1 Importing libraries

In [86]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.tsa.stattools import adfuller
%matplotlib inline
```

In [87]:

```
# Set parameters for better visualization
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (15, 10)
```

2 Reading the cleaned Data

In [88]:

```
df = pd.read_excel("MonthWiseMarketArrivals_ChennaiClean.xlsx")
df.head()
```

Out[88]:

	market	month	year	quantity	priceMin	priceMax	priceMod	date
0	CHENNAI	January	2004	103400	798	1019	910	2004-01-01
1	CHENNAI	February	2004	87800	776	969	873	2004-02-01
2	CHENNAI	March	2004	102180	506	656	580	2004-03-01
3	CHENNAI	April	2004	83300	448	599	527	2004-04-01
4	CHENNAI	May	2004	84850	462	596	529	2004-05-01

In [89]:

```
# change the date column to time interval column
df.date = pd.DatetimeIndex(df.date)
```

In [90]:

```
# change the index to date column
df = df.sort_values(by="date")
df.index = pd.PeriodIndex(df.date, freq="M")
df.head()
```

Out[90]:

	market	month	year	quantity	priceMin	priceMax	priceMod	date
date								
2004-01	CHENNAI	January	2004	103400	798	1019	910	2004-01-01
2004-02	CHENNAI	February	2004	87800	776	969	873	2004-02-01
2004-03	CHENNAI	March	2004	102180	506	656	580	2004-03-01
2004-04	CHENNAI	April	2004	83300	448	599	527	2004-04-01
2004-05	CHENNAI	May	2004	84850	462	596	529	2004-05-01

3 Neglecting unimportant variables

In [91]:

```
df = df.drop(["market","month","year","priceMin","priceMax"], axis=1)
df.tail()
```

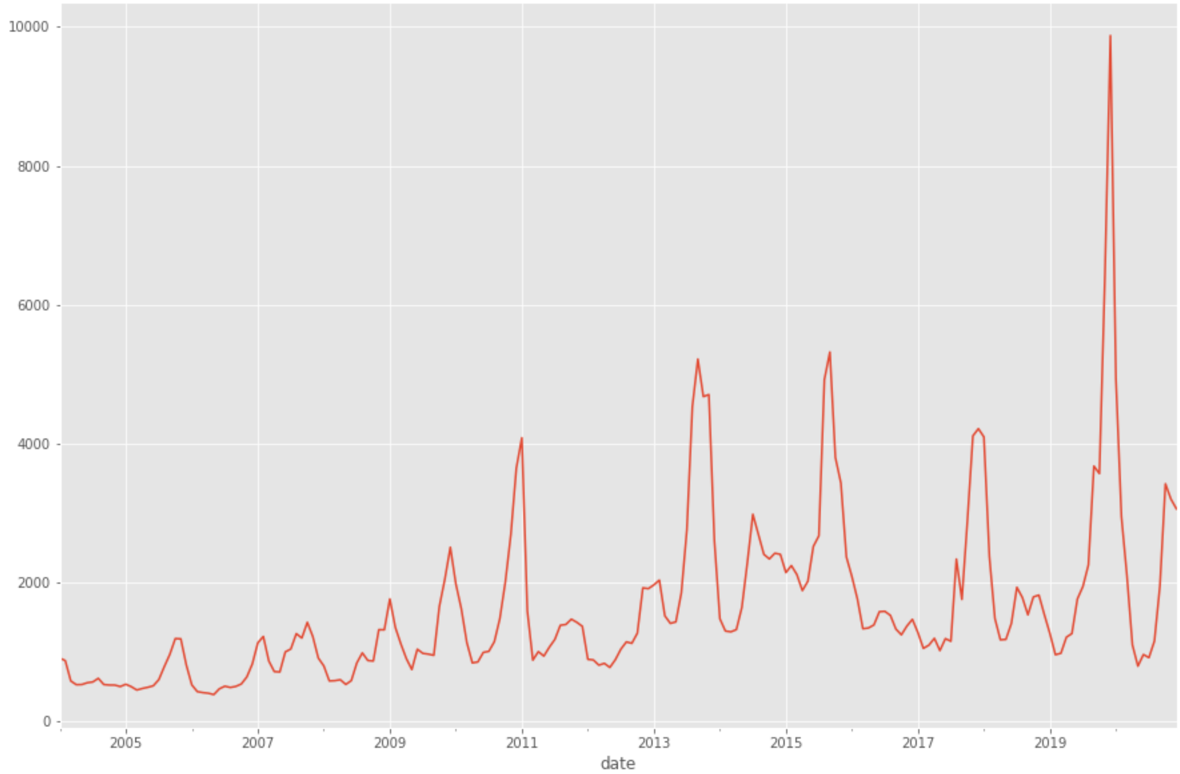
Out[91]:

	quantity	priceMod	date
date			
2020-08	92024	1155	2020-08-01
2020-09	91436	1950	2020-09-01
2020-10	90849	3420	2020-10-01
2020-11	90262	3200	2020-11-01
2020-12	89675	3060	2020-12-01

4 Date vs Price

In [92]:

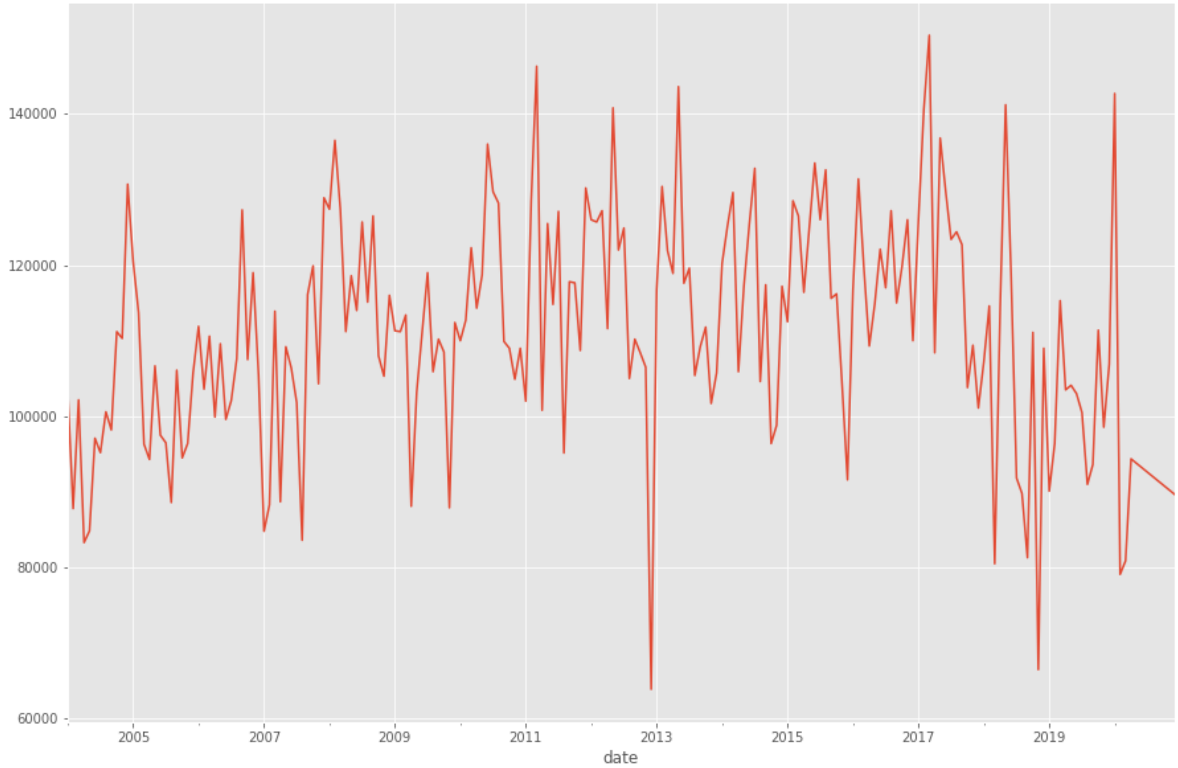
```
df.priceMod.plot()
plt.show()
```



5 Date vs Quantity

In [93]:

```
df.quantity.plot()
plt.show()
```



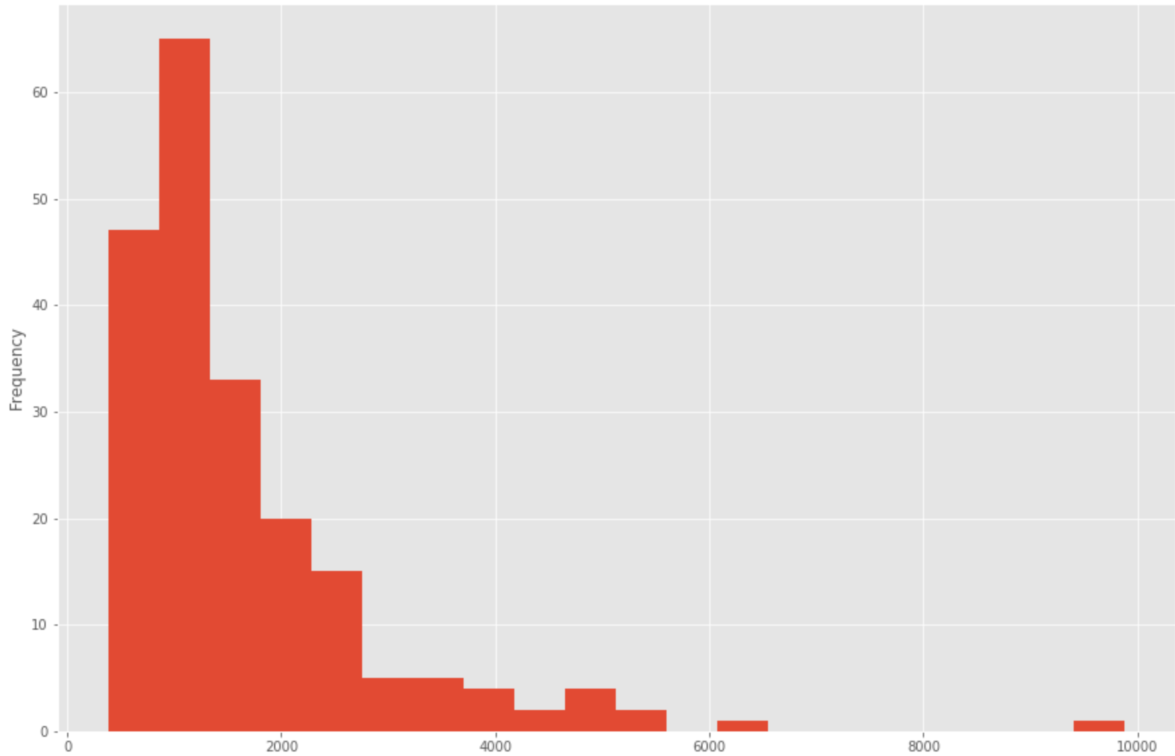
6 Price distribution

In [94]:

```
df.priceMod.plot(kind="hist", bins=20)
```

Out[94]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2871f0e8a08>
```



7 Logged Price

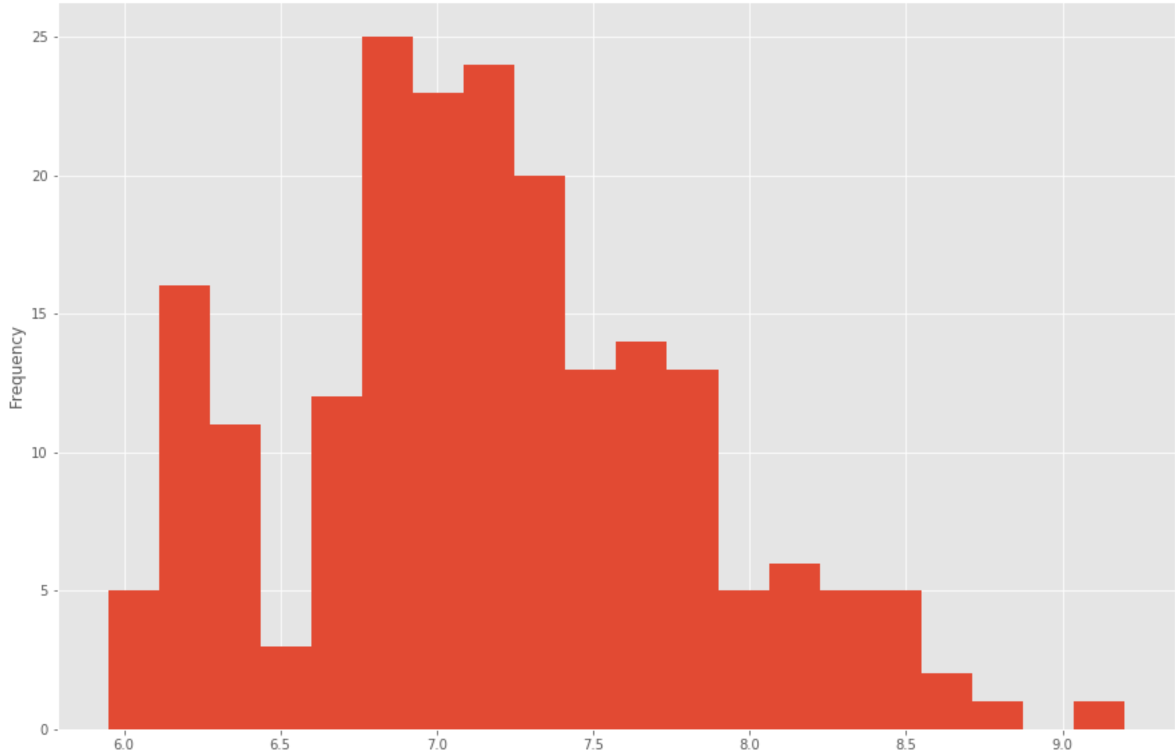
Log-transformations can help to stabilize the variance of a time series. Let see using an example:

In [95]:

```
df["log_priceMod"] = np.log(df.priceMod)
df.log_priceMod.plot(kind="hist", bins=20)
```

Out[95]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x287200adc88>
```



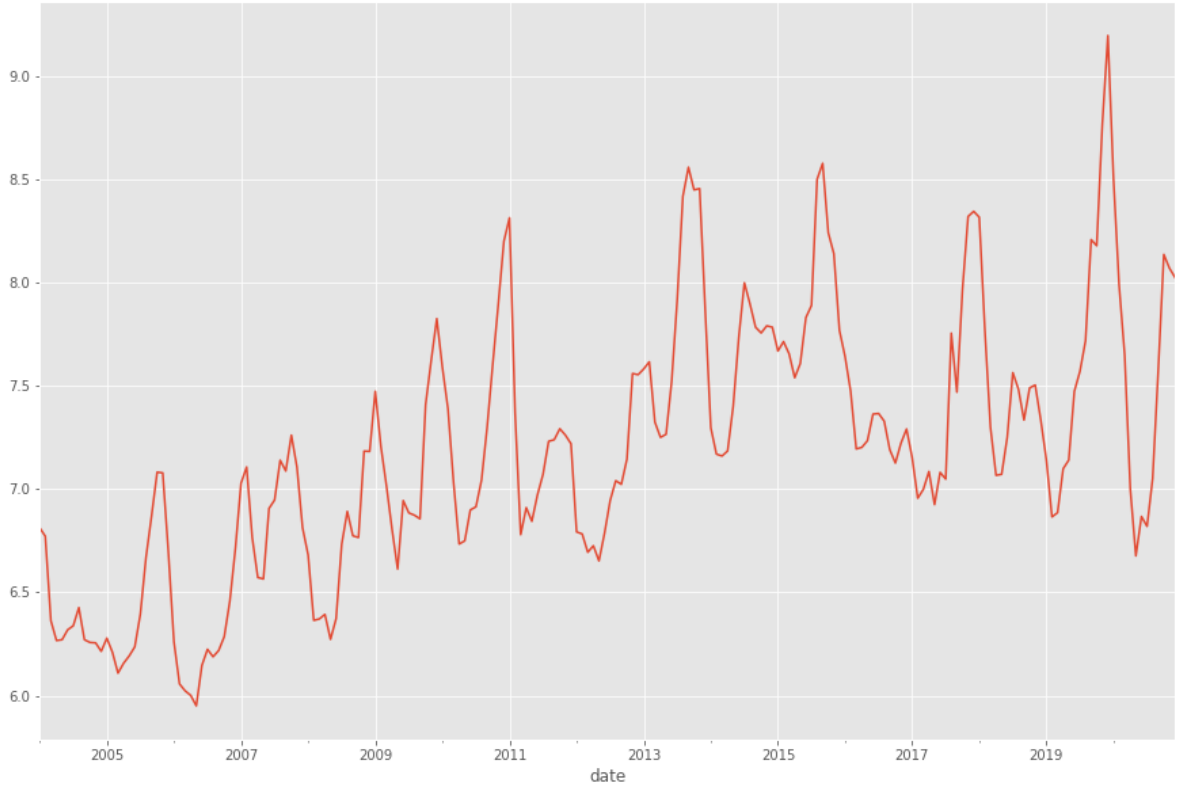
The above histogram is more look like normal distribution

In [96]:

```
df.log_priceMod.plot()
```

Out[96]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2871f20c7c8>
```



8 Basic Time Series Model

We will build a time-series forecasting model to get a forecast for Onion prices. Let us start with the three most basic models -

1. Mean Constant Model
2. Linear Trend Model
3. Random Walk Model

9 1. Mean Constant Model

In [97]:

```
df_mean = df.log_priceMod.mean()
df["mean_price"] = np.exp(df_mean)
df.head()
```

Out[97]:

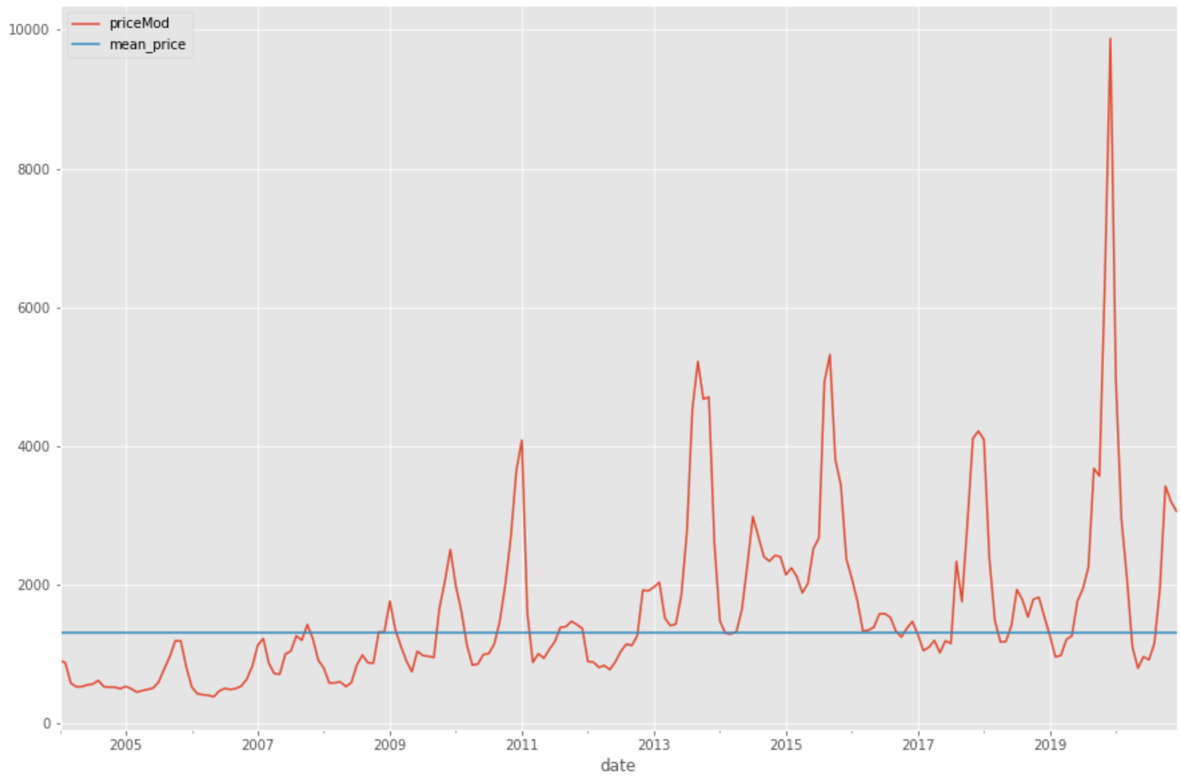
date	quantity	priceMod	date	log_priceMod	mean_price
2004-01	103400	910	2004-01-01	6.813445	1312.94077
2004-02	87800	873	2004-02-01	6.771936	1312.94077
2004-03	102180	580	2004-03-01	6.363028	1312.94077
2004-04	83300	527	2004-04-01	6.267201	1312.94077
2004-05	84850	529	2004-05-01	6.270988	1312.94077

In [98]:

```
df.plot(kind="line", x="date", y=["priceMod", "mean_price"])
```

Out[98]:

<matplotlib.axes._subplots.AxesSubplot at 0x28720734648>



10 Evaluate this model using RSME

In [99]:

```
def RMSE(actual, predicted):
    mse = (actual - predicted)**2
    rmse = np.sqrt(mse.sum()/mse.count())
    return rmse
```

In [100]:

```
mean_modelRMSE = RMSE(df.priceMod, df.mean_price)
mean_modelRMSE
```

Out[100]:

```
1271.6884180325062
```

In [101]:

```
Result_df = pd.DataFrame(columns =["Model","Actual","Forcast","RMSE"])
Result_df.loc[0,"Model"] = "Mean Model"
Result_df.loc[0,"Actual"] = "3060"
Result_df.loc[0,"Forcast"] = np.exp(df_mean)
Result_df.loc[0,"RMSE"] = mean_modelRMSE
Result_df
```

Out[101]:

	Model	Actual	Forcast	RMSE
0	Mean Model	3060	1312.94	1271.69

11 2. Linear Trend Model

Let us start by plotting a linear trend model between log_priceMod and time.

However to do linear regression, we need a numeric indicator for time period - Let us create that

In [102]:

```
df.head()
```

Out[102]:

	quantity	priceMod	date	log_priceMod	mean_price
date					
2004-01	103400	910	2004-01-01	6.813445	1312.94077
2004-02	87800	873	2004-02-01	6.771936	1312.94077
2004-03	102180	580	2004-03-01	6.363028	1312.94077
2004-04	83300	527	2004-04-01	6.267201	1312.94077
2004-05	84850	529	2004-05-01	6.270988	1312.94077

In [103]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
PeriodIndex: 204 entries, 2004-01 to 2020-12
Freq: M
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   quantity      204 non-null    int64
 1   priceMod      204 non-null    int64
 2   date          204 non-null    datetime64[ns]
 3   log_priceMod  204 non-null    float64
 4   mean_price    204 non-null    float64
dtypes: datetime64[ns](1), float64(2), int64(2)
memory usage: 9.6 KB
```

In [104]:

```
# Converting the date into datetinme delta starting from 0
df["timeindex"] = df.date - df.date.min()
df.head()
```

Out[104]:

	quantity	priceMod	date	log_priceMod	mean_price	timeindex
date						
2004-01	103400	910	2004-01-01	6.813445	1312.94077	0 days
2004-02	87800	873	2004-02-01	6.771936	1312.94077	31 days
2004-03	102180	580	2004-03-01	6.363028	1312.94077	60 days
2004-04	83300	527	2004-04-01	6.267201	1312.94077	91 days
2004-05	84850	529	2004-05-01	6.270988	1312.94077	121 days

In [105]:

```
df.dtypes
```

Out[105]:

```
quantity                  int64
priceMod                  int64
date             datetime64[ns]
log_priceMod            float64
mean_price              float64
timeindex       timedelta64[ns]
dtype: object
```

In [106]:

```
# converting the timeindex into months using timedelta
df["timeindex"] = df["timeindex"]/np.timedelta64(1,"M")
df.head()
```

Out[106]:

date	quantity	priceMod	date	log_priceMod	mean_price	timeindex
2004-01	103400	910	2004-01-01	6.813445	1312.94077	0.000000
2004-02	87800	873	2004-02-01	6.771936	1312.94077	1.018501
2004-03	102180	580	2004-03-01	6.363028	1312.94077	1.971293
2004-04	83300	527	2004-04-01	6.267201	1312.94077	2.989794
2004-05	84850	529	2004-05-01	6.270988	1312.94077	3.975441

In [107]:

```
df["timeindex"] = df["timeindex"].round(0).astype(int)
df.tail()
```

Out[107]:

date	quantity	priceMod	date	log_priceMod	mean_price	timeindex
2020-08	92024	1155	2020-08-01	7.051856	1312.94077	199
2020-09	91436	1950	2020-09-01	7.575585	1312.94077	200
2020-10	90849	3420	2020-10-01	8.137396	1312.94077	201
2020-11	90262	3200	2020-11-01	8.070906	1312.94077	202
2020-12	89675	3060	2020-12-01	8.026170	1312.94077	203

12 Apply the linear model

In [108]:

```
linear_model = smf.ols('log_priceMod ~ timeindex', data = df).fit()
linear_model.summary()
```

Out[108]:

OLS Regression Results

Dep. Variable:	log_priceMod	**R-squared:**	0.431
Model:	OLS	**Adj. R-squared:**	0.428
Method:	Least Squares	**F-statistic:**	153.2
Date:	Sun, 20 Dec 2020	**Prob (F-statistic):**	1.50e-26
Time:	16:55:15	**Log-Likelihood:**	-137.36
No. Observations:	204	**AIC:**	278.7
Df Residuals:	202	**BIC:**	285.4
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	6.4679	0.067	97.232	0.000	6.337	6.599
timeindex	0.0070	0.001	12.376	0.000	0.006	0.008

Omnibus:	9.357	**Durbin-Watson:**	0.276
Prob(Omnibus):	0.009	**Jarque-Bera (JB):**	9.379
Skew:	0.512	**Prob(JB):**	0.00919
Kurtosis:	3.232	**Cond. No.**	234.

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [109]:

```
linear_model_pred = linear_model.predict()
linear_model_pred
```

Out[109]:

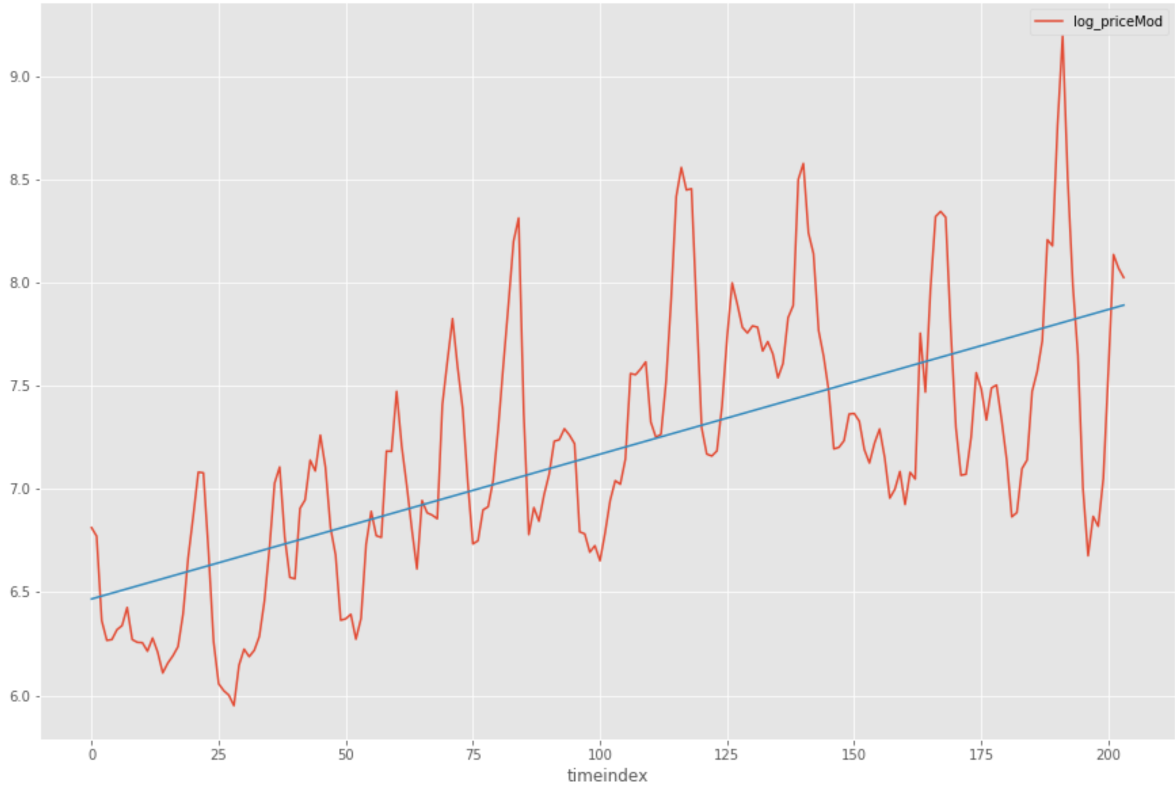
```
array([6.46792105, 6.47493685, 6.48195265, 6.48896845, 6.49598425,
       6.50300005, 6.51001585, 6.51703165, 6.52404745, 6.53106325,
       6.53807905, 6.54509485, 6.55211065, 6.55912645, 6.56614225,
       6.57315805, 6.58017385, 6.58718965, 6.59420545, 6.60122125,
       6.60823705, 6.61525285, 6.62226865, 6.62928445, 6.63630025,
       6.64331605, 6.65033185, 6.65734765, 6.66436345, 6.67137925,
       6.67839505, 6.68541085, 6.69242665, 6.69944245, 6.70645825,
       6.71347405, 6.72048985, 6.72750565, 6.73452145, 6.74153725,
       6.74855305, 6.75556885, 6.76258465, 6.76960045, 6.77661625,
       6.78363205, 6.79064785, 6.79766365, 6.80467945, 6.81169525,
       6.81871105, 6.82572685, 6.83274265, 6.83975845, 6.84677425,
       6.85379005, 6.86080585, 6.86782166, 6.87483746, 6.88185326,
       6.88886906, 6.89588486, 6.90290066, 6.90991646, 6.91693226,
       6.92394806, 6.93096386, 6.93797966, 6.94499546, 6.95201126,
       6.95902706, 6.96604286, 6.97305866, 6.98007446, 6.98709026,
       6.99410606, 7.00112186, 7.00813766, 7.01515346, 7.02216926,
       7.02918506, 7.03620086, 7.04321666, 7.05023246, 7.05724826,
       7.06426406, 7.07127986, 7.07829566, 7.08531146, 7.09232726,
       7.09934306, 7.10635886, 7.11337466, 7.12039046, 7.12740626,
       7.13442206, 7.14143786, 7.14845366, 7.15546946, 7.16248526,
       7.16950106, 7.17651686, 7.18353266, 7.19054846, 7.19756426,
       7.20458006, 7.21159586, 7.21861166, 7.22562746, 7.23264326,
       7.23965906, 7.24667486, 7.25369066, 7.26070647, 7.26772227,
       7.27473807, 7.28175387, 7.28876967, 7.29578547, 7.30280127,
       7.30981707, 7.31683287, 7.32384867, 7.33086447, 7.33788027,
       7.34489607, 7.35191187, 7.35892767, 7.36594347, 7.37295927,
       7.37997507, 7.38699087, 7.39400667, 7.40102247, 7.40803827,
       7.41505407, 7.42206987, 7.42908567, 7.43610147, 7.44311727,
       7.45013307, 7.45714887, 7.46416467, 7.47118047, 7.47819627,
       7.48521207, 7.49222787, 7.49924367, 7.50625947, 7.51327527,
       7.52029107, 7.52730687, 7.53432267, 7.54133847, 7.54835427,
       7.55537007, 7.56238587, 7.56940167, 7.57641747, 7.58343327,
       7.59044907, 7.59746487, 7.60448067, 7.61149647, 7.61851227,
       7.62552807, 7.63254387, 7.63955967, 7.64657547, 7.65359128,
       7.66060708, 7.66762288, 7.67463868, 7.68165448, 7.68867028,
       7.69568608, 7.70270188, 7.70971768, 7.71673348, 7.72374928,
       7.73076508, 7.73778088, 7.74479668, 7.75181248, 7.75882828,
       7.76584408, 7.77285988, 7.77987568, 7.78689148, 7.79390728,
       7.80092308, 7.80793888, 7.81495468, 7.82197048, 7.82898628,
       7.83600208, 7.84301788, 7.85003368, 7.85704948, 7.86406528,
       7.87108108, 7.87809688, 7.88511268, 7.89212848])
```

In [110]:

```
df.plot(kind = "line", x="timeindex", y="log_priceMod")
plt.plot(df.timeindex, linear_model_pred)
```

Out[110]:

```
[<matplotlib.lines.Line2D at 0x28720c1a908>]
```

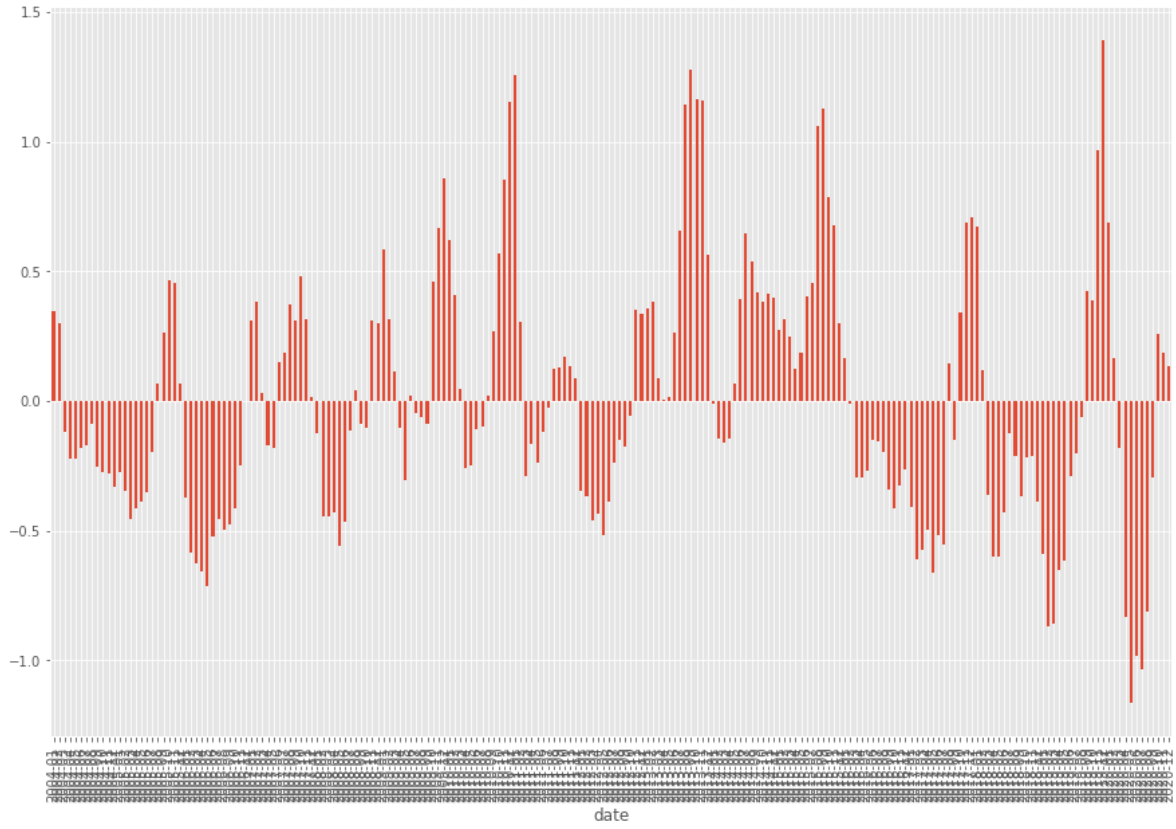


In [111]:

```
linear_model.resid.plot(kind="bar")
```

Out[111]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x28720c68e88>
```



What measures can we check to see if the model is good?

It is seen here (and also evident on the regression line plot, if you look closely) that the linear trend model has a tendency to make an error of the same sign for many periods in a row. This tendency is measured in statistical terms by the lag-1 autocorrelation and Durbin-Watson statistic. If there is no time pattern, the lag-1 autocorrelation should be very close to zero, and the Durbin-Watson statistic ought to be very close to 2, which is not the case here. If the model has succeeded in extracting all the "signal" from the data, there should be no pattern at all in the errors: the error in the next period should not be correlated with any previous errors. The linear trend model obviously fails the autocorrelation test in this case.

1. Durbin Watson statistic is a test for autocorrelation in a data set.
2. The DW statistic always has a value between zero and 4.0.
3. A value of 2.0 means there is no autocorrelation detected in the sample. Values from zero to 2.0 indicate positive autocorrelation and values from 2.0 to 4.0 indicate negative autocorrelation

A stock price displaying positive autocorrelation would indicate that the price yesterday has a positive correlation on the price today—so if the stock fell yesterday, it is also likely that it falls today. A security that has a negative autocorrelation, on the other hand, has a negative influence on itself over time—so that if it fell yesterday, there is a greater likelihood it will rise today.

In [112]:

```
# Manual Calculation
model_linear_forecast_manual = 0.0077 * 203 + 6.4679
model_linear_forecast_manual
```

Out[112]:

```
8.031
```

In [113]:

```
df["linear_price"] = np.exp(linear_model_pred)
df.head()
```

Out[113]:

	quantity	priceMod	date	log_priceMod	mean_price	timeindex	linear_price
date							
2004-01	103400	910	2004-01-01	6.813445	1312.94077	0	644.143189
2004-02	87800	873	2004-02-01	6.771936	1312.94077	1	648.678259
2004-03	102180	580	2004-03-01	6.363028	1312.94077	2	653.245258
2004-04	83300	527	2004-04-01	6.267201	1312.94077	3	657.844411
2004-05	84850	529	2004-05-01	6.270988	1312.94077	4	662.475944

In [114]:

```
linear_model_RMSE = RMSE(df.priceMod, df.linear_price)
linear_model_RMSE
```

Out[114]:

```
1096.569541901011
```

In [115]:

```
Result_df.loc[1,"Model"] = "Linear Model"
Result_df.loc[1,"Actual"] = "3060"
Result_df.loc[1,"Forcast"] = np.exp(model_linear_forecast_manual)
Result_df.loc[1,"RMSE"] = linear_model_RMSE
Result_df
```

Out[115]:

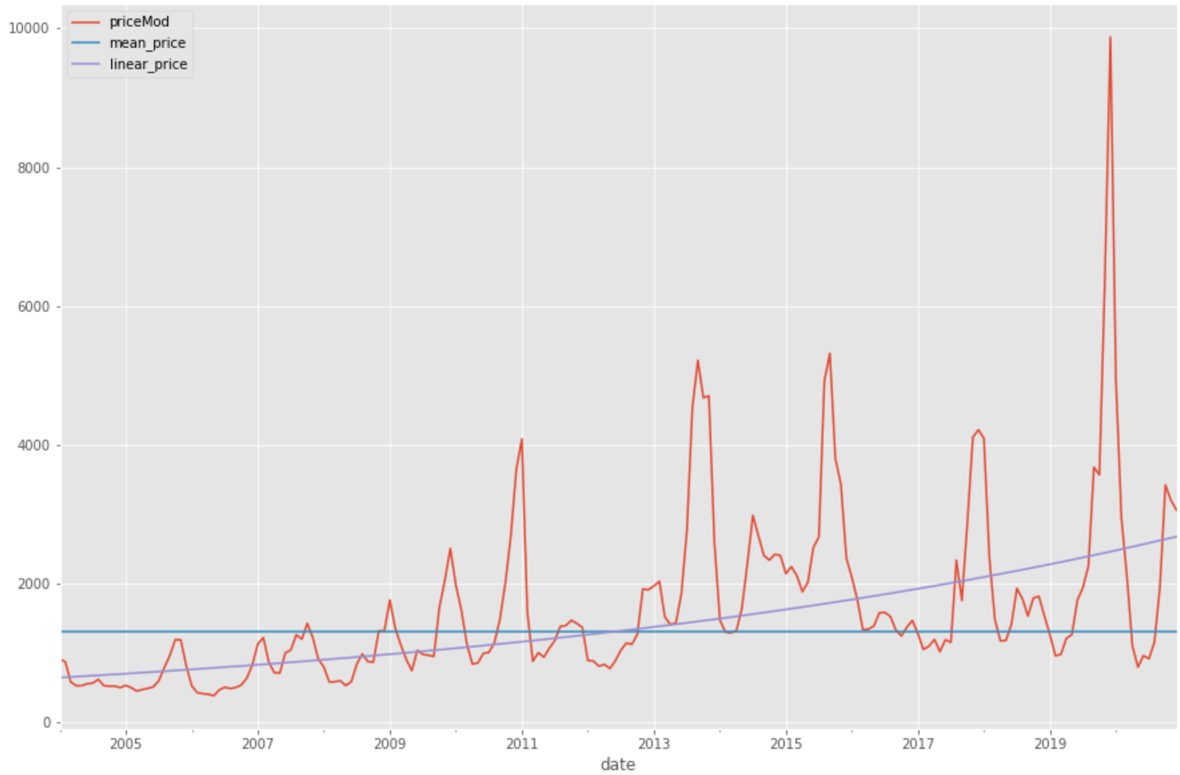
	Model	Actual	Forcast	RMSE
0	Mean Model	3060	1312.94	1271.69
1	Linear Model	3060	3074.81	1096.57

In [116]:

```
df.plot(kind="line", x="date", y=["priceMod", "mean_price","linear_price"])
```

Out[116]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x287224c2dc8>
```



In [117]:

```
linear_model_quant = smf.ols('log_priceMod ~ timeindex + np.log(quantity)', data = df).fit(
linear_model_quant.summary()
```

Out[117]:

OLS Regression Results

Dep. Variable:	log_priceMod	**R-squared:**	0.432
Model:	OLS	**Adj. R-squared:**	0.426
Method:	Least Squares	**F-statistic:**	76.32
Date:	Sun, 20 Dec 2020	**Prob (F-statistic):**	2.19e-25
Time:	16:55:18	**Log-Likelihood:**	-137.30
No. Observations:	204	**AIC:**	280.6
Df Residuals:	201	**BIC:**	290.6
Df Model:	2		
Covariance Type:	nonrobust		

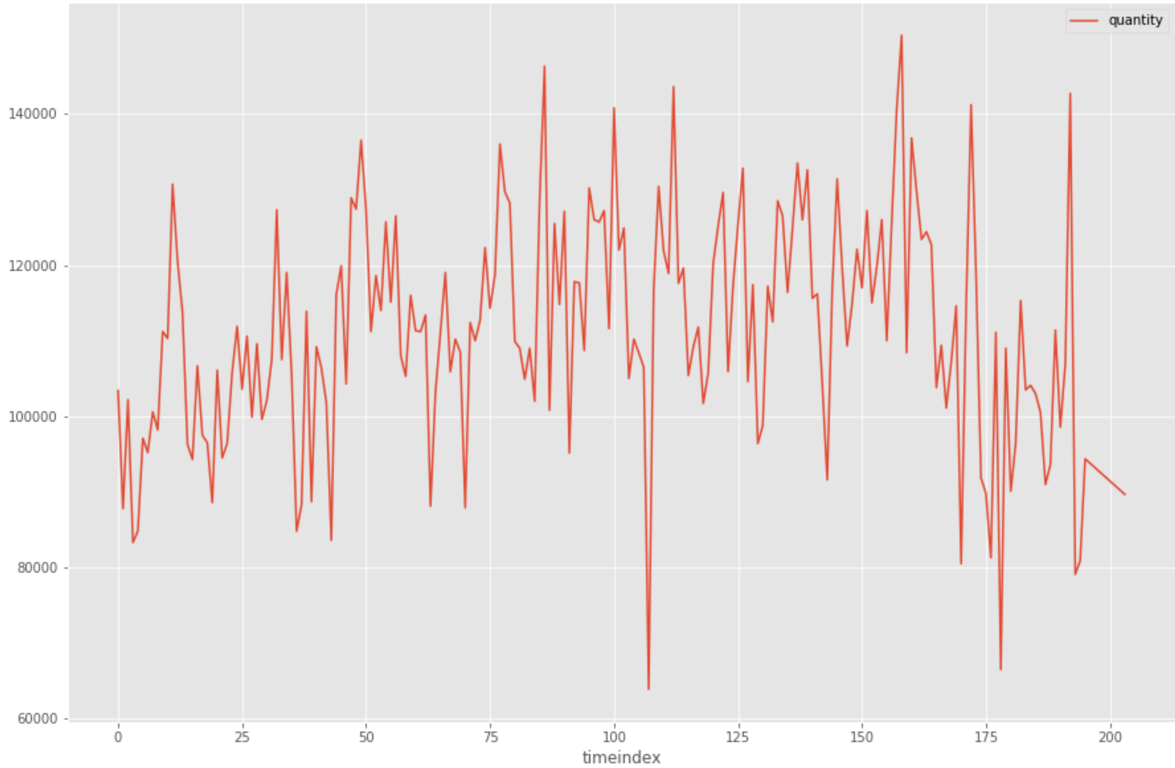
	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	7.4509	2.760	2.699	0.008	2.008	12.894
timeindex	0.0070	0.001	12.350	0.000	0.006	0.008
np.log(quantity)	-0.0847	0.238	-0.356	0.722	-0.554	0.384

Omnibus:	9.262	**Durbin-Watson:**	0.273
Prob(Omnibus):	0.010	**Jarque-Bera (JB):**	9.250
Skew:	0.506	**Prob(JB):**	0.00980
Kurtosis:	3.252	**Cond. No.**	9.75e+03

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 9.75e+03. This might indicate that there are
strong multicollinearity or other numerical problems.

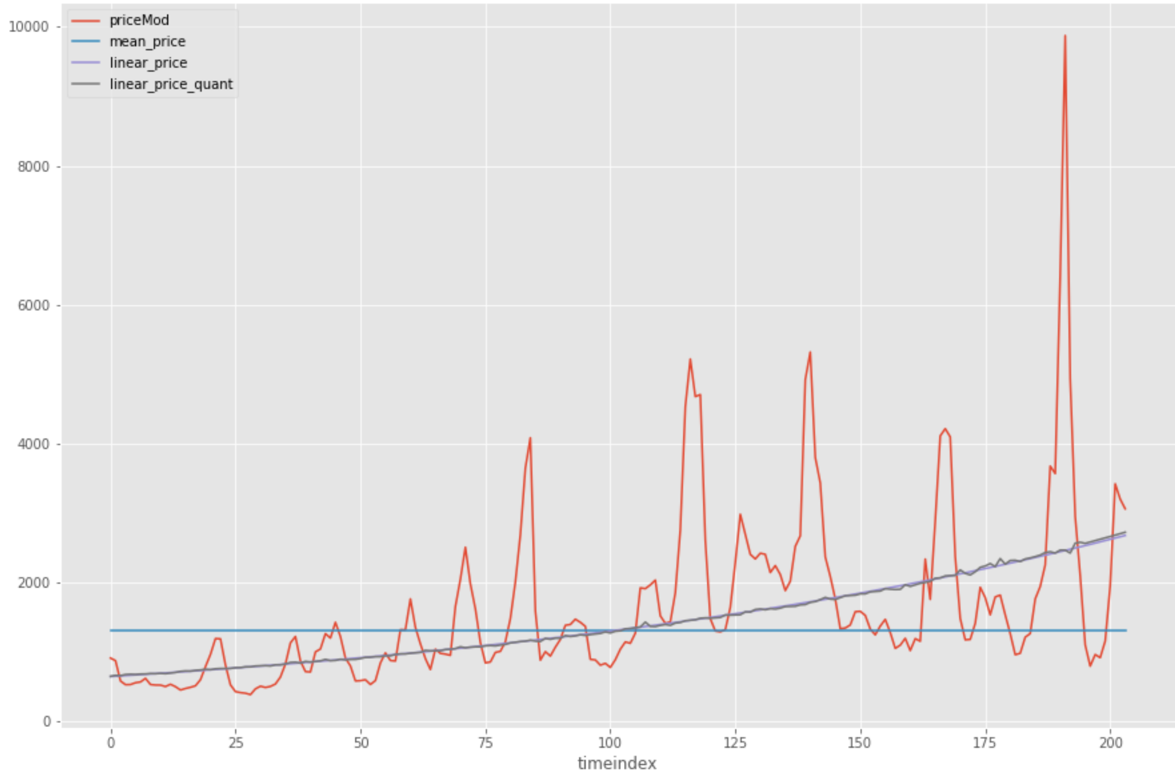
In [118]:

```
df["linear_price_quant"] = np.exp(linear_model_quant.predict())
df.plot(kind = "line", x="timeindex", y = "quantity")
plt.show()
```



In [119]:

```
df.plot(kind="line", x="timeindex", y = ["priceMod", "mean_price",
                                          "linear_price", "linear_price_quant"])
plt.show()
```



13 3. Random Walk Model

When faced with a time series that shows irregular growth, the best strategy may not be to try to directly predict the level of the series at each period (i.e., the quantity Yt). Instead, it may be better to try to predict the change that occurs from one period to the next (i.e., the quantity Yt - Yt-1).

That is, it may be better to look at the first difference of the series, to see if a predictable pattern can be found there. For purposes of one-period-ahead forecasting, it is just as good to predict the next change as to predict the next level of the series, since the predicted change can be added to the current level to yield a predicted level. The simplest case of such a model is one that always predicts that the next change will be zero, as if the series is equally likely to go up or down in the next period regardless of what it has done in the past.

There are two types of random walks

1. Random walk without drift (no constant or intercept)
2. Random walk with drift (with a constant term)

In [120]:

```
df["shift_log_priceMod"] = df.log_priceMod.shift()
df.head()
```

Out[120]:

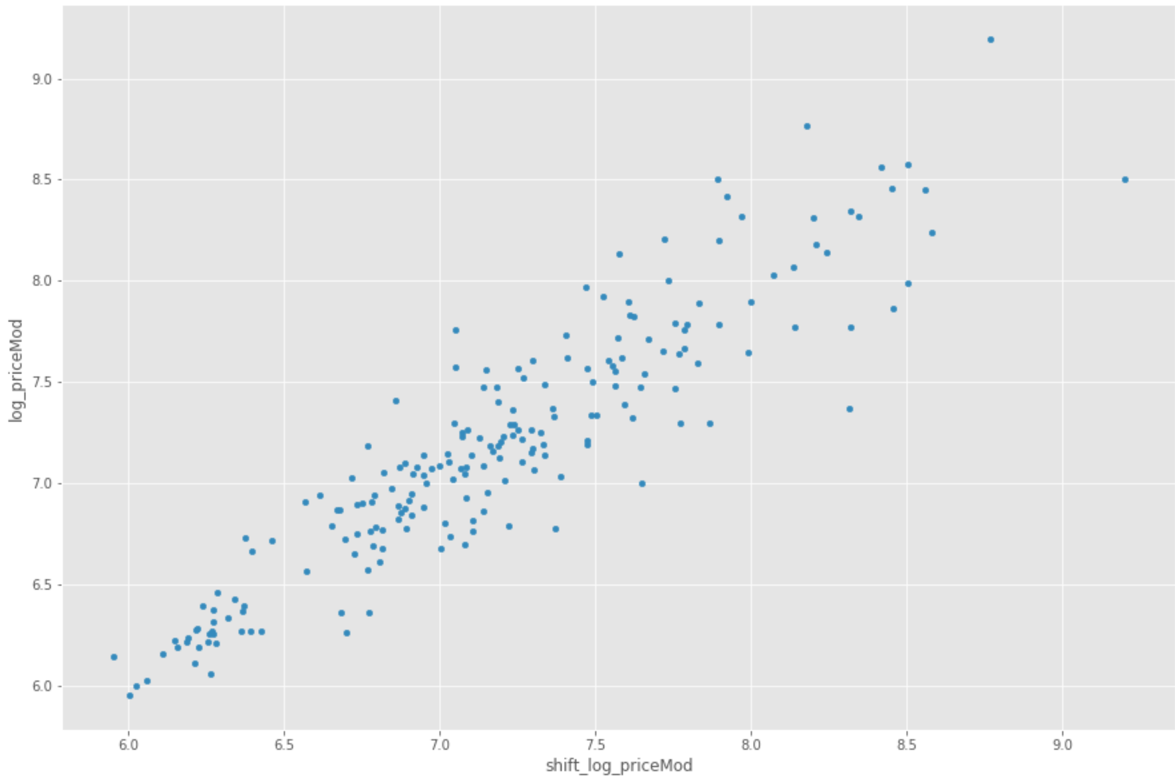
date	quantity	priceMod	date	log_priceMod	mean_price	timeindex	linear_price	linear_pric
2004-01	103400	910	2004-01-01	6.813445	1312.94077	0	644.143189	647
2004-02	87800	873	2004-02-01	6.771936	1312.94077	1	648.678259	660
2004-03	102180	580	2004-03-01	6.363028	1312.94077	2	653.245258	657
2004-04	83300	527	2004-04-01	6.267201	1312.94077	3	657.844411	673
2004-05	84850	529	2004-05-01	6.270988	1312.94077	4	662.475944	676

In [121]:

```
df.plot(kind="scatter", x="shift_log_priceMod", y ="log_priceMod", s=20 )
```

Out[121]:

<matplotlib.axes._subplots.AxesSubplot at 0x2871f284c88>

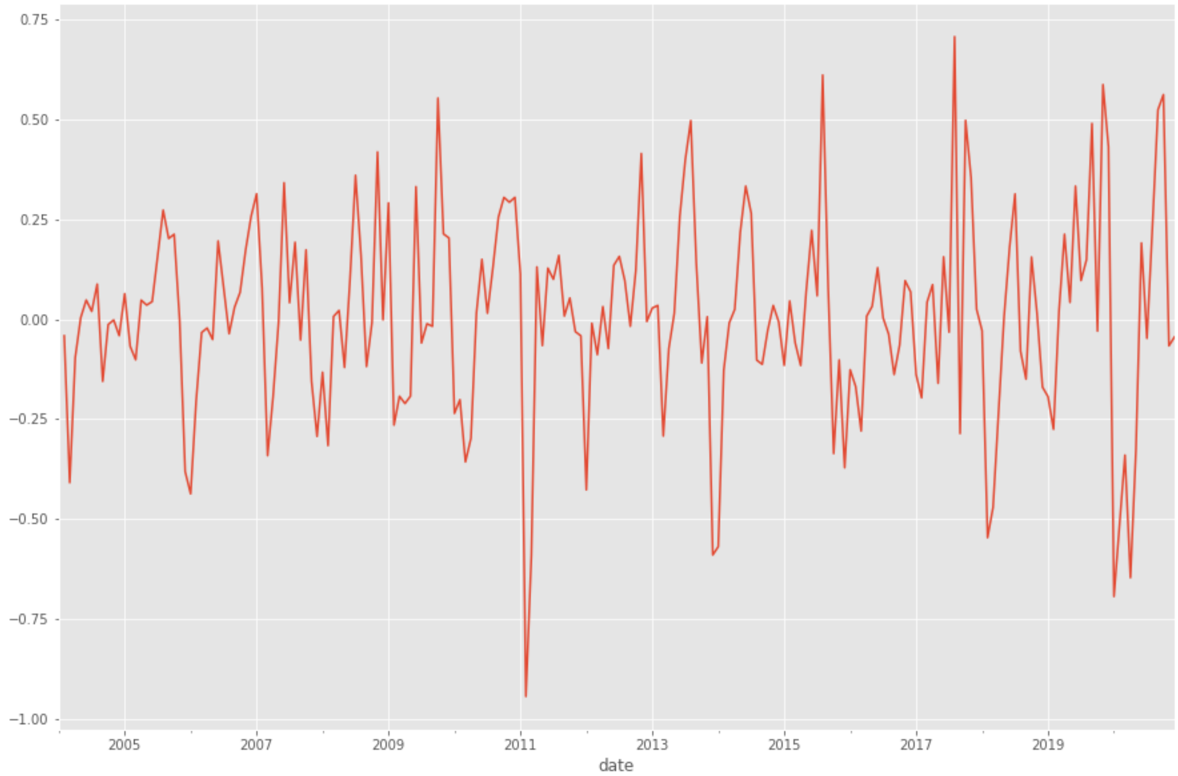


In [122]:

```
df["log_priceMod_diff"] = df.log_priceMod - df.shift_log_priceMod
df.log_priceMod_diff.plot()
```

Out[122]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2871c865988>
```



In [123]:

```
df["random_price"] = np.exp(df.shift_log_priceMod)
df.head()
```

Out[123]:

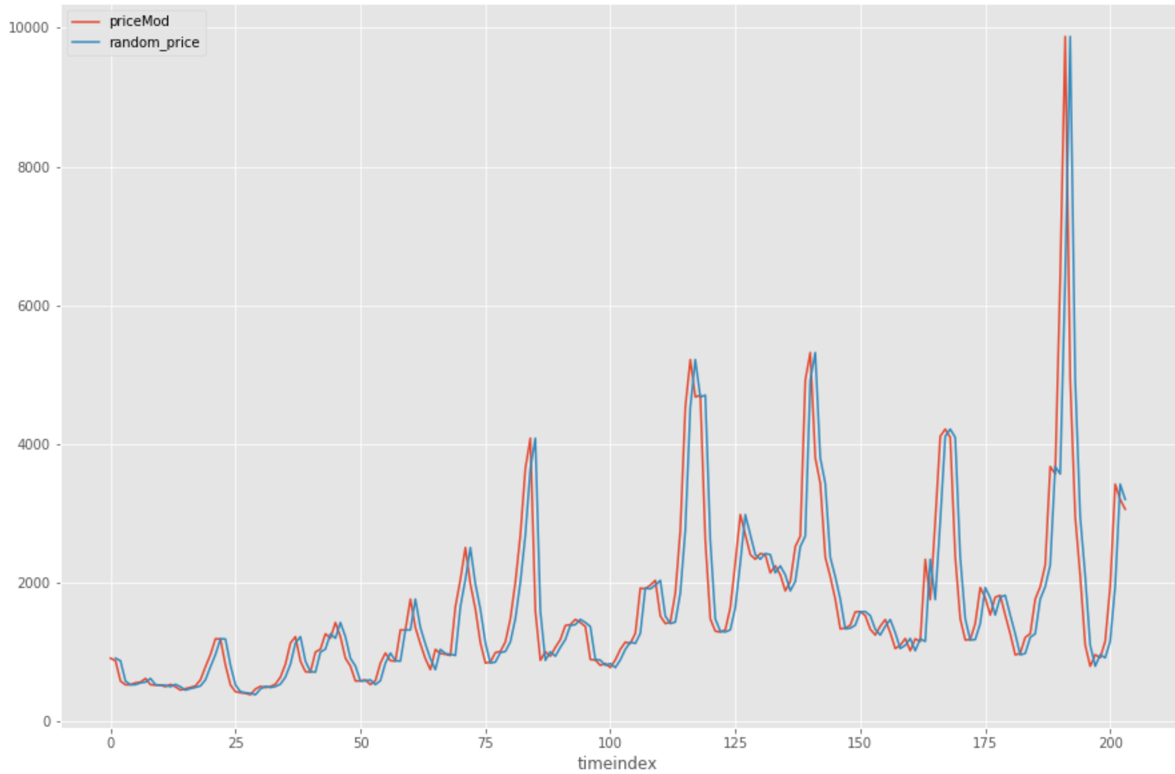
	quantity	priceMod	date	log_priceMod	mean_price	timeindex	linear_price	linear_pric
date								
2004-01	103400	910	2004-01-01	6.813445	1312.94077	0	644.143189	647
2004-02	87800	873	2004-02-01	6.771936	1312.94077	1	648.678259	660
2004-03	102180	580	2004-03-01	6.363028	1312.94077	2	653.245258	657
2004-04	83300	527	2004-04-01	6.267201	1312.94077	3	657.844411	673
2004-05	84850	529	2004-05-01	6.270988	1312.94077	4	662.475944	676

In [124]:

```
# lets compare random price and actual price
df.plot(kind="line", x="timeindex", y = ["priceMod", "random_price"])
```

Out[124]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x2871f538208>
```



In [125]:

```
# evaluate the random walk model
random_model_RMSE = RMSE(df.priceMod, df.random_price)
random_model_RMSE
```

Out[125]:

```
701.6611605586083
```

In [126]:

```
Result_df.loc[2,"Model"] = "Random Model"
Result_df.loc[2,"Actual"] = "3060"
Result_df.loc[2,"Forcast"] = np.exp(df.shift_log_priceMod[-1])
Result_df.loc[2,"RMSE"] = random_model_RMSE
Result_df
```

Out[126]:

	Model	Actual	Forcast	RMSE
0	Mean Model	3060	1312.94	1271.69
1	Linear Model	3060	3074.81	1096.57
2	Random Model	3060	3200	701.661

In [127]:

```
df.plot(kind="line", x="timeindex", y = ["priceMod", "mean_price", "linear_price", "random_
```

Out[127]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x287200af4c8>
```