

This member-only story is on us. [Upgrade](#) to access all of Medium.

◆ Member-only story

Algorithmic Trading with the Know Sure Thing indicator in Python

A complete guide on using and implementing the KST indicator in Python



Nikhil Adithyan · Follow

Published in CodeX · 20 min read · May 30, 2021

225

3



...

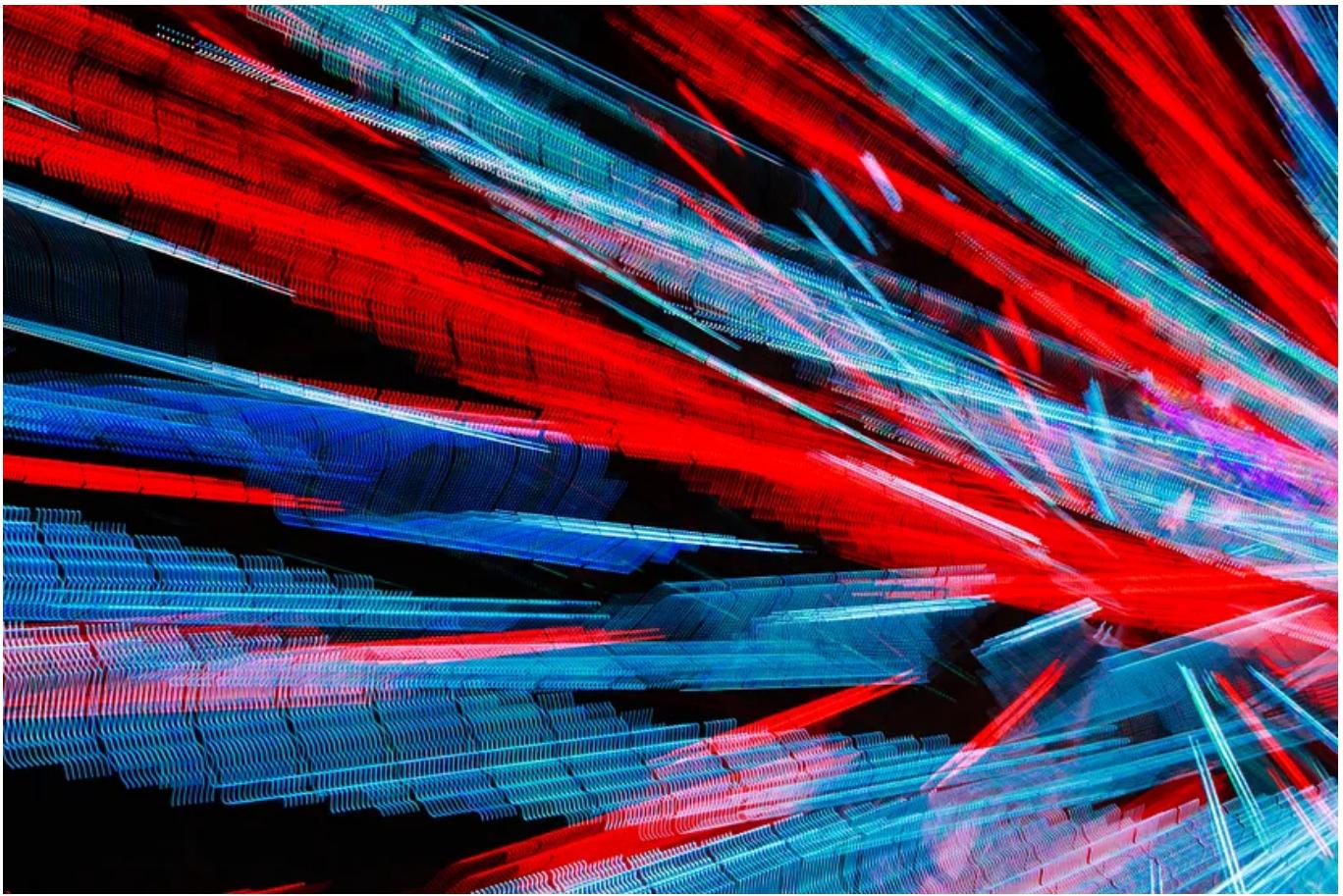


Photo by [Fabio Ballasina](#) on [Unsplash](#)

While I backtested a trading strategy with one of the most popular momentum oscillators, the Moving Average Convergence/Divergence (MACD), the results were amazing. Today, I found a clone of the MACD indicator whose performance is even more efficient. It's none other than the Know Sure Thing, shortly known as the KST indicator.

In this article, we will first build some basic intuitions on what the KST indicator is all about, and methods to calculate the indicator. After that, we will proceed to the programming part where we use Python to build the indicator from scratch, construct a simple trading strategy based on the indicator, backtest the strategy on Tesla stock and compare its performance with that of SPY ETF (an ETF whose purpose is to track the movements of the S&P 500 market index). With that being said, let's dive into the article!

Before moving on, if you want to backtest your trading strategies without any coding, there is a solution for it. It is BacktestZone. It is a platform to backtest any number of trading strategies on different types of tradeable assets for free without coding. You can use the tool right away using the link here: <https://www.backtestzone.com/>

Rate Of Change (ROC)

Before moving on to exploring the KST indicator, it is essential to know what the Rate Of Change indicator has got to offer since the Know Sure Thing indicator is based on ROC. The Rate Of Change indicator is a momentum indicator that is used by traders as an instrument to determine the percentage change in price from the current closing price and the price of a specified number of periods ago. Unlike other momentum indicators like the RSI and CCI, the Rate Of Change indicator is an unbounded oscillator whose values does not bound between certain limits.

To calculate the readings of ROC, we have to first determine the 'n' value which is nothing but how many periods ago the current closing price is compared to. The determination of 'n' varies from one trader to another but the traditional setting is 9 (widely used for short-term trading). With 9 as the 'n' value, the readings of the ROC indicator are calculated as follows:

First, the closing price of 9 periods ago is subtracted from the current closing price. This difference is then divided by the closing price of 9 periods ago and multiplied by 100. The calculation can be mathematically represented as follows:

```
ROC 9 = [ ( C CLOSE - PREV9 CLOSE ) / PREV9 CLOSE ] * 100
```

```
where,  
C.CLOSE = Current Closing Price  
PREV9.CLOSE = Closing Price of 9 Periods ago
```

Usually, the ROC is plotted below the plot of a stock's closing price against a zero line. If the readings of the ROC indicator are observed to be above the zero line, then the market is considered to reveal a sturdy upward momentum, and similarly, if the values are below the zero line, the market is considered to reveal a strong downward momentum. This is one way of using the ROC indicator and other usages include generating potential buy and sell signals, identifying the market state (overbought or oversold), and detecting divergences.

That's what the ROC and its calculation are all about. Now we are ready to jump on to exploring the main idea of this article, the Know Sure Thing indicator.

Know Sure Thing (KST)

The Know Sure Thing indicator is an unbounded momentum oscillator that is widely used by traders to understand the readings of the ROC indicator feasibly. The KST indicator is based on four different timeframes of smoothed ROC and combines the collective data into one oscillator. The Know Sure Thing indicator is composed of two components:

KST line: The first component is the KST line itself. To calculate the readings of the KST line, we have to first determine four ROCs with 10, 15, 20, 30 as the 'n' values respectively. Then each ROC is smoothed using a Simple Moving Average with 10, 10, 10, 15 as the lookback period respectively. This smoothed ROC is known as ROCSMA. After attaining the ROCSMA for four different timeframes, we have to multiply the first ROCSMA with one, the

second ROCSMA with two, the third ROCSMA with three, and the fourth with four. Finally, these four products are added to each other. The calculation of the KST line can be mathematically represented as follows:

$$KL = (ROCSMA1 * 1) + (ROCSMA2 * 2) + (ROCSMA3 * 3) + (ROCSMA4 * 4)$$

where,

KL = KST Line

ROCSMA1 = ROC 10 smoothed with SMA 10

ROCSMA2 = ROC 15 smoothed with SMA 10

ROCSMA3 = ROC 20 smoothed with SMA 10

ROCSMA4 = ROC 30 smoothed with SMA 15

Signal line: Now, the second component of the Know Sure Thing indicator is the Signal line component. This component is nothing but the smoothed version of the KST line. To smooth the values of the KST line, the Simple Moving Average with 9 as the lookback period is widely used. The calculation of the Signal line looks something as follows:

$$\text{SIGNAL LINE} = \text{SMA9 (KST LINE)}$$

There are many types of strategies that are built with the KST indicator like the divergence trading strategy, zero line crossover, and so on. Some suggest that it can also be used as an instrument to identify overbought and oversold levels but I personally think that it won't be as efficient as other indicators like RSI since the KST indicator is an unbounded oscillator. In today's article, we are going to discuss and implement a basic strategy called the crossover strategy.

The crossover strategy reveals a buy signal whenever the KST line crosses from below to above the Signal line. A sell signal is revealed whenever the KST line crosses from above the below the Signal line. The strategy can be represented as follows:

```
IF P.KST LINE < P.SIGNAL LINE AND C.KST LINE > C.SIGNAL LINE => BUY  
IF P.KST LINE > P.SIGNAL LINE AND C.KST LINE < C.SIGNAL LINE => SELL
```

This concludes our theory part on the Know Sure Thing indicator. Now, let's proceed to the programming part where we are going to use python to build the indicator from scratch, construct the crossover trading strategy, backtest the strategy on Tesla stock, compare the crossover strategy returns with those of the SPY ETF. Without further ado, let's jump into the programming part and do some coding! Before moving on, a note on disclaimer: This article's sole purpose is to educate people and must be considered as an information piece but not as investment advice or so.

Implementation in Python

The coding part is classified into various steps as follows:

1. Importing Packages
2. Extracting Stock Data from EODHD
3. ROC Calculation
4. Know Sure Thing Calculation
5. Know Sure Thing indicator Plot
6. Creating the Trading Strategy
7. Plotting the Trading Lists
8. Creating our Position
9. Backtesting
10. SPY ETF Comparison

We will be following the order mentioned in the above list and buckle up your seat belts to follow every upcoming coding part.

Step-1: Importing Packages

Importing the required packages into the python environment is a non-skippable step. The primary packages are going to be Pandas to work with data, NumPy to work with arrays and for complex functions, Matplotlib for plotting purposes, and Requests to make API calls. The secondary packages are going to be Math for mathematical functions and Termcolor for font customization (optional).

Python Implementation:

IMPORTING PACKAGES

```
import pandas as pd
import numpy as np
import requests
import matplotlib.pyplot as plt
from math import floor
from termcolor import colored as cl

plt.style.use('fivethirtyeight')
plt.rcParams['figure.figsize'] = (20,10)
```

With the required packages imported into Python, we can proceed to fetch historical data for Tesla using EODHD's OHLC split-adjusted data API endpoint.

Step-2: Extracting data from EODHD

In this phase, we're set to retrieve the historical stock data for Tesla using the OHLC split-adjusted API endpoint provided by EODHD. It's important to note that EOD Historical Data (EODHD) is a reliable provider of financial APIs,

encompassing an extensive array of market data, including historical data and economic news. Be sure to possess an EODHD account and access your secret API key, a crucial element for data extraction via the API.

Python Implementation:

```
# EXTRACTING STOCK DATA

def get_historical_data(symbol, start_date):
    api_key = 'YOUR API KEY'
    api_url = f'https://eodhistoricaldata.com/api/technical/{symbol}?order=a&fmt=json'
    raw_df = requests.get(api_url).json()
    df = pd.DataFrame(raw_df)
    df.date = pd.to_datetime(df.date)
    df = df.set_index('date')
    return df

tsla = get_historical_data('TSLA', '2019-01-01')
tsla.tail()
```

Output:

	open	high	low	close	volume
datetime					
2019-01-02	61.22000	63.02600	59.76000	62.02400	58293000.0
2019-01-03	61.40000	61.88000	59.47600	60.07200	34826000.0
2019-01-04	61.20000	63.60000	60.54600	63.53800	36970500.0
2019-01-07	64.34400	67.34800	63.55000	66.99200	37756000.0

Open in app ↗

	open	high	low	close	volume
datetime					
2021-05-24	581.59998	614.47998	573.65002	606.44000	34558089.0
2021-05-25	607.31000	613.98999	595.71002	604.69000	27933100.0
2021-05-26	607.56000	626.16998	601.50000	619.13000	28582600.0
2021-05-27	620.23999	631.13000	616.21002	630.84998	26314300.0
2021-05-28	628.50000	635.59003	622.38000	625.21997	22691200.0

Image by Author

Code Explanation: We begin by defining a function named ‘get_historical_data,’ which takes the stock symbol (‘symbol’) and the start date for historical data (‘start_date’) as parameters. Inside the function, we define the API key and URL, then retrieve the historical data in JSON format using the ‘get’ function and store it in the ‘raw_df’ variable. After cleaning and formatting the raw JSON data, we return it as a Pandas dataframe. Finally, we call this function to fetch Tesla’s historical data from the start of 2020 and store it in the ‘tsla’ variable.

Step-3: ROC Calculation

In this step, we are going to define a function to calculate the values of the Rate Of Change Indicator for a given series.

Python Implementation:

ROC CALCULATION

```
def get_roc(close, n):
    difference = close.diff(n)
    nprev_values = close.shift(n)
    roc = (difference / nprev_values) * 100
    return roc
```

Code Explanation: We are first defining a function named ‘get_roc’ that takes the stock’s closing price (‘close’) and the ‘n’ value (‘n’) as parameters. Inside the function, we are first taking the difference between the current closing price and the closing price for a specified number of periods ago using the ‘diff’ function provided by the Pandas package. With the help of the ‘shift’ function, we are taking into account the closing price for a specified number of periods ago and stored it into the ‘nprev_values’ variable. Then, we are substituting the determined values into the ROC indicator formula we discussed before to calculate the values and finally returned the data.

Step-4: Know Sure Thing Calculation

In this step, we are going to calculate the components of the Know Sure Thing indicator by following the methods and formula we discussed before.

Python Implementation:

KST CALCULATION

```

def get_kst(close, sma1, sma2, sma3, sma4, roc1, roc2, roc3, roc4,
signal):
    rcma1 = get_roc(close, roc1).rolling(sma1).mean()
    rcma2 = get_roc(close, roc2).rolling(sma2).mean()
    rcma3 = get_roc(close, roc3).rolling(sma3).mean()
    rcma4 = get_roc(close, roc4).rolling(sma4).mean()
    kst = (rcma1 * 1) + (rcma2 * 2) + (rcma3 * 3) + (rcma4 * 4)
    signal = kst.rolling(signal).mean()
    return kst, signal

tsla['kst'], tsla['signal_line'] = get_kst(tsla['close'], 10, 10, 10,
15, 10, 15, 20, 30, 9)
tsla = tsla[tsla.index >= '2020-01-01']
tsla.tail()

```

Output:

	open	high	low	close	volume	kst	signal_line
datetime							
2021-05-24	581.59998	614.47998	573.65002	606.44000	34558089.0	-138.083629	-91.017267
2021-05-25	607.31000	613.98999	595.71002	604.69000	27933100.0	-140.464066	-103.271480
2021-05-26	607.56000	626.16998	601.50000	619.13000	28582600.0	-137.755056	-113.254657
2021-05-27	620.23999	631.13000	616.21002	630.84998	26314300.0	-132.600478	-120.626382
2021-05-28	628.50000	635.59003	622.38000	625.21997	22691200.0	-130.822758	-125.945937

Image by Author

Code Explanation: Firstly, we are defining a function named ‘get_kst’ that takes the closing price of the stock (‘close’), four lookback periods for smoothing the ROC values (‘sma1’, ‘sma2’, ‘sma3’, ‘sma4’), four ‘n’ values of ROC (‘roc1’, ‘roc2’, ‘roc3’, ‘roc4’), and the lookback period for the signal line (‘signal’) as parameters.

Inside the function, we are first calculating the four ROCSMA values using the ‘rolling’ function provided by the Pandas package and the ‘get_roc’

function we created previously. Then, we are substituting the calculated ROCSMAs into the formula we discussed before to determine the readings of the KST line. Then, we are smoothing the KST line values with the ‘rolling’ function to get the values of the Signal line and stored them into the ‘signal’ variable.

Finally, we are returning both the calculated components of the KST indicator and calling the created function to store Tesla’s KST line and the Signal line readings.

Step-5: Know Sure Thing indicator Plot

In this step, we are going to plot the calculated components of the KST indicator values of Tesla to make more sense out of them. The main aim of this part is not on the coding section but instead to observe the plot to gain a solid understanding of the Know Sure Thing technical indicator.

Python Implementation:

```
# KST INDICATOR PLOT

ax1 = plt.subplot2grid((11,1), (0,0), rowspan = 5, colspan = 1)
ax2 = plt.subplot2grid((11,1), (6,0), rowspan = 5, colspan = 1)
ax1.plot(tsla['close'], linewidth = 2.5)
ax1.set_title('TSLA CLOSING PRICES')
ax2.plot(tsla['kst'], linewidth = 2, label = 'KST', color = 'orange')
ax2.plot(tsla['signal_line'], linewidth = 2, label = 'SIGNAL', color = 'mediumorchid')
ax2.legend()
ax2.set_title('TSLA KST')
plt.show()
```

Output:

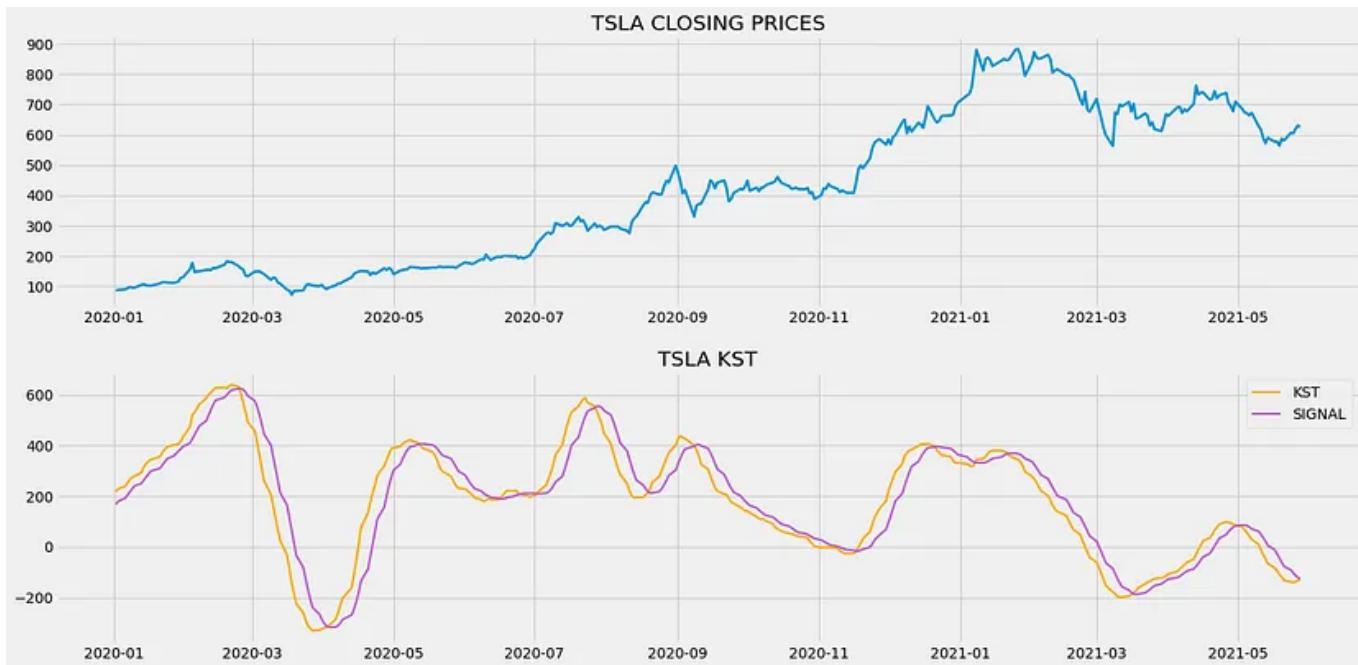


Image by Author

The above chart is divided into two panels: The upper panel with the closing price of Tesla, and the lower panel with the components of the Know Sure Thing Indicator. It can be noticed that the readings of both the components are indefinite and not bounded between certain limits because the Know Sure Thing indicator is an unbounded oscillator. This is the reason for the absence of the overbought and oversold levels in the chart that is widely plotted in other momentum oscillators. But, some propose that this indicator can be used to determine overbought and oversold levels but differ from one stock to another. This means, unlike other momentum oscillators which have default overbought and oversold levels, it is necessary to analyze the movement and readings of KST's components to determine the overbought and oversold levels. In our case, it would be optimal to spot the overbought level at 100 and the oversold level at -100.

One special characteristic feature of the KST indicator is that apart from determining the overbought and oversold levels or spotting divergence, it can be used as a tool to detect ranging markets (markets that show no trend

or momentum but move back and forth between specific high and low price ranges). Whenever both the components cross back and forth to each other, then the market is considered to be ranging. As Tesla's stock shows huge price movements, this phenomenon can be observed only a very few times (actually two: one around June and July in the previous year, the other at the starting of 2021).

Step-6: Creating the trading strategy

In this step, we are going to implement the discussed Know Sure Thing crossover trading strategy in python.

Python Implementation:

```
# KST CROSSOVER TRADING STRATEGY

def implement_kst_strategy(prices, kst_line, signal_line):
    buy_price = []
    sell_price = []
    kst_signal = []
    signal = 0

    for i in range(len(kst_line)):

        if kst_line[i-1] < signal_line[i-1] and kst_line[i] >
signal_line[i]:
            if signal != 1:
                buy_price.append(prices[i])
                sell_price.append(np.nan)
                signal = 1
                kst_signal.append(signal)
            else:
                buy_price.append(np.nan)
                sell_price.append(np.nan)
                kst_signal.append(0)

        elif kst_line[i-1] > signal_line[i-1] and kst_line[i] <
signal_line[i]:
            if signal != -1:
                buy_price.append(np.nan)
                sell_price.append(prices[i])
                signal = -1
            else:
                buy_price.append(prices[i])
                sell_price.append(np.nan)
                kst_signal.append(0)
```

```

        kst_signal.append(signal)
    else:
        buy_price.append(np.nan)
        sell_price.append(np.nan)
        kst_signal.append(0)

    else:
        buy_price.append(np.nan)
        sell_price.append(np.nan)
        kst_signal.append(0)

return buy_price, sell_price, kst_signal

buy_price, sell_price, kst_signal =
implement_kst_strategy(tsla['close'], tsla['kst'],
tsla['signal_line'])

```

Code Explanation: First, we are defining a function named ‘implement_kst_strategy’ which takes the stock prices (‘prices’), the readings of the KST line (‘kst_line’), and the readings of the Signal line (‘signal_line’) as parameters.

Inside the function, we are creating three empty lists (buy_price, sell_price, and kst_signal) in which the values will be appended while creating the trading strategy.

After that, we are implementing the trading strategy through a for-loop. Inside the for-loop, we are passing certain conditions, and if the conditions are satisfied, the respective values will be appended to the empty lists. If the condition to buy the stock gets satisfied, the buying price will be appended to the ‘buy_price’ list, and the signal value will be appended as 1 representing to buy the stock. Similarly, if the condition to sell the stock gets satisfied, the selling price will be appended to the ‘sell_price’ list, and the signal value will be appended as -1 representing to sell the stock.

Finally, we are returning the lists appended with values. Then, we are calling the created function and stored the values into their respective variables. The list doesn't make any sense unless we plot the values. So, let's plot the values of the created trading lists.

Step-7: Plotting the trading signals

In this step, we are going to plot the created trading lists to make sense out of them.

Python Implementation:

```
# TRADING SIGNALS PLOT

ax1 = plt.subplot2grid((11,1), (0,0), rowspan = 5, colspan = 1)
ax2 = plt.subplot2grid((11,1), (6,0), rowspan = 5, colspan = 1)
ax1.plot(tsla['close'], linewidth = 2, label = 'TSLA')
ax1.plot(tsla.index, buy_price, marker = '^', markersize = 12,
linewidth = 0, color = 'green', label = 'BUY SIGNAL')
ax1.plot(tsla.index, sell_price, marker = 'v', markersize = 12,
linewidth = 0, color = 'r', label = 'SELL SIGNAL')
ax1.legend()
ax1.set_title('TSLA KST TRADING SIGNALS')
ax2.plot(tsla['kst'], linewidth = 2, label = 'KST', color = 'orange')
ax2.plot(tsla['signal_line'], linewidth = 2, label = 'SIGNAL', color
= 'mediumorchid')
ax2.legend()
ax2.set_title('TSLA KST')
plt.show()
```

Output:

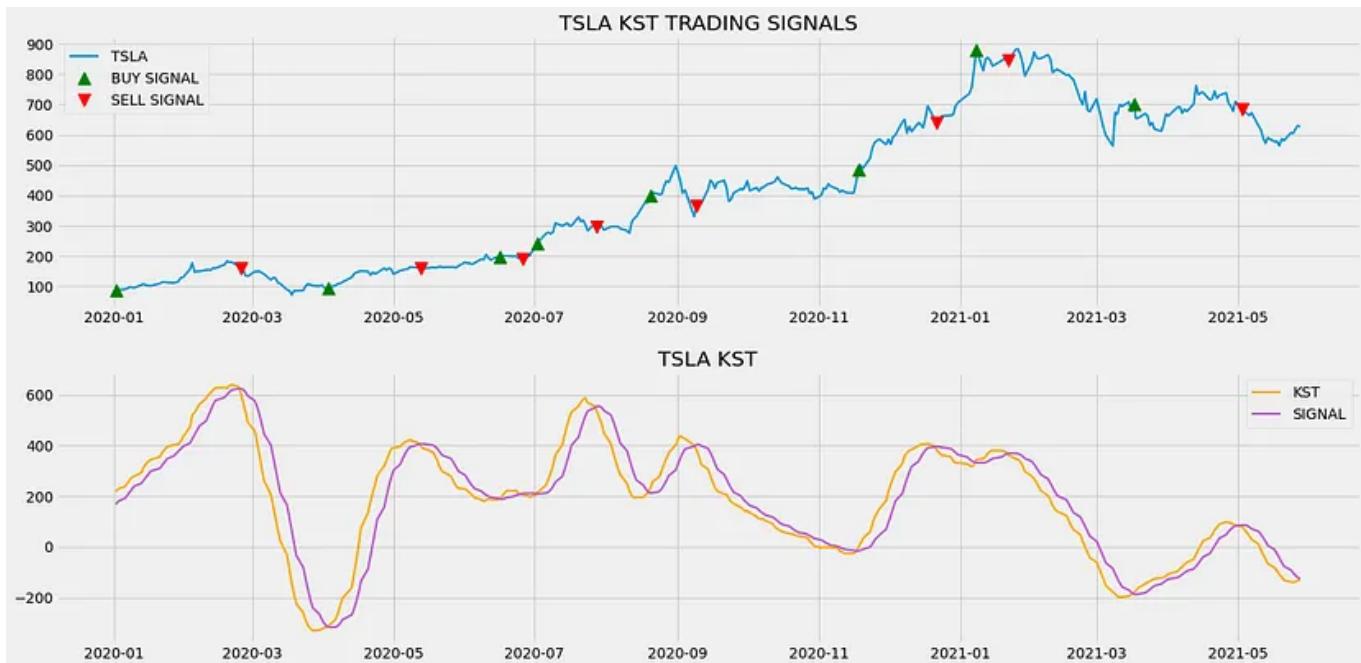


Image by Author

Code Explanation: We are plotting the readings of the KST indicator's components along with the buy and sell signals generated by the crossover trading strategy. We can observe that whenever the KST line crosses from below to above the Signal line, a green-colored buy signal is plotted in the chart. Similarly, whenever the KST line crosses from above to below the Signal line, a red-colored sell signal is plotted in the chart.

Step-8: Creating our Position

In this step, we are going to create a list that indicates 1 if we hold the stock or 0 if we don't own or hold the stock.

Python Implementation:

STOCK POSITION

```
position = []
for i in range(len(kst_signal)):
    if kst_signal[i] > 1:
        position.append(0)
```

```

else:
    position.append(1)

for i in range(len(tsla['close'])):
    if kst_signal[i] == 1:
        position[i] = 1
    elif kst_signal[i] == -1:
        position[i] = 0
    else:
        position[i] = position[i-1]

close_price = tsla['close']
kst = tsla['kst']
signal_line = tsla['signal_line']
kst_signal = pd.DataFrame(kst_signal).rename(columns =
{0:'kst_signal'}).set_index(tsla.index)
position = pd.DataFrame(position).rename(columns =
{0:'kst_position'}).set_index(tsla.index)

frames = [close_price, kst, signal_line, kst_signal, position]
strategy = pd.concat(frames, join = 'inner', axis = 1)

strategy

```

Output:

	close	kst	signal_line	kst_signal	kst_position
datetime					
2020-02-24	166.75800	632.572816	624.494163	0	1
2020-02-25	159.98199	605.734378	625.068576	-1	0
2020-02-26	155.75999	569.361461	620.550388	0	0
2020-02-27	135.80000	534.169053	611.108291	0	0
2020-02-28	133.59801	491.133661	595.916520	0	0

Image by Author

Code Explanation: First, we are creating an empty list named ‘position’. We are passing two for-loops, one is to generate values for the ‘position’ list to just match the length of the ‘signal’ list. The other for-loop is the one we are using to generate actual position values. Inside the second for-loop, we are iterating over the values of the ‘signal’ list, and the values of the ‘position’ list get appended concerning which condition gets satisfied. The value of the position remains 1 if we hold the stock or remains 0 if we sold or don’t own the stock. Finally, we are doing some data manipulations to combine all the created lists into one dataframe.

From the output being shown, we can see that in the first row our position in the stock has remained 1 (since there isn’t any change in the KST signal) but our position suddenly turned to -1 as we sold the stock when the KST trading signal represents a sell signal (-1). Our position will remain 0 until some changes in the trading signal occur. Now it’s time to do implement some backtesting process!

Step-9: Backtesting

Before moving on, it is essential to know what backtesting is. Backtesting is the process of seeing how well our trading strategy has performed on the given stock data. In our case, we are going to implement a backtesting process for our KST indicator crossover trading strategy over the Tesla stock data.

Python Implementation:

BACKTESTING

```
tsla_ret = pd.DataFrame(np.diff(tsla['close'])).rename(columns = {0:'returns'})  
kst_strategy_ret = []
```

```

for i in range(len(tsla_ret)):
    returns = tsla_ret['returns'][i]*strategy['kst_position'][i]
    kst_strategy_ret.append(returns)

kst_strategy_ret_df = pd.DataFrame(kst_strategy_ret).rename(columns =
{0:'kst_returns'})
investment_value = 100000
number_of_stocks = floor(investment_value/tsla['close'][0])
kst_investment_ret = []

for i in range(len(kst_strategy_ret_df['kst_returns'])):
    returns = number_of_stocks*kst_strategy_ret_df['kst_returns'][i]
    kst_investment_ret.append(returns)

kst_investment_ret_df =
pd.DataFrame(kst_investment_ret).rename(columns =
{0:'investment_returns'})
total_investment_ret =
round(sum(kst_investment_ret_df['investment_returns']), 2)
profit_percentage =
floor((total_investment_ret/investment_value)*100)
print(cl('Profit gained from the KST strategy by investing $100k in
TSLA : {}'.format(total_investment_ret), attrs = ['bold']))
print(cl('Profit percentage of the KST strategy :
{}%'.format(profit_percentage), attrs = ['bold']))

```

Output:

Profit gained from the KST strategy by investing \$100k in TSLA :
295766.23
Profit percentage of the KST strategy : **295%**

Code Explanation: First, we are calculating the returns of the Tesla stock using the ‘diff’ function provided by the NumPy package and we have stored it as a dataframe into the ‘tsla_ret’ variable. Next, we are passing a for-loop to iterate over the values of the ‘tsla_ret’ variable to calculate the returns we gained from our Know Sure Thing indicator trading strategy, and these returns values are appended to the ‘kst_strategy_ret’ list. Next, we are

converting the ‘kst_strategy_ret’ list into a dataframe and stored it into the ‘kst_strategy_ret_df’ variable.

Next comes the backtesting process. We are going to backtest our strategy by investing a hundred thousand USD into our trading strategy. So first, we are storing the amount of investment into the ‘investment_value’ variable. After that, we are calculating the number of Tesla stocks we can buy using the investment amount. You can notice that I’ve used the ‘floor’ function provided by the Math package because, while dividing the investment amount by the closing price of Tesla stock, it spits out an output with decimal numbers. The number of stocks should be an integer but not a decimal number. Using the ‘floor’ function, we can cut out the decimals. Remember that the ‘floor’ function is way more complex than the ‘round’ function. Then, we are passing a for-loop to find the investment returns followed by some data manipulation tasks.

Finally, we are printing the total return we got by investing a hundred thousand into our trading strategy and it is revealed that we have made an approximate profit of two hundred and twenty-nine thousand USD in one year. That’s awesome! Now, let’s compare our returns with SPY ETF (an ETF designed to track the S&P 500 stock market index) returns.

Step-10: SPY ETF Comparison

This step is optional but it is highly recommended as we can get an idea of how well our trading strategy performs against a benchmark (SPY ETF). In this step, we are going to extract the data of the SPY ETF using the ‘get_historical_data’ function we created and compare the returns we get from the SPY ETF with our KST crossover trading strategy returns on Tesla.

Python Implementation:

SPY ETF COMPARISON

```

def get_benchmark(start_date, investment_value):
    spy = get_historical_data('SPY', start_date)['close']
    benchmark = pd.DataFrame(np.diff(spy)).rename(columns =
{0:'benchmark_returns'})

    investment_value = investment_value
    number_of_stocks = floor(investment_value/spy[-1])
    benchmark_investment_ret = []

    for i in range(len(benchmark['benchmark_returns'])):
        returns = number_of_stocks*benchmark['benchmark_returns'][i]
        benchmark_investment_ret.append(returns)

    benchmark_investment_ret_df =
pd.DataFrame(benchmark_investment_ret).rename(columns =
{0:'investment_returns'})
    return benchmark_investment_ret_df

benchmark = get_benchmark('2020-01-01', 100000)
investment_value = 100000
total_benchmark_investment_ret =
round(sum(benchmark['investment_returns']), 2)
benchmark_profit_percentage =
floor((total_benchmark_investment_ret/investment_value)*100)
print(cl('Benchmark profit by investing $100k :
{}').format(total_benchmark_investment_ret), attrs = ['bold']))
print(cl('Benchmark Profit percentage :
{}%').format(benchmark_profit_percentage), attrs = ['bold']))
print(cl('KST Strategy profit is {}% higher than the Benchmark
Profit'.format(profit_percentage - benchmark_profit_percentage),
attrs = ['bold']))

```

Output:

Benchmark profit by investing \$100k : 22650.46
Benchmark Profit percentage : 22%
KST Strategy profit is 273% higher than the Benchmark Profit

Code Explanation: The code used in this step is almost similar to the one used in the previous backtesting step but, instead of investing in Tesla, we are investing in SPY ETF by not implementing any trading strategies. From the output, we can see that our Know Sure Thing crossover trading strategy has outperformed the SPY ETF by 273%. That's great!

Final Thoughts!

After an overwhelming process of crushing both theory and coding parts, we have successfully learned what the Know Sure Thing indicator is all about, the math behind it, and how to build a simple KST crossover trading strategy in python.

Even though we made a wonderful profit with our KST strategy and outperformed the SPY ETF, our strategy returns are still lesser than the actual Tesla stock returns. This is because of one major drawback of the KST indicator. That is nothing but the KST indicator is prone to revealing a lot of false signals during the periods of ranging markets and this can be observed in our crossover trading signal plot where a lot of unnecessary trading signals are revealed by our trading strategy.

The only way to optimize and resolve this problem is by adding another technical indicator that acts as a gauge to filter non-authentic or false signals given by the trading strategy. In my opinion, the Choppiness Index would work wonderfully with the KST indicator since it is an indicator dedicated to tracking whether a market is ranging or not, also, it is one of the most accurate indicators to do so. So it is highly recommended to run as many backtests as possible with a KST trading strategy that uses another indicator to differentiate the false signals from the authentic ones and by doing this will take your results to the next level.

With that being said, you've reached the end of the article. If you forgot to follow any of the coding parts, don't worry. I've provided the full source code at the bottom of the article. Hope you found and learned something useful from this article. Happy learning!

Full code:

IMPORTING PACKAGES

```
import pandas as pd
import numpy as np
import requests
import matplotlib.pyplot as plt
from math import floor
from termcolor import colored as cl

plt.style.use('fivethirtyeight')
plt.rcParams['figure.figsize'] = (20,10)
```

EXTRACTING STOCK DATA

```
def get_historical_data(symbol, start_date):
    api_key = 'YOUR API KEY'
    api_url = f'https://api.twinkledata.com/time_series?symbol={symbol}&interval=1day&outputsize=5000&apikey={api_key}'
    raw_df = requests.get(api_url).json()
    df =
    pd.DataFrame(raw_df['values']).iloc[::-1].set_index('datetime').astype(float)
    df = df[df.index >= start_date]
    df.index = pd.to_datetime(df.index)
    return df

tsla = get_historical_data('TSLA', '2019-01-01')
print(tsla)
```

ROC CALCULATION

```
def get_roc(close, n):
    difference = close.diff(n)
    nprev_values = close.shift(n)
    roc = (difference / nprev_values) * 100
    return roc
```

KST CALCULATION

```
def get_kst(close, sma1, sma2, sma3, sma4, roc1, roc2, roc3, roc4,
signal):
    rcma1 = get_roc(close, roc1).rolling(sma1).mean()
    rcma2 = get_roc(close, roc2).rolling(sma2).mean()
    rcma3 = get_roc(close, roc3).rolling(sma3).mean()
    rcma4 = get_roc(close, roc4).rolling(sma4).mean()
    kst = (rcma1 * 1) + (rcma2 * 2) + (rcma3 * 3) + (rcma4 * 4)
    signal = kst.rolling(signal).mean()
    return kst, signal

tsla['kst'], tsla['signal_line'] = get_kst(tsla['close'], 10, 10, 10,
15, 10, 15, 20, 30, 9)
tsla = tsla[tsla.index >= '2020-01-01']
print(tsla.tail())
```

KST INDICATOR PLOT

```
ax1 = plt.subplot2grid((11,1), (0,0), rowspan = 5, colspan = 1)
ax2 = plt.subplot2grid((11,1), (6,0), rowspan = 5, colspan = 1)
ax1.plot(tsla['close'], linewidth = 2.5)
ax1.set_title('TSLA CLOSING PRICES')
ax2.plot(tsla['kst'], linewidth = 2, label = 'KST', color = 'orange')
ax2.plot(tsla['signal_line'], linewidth = 2, label = 'SIGNAL', color =
'mediumorchid')
ax2.legend()
ax2.set_title('TSLA KST')
plt.show()
```

KST CROSSOVER TRADING STRATEGY

```
def implement_kst_strategy(prices, kst_line, signal_line):
    buy_price = []
    sell_price = []
    kst_signal = []
    signal = 0

    for i in range(len(kst_line)):

        if kst_line[i-1] < signal_line[i-1] and kst_line[i] >
signal_line[i]:
            if signal != 1:
                buy_price.append(prices[i])
                sell_price.append(np.nan)
                signal = 1
                kst_signal.append(signal)
```

```

        else:
            buy_price.append(np.nan)
            sell_price.append(np.nan)
            kst_signal.append(0)

        elif kst_line[i-1] > signal_line[i-1] and kst_line[i] <
signal_line[i]:
            if signal != -1:
                buy_price.append(np.nan)
                sell_price.append(prices[i])
                signal = -1
                kst_signal.append(signal)
            else:
                buy_price.append(np.nan)
                sell_price.append(np.nan)
                kst_signal.append(0)

        else:
            buy_price.append(np.nan)
            sell_price.append(np.nan)
            kst_signal.append(0)

    return buy_price, sell_price, kst_signal
}

buy_price, sell_price, kst_signal =
implement_kst_strategy(tsla['close'], tsla['kst'],
tsla['signal_line'])

```

TRADING SIGNALS PLOT

```

ax1 = plt.subplot2grid((11,1), (0,0), rowspan = 5, colspan = 1)
ax2 = plt.subplot2grid((11,1), (6,0), rowspan = 5, colspan = 1)
ax1.plot(tsla['close'], linewidth = 2, label = 'TSLA')
ax1.plot(tsla.index, buy_price, marker = '^', markersize = 12,
linewidth = 0, color = 'green', label = 'BUY SIGNAL')
ax1.plot(tsla.index, sell_price, marker = 'v', markersize = 12,
linewidth = 0, color = 'r', label = 'SELL SIGNAL')
ax1.legend()
ax1.set_title('TSLA KST TRADING SIGNALS')
ax2.plot(tsla['kst'], linewidth = 2, label = 'KST', color = 'orange')
ax2.plot(tsla['signal_line'], linewidth = 2, label = 'SIGNAL', color
= 'mediumorchid')
ax2.legend()
ax2.set_title('TSLA KST')
plt.show()

```

STOCK POSITION

```
position = []
```

```

for i in range(len(kst_signal)):
    if kst_signal[i] > 1:
        position.append(0)
    else:
        position.append(1)

for i in range(len(tsla['close'])):
    if kst_signal[i] == 1:
        position[i] = 1
    elif kst_signal[i] == -1:
        position[i] = 0
    else:
        position[i] = position[i-1]

close_price = tsla['close']
kst = tsla['kst']
signal_line = tsla['signal_line']
kst_signal = pd.DataFrame(kst_signal).rename(columns =
{0:'kst_signal'}).set_index(tsla.index)
position = pd.DataFrame(position).rename(columns =
{0:'kst_position'}).set_index(tsla.index)

frames = [close_price, kst, signal_line, kst_signal, position]
strategy = pd.concat(frames, join = 'inner', axis = 1)

print(strategy)
print(strategy[35:40])

```

BACKTESTING

```

tsla_ret = pd.DataFrame(np.diff(tsla['close'])).rename(columns =
{0:'returns'})
kst_strategy_ret = []

for i in range(len(tsla_ret)):
    returns = tsla_ret['returns'][i]*strategy['kst_position'][i]
    kst_strategy_ret.append(returns)

kst_strategy_ret_df = pd.DataFrame(kst_strategy_ret).rename(columns =
{0:'kst_returns'})
investment_value = 100000
number_of_stocks = floor(investment_value/tsla['close'][0])
kst_investment_ret = []

for i in range(len(kst_strategy_ret_df['kst_returns'])):
    returns = number_of_stocks*kst_strategy_ret_df['kst_returns'][i]
    kst_investment_ret.append(returns)

kst_investment_ret_df =
pd.DataFrame(kst_investment_ret).rename(columns =

```

```
{0:'investment_returns'})}
total_investment_ret =
round(sum(kst_investment_ret_df['investment_returns']), 2)
profit_percentage =
floor((total_investment_ret/investment_value)*100)
print(cl('Profit gained from the KST strategy by investing $100k in
TSLA : {}'.format(total_investment_ret), attrs = ['bold']))
print(cl('Profit percentage of the KST strategy :
{}%'.format(profit_percentage), attrs = ['bold']))
```

SPY ETF COMPARISON

```
def get_benchmark(start_date, investment_value):
    spy = get_historical_data('SPY', start_date)['close']
    benchmark = pd.DataFrame(np.diff(spy)).rename(columns =
{0:'benchmark_returns'})

    investment_value = investment_value
    number_of_stocks = floor(investment_value/spy[-1])
    benchmark_investment_ret = []

    for i in range(len(benchmark['benchmark_returns'])):
        returns = number_of_stocks*benchmark['benchmark_returns'][i]
        benchmark_investment_ret.append(returns)

    benchmark_investment_ret_df =
pd.DataFrame(benchmark_investment_ret).rename(columns =
{0:'investment_returns'})
    return benchmark_investment_ret_df

benchmark = get_benchmark('2020-01-01', 100000)

investment_value = 100000
total_benchmark_investment_ret =
round(sum(benchmark['investment_returns']), 2)
benchmark_profit_percentage =
floor((total_benchmark_investment_ret/investment_value)*100)
print(cl('Benchmark profit by investing $100k :
{}'.format(total_benchmark_investment_ret), attrs = ['bold']))
print(cl('Benchmark Profit percentage :
{}%'.format(benchmark_profit_percentage), attrs = ['bold']))
print(cl('KST Strategy profit is {}% higher than the Benchmark
Profit'.format(profit_percentage - benchmark_profit_percentage),
attrs = ['bold']))
```

Finance

Investing

Education

Programming

Python



Written by Nikhil Adithyan

7.8K Followers · Writer for CodeX

Follow



Founder @BacktestZone (<https://www.backtestzone.com/>), a no-code backtesting platform | Top Writer | Connect with me on LinkedIn: <https://bit.ly/3yNuwCJ>

More from Nikhil Adithyan and CodeX



 Nikhil Adithyan in DataDrivenInvestor

Stock Price Prediction with Quantum Machine Learning in...

An overview of the challenges and opportunities

17 min read · Jan 22, 2024



 Anmol Tomar in CodeX

Say Goodbye to Loops in Python, and Welcome Vectorization!

Use Vectorization—a super-fast alternative to loops in Python

★ · 5 min read · Dec 27, 2023

1.5K

17



•••



4.3K

50



•••



Anmol Tomar in CodeX

Don't use loc/iloc with Loops In Python, Instead, Use This!

Run your loops at a 60X faster speed

· 3 min read · Jan 23, 2024

1.3K

10



•••



2.8K

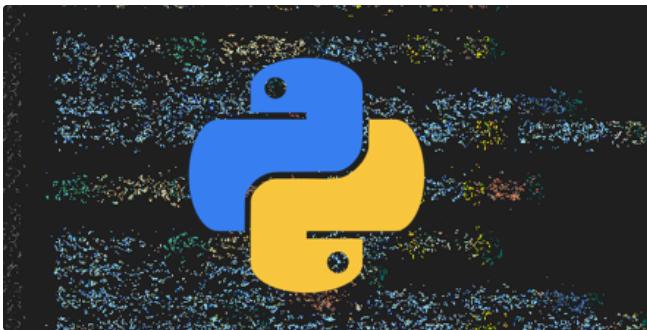
46



•••

[See all from Nikhil Adithyan](#)[See all from CodeX](#)

Recommended from Medium



Jaden Duxfield

Using python to develop day trading strategies

Introduction:

10 min read · Oct 17, 2023

34

1



...

Dapo willis

Forex Millionaire: The Concepts Made me \$15 Million Trading

The article looks into the trading concepts that made me \$15 million trading.

9 min read · Dec 18, 2023

92

3



...

Lists



Coding & Development

11 stories · 449 saves



General Coding Knowledge

20 stories · 924 saves



Predictive Modeling w/ Python

20 stories · 914 saves



ChatGPT

21 stories · 466 saves





Nikhil Adithyan in DataDrivenInvestor



Paul Lenosky

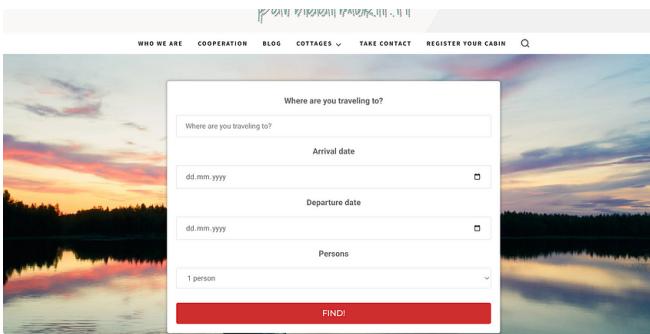
Create a Stock Chatbot with your own CSV Data

An Explorative Study with Python

10 min read · 3 days ago



67



I have created an indicator that actually makes money, unlike any...

How the indicator works in a nutshell

7 min read · Feb 9, 2024



232



12



Artturi Jalli

I Built an App in 6 Hours that Makes \$1,500/Mo

Copy my strategy!



· 3 min read · Jan 23, 2024



8.6K



118



How to plot Bollinger Bands in Python

Introduction

3 min read · Oct 6, 2023



5



[See more recommendations](#)