorted_top5['City'] == 'Ce
San_Marcos = austin_metro_sorted_top5.loc[austin_metro_sorted_top5['City'] == 'Sa
Leander = austin_metro_sorted_top5.loc[austin_metro_sorted_top5['City'] == 'Leand
Round_Rock = austin_metro_sorted_top5.loc[austin_metro_sorted_top5['City'] == 'Ro
```

In [17]:
```
melt_Pflugerville = melt_data(Pflugerville, time_series = 'yes')
melt_Cedar_Park = melt_data(Cedar_Park, time_series = 'yes')
melt_San_Marcos = melt_data(San_Marcos, time_series = 'yes')
melt_Leander = melt_data(Leander, time_series = 'yes')
melt_Round_Rock = melt_data(Round_Rock, time_series = 'yes')
```

In [18]:
```
def line_chart(ts):
    fig = px.line(ts, x=ts.index, y = ts['value'], title = 'Avg. Median Sale Pric
    fig.update_xaxes(rangeslider_visible=True)
    fig.show()
```

In [280]:
```
plot_dict = {'Pflugerville':melt_Pflugerville, 'Cedar_Park':melt_Cedar_Park, 'San
for key, val in plot_dict.items():
    print('City: ', key)
    line_chart(val)

#come back to clean all this up later - make code more streamlined
```




We see that these 5 cities have larger drops in the 2003-2004 recession than Austin did. However, the increases post-2011 are of the same relative magnitude as Austin proper:

In [20]:
```
plot_list = [melt_Pflugerville, melt_Cedar_Park, melt_San_Marcos, melt_Leander, r
for city in plot_list:
    city_slice = df['2011':'2017']
    year_groups = city_slice.groupby(pd.Grouper(freq = 'A'))
    city_ts_annual = pd.DataFrame()

    for yr, group in year_groups:
        city_ts_annual[yr.year] = group.values.ravel()

    print(((city_ts_annual[2017].mean() - city_ts_annual[2011].mean())/city_ts_ar
```

```
42.62631363019532
42.62631363019532
42.62631363019532
42.62631363019532
42.62631363019532
```

# Percentage Growth from 2011 - 2017: Top 5 Cities by Size

- Cedar Park: 47%
- Leander: 46.5%
- Round Rock: 42.6%
- Pflugerville: 39%
- San Marcos: 38.5%

These compare favorably to Austin in general, which increased by 42.6% from 2011 - 2017. Although the prices in these cities don't reach the top prices in Austin, we can see that the growth in Austin has spurred growth in the surrounding cities in the metropolitan area.

# Data Exploration: Comparing Austin to other major cities

It might also be useful to compare Austin's housing market to other big and growing cities across the country. Has Austin performed as well as others? Was it affected more or less by the recession than other big cities?

In [24]:
```
nyc = df.loc[df['City'] == 'New York']
dc = df.loc[df['State'] == 'DC']
sf = df.loc[df['City'] == 'San Francisco']
la = df.loc[df['City'] == 'Los Angeles']

melt_nyc = melt_data(nyc, time_series = 'yes')
melt_dc = melt_data(dc, time_series = 'yes')
melt_sf = melt_data(sf, time_series = 'yes')
melt_la = melt_data(la, time_series = 'yes')
```

In [25]:
```
line_chart(melt_nyc)
```

In [26]:
```
line_chart(melt_dc)
```

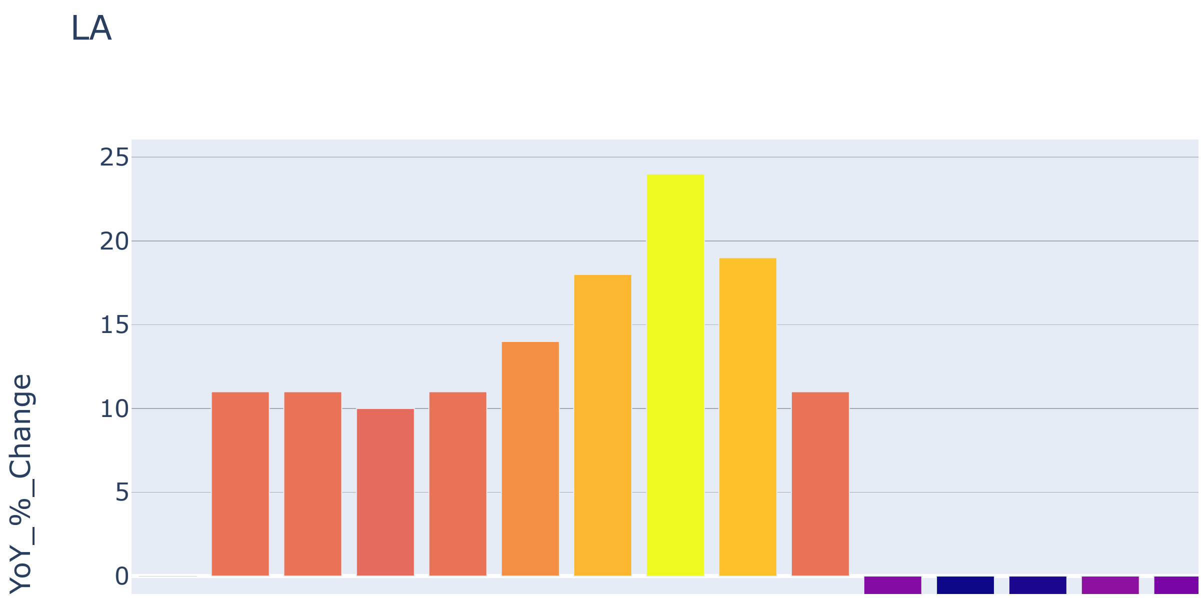
In [27]:
```
line_chart(melt_sf)
```

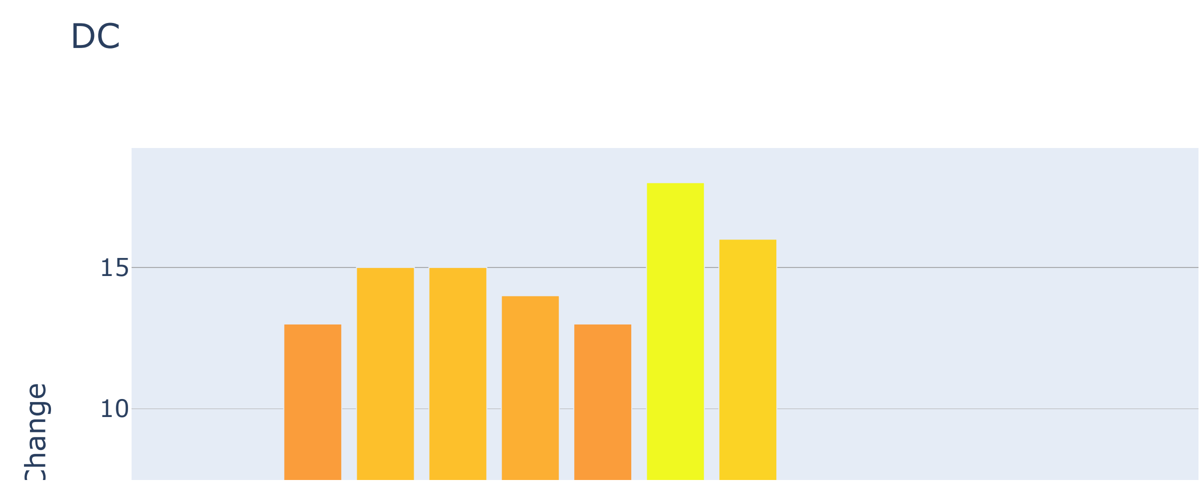
In [28]:
```
line_chart(melt_la)
```

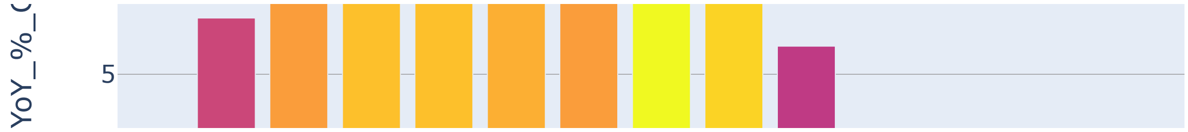

Austin certainly has a similar trend to these other cities, but the drop appears to be much less significant.

In [29]:
```
melt_la.rename(columns = {'value':'median_sale_price'}, inplace = True)
melt_dc.rename(columns = {'value':'median_sale_price'}, inplace = True)
melt_nyc.rename(columns = {'value':'median_sale_price'}, inplace = True)
melt_sf.rename(columns = {'value':'median_sale_price'}, inplace = True)
```

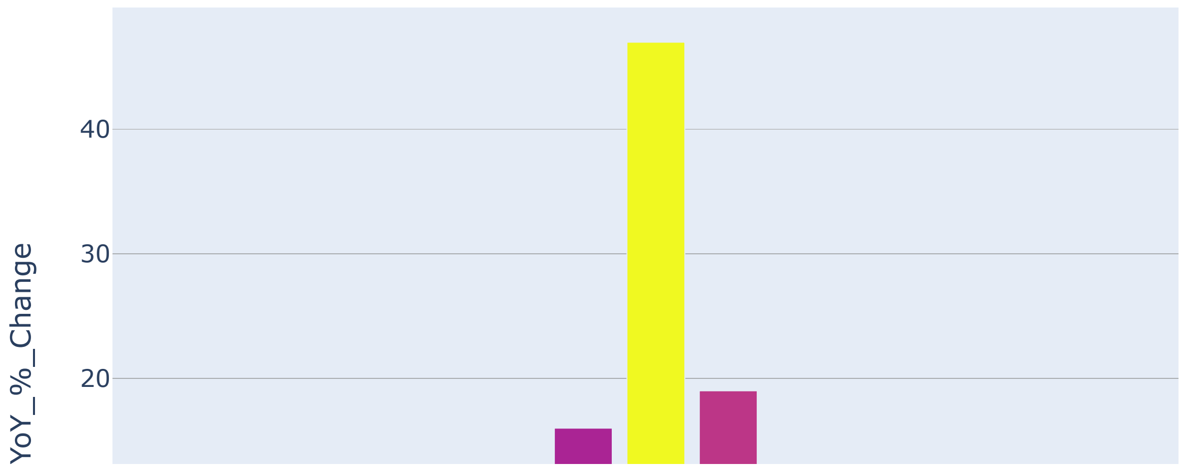
In [30]:
```
year_group_dict = {'LA': melt_la, 'DC': melt_dc, 'NYC': melt_nyc, 'SF': melt_sf,
for key, val in year_group_dict.items():
    sliced = val['1997':'2017']
    year_groups_mean = sliced.groupby(pd.Grouper(freq = 'A')).mean()
    year_groups_mean['YoY_%_Change'] = year_groups_mean['median_sale_price'].pct_
    year_groups_mean['YoY_%_Change'][1:] = year_groups_mean['YoY_%_Change'][1:].a

    fig = px.bar(year_groups_mean, x=year_groups_mean.index, y='YoY_%_Change', ti
    fig.show()
```



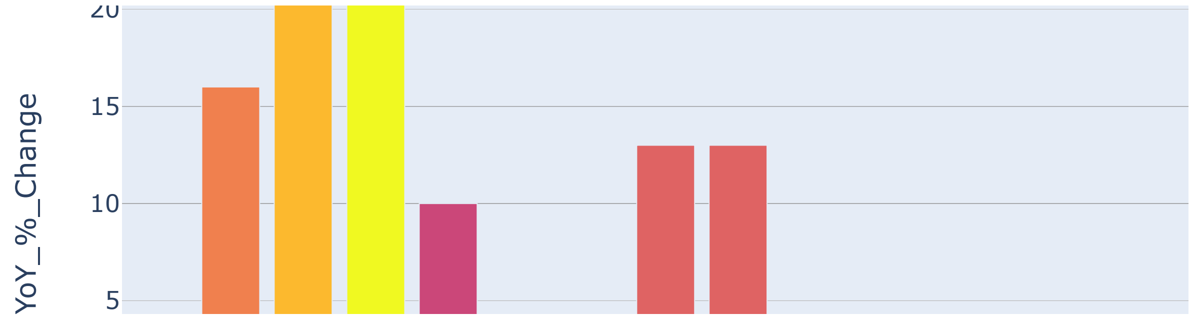



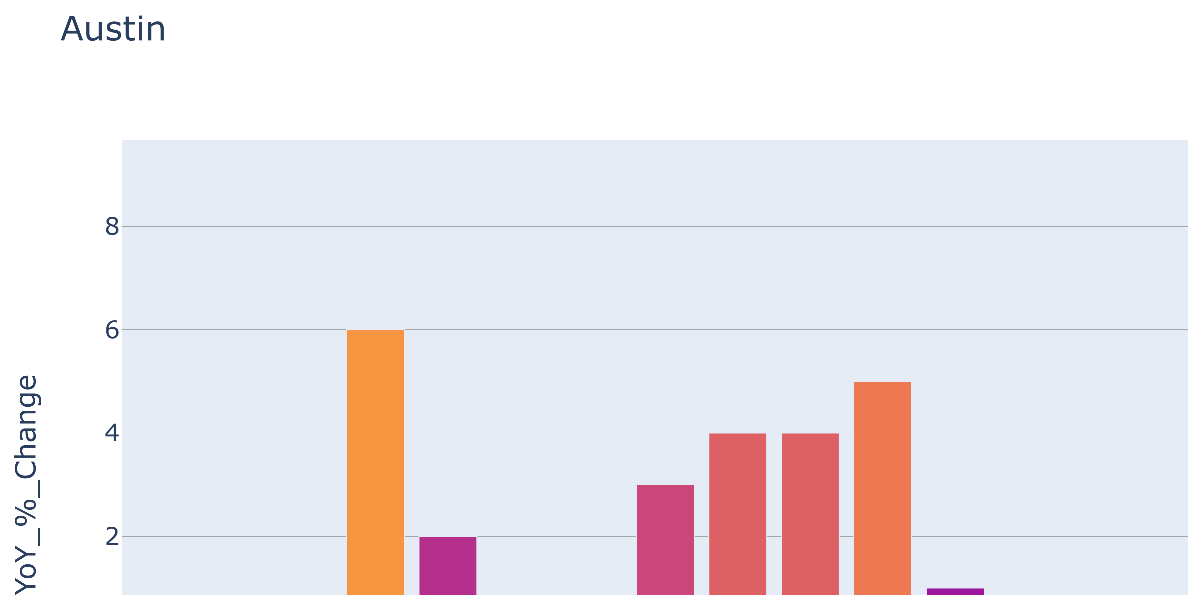



NYC




SF



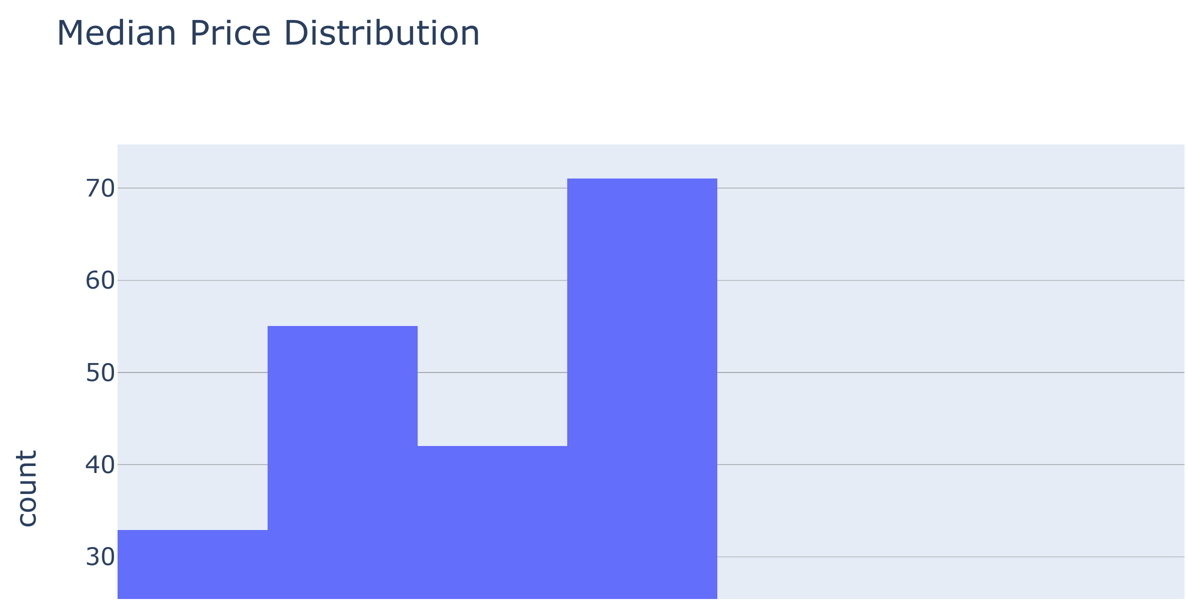




All cities had some decline in growth during the recession besides NYC, which saw prices essentially hold steady. The other 4 major cities had rebounds orders of magnitude larger than Austin on a YoY basis, but Austin prices are clearly growing, and there will be more room to go in the coming years. DC, a smaller city closer to Austin's population, had similar YoY growth rates since 2011, averaging about 7% for the last 6 years in the dataset, which is roughly what Austin saw.

# Preparing to Model

Now that we've explored the data, let's dive into time series modeling and figuring out we can find the 5 best zipcodes in Austin to invest in. As the investment firm is only looking to invest for 2 years, they simply want the 5 zipcodes with the highest projected 2 year growth rates, regardless of risk. To get started, let's first look at the distribution of our time series data.

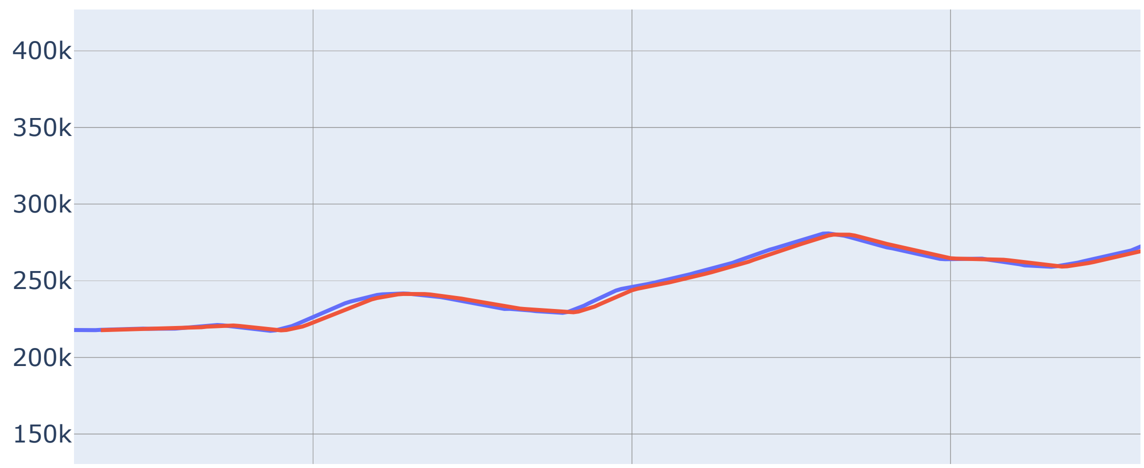
In [31]:
```
fig = px.histogram(austin_ts, x='median_sale_price',
                   title = 'Median Price Distribution')
fig.show()
```




Our data is skewed to the right so we don't have a normally distributed dataset. Prior to modeling, we'll need to log transform this data to make it more normal.

Next, let's see if our data is stationary or not. Time series models assume that your data is stationary, so it's important we can get a constant mean across all time periods.

In [32]:
```
roll_mean = austin_ts.rolling(window = 6, center = False).mean()
roll_std = austin_ts.rolling(window = 6, center = False).std()

fig = go.Figure()
fig.add_trace(go.Scatter(x=austin_ts.index, y=austin_ts['median_sale_price'],
                    mode='lines',
                    name='Original'))
fig.add_trace(go.Scatter(x=roll_mean.index, y=roll_mean['median_sale_price'],
                    mode='lines',
                    name='Rolling Mean'))
fig.add_trace(go.Scatter(x=roll_std.index, y=roll_std['median_sale_price'],
                    mode='lines', name='Rolling Std. Deviation'))

fig.show()
```




There's certainly a trend in our data, and the rolling mean isn't constant, implying that this is non-stationary data. Let's use a Dickey-Fuller test to confirm.

In [33]:
```
def stationarity_test(ts):
    '''
    Courtesy of Lindsey Berlin.
    This function takes in a time series dataframe and performs a Dickey-Fuller t
    The null hypothesis is that the data is non-stationary, so when we can reject
    we have stationary data.
    '''
    dftest = adfuller(ts)

    dfoutput = pd.Series(dftest[0:4], index = ['Test Statistic', 'p-value', '#lag
    for key, value in dftest[4].items():
        dfoutput['Critical Value (%s)' %key] = value
    print('Results of Dickey-Fuller test: \n')

    return print(dfoutput)
```

In [34]:
```
austin_ts = austin_ts['1997': '2017']
stationarity_test(austin_ts)
```

