Basics of Single Time-Series

Comparison of Traditional TS to Prophet

Reza Rashetnia July-2020

This Kernel developed and derived from one of my solved Kaggle competitions. The objective is to overview a single time series forecast general steps.

Time Series forecasting:

The act of predicting the future by understanding the past.

Time Series Components:

- **Level**: The baseline value for the series if it were a straight line
- **Trend**: The general tendency of a time series to increase, decrease or stagnate over a long period of time
- **Seasonal**: Fluctuations within a year during the season. The important factors causing seasonal variations are climate and weather conditions, customs, traditional habits, etc.
    - Seasonality refers to the property of a time series that displays periodical patterns that repeats at a constant frequency (m).
- **Cyclical**: describes the medium-term changes in the series, caused by circumstances, which repeat in cycles. The duration of a cycle extends over longer period of time, usually two or more years. For example a business cycle consists of four phases:

```
i) Prosperity
ii) Decline
iii) Depression
iv) Recovery
* Cycles are seasons that do not occur at a fixed rate.
```

- **Noise/irregular**: irregular or random variations in a time series are caused by unpredictable influences, which are not regular and also do not repeat in a particular pattern. These variations are caused by incidences such as war, strike, earthquake, flood, revolution, etc. There is no defined statistical technique for measuring random fluctuations in a time series.

Additive vs. Multiplicative decomposition models:

When series contains trend, seasonality and noise, it is important to understand how they interact.

- Multiplicative model: Based on the assumption that four components of a time series are not necessarily independent and they can affect one another:
$$Y(t) = T(t) \times S(t) \times C(t) \times I(t)$$
- Additive model: To represent a time series as a combination of patterns at different scales such as daily, weekly, seasonally, and yearly, along with an overall trend:
$$Y(t) = T(t) + S(t) + C(t) + I(t)$$
Decoposition is the deconstruction of the series data into its various components: trend, cycle, noise, and seasonal when those exist. Two types of decompositions are: **Additive** and **Multiplicative**.

Example: Total sales prediction for next month

This example is based on one of Kaggle competions. In this example we are provided with daily sales for each store-item combination of a Russian software company. Our task is to predict sales at a monthly level. This example is solved in following steps:

- Package imports and data
- Basic Exploration/EDA
- Stationarity
- Seasonality , Trend and Remainder
- AR , MA , ARMA , ARIMA
- Selecting P and Q using AIC
- Prophet

In [1]:
```
# IMPORTS
# ----------------------------------------------------
# Basic Packages
import numpy as np
import pandas as pd
import random as rd
import datetime
import matplotlib.pyplot as plt
import seaborn as sns
# ----------------------------------------------------
# Time Series
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from pandas.plotting import autocorrelation_plot
from statsmodels.tsa.stattools import adfuller, acf, pacf, arma_order_
select_ic
import statsmodels.formula.api as smf
import statsmodels.tsa.api as smt
import statsmodels.api as sm
import scipy.stats as scs

from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
# ----------------------------------------------------
# setting
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
# Read the data
sales      = pd.read_csv("sales_train.csv")
```

In [3]:
```
sales.head()
```

Out[3]:

	date	date_block_num	shop_id	item_id	item_price	item_cnt_day
0	02.01.2013	0	59	22154	999.00	1.0
1	03.01.2013	0	25	2552	899.00	1.0
2	05.01.2013	0	25	2552	899.00	-1.0
3	06.01.2013	0	25	2554	1709.05	1.0
4	15.01.2013	0	25	2555	1099.00	1.0

In [4]:
```
# Date must be corrected
print(sales.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2935849 entries, 0 to 2935848
Data columns (total 6 columns):
 #   Column          Dtype  
---  ------          -----  
 0   date            object 
 1   date_block_num  int64  
 2   shop_id         int64  
 3   item_id         int64  
 4   item_price      float64
 5   item_cnt_day    float64
dtypes: float64(2), int64(3), object(1)
memory usage: 134.4+ MB
None
```

In [5]:
```
#formatting the date column correctly
sales.date = sales.date.apply(lambda x:datetime.datetime.strptime(x,'%
d.%m.%Y'))
# check
print(sales.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2935849 entries, 0 to 2935848
Data columns (total 6 columns):
 #   Column          Dtype         
---  ------          -----         
 0   date            datetime64[ns]
 1   date_block_num  int64         
 2   shop_id         int64         
 3   item_id         int64         
 4   item_price      float64       
 5   item_cnt_day    float64       
dtypes: datetime64[ns](1), float64(2), int64(3)
memory usage: 134.4 MB
None
```

In [6]:
```
sales.head()
```

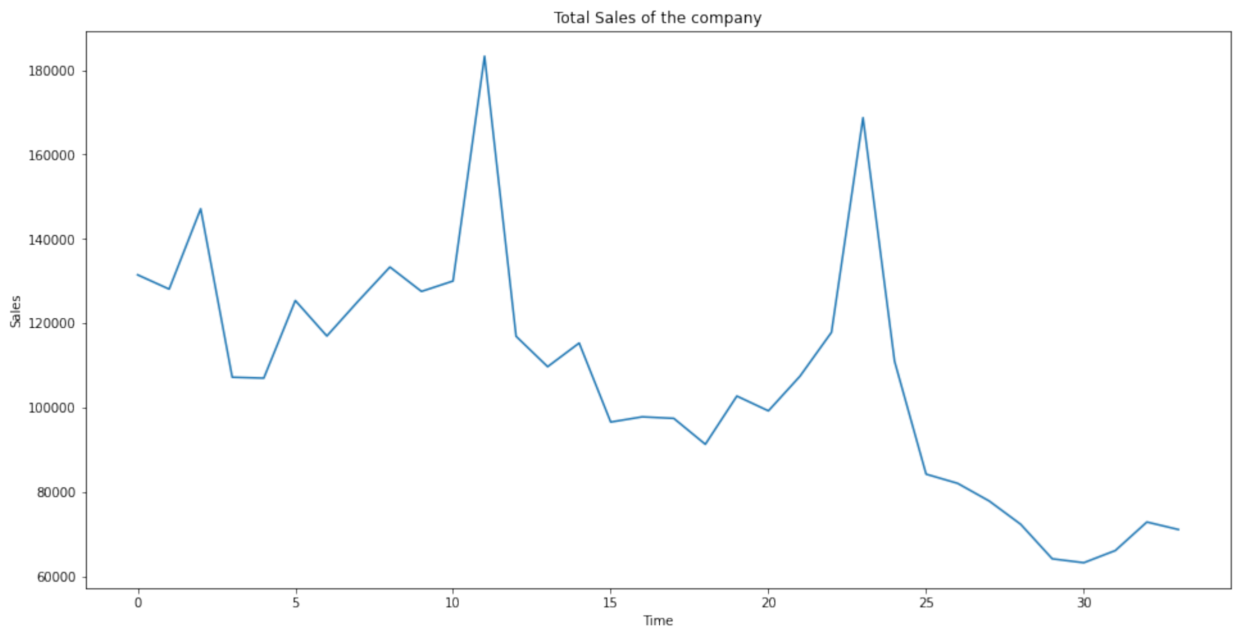
Out[6]:

	date	date_block_num	shop_id	item_id	item_price	item_cnt_day
0	2013-01-02	0	59	22154	999.00	1.0
1	2013-01-03	0	25	2552	899.00	1.0
2	2013-01-05	0	25	2552	899.00	-1.0
3	2013-01-06	0	25	2554	1709.05	1.0
4	2013-01-15	0	25	2555	1099.00	1.0

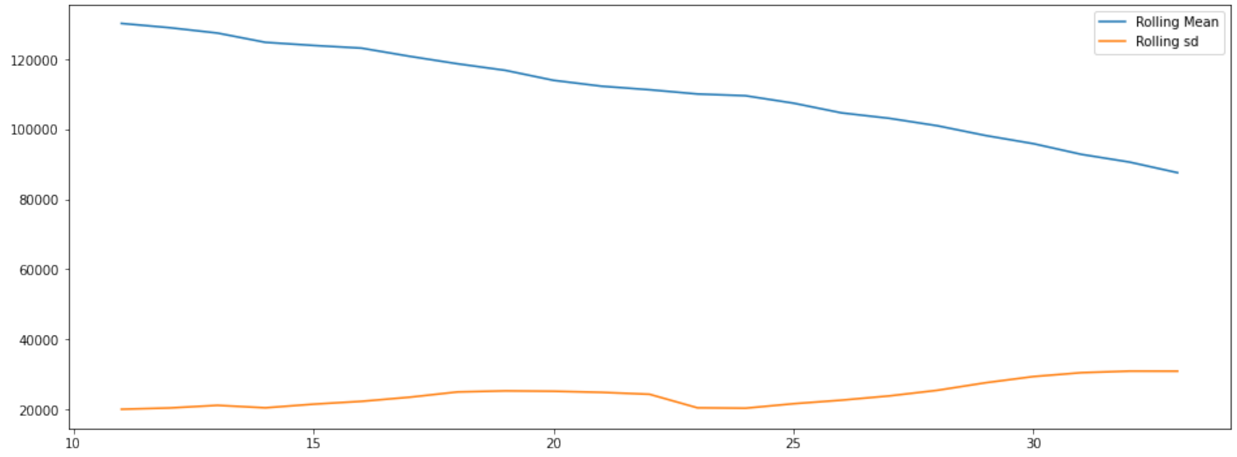
Single Time-Series

Let's compute the total sales per month and plot that data.

In [7]:
```
ts = sales.groupby(["date_block_num"])["item_cnt_day"].sum()
ts.astype('float')
plt.figure(figsize=(16,8))
plt.title('Total Sales of the company')
plt.xlabel('Time')
plt.ylabel('Sales')
plt.plot(ts);
```

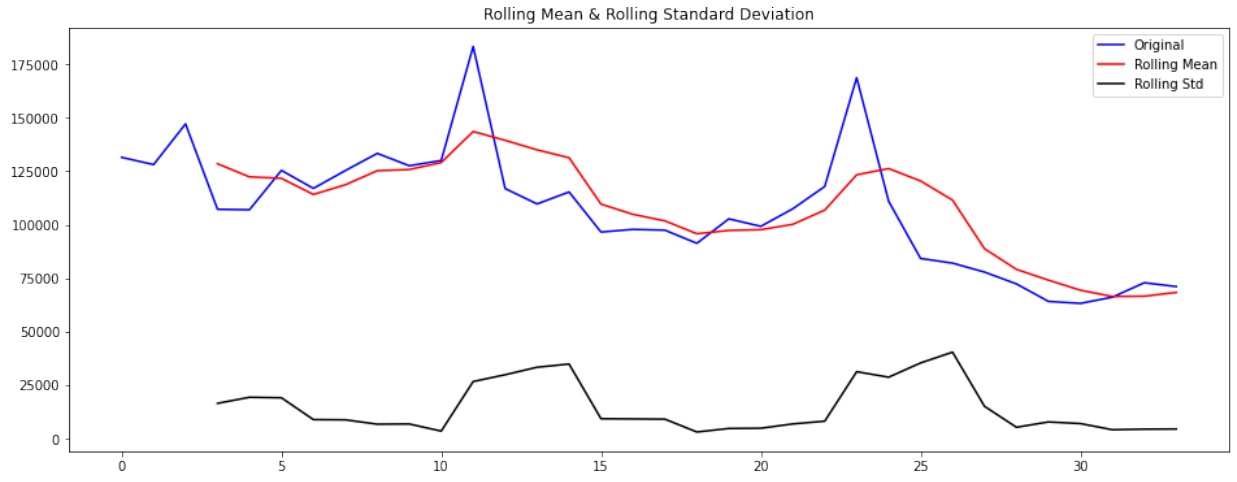


In [8]:
```
plt.figure(figsize=(16,6))
plt.plot(ts.rolling(window=12,center=False).mean(),label='Rolling Mean
');
plt.plot(ts.rolling(window=12,center=False).std(),label='Rolling sd');
plt.legend();
```

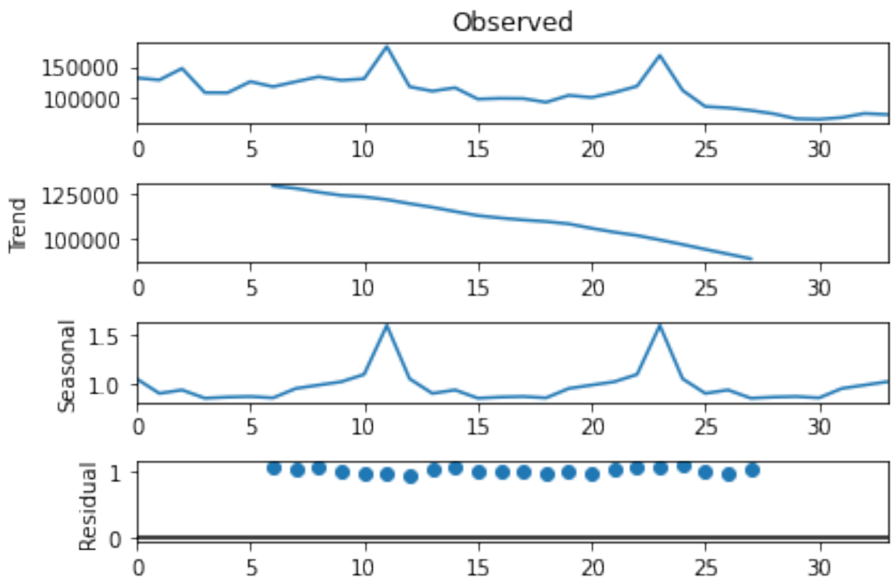


Observation:There is an obvious "seasonality" (Eg: peak sales around a time of year) and a decreasing "Trend". Let's check that with a quick decomposition into Trend, seasonality and residuals.

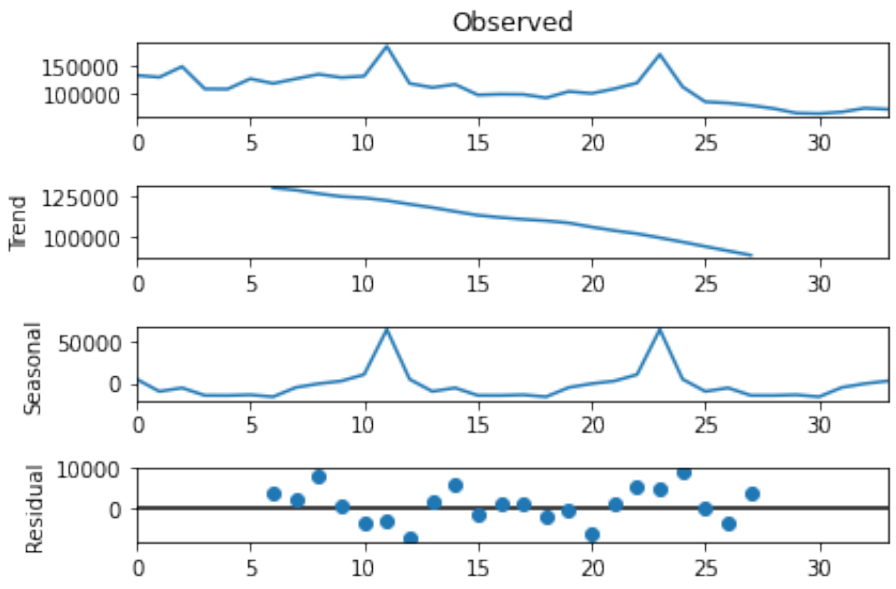
In [9]:
```
rolling_mean = ts.rolling(window = 4).mean()
rolling_std = ts.rolling(window = 4).std()
plt.figure(figsize=(16,6))
plt.plot(ts, color = 'blue', label = 'Original')
plt.plot(rolling_mean, color = 'red', label = 'Rolling Mean')
plt.plot(rolling_std, color = 'black', label = 'Rolling Std')
plt.legend(loc = 'best')
plt.title('Rolling Mean & Rolling Standard Deviation')
plt.show()
```



In [10]:
```
import statsmodels.api as sm
# multiplicative
res = sm.tsa.seasonal_decompose(ts.values,freq=12,model="multiplicativ
e")
#plt.figure(figsize=(16,12))
fig = res.plot()
#fig.show()
```



In [11]:
```
# Additive model
res = sm.tsa.seasonal_decompose(ts.values,freq=12,model="additive")
#plt.figure(figsize=(16,12))
fig = res.plot()
#fig.show()
```



we assume an additive model, then we can write

$$y(t) = S(t) + T(t) + \epsilon(t)$$

where yt is the data at period t, St is the seasonal component at period t, Tt is the trend-cycle component at period tt and Et is the remainder (or irregular or error) component at period t Similarly for Multiplicative model,

$$y(t) = S(t) \times T(t) \times \epsilon(t)$$

Stationarity

What does it mean for data to be stationary?

1. The mean of the series should not be a function of time.
2. The variance of the series should not be a function of time (The varying of data not spread over time).
3. The covariance of the ith term and the (i+m)th term should not be a function of time.

Stationarity refers to time-invariance of a series. It means that two points in a time series are related to each other by only how far apart they are, and not by the direction (forward/backward). This is a necessary condition for building time series model. The concept of stationarity is a mathematical idea constructed to simplify the theoretical and practical development of stochastic processes.

Weakly Stationary: For practical applications, the assumption of strong stationarity is not always needed. A stochastic process is said to be Weakly Stationary of order k if the statistical moments of the process up to that order depend only on time differences and not upon the time of occurrences of the data being used to estimate the moments. For example a stochastic process $x(t), t = 0, 1, 2, \ldots$ is second order stationary if it has time independent mean and variance and the covariance values $Cov(x_t, x_{t-s})$ depend only on s.

There are multiple tests that can be used to check stationarity:

1. ADF (Augmented Dicky Fuller Test)
2. KPSS
3. PP (Phillips-Perron test)

I used ADF which is the most commonly used one. Mathematical tests like Dickey and Fuller are generally used to detect stationarity in a time series data. Usually time series, showing trend or seasonal patterns are non-stationary in nature. In such cases, differencing and power transformations are often used to remove the trend and to make the series stationary.

- **Dickey Fuller Test of Stationarity**: One of the statistical tests for checking stationarity. Here the null hypothesis is that the TS is non-stationary. The test results comprise of a Test Statistic and some Critical Values for difference confidence levels. If the *‘Test Statistic’* is less than the *‘Critical Value’*, we can reject the null hypothesis and say that the series is stationary. Below we have to test if $\rho - 1$ is significantly different than zero or not. If the null hypothesis gets rejected, we'll get a stationary time series.

$$x(t) = \rho \times x(t-1) + \epsilon(t)$$
$$x(t) - x(t-1) = (\rho - 1) \times x(t-1) + \epsilon(t)$$

- **Model Parsimony Principle**: According to this principle, which is similar to similar to the Occam's razor principle, always the model with smallest possible number of parameters is to be selected so as to provide an adequate representation of the underlying time series data. The more complicated the model, the more possibilities will arise for departure from the actual model assumptions. Also, the increase of model parameters increases the risk of overfitting.

ADF(Augmented Dicky Fuller Test)

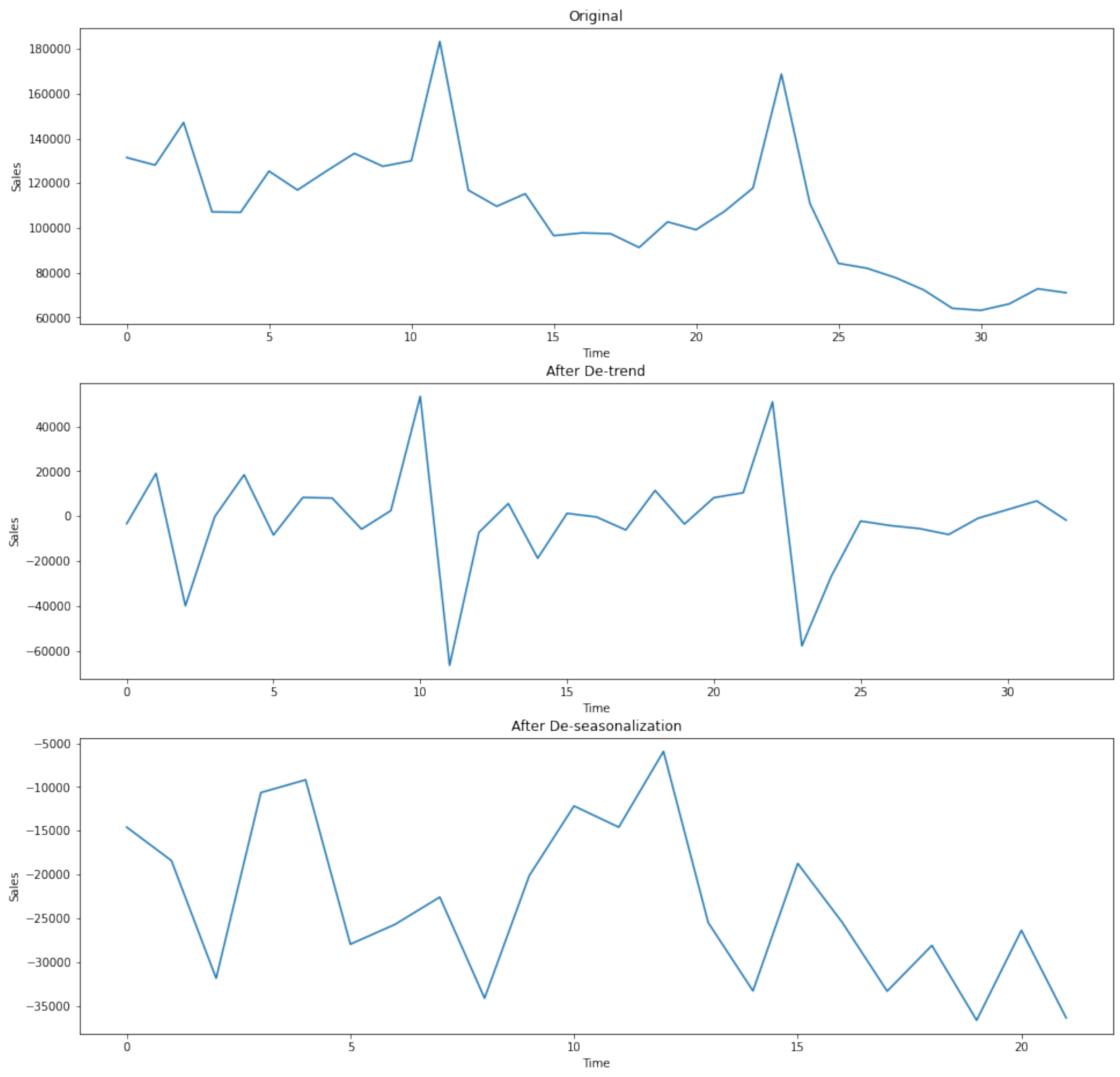
In [12]:
```
# Stationarity tests
def test_stationarity(timeseries):

    #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)

test_stationarity(ts)
```

```
Results of Dickey-Fuller Test:
Test Statistic                 -2.395704
p-value                         0.142953
#Lags Used                      0.000000
Number of Observations Used    33.000000
Critical Value (1%)            -3.646135
Critical Value (5%)            -2.954127
Critical Value (10%)           -2.615968
dtype: float64
```

In [13]:
```
# to remove trend
from pandas import Series as Series
# create a differenced series
def difference(dataset, interval=1):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return Series(diff)

# invert differenced forecast
def inverse_difference(last_ob, value):
    return value + last_ob
```

In [14]:
```
ts=sales.groupby(["date_block_num"])["item_cnt_day"].sum()
ts.astype('float')
plt.figure(figsize=(16,16))
plt.subplot(311)
plt.title('Original')
plt.xlabel('Time')
plt.ylabel('Sales')
plt.plot(ts)
plt.subplot(312)
plt.title('After De-trend')
plt.xlabel('Time')
plt.ylabel('Sales')
new_ts=difference(ts)
plt.plot(new_ts)
plt.plot()

plt.subplot(313)
plt.title('After De-seasonalization')
plt.xlabel('Time')
plt.ylabel('Sales')
new_ts=difference(ts,12)       # assuming the seasonality is 12 months
long
plt.plot(new_ts)
plt.plot()
```

Out[14]: []



In [15]:
```
# now testing the stationarity again after de-seasonality
test_stationarity(new_ts)
```

```
Results of Dickey-Fuller Test:
Test Statistic                 -3.270101
p-value                         0.016269
#Lags Used                      0.000000
Number of Observations Used    21.000000
Critical Value (1%)            -3.788386
Critical Value (5%)            -3.013098
Critical Value (10%)           -2.646397
dtype: float64
```

Now after the transformations, our p-value for the DF test is well within 5 %. Hence we can assume Stationarity of the series We can easily get back the original series using the inverse transform function that we have defined above.

Now let's dive into making the forecasts!

Traditional Forecasting

Here is the process:

1. Visualize the time series
2. Stationarize the series
3. Plot ACF/PACF charts and find optimal paramters
4. Build the ARIMA model
5. Make Prediction

Linear vs. non-linear models: A time series model is said to be linear or non-linear depending on whether the current value of the series is a linear or non-linear function of past observations.

$$x_t = \sum_{j=1}^{p} a_j x_{t-j} + \epsilon_t$$

Linear models: Autoregressive (AR) & Moving Average (MA) → combining these two leads to ARMA and Autoregressive Fractionally Integrated Moving Average (ARFIMA). Seasonal ARIMA or SARIMA is used for seasonal time series forecasting.

Box-Jenkins principle ARIMA and its derivate models are based on this principle. The Box-Jenkins Model forecasts data using three principles,

- **autoregression (p)**: tests the data for its level of stationarity.
- **differencing**: If the data being used is non-stationary it will need to be differenced (d).
- **moving average**: The data is also tested for its moving average fit which is done in part q of the analysis process. These three principles are known as p, d and q respectively. Each principle is used in the Box-Jenkins analysis and together they are collectively shown as ARIMA (p, d, q). Overall, initial analysis of the data prepares it for forecasting by determining the parameters (p, d and q) which are applied to develop a forecast. ARMA = AR(p) + MA(q) suitable for univariate time-forecasting AR or MA are not applicable on non-stationary series AR(p)

$$x_t = c + \sum_{j=1}^{p} a_j x_{t-j} + \epsilon_t$$

The integer p is the order of the model. c is sometimes omitted for simplicity MA(q): Just as an AR(p) model regress against past values of the series, an MA(q) model uses past errors as the explanatory variables.

$$x_t = \mu + \sum_{j=1}^{q} b_j \epsilon_{t-j} + \epsilon_t$$

μ is mean of the series. q is order of the model. Conceptually a moving average model is a linear regression of the current observation of the time series against the random shocks (error terms) of one or more prior observations AR vs. MA: the primary difference between an AR and MA model is based on the correlation between time series objects at different time points. The correlation between x(t) and x(t-n) for n > order of MA is always zero. However, in the AR models the correlation of x(t) and x(t-n) gradually declines with n becoming larger. ϵ_i : error terms The error terms or random shocks are assumed to be a white noise process, i.e. a sequence of independent and identically distributed (i.i.d) random variables with zero mean and a constant variance ϵ^2. Generally, the random shocks are assumed to follow the typical normal distribution. ARMA(p,q):

$$x_t = c + \epsilon_t + \sum_{j=1}^{p} a_j x_{t-j} + \sum_{j=1}^{q} b_j \epsilon_{t-j}$$

ARMA stationary condition depends on AR as MA is always stationary. ARIMA: ARMA can be used for stationary time series only. Time series that contain trend and seasonal patterns are non-stationary in nature making ARMA models non-adequate. ARIMA makes the non-stationary models stationary by applying finite differencing of the data points. The integer d controls the level of differencing. Generally d=1 is enough in most cases. When d=0, then it reduces to an ARMA(p,q) model. SARIMA ARIMA works for non-stationary non-seasonal data. In SARIMA seasonal differencing of appropriate order is used to remove non-stationarity from the series. A first order seasonal difference is the difference between an observation and the corresponding observation from the previous year and is calculated as $z_t = x_t - x_{t-s}$. For monthly time series s = 12 and for quarterly time series s = 4. Measuring forecast performance The common measures in linear regression can be used.

AR, MA and ARMA models:

ARIMA is a model which is used for predicting future trends on a time series data. It is model that form of regression analysis.

- AR (Autoregression): Model that shows a changing variable that regresses on its own lagged/prior values.
- I (Integrated): Differencing of raw observations to allow for the time series to become stationary
- MA (Moving average): Dependency between an observation and a residual error from a moving average model For ARIMA models, a standard notation would be ARIMA with p, d, and q, where integer values substitute for the parameters to indicate the type of ARIMA model used.
- p: the number of lag observations in the model; also known as the lag order.
- d: the number of times that the raw observations are differenced; also known as the degree of differencing.
- q: the size of the moving average window; also known as the order of the moving average.

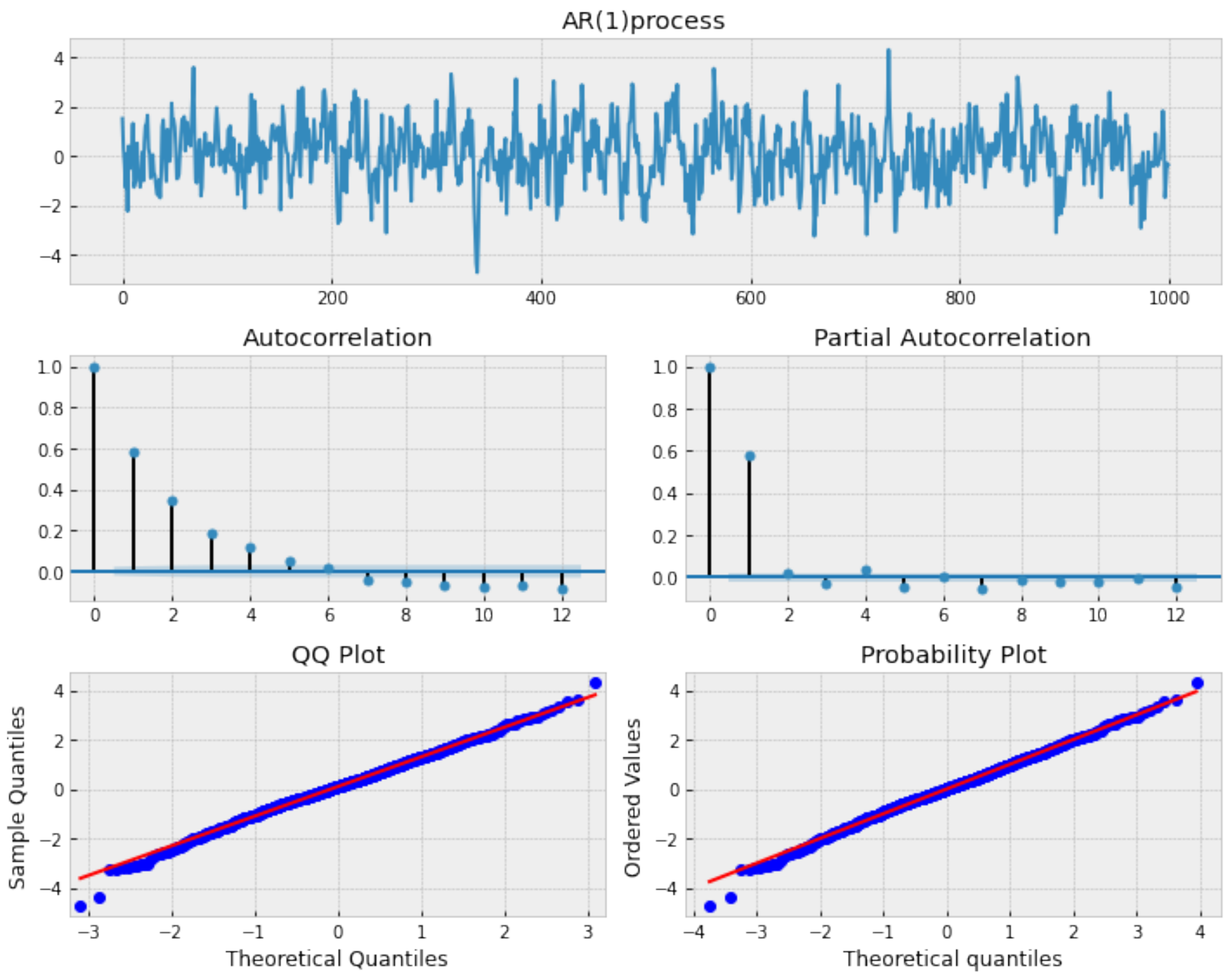
In the domain of machine learning, there is a collection techniques for manipulating and interpreting variables that depend on time. Among these include ARIMA which can remove the trend component in order to accurately predict future values.

Now, How do we find out, if our time-series in AR process or MA process?

Let's find out!

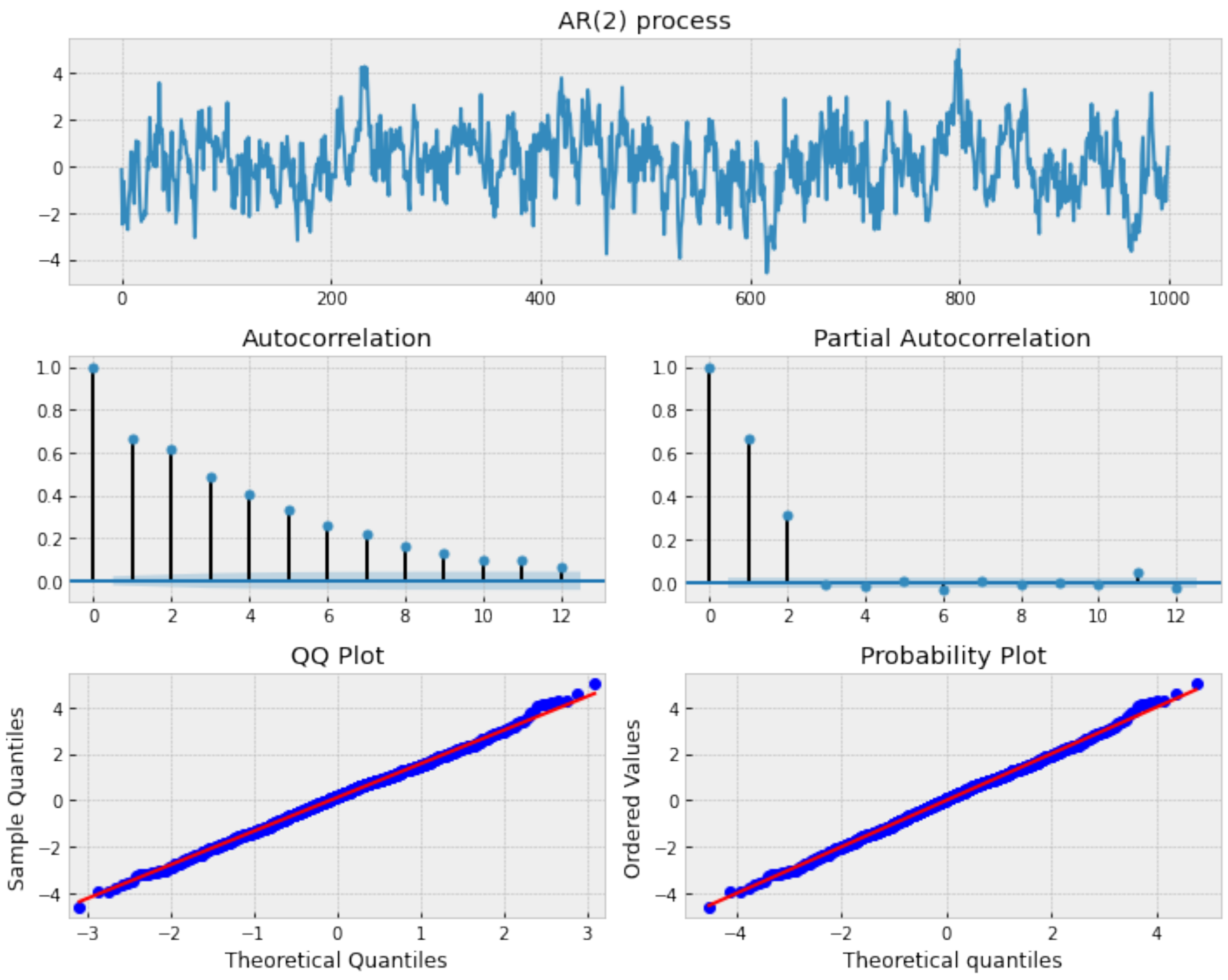
In [16]:
```
def tsplot(y, lags=None, figsize=(10, 8), style='bmh',title=''):
    if not isinstance(y, pd.Series):
        y = pd.Series(y)
    with plt.style.context(style):
        fig = plt.figure(figsize=figsize)
        #mpl.rcParams['font.family'] = 'Ubuntu Mono'
        layout = (3, 2)
        ts_ax = plt.subplot2grid(layout, (0, 0), colspan=2)
        acf_ax = plt.subplot2grid(layout, (1, 0))
        pacf_ax = plt.subplot2grid(layout, (1, 1))
        qq_ax = plt.subplot2grid(layout, (2, 0))
        pp_ax = plt.subplot2grid(layout, (2, 1))

        y.plot(ax=ts_ax)
        ts_ax.set_title(title)
        smt.graphics.plot_acf(y, lags=lags, ax=acf_ax, alpha=0.5)
        smt.graphics.plot_pacf(y, lags=lags, ax=pacf_ax, alpha=0.5)
        sm.qqplot(y, line='s', ax=qq_ax)
        qq_ax.set_title('QQ Plot')
        scs.probplot(y, sparams=(y.mean(), y.std()), plot=pp_ax)

        plt.tight_layout()
    return
```

In [17]:
```
# Simulate an AR(1) process with alpha = 0.6
np.random.seed(1)
n_samples = int(1000)
a = 0.6
x = w = np.random.normal(size=n_samples)

for t in range(n_samples):
    x[t] = a*x[t-1] + w[t]
limit=12
_ = tsplot(x, lags=limit,title="AR(1)process")
```



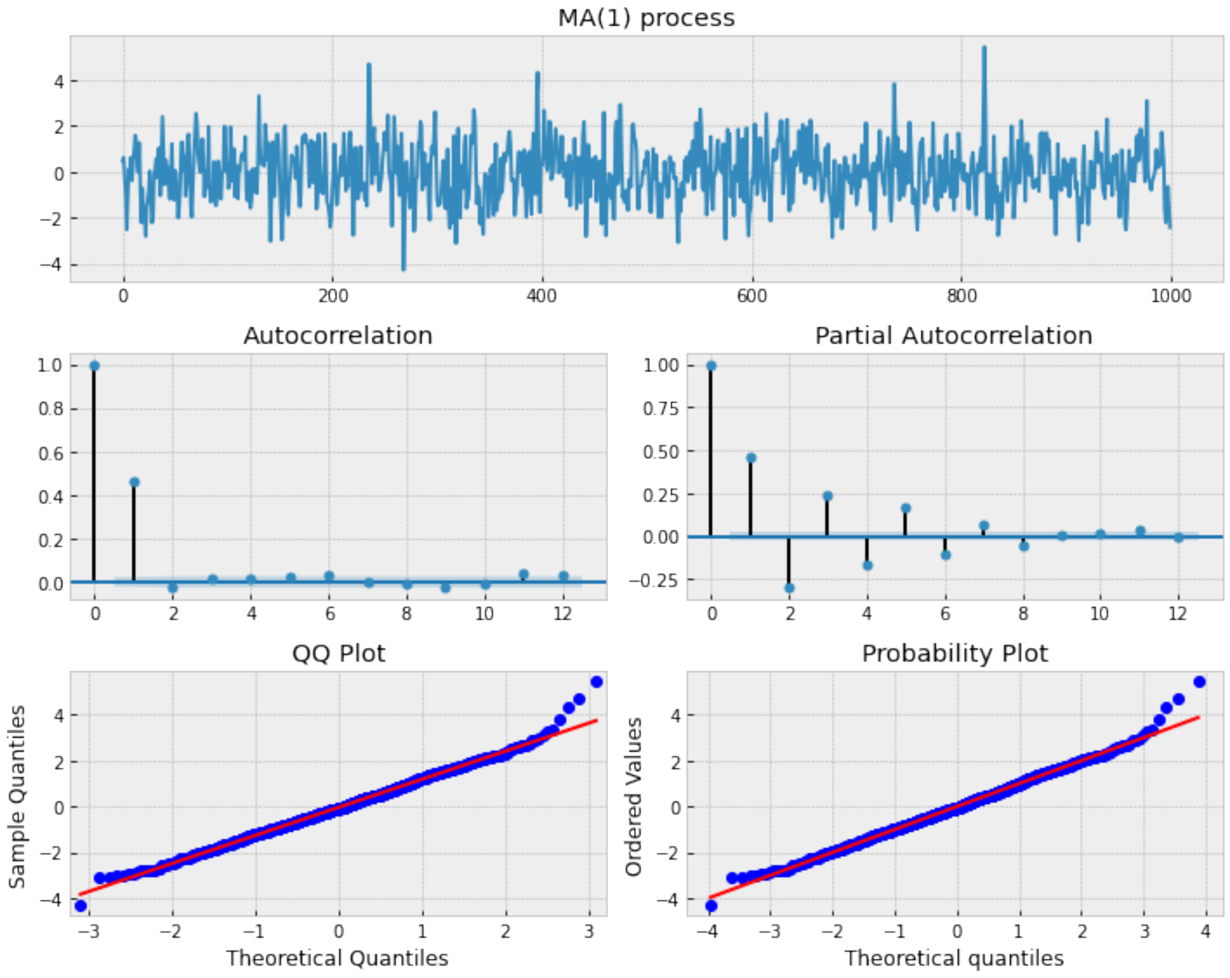
AR(1) process -- has ACF tailing out and PACF cutting off at lag=1

In [18]:
```python
# Simulate an AR(2) process

n = int(1000)
alphas = np.array([.444, .333])
betas = np.array([0.])

# Python requires us to specify the zero-lag value which is 1
# Also note that the alphas for the AR model must be negated
# We also set the betas for the MA equal to 0 for an AR(p) model
# For more information see the examples at statsmodels.org
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]

ar2 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n)
_ = tsplot(ar2, lags=12,title="AR(2) process")
```



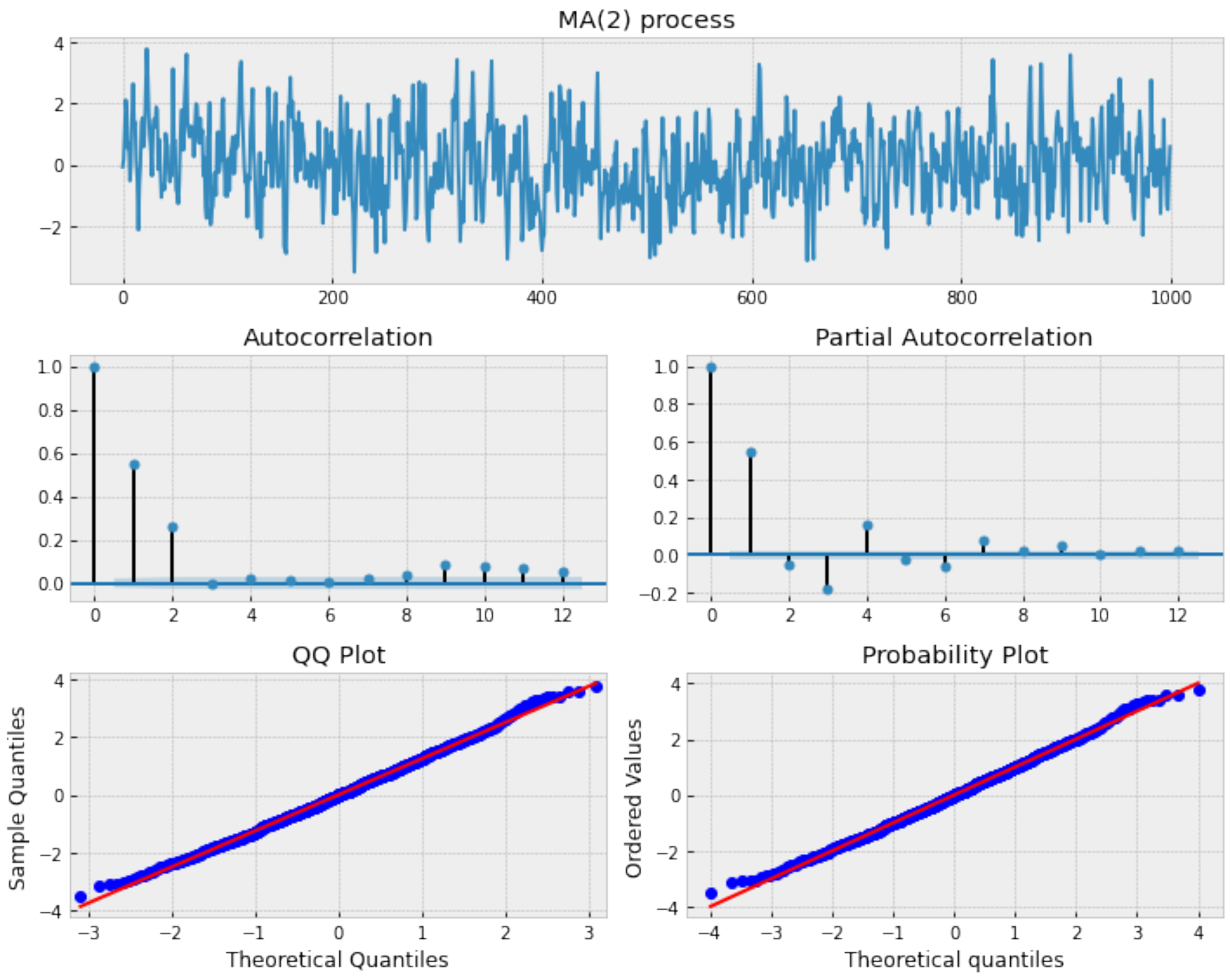
AR(2) process -- has ACF tailing out and PACF cutting off at lag=2

In [19]:
```
# Simulate an MA(1) process
n = int(1000)
# set the AR(p) alphas equal to 0
alphas = np.array([0.])
betas = np.array([0.8])
# add zero-lag and negate alphas
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]
ma1 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n)
limit=12
_ = tsplot(ma1, lags=limit,title="MA(1) process")
```



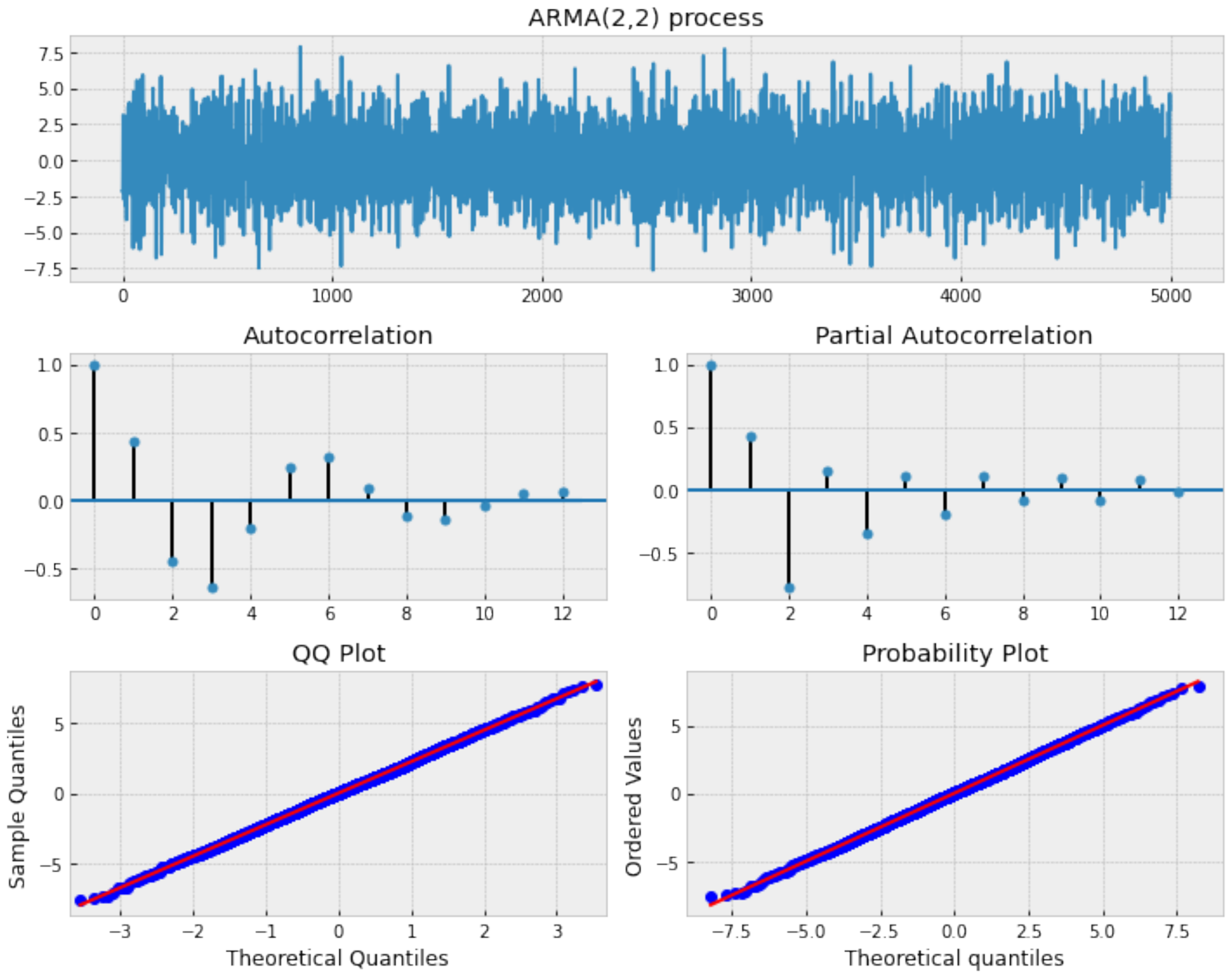
MA(1) process -- has ACF cut off at lag=1

In [20]:
```python
# Simulate MA(2) process with betas 0.6, 0.4
n = int(1000)
alphas = np.array([0.])
betas = np.array([0.6, 0.4])
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]

ma3 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n)
_ = tsplot(ma3, lags=12,title="MA(2) process")
```



MA(2) process -- has ACF cut off at lag=2

In [21]:
```
# Simulate an ARMA(2, 2) model with alphas=[0.5,-0.25] and betas=[0.5,
-0.3]
max_lag = 12

n = int(5000) # lots of samples to help estimates
burn = int(n/10) # number of samples to discard before fit

alphas = np.array([0.8, -0.65])
betas = np.array([0.5, -0.7])
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]

arma22 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n, burnin=burn
)
_ = tsplot(arma22, lags=max_lag,title="ARMA(2,2) process")
```



Now things get a little hazy. Its not very clear/straight-forward. A nifty summary of the above plots:

ACF Shape	Indicated Model
Exponential, decaying to zero	Autoregressive model. Use the partial autocorrelation plot to identify the order of the autoregressive model
Alternating positive and negative, decaying to zero Autoregressive model. Use the partial autocorrelation plot to help identify the order.	
One or more spikes, rest are essentially zero	Moving average model, order identified by where plot becomes zero.
Decay, starting after a few lags	Mixed autoregressive and moving average (ARMA) model.
All zero or close to zero	Data are essentially random.
High values at fixed intervals	Include seasonal autoregressive term.
No decay to zero	Series is not stationary

Let's use a systematic approach to finding the order of AR and MA processes.

In [22]:
```
# pick best order by aic
# smallest aic value wins
best_aic = np.inf
best_order = None
best_mdl = None

rng = range(5)
for i in rng:
    for j in rng:
        try:
            tmp_mdl = smt.ARMA(arma22, order=(i, j)).fit(method='mle',
trend='nc')
            tmp_aic = tmp_mdl.aic
            if tmp_aic < best_aic:
                best_aic = tmp_aic
                best_order = (i, j)
                best_mdl = tmp_mdl
        except: continue


print('aic: {:6.5f} | order: {}'.format(best_aic, best_order))
```

```
aic: 15326.68109 | order: (2, 2)
```

We've correctly identified the order of the simulated process as ARMA(2,2). Lets use it for the sales time-series.

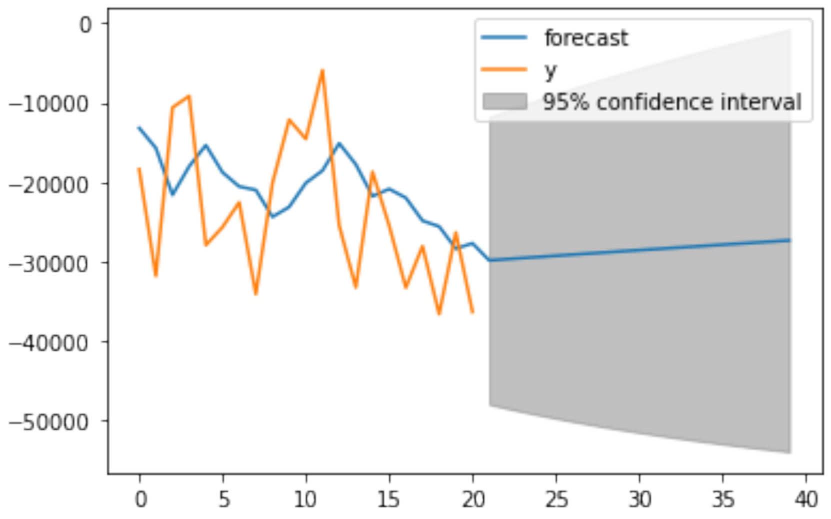
In [23]:
```
#
# pick best order by aic
# smallest aic value wins
best_aic = np.inf
best_order = None
best_mdl = None

rng = range(5)
for i in rng:
    for j in rng:
        try:
            tmp_mdl = smt.ARMA(new_ts.values, order=(i, j)).fit(method
='mle', trend='nc')
            tmp_aic = tmp_mdl.aic
            if tmp_aic < best_aic:
                best_aic = tmp_aic
                best_order = (i, j)
                best_mdl = tmp_mdl
        except: continue


print('aic: {:6.5f} | order: {}'.format(best_aic, best_order))
```

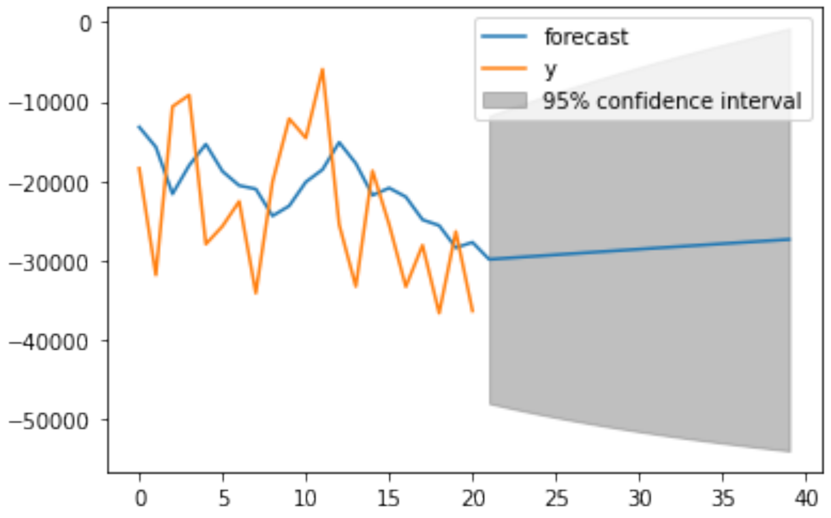
```
aic: 472.99703 | order: (1, 1)
```

Simply use best_mdl.predict() to predict the next values

In [24]:
```
best_mdl.plot_predict(1,40)
```

Out[24]:





In [25]:

```
# adding the dates to the Time-series as index
ts=sales.groupby(["date_block_num"])["item_cnt_day"].sum()
ts.index=pd.date_range(start = '2013-01-01',end='2015-10-01', freq = '
MS')
ts=ts.reset_index()
ts.head()
```

Out[25]:

	index	item_cnt_day
0	2013-01-01	131479.0
1	2013-02-01	128090.0
2	2013-03-01	147142.0
3	2013-04-01	107190.0
4	2013-05-01	106970.0

Prophet

Prophet: Recently open-sourced by Facebook research. It's a very promising tool, that is often a very handy and quick solution to the frustrating flatline. Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. In its essence, Prophet library utilizes the additive regression model.

Sure, one could argue that with proper pre-processing and carefully tuning the parameters the above graph would not happen.

But the truth is that most of us don't either have the patience or the expertise to make it happen.

Also, there is the fact that in most practical scenarios- there is often a lot of time-series that needs to be predicted. Sometimes we need to predict multi level combinations which could be in the thousands.(ie) predict 1000s of parameters!

Another neat functionality is that it follows the typical sklearn syntax.

At its core, the Prophet procedure is an additive regression model with four main components:

A piecewise linear or logistic growth curve trend. Prophet automatically detects changes in trends by selecting changepoints from the data. A yearly seasonal component modeled using Fourier series. A weekly seasonal component using dummy variables. A user-provided list of important holidays.

In [26]:
```
from fbprophet import Prophet
#prophet reqiures a pandas df at the below config
# ( date column named as DS and the value column as Y)
ts.columns=['ds','y']
model = Prophet( yearly_seasonality=True) #instantiate Prophet with on
ly yearly seasonality as our data is monthly
model.fit(ts) #fit the model with your dataframe
```

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly
_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_s
easonality=True to override this.
```

Out[26]:
```
<fbprophet.forecaster.Prophet at 0x131845a00>
```

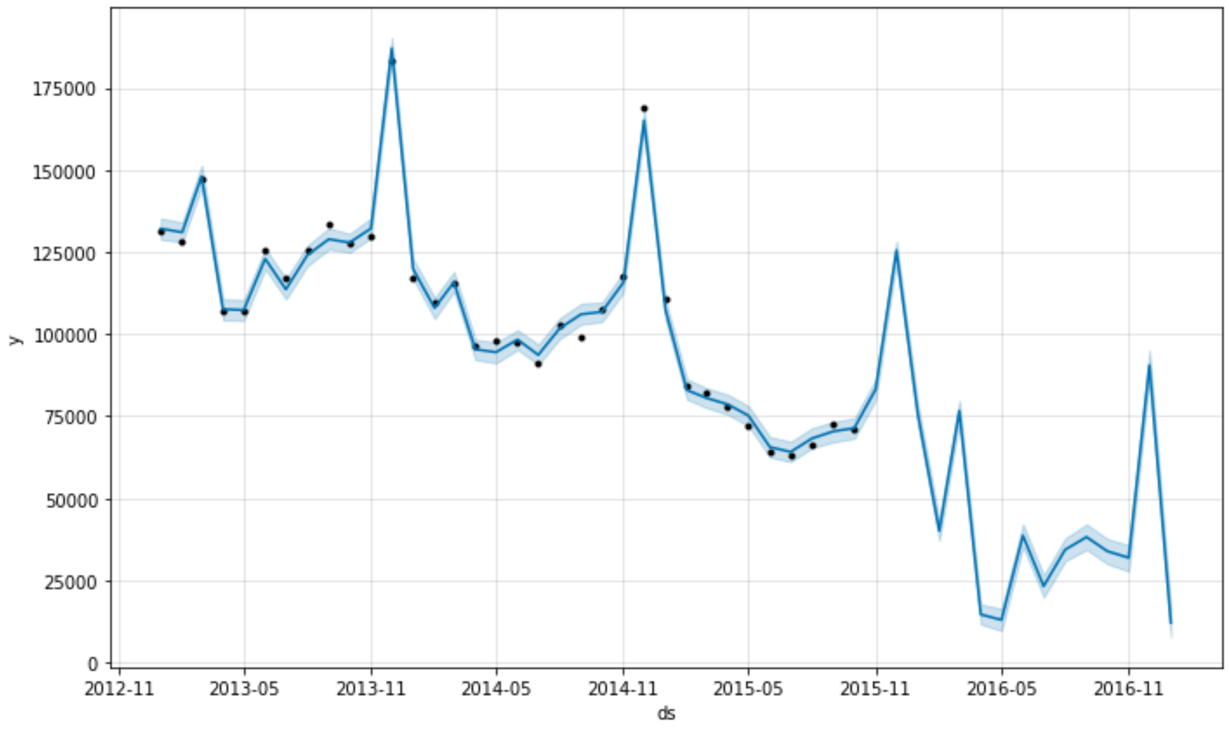
In [27]:
```
# predict for five months in the furure and MS - month start is the fr
equency
future = model.make_future_dataframe(periods = 15, freq = 'MS')
# now lets make the forecasts
forecast = model.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

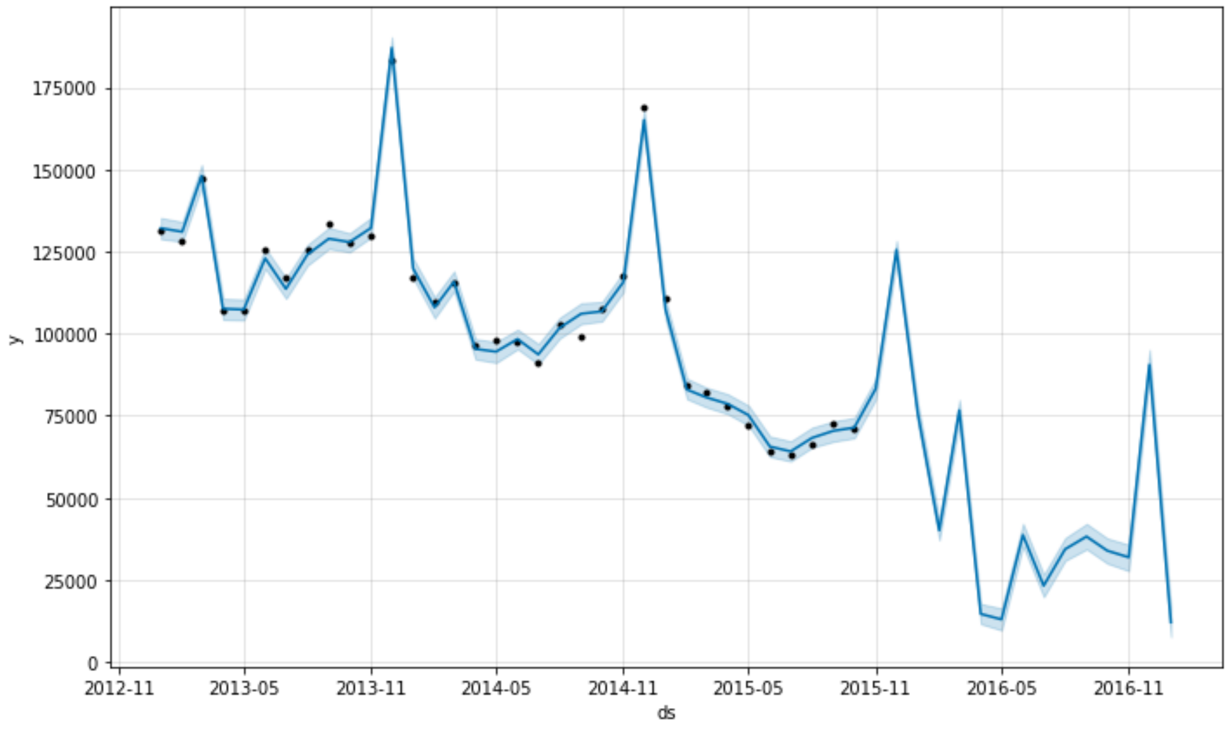
Out[27]:

	ds	yhat	yhat_lower	yhat_upper
44	2016-09-01	38214.609496	34388.645257	42136.103291
45	2016-10-01	33905.676617	30051.347546	37704.506767
46	2016-11-01	31890.319472	27787.453419	35751.410084
47	2016-12-01	90549.521473	86393.261744	95079.376010
48	2017-01-01	12121.271925	7731.440892	16680.101206

In [28]:
```
model.plot(forecast)
```

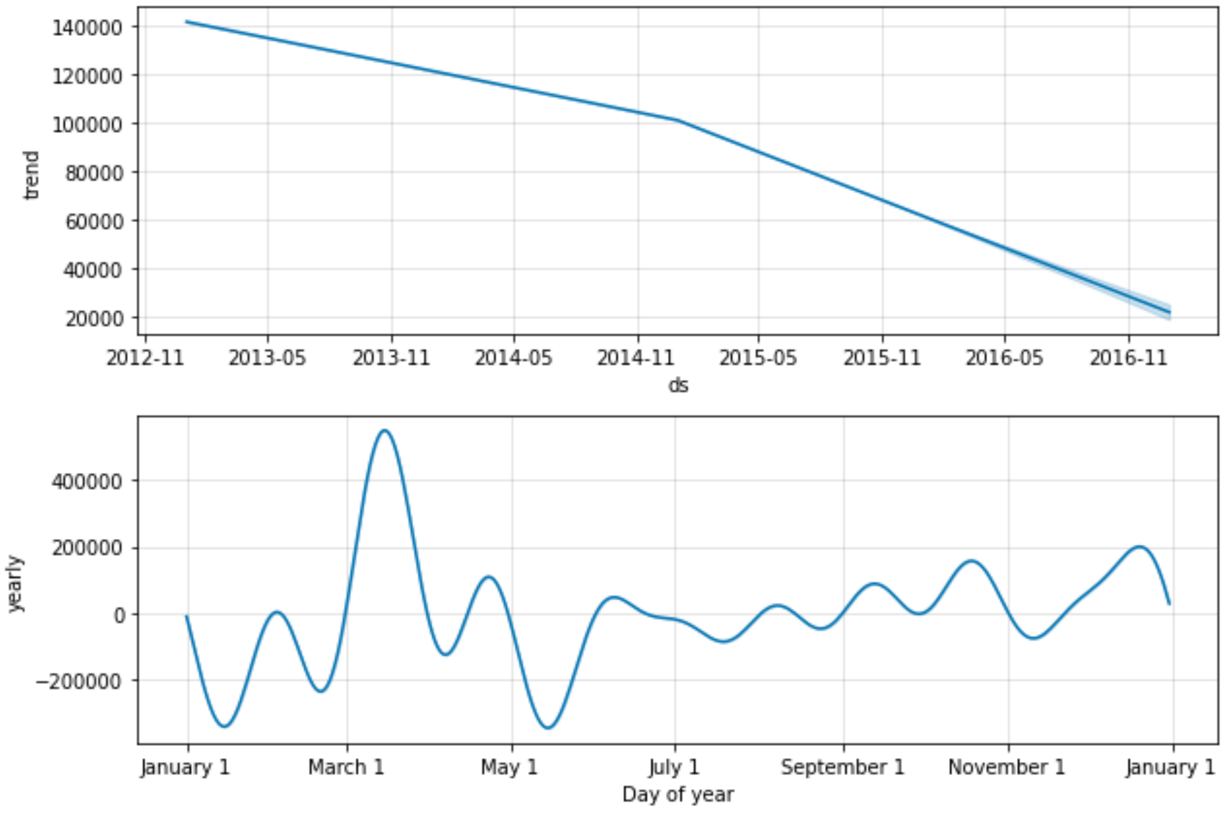
Out[28]:

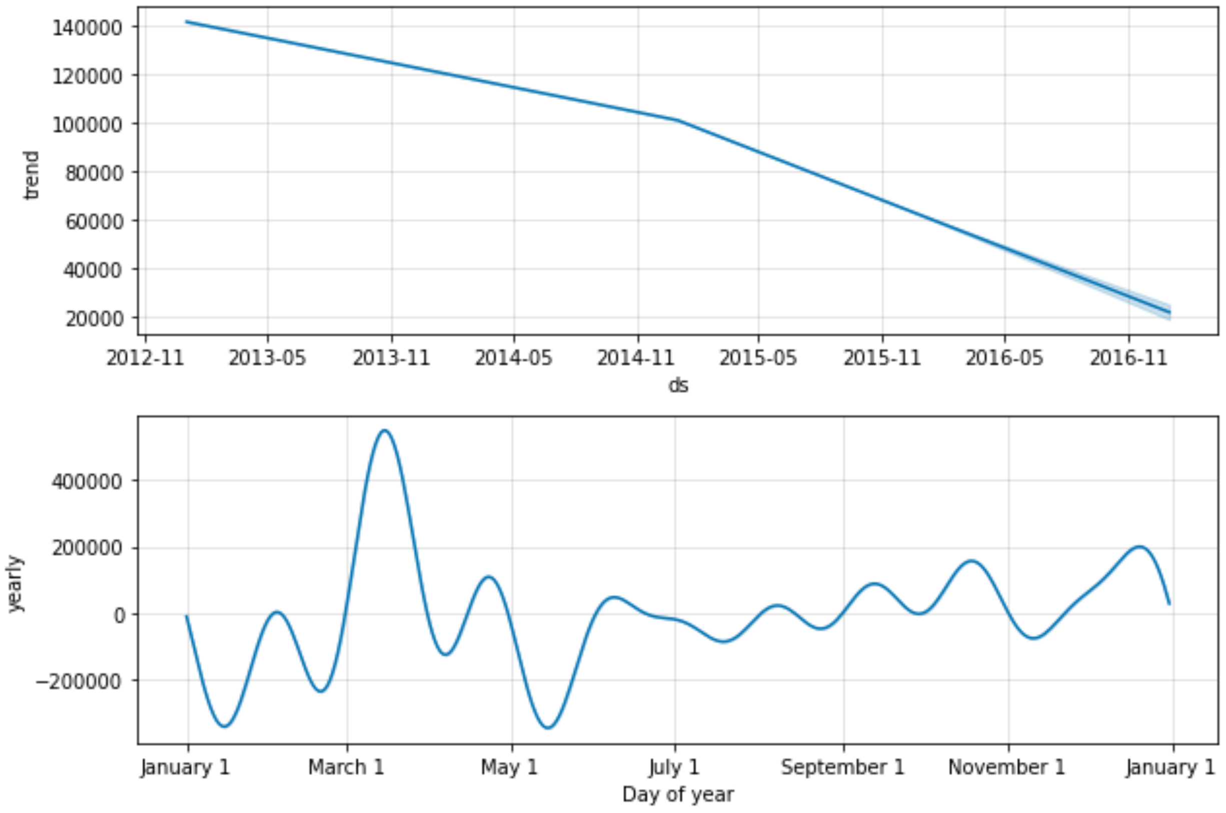




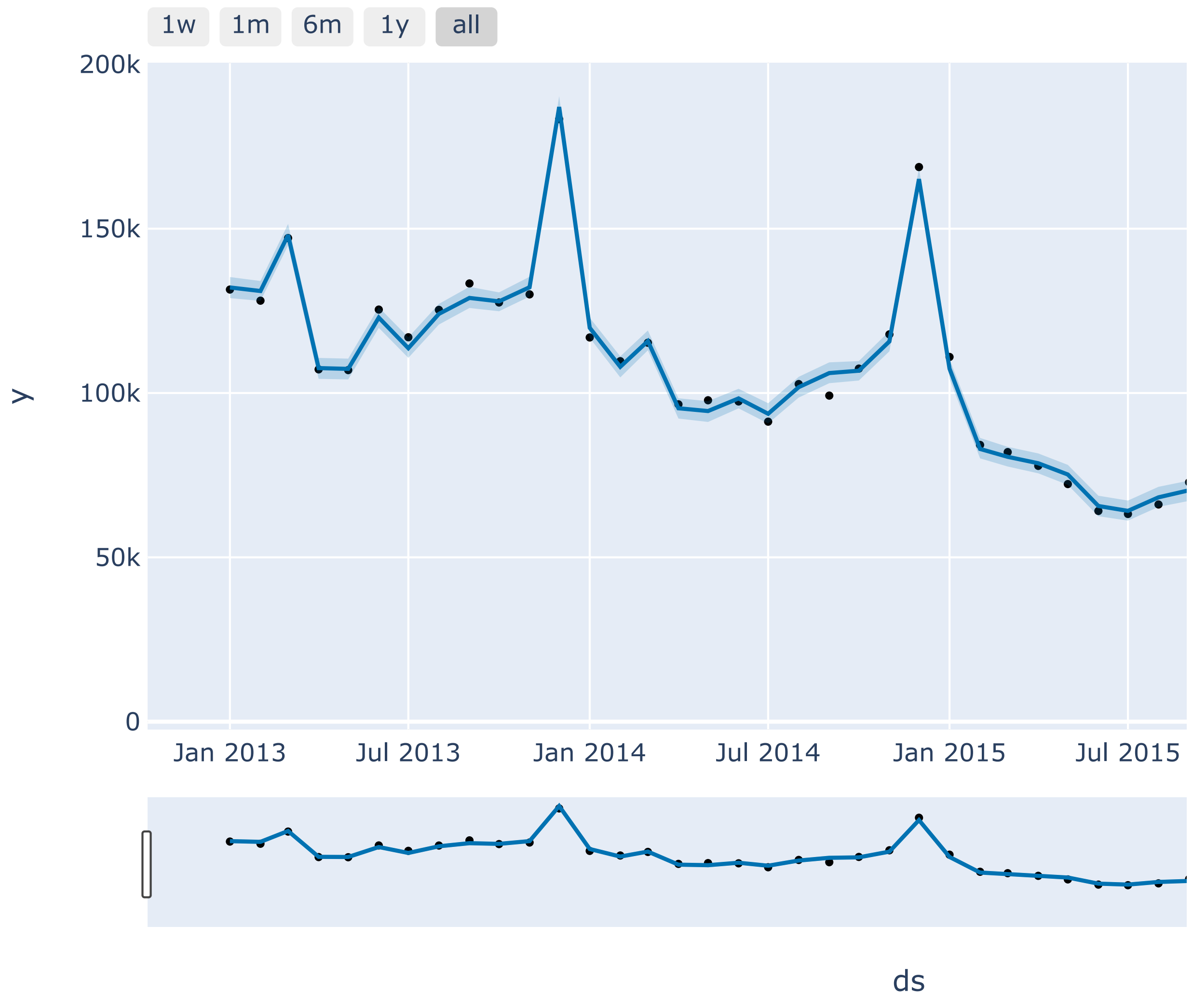
In [29]:
```
model.plot_components(forecast)
```

Out[29]:


trend
140000
120000
100000
80000
60000
40000
20000
2012-11
2013-05
2013-11
2014-05
2014-11
2015-05
2015-11
2016-05
2016-11
ds
yearly
400000
200000
0
−200000
January 1
March 1
May 1
July 1
September 1
November 1
January 1
Day of year


trend
140000
120000
100000
80000
60000
40000
20000
2012-11
2013-05
2013-11
2014-05
2014-11
2015-05
2015-11
2016-05
2016-11
ds
yearly
400000
200000
0
−200000
January 1
March 1
May 1
July 1
September 1
November 1
January 1
Day of year

In [30]:
```
# Python
from fbprophet.plot import plot_plotly
import plotly.offline as py
py.init_notebook_mode()

fig = plot_plotly(model, forecast)  # This returns a plotly Figure
py.iplot(fig)
```



Awesome. The trend and seasonality from Prophet look similar to the ones that we had earlier using the traditional methods.

Summary

I like Prophet more! What do you think?

In [31]:
```
from IPython.display import Image
Image(url='finished.gif')
```

Out[31]: