Reference

DataCamp course

Correlation and Autocorrelation

- Correlation describes the relationship between two time series and autocorrelation describes the relationship of a time series with its past values.
- Correlation (in the sense of normalized covariance) describes a linear relation about how two random variables come together or not. Therefore time-series prediction models based on correlation, autocorrelation, or regression etc. all belong to linear models.

Merging Time Series With Different Dates

- One way to see the dates that the stock market is open and the bond market is closed is to convert both indexes of dates into sets and take the difference in sets.
- The pandas .join() method is a convenient tool to merge the stock and bond DataFrames on dates when both markets are open.

In [ ]:
```
import pandas as pd
set_stock_dates = set(stocks.index)
set_bond_dates = set(bonds.index)
print(set_stock_dates - set_bond_dates)

# Merge stocks and bonds DataFrames using join()
stocks_and_bonds = stocks.join(bonds,how='inner')
#To get the intersection of dates, use the argument how='inner'
```

Correlation of Stocks and Bonds

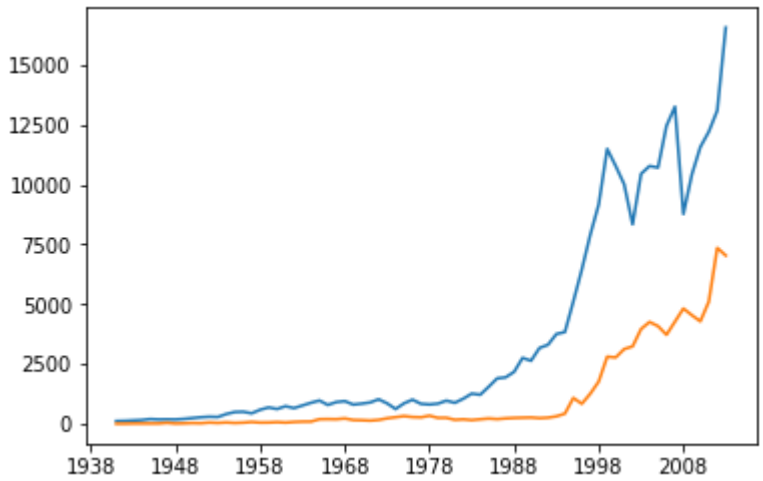
Scatter plots are also useful for visualizing the correlation between the two variables. Keep in mind that we should compute the **correlations on the percentage changes rather than the levels**. Otherwise, even almost no correlation, there will be a big correlation coefficient whenever there is a general similar trend.

In [ ]:
```
returns = stocks_and_bonds.pct_change()
correlation = returns['SP500'].corr(returns['US10Y'])
print("Correlation of stocks and interest rates: ", correlation)
plt.scatter(returns['SP500'], returns['US10Y'])
plt.show()
```

Flying Saucers Aren't Correlated to Flying Markets

Two trending series may show a strong correlation even if they are completely unrelated. This is referred to as **"spurious correlation"**. That's why when you look at the correlation of say, two stocks, you should look at the correlation of their returns (percent change) and not their levels.

In [6]:
```
import matplotlib.pyplot as plt
import pandas as pd
levels = pd.read_csv('levels.csv', header = None, index_col = 0, parse_dates=True)
levels.columns = ['DJI','UFO']
plt.plot(levels['DJI'])
plt.plot(levels['UFO'])
plt.show()
#my code above

correlation1 = levels['DJI'].corr(levels['UFO'])
print("Correlation of levels: ", correlation1)

# Compute correlation of percent changes
changes = levels.pct_change()
correlation2 = changes['DJI'].corr(changes['UFO'])
print("Correlation of changes: ", correlation2)
```




```
Correlation of levels:  0.9399762210726432
Correlation of changes:  0.06026935462405376
```

Looking at a Regression's R-Squared

R-squared measures how closely the data fit the regression line, so the R-squared in a simple regression is related to the correlation between the two variables. In particular, **the magnitude of the correlation is the square root of the R-squared and the sign of the correlation is the sign of the regression coefficient**.

Use the statistical package statsmodels, which performs much of the statistical modeling and testing that is found in R and software packages like SAS and MATLAB.

In the following example, we use the percentage change (returns) to do linear regression. So the first row is taken out from the original data. Sometimes we also use prices rather than levels to calculate the correlation, or linear regression, as when we predict prices of an asset.

In [ ]:
```
import statsmodels.api as sm
correlation = x.corr(y)
print("The correlation between x and y is %4.2f" %(correlation))

# Convert the Series x to a DataFrame and name the column x. So same data, different types.
x = pd.DataFrame(x, columns=['x'])
x = sm.add_constant(x) #Without this sentence, there will be no intercept.
result = sm.OLS(y,x).fit()
print(result.summary())
```

A Popular Strategy Using Autocorrelation

A more mathematical way to describe mean reversion is to say that stock returns are **negatively autocorrelated.** This simple idea is actually the basis for a popular hedge fund strategy. See more [https://www.quantopian.com/posts/enhancing-short-term-mean-reversion-strategies-1 (https://www.quantopian.com/posts/enhancing-short-term-mean-reversion-strategies-1)](https://www.quantopian.com/posts/enhancing-short-term-mean-reversion-strategies-1).

In [ ]:
```
MSFT = MSFT.resample(rule='W', how='last')
returns = MSFT.pct_change()
autocorrelation = returns['Adj Close'].autocorr()
print("The autocorrelation of weekly returns is %4.2f" %(autocorrelation))
#Negative autocorrelation indicates mean reversion. Otherwise, suggests trending.
```

Are Interest Rates Autocorrelated?

The autocorrelation of daily changes in interest rates is close to zero. However, the annual counterpart is negative. This makes some economic sense: over long horizons, when interest rates go up, the economy tends to slow down, which consequently causes interest rates to fall, and vice versa.

In [ ]:
```
daily_data['change_rates'] = daily_data.diff()
autocorrelation_daily = daily_data['change_rates'].autocorr()
print("The autocorrelation of daily interest rate changes is %4.2f" %(autocorrelation_daily)) #output is 0.07

# Convert the daily data to annual data
annual_data = daily_data['US10Y'].resample(rule='W').last()
annual_data['diff_rates'] = annual_data.diff()
autocorrelation_annual = annual_data['diff_rates'].autocorr()
print("The autocorrelation of annual interest rate changes is %4.2f" %(autocorrelation_annual)) #output is -0.22
```

Some Simple Time Series

Some simple time series models: **white noise and a random walk. What is their difference?**

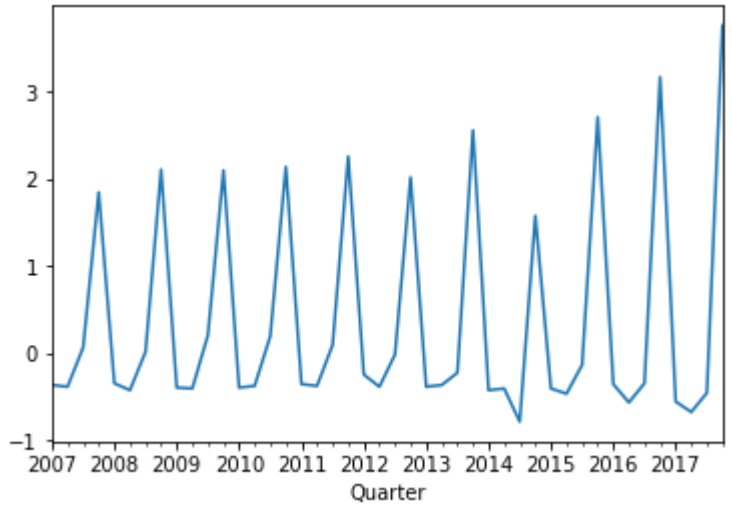
https://www.researchgate.net/post/How_should_I_understand_the_difference_between_the_Random_walk_and_the_Red_noise (https://www.researchgate.net/post/How_should_I_understand_the_difference_between_the_Random_walk_and_the_Red_noise)
Random walks and noises are very different stochastic processes. White noise have values that are independent: the value of the noise at time t is a random variable that is independent of the value at time s, provided t and s are not equal. In fact this is an approximation because the notion of value here is not well defined, but this is the idea. A random walk is a **Markov process**, and the most classical is the Brownian motion. This means that the values at times t and s are not independent. But **the increases are independent**. To make it simple, a **Brownian motion is the integral of white noise**, and more generally **random walks are the integral of some "noise"**. Since this integral defines the random walk, there is a relationship (integration/derivation) between "noises" and random walks, but they really differ. E.g. a random walk is continuous while a noise is discontinuous.

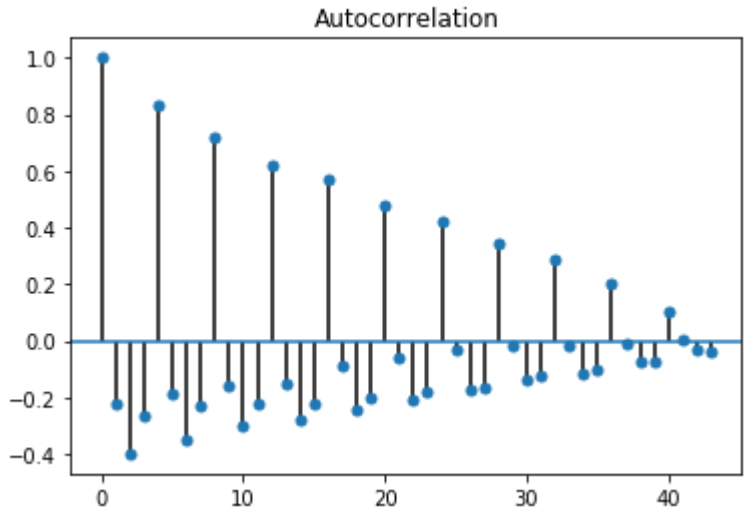
See comparison between white noise and random walk in later cells.

Taxing Exercise: Compute the ACF (auto-correlation function)

- ACF allow us to see autocorrelation over many lags.
- The quarterly earnings for H&R Block shows extreme cyclicality of its earnings, due to quartly tax.
- Shows what the autocorrelation function looks like for cyclical earnings data.
- The ACF at lag=0 is always one. The confidence interval for the ACF is not shown by setting alpha=1.

In [2]:
```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf

HRB = pd.read_csv('data/HRB.csv',index_col='Quarter', parse_dates=True)
# Note the index column with names 2007Q1, 2007Q2, ... will be automatically changed to
# datatime type 2007-01-01, 2007-04-01
HRB['Earnings'].plot()

acf_array = acf(HRB)

# Plot the acf function
plot_acf(HRB, alpha=1)
plt.show()
```





Notice the strong positive autocorrelation at lags 4, 8, 12, 16,20, ...

Are We Confident This Stock is Mean Reverting?

We saw that the autocorrelation of MSFT's weekly stock returns was -0.16. That autocorrelation seems large, but is it statistically significant? **In other words, can you say that there is less than a 5% chance that we would observe such a large negative autocorrelation if the true autocorrelation were really zero? And are there any autocorrelations at other lags that are significantly different from zero?**

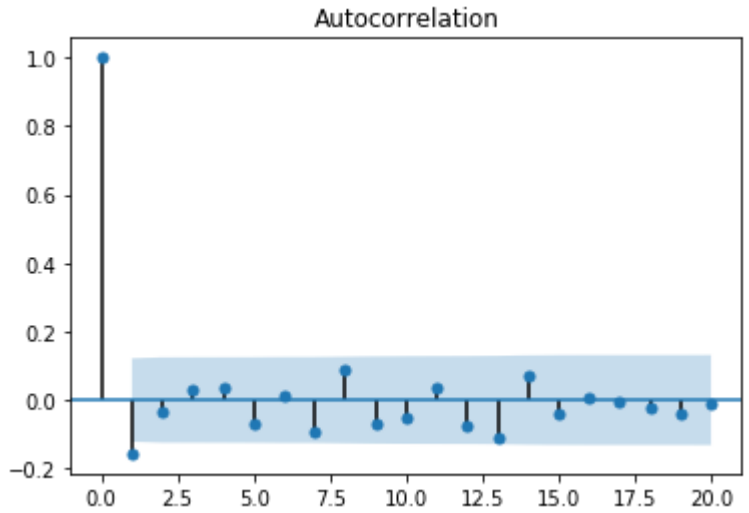
Even if the true autocorrelations were zero at all lags, in a finite sample of returns you won't see the estimate of the autocorrelations exactly zero. In fact, the standard deviation of the sample autocorrelation is $\frac{1}{\sqrt{N}}$ (**Need figure out this. Should be related to the STD of sampling distribution, or SE.**), where N is the number of observations, so if $N = 100$, for example, the standard deviation of the ACF is 0.1, and since 95% of a normal curve is between +1.96 and -1.96 standard deviations from the mean, the 95% confidence interval is $\pm\frac{1.96}{\sqrt{N}}$. This approximation only holds when the true autocorrelations are all zero.

Compute the actual and approximate confidence interval for the ACF, and compare it to the lag-one autocorrelation of -0.16 from the last chapter.

In [8]:

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf
from math import sqrt

MSFT = pd.read_csv('data/MSFT.csv',index_col='Date', parse_dates=True)
# Note the index column with names 2007Q1, 2007Q2, ... will be automatically changed to datatime type 2007-01-01,
# print(MSFT.head())
MSFT = MSFT.resample(rule='W').last()

returns = MSFT.pct_change()

returns = returns.drop(pd.Timestamp('2012-08-12'))
# returns = returns.dropna() #In this particular case, I can also use dropna().

autocorrelation = returns['Adj Close'].autocorr()

nobs = len(returns)
conf = 1.96/sqrt(nobs)
print("The approximate confidence interval is +/- %4.2f" %(conf))

# Plot the autocorrelation function with 95% confidence intervals and 20 lags using plot_acf
plot_acf(returns, alpha=0.05, lags=20)
# 5% chance that if true autocorrelation is zero, it will fall outside blue band. Significance level
# is usually defined beyond 95% Confidence interval.
plt.show()
```

```
-0.15681320218088093
The approximate confidence interval is +/- 0.12
```



Notice that the autocorrelation with lag 1 is significantly negative, but none of the other lags are significantly different from zero

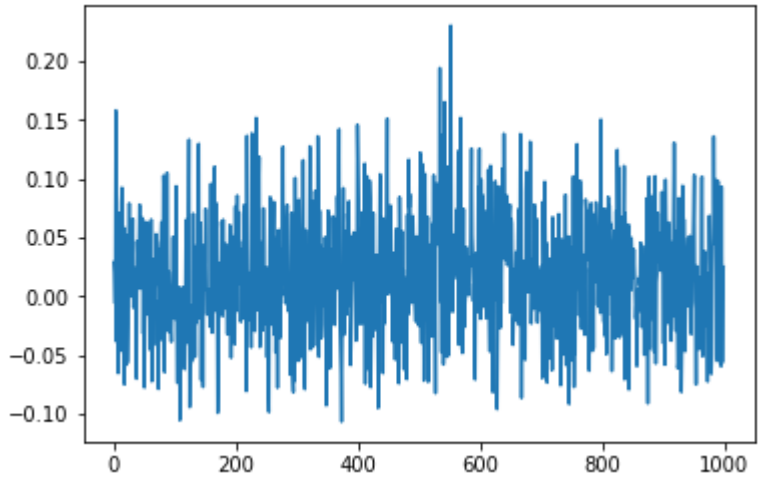
There is no lags in autocorrelation of S&P 500 that is significantly from zero (either negative or positive). This indicates there are no either trending, or mean reversion, right?

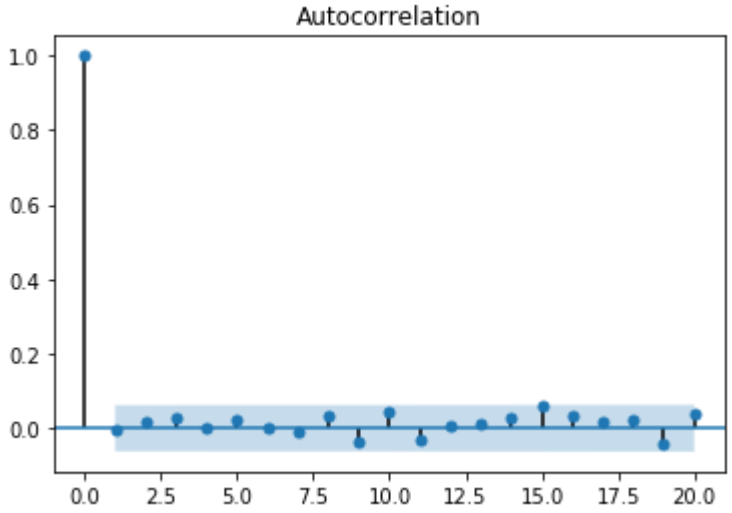
Can't Forecast White Noise

See comparison of white noise and random walk later.
Comment. Do not confuse white noise with the flat spectral line. As shown below, **it can be generated with Gaussian distribution. The key values at different time points are independent**

In [5]:
```
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf

returns = np.random.normal(loc=0.02, scale=0.05, size=1000) #generate random returns.

mean = np.mean(returns)
std = np.std(returns)
print("The mean is %5.3f and the standard deviation is %5.3f" %(mean,std))

plt.plot(returns)
plt.show()

# Plot autocorrelation function of white noise returns
plot_acf(returns, lags=20)
plt.show()
```

```
The mean is 0.019 and the standard deviation is 0.051
```





Generate a Random Walk

About of random walk

- Random variable X is the sum of many other random variables $X_i, i = 1, 2, \ldots$. If each X_i is taken as a single-step walk, then X can describe a random walk.
- Because X is a random variable, the processes it describe are random and thus we **cannot expect return if the price of a stock follows a random walk.**
- Each random walk is a realization of the random variable X. This also helps understand the stochastic process. That is, each process is similar to a random walk, and we can have many processes.
- The end values of random walks form a normal distribution, if random variable X is a sum of too many random variables X_i. This is from central limit theorem. **This indicates, when we perform 1000 random walks (each random walk is from e.g. 500 steps), then the 1000 end values are random. This is why we cannot expect return from random walk.**

Whereas **stock returns** are often modeled **as white noise**, **stock prices** closely follow **a random walk** (see the downloaded slide). In other words, today's price is yesterday's price plus some random noise.

Ways of random walk generation

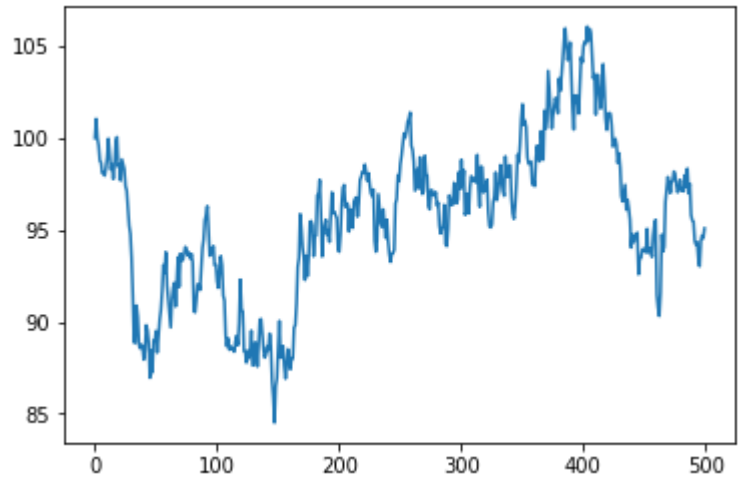
- See random walk generation and simple time series analysis in the notes 'time series manipulation' elsewhere.
- See statistical thinking notes elsewhere for stairs climbing.

Compare the random walk and white noise

- Autocorrelation at all lags are zero for white noise. How about random walk?
- Stock returns are often modeled as white noise. Unfortunately, for white noise, we cannot forecast future observations based on the past due to the zero autocorrelations at all lags.
- By taking first differences of a random walk, you get a stationary white noise process (Always for any lags?)

In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
# Generate 500 random steps with mean=0 and standard deviation=1
steps = np.random.normal(loc=0, scale=1.0, size=500)

# Set first element to 0 so that the first price will be the starting stock price
steps[0]=0
# Simulate stock prices, P with a starting price of 100
P = 100 + np.cumsum(steps)
# See other places, can also use percentage and cumprod to generate random walk.
plt.plot(P)
```

Out[5]:
```
[<matplotlib.lines.Line2D at 0x1e3e6785c50>]
```

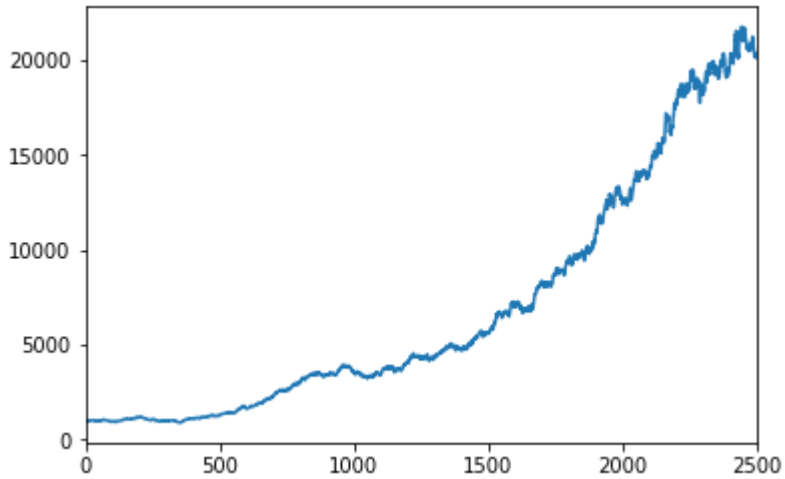


The following code is from another course:

Random walk I

See explanation to random walk in the notes for 'introduction to time series analysis in Python'.

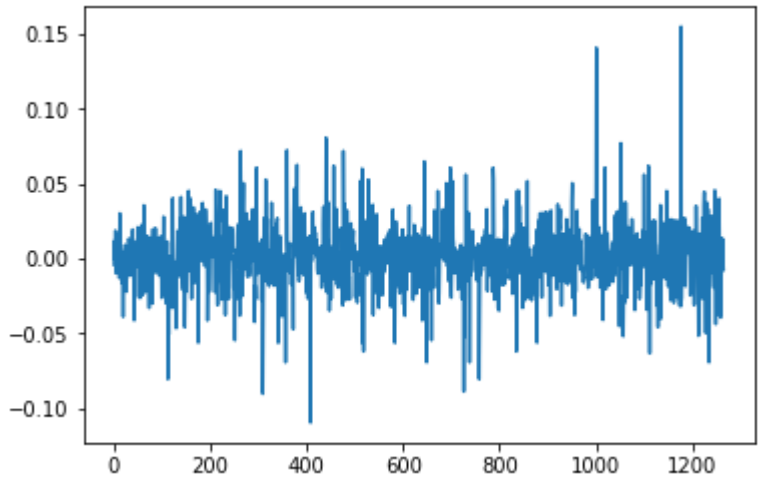
In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
seed = 42

random_walk = np.random.normal(loc=.001, scale=0.01, size=2500)
#loc is mu, scale is variance.
random_walk = pd.Series(random_walk)
random_prices = random_walk.add(1).cumprod()
#random walk sometimes uses np.cumsum() or .add(1).cumprod(). See notes elsewhere.

random_prices.mul(1000).plot()
plt.show();
```

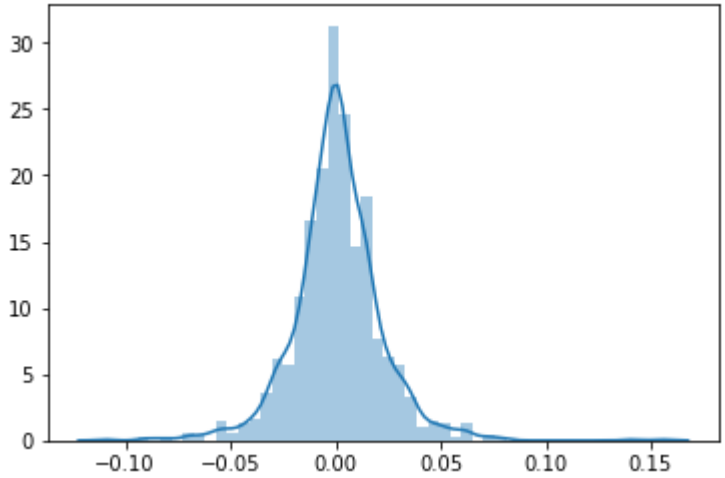


Random walk II

Build a random walk using historical **returns** from Facebook's stock price since IPO through the end of May 31, 2017.

In [8]:
```
import seaborn as sns
import pandas as pd
import numpy as np
fb = pd.read_csv('fb.csv', index_col = 'date', parse_dates = True)
seed = 42

daily_returns = fb.pct_change().dropna()

daily_returns = np.array(daily_returns['close'])
n_obs = len(daily_returns)

random_walk = np.random.choice(daily_returns, size=n_obs)

plt.plot(random_walk) #As plot show, this is not random_walk yet. In fact, returns are often modeled by white noi
plt.show()

# Convert random_walk to pd.series
random_walk = pd.Series(random_walk)

# Plot random_walk distribution
sns.distplot(random_walk)
# Understand the fundamentals of why a random walk can be described by an approximate normal distribution.
# The randoms value of each step of random walk, or the end-value of many random walks both follow
# approximate normal distribution.
plt.show()
```





Random walk III

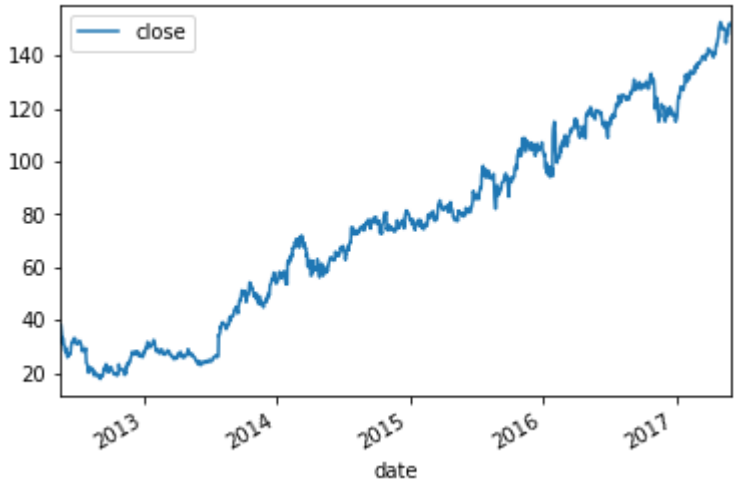
Start off with a random sample of returns like the one in the last exercise and use it to create a random stock price path. Then compare the simulated stock price with the original price.

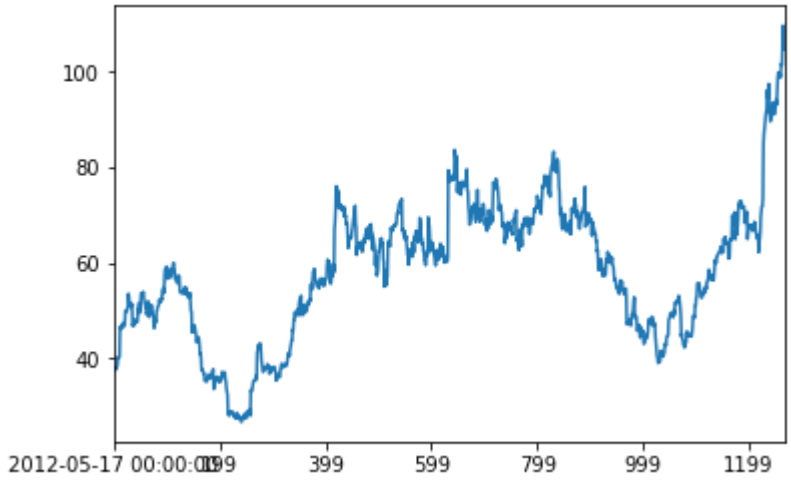
In [9]:

```
import seaborn as sns
import pandas as pd
import numpy as np
fb = pd.read_csv('fb.csv', index_col = 'date', parse_dates = True)
seed = 42
daily_returns = fb.pct_change().dropna()
daily_returns = np.array(daily_returns['close'])
n_obs = len(daily_returns)
random_walk = np.random.choice(daily_returns, size=n_obs)
random_walk = pd.Series(random_walk)
print(len(fb),len(random_walk)) #check how many Nan dropped
print(random_walk.head())
# Code copied from previous cell.

start = fb.close.first('D')
random_walk = random_walk.add(1)
random_price = start.append(random_walk)

random_price = random_price.cumprod()
print((random_price.head()))

fb.plot()
plt.show()

random_price.plot()
plt.show()

#May combine the two set of data into one DataFrame, and then plot. The following code need modify.
# fb['random'] = random_price
# print(fb.head())
# fb.plot()
# plt.show()
```

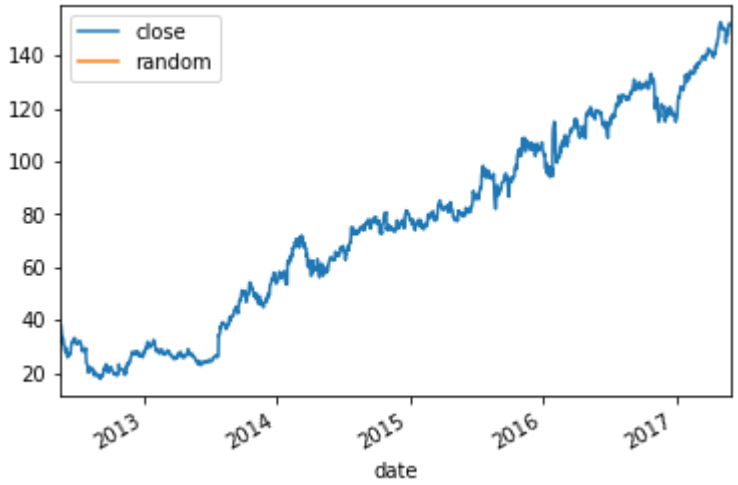
```
1267 1266
0   -0.004843
1    0.000152
2    0.002556
3   -0.007000
4    0.014615
dtype: float64
2012-05-17 00:00:00    38.000000
0                      37.815981
```

```
1                       37.821730
2                       37.918420
3                       37.652991
dtype: float64
```






```
            close  random
date
2012-05-17  38.00    38.0
2012-05-18  38.23     NaN
2012-05-21  34.03     NaN
2012-05-22  31.00     NaN
2012-05-23  32.00     NaN
```



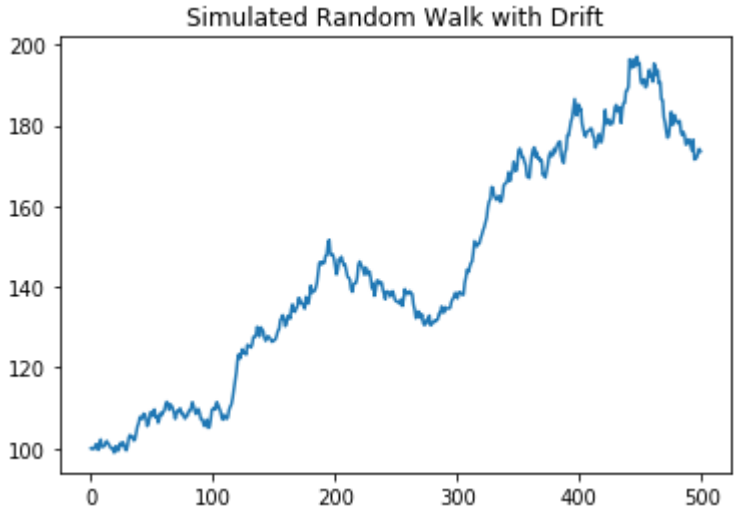
Get the Drift

In the last exercise, stock prices is simulated with a random walk. Now we extend this in two ways.

- Add a drift to random walk because stock price tend to drift up over time.
- Make the noise multiplicative rather than additive, because additive might give negative prices. So in simulating stock prices, using cumprod is better than cumsum().

For stock prices, random walk + cumprod + drift is better than random walk only.

In [7]:
```
steps = np.random.normal(loc=0.001, scale=0.01, size=500) + 1
steps[0]=1
P = 100 * np.cumprod(steps)
#Often we use percentage returns to calculate random walk.

plt.plot(P)
plt.title("Simulated Random Walk with Drift")
plt.show()
```



Are Stock Prices a Random Walk?

Most stock prices follow a random walk (perhaps with a drift). Use the 'Augmented Dickey-Fuller Test' from the statsmodels library to show that it does indeed follow a random walk. With the ADF test, the "null hypothesis" is that the series follows a random walk. Therefore, a low p-value (say less than 5%) means we can reject the null hypothesis that the series is a random walk.

As mentioned earlier, we cannot expect return if the price of a stock follows a random walk. So this section we cannot expect return for most stocks? How to reconcile this with reality?

In [12]:
```
# Import the adfuller module from statsmodels
from statsmodels.tsa.stattools import adfuller
import pandas as pd
AMZN = pd.read_csv('data/AMZN.csv',index_col='Date', parse_dates=True)

results = adfuller(AMZN['Adj Close'])
print(results)

print('The p-value of the test on prices is: ' + str(results[1]))
```

```
(4.025168525770744, 1.0, 33, 5054, {'1%': -3.4316445438146865, '5%': -2.862112049726916, '10%': -2.567074502532
1304}, 30308.64216426981)
The p-value of the test on prices is: 1.0
```

According to this test, we cannot reject the hypothesis that Amazon prices follow a random walk. In the next exercise, you'll look at Amazon returns.

How About Stock Returns?

Previous exercise showed that Amazon stock prices, contained in the DataFrame AMZN follow a random walk. Now we do the same thing for Amazon returns (percent change in prices) and show that the returns do not follow a random walk.

In [ ]:
```
from statsmodels.tsa.stattools import adfuller
AMZN_ret = AMZN.pct_change()
AMZN_ret = AMZN_ret.dropna()
print('The p-value of the test on returns is: ' + str(results[1]))
```

The p-value is extremely small, so we can easily reject the hypothesis that returns are a random walk at all levels of significance.

What is Stationarity?

- Strong stationarity: entire distribution of data is time-invariant.
- Weak stationarity: mean, variance and autocorrelation are time-invariant (i.e., for autocorrelation, $corr(X_t, X_{t-\tau})$ is only a function of τ.

Seasonal Adjustment During Tax Season

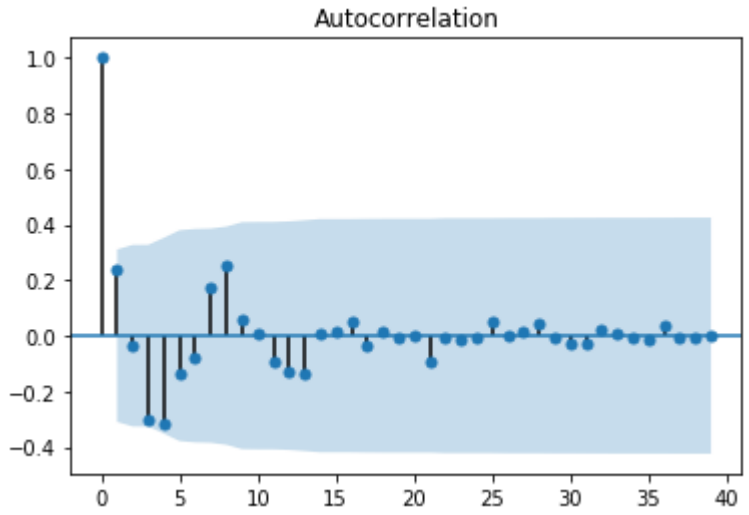
Many time series exhibit strong seasonal behavior. The procedure for removing the seasonal component of a time series is called seasonal adjustment. Most economic data published by the government is seasonally adjusted.

You saw earlier that **by taking first differences of a random walk, you get a stationary white noise process**. For seasonal adjustments, instead of taking first differences, we take differences with a lag corresponding to the periodicity.

Apply a seasonal adjustment by **taking the fourth difference** (four represents the periodicity of the series). Then compute the autocorrelation of the transformed series.

In [20]:
```
from statsmodels.graphics.tsaplots import plot_acf
import pandas as pd
HRB = pd.read_csv('data/HRB.csv',index_col='Quarter', parse_dates=True)
#print(HRB.head())
#print(len(HRB))
# Seasonally adjust quarterly earnings
HRBsa = HRB.diff(4)

# Print the first 10 rows of the seasonally adjusted series
print(HRBsa.head(10))

# Drop the NaN data in the first three three rows
HRBsa = HRBsa.dropna()
# Plot the autocorrelation function of the seasonally adjusted series
plot_acf(HRBsa)
plt.show()
```

```
            Earnings
Quarter
2007-01-01       NaN
2007-04-01       NaN
2007-07-01       NaN
2007-10-01       NaN
2008-01-01      0.02
2008-04-01     -0.04
2008-07-01     -0.05
2008-10-01      0.26
2009-01-01     -0.05
2009-04-01      0.02
```



By seasonally adjusting the series, we eliminated the seasonal pattern in the autocorrelation function. see plots in earlier cells.

Autoregressive (AR) Models

These models use past values of the series to predict the current value.
Comments:

- Compare AR to ACF.
- Compare AR to linear regression.

Side note:

- A regression model, such as linear regression, models an output value based on a linear combination of input values. ... Because the regression model uses data from the same input variable at previous time steps, it is referred to as an autoregression (regression of self).
- The definition of $AR(p)$

$$X_t = c + \sum_{i=1}^{p} \phi_i X_{t-i} + \epsilon_t$$

  where ϕ_i are the parameters of the model, c is a constant, and ϵ_t is white noise.
- Autoregressive concepts are used by technical analysts to forecast securities prices. For instance, trends, moving averages, and regressions take into account past prices in an effort to create forecasts of future price movement. The key difference is that many technical indicators try to capture the complex nonlinearity of financial prices to maximize profits, while autoregressive models strictly

seek to minimize the mean squared error and may yield more accurate forecasts for linear underlying processes.

- The following comments need be verified later.
  - From above, we know that AR models are linear models.
  - Among other ways to determine the parameters ϕ, it should be possible to determine these parameters by machine learning techniques. It is easy to generate many feature-target samples for parameter estimate. This helps to connect AR models with machine learning.
  - **Either the AR here or the MA models introduced later are both the problems of statistical estimation. In the linear region, the AR and MA, or the combined ARMA models should be equivalent to the linear machine learning algorithms.**

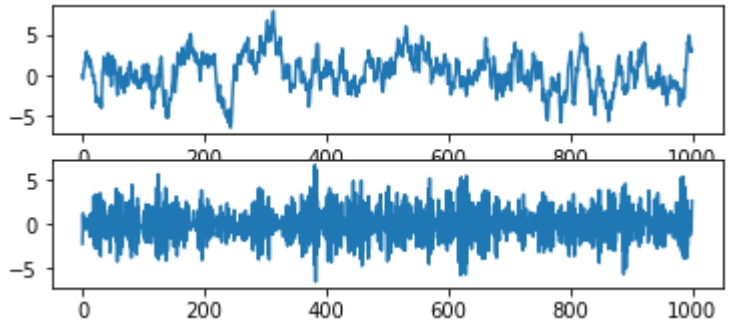
Simulate AR(1) Time Series

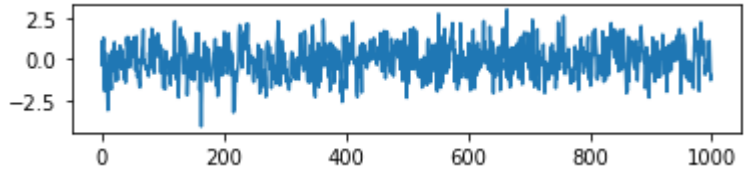
Simulate and plot a few AR(1) time series, each with a different parameter, ϕ ,a large positive ϕ and a large negative ϕ.

There are a few conventions when using the arima_process module that require some explanation.

- First, these routines were made very generally to handle both **AR and MA models (See more details on the notes for time series machine learning)**.
- Second, when inputting the coefficients, you must include the zero-lag coefficient of 1, and the sign of the other coefficients is opposite what we have been using (to be consistent with the time series literature in signal processing). For example, for an AR(1) process with $\phi = 0.9$, the array representing the AR parameters would be ar = np.array([1, -0.9])

In [23]:
```
from statsmodels.tsa.arima_process import ArmaProcess

# Plot 1: AR parameter = +0.9
plt.subplot(3,1,1)
ar1 = np.array([1, -0.9])
ma1 = np.array([1])
AR_object1 = ArmaProcess(ar1, ma1)
simulated_data_1 = AR_object1.generate_sample(nsample=1000)
plt.plot(simulated_data_1)

# Plot 2: AR parameter = -0.9
plt.subplot(3,1,2)
ar2 = np.array([1, 0.9])
ma2 = np.array([1])
AR_object2 = ArmaProcess(ar2, ma2)
simulated_data_2 = AR_object2.generate_sample(nsample=1000)
plt.plot(simulated_data_2)
plt.show()

# Plot 2: AR parameter = 0.3
plt.subplot(3,1,3)
ar3 = np.array([1, -0.3])
ma3 = np.array([1])
AR_object3 = ArmaProcess(ar3, ma3)
simulated_data_3 = AR_object3.generate_sample(nsample=1000)
plt.plot(simulated_data_3)
plt.show()
```

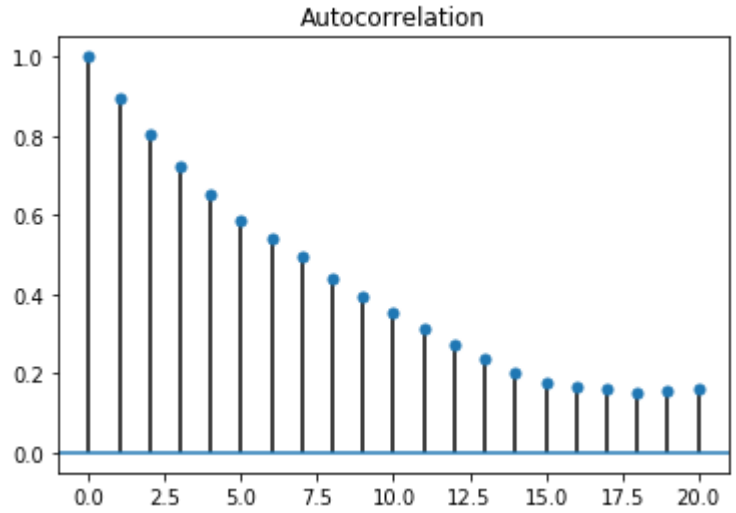


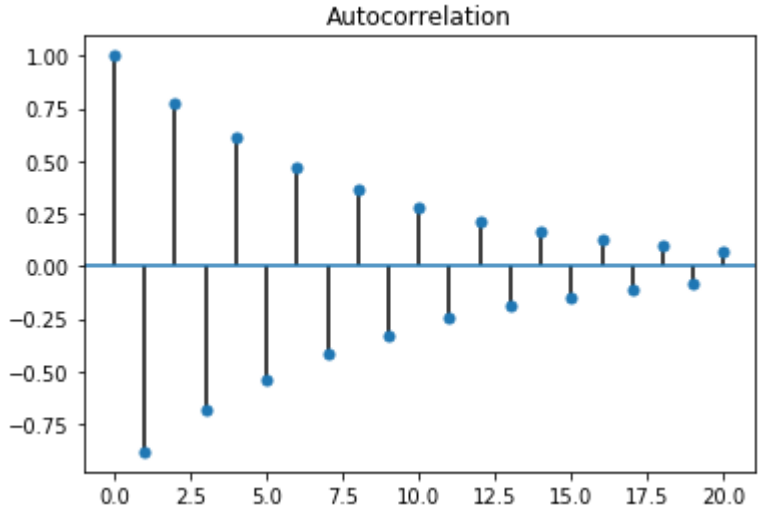


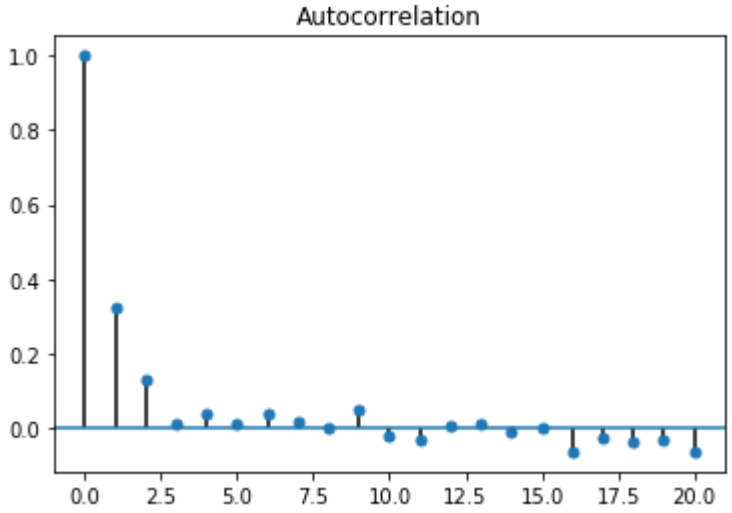
Compare the ACF for Several AR Time Series

The autocorrelation function decays exponentially for an AR time series at a rate of the AR parameter. For example, if the AR parameter, $\phi = 0.9$, the first-lag autocorrelation will be 0.9, the second-lag will be $0.9^2 = 0.81$, the third-lag will be $0.9^3 = 0.729$, etc. A smaller AR parameter will have a steeper decay, and for a negative AR parameter, say -0.9, the decay will flip signs, so the first-lag autocorrelation will be -0.9, the second-lag will be $(-0.9)^2 = 0.81$, the third-lag will be $(-0.9)^3 = -0.729$, etc.

In [24]:
```
# Import the plot_acf module from statsmodels
from statsmodels.graphics.tsaplots import plot_acf

# Plot 1: AR parameter = +0.9
# Data are generated in previous cells.
plot_acf(simulated_data_1, alpha=1, lags=20)
plt.show()

# Plot 2: AR parameter = -0.9
plot_acf(simulated_data_2, alpha=1, lags=20)
plt.show()

# Plot 3: AR parameter = +0.3
plot_acf(simulated_data_3, alpha=1, lags=20)
plt.show()
```







Estimating an AR Model

You will estimate the AR(1) parameter, ϕ, of one of the simulated series that you generated in the earlier exercise. Since the parameters are known for a simulated series, it is a good way to understand the estimation routines before applying it to real data.

For simulated_data_1 with a true ϕ of 0.9, you will print out the estimate of ϕ. In addition, you will also print out the entire output that is produced when you fit a time series, so you can get an idea of what other tests and summary statistics are available in statsmodels.

In [25]:
```
# Import the ARMA module from statsmodels
from statsmodels.tsa.arima_model import ARMA

# Fit an AR(1) model to the first simulated data
mod = ARMA(simulated_data_1, order=(1,0))
res = mod.fit()

# Print out summary information on the fit
print(res.summary())

# Print out the estimate for the constant and for phi
print("When the true phi=0.9, the estimate of phi (and the constant) are:")
print(res.params)
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
```

ARMA Model Results

Dep. Variable:	y	No. Observations:	1000
Model:	ARMA(1, 0)	Log Likelihood	-1409.485
Method:	css-mle	S.D. of innovations	0.990
Date:	Thu, 03 Jan 2019	AIC	2824.970
Time:	17:22:19	BIC	2839.693
Sample:	0	HQIC	2830.566

	coef	std err	z	P>|z|	[0.025	0.975]
const	0.2937	0.303	0.971	0.332	-0.299	0.887
ar.L1.y	0.8974	0.014	64.641	0.000	0.870	0.925

Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.1143	+0.0000j	1.1143	0.0000

```
--------------------------------------------------------------------------------
When the true phi=0.9, the estimate of phi (and the constant) are:
[0.29365004 0.89743941]
```

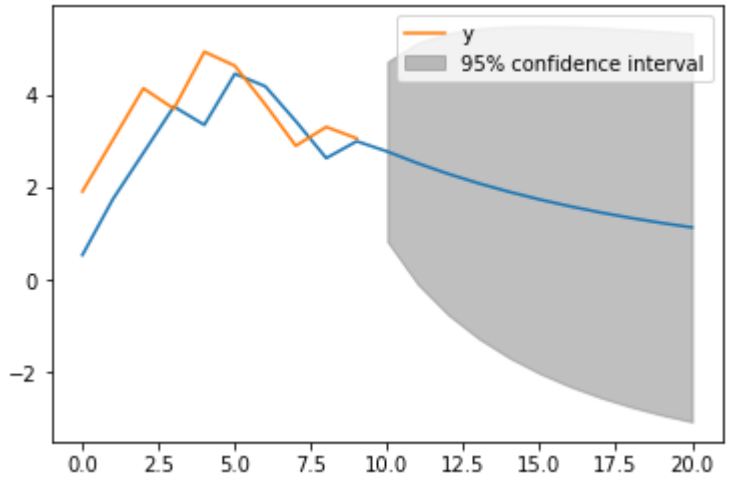
Forecasting with an AR Model

In addition to estimating the parameters of a model that you did in the last exercise, you can also do forecasting, both in-sample and out-of-sample using statsmodels. The in-sample is a forecast of the next data point using the data up to that point, and the out-of-sample forecasts any number of data points in the future. These forecasts can be made using either the predict() method if you want the forecasts in the form of a series of data, or using the plot_predict() method if you want a plot of the forecasted data. You supply the starting point for forecasting and the ending point, which can be any number of data points after the data set ends.

For the simulated series simulated_data_1 with ϕ=0.9, you will plot in-sample and out-of-sample forecasts.

In [26]:
```
# Import the ARMA module from statsmodels
from statsmodels.tsa.arima_model import ARMA

# Forecast the first AR(1) model
mod = ARMA(simulated_data_1, order=(1,0))
res = mod.fit()
res.plot_predict(start=990, end=1010)
plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:577: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
```



Notice how, when phi is high like here, the forecast gradually moves to the long term mean of zero, but if phi were low, it would move much quicker to the long term mean. Try it out and see for yourself!

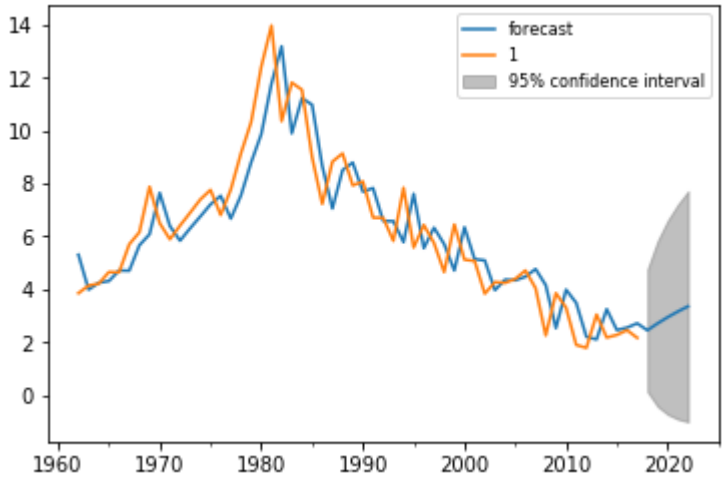
Let's Forecast Interest Rates

You will now use the forecasting techniques you learned in the last exercise and apply it to real data rather than simulated data. You will revisit a dataset from the first chapter: the annual data of 10-year interest rates going back 56 years, which is in a Series called interest_rate_data. Being able to forecast interest rates is of enormous importance, not only for bond investors but also for individuals like new homeowners who must decide between fixed and floating rate mortgages.

You saw in the first chapter that there is some mean reversion in interest rates over long horizons. In other words, when interest rates are high, they tend to drop and when they are low, they tend to rise over time. Currently they are below long-term rates, so they are expected to rise, but an AR model attempts to quantify how much they are expected to rise.

In [41]:
```
import pandas as pd
from statsmodels.tsa.arima_model import ARMA
interest_rate_data = pd.read_csv('data/interestRate.txt',sep = '\s+', index_col=0, header = None)
interest_rate_data.index = pd.to_datetime(interest_rate_data.index)
interest_rate_data

# Forecast interest rates using an AR(1) model
mod = ARMA(interest_rate_data, order=(1,0))
res = mod.fit()

# Plot the original series and the forecasted series
res.plot_predict(start=0, end='2022')
plt.legend(fontsize=8)
plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:577: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
```



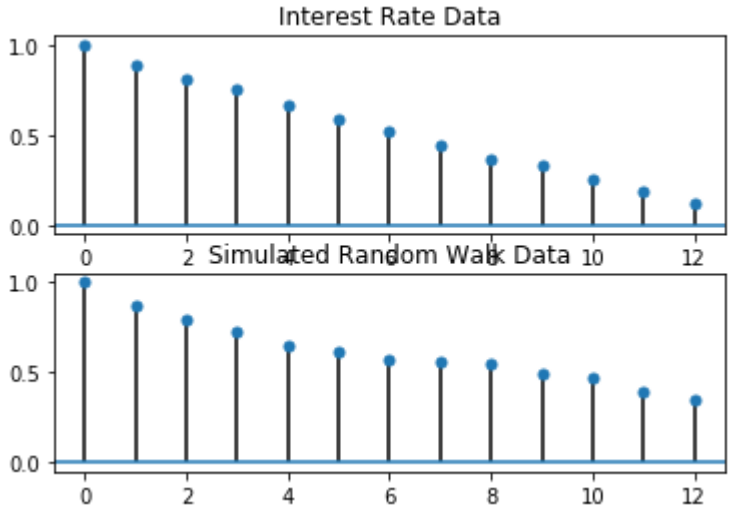
Compare AR Model with Random Walk

Sometimes it is difficult to distinguish between a time series that is slightly mean reverting and a time series that does not mean revert at all, like a random walk. You will compare the ACF for the slightly mean-reverting interest rate series of the last exercise with a simulated random walk with the same number of observations.

You should notice when plotting the autocorrelation of these two series side-by-side that they look very similar.

In [43]:

```
simulated_data = np.array([ 5.        ,  4.77522278,  5.60354317,  5.96406402,  5.97965372,
        6.02771876,  5.5470751 ,  5.19867084,  5.01867859,  5.50452928,
        5.89293842,  4.6220103 ,  5.06137835,  5.33377592,  5.09333293,
        5.37389022,  4.9657092 ,  5.57339283,  5.48431854,  4.68588587,
        5.25218625,  4.34800798,  4.34544412,  4.72362568,  4.12582912,
        3.54622069,  3.43999885,  3.77116252,  3.81727011,  4.35256176,
        4.13664247,  3.8745768 ,  4.01630403,  3.71276593,  3.55672457,
        3.07062647,  3.45264414,  3.28123729,  3.39193866,  3.02947806,
        3.88707349,  4.28776889,  3.47360734,  3.33260631,  3.09729579,
        2.94652178,  3.50079273,  3.61020341,  4.23021143,  3.94289347,
        3.58422345,  3.18253962,  3.26132564,  3.19777388,  3.43527681,
        3.37204482])

# Import the plot_acf module from statsmodels
from statsmodels.graphics.tsaplots import plot_acf

# Plot the interest rate series and the simulated random walk series side-by-side
fig, axes = plt.subplots(2,1)

# Plot the autocorrelation of the interest rate series in the top plot
fig = plot_acf(interest_rate_data, alpha=1, lags=12, ax=axes[0])

# Plot the autocorrelation of the simulated random walk series in the bottom plot
fig = plot_acf(simulated_data, alpha=1, lags=12, ax=axes[1])

# Label axes
axes[0].set_title("Interest Rate Data")
axes[1].set_title("Simulated Random Walk Data")
plt.show()
```

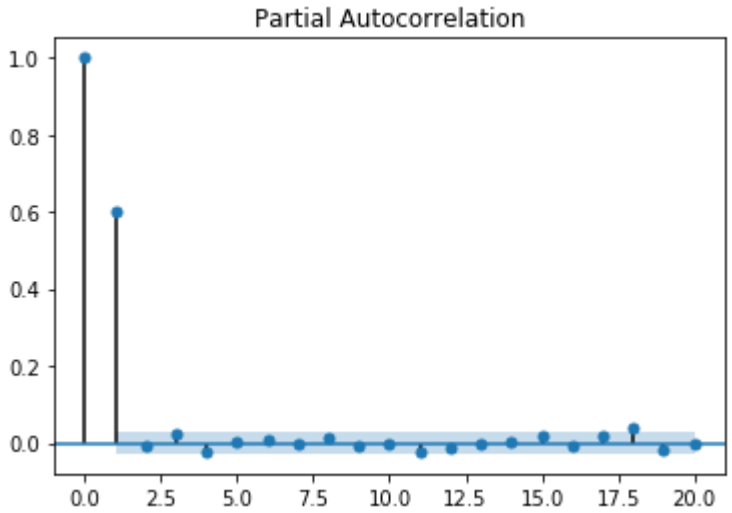


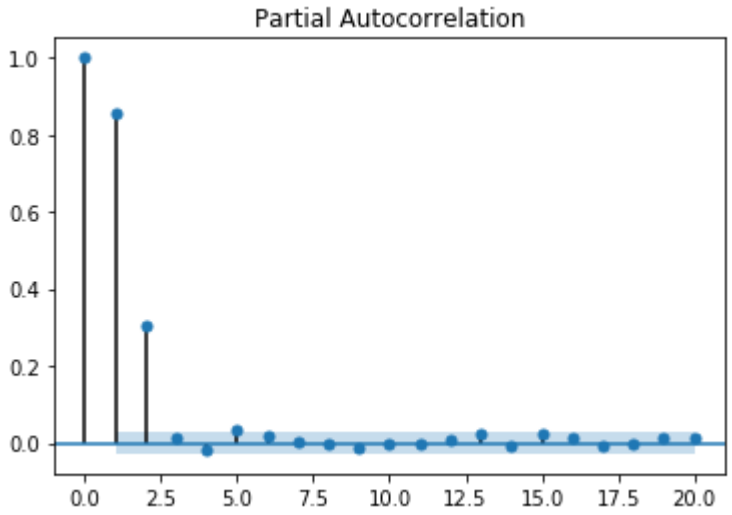
Estimate Order of Model: PACF

One useful tool to identify the order of an AR model is to look at the Partial Autocorrelation Function (PACF). In this exercise, you will simulate two time series, an AR(1) and an AR(2), and calculate the sample PACF for each. You will notice that for an AR(1), the PACF should have a significant lag-1 value, and roughly zeros after that. And for an AR(2), the sample PACF should have significant lag-1 and lag-2 values, and zeros after that.

Just like you used the plot_acf function in earlier exercises, here you will use a function called plot_pacf in the statsmodels module.

In [44]:
```
# Import the modules for simulating data and for plotting the PACF
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.graphics.tsaplots import plot_pacf

# Simulate AR(1) with phi=+0.6
ma = np.array([1])
ar = np.array([1, -0.6])
AR_object = ArmaProcess(ar, ma)
simulated_data_1 = AR_object.generate_sample(nsample=5000)

# Plot PACF for AR(1)
plot_pacf(simulated_data_1, lags=20)
plt.show()

# Simulate AR(2) with phi1=+0.6, phi2=+0.3
ma = np.array([1])
ar = np.array([1, -0.6, -0.3])
AR_object = ArmaProcess(ar, ma)
simulated_data_2 = AR_object.generate_sample(nsample=5000)


# Plot PACF for AR(2)
plot_pacf(simulated_data_2, lags=20)
plt.show()
```





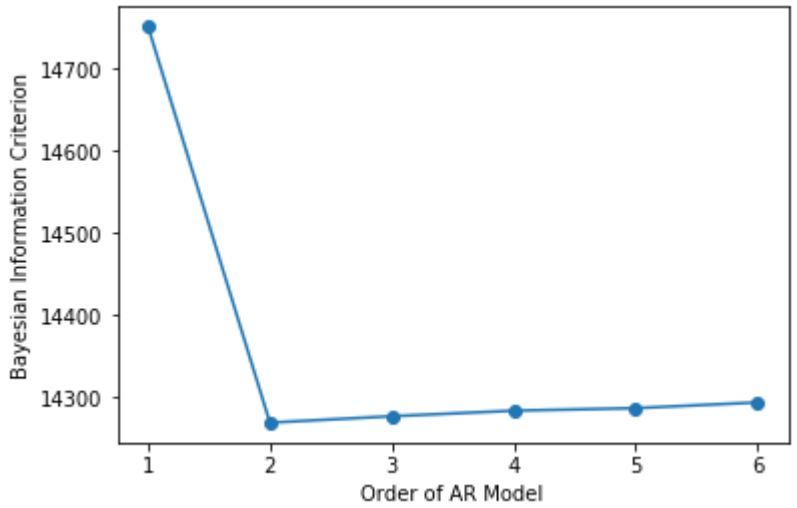
Notice that the number of significant lags for the PACF indicate the order of the AR model.

Estimate Order of Model: Information Criteria

Another tool to identify the order of a model is to look at the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC). These measures compute the goodness of fit with the estimated parameters, but apply a penalty function on the number of parameters in the model. You will take the AR(2) simulated data from the last exercise, saved as simulated_data_2, and compute the BIC as you vary the order, p, in an AR(p) from 0 to 6.

In [45]:
```
# Import the module for estimating an ARMA model
from statsmodels.tsa.arima_model import ARMA

# Fit the data to an AR(p) for p = 0,...,6 , and save the BIC
BIC = np.zeros(7)
for p in range(7):
    mod = ARMA(simulated_data_2, order=(p,0))
    res = mod.fit()
# Save BIC for AR(p)
    BIC[p] = res.bic

# Plot the BIC as a function of p
plt.plot(range(1,7), BIC[1:7], marker='o')
plt.xlabel('Order of AR Model')
plt.ylabel('Bayesian Information Criterion')
plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
```



Moving Average (MA) and ARMA Models

In this chapter you'll learn about another kind of model, the moving average, or MA, model. You will also see how to combine AR and MA models into a powerful ARMA model.

Side notes

- In time series analysis, the moving-average model (MA model), also known as moving-average process, is a common approach for modeling univariate time series. The moving-average model specifies that the output variable depends linearly on the current and various past values of a stochastic (imperfectly predictable) term.
- Together with the autoregressive (AR) model, the moving-average model is a special case and key component of the more general ARMA and ARIMA models of time series, which have a more complicated stochastic structure.
- The moving-average model **should not be confused with the moving average**, a distinct concept despite some similarities.
- Contrary to the AR model, the finite MA model is always stationary.
- Definition
$$X_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q}$$
where μ is the mean of the series and, the θ are parameters and ϵ are white noise error terms.
- From above, we know moving-average model is conceptually a linear regression of the current value of the series against current and previous (observed) white noise error terms or random shocks. The random shocks at each point are assumed to be mutually independent and to come from the same distribution, typically a normal distribution, with location at zero and constant scale.
- See the comparison of AR and MR in https://en.wikipedia.org/wiki/Moving-average_model (https://en.wikipedia.org/wiki/Moving-average_model).
- Fitting the MA estimates is more complicated than it is in autoregressive models (AR models), because the lagged error terms are not observable. This means that iterative non-linear fitting procedures need to be used in place of linear least squares.

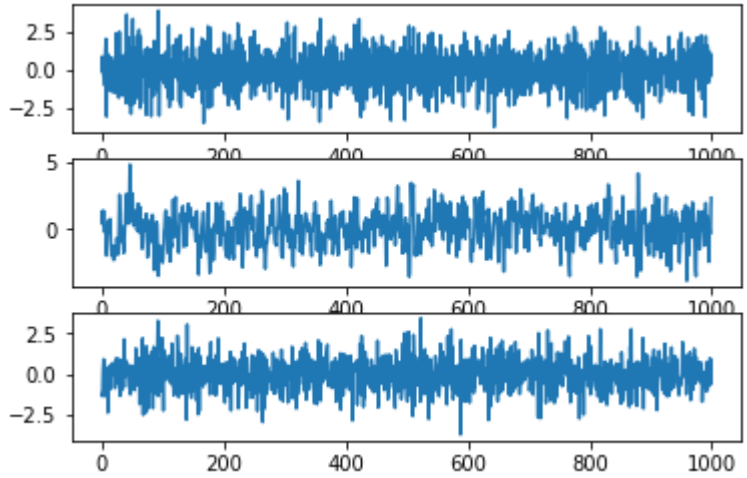
Simulate MA(1) Time Series

You will simulate and plot a few MA(1) time series, each with a different parameter, θ, using the arima_process module in statsmodels, just as you did in the last chapter for AR(1) models. You will look at an MA(1) model with a large positive θ and a large negative θ.

As in the last chapter, when inputting the coefficients, you must include the zero-lag coefficient of 1, **but unlike the last chapter on AR models, the sign of the MA coefficients is what we would expect.** For example, for an MA(1) process with $\theta = -0.9$, the array representing the MA parameters would be ma = np.array([1, -0.9])

In [47]:

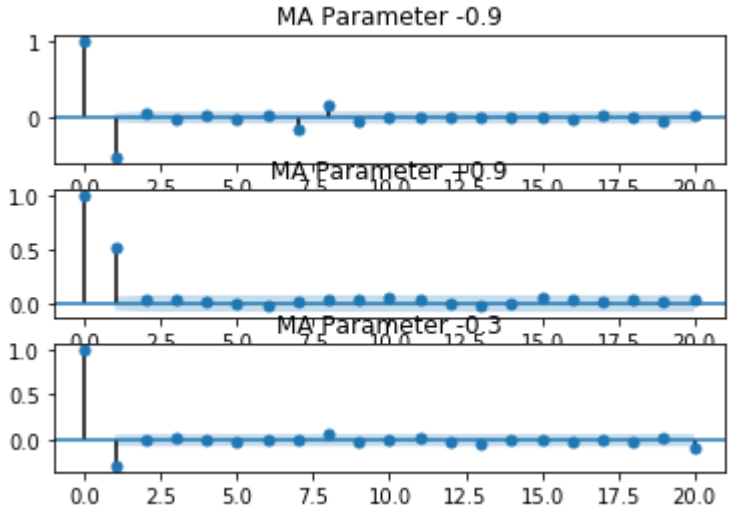
```
# import the module for simulating data
from statsmodels.tsa.arima_process import ArmaProcess

# Plot 1: MA parameter = -0.9
plt.subplot(3,1,1)
ar1 = np.array([1])
ma1 = np.array([1, -0.9])
MA_object1 = ArmaProcess(ar1, ma1)
simulated_data_1 = MA_object1.generate_sample(nsample=1000)
plt.plot(simulated_data_1)

# Plot 2: MA parameter = +0.9
plt.subplot(3,1,2)
ar2 = np.array([1])
ma2 = np.array([1, 0.9])
MA_object2 = ArmaProcess(ar2, ma2)
simulated_data_2 = MA_object2.generate_sample(nsample=1000)
plt.plot(simulated_data_2)

# Plot 2: MA parameter = -0.3
plt.subplot(3,1,3)
ar3 = np.array([1])
ma3 = np.array([1, -0.3])
MA_object3 = ArmaProcess(ar3, ma3)
simulated_data_3 = MA_object3.generate_sample(nsample=1000)
plt.plot(simulated_data_3)

plt.show()
```



Compute the ACF for Several MA Time Series

Unlike an AR(1), an MA(1) model has no autocorrelation beyond lag 1, an MA(2) model has no autocorrelation beyond lag 2, etc. The lag-1 autocorrelation for an MA(1) model is not θ, but rather $\frac{\theta}{1+\theta^2}$. For example, if the MA parameter, θ, is = +0.9, the first-lag autocorrelation will be $0.9/(1+(0.9)^2) = 0.497$, and the autocorrelation at all other lags will be zero. If the MA parameter, θ, is -0.9, the first-lag autocorrelation will be $-0.9/(1+(-0.9)^2) = -0.497$.

In [48]:
```
# Import the plot_acf module from statsmodels
from statsmodels.graphics.tsaplots import plot_acf

# Plot three ACF on same page for comparison using subplots
fig, axes = plt.subplots(3,1)

# Plot 1: AR parameter = -0.9
plot_acf(simulated_data_1, lags=20, ax=axes[0])
axes[0].set_title("MA Parameter -0.9")

# Plot 2: AR parameter = +0.9
plot_acf(simulated_data_2, lags=20, ax=axes[1])
axes[1].set_title("MA Parameter +0.9")

# Plot 3: AR parameter = -0.3
plot_acf(simulated_data_3, lags=20, ax=axes[2])
axes[2].set_title("MA Parameter -0.3")
plt.show()
```



Estimating an MA Model

You will estimate the MA(1) parameter, θ, of one of the simulated series that you generated in the earlier exercise. Since the parameters are known for a simulated series, it is a good way to understand the estimation routines before applying it to real data.

For simulated_data_1 with a true θ of -0.9, you will print out the estimate of θ. In addition, you will also print out the entire output that is produced when you fit a time series, so you can get an idea of what other tests and summary statistics are available in statsmodels.

In [49]:
```
# Import the ARMA module from statsmodels
from statsmodels.tsa.arima_model import ARMA

# Fit an MA(1) model to the first simulated data
mod = ARMA(simulated_data_1, order=(0,1))
res = mod.fit()

# Print out summary information on the fit
print(res.summary())

# Print out the estimate for the constant and for theta
print("When the true theta=-0.9, the estimate of theta (and the constant) are:")
print(res.params)
```

```
                              ARMA Model Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 1000
Model:                     ARMA(0, 1)   Log Likelihood               -1394.932
Method:                       css-mle   S.D. of innovations              0.975
Date:                Thu, 03 Jan 2019   AIC                           2795.864
Time:                        20:02:23   BIC                           2810.587
Sample:                             0   HQIC                          2801.459

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0031      0.002     -1.361      0.174      -0.007       0.001
ma.L1.y       -0.9280      0.012    -75.083      0.000      -0.952      -0.904
                                    Roots
=============================================================================
                  Real           Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
MA.1            1.0776           +0.0000j            1.0776            0.0000
-----------------------------------------------------------------------------
When the true theta=-0.9, the estimate of theta (and the constant) are:
[-0.00306454 -0.92797364]

C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
```

```
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
```

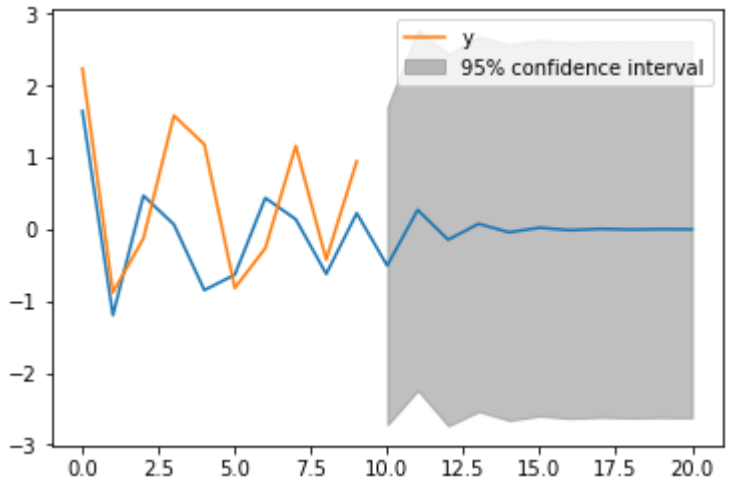
Forecasting with MA Model

As you did with AR models, you will use MA models to forecast in-sample and out-of-sample data using statsmodels.

For the simulated series simulated_data_1 with $\theta = -0.9$, you will plot in-sample and out-of-sample forecasts. **One big difference** you will see between out-of-sample forecasts with an MA(1) model and an AR(1) model is that the MA(1) forecasts more than one period in the future are simply the mean of the sample.

In [50]:
```
# Import the ARMA module from statsmodels
from statsmodels.tsa.arima_model import ARMA

# Forecast the first MA(1) model
mod = ARMA(simulated_data_1, order=(1,0))
res = mod.fit()
res.plot_predict(start=990, end=1010)
plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:577: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
```



Notice that the out-of-sample forecasts are flat into the future after the first data point.

High Frequency Stock Prices

Higher frequency stock data is well modelled by an MA(1) process, so it's a nice application of the models in this chapter.

The DataFrame intraday contains one day's prices (on September 1, 2017) for Sprint stock (ticker symbol "S") sampled at a frequency of one minute. The stock market is open for 6.5 hours (390 minutes), from 9:30am to 4:00pm.

Before you can analyze the time series data, you will have to clean it up a little, which you will do in this and the next two exercises. When you look at the first few rows (see the IPython Shell), you'll notice several things. First, there are no column headers.The data is not time stamped from 9:30 to 4:00, but rather goes from 0 to 390. And you will notice that the first date is the odd-looking "a1504272600". The number after the "a" is Unix time which is the number of seconds since January 1, 1970. This is how this dataset separates each day of intraday data.

If you look at the data types, you'll notice that the DATE column is an object, which here means a string. You will need to change that to numeric before you can clean up some missing data.

The source of the minute data is Google Finance (see here on how the data was downloaded).

The datetime module has already been imported for you.

In [71]:

```
import datetime
import pandas as pd
original = pd.read_csv('data/Sprint_Intraday.txt',header = None)
original.head()

#Change the first date to zero
original.iloc[0,0] = 0
intraday = original.iloc[:,0:2]

intraday.head()
# Change the column headers to 'DATE' and 'CLOSE'
intraday.columns = ['DATE','CLOSE']
intraday.head()
# Examine the data types for each column
print(intraday.dtypes)

# Convert DATE column to numeric
intraday['DATE'] = pd.to_numeric(intraday['DATE'])

# Make the `DATE` column the new index
intraday = intraday.set_index('DATE')
```

```
DATE      object
CLOSE    float64
dtype: object
```

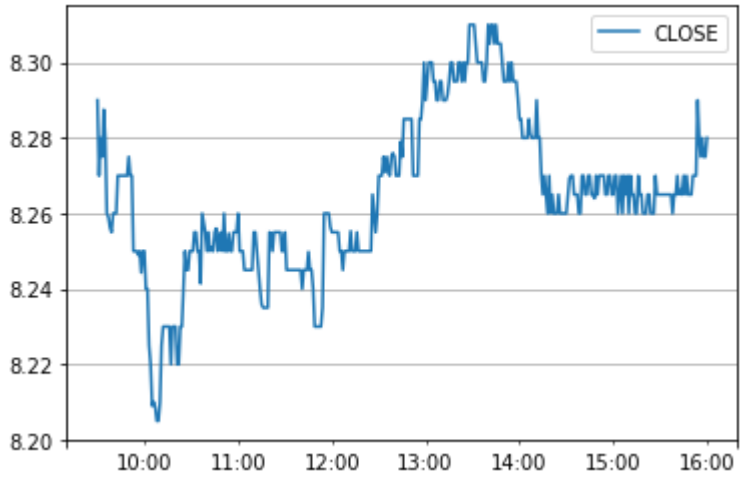
More Data Cleaning: Missing Data

When you print out the length of the DataFrame intraday, you will notice that a few rows are missing. There will be missing data if there are no trades in a particular one-minute interval. One way to see which rows are missing is to take the difference of two sets: the full set of every minute and the set of the DataFrame index which contains missing rows. You can fill in the missing rows with the .reindex() method, convert the index to time of day, and then plot the data.

Stocks trade at discrete one-cent increments (although a small percentage of trades occur in between the one-cent increments) rather than at continuous prices, and when you plot the data you should observe that there are long periods when the stock bounces back and forth over a one cent range. This is sometimes referred to as "bid/ask bounce".

In [72]:
```
# Notice that some rows are missing
print("The length of the DataFrame is: ",len(intraday))

# Find the missing rows
print("Missing rows: ", set(range(391)) - set(intraday.index))

# Fill in the missing rows
intraday = intraday.reindex(range(391), method='ffill')

# Change the index to the intraday times
intraday.index = pd.date_range(start='2017-08-28 9:30', end='2017-08-28 16:00', freq='1min')

# Plot the intraday time series
intraday.plot(grid=True)
plt.show()
```

```
The length of the DataFrame is:  389
Missing rows:  {182, 14}
```



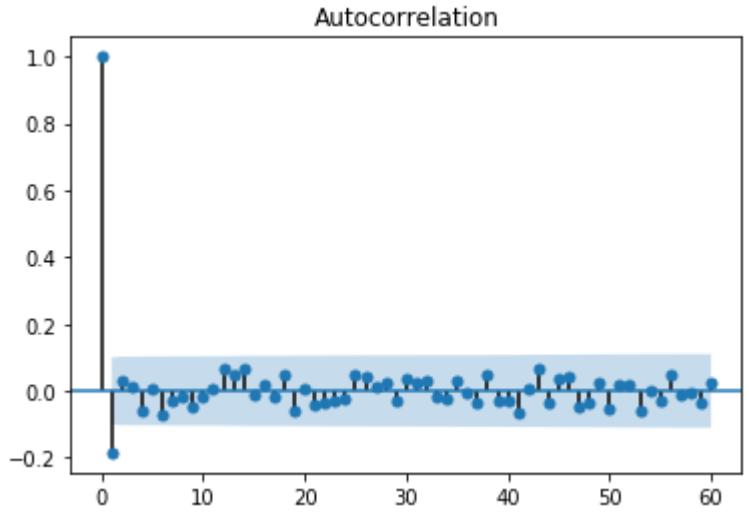
Missing data is common with high frequency financial time series, so good job fixing that.

Applying an MA Model

The bouncing of the stock price between bid and ask induces a negative first order autocorrelation, but no autocorrelations at lags higher than 1. You get the same ACF pattern with an MA(1) model. Therefore, you will fit an MA(1) model to the intraday stock data from the last exercise.

The first step is to compute minute-by-minute returns from the prices in intraday, and plot the autocorrelation function. You should observe that the ACF looks like that for an MA(1) process. Then, fit the data to an MA(1), the same way you did for simulated data.

In [73]:
```
# Import plot_acf and ARMA modules from statsmodels
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.arima_model import ARMA

# Compute returns from prices
returns = intraday.pct_change()
returns = returns.dropna()

# Plot ACF of returns with lags up to 60 minutes
plot_acf(returns, lags=60)
plt.show()

# Fit the data to an MA(1) model
mod = ARMA(returns, order=(0,1))
res = mod.fit()
print(res.params)
```




```
const         -0.000002
ma.L1.CLOSE   -0.179273
dtype: float64
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conve
rsion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will b
e treated as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conve
```

```
rsion of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future,
it will be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
```

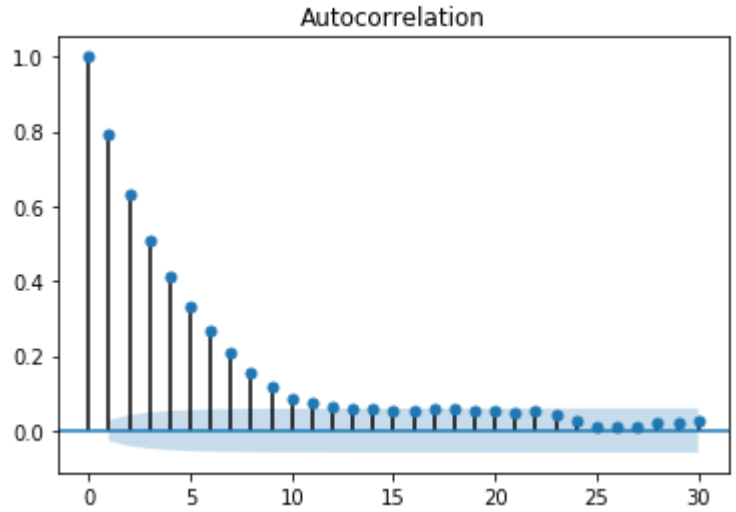
Notice the significant negative lag-1 autocorrelation, just like for an MA(1) model.

Equivalence of AR(1) and MA(infinity)

To better understand the relationship between MA models and AR models, you will demonstrate that an AR(1) model is equivalent to an MA(∞) model with the appropriate parameters.

You will simulate an MA model with parameters $0.8, 0.8^2, 0.8^3, \ldots$ for a large number (30) lags and show that it has the same Autocorrelation Function as an AR(1) model with $\phi = 0.8$.

In [74]:
```
# import the modules for simulating data and plotting the ACF
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.graphics.tsaplots import plot_acf

# Build a list MA parameters
ma = [.8**i for i in range(30)]

# Simulate the MA(30) model
ar = np.array([1])
AR_object = ArmaProcess(ar, ma)
simulated_data = AR_object.generate_sample(nsample=5000)

# Plot the ACF
plot_acf(simulated_data, lags=30)
plt.show()
```



Notice that the ACF looks the same as an AR(1) with parameter 0.8

Putting It All Together

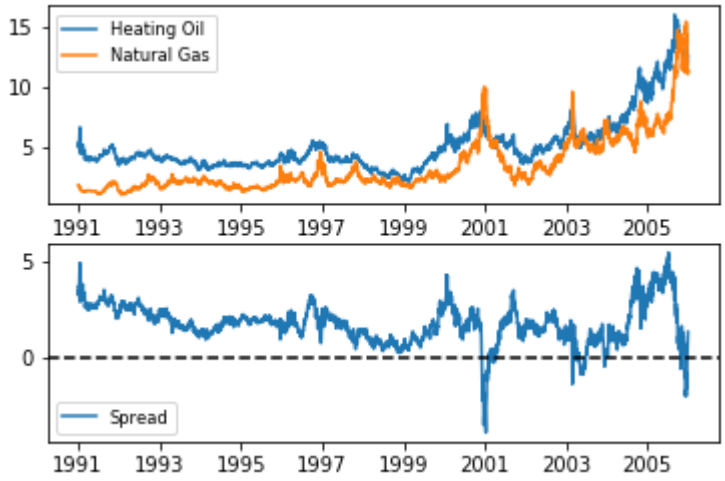
See good example in the slides downloaded.
This chapter will show you how to model two series jointly using cointegration models. Then you'll wrap up with a case study where you look at a time series of temperature data from New York City.

A Dog on a Leash? (Part 1)

The Heating Oil and Natural Gas prices are pre-loaded in DataFrames HO and NG. First, plot both price series, which look like random walks. Then plot the difference between the two series, which should look more like a mean reverting series (to put the two series in the same units, we multiply the heating oil prices, in $/gallon, by 7.25, which convert sitto/$millionBTU, which is the same units as Natural Gas).

The data for continuous futures (each contract has to be spliced together in a continuous series as contracts expire) was obtained from Quandl.

In [75]:
```
import pandas as pd
HO = pd.read_csv('data/CME_HO1.csv',index_col='Date', parse_dates=True)
NG = pd.read_csv('data/CME_NG1.csv',index_col='Date', parse_dates=True)

# Plot the prices separately
plt.subplot(2,1,1)
plt.plot(7.25*HO, label='Heating Oil')
plt.plot(NG, label='Natural Gas')
plt.legend(loc='best', fontsize='small')

# Plot the spread
plt.subplot(2,1,2)
plt.plot(7.25 *HO-NG, label='Spread')
plt.legend(loc='best', fontsize='small')
plt.axhline(y=0, linestyle='--', color='k')
plt.show()
```



Notice from the plot that when Heating Oil briefly dipped below Natural Gas, it quickly reverted back up.

A Dog on a Leash? (Part 2)

To verify that HO and NG are cointegrated, First apply the Dickey-Fuller test to HO and NG separately to show they are random walks. Then apply the test to the difference, which should strongly reject the random walk hypothesis. The Heating Oil and Natural Gas prices are pre-loaded in DataFrames HO and NG.

In [76]:
```
# Import the adfuller module from statsmodels
from statsmodels.tsa.stattools import adfuller

# Compute the ADF for HO and NG
result_HO = adfuller(HO['Close'])
print("The p-value for the ADF test on HO is ", result_HO[1])
result_NG = adfuller(NG['Close'])
print("The p-value for the ADF test on NG is ", result_NG[1])

# Compute the ADF of the spread
result_spread = adfuller(7.25 * HO['Close'] - NG['Close'])
print("The p-value for the ADF test on the spread is ", result_spread[1])
```

```
The p-value for the ADF test on HO is  0.019831028071626525
The p-value for the ADF test on NG is  0.004547284956542479
The p-value for the ADF test on the spread is  0.00011887051827353092
```

As we expected, we cannot reject the hypothesis that the individual futures are random walks, but we can reject that the spread is a random walk.

Are Bitcoin and Ethereum Cointegrated?

Cointegration involves two steps: regressing one time series on the other to get the cointegration vector, and then perform an ADF test on the residuals of the regression. In the last example, there was no need to perform the first step since we implicitly assumed the cointegration vector was (1,−1). In other words, we took the difference between the two series (after doing a units conversion). Here, you will do both steps.

You will regress the value of one cryptocurrency, bitcoin (BTC), on another cryptocurrency, ethereum (ETH). If we call the regression coefficient b, then the cointegration vector is simply (1,−b). Then perform the ADF test on BTC −b ETH. Bitcoin and Ethereum prices are pre-loaded in DataFrames BTC and ETH.

Bitcoin data are in DataFrame BTC and Ethereum data are in ETH.

In [ ]:
```
# Import the statsmodels module for regression and the adfuller function
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller

# Regress BTC on ETH
ETH = sm.add_constant(ETH)
result = sm.OLS(BTC,ETH).fit()

# Compute ADF
b = result.params[1]
adf_stats = adfuller(BTC['Price'] - b*ETH['Price'])
print("The p-value for the ADF test is ", adf_stats[1])
```

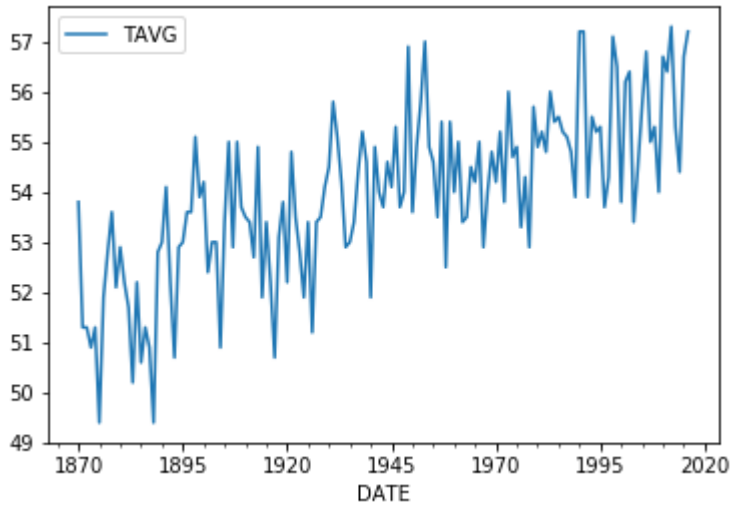
The p-value for the ADF test is 0.0233690023235. The data suggests that Bitcoin and Ethereum are cointegrated.

Is Temperature a Random Walk (with Drift)?

An ARMA model is a simplistic approach to forecasting climate changes, but it illustrates many of the topics covered in this class.

The DataFrame temp_NY contains the average annual temperature in Central Park, NY from 1870-2016 (the data was downloaded from the NOAA here). Plot the data and test whether it follows a random walk (with drift).

In [24]:
```
import pandas as pd
temp_NY = pd.read_csv('data/temp_NY.txt',sep = '\s+',header = None)
temp_NY.columns = ['DATE','TAVG']
#temp_NY.index = temp_NY['DATE']
#temp_NY.set_index('DATE')
temp_NY = temp_NY.set_index('DATE')
temp_NY.index = pd.to_datetime(temp_NY.index, format='%Y')

# Import the adfuller function from the statsmodels module
from statsmodels.tsa.stattools import adfuller
import matplotlib.pyplot as plt

# Plot average temperatures
temp_NY.plot()
plt.show()

# Compute and print ADF p-value
result = adfuller(temp_NY['TAVG'])
print("The p-value for the ADF test is ", result[1])
```



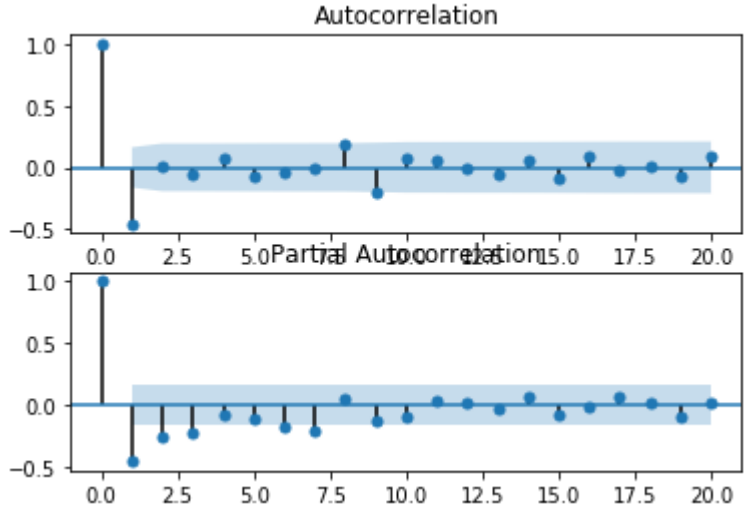

```
The p-value for the ADF test is  0.5832938987871106
```

The data seems to follow a random walk with drift.

Getting "Warmed" Up: Look at Autocorrelations

Since the temperature series, temp_NY, is a random walk with drift, take first differences to make it stationary. Then compute the sample ACF and PACF. This will provide some guidance on the order of the model.

In [25]:
```
# Import the modules for plotting the sample ACF and PACF
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Take first difference of the temperature Series
chg_temp = temp_NY.diff()
chg_temp = chg_temp.dropna()

# Plot the ACF and PACF on the same page
fig, axes = plt.subplots(2,1)

# Plot the ACF
plot_acf(chg_temp, lags=20, ax=axes[0])

# Plot the PACF
plot_pacf(chg_temp, lags=20, ax=axes[1])
plt.show()
```



There is no clear pattern in the ACF and PACF except the negative lag-1 autocorrelation in the ACF.

About ARMA (side note)

- ARMA model is simply the merger between AR(p) and MA(q) models:
  - AR(p) models try to explain the momentum and mean reversion effects often observed in trading markets (market participant effects).
  - MA(q) models try to capture the shock effects observed in the white noise terms. These shock effects could be thought of as unexpected events affecting the observation process e.g. Surprise earnings, wars, attacks, etc.
- ARMA model attempts to capture both of these aspects when modelling financial time series. ARMA model does not take into account volatility clustering, a key empirical phenomena of many financial time series.
- In the statistical analysis of time series, autoregressive–moving-average (ARMA) models provide a parsimonious description of a (weakly) stationary stochastic process in terms of two polynomials, one for the autoregression (AR) and the second for the moving average (MA).
- Definition: ARMA(p, q) refers to the model with p autoregressive terms and q moving-average terms.

$$X_t = c + \epsilon_t + \sum_{i=1}^{p} \phi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}$$

Which ARMA Model is Best?

Fit the temperature data to an AR(1), AR(2), MA(1), and ARMA(1,1) and see which model is the best fit, using the AIC criterion.

In [26]:
```
# Import the module for estimating an ARMA model
from statsmodels.tsa.arima_model import ARMA

# Fit the data to an AR(1) model and print AIC:
mod = ARMA(chg_temp, order=(1,0))
res = mod.fit()
print("The AIC for an AR(1) is: ", res.aic)

# Fit the data to an AR(2) model and print AIC:
mod = ARMA(chg_temp, order=(2,0))
res = mod.fit()
print("The AIC for an AR(2) is: ", res.aic)

# Fit the data to an MA(1) model and print AIC:
mod = ARMA(chg_temp, order=(0,1))
res = mod.fit()
print("The AIC for an MA(1) is: ", res.aic)

# Fit the data to an ARMA(1,1) model and print AIC:
mod = ARMA(chg_temp, order=(1,1))
res = mod.fit()
print("The AIC for an ARMA(1,1) is: ", res.aic)
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):

The AIC for an AR(1) is:  510.5346898313911
The AIC for an AR(2) is:  501.92741231602287
The AIC for an MA(1) is:  469.3909741274666
The AIC for an ARMA(1,1) is:  469.07291682954957

C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\base\model.py:496: ConvergenceWarning: Maximum Likelihoo
d optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

The MA(1) and ARMA(1,1) have the two lowest AIC values.

Don't Throw Out That Winter Coat Yet

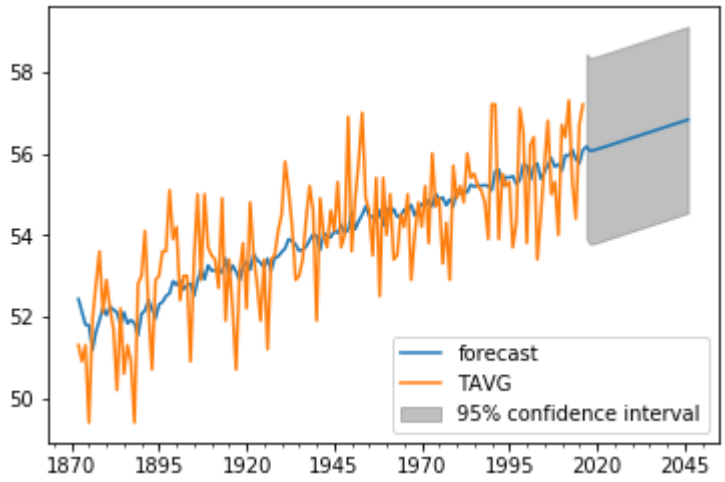
Finally, you will forecast the temperature over the next 30 years using an ARMA(1,1) model, including confidence bands around that estimate. Keep in mind that the estimate of the drift will have a much bigger impact on long range forecasts than the ARMA parameters.

Earlier, you determined that the temperature data follows a random walk and you looked at first differencing the data. You will use the ARIMA module on the temperature data, pre-loaded in the DataFrame temp_NY, but the forecast would be the same as using the ARMA module on changes in temperature, and then using cumulative sums of these changes to get the temperature.

The data is in a DataFrame called temp_NY.

In [27]:
```
# Import the ARIMA module from statsmodels
from statsmodels.tsa.arima_model import ARIMA

# Forecast interest rates using an AR(1) model
mod = ARIMA(temp_NY, order=(1,1,1))
res = mod.fit()

# Plot the original series and the forecasted series
res.plot_predict(start='1872-01-01', end='2046-01-01')
plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:646: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:650: FutureWarning: Conversi
on of the second argument of issubdtype from `complex` to `np.complexfloating` is deprecated. In future, it wil
l be treated as `np.complex128 == np.dtype(complex).type`.
  elif issubdtype(paramsdtype, complex):
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\base\model.py:496: ConvergenceWarning: Maximum Likelihoo
d optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
C:\Users\ljyan\Anaconda3\lib\site-packages\statsmodels\tsa\kalmanf\kalmanfilter.py:577: FutureWarning: Conversi
on of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be trea
ted as `np.float64 == np.dtype(float).type`.
  if issubdtype(paramsdtype, float):
```



According to the model (almost entirely due to the trend, the temperature is expected to be about 0.6 degrees higher in 30 years, but the 95% confidence interval around that is over 5 degrees.

Advanced Topics

In the introductory course about time series, we learned some models. This lays a background for learning other more advanced models such as

- GARCH Models
- Nonlinear Models
- Multivariate Time Series Models
- Regime Switching Models
- State Space Models and Kalman Filtering