```
Results of Dickey-Fuller test: 

Test Statistic                   0.632548
p-value                          0.988391
#lags used                      15.000000
Number of Observations Used    236.000000
Critical Value (1%)             -3.458366
Critical Value (5%)             -2.873866
Critical Value (10%)            -2.573339
dtype: float64
```

Our p-value here is much greater than the .05 cutoff, meaning that we fail to reject the null hypothesis that this data is non-stationary.

Let's decompose our time series to identify the seasonality from the overall trend.

In [35]:
```
def decompose(ts, model = 'additive'):
    '''
    This function takes in a time series dataframe and runs the statsmodels decom
    while also graphing our results.

    Inputs: ts = time series
            model = 'additive' or 'multiplicative'
    Output: 4 graphs illustrating the trend, seasonality, and residuals
    '''

    decomposition = sm.tsa.seasonal_decompose(ts, model=model)
    trend = decomposition.trend
    seasonal = decomposition.seasonal
    residual = decomposition.resid


    plt.figure(figsize=(12,8))
    plt.subplot(411)
    plt.plot(ts, label='Original', color='blue')
    plt.legend(loc='best')
    plt.subplot(412)
    plt.plot(trend, label='Trend', color='blue')
    plt.legend(loc='best')
    plt.subplot(413)
    plt.plot(seasonal,label='Seasonality', color='blue')
    plt.legend(loc='best')
    plt.subplot(414)
    plt.plot(residual, label='Residuals', color='blue')
    plt.legend(loc='best')
    plt.tight_layout()
```

In [36]:
```
decompose(austin_ts)
```


We can see a clear upward trend as well as consistent seasonality every year. Although judging by the actual trend line, the seasonality does not appear to be significant.

Next, I'll use a pivot table to move each zipcode to its own column

In [37]:
```
austin_ts2 = austin_df.copy()
austin_ts2['date'] = pd.to_datetime(austin_ts2['date'])
austin_ts2.set_index('date', inplace = True)
austin_ts2.head()
```

Out[37]:

| | RegionID | Zipcode | City | State | Metro | CountyName | SizeRank | median_sale_price |
|---|---|---|---|---|---|---|---|---|
| **date** | | | | | | | | |
| **1996-04-01** | 92617 | 78704 | Austin | TX | Austin | Travis | 67 | 221300.0 |
| **1996-04-01** | 92654 | 78745 | Austin | TX | Austin | Travis | 99 | 135000.0 |
| **1996-04-01** | 92667 | 78758 | Austin | TX | Austin | Travis | 423 | 129000.0 |
| **1996-04-01** | 92651 | 78741 | Austin | TX | Austin | Travis | 433 | 93800.0 |
| **1996-04-01** | 92662 | 78753 | Austin | TX | Austin | Travis | 503 | 111300.0 |

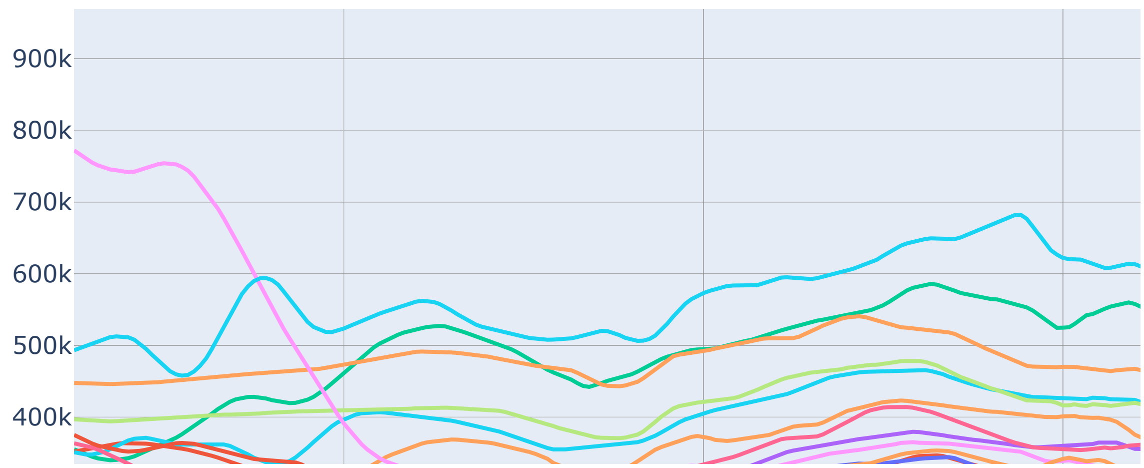
In [38]:
```
austin_zip = austin_ts2.pivot_table(index = 'date', columns = 'Zipcode', values =
austin_zip.columns = austin_zip.columns.astype(str)
austin_zip.head()
```

Out[38]:

| Zipcode | 78617 | 78702 | 78703 | 78704 | 78705 | 78717 | 78721 | 78722 | 78723 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **date** | | | | | | | | | | |
| **1996-04-01** | 121900.0 | 55600.0 | 355200.0 | 221300.0 | 197500.0 | 200800.0 | 69200.0 | 76200.0 | 97600.0 | 9 |
| **1996-05-01** | 120500.0 | 56700.0 | 351300.0 | 221100.0 | 199300.0 | 200400.0 | 68800.0 | 76400.0 | 99000.0 | 9 |
| **1996-06-01** | 119000.0 | 57900.0 | 347800.0 | 221000.0 | 201000.0 | 200500.0 | 68400.0 | 76500.0 | 100300.0 | 9 |
| **1996-07-01** | 117400.0 | 59300.0 | 344900.0 | 221000.0 | 202700.0 | 201100.0 | 68100.0 | 76700.0 | 101400.0 | 9 |
| **1996-08-01** | 116000.0 | 60800.0 | 342400.0 | 221300.0 | 204300.0 | 202300.0 | 67800.0 | 77000.0 | 101900.0 | 9 |

5 rows × 38 columns

In [39]:
```
fig = go.Figure()

for col in austin_zip.columns:
    fig.add_trace(go.Scatter(x=austin_zip.index, y=austin_zip[col],
                        mode='lines',
                        name='{col}'.format(col=col)))
fig.show()
```



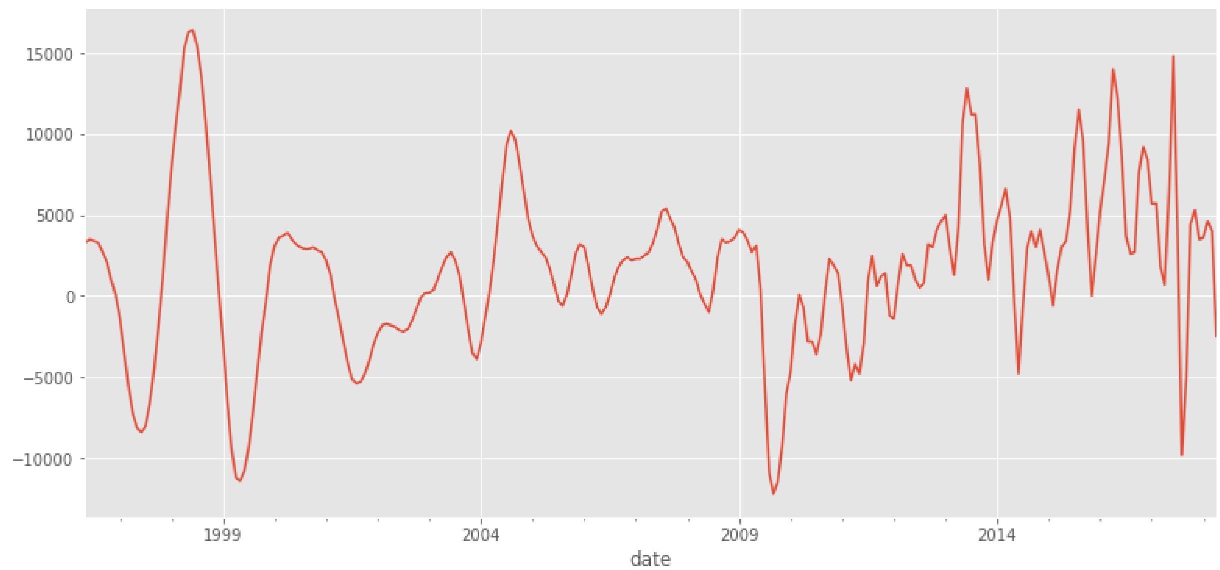

We see two zipcodes, 78746 and 78703, that have far higher average median prices than all the rest. The two zipcodes also seem to have the steepest line, illustrating that they're experiencing high growth. It will be interesting to see if the model shows these two to be in the top 5.

We'll most likely have to log transform this data, but let's first see how differencing affects the stationarity.

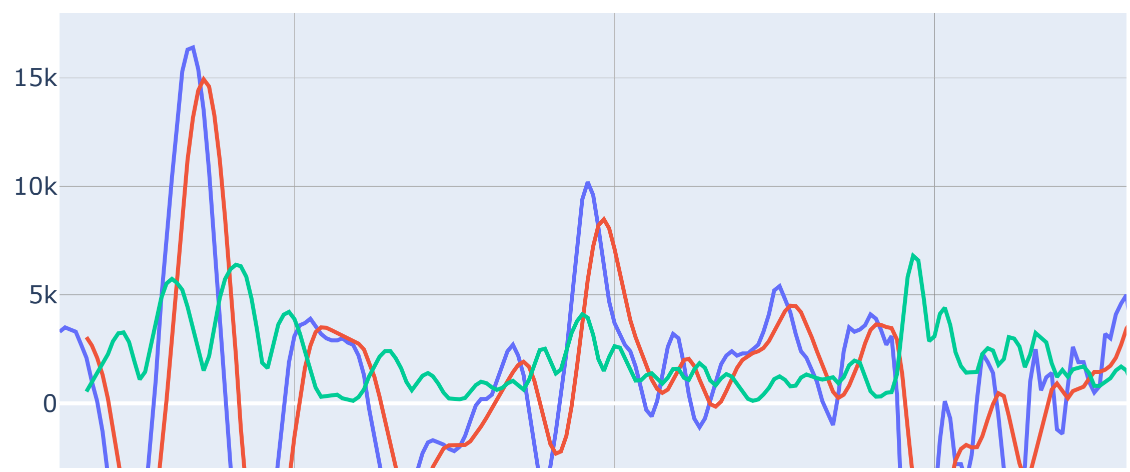
## Removing Stationarity

In [40]:
```
#using our top zipcode from observed data
austin_78746 = austin_zip['78746']
austin_78746_diff = austin_78746.diff(periods = 1)
austin_78746_diff.dropna(inplace = True)

plt.figure(figsize = (13, 6))
austin_78746_diff.plot()
```

Out[40]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2014a85c668>
```



In [41]:
```
#plotting the rolling mean and standard deviation with a window of 6
roll_mean_78746 = austin_78746_diff.rolling(window = 6, center = False).mean()
roll_std_78746 = austin_78746_diff.rolling(window = 6, center = False).std()

fig = go.Figure()
fig.add_trace(go.Scatter(x=austin_78746_diff.index, y=austin_78746_diff.values,
                    mode='lines',
                    name='Original'))
fig.add_trace(go.Scatter(x=roll_mean_78746.index, y=roll_mean_78746.values,
                    mode='lines',
                    name='Rolling Mean'))
fig.add_trace(go.Scatter(x=roll_mean_78746.index, y=roll_std_78746.values,
                    mode='lines', name='Rolling Std. Deviation'))

fig.show()
```



In [42]:
```
stationarity_test(austin_78746_diff)
```

```
Results of Dickey-Fuller test:

Test Statistic                  -3.203893
p-value                          0.019760
#lags used                      14.000000
Number of Observations Used    249.000000
Critical Value (1%)             -3.456888
Critical Value (5%)             -2.873219
Critical Value (10%)            -2.572994
dtype: float64
```

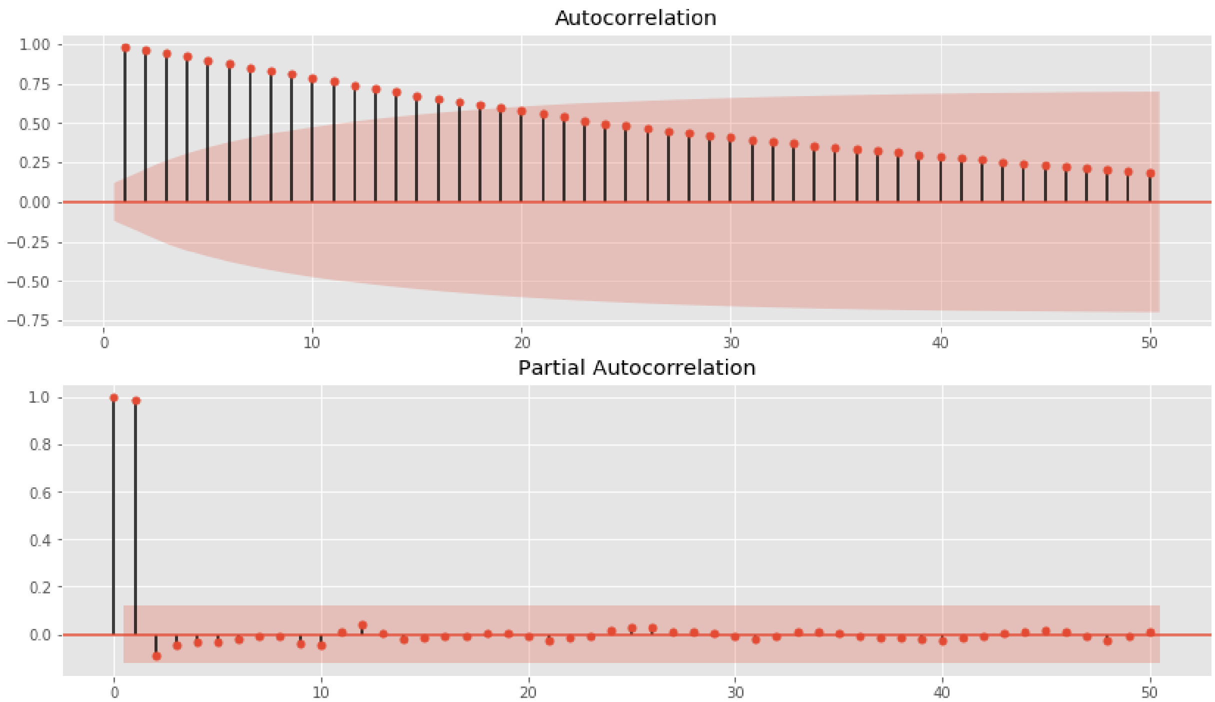
We see with this stationarity test that one-period differencing is enough to create a stationary dataset, which is encouraging. When modeling, we'll be sure to have a differencing of 1 in our logged dataset.

## Autocorrelation and Partial Autocorrelation

Autocorrelation and Partial Autocorrelation are used to illustrate how time lags are correlated with each other. If we see a spike at 12 lags, this might tell us that the data has a yearly seasonal relationship. For example, housing prices in June 2020 could be very correlated with June 2019 prices. This information can help us with deciding what parameters to use in our ARIMA models.

We'll use this undifferenced dataframe to see what the autocorrelation looks like.

In [43]:
```
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(14,8))

#ACF
plot_acf(austin_78746, lags=50, zero=False, ax=ax1)

#PACF
plot_pacf(austin_78746,lags=50, ax=ax2)
plt.show()
```




In the autocorrelation plot, up to about 18 lags, there is meaningful correlation amongst time lags. Partial autocorrelation shows correlation only at 1 and 2 lags. In our model, 1 might be our best autoregressive term.

# SARIMAX Modeling

Now, we'll use SARIMA model to forecast future home prices to estimate what growths rate will be.

In [265]:
```
from sklearn import metrics

#first, let's import our reporting metrics
#function courtesy of Lindsey Berlin

def report_metrics(y_true, y_pred, print_ = None):
    if print_ == 'yes':
        print("Explained Variance: ", metrics.explained_variance_score(y_true, y_
        print("MAE: ", np.exp(metrics.mean_absolute_error(y_true, y_pred)))
        print("RMSE: ", np.exp(metrics.mean_squared_error(y_true, y_pred, squarec
        print("r^2: ", metrics.r2_score(y_true, y_pred))
    return metrics.explained_variance_score(y_true, y_pred), np.exp(metrics.mean_
```

To get more accurate results and to not be overly influenced by fluctuations due to the recession, we're going to use data only in the years 2010 - 2017. To begin, we'll use a logged series for just the time series for the 78746 zipcode, and if it works, we'll extrapolate that model to the rest of the zipcodes in Austin.

In [45]:
```
austin_78746_2010 = austin_78746['2010': '2017']
austin_78746_2010 = austin_78746_2010.resample('MS').mean()
austin_78746_2010_log = np.log(austin_78746_2010)
```

In [46]:
```
#Define the p, d and q parameters to take any value between 0 and 3
#this takes a long time to run
p = d = q = range(0, 3)

#Generate all different combinations of p, q and q triplets
pdq = list(itertools.product(p, d, q))

#Generate all different combinations of seasonal p, q and q triplets
pdqs = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]
```

In [270]:
```
#I found thi resulted in better parameters than the out-of-the-box auto arima fur
#adding in trend t to see if I get different parameters
def pdq_test(ts):
    '''
    This function takes in a time series and performs a grid search to find the b
    for the p, d, and q parameters. It's best to use unaltered data here because
    parameters to deal with non-stationarity and seasonality.

    Input: Time-series
    Output: Optimal combination for order and seasonal order

    Function courtesy of Flatiron
    '''
    ans = []
    for comb in pdq:
        for combs in pdqs:
            try:
                mod = sm.tsa.statespace.SARIMAX(ts,
                                                order=comb,
                                                seasonal_order=combs,
                                                enforce_stationarity=False,
                                                enforce_invertibility=False,
                                                #trend = t to account for linear
                                                trend = 't',
                                                error_action = 'ignore')


                output = mod.fit()
                ans.append([comb, combs, output.aic])
                print('ARIMA {} x {}12 : AIC Calculated ={}'.format(comb, combs,
            except:
                continue
    # Find the parameters with minimal AIC value
    ans_df = pd.DataFrame(ans, columns=['pdq', 'pdqs', 'aic'])

    return ans_df.loc[ans_df['aic'].idxmin()]
```

In [222]:
```
def arima_fit(ts, order = (1,1,1), seasonal_order = (1,1,1,12), plot = None):
    '''
    Using the best parameters from the pdq_test for order and seasonal order, thi
    onto our time series data.

    Inputs: Time-series
            Order
            Seasonal order
    Output: Output from fitting SARIMAX model
    '''
    ARIMA_MODEL = sm.tsa.statespace.SARIMAX(ts,
                                    order=order,
                                    seasonal_order=seasonal_order,
                                    enforce_stationarity=False,
                                    enforce_invertibility=False,
                                        #trend = t to account for linear rela
                                        trend = 't',
                                       error_action = 'ignore')

    # Fit the model and print results
    output = ARIMA_MODEL.fit()

    if plot =='yes':

        print(output.summary().tables[1])

        output.plot_diagnostics(figsize=(15, 18))
        plt.show()

    return output
```

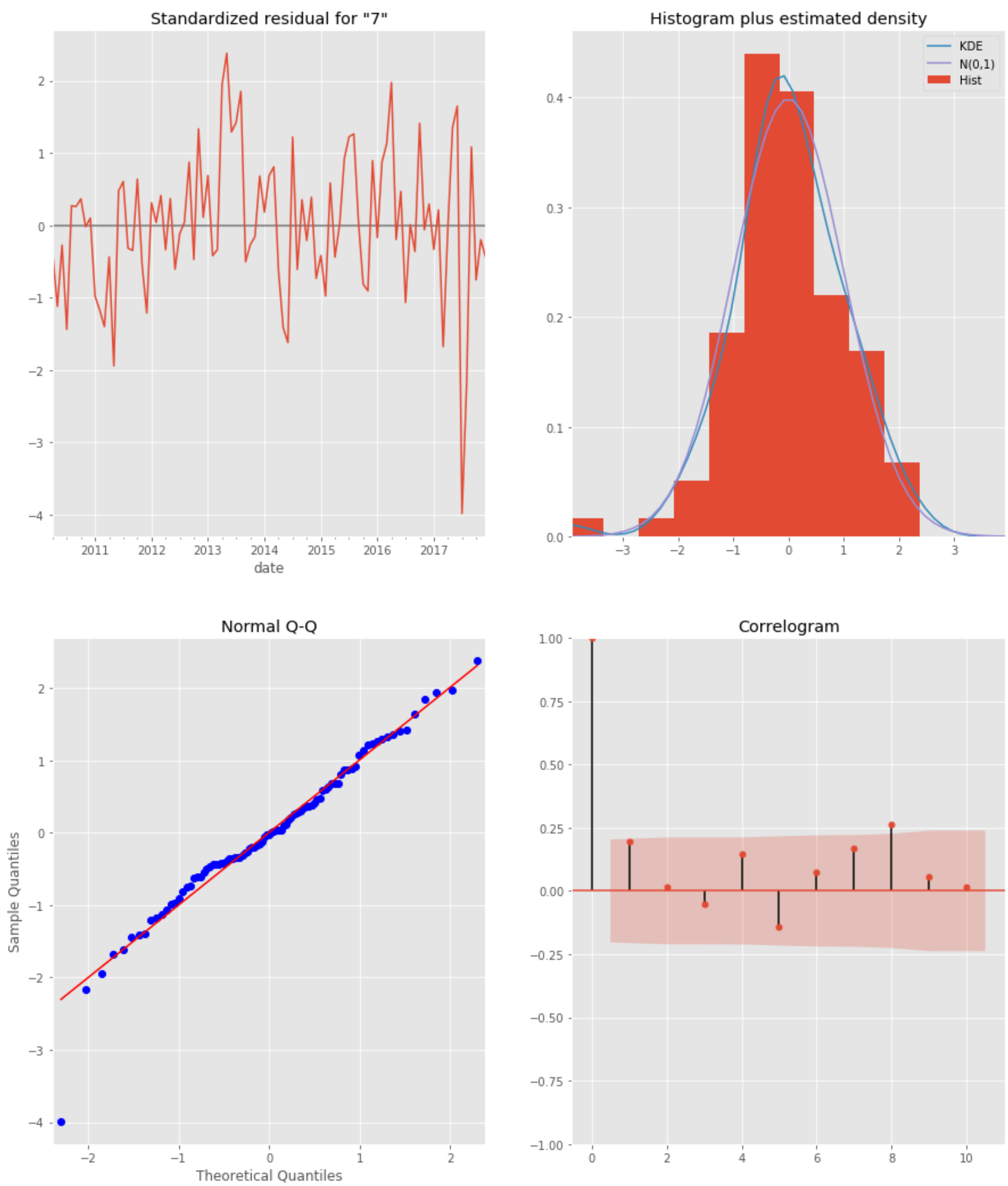
In [271]:
```
def dynamic_predictions(output, ts, time = '2016-01-01', plot = None):
    '''
    This function attempts to predict future years data. The default is 2 years (
    Inputs: Output from model
            Time-series
            time (date at which the model begins to make predictions - 2016 impli
    Output: Forecast
            Scoring metrics
    '''

    pred = output.get_prediction(start=pd.to_datetime(time), dynamic=True, full_r
    pred_conf = pred.conf_int()

    if plot == 'yes':

        rcParams['figure.figsize'] = 15, 6



        # Plot observed values
        ax = ts['2010':].plot(label='observed')

        # Plot predicted values
        pred.predicted_mean.plot(ax=ax, label='Dynamic Forecast', alpha=0.9)

        # Plot the range for confidence intervals
        ax.fill_between(pred_conf.index,
                        pred_conf.iloc[:, 0],
                        pred_conf.iloc[:, 1], color='g', alpha=0.5)

        # Set axes labels
        ax.set_xlabel('Date')
        ax.set_ylabel('Avg. Median Sale Price')
        plt.legend()

        plt.show()


    #Get the real and predicted values
    austin_forecasted = pred.predicted_mean
    austin_truth = ts['2016-01-01':]

    return pred, austin_forecasted, austin_truth
```

In [272]:
```
pdq_test(austin_78746_2010_log)
```

```
ARIMA (2, 2, 0) x (2, 1, 2, 12)12 : AIC Calculated =1795.629673404822
ARIMA (2, 2, 0) x (2, 2, 0, 12)12 : AIC Calculated =-308.83098844364804
ARIMA (2, 2, 0) x (2, 2, 1, 12)12 : AIC Calculated =-304.52493055080174
ARIMA (2, 2, 0) x (2, 2, 2, 12)12 : AIC Calculated =-278.8183449506648
ARIMA (2, 2, 1) x (0, 0, 0, 12)12 : AIC Calculated =-781.0014430362819
ARIMA (2, 2, 1) x (0, 0, 1, 12)12 : AIC Calculated =-672.2594125953299
ARIMA (2, 2, 1) x (0, 0, 2, 12)12 : AIC Calculated =2372.4474051797406
ARIMA (2, 2, 1) x (0, 1, 0, 12)12 : AIC Calculated =-614.5433334067725
ARIMA (2, 2, 1) x (0, 1, 1, 12)12 : AIC Calculated =-535.5581846636784
ARIMA (2, 2, 1) x (0, 1, 2, 12)12 : AIC Calculated =2701.3331946198136
ARIMA (2, 2, 1) x (0, 2, 0, 12)12 : AIC Calculated =-452.7871035572231
ARIMA (2, 2, 1) x (0, 2, 1, 12)12 : AIC Calculated =-383.27038851075304
ARIMA (2, 2, 1) x (0, 2, 2, 12)12 : AIC Calculated =-285.2179622931178
ARIMA (2, 2, 1) x (1, 0, 0, 12)12 : AIC Calculated =-674.2621464213549
ARIMA (2, 2, 1) x (1, 0, 1, 12)12 : AIC Calculated =-670.9027903858969
ARIMA (2, 2, 1) x (1, 0, 2, 12)12 : AIC Calculated =2375.4001357029583
ARIMA (2, 2, 1) x (1, 1, 0, 12)12 : AIC Calculated =-535.0434923791036
ARIMA (2, 2, 1) x (1, 1, 1, 12)12 : AIC Calculated =-530.4273260612075
ARIMA (2, 2, 1) x (1, 1, 2, 12)12 : AIC Calculated =2698.9313371059716
```

In [146]:
```
output = arima_fit(austin_78746_2010_log, order = (1,0,2), seasonal_order = (0,0,
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
drift       6.954e-05   3.59e-05      1.936      0.053   -8.56e-07       0.000
ar.L1          1.0000      0.000   5981.690      0.000       1.000       1.000
ma.L1          1.3101      0.081     16.195      0.000       1.152       1.469
ma.L2          0.6677      0.097      6.894      0.000       0.478       0.857
sigma2      8.012e-06   1.02e-06      7.866      0.000    6.02e-06       1e-05
==============================================================================
```



In [224]:
```
def two_year_future_forecast(ts, output, log = None, plot = None):
    # Get forecast 24 steps ahead in future
    prediction_future = output.get_forecast(steps=24)

    # Get confidence intervals of forecasts
    pred_conf = prediction_future.conf_int()

    if plot == 'yes':

        # Plot future predictions with confidence intervals
        ax = ts['2010':].plot(label='observed', figsize=(20, 15))
        prediction_future.predicted_mean.plot(ax=ax, label='Forecast')
        ax.fill_between(pred_conf.index,
                        pred_conf.iloc[:, 0],
                        pred_conf.iloc[:, 1], color='k', alpha=0.25)
        ax.set_xlabel('Date')
        ax.set_ylabel('Avg. Median Sale Price')

        plt.legend()
        plt.show()

    returns_df = pd.DataFrame(columns = ['Zipcode', 'two_year_returns_(%)'])

    if log == 'yes':
        pred_future_unlogged = np.exp(prediction_future.predicted_mean)
        predictions_year_groups = pred_future_unlogged.groupby(pd.Grouper(freq =

        observed_year_groups = np.exp(ts).groupby(pd.Grouper(freq = 'A'))
        two_year_return = ((predictions_year_groups.mean()['2019'].values - obser

        ts_df = pd.DataFrame(ts)
        for col in ts_df.columns:
            new_row = {'Zipcode':col, 'two_year_returns_(%)': two_year_return}
            returns_df.loc[len(returns_df.index)] = new_row
            returns_df['two_year_returns_(%)'] = round(returns_df['two_year_retur

        #return returns_df

    else:
        pred_future_unlogged = prediction_future.predicted_mean
        predictions_year_groups = pred_future_unlogged.groupby(pd.Grouper(freq =

        observed_year_groups = ts.groupby(pd.Grouper(freq = 'A'))
        two_year_return = ((predictions_year_groups.mean()['2019'].values - obser


        ts_df = pd.DataFrame(ts)
        for col in ts_df.columns:
            new_row = {'Zipcode':col, 'two_year_returns_(%)': two_year_return}
            returns_df.loc[len(returns_df.index)] = new_row

        #return returns_df
    return prediction_future
```

# Modeling across all of the Austin Zipcodes

In [274]:
```
#creating an empty dataframe to store all of our zipcodes and associated returns
returns_df = pd.DataFrame(columns = ['Zipcode', 'two_year_returns_(%)', 'r2', 'ex

for col in austin_zip.columns:
    #create a logged series for each austin zip code from 2010 - 2017 (only full
    ts_log_2010 = np.log(austin_zip[col]['2010':'2017'])
    ts_log_2010 = ts_log_2010.resample('MS').mean()

    #fitting the model
    output = arima_fit(ts_log_2010, order = (1,0,2), seasonal_order = (0,0,0,12)

    #using the dynamic predictions function to produce predicted and forecasted r
    pred, austin_forecasted, austin_truth = dynamic_predictions(output, ts_log_20

    #using the two_year_future_forecast function to find the two year future pred
    pred_future = two_year_future_forecast(ts_log_2010, output, log = 'yes')

    #unlogging predictions to see them in real monetary value, grouping by year t
    pred_future_unlogged = np.exp(pred_future.predicted_mean)
    predictions_year_groups = pred_future_unlogged.groupby(pd.Grouper(freq = 'A')

    #same as above but with observed values
    observed_year_groups = np.exp(ts_log_2010).groupby(pd.Grouper(freq = 'A'))

    #calculating % return from observed 2017 price to 2019 price
    two_year_return = ((predictions_year_groups.mean()['2019'].values - observed_

    #reporting metrics
    explained_var, mae, rmse, r2 = report_metrics(austin_truth, austin_forecasted

    #fill in zipcode and return data into dataframe
    ts_df = pd.DataFrame(ts_log_2010)
    for col in ts_df.columns:
        new_row = {'Zipcode':col, 'two_year_returns_(%)': two_year_return, 'r2':r
        returns_df.loc[len(returns_df.index)] = new_row
        returns_df['two_year_returns_(%)'] = round(returns_df['two_year_returns_(

returns_df.head()
```

Out[274]:

| | Zipcode | two_year_returns_(%) | r2 | explained_var | rmse |
|---|---|---|---|---|---|
| **0** | 78617 | 24 | 0.659254 | 0.867296 | 1.035961 |
| **1** | 78702 | 28 | 0.149472 | 0.370720 | 1.036816 |
| **2** | 78703 | 16 | 0.868692 | 0.955030 | 1.017449 |
| **3** | 78704 | 16 | 0.960397 | 0.978925 | 1.008609 |
| **4** | 78705 | 3 | -0.802685 | -0.675804 | 1.035823 |

In [275]:
```
#dropping zipcodes where the model produced a negative R^2 value
zip_list = ['78705', '78730', '78731', '78735', '78751', '78752', '78759']
returns_df.drop(returns_df.loc[returns_df['Zipcode'].isin(zip_list)].index, inpla
```

In [278]:
```
returns_df.sort_values(by = 'two_year_returns_(%)', ascending = False).head(10)
```

Out[278]:

| | Zipcode | two_year_returns_(%) | r2 | explained_var | rmse |
|---|---|---|---|---|---|
| **23** | 78744 | 41 | 0.950731 | 0.965794 | 1.016527 |
| **6** | 78721 | 40 | 0.574075 | 0.905937 | 1.055582 |
| **32** | 78753 | 35 | 0.710833 | 0.919867 | 1.042395 |
| **22** | 78741 | 32 | 0.510580 | 0.620442 | 1.041058 |
| **9** | 78724 | 32 | 0.249127 | 0.770025 | 1.063418 |
| **36** | 78758 | 31 | 0.335847 | 0.767313 | 1.068786 |
| **8** | 78723 | 30 | 0.812586 | 0.813228 | 1.021536 |
| **10** | 78725 | 29 | 0.547158 | 0.599704 | 1.031258 |
| **1** | 78702 | 28 | 0.149472 | 0.370720 | 1.036816 |
| **12** | 78727 | 24 | 0.682508 | 0.879033 | 1.041453 |

In [ ]:

In [ ]:

# Conclusions and Further Analysis

In [ ]: