```python
import pandas as pd
import datetime
from pandas_datareader import data as pdr
import fix_yahoo_finance as yf
import matplotlib.pyplot as plt


# Compute the Bollinger Bands
def BBANDS(data, ndays):
    data_close = data['Close']
    MA = pd.Series(data_close.rolling(window=ndays, center=False).mean())
    SD = pd.Series(data_close.rolling(window=ndays, center=False).std())
    b1 = MA + (2 * SD)
    B1 = pd.Series(b1, name='Upper BollingerBand')
    data = data.join(B1)

    b2 = MA - (2 * SD)
    B2 = pd.Series(b2, name='Lower BollingerBand')
    data = data.join(B2)

    return data


# Commodity Channel Index
def CCI(data, ndays):
 TP = (data['High'] + data['Low'] + data['Close']) / 3
 #print TP.rolling(window=ndays,center=False).mean()
 CCI = pd.Series((TP - TP.rolling(window=ndays,center=False).mean()) / (0.015 * TP.rolling(window=ndays,center=False).std()),
 name = 'CCI')
 data = data.join(CCI)
 return data


# Ease of Movement
def EVM(data, ndays):
 dm = ((data['High'] + data['Low'])/2) - ((data['High'].shift(1) + data['Low'].shift(1))/2)
 br = (data['Volume'] / 100000000) / ((data['High'] - data['Low']))
 EVM = dm / br
 EVM_MA = pd.Series(EVM.rolling(window=ndays,center=False).mean(), name = 'EVM')
 data = data.join(EVM_MA)
 return data


```

```
# Force Index
def ForceIndex(data, ndays):
 FI = pd.Series(data['Close'].diff(ndays) * data['Volume'],
 name = 'ForceIndex')
 data = data.join(FI)
 return data


# Simple Moving Average
def SMA(data, ndays):
 data_close = data["Close"]
 SMA = pd.Series(data_close.rolling(window=ndays,center=
False).mean(), name = 'SMA')
 data = data.join(SMA)
 return data


# Exponentially-weighted Moving Average
def EWMA(data, ndays):
 data_close = data["Close"]
 EMA = pd.Series(data_close.ewm(ignore_na=False,span=ndays,
min_periods=ndays-1,adjust=True).mean(),
 name = 'EWMA_' + str(ndays))
 data = data.join(EMA)
 return data


# Rate of Change (ROC)
def ROC(data,n):
 N = data['Close'].diff(n)
 D = data['Close'].shift(n)
 ROC = pd.Series(N/D,name='Rate of Change')
 data = data.join(ROC)
 return data




##############
####from Bruno Franca and Peter Bakker, I refer from https
://www.quantopian.com/posts/technical-analysis-indicators-
without-talib-code
#############


#Moving Average
def MA(df, n):
    MA = pd.Series(pd.rolling_mean(df['Close'], n), name =
'MA_' + str(n))
    df = df.join(MA)
    return df

#Exponential Moving Average
def EMA(df, n):
    EMA = pd.Series(pd.ewma(df['Close'], span = n, min_periods = n - 1), name = 'EMA_' + str(n))
    df = df.join(EMA)
    return df

#Momentum
def MOM(df, n):
    M = pd.Series(df['Close'].diff(n), name = 'Momentum_' + str(n))
    df = df.join(M)
    return df

#Rate of Change
def ROC(df, n):
    M = df['Close'].diff(n - 1)
    N = df['Close'].shift(n - 1)
    ROC = pd.Series(M / N, name = 'ROC_' + str(n))
    df = df.join(ROC)
    return df

#Average True Range
def ATR(df, n):
    i = 0
    TR_l = [0]
    while i < df.index[-1]:
        TR = max(df.get_value(i + 1, 'High'), df.get_value(i, 'Close')) - min(df.get_value(i + 1, 'Low'), df.get_value(i, 'Close'))
        TR_l.append(TR)
        i = i + 1
    TR_s = pd.Series(TR_l)
    ATR = pd.Series(pd.ewma(TR_s, span = n, min_periods = n), name = 'ATR_' + str(n))
    df = df.join(ATR)
    return df

# #Bollinger Bands
# def BBANDS(df, n):
#     MA = pd.Series(pd.rolling_mean(df['Close'], n))
#     MSD = pd.Series(pd.rolling_std(df['Close'], n))
#     b1 = 4 * MSD / MA
#     B1 = pd.Series(b1, name = 'BollingerB_' + str(n))
#     df = df.join(B1)
#     b2 = (df['Close'] - MA + 2 * MSD) / (4 * MSD)
#     B2 = pd.Series(b2, name = 'Bollinger%b_' + str(n))
#     df = df.join(B2)
#     return df

#Pivot Points, Supports and Resistances
def PPSR(df):
    PP = pd.Series((df['High'] + df['Low'] + df['Close']) / 3)
    R1 = pd.Series(2 * PP - df['Low'])
    S1 = pd.Series(2 * PP - df['High'])
    R2 = pd.Series(PP + df['High'] - df['Low'])
    S2 = pd.Series(PP - df['High'] + df['Low'])
    R3 = pd.Series(df['High'] + 2 * (PP - df['Low']))
    S3 = pd.Series(df['Low'] - 2 * (df['High'] - PP))
    psr = {'PP':PP, 'R1':R1, 'S1':S1, 'R2':R2, 'S2':S2, 'R3':R3, 'S3':S3}
    PSR = pd.DataFrame(psr)
    df = df.join(PSR)
    return df

#Stochastic oscillator %K
def STOK(df):
    SOk = pd.Series((df['Close'] - df['Low']) / (df['High'] - df['Low']), name = 'SO%k')
    df = df.join(SOk)
    return df

#Stochastic oscillator %D
def STO(df, n):
    SOk = pd.Series((df['Close'] - df['Low']) / (df['High'] - df['Low']), name = 'SO%k')
    SOd = pd.Series(SOk.ewm(ignore_na=False,span=n, min_periods=n-1,adjust=True).mean() , name = 'SO%d_' + str(n))
    df = df.join(SOd)
    return df

#Trix
def TRIX(df, n):
    EX1 = pd.ewma(df['Close'], span = n, min_periods = n - 1)
    EX2 = pd.ewma(EX1, span = n, min_periods = n - 1)
    EX3 = pd.ewma(EX2, span = n, min_periods = n - 1)
    i = 0
    ROC_l = [0]
    while i + 1 <= df.index[-1]:
        ROC = (EX3[i + 1] - EX3[i]) / EX3[i]
        ROC_l.append(ROC)
        i = i + 1
    Trix = pd.Series(ROC_l, name = 'Trix_' + str(n))
    df = df.join(Trix)
    return df
```

```python

#Average Directional Movement Index
def ADX(df, n, n_ADX):
    i = 0
    UpI = []
    DoI = []
    while i + 1 <= df.index[-1]:
        UpMove = df.get_value(i + 1, 'High') - df.get_value(i, 'High')
        DoMove = df.get_value(i, 'Low') - df.get_value(i + 1, 'Low')
        if UpMove > DoMove and UpMove > 0:
            UpD = UpMove
        else: UpD = 0
        UpI.append(UpD)
        if DoMove > UpMove and DoMove > 0:
            DoD = DoMove
        else: DoD = 0
        DoI.append(DoD)
        i = i + 1
    i = 0
    TR_l = [0]
    while i < df.index[-1]:
        TR = max(df.get_value(i + 1, 'High'), df.get_value(i, 'Close')) - min(df.get_value(i + 1, 'Low'), df.get_value(i, 'Close'))
        TR_l.append(TR)
        i = i + 1
    TR_s = pd.Series(TR_l)
    ATR = pd.Series(pd.ewma(TR_s, span = n, min_periods = n))
    UpI = pd.Series(UpI)
    DoI = pd.Series(DoI)
    PosDI = pd.Series(pd.ewma(UpI, span = n, min_periods = n - 1) / ATR)
    NegDI = pd.Series(pd.ewma(DoI, span = n, min_periods = n - 1) / ATR)
    ADX = pd.Series(pd.ewma(abs(PosDI - NegDI) / (PosDI + NegDI), span = n_ADX, min_periods = n_ADX - 1), name = 'ADX_' + str(n) + '_' + str(n_ADX))
    df = df.join(ADX)
    return df

#MACD, MACD Signal and MACD difference
def MACD(df, n_fast, n_slow):
    data_close = df['Close']
    EMAfast = pd.Series(data_close.ewm(span=n_fast, min_periods=n_slow-1,adjust=True,ignore_na=False).mean())
    EMAslow = pd.Series(data_close.ewm(span=n_slow, min_periods=n_slow-1,adjust=True,ignore_na=False).mean())
    MACD = pd.Series(EMAfast - EMAslow, name = 'MACD_' + str(n_fast) + '_' + str(n_slow))
    MACDsign = pd.Series(MACD.ewm(span=9, min_periods=8, adjust=True,ignore_na=False).mean(), name = 'MACDsign_' + str(n_fast) + '_' + str(n_slow))
    MACDdiff = pd.Series(MACD - MACDsign, name = 'MACDdiff_' + str(n_fast) + '_' + str(n_slow))
    df = df.join(MACD)
    df = df.join(MACDsign)
    df = df.join(MACDdiff)
    return df



#Mass Index
def MassI(df):
    Range = df['High'] - df['Low']
    EX1 = pd.ewma(Range, span = 9, min_periods = 8)
    EX2 = pd.ewma(EX1, span = 9, min_periods = 8)
    Mass = EX1 / EX2
    MassI = pd.Series(pd.rolling_sum(Mass, 25), name = 'Mass Index')
    df = df.join(MassI)
    return df

#Vortex Indicator: http://www.vortexindicator.com/VFX_VORTEX.PDF
def Vortex(df, n):
    i = 0
    TR = [0]
    while i < df.index[-1]:
        Range = max(df.get_value(i + 1, 'High'), df.get_value(i, 'Close')) - min(df.get_value(i + 1, 'Low'), df.get_value(i, 'Close'))
        TR.append(Range)
        i = i + 1
    i = 0
    VM = [0]
    while i < df.index[-1]:
        Range = abs(df.get_value(i + 1, 'High') - df.get_value(i, 'Low')) - abs(df.get_value(i + 1, 'Low') - df.get_value(i, 'High'))
        VM.append(Range)
        i = i + 1
    VI = pd.Series(pd.rolling_sum(pd.Series(VM), n) / pd.rolling_sum(pd.Series(TR), n), name = 'Vortex_' + str(n))
    df = df.join(VI)
    return df


```

```python



#KST Oscillator
def KST(df, r1, r2, r3, r4, n1, n2, n3, n4):
    M = df['Close'].diff(r1 - 1)
    N = df['Close'].shift(r1 - 1)
    ROC1 = M / N
    M = df['Close'].diff(r2 - 1)
    N = df['Close'].shift(r2 - 1)
    ROC2 = M / N
    M = df['Close'].diff(r3 - 1)
    N = df['Close'].shift(r3 - 1)
    ROC3 = M / N
    M = df['Close'].diff(r4 - 1)
    N = df['Close'].shift(r4 - 1)
    ROC4 = M / N
    KST = pd.Series(pd.rolling_sum(ROC1, n1) + pd.rolling_sum(ROC2, n2) * 2 + pd.rolling_sum(ROC3, n3) * 3 + pd.rolling_sum(ROC4, n4) * 4, name = 'KST_' + str(r1) + '_' + str(r2) + '_' + str(r3) + '_' + str(r4) + '_' + str(n1) + '_' + str(n2) + '_' + str(n3) + '_' + str(n4))
    df = df.join(KST)
    return df

#Relative Strength Index touble!!
def RSI(df, n):
    i = 0
    UpI = [0]
    DoI = [0]

    for i in range(len(df.index)-1):
        # print range(len(df.index)-1)[-1]
        # print i
        UpMove = df.loc[df.index[i+1]]['High'] - df.loc[df.index[i]]['High']
        DoMove = df.loc[df.index[i+1]]['Low'] - df.loc[df.index[i]]['Low']
        if UpMove > DoMove and UpMove > 0:
            UpD = UpMove
        else: UpD = 0
        UpI.append(UpD)
        if DoMove > UpMove and DoMove > 0:
            DoD = DoMove
        else: DoD = 0
        DoI.append(DoD)

    UpI = pd.Series(UpI)
    DoI = pd.Series(DoI)
    PosDI = pd.Series(UpI.ewm(span=n, min_periods=n-1,
adjust=True,ignore_na=True).mean())
    NegDI = pd.Series(DoI.ewm(span=n, min_periods=n-1, adjust=True,ignore_na=True).mean())
    RSI = pd.Series(PosDI / (PosDI + NegDI), name = 'RSI_' + str(n))
    df = df.join(RSI)
    return df

#True Strength Index
def TSI(df, r, s):
    M = pd.Series(df['Close'].diff(1))
    aM = abs(M)
    EMA1 = pd.Series(pd.ewma(M, span = r, min_periods = r - 1))
    aEMA1 = pd.Series(pd.ewma(aM, span = r, min_periods = r - 1))
    EMA2 = pd.Series(pd.ewma(EMA1, span = s, min_periods = s - 1))
    aEMA2 = pd.Series(pd.ewma(aEMA1, span = s, min_periods = s - 1))
    TSI = pd.Series(EMA2 / aEMA2, name = 'TSI_' + str(r) + '_' + str(s))
    df = df.join(TSI)
    return df

#Accumulation/Distribution
def ACCDIST(df, n):
    ad = (2 * df['Close'] - df['High'] - df['Low']) / (df['High'] - df['Low']) * df['Volume']
    M = ad.diff(n - 1)
    N = ad.shift(n - 1)
    ROC = M / N
    AD = pd.Series(ROC, name = 'Acc/Dist_ROC_' + str(n))
    df = df.join(AD)
    return df

#Chaikin Oscillator
def Chaikin(df):
    ad = (2 * df['Close'] - df['High'] - df['Low']) / (df['High'] - df['Low']) * df['Volume']
    Chaikin = pd.Series(ad.ewm(ignore_na=False,span=3, min_periods=3-1,adjust=True).mean() - ad.ewm(ignore_na=False,span=10,min_periods=10-1,adjust=True).mean(), name = 'Chaikin')
    df = df.join(Chaikin)
    return df

#Money Flow Index and Ratio
def MFI(df, n):
    PP = (df['High'] + df['Low'] + df['Close']) / 3
    i = 0
    PosMF = [0]
    while i < df.index[-1]:
        if PP[i + 1] > PP[i]:
            PosMF.append(PP[i + 1] * df.get_value(i + 1, 'Volume'))
        else:
            PosMF.append(0)
        i = i + 1
    PosMF = pd.Series(PosMF)
    TotMF = PP * df['Volume']
    MFR = pd.Series(PosMF / TotMF)
    MFI = pd.Series(pd.rolling_mean(MFR, n), name = 'MFI_' + str(n))
    df = df.join(MFI)
    return df

#On-balance Volume
def OBV(df, n):
    i = 0
    OBV = [0]
    while i < df.index[-1]:
        if df.get_value(i + 1, 'Close') - df.get_value(i, 'Close') > 0:
            OBV.append(df.get_value(i + 1, 'Volume'))
        if df.get_value(i + 1, 'Close') - df.get_value(i, 'Close') == 0:
            OBV.append(0)
        if df.get_value(i + 1, 'Close') - df.get_value(i, 'Close') < 0:
            OBV.append(-df.get_value(i + 1, 'Volume'))
        i = i + 1
    OBV = pd.Series(OBV)
    OBV_ma = pd.Series(pd.rolling_mean(OBV, n), name = 'OBV_' + str(n))
    df = df.join(OBV_ma)
    return df

#Force Index
def FORCE(df, n):
    F = pd.Series(df['Close'].diff(n) * df['Volume'].diff(n), name = 'Force_' + str(n))
    df = df.join(F)
    return df

#Ease of Movement
def EOM(df, n):
    EoM = (df['High'].diff(1) + df['Low'].diff(1)) * (df['High'] - df['Low']) / (2 * df['Volume'])
    Eom_ma = pd.Series(pd.rolling_mean(EoM, n), name = '
EoM_' + str(n))
    df = df.join(Eom_ma)
    return df

# #Commodity Channel Index
# def CCI(df, n):
#     PP = (df['High'] + df['Low'] + df['Close']) / 3
#     CCI = pd.Series((PP - pd.rolling_mean(PP, n)) / pd.rolling_std(PP, n), name = 'CCI_' + str(n))
#     df = df.join(CCI)
#     return df

#Coppock Curve
def COPP(df, n):
    M = df['Close'].diff(int(n * 11 / 10) - 1)
    N = df['Close'].shift(int(n * 11 / 10) - 1)
    ROC1 = M / N
    M = df['Close'].diff(int(n * 14 / 10) - 1)
    N = df['Close'].shift(int(n * 14 / 10) - 1)
    ROC2 = M / N
    Copp = pd.Series(pd.ewma(ROC1 + ROC2, span = n, min_periods = n), name = 'Copp_' + str(n))
    df = df.join(Copp)
    return df

#Keltner Channel
def KELCH(df, n):
    KelChM = pd.Series(pd.rolling_mean((df['High'] + df['Low'] + df['Close']) / 3, n), name = 'KelChM_' + str(n))
    KelChU = pd.Series(pd.rolling_mean((4 * df['High'] - 2 * df['Low'] + df['Close']) / 3, n), name = 'KelChU_' + str(n))
    KelChD = pd.Series(pd.rolling_mean((-2 * df['High'] + 4 * df['Low'] + df['Close']) / 3, n), name = 'KelChD_' + str(n))
    df = df.join(KelChM)
    df = df.join(KelChU)
    df = df.join(KelChD)
    return df

#Ultimate Oscillator
def ULTOSC(df):
    i = 0
    TR_l = [0]
    BP_l = [0]
    while i < df.index[-1]:
        TR = max(df.get_value(i + 1, 'High'), df.get_value(i, 'Close')) - min(df.get_value(i + 1, 'Low'), df.get_value(i, 'Close'))
        TR_l.append(TR)
        BP = df.get_value(i + 1, 'Close') - min(df.get_value(i + 1, 'Low'), df.get_value(i, 'Close'))
        BP_l.append(BP)
        i = i + 1
    UltO = pd.Series((4 * pd.rolling_sum(pd.Series(BP_l), 7) / pd.rolling_sum(pd.Series(TR_l), 7)) + (2 * pd.rolling_sum(pd.Series(BP_l), 14) / pd.rolling_sum(pd.Series(TR_l), 14)) + (pd.rolling_sum(pd.Series(BP_l), 28) / pd.rolling_sum(pd.Series(TR_l), 28)), name = 'Ultimate_Osc')
    df = df.join(UltO)
    return df

#Donchian Channel
def DONCH(df, n):
    i = 0
    DC_l = []
    while i < n - 1:
        DC_l.append(0)
        i = i + 1
    i = 0
    while i + n - 1 < df.index[-1]:
        DC = max(df['High'].ix[i:i + n - 1]) - min(df['Low'].ix[i:i + n - 1])
        DC_l.append(DC)
        i = i + 1
    DonCh = pd.Series(DC_l, name = 'Donchian_' + str(n))
    DonCh = DonCh.shift(n - 1)
    df = df.join(DonCh)
    return df

#Standard Deviation
def STDDEV(df, n):
    df = df.join(pd.Series(pd.rolling_std(df['Close'], n), name = 'STD_' + str(n)))
    return df
```