In [1]:
```python
import numpy as np
import pandas as pd
import datetime
```

Feature Engineering

- Feature Engineering is the process of transforming data to increase the predictive performance of machine learning models.

Data normalization:

- Data Normalization could also be a typical practice in machine learning which consists of transforming numeric columns to a standard scale.

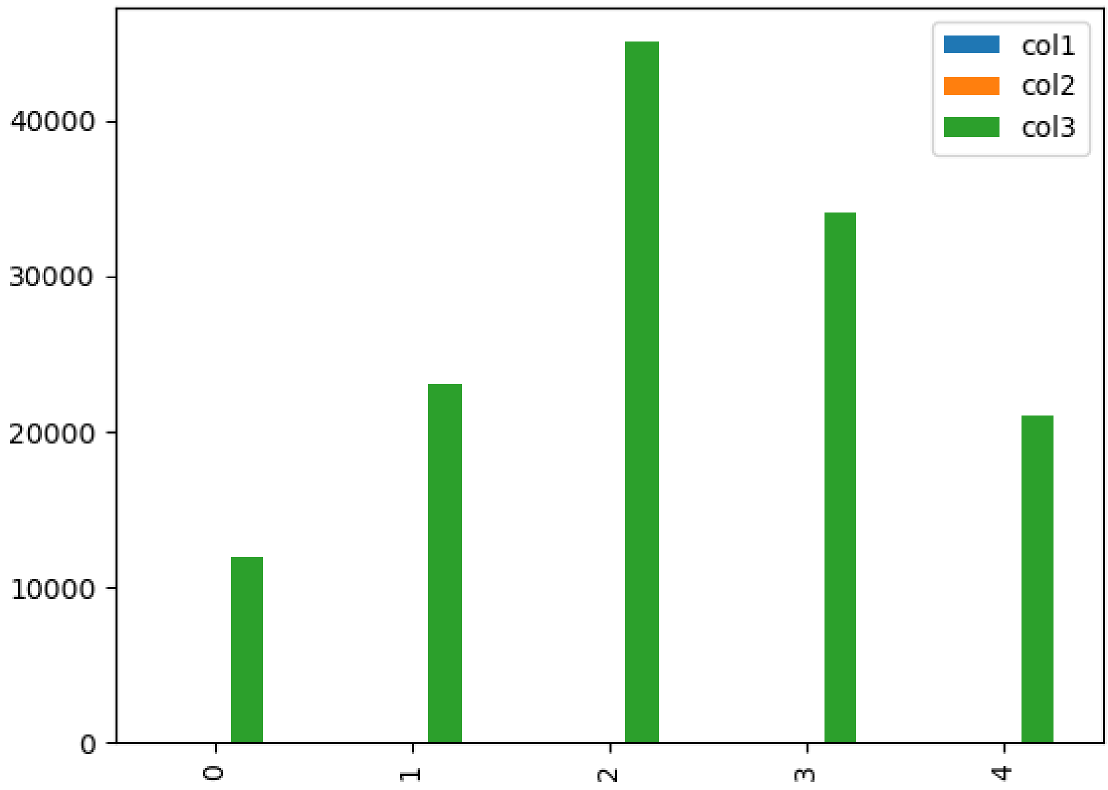
In [2]:
```python
data = {
  'col1': [1,2,3,4,5],
  'col2': [8,7,3,6,4],
  'col3': [12000, 23000, 45000, 34000, 21000]
}
df = pd.DataFrame(data)

print(df)
```

```
   col1  col2   col3
0     1     8  12000
1     2     7  23000
2     3     3  45000
3     4     6  34000
4     5     4  21000
```

In [3]:
```python
import matplotlib.pyplot as plt
df.plot(kind = 'bar')
# When plotting the graph, we can see that,
# col3 has dominated the graph and we cannot observe the others.
```

Out[3]: <Axes: >



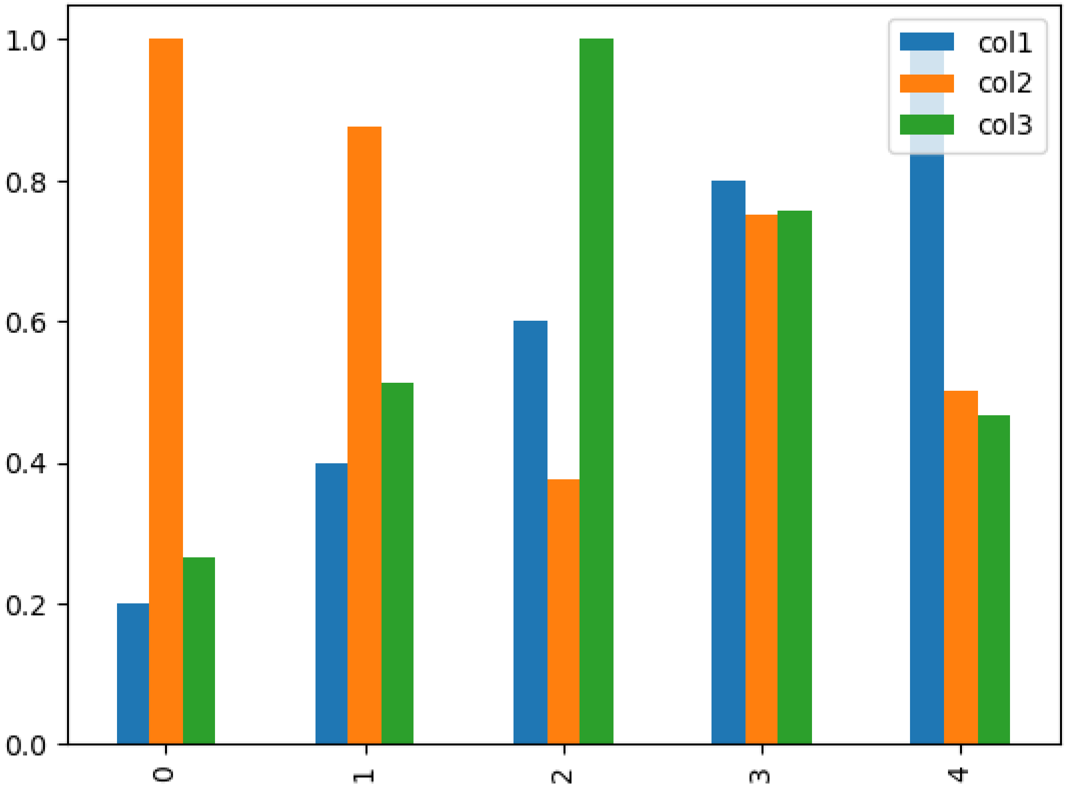
In [4]:
```python
# So, we can scale all column data in smaller ranges
for column in df.columns:
  df[column] = df[column]  / df[column].abs().max()

print(df)
# Now, we can see all the col values are closer to each other
```

```
   col1   col2      col3
0   0.2  1.000  0.266667
1   0.4  0.875  0.511111
2   0.6  0.375  1.000000
3   0.8  0.750  0.755556
4   1.0  0.500  0.466667
```

In [5]:
```python
import matplotlib.pyplot as plt
df.plot(kind = 'bar')
# Now, we can observe features from the graph.
```

Out[5]:
```
<Axes: >
```



Scaling:

- In cases where all the columns have a significant difference in their scales, are needed to be modified in such a way that all those values fall into the same scale. This process is called **Scaling**.

- There are two most common techniques of how to scale columns of Pandas dataframe –
  1. Min-Max Normalization
  2. Standardization.

In [6]:
```python
df = pd.read_csv('IRIS.csv')

print(df.head())
```

```
   sepal_length  sepal_width  petal_length  petal_width      species
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
```

1. min-max Normalization:

- Here, all the values are scaled in between the range of [0,1] where 0 is the minimum value and 1 is the maximum value.
- The formula for Min-Max Normalization is –

```
x_norm = (x - x_min) / (x_max - x_min)
```

In [7]:
```python
print(df.info())
# We can perform scaling on numeric data cols only, so for scaling purpose,
# have to drop the last col with "species" for this operation only
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
None
```

In [8]:
```python
new_df = df.drop('species', axis=1) # Have to use axis=1, for column

# Now, perform scaling on each value of each column
df_norm = ((new_df-new_df.min()) / (new_df.max()-new_df.min()))

# Now, merge the normalized df with the dropped species col
final_df = pd.concat((df_norm, df.species), axis=1)

print(final_df)
```

```
     sepal_length  sepal_width  petal_length  petal_width         species
0        0.222222     0.625000      0.067797     0.041667     Iris-setosa
1        0.166667     0.416667      0.067797     0.041667     Iris-setosa
2        0.111111     0.500000      0.050847     0.041667     Iris-setosa
3        0.083333     0.458333      0.084746     0.041667     Iris-setosa
4        0.194444     0.666667      0.067797     0.041667     Iris-setosa
..            ...          ...           ...          ...             ...
145      0.666667     0.416667      0.711864     0.916667  Iris-virginica
146      0.555556     0.208333      0.677966     0.750000  Iris-virginica
147      0.611111     0.416667      0.711864     0.791667  Iris-virginica
148      0.527778     0.583333      0.745763     0.916667  Iris-virginica
149      0.444444     0.416667      0.694915     0.708333  Iris-virginica

[150 rows x 5 columns]
```

In [9]:
```python
# Using MinMaxScaler from sklearn

from sklearn.preprocessing import MinMaxScaler
import pandas as pd

# Drop species col, having str data
new_df = df.drop('species', axis=1)

scaler = MinMaxScaler()

df_scaled = scaler.fit_transform(new_df.to_numpy())
df_scaled = pd.DataFrame(df_scaled, columns=[
  'sepal_length', 'sepal_width', 'petal_length', 'petal_width'])

print("Scaled Dataset Using MinMaxScaler")
df_scaled.head()
```

```
Scaled Dataset Using MinMaxScaler
```

Out[9]:

	sepal_length	sepal_width	petal_length	petal_width
0	0.222222	0.625000	0.067797	0.041667
1	0.166667	0.416667	0.067797	0.041667
2	0.111111	0.500000	0.050847	0.041667
3	0.083333	0.458333	0.084746	0.041667
4	0.194444	0.666667	0.067797	0.041667

2. Standardization:

- Standardization doesn't have any fixed minimum or maximum value.
- Here, the values of all the columns are scaled in such a way that they all have a mean equal to 0 and standard deviation equal to 1.
- This scaling technique works well with outliers.
- Thus, this technique is preferred if outliers are present in the dataset.

In [10]:
```python
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Drop species col, having str data
new_df = df.drop('species', axis=1)

std_scaler = StandardScaler()

df_scaled = std_scaler.fit_transform(new_df.to_numpy())
df_scaled = pd.DataFrame(df_scaled, columns=[
'sepal_length','sepal_width','petal_length','petal_width'])

print("Scaled Dataset Using StandardScaler")
df_scaled.head()
```

```
Scaled Dataset Using StandardScaler
```

Out[10]:

	sepal_length	sepal_width	petal_length	petal_width
0	-0.900681	1.032057	-1.341272	-1.312977
1	-1.143017	-0.124958	-1.341272	-1.312977
2	-1.385353	0.337848	-1.398138	-1.312977
3	-1.506521	0.106445	-1.284407	-1.312977
4	-1.021849	1.263460	-1.341272	-1.312977

Time Series Analysis and Resampling

- Pandas provide a different set of tools using which we can perform all the necessary tasks on date-time data.

Working with datetime data:

In [11]:
```python
#1: Create a dates dataframe

# Create dates dataframe with frequency
data = pd.date_range('1/1/2011', periods = 15, freq ='M')

data
```

Out[11]:
```
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-30',
               '2011-05-31', '2011-06-30', '2011-07-31', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-31',
               '2012-01-31', '2012-02-29', '2012-03-31'],
              dtype='datetime64[ns]', freq='M')
```

In [12]:
```
# 2: Current date and time

# x = pd.datetime.now()
# x.date, x.month, x.year

stamp = pd.Timestamp(datetime.datetime(2023, 8, 4))
stamp = pd.Timestamp(datetime.datetime.now())
print(stamp)

result = stamp.today()
print(result)
print(stamp.day_name())
print(stamp.day_of_week)
print(stamp.weekday())
print(stamp.hour)
print(stamp.day)
print(stamp.days_in_month)
print(stamp.date())
print(stamp.dayofyear)
print(stamp.daysinmonth)
print(stamp.minute)
```

```
2024-02-04 16:28:58.716884
2024-02-04 16:28:58.717888
Sunday
6
6
16
4
29
2024-02-04
35
29
28
```

In [13]:
```
# Create date and time with dataframe
rng = pd.DataFrame()
rng['date'] = pd.date_range(datetime.datetime.now(), periods = 72, freq ='H

# Print the dates in dd-mm-yy format
rng[:5]

# Create features for year, month, day, hour, and minute
rng['year'] = rng['date'].dt.year
rng['month'] = rng['date'].dt.month
rng['day'] = rng['date'].dt.day
rng['hour'] = rng['date'].dt.hour
rng['minute'] = rng['date'].dt.minute
rng['second'] = rng['date'].dt.second

# Print the dates divided into features
rng.head(3)
```

Out[13]:

	date	year	month	day	hour	minute	second
0	2024-02-04 16:28:58.727967	2024	2	4	16	28	58
1	2024-02-04 17:28:58.727967	2024	2	4	17	28	58
2	2024-02-04 18:28:58.727967	2024	2	4	18	28	58

Resampling and time-based indexing:

Handling time zones and date offsets:

- Dateoffsets are a standard kind of date increment used for a date range in Pandas.
- DateOffsets can be created to move dates forward a given number of valid dates.
- Pandas **tseries.offsets.DateOffset** is used to create standard kind of date increment used for a date range.
- **Syntax:**

  ```
  pandas.tseries.offsets.DateOffset(n=1, normalize=False, **kw
  ds)
  ```

  Parameters:

  - n : The number of time periods the offset represents.
  - normalize : Whether to round the result of a DateOffset addition down to the previous midnight.
  - level : int, str, default None
  - **kwds : Temporal parameter that add to or replace the offset value. Parameters that add to the offset (like Timedelta): years, months etc.
  - Returns : DateOffsets

In [14]:
```
# Creating Timestamp
ts = pd.Timestamp('2019-10-10 07:15:11')
print(ts)

# Create the DateOffset
do = pd.tseries.offsets.DateOffset(n = 2)
print(do)

# We can now add the DateOffset to any date, to increment the date
new_date = ts + do
print(new_date)
# Thus, the date would move forward by 2 days.
```

```
2019-10-10 07:15:11
<2 * DateOffsets>
2019-10-12 07:15:11
```

In [15]:
```
# Providing additional arguments to the DateOffset

today = pd.Timestamp(datetime.datetime.now())

# Create new DateOffset
date_off = pd.tseries.offsets.DateOffset(days=10, hours=3, minutes=15)
# Move the timestamp by 10 days, 3 hrs and 15 mins

future_date = today + date_off

print(today)
print(future_date)
```

```
2024-02-04 16:28:58.754354
2024-02-14 19:43:58.754354
```

Working with datetime data in pandas:

Datetime features can be divided into two categories:

- The first one: time moments in a period
- Second: the time passed since a particular period.
- These features can be very useful to understand the patterns in the data.

Divide a given date into features –

- **pandas.Series.dt.year** returns the year of the date time.
- **pandas.Series.dt.month** returns the month of the date time.
- **pandas.Series.dt.day** returns the day of the date time.
- **pandas.Series.dt.hour** returns the hour of the date time.
- **pandas.Series.dt.minute** returns the minute of the date time.

In [16]:

```
#3: Break date and time into separate features

# Create date and time with dataframe
rng = pd.DataFrame()
rng['date'] = pd.date_range(datetime.datetime.now().date(), periods = 72, f

# Print the dates in dd-mm-yy format
rng[:5]

# Create features for year, month, day, hour, and minute
rng['year'] = rng['date'].dt.year
rng['month'] = rng['date'].dt.month
rng['day'] = rng['date'].dt.day
rng['hour'] = rng['date'].dt.hour
rng['minute'] = rng['date'].dt.minute

# Print the dates divided into features
rng.head(10)
```

Out[16]:

	date	year	month	day	hour	minute
0	2024-02-04 00:00:00	2024	2	4	0	0
1	2024-02-04 01:00:00	2024	2	4	1	0
2	2024-02-04 02:00:00	2024	2	4	2	0
3	2024-02-04 03:00:00	2024	2	4	3	0
4	2024-02-04 04:00:00	2024	2	4	4	0
5	2024-02-04 05:00:00	2024	2	4	5	0
6	2024-02-04 06:00:00	2024	2	4	6	0
7	2024-02-04 07:00:00	2024	2	4	7	0
8	2024-02-04 08:00:00	2024	2	4	8	0
9	2024-02-04 09:00:00	2024	2	4	9	0

In [17]:
```
# Import dataset
data = pd.read_csv('COVID_19_Containment_measures_data.csv')

# print(data.loc[0])

sample = {'id': data['ID'], 'country': data['Country'], 'start': data['Date
          'end': data['Date end intended']}
df = pd.DataFrame(sample)

# print(df)
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1703 entries, 0 to 1702
Data columns (total 16 columns):
 #   Column                           Non-Null Count  Dtype
---  ------                           --------------  -----
 0   ID                               820 non-null    object
 1   Applies To                       29 non-null     object
 2   Country                          1675 non-null   object
 3   Date Start                       1639 non-null   object
 4   Date end intended                242 non-null    object
 5   Description of measure implemented 1640 non-null   object
 6   Exceptions                       41 non-null     object
 7   Implementing City                127 non-null    object
 8   Implementing State/Province      179 non-null    object
 9   Keywords                         1615 non-null   object
 10  Quantity                         302 non-null    float64
 11  Source                           1517 non-null   object
 12  Target city                      1 non-null      object
 13  Target country                   132 non-null    object
 14  Target region                    29 non-null     object
 15  Target state                     0 non-null      float64
dtypes: float64(2), object(14)
memory usage: 213.0+ KB
None
```

In [18]:
```
# Convert the Time column to datetime format
df['start'] = pd.to_datetime(df.start)
df['end'] = pd.to_datetime(df.end)

print(df.head())
```

```
    id         country      start end
0  163         Austria 2020-03-16 NaT
1  132         Germany 2020-02-01 NaT
2  578  United Kingdom 2020-03-20 NaT
3  372  United Kingdom 2020-03-16 NaT
4  357  United Kingdom 2020-03-16 NaT
```

In [19]:
```
df.dtypes
# Thus, shows that, start and end columns are converted to datetime type
```

Out[19]:
```
id                 object
country            object
start      datetime64[ns]
end        datetime64[ns]
dtype: object
```

In [20]:
```python
# Get details from the date type from the DF

# Get hour detail from time data
print(df.start.dt.hour.head())

# Get name of each date
# df.start.dt.weekday_name.head()


# Get ordinal day of the year
print(df.start.dt.dayofyear.head())
```

```
0    0.0
1    0.0
2    0.0
3    0.0
4    0.0
Name: start, dtype: float64
0    76.0
1    32.0
2    80.0
3    76.0
4    76.0
Name: start, dtype: float64
```

Resampling time series data (e.g., downsampling and upsampling):

- Pandas **dataframe.resample()** function is primarily used for time series data.
- A time series is a series of data points indexed (or listed or graphed) in time order.
- It is a Convenience method for frequency conversion and resampling of time series.
- Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

- **Syntax :**

```
  DataFrame.resample(rule, how=None, axis=0, fill_method=None,
closed=None, label=None, convention='start', kind=None, loffse
t=None, limit=None, base=0, on=None, level=None)
```

- **Parameters :**
  - **rule :** the offset string or object representing target conversion
  - **axis :** int, optional, default 0
  - **closed :** {'right', 'left'}
  - **label :** {'right', 'left'}
  - **convention :** For PeriodIndex only, controls whether to use the start or end of rule
  - **loffset :** Adjust the resampled time labels
  - **base :** For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0.
  - **on :** For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.
  - **level :** For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

- Resampling generates a unique sampling distribution on the basis of the actual data.
- Most commonly used time series frequency are –
  - **W :** weekly frequency
  - **M :** month end frequency
  - **SM :** semi-month end frequency (15th and end of month)
  - **Q :** quarter end frequency

In [21]:
```
data = pd.read_csv('COVID_19_Containment_measures_data.csv')
sample = {'id': data['ID'], 'country': data['Country'], 'start_date': data|
          'end': data['Date end intended'], 'quantity': data['Quantity']}
covid_df = pd.DataFrame(sample)
covid_df.dropna(inplace=True)
covid_df.to_csv('covid_data.csv')
```

In [22]:
```
# By default the "start_date" column was in string format,
# we need to convert it into date-time format

# parse_dates =["start_date"], converts the "start_date"
# column to date-time format. We know that
# resampling works with time-series data only
# so convert "start_date" column to index

# index_col ="start_date", makes "start_date" column, the index of the data
df = pd.read_csv('covid_data.csv', parse_dates =["start_date"], index_col =

# Printing the dataframe
df
```

Out[22]:

start_date	Unnamed: 0	id	country	end	quantity
2020-03-16	378	254	Finland	Apr 13, 2020	10.0
2020-03-16	414	292	Lithuania	Mar 30, 2020	50.0
2020-03-13	452	345	Latvia	Apr 14, 2020	200.0
2020-03-15	453	342	Netherlands	Apr 06, 2020	30.0
2020-03-15	468	364	Ireland	Mar 29, 2020	10.0
2020-03-16	503	396	United States	Mar 30, 2020	1000.0
2020-03-08	529	447	Romania	Mar 31, 2020	1000.0
2020-03-11	530	436	Romania	Mar 31, 2020	100.0
2020-03-12	533	442	Norway	Mar 26, 2020	50.0
2020-03-13	544	459	Germany	Apr 20, 2020	90.0
2020-03-10	599	588	Russia	Apr 10, 2020	5000.0
2020-03-16	613	583	Russia	Apr 10, 2020	50.0
2020-03-13	615	574	Russia	Mar 20, 2020	1000.0
2020-03-16	632	618	US: Oregon	Apr 16, 2020	25.0
2020-03-18	940	861	Denmark	Apr 13, 2020	10.0

In [23]:
```python
# Resampling the time series data based on weekly frequency
# we apply it on stock open price 'W' indicates week
weekly_resampled_data = df.quantity.resample('W').mean()

# find the mean opening price of each week
# for each week over the period
weekly_resampled_data
```

Out[23]:
```
start_date
2020-03-08    1000.000000
2020-03-15     810.000000
2020-03-22     190.833333
Freq: W-SUN, Name: quantity, dtype: float64
```

Working with time-based data in pandas (e.g., datetime index):

In [24]:
```python
# index_col ="start_date", makes "start_date" column, the index of the data
df = pd.read_csv('covid_data.csv', parse_dates =["start_date"], index_col =

df['end'] = pd.to_datetime(df.end)
# Printing the dataframe
df
```

Out[24]:

	Unnamed: 0	id	country	end	quantity
start_date					
2020-03-16	378	254	Finland	2020-04-13	10.0
2020-03-16	414	292	Lithuania	2020-03-30	50.0
2020-03-13	452	345	Latvia	2020-04-14	200.0
2020-03-15	453	342	Netherlands	2020-04-06	30.0
2020-03-15	468	364	Ireland	2020-03-29	10.0
2020-03-16	503	396	United States	2020-03-30	1000.0
2020-03-08	529	447	Romania	2020-03-31	1000.0
2020-03-11	530	436	Romania	2020-03-31	100.0
2020-03-12	533	442	Norway	2020-03-26	50.0
2020-03-13	544	459	Germany	2020-04-20	90.0
2020-03-10	599	588	Russia	2020-04-10	5000.0
2020-03-16	613	583	Russia	2020-04-10	50.0
2020-03-13	615	574	Russia	2020-03-20	1000.0
2020-03-16	632	618	US: Oregon	2020-04-16	25.0
2020-03-18	940	861	Denmark	2020-04-13	10.0

In [ ]: