In [36]:
```
import pandas as pd
import os
%pylab inline
import matplotlib.pylab as plt
%matplotlib inline
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 15, 6
#if os.path.exists("Wallmart_Database.db") : os.remove("Wallmart_Databas
e.db")
import sqlite3
from datetime import datetime
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.tsa.arima_model import ARIMA
```

```
Populating the interactive namespace from numpy and matplotlib

/Users/azizmamatov/anaconda/lib/python2.7/site-packages/IPython/core/ma
gics/pylab.py:161: UserWarning: pylab import has clobbered these variab
les: ['plt', 'axes', 'datetime']
`%matplotlib` prevents importing * from pylab and numpy
  "\n`%matplotlib` prevents importing * from pylab and numpy"
```

In [10]:
```
!pwd
#%cd
%cd azizmamatov/
```

```
/Users
/Users/azizmamatov
```

Creating SQLite database out of csv files

Should be done once. We can then create pd series out of the database by ####pd.read_sql_query('Query;',conn)

In [3]:
```
'''
df_stores = pd.read_csv('Downloads/Walmart_Data/stores.csv')
df_features = pd.read_csv('Downloads/Walmart_Data/features.csv')
df_train = pd.read_csv('Downloads/Walmart_Data/train.csv')

conn = sqlite3.connect('Downloads/Wallmart_Database.db')
df_stores.to_sql('Stores_Table',conn)
df_features.to_sql('Features_Table',conn)
df_train.to_sql('Train_Table',conn)
sql_string = 'Select * from Stores_Table'
df_x = pd.read_sql('Select * from Stores_Table', conn)
df_x.head(5)
'''
```

Out[3]:
```
"\ndf_stores = pd.read_csv('Downloads/Walmart_Data/stores.csv')\ndf_fea
tures = pd.read_csv('Downloads/Walmart_Data/features.csv')\ndf_train =
 pd.read_csv('Downloads/Walmart_Data/train.csv')
  \nconn = sqlite3.connect('Downloads/Wallmart_Database.db')\ndf_store
s.to_sql('Stores_Table',conn)\ndf_features.to_sql('Features_Table',con
n)\ndf_train.to_sql('Train_Table',conn)                            \nsql_
string = 'Select * from Stores_Table'\ndf_x = pd.read_sql('Select * fro
m Stores_Table', conn)\ndf_x.head(5)\n"
```

SQL queries and creation of df

In [12]:
```
conn = sqlite3.connect('Downloads/Wallmart_Database.db')
cur = conn.cursor()
sql_string = 'Select * from Stores_Table Join Features_Table Using(Stor
e);'
df_y = pd.read_sql(sql_string, conn)
df_y.describe()
df_y[df_y["Size"]==df_y["Size"].max()]
```

```
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/numpy/lib/funct
ion_base.py:3834: RuntimeWarning: Invalid value encountered in percenti
le
  RuntimeWarning)
```

Out[12]:

	index	Store	Type	Size	index	Date	Temperature	Fuel_Price	MarkDown1	Ma
2184	12	13	A	219622	2184	2010-02-05	31.53	2.666	NaN	Nal
2185	12	13	A	219622	2185	2010-02-12	33.16	2.671	NaN	Nal
2186	12	13	A	219622	2186	2010-02-19	35.70	2.654	NaN	Nal
2187	12	13	A	219622	2187	2010-02-26	29.98	2.667	NaN	Nal
2188	12	13	A	219622	2188	2010-03-05	40.65	2.681	NaN	Nal
2189	12	13	A	219622	2189	2010-03-12	37.62	2.733	NaN	Nal
2190	12	13	A	219622	2190	2010-03-19	42.49	2.782	NaN	Nal
2191	12	13	A	219622	2191	2010-03-26	41.48	2.819	NaN	Nal
2192	12	13	A	219622	2192	2010-04-02	42.15	2.842	NaN	Nal
2193	12	13	A	219622	2193	2010-04-09	38.97	2.877	NaN	Nal
2194	12	13	A	219622	2194	2010-04-16	50.39	2.915	NaN	Nal
2195	12	13	A	219622	2195	2010-04-23	55.66	2.936	NaN	Nal
2196	12	13	A	219622	2196	2010-04-30	48.33	2.941	NaN	Nal
2197	12	13	A	219622	2197	2010-05-07	44.42	2.948	NaN	Nal
2198	12	13	A	219622	2198	2010-05-14	50.15	2.962	NaN	Nal
2199	12	13	A	219622	2199	2010-05-21	57.71	2.950	NaN	Nal
2200	12	13	A	219622	2200	2010-05-28	53.11	2.908	NaN	Nal
2201	12	13	A	219622	2201	2010-06-04	59.85	2.871	NaN	Nal
2202	12	13	A	219622	2202	2010-06-11	65.24	2.841	NaN	Nal

	index	Store	Type	Size	index	Date	Temperature	Fuel_Price	MarkDown1	Ma
2203	12	13	A	219622	2203	2010-06-18	58.41	2.819	NaN	Na
2204	12	13	A	219622	2204	2010-06-25	71.83	2.820	NaN	Na
2205	12	13	A	219622	2205	2010-07-02	78.82	2.814	NaN	Na
2206	12	13	A	219622	2206	2010-07-09	71.33	2.802	NaN	Na
2207	12	13	A	219622	2207	2010-07-16	77.79	2.791	NaN	Na
2208	12	13	A	219622	2208	2010-07-23	82.27	2.797	NaN	Na
2209	12	13	A	219622	2209	2010-07-30	78.94	2.797	NaN	Na
2210	12	13	A	219622	2210	2010-08-06	81.24	2.802	NaN	Na
2211	12	13	A	219622	2211	2010-08-13	74.93	2.837	NaN	Na
2212	12	13	A	219622	2212	2010-08-20	76.34	2.850	NaN	Na
2213	12	13	A	219622	2213	2010-08-27	75.31	2.854	NaN	Na
...	...	...	...	...	...	...	...	...	...	...
2336	12	13	A	219622	2336	2013-01-04	13.43	3.066	4914.57	39
2337	12	13	A	219622	2337	2013-01-11	20.00	2.982	3726.82	17
2338	12	13	A	219622	2338	2013-01-18	11.44	2.914	6847.96	52
2339	12	13	A	219622	2339	2013-01-25	14.75	2.927	3250.58	25
2340	12	13	A	219622	2340	2013-02-01	30.44	3.029	21473.20	19
2341	12	13	A	219622	2341	2013-02-08	26.11	3.192	103184.98	10
2342	12	13	A	219622	2342	2013-02-15	27.12	3.323	17771.48	69

	index	Store	Type	Size	index	Date	Temperature	Fuel_Price	MarkDown1	Ma
2343	12	13	A	219622	2343	2013-02-22	29.97	3.459	10360.46	874
2344	12	13	A	219622	2344	2013-03-01	26.97	3.521	8315.84	126
2345	12	13	A	219622	2345	2013-03-08	37.63	3.526	29775.13	279
2346	12	13	A	219622	2346	2013-03-15	46.72	3.518	7748.56	NaN
2347	12	13	A	219622	2347	2013-03-22	42.94	3.518	15752.69	NaN
2348	12	13	A	219622	2348	2013-03-29	41.71	3.518	8423.57	NaN
2349	12	13	A	219622	2349	2013-04-05	53.84	3.547	25061.60	165
2350	12	13	A	219622	2350	2013-04-12	45.29	3.576	5553.66	657
2351	12	13	A	219622	2351	2013-04-19	41.07	3.559	2604.11	186
2352	12	13	A	219622	2352	2013-04-26	48.17	3.541	3664.88	NaN
2353	12	13	A	219622	2353	2013-05-03	54.51	3.535	15599.02	9.0
2354	12	13	A	219622	2354	2013-05-10	59.55	3.543	4974.55	349
2355	12	13	A	219622	2355	2013-05-17	70.01	3.609	10484.85	332
2356	12	13	A	219622	2356	2013-05-24	57.25	3.720	4055.58	174
2357	12	13	A	219622	2357	2013-05-31	60.51	3.773	5026.09	342
2358	12	13	A	219622	2358	2013-06-07	67.49	3.779	15752.93	113
2359	12	13	A	219622	2359	2013-06-14	76.41	3.771	5574.52	798
2360	12	13	A	219622	2360	2013-06-21	70.49	3.740	5531.43	450
2361	12	13	A	219622	2361	2013-06-28	75.24	3.726	7171.47	928

	index	Store	Type	Size	index	Date	Temperature	Fuel_Price	MarkDown1	Ma
2362	12	13	A	219622	2362	2013-07-05	85.58	3.696	22841.84	322
2363	12	13	A	219622	2363	2013-07-12	78.93	3.666	7062.38	156
2364	12	13	A	219622	2364	2013-07-19	80.81	3.665	2973.47	144
2365	12	13	A	219622	2365	2013-07-26	83.62	3.669	346.31	137

182 rows × 16 columns

In [ ]:
```
conn = sqlite3.connect('Downloads/Wallmart_Database.db')
cur = conn.cursor()
cur.execute("SELECT * FROM Stores_Table order by(Store) desc limit
5;").fetchall() #query on getting store information
cur.execute("SELECT count(Store) FROM Train_Table;").fetchall() #number
 of records from Train_table
#ordered by size and limited
```

Number of records from joint Stores_Table and Features_Table

In [ ]:
```
pd.read_sql_query('select count(*) from (Select * from Stores_Table Join
 Features_Table Using(Store));',conn)
```

Pulling df out of SQL DB

dept is not correct for some reason

In [13]:
```
#df = pd.read_sql_query('Select * from Stores_Table Join Features_Table
 Using(Store);',conn)
df = pd.read_sql_query('Select * from Train_Table;',conn)
print df.dtypes, df.describe()
```

```
index             int64
Store             int64
Dept              int64
Date             object
Weekly_Sales    float64
IsHoliday         int64
dtype: object                 index          Store           Dept   Week
ly_Sales  \
count  421570.000000  421570.000000  421570.000000  421570.000000
mean   210784.500000      22.200546      44.260317   15981.258123
std    121696.920828      12.785297      30.492054   22711.183519
min         0.000000       1.000000       1.000000   -4988.940000
25%    105392.250000      11.000000      18.000000    2079.650000
50%    210784.500000      22.000000      37.000000    7612.030000
75%    316176.750000      33.000000      74.000000   20205.852500
max    421569.000000      45.000000      99.000000  693099.360000

           IsHoliday
count  421570.000000
mean        0.070358
std         0.255750
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000
```

Creating df out of CSV file

Converting 'Date' column to date type for time series purposes

In [14]:
```
df_train = pd.read_csv('Downloads/Walmart_Data/train.csv')

df.tail(5)
df['Date'] = pd.to_datetime(df['Date'])
print df.describe(), df.dtypes
df.tail(5)
```

```
               index          Store           Dept   Weekly_Sales  \
count  421570.000000  421570.000000  421570.000000  421570.000000
mean   210784.500000      22.200546      44.260317   15981.258123
std    121696.920828      12.785297      30.492054   22711.183519
min         0.000000       1.000000       1.000000   -4988.940000
25%    105392.250000      11.000000      18.000000    2079.650000
50%    210784.500000      22.000000      37.000000    7612.030000
75%    316176.750000      33.000000      74.000000   20205.852500
max    421569.000000      45.000000      99.000000  693099.360000

           IsHoliday
count  421570.000000
mean        0.070358
std         0.255750
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000   index                    int64
Store                     int64
Dept                      int64
Date             datetime64[ns]
Weekly_Sales            float64
IsHoliday                 int64
dtype: object
```

Out[14]:

	index	Store	Dept	Date	Weekly_Sales	IsHoliday
421565	421565	45	98	2012-09-28	508.37	0
421566	421566	45	98	2012-10-05	628.10	0
421567	421567	45	98	2012-10-12	1061.02	0
421568	421568	45	98	2012-10-19	760.01	0
421569	421569	45	98	2012-10-26	1076.80	0

In [15]:
```
#df[df['Store']==df['Weekly_Sales'].idxmax()]
df.ix[df['Weekly_Sales'].idxmax()]
```

Out[15]:
```
index                         95373
Store                            10
Dept                             72
Date            2010-11-26 00:00:00
Weekly_Sales                 693099
IsHoliday                         1
Name: 95373, dtype: object
```

Adding Shop and Type data to Train table

Can't join tables fully as it becomes too large (3 bln rows)

In [16]:
```
conn = sqlite3.connect('Downloads/Wallmart_Database.db')
cur = conn.cursor()
df_type = pd.read_sql_query('Select * from Train_Table Join (select (Typ
e), (Store) from Stores_Table) Using(Store);',conn)
```

In [17]:
```
df_type['Date'] = pd.to_datetime(df_type['Date']) # for time series we n
eed to convert the object type to date type
print df_type.describe(), df_type.head(5), df_type.dtypes
```

```
                index          Store           Dept   Weekly_Sales  \
count  421570.000000  421570.000000  421570.000000  421570.000000
mean   210784.500000      22.200546      44.260317   15981.258123
std    121696.920828      12.785297      30.492054   22711.183519
min         0.000000       1.000000       1.000000   -4988.940000
25%    105392.250000      11.000000      18.000000    2079.650000
50%    210784.500000      22.000000      37.000000    7612.030000
75%    316176.750000      33.000000      74.000000   20205.852500
max    421569.000000      45.000000      99.000000  693099.360000

           IsHoliday
count  421570.000000
mean        0.070358
std         0.255750
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000      index  Store  Dept       Date  Weekly_Sales
 IsHoliday Type
0      0      1     1 2010-02-05      24924.50          0    A
1      1      1     1 2010-02-12      46039.49          1    A
2      2      1     1 2010-02-19      41595.55          0    A
3      3      1     1 2010-02-26      19403.54          0    A
4      4      1     1 2010-03-05      21827.90          0    A index
                 int64
Store                   int64
Dept                    int64
Date           datetime64[ns]
Weekly_Sales          float64
IsHoliday               int64
Type                   object
dtype: object
```

Setting date as index for time series purposes

Alternatively, it could be directly made when csv file was uploaded to pd. data = pd.read_csv('File.csv', parse_dates='Month', index_col='Month',date_parser=dateparse)

In [18]:
```
#df_type.set_index('Date', inplace=True)
df_type2 = df_type.set_index(pd.DatetimeIndex(df_type['Date']))

df_type2.index
```

Out[18]:
```
DatetimeIndex(['2010-02-05', '2010-02-12', '2010-02-19', '2010-02-26',
               '2010-03-05', '2010-03-12', '2010-03-19', '2010-03-26',
               '2010-04-02', '2010-04-09',
               ...
               '2012-08-24', '2012-08-31', '2012-09-07', '2012-09-14',
               '2012-09-21', '2012-09-28', '2012-10-05', '2012-10-12',
               '2012-10-19', '2012-10-26'],
              dtype='datetime64[ns]', length=421570, freq=None)
```

In [19]:
```
#df_type2['2010'][df_type2['Store']==1] not working takes too much time
df_type2['2010'].mean() #means accross the df belonging to 2010 year
tf= df_type2[['Store', 'Weekly_Sales']] #creating timeseries df with sto
re and
tf.head(5)
```

Out[19]:

	Store	Weekly_Sales
2010-02-05	1	24924.50
2010-02-12	1	46039.49
2010-02-19	1	41595.55
2010-02-26	1	19403.54
2010-03-05	1	21827.90

Determining stationary series

Apparently according to https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/ (https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/) it is important to determine if the series are stationary (mean, variance reamin constant over time). Most TS models work on stationary models.

In [ ]:
```
plt.plot(tf) #too much data, need to see one shop only
```

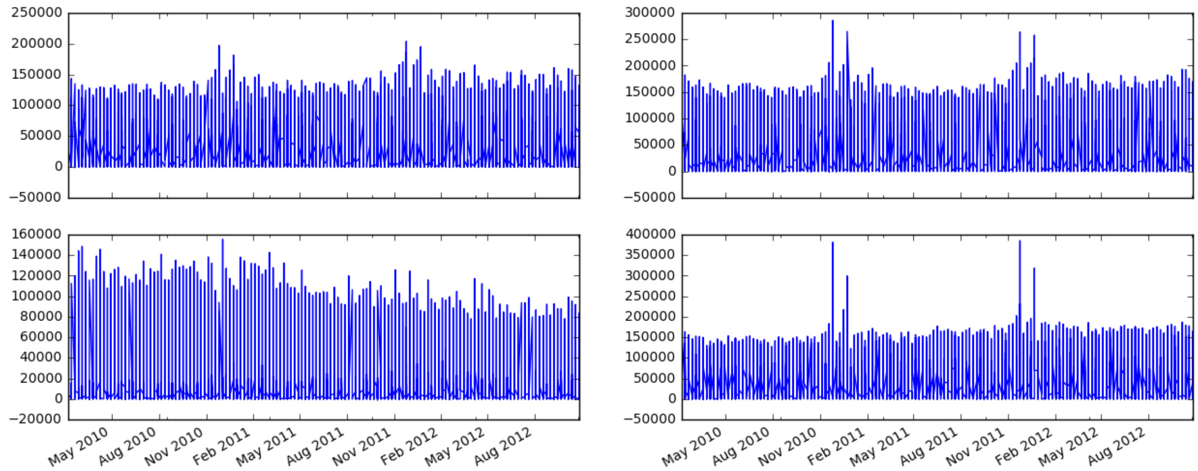
In [ ]:
```
tf_store1 = tf['Weekly_Sales'][tf['Store']==1] #only sales data for stor
e No.1 but for many departments,
#so they all should be grouped
plt.plot(tf_store1) #only for store no 1, showing too many lines for som
e reason
```

In [ ]:
```
tf_store1.describe()
tf_store1.plot.line(x=None, y=None)
```

In [ ]:
```
tf_store1.head(5)
```

In [20]:
```
#plotting several plots to see if
fig, axes = plt.subplots(nrows=2, ncols=2, sharex = True)
tf['Weekly_Sales'][tf['Store']==1].plot.line(ax=axes[0,0])
tf['Weekly_Sales'][tf['Store']==2].plot.line(ax=axes[0,1])
tf['Weekly_Sales'][tf['Store']==3].plot.line(ax=axes[1,0])
tf['Weekly_Sales'][tf['Store']==4].plot.line(ax=axes[1,1])
```

Out[20]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x115665b10>
```



Ducker - Fuller test for stationarity

https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/ (https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/)

In [21]:
```
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):

    #Determing rolling statistics
    rolmean = pd.rolling_mean(timeseries, window=12)
    rolstd = pd.rolling_std(timeseries, window=12)

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

    #Perform Dickey-Fuller test:
    print 'Results of Dickey-Fuller Test:'
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-
value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print dfoutput
```

In [ ]:
```
test_stationarity(tf_store1)
# this didn't work as even thogh the store is only one but it has 99 dep
artment data which should be summarized.
```

Uploading new table to Database to have a better timeseries data

It is a good idea to have a time series based data in the database. We should also produce a df with sales grouped by stores to avoid all these departments

In [22]:
```
df_type2.head(5)
```

Out[22]:

	index	Store	Dept	Date	Weekly_Sales	IsHoliday	Type
2010-02-05	0	1	1	2010-02-05	24924.50	0	A
2010-02-12	1	1	1	2010-02-12	46039.49	1	A
2010-02-19	2	1	1	2010-02-19	41595.55	0	A
2010-02-26	3	1	1	2010-02-26	19403.54	0	A
2010-03-05	4	1	1	2010-03-05	21827.90	0	A

In [ ]:
```
# it can only be executed once, as the Table already exists
#df_type2.to_sql('TS_Train_Table',conn)
```

In [23]:
```
df_salesstores = pd.read_sql_query('Select (Store), (Date), sum(Weekly_S
ales) from TS_Train_Table group by (Date),(Store);',conn)
```

In [24]:
```
print df_salesstores.head(5), df_salesstores.dtypes, df_salesstores.desc
ribe
```

```
   Store                 Date  sum(Weekly_Sales)
0      1  2010-02-05 00:00:00         1643690.90
1      2  2010-02-05 00:00:00         2136989.46
2      3  2010-02-05 00:00:00          461622.22
3      4  2010-02-05 00:00:00         2135143.87
4      5  2010-02-05 00:00:00          317173.10 Store
int64
Date                  object
sum(Weekly_Sales)    float64
dtype: object <bound method DataFrame.describe of       Store
       Date  sum(Weekly_Sales)
0         1  2010-02-05 00:00:00         1643690.90
1         2  2010-02-05 00:00:00         2136989.46
2         3  2010-02-05 00:00:00          461622.22
3         4  2010-02-05 00:00:00         2135143.87
4         5  2010-02-05 00:00:00          317173.10
5         6  2010-02-05 00:00:00         1652635.10
6         7  2010-02-05 00:00:00          496725.44
7         8  2010-02-05 00:00:00         1004137.09
8         9  2010-02-05 00:00:00          549505.55
9        10  2010-02-05 00:00:00         2193048.75
10       11  2010-02-05 00:00:00         1528008.64
11       12  2010-02-05 00:00:00         1100046.37
12       13  2010-02-05 00:00:00         1967220.53
13       14  2010-02-05 00:00:00         2623469.95
14       15  2010-02-05 00:00:00          652122.44
15       16  2010-02-05 00:00:00          477409.30
16       17  2010-02-05 00:00:00          789036.02
17       18  2010-02-05 00:00:00         1205307.50
18       19  2010-02-05 00:00:00         1507637.17
19       20  2010-02-05 00:00:00         2401395.47
20       21  2010-02-05 00:00:00          798593.88
21       22  2010-02-05 00:00:00         1033017.37
22       23  2010-02-05 00:00:00         1364721.58
23       24  2010-02-05 00:00:00         1388725.63
24       25  2010-02-05 00:00:00          677231.63
25       26  2010-02-05 00:00:00         1034119.21
26       27  2010-02-05 00:00:00         1874289.79
27       28  2010-02-05 00:00:00         1672352.29
28       29  2010-02-05 00:00:00          538634.46
29       30  2010-02-05 00:00:00          465108.52
...     ...                  ...                ...
6405     16  2012-10-26 00:00:00          475770.14
6406     17  2012-10-26 00:00:00          943465.29
6407     18  2012-10-26 00:00:00         1127516.25
6408     19  2012-10-26 00:00:00         1322117.96
6409     20  2012-10-26 00:00:00         2031650.55
6410     21  2012-10-26 00:00:00          675202.87
6411     22  2012-10-26 00:00:00         1094422.69
6412     23  2012-10-26 00:00:00         1347454.59
6413     24  2012-10-26 00:00:00         1307182.29
6414     25  2012-10-26 00:00:00          688940.94
6415     26  2012-10-26 00:00:00          958619.80
6416     27  2012-10-26 00:00:00         1703047.74
6417     28  2012-10-26 00:00:00         1213860.61
6418     29  2012-10-26 00:00:00          534970.68
6419     30  2012-10-26 00:00:00          439424.50
```

```
6420     31  2012-10-26 00:00:00         1340232.55
6421     32  2012-10-26 00:00:00         1219979.29
6422     33  2012-10-26 00:00:00          253731.13
6423     34  2012-10-26 00:00:00          956987.81
6424     35  2012-10-26 00:00:00          865137.60
6425     36  2012-10-26 00:00:00          272489.41
6426     37  2012-10-26 00:00:00          534738.43
6427     38  2012-10-26 00:00:00          417290.38
6428     39  2012-10-26 00:00:00         1569502.00
6429     40  2012-10-26 00:00:00          921264.52
6430     41  2012-10-26 00:00:00         1316542.59
6431     42  2012-10-26 00:00:00          514756.08
6432     43  2012-10-26 00:00:00          587603.55
6433     44  2012-10-26 00:00:00          361067.07
6434     45  2012-10-26 00:00:00          760281.43

[6435 rows x 3 columns]>
```

In [25]:

```
#transforming data type from int64 to datetime
df_salesstores['Date'] = pd.to_datetime(df_salesstores['Date'])
```

In [26]:

```
#setting up date as index for time series purposes
df_salesstore =
df_salesstores.set_index(pd.DatetimeIndex(df_salesstores['Date']))
print df_salesstore.index, df_salesstore.head(5)
```

```
DatetimeIndex(['2010-02-05', '2010-02-05', '2010-02-05', '2010-02-05',
               '2010-02-05', '2010-02-05', '2010-02-05', '2010-02-05',
               '2010-02-05', '2010-02-05',
               ...
               '2012-10-26', '2012-10-26', '2012-10-26', '2012-10-26',
               '2012-10-26', '2012-10-26', '2012-10-26', '2012-10-26',
               '2012-10-26', '2012-10-26'],
              dtype='datetime64[ns]', length=6435, freq=None)
   Store       Date  sum(Weekly_Sales)
2010-02-05      1 2010-02-05        1643690.90
2010-02-05      2 2010-02-05        2136989.46
2010-02-05      3 2010-02-05         461622.22
2010-02-05      4 2010-02-05        2135143.87
2010-02-05      5 2010-02-05         317173.10
```

In [27]:

```
# preparing timeseries df for Ducker - Fuller test
ts_store = df_salesstore['sum(Weekly_Sales)'][df_salesstore['Store']==1]
ts_store.head(5)
```

Out[27]:

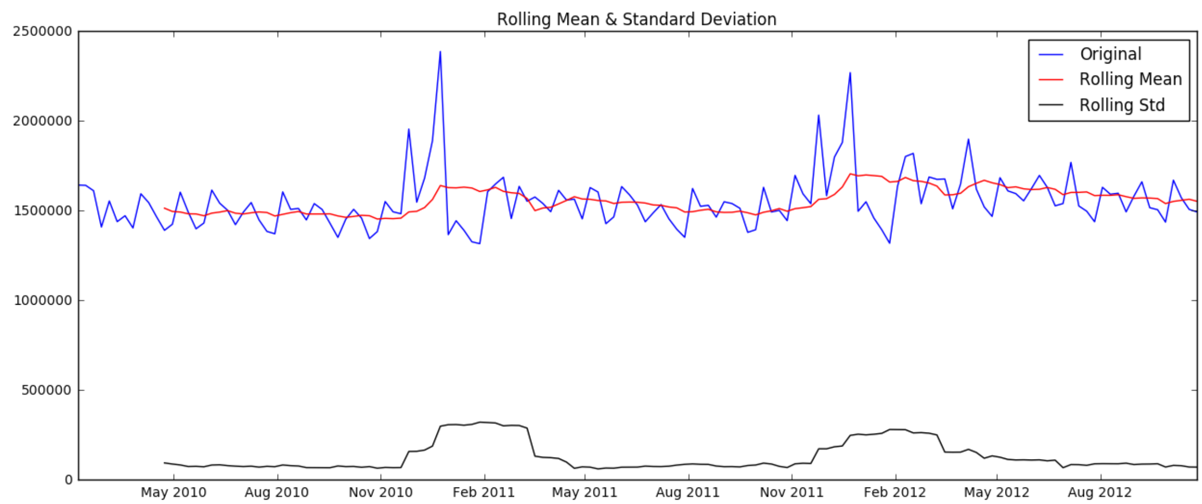
```
2010-02-05    1643690.90
2010-02-12    1641957.44
2010-02-19    1611968.17
2010-02-26    1409727.59
2010-03-05    1554806.68
Name: sum(Weekly_Sales), dtype: float64
```

Ducker - Fuller test on processed data

It looks like the the time series are stationary and as such are subject to time series modelling. Null-theory in this analysis is that the data is not stationary, and as our test stat is around 5, it means that we can reject the null-theory. If time series were stationary, there are certain techniques to make it stationary - https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/ (https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/)

In [28]:
```
# function to test stationarity
test_stationarity(ts_store)
```

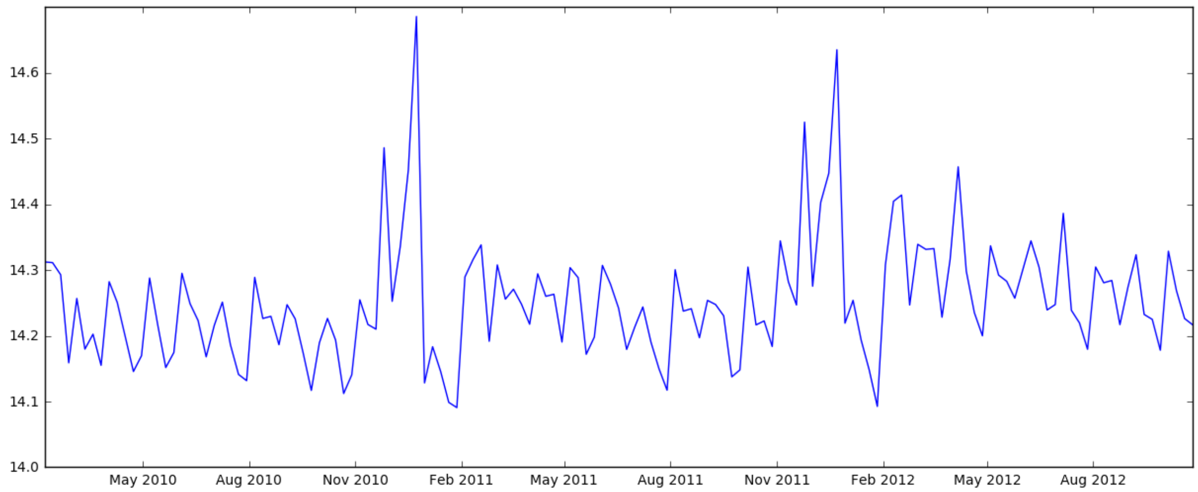
```
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/ipykernel/__mai
n__.py:5: FutureWarning: pd.rolling_mean is deprecated for Series and w
ill be removed in a future version, replace with
        Series.rolling(window=12,center=False).mean()
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/ipykernel/__mai
n__.py:6: FutureWarning: pd.rolling_std is deprecated for Series and wi
ll be removed in a future version, replace with
        Series.rolling(window=12,center=False).std()
```




```
Results of Dickey-Fuller Test:
Test Statistic                  -5.102186
p-value                          0.000014
#Lags Used                       4.000000
Number of Observations Used    138.000000
Critical Value (5%)             -2.882722
Critical Value (1%)             -3.478648
Critical Value (10%)            -2.578065
dtype: float64
```

In [29]:
```
#log transforming the data to reduce the trend and to use in future anal
ysis
ts_log = np.log(ts_store)
plt.plot(ts_log)
```

Out[29]: [<matplotlib.lines.Line2D at 0x10f702f10>]



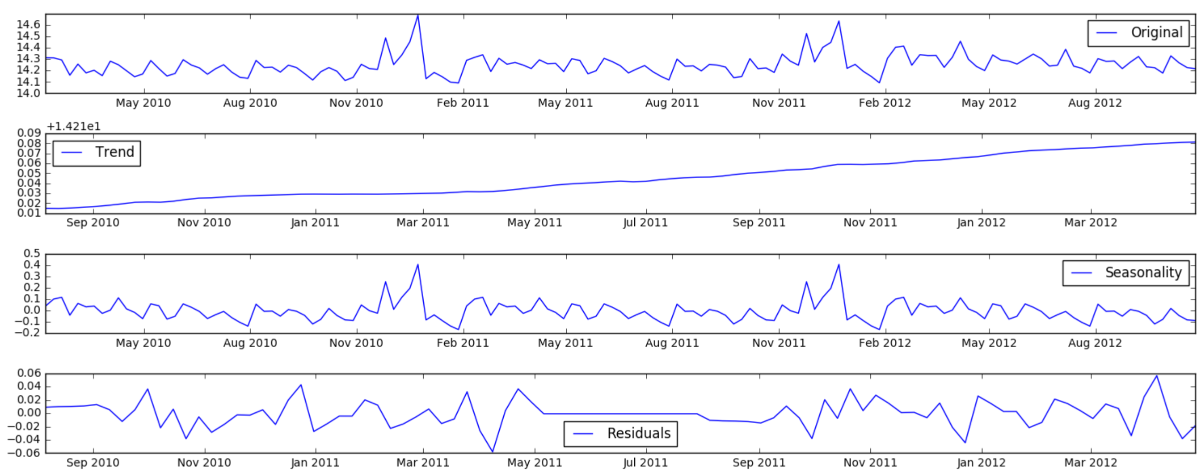
In [ ]:
```
# One of the most common methods of dealing with both trend and seasonal
ity is differencing. In this technique,
#we take the difference of the observation at a particular instant with
 that at the previous instant.
#This mostly works well in improving stationarity.
#First order differencing can be done in Pandas as:
ts_log_diff = ts_log - ts_log.shift()
plt.plot(ts_log_diff)
```

Decomposing

In this approach, both trend and seasonality are modeled separately and the remaining part of the series is returned. I'll skip the statistics and come to the results:

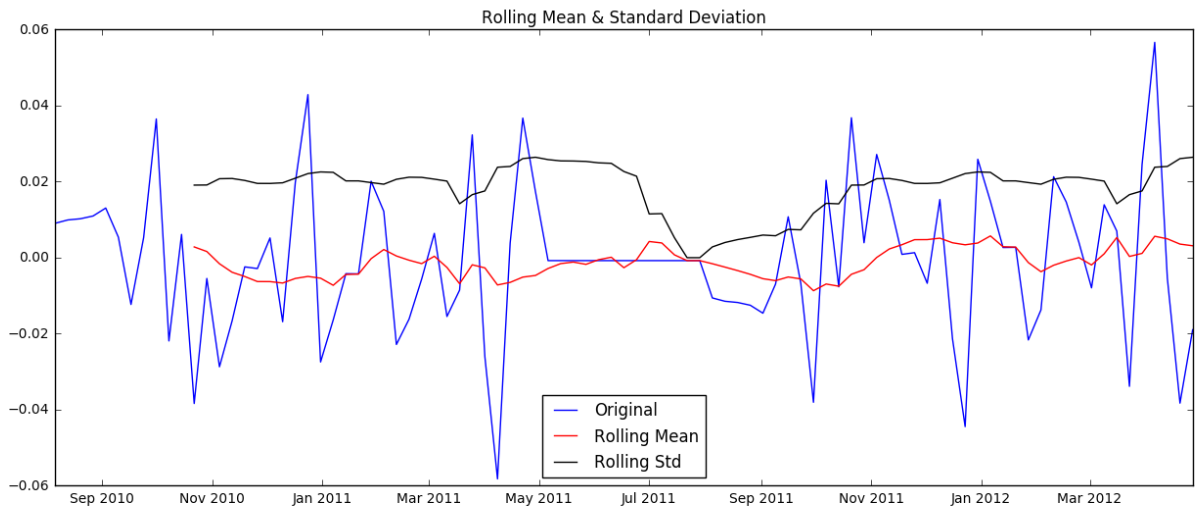
In [30]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(ts_log, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```

```
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/statsmodels/ts
a/filters/filtertools.py:28: VisibleDeprecationWarning: using a non-int
eger number instead of an integer will result in an error in the future
  return np.r_[[np.nan] * head, x, [np.nan] * tail]
```



In [31]:
```
# trend and seasonality above are separated frmo data and we can model t
he residuals. We will check the stationarity
#of residuals.
ts_log_decompose = residual
ts_log_decompose.dropna(inplace=True)
test_stationarity(ts_log_decompose)
```

```
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/ipykernel/__mai
n__.py:5: FutureWarning: pd.rolling_mean is deprecated for Series and w
ill be removed in a future version, replace with
        Series.rolling(window=12,center=False).mean()
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/ipykernel/__mai
n__.py:6: FutureWarning: pd.rolling_std is deprecated for Series and wi
ll be removed in a future version, replace with
        Series.rolling(window=12,center=False).std()
```




```
Results of Dickey-Fuller Test:
Test Statistic                 -8.408962e+00
p-value                         2.129417e-13
#Lags Used                      1.000000e+00
Number of Observations Used     8.900000e+01
Critical Value (5%)            -2.894607e+00
Critical Value (1%)            -3.506057e+00
Critical Value (10%)           -2.584410e+00
dtype: float64
```

The Dickey-Fuller test statistic is significantly lower than the 1% critical value. So this TS is very close to stationary. You can try advanced decomposition techniques as well which can generate better results. Also, you should note that converting the residuals into original values for future data in not very intuitive in this case.

In [34]:
```
ts_log_diff = ts_log - ts_log.shift()
```

There are different techniques to forecast and we need to be careful with them and estimate:

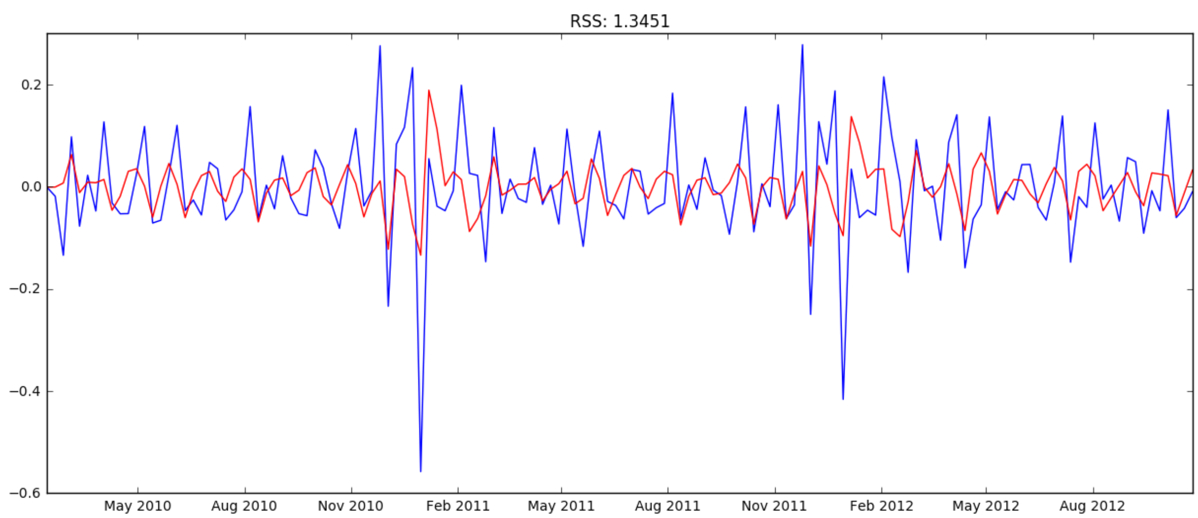
Number of AR (Auto-Regressive) terms (p): AR terms are just lags of dependent variable. For instance if p is 5, the predictors for x(t) will be x(t-1)….x(t-5). Number of MA (Moving Average) terms (q): MA terms are lagged forecast errors in prediction equation. For instance if q is 5, the predictors for x(t) will be e(t-1)….e(t-5) where e(i) is the difference between the moving average at ith instant and actual value. Number of Differences (d): These are the number of nonseasonal differences, i.e. in this case we took the first order difference. So either we can pass that variable and put d=0 or pass the original variable and put d=1. Both will generate same results.

However we will not do it here

AR model - Autoregressive Model

In [37]:
```
model = ARIMA(ts_log, order=(2, 1, 0))
results_AR = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_AR.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_AR.fittedvalues-ts_log_diff)**2))
```

Out[37]: <matplotlib.text.Text at 0x11a1df890>

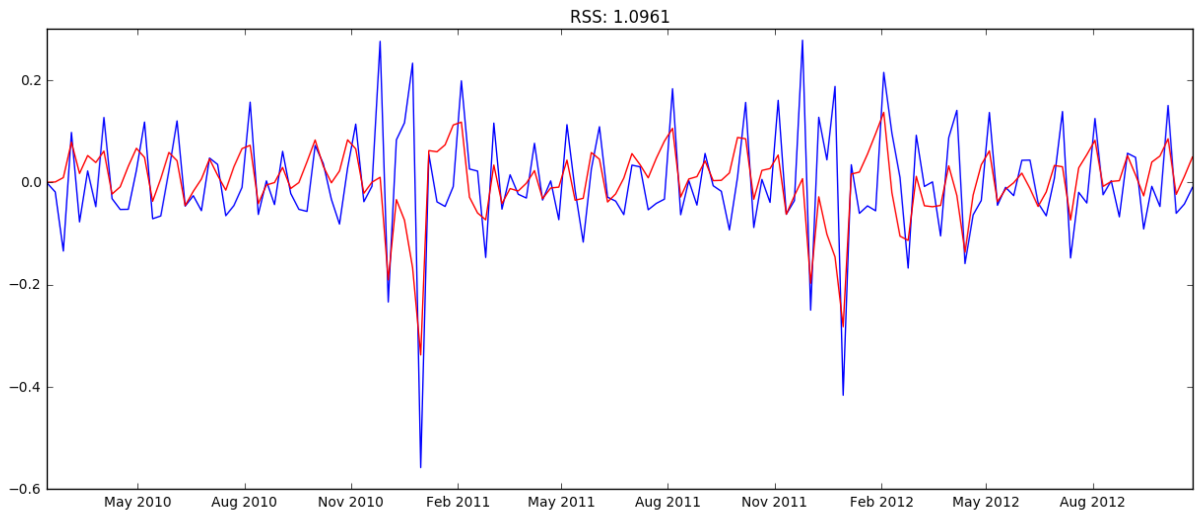


MA model - Moving average model

In [44]:
```
model = ARIMA(ts_log, order=(0, 1, 2))
results_MA = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_MA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_MA.fittedvalues-ts_log_diff)**2))
```

```
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/statsmodels/bas
e/model.py:466: ConvergenceWarning: Maximum Likelihood optimization fai
led to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

Out[44]: <matplotlib.text.Text at 0x11a4800d0>

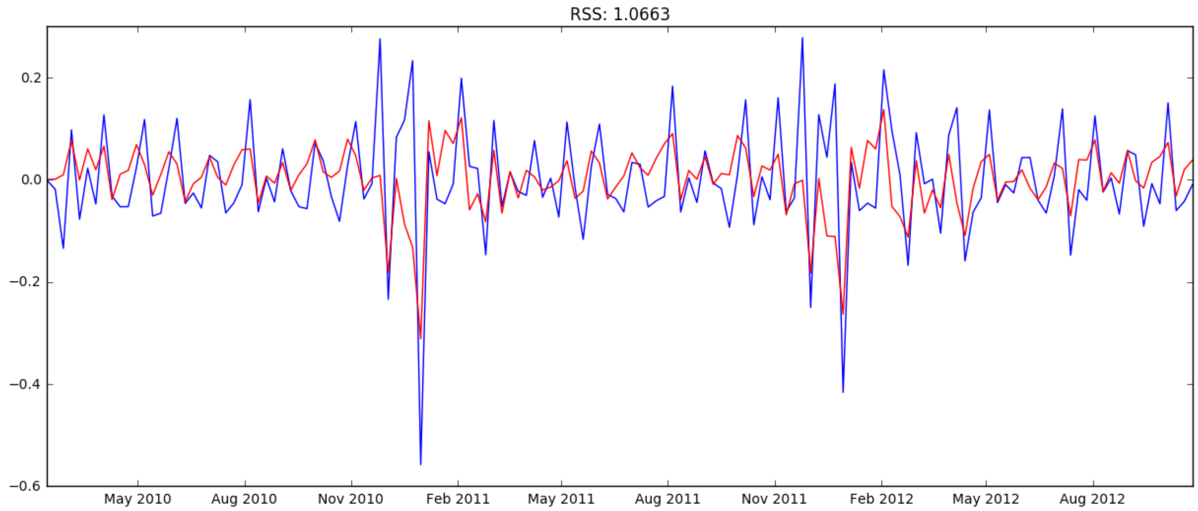


Combined Model

In [45]:
```
model = ARIMA(ts_log, order=(2, 1, 2))
results_ARIMA = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_ARIMA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_ARIMA.fittedvalues-ts_log_diff)**2))
```

```
/Users/azizmamatov/anaconda/lib/python2.7/site-packages/statsmodels/bas
e/model.py:466: ConvergenceWarning: Maximum Likelihood optimization fai
led to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

Out[45]: <matplotlib.text.Text at 0x119eec290>



Now we need to take these value back to the original scale

Since the combined model gave best result, lets scale it back to the original values and see how well it performs there. First step would be to store the predicted results as a separate series and observe it.

In [47]:
```
predictions_ARIMA_diff = pd.Series(results_ARIMA.fittedvalues, copy=True)
print predictions_ARIMA_diff.head()
```

```
2010-02-12    0.000498
2010-02-19    0.001117
2010-02-26    0.009777
2010-03-05    0.077008
2010-03-12   -0.000462
dtype: float64
```

First week is missing as we took lag of 1

You can quickly do some back of mind calculations using previous output to check if these are correct. Next we've to add them to base number. For this lets create a series with all values as base number and add the differences to it. This can be done as:

In [49]:
```
predictions_ARIMA_diff_cumsum = predictions_ARIMA_diff.cumsum()
print predictions_ARIMA_diff_cumsum.head()
predictions_ARIMA_log = pd.Series(ts_log.ix[0], index=ts_log.index)
predictions_ARIMA_log = predictions_ARIMA_log.add(predictions_ARIMA_diff
_cumsum,fill_value=0)
predictions_ARIMA_log.head()
```

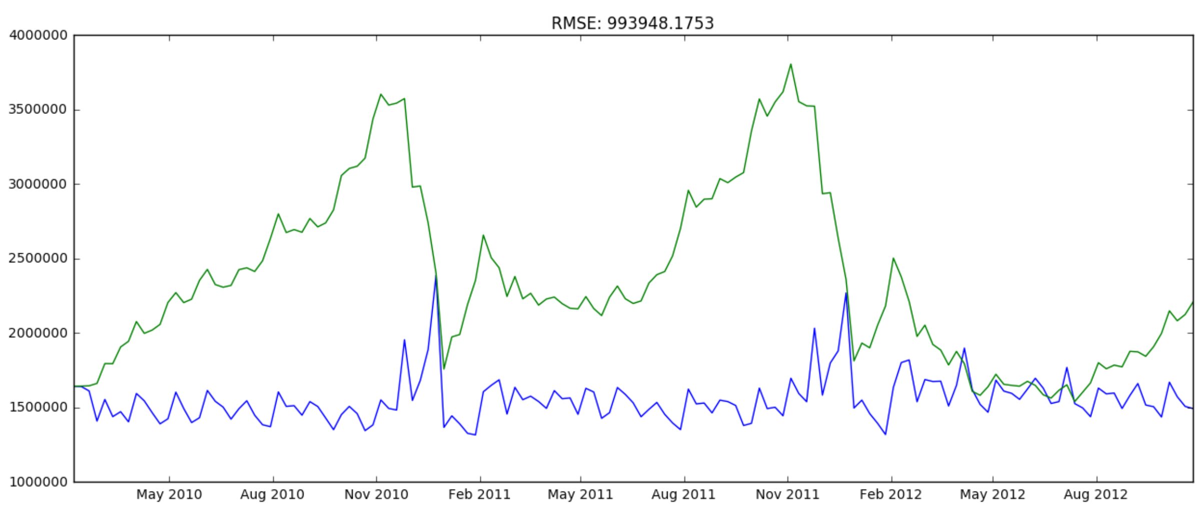
```
2010-02-12    0.000498
2010-02-19    0.001616
2010-02-26    0.011393
2010-03-05    0.088401
2010-03-12    0.087939
dtype: float64
```

Out[49]:
```
2010-02-05    14.312455
2010-02-12    14.312953
2010-02-19    14.314070
2010-02-26    14.323848
2010-03-05    14.400856
dtype: float64
```

Here the first element is base number itself and from thereon the values cumulatively added. Last step is to take the exponent and compare with the original series.

In [53]:
```
predictions_ARIMA = np.exp(predictions_ARIMA_log)
plt.plot(ts_store)
plt.plot(predictions_ARIMA)
plt.title('RMSE: %.4f'% np.sqrt(sum((predictions_ARIMA-
ts_store)**2)/len(ts_store)))
```

Out[53]:
```
<matplotlib.text.Text at 0x11706c550>
```



Connection with SQLite database should be stopped at the end of the session

In [54]:
```
cur.close()
conn.close()
```

In [ ]: