stock-pandas

Apr 07, 2023

Contents

stock-pandas inherits and extends `pandas.DataFrame` to support:

- Stock Statistics
- Stock Indicators, including:
  - Trend-following momentum indicators, such as ***MA***, ***EMA***, ***MACD***, ***BBI***
  - Dynamic support and resistance indicators, such as ***BOLL***
  - Over-bought / over-sold indicators, such as ***KDJ***, ***RSI***
  - Other indicators, such as ***LLV***, ***HHV***
  - For more indicators, welcome to request a proposal, or fork and send me a pull request, or extend stock-pandas yourself. You might read the Advanced Sections below.
- To *cumulate* kline data based on a given time frame, so that it could easily handle real-time data updates.

`stock-pandas` makes automatical trading much easier. `stock-pandas` requires Python >= **3.6** and Pandas >= **1.0.0**(for now)

With the help of `stock-pandas` and mplfinance, we could easily draw something like:

The code example is available at here.

CHAPTER 1

Install

For now, before installing `stock-pandas` in your environment

1.1 Have g++ compiler installed

```
# With yum, for CentOS, Amazon Linux, etc
yum install gcc-c++

# With apt-get, for Ubuntu
apt-get install g++

# For macOS, install XCode commandline tools
xcode-select --install
```

If you use docker with `Dockerfile` and use python image,

```
FROM python:3.8

...
```

The default `python:3.8` image already contains g++, so we do not install g++ additionally.

1.2 Install `stock-pandas`

```
# Installing `stock-pandas` requires `numpy` to be installed first
pip install numpy

pip install stock-pandas
```

Be careful, you still need to install `numpy` explicitly even if `numpy` and `stock-pandas` both are contained in `requirement.txt`

```
numpy
stock-pandas
other-dependencies
...
```

```
pip install numpy

pip install -r requirement.txt
```

CHAPTER 2

Usage

```
from stock_pandas import StockDataFrame

# or
import stock_pandas as spd
```

We also have some examples with annotations in the `example` directory, you could use JupyterLab or Jupyter notebook to play with them.

2.1 StockDataFrame

`StockDataFrame` inherits from `pandas.DataFrame`, so if you are familiar with `pandas.DataFrame`, you are already ready to use `stock-pandas`

```
import pandas as pd
stock = StockDataFrame(pd.read_csv('stock.csv'))
```

As we know, we could use `[]`, which called **pandas indexing** (a.k.a. `__getitem__` in python) to select out lower-dimensional slices. In addition to indexing with `colname` (column name of the `DataFrame`), we could also do indexing by `directives`.

```
stock[directive] # Gets a pandas.Series

stock[[directive0, directive1]] # Gets a StockDataFrame
```

We have an example to show the most basic indexing using `[directive]`

```
stock = StockDataFrame({
    'open' : ...,
    'high' : ...,
    'low'  : ...,
    'close': [5, 6, 7, 8, 9]
```

(continues on next page)

(continued from previous page)

```
})

stock['ma:2']

# 0    NaN
# 1    5.5
# 2    6.5
# 3    7.5
# 4    8.5
# Name: ma:2,close, dtype: float64
```

Which prints the 2-period simple moving average on column `"close"`.

2.1.1 Parameters

- **date_col** `Optional[str] = None` If set, then the column named `date_col` will convert and set as `DateTimeIndex` of the data frame
- **to_datetime_kwargs** `dict = {}` the keyworded arguments to be passed to `pandas.to_datetime()`. It only takes effect if `date_col` is specified.
- **time_frame** `str | TimeFrame | None = None` time frame of the stock. For now, only the following time frames are supported:
  - `'1m'` or `TimeFrame.M1`
  - `'3m'` or `TimeFrame.M3`
  - `'5m'` or `TimeFrame.M5`
  - `'15m'` or `TimeFrame.M15`
  - `'30m'` or `TimeFrame.M30`
  - `'1h'` or `TimeFrame.H1`
  - `'2h'` or `TimeFrame.H2`
  - `'4h'` or `TimeFrame.H4`
  - `'6h'` or `TimeFrame.H6`
  - `'8h'` or `TimeFrame.H8`
  - `'12h'` or `TimeFrame.H12`

2.2 stock.exec(directive: str, create_column: bool=False) -> np.ndarray

Executes the given directive and returns a numpy ndarray according to the directive.

```
stock['ma:5'] # returns a Series

stock.exec('ma:5', create_column=True) # returns a numpy ndarray
```

```
# This will only calculate without creating a new column in the dataframe
stock.exec('ma:20')
```

The difference between `stock[directive]` and `stock.exec(directive)` is that

- the former will create a new column for the result of `directive` as a cache for later use, while `stock.exec(directive)` does not unless we pass the parameter `create_column` as `True`
- the former one accepts other pandas indexing targets, while `stock.exec(directive)` only accepts a valid **stock-pandas** directive string
- the former one returns a `pandas.Series` or `StockDataFrame` object while the latter one returns an `np.ndarray`

2.3 stock.alias(alias: str, name: str) -> None

Defines column alias or directive alias

- **alias** `str` the alias name
- **name** `str` the name of an existing column or the directive string

```
# Some plot library such as `mplfinance` requires a column named capitalized `Open`,
# but it is ok, we could create an alias.
stock.alias('Open', 'open')

stock.alias('buy_point', 'kdj.j < 0')
```

2.4 stock.get_column(key: str) -> pd.Series

Directly gets the column value by `key`, returns a pandas `Series`.

If the given `key` is an alias name, it will return the value of corresponding original column.

If the column is not found, a `KeyError` will be raised.

```
stock = StockDataFrame({
    'open' : ...,
    'high' : ...,
    'low'  : ...,
    'close': [5, 6, 7, 8, 9]
})

stock.get_column('close')
# 0    5
# 1    6
# 2    7
# 3    8
# 4    9
# Name: close, dtype: float64
```

```
try:
    stock.get_column('Close')
except KeyError as e:
    print(e)

    # KeyError: column "Close" not found
```

(continues on next page)

(continued from previous page)

```
stock.alias('Close', 'close')

stock.get_column('Close')
# The same as `stock.get_column('close')`
```

2.5 stock.append(other, *args, **kwargs) -> StockDataFrame

Appends rows of `other` to the end of caller, returning a new object.

This method has nearly the same behavior of `pandas.DataFrame.append()`, but instead it returns an instance of `StockDataFrame`, and it applies `date_col` to the newly-appended row(s) if possible.

2.6 stock.directive_stringify(directive: str) -> str

Since 0.26.0

Gets the full name of the `directive` which is also the actual column name of the data frame

```
stock.directive_stringify('kdj.j')
# "kdj.j:9,3,3,50.0"
```

And also

```
from stock_pandas import

directive_stringify('kdj.j')
# "kdj.j:9,3,3,50.0"
```

Actually, `directive_stringify` does not rely on StockDataFrame instances.

2.7 stock.rolling_calc(size, on, apply, forward, fill) -> np.ndarray

Since 0.27.0

Applies a 1-D function along the given column or directive `on`

- **size** `int` the size of the rolling window
- **on** `str | Directive` along which the function should be applied
- **apply** `Callable[[np.ndarray], Any]` the 1-D function to apply
- **forward?** `bool = False` whether we should look backward (default value) to get each rolling window or not
- **fill?** `Any = np.nan` the value used to fill where there are not enough items to form a rolling window

```
stock.rolling_calc(5, 'open', max)

# Whose return value equals to
stock['hhv:5,open'].to_numpy()
```

2.8 stock.cumulate() -> StockDataFrame

Cumulate the current data frame `stock` based on its time frame setting

```
StockDataFrame(one_minute_kline_data_frame, time_frame='5m').cumulate()

# And you will get a 5-minute kline data
```

see *Cumulation and DatetimeIndex* for details

2.9 stock.cum_append(other: DataFrame) -> StockDataFrame

Append `other` to the end of the current data frame `stock` and apply cumulation on them. And the following slice of code is equivalent to the above one:

```
StockDataFrame(time_frame='5m').cum_append(one_minute_kline_data_frame)
```

see *Cumulation and DatetimeIndex* for details

2.10 stock.fulfill() -> self

Since 1.2.0

Fulfill all stock indicator columns. By default, adding new rows to a `StockDataFrame` will not update stock indicators of the new row.

Stock indicators will only be updated when accessing the stock indicator column or calling `stock.fulfill()`

Check the test cases for details

2.11 directive_stringify(directive_str) -> str

since 0.30.0

Similar to `stock.directive_stringify()` but could be called without class initialization

```
from stock_pandas import directive_stringify

directive_stringify('boll')
# boll:21,close
```

CHAPTER 3

Cumulation and DatetimeIndex

Suppose we have a csv file containing kline data of a stock in 1-minute time frame

```
csv = pd.read_csv(csv_path)

print(csv)
```

```
                   date   open   high    low  close    volume
0   2020-01-01 00:00:00  329.4  331.6  327.6  328.8  14202519
1   2020-01-01 00:01:00  330.0  332.0  328.0  331.0  13953191
2   2020-01-01 00:02:00  332.8  332.8  328.4  331.0  10339120
3   2020-01-01 00:03:00  332.0  334.2  330.2  331.0   9904468
4   2020-01-01 00:04:00  329.6  330.2  324.9  324.9  13947162
5   2020-01-01 00:04:00  329.6  330.2  324.8  324.8  13947163    <- There is an␣
↪update of
                                                              2020-01-01␣
↪00:04:00
...
16  2020-01-01 00:16:00  333.2  334.8  331.2  334.0  12428539
17  2020-01-01 00:17:00  333.0  333.6  326.8  333.6  15533405
18  2020-01-01 00:18:00  335.0  335.2  326.2  327.2  16655874
19  2020-01-01 00:19:00  327.0  327.2  322.0  323.0  15086985
```

Noted that duplicated records of a same timestamp will not be cumulated. The records except the latest one will be disgarded.

```
stock = StockDataFrame(
    csv,
    date_col='date',
    # Which is equivalent to `time_frame=TimeFrame.M5`
    time_frame='5m'
)

print(stock)
```

```
                     open   high    low  close     volume
2020-01-01 00:00:00  329.4  331.6  327.6  328.8  14202519
2020-01-01 00:01:00  330.0  332.0  328.0  331.0  13953191
2020-01-01 00:02:00  332.8  332.8  328.4  331.0  10339120
2020-01-01 00:03:00  332.0  334.2  330.2  331.0   9904468
2020-01-01 00:04:00  329.6  330.2  324.9  324.9  13947162
2020-01-01 00:04:00  329.6  330.2  324.8  324.8  13947162
...
2020-01-01 00:16:00  333.2  334.8  331.2  334.0  12428539
2020-01-01 00:17:00  333.0  333.6  326.8  333.6  15533405
2020-01-01 00:18:00  335.0  335.2  326.2  327.2  16655874
2020-01-01 00:19:00  327.0  327.2  322.0  323.0  15086985
```

You must have figured it out that the data frame now has `DatetimeIndex`es.

But it will not become a 15-minute kline data unless we cumulate it, and only cumulates new frames if you use `stock.cum_append(them)` to cumulate `them`.

```
stock_15m = stock.cumulate()

print(stock_15m)
```

Now we get a 15-minute kline

```
                     open   high    low  close      volume
2020-01-01 00:00:00  329.4  334.2  324.8  324.8  62346461.0
2020-01-01 00:05:00  325.0  327.8  316.2  322.0  82176419.0
2020-01-01 00:10:00  323.0  327.8  314.6  327.6  74409815.0
2020-01-01 00:15:00  330.0  335.2  322.0  323.0  82452902.0
```

For more details and about how to get full control of everything, check the online Google Colab notebook here.

CHAPTER 4

Syntax of `directive`

```
directive := command | command operator expression
operator := '/' | '\' | '><' | '<' | '<=' | '==' | '>=' | '>'
expression := float | command

command := command_name | command_name : arguments
command_name := main_command_name | main_command_name.sub_command_name
main_command_name := alphabets
sub_command_name := alphabets

arguments := argument | argument , arguments
argument := empty_string | string | ( directive )
```

4.1 `directive` Example

Here lists several use cases of column names

```
# The middle band of bollinger bands
#   which is actually a 20-period (default) moving average
stock['boll']

# kdj j less than 0
# This returns a series of bool type
stock['kdj.j < 0']

# kdj %K cross up kdj %D
stock['kdj.k / kdj.d']

# 5-period simple moving average
stock['ma:5']

# 10-period simple moving average on open prices
```

(continues on next page)

(continued from previous page)

```
stock['ma:10,open']

# Dataframe of 5-period, 10-period, 30-period ma
stock[[
    'ma:5',
    'ma:10',
    'ma:30'
]]

# Which means we use the default values of the first and the second parameters,
# and specify the third parameter
stock['macd:,,10']

# We must wrap a parameter which is a nested command or directive
stock['increase:(ma:20,close),3']

# stock-pandas has a powerful directive parser,
# so we could even write directives like this:
stock['''
repeat
    :
        (
            column:close > boll.upper
        ),
        5
''']
```

CHAPTER 5

Built-in Commands of Indicators

Document syntax explanation:

- **param0** `int` which means `param0` is a required parameter of type `int`.
- **param1?** `str='close'` which means parameter `param1` is optional with default value `'close'`.

Actually, all parameters of a command are of string type, so the `int` here means an interger-like string.

5.1 `ma`, simple Moving Averages

```
ma:<period>,<column>
```

Gets the `period`-period simple moving average on column named `column`.

`SMA` is often confused between simple moving average and smoothed moving average.

So `stock-pandas` will use `ma` for simple moving average and `smma` for smoothed moving average.

- **period** `int` (required)
- **column?** `enum<'open'|'high'|'low'|'close'>='close'` Which column should the calculation based on. Defaults to `'close'`

```
# which is equivalent to `stock['ma:5,close']`
stock['ma:5']

stock['ma:10,open']
```

5.2 `ema`, Exponential Moving Average

```
ema:<period>,<column>
```

Gets the Exponential Moving Average, also known as the Exponential Weighted Moving Average.

The arguments of this command is the same as `ma`.

5.3 `macd`, Moving Average Convergence Divergence

```
macd:<fast_period>,<slow_period>
macd.signal:<fast_period>,<slow_period>,<signal_period>
macd.histogram:<fast_period>,<slow_period>,<signal_period>
```

- **fast_period?** `int=12` fast period (short period). Defaults to `12`.
- **slow_period?** `int=26` slow period (long period). Defaults to `26`
- **signal_period?** `int=9` signal period. Defaults to `9`

```
# macd
stock['macd']
stock['macd.dif']

# macd signal band, which is a shortcut for stock['macd.signal']
stock['macd.s']
stock['macd.signal']
stock['macd.dea']

# macd histogram band, which is equivalent to stock['macd.h']
stock['macd.histogram']
stock['macd.h']
stock['macd.macd']
```

5.4 `boll`, BOLLinger bands

```
boll:<period>,<column>
boll.upper:<period>,<times>,<column>
boll.lower:<period>,<times>,<column>
```

- **period?** `int=20`
- **times?** `float=2.`
- **column?** `str='close'`

```
# boll
stock['boll']

# bollinger upper band, a shortcut for stock['boll.upper']
stock['boll.u']
stock['boll.upper']

# bollinger lower band, which is equivalent to stock['boll.l']
stock['boll.lower']
stock['boll.l']
```

5.5 `rsv`, Raw Stochastic Value

```
rsv:<period>
```

Calculates the raw stochastic value which is often used to calculate KDJ

5.6 `kdj`, a variety of stochastic oscillator

The variety of Stochastic Oscillator indicator created by Dr. George Lane, which follows the formula:

```
RSV = rsv(period_rsv)
%K = ema(RSV, period_k)
%D = ema(%K, period_d)
%J = 3 * %K - 2 * %D
```

And the `ema` here is the exponential weighted moving average with initial value as `init_value`.

PAY ATTENTION that the calculation forumla is different from wikipedia, but it is much popular and more widely used by the industry.

Directive Arguments:

```
kdj.k:<period_rsv>,<period_k>,<init_value>
kdj.d:<period_rsv>,<period_k>,<period_d>,<init_value>
kdj.j:<period_rsv>,<period_k>,<period_d>,<init_value>
```

- **period_rsv?** `int=9` The period for calculating RSV, which is used for K%
- **period_k?** `int=3` The period for calculating the EMA of RSV, which is used for K%
- **period_d?** `int=3` The period for calculating the EMA of K%, which is used for D%
- **init_value?** `float=50.0` The initial value for calculating ema. Trading softwares of different companies usually use different initial values each of which is usually `0.0`, `50.0` or `100.0`.

```
# The %D series of KDJ
stock['kdj.d']
# which is equivalent to
stock['kdj.d:9,3,3,50.0']

# The KDJ serieses of with parameters 9, 9, and 9
stock[['kdj.k:9,9', 'kdj.d:9,9,9', 'kdj.j:9,9,9']]
```

5.7 `kdjc`, another variety of stochastic oscillator

Unlike `kdj`, `kdjc` uses **close** value instead of high and low value to calculate `rsv`, which makes the indicator more sensitive than `kdj`

The arguments of `kdjc` are the same as `kdj`

5.8 rsi, Relative Strength Index

```
rsi:<period>
```

Calculates the N-period RSI (Relative Strength Index)

- **period** `int` The period to calculate RSI. `period` should be an int which is larger than `1`

5.9 bbi, Bull and Bear Index

```
bbi:<a>,<b>,<c>,<d>
```

Calculates indicator BBI (Bull and Bear Index) which is the average of `ma:3`, `ma:6`, `ma:12`, `ma:24` by default

- **a?** `int=3`
- **b?** `int=6`
- **c?** `int=12`
- **d?** `int=24`

5.10 llv, Lowest of Low Values

```
llv:<period>,<column>
```

Gets the lowest of low prices in N periods

- **period** `int`
- **column?** `str='low'` Defaults to `'low'`. But you could also get the lowest value of close prices

```
# The 10-period lowest prices
stock['llv:10']

# The 10-period lowest close prices
stock['llv:10,close']
```

5.11 hhv, Highest of High Values

```
hhv:<period>,<column>
```

Gets the highest of high prices in N periods. The arguments of `hhv` is the same as `llv`

CHAPTER 6

Built-in Commands for Statistics

6.1 column

```
column:<name>
```

Just gets the series of a column. This command is designed to be used together with an operator to compare with another command or as a parameter of some statistics command.

- **name** `str` the name of the column

```
# A bool-type series indicates whether the current price is higher than the upper␣
↪bollinger band
stock['column:close > boll.upper']
```

6.2 increase

```
increase:<on>,<repeat>,<step>
```

Gets a `bool`-type series each item of which is `True` if the value of indicator `on` increases in the last `period`-period.

- **on** `str` the command name of an indicator on what the calculation should be based
- **repeat?** `int=1`
- **direction?** `1 | -1` the direction of "increase". `-1` means decreasing

For example:

```
# Which means whether the `ma:20,close` line
# (a.k.a. 20-period simple moving average on column `'close'`)
# has been increasing repeatedly for 3 times (maybe 3 days)
stock['increase:(ma:20,close),3']
```

(continues on next page)

(continued from previous page)

```
# If the close price has been decreasing repeatedly for 5 times (maybe 5 days)
stock['increase:close,5,-1']
```

6.3 style

```
style:<style>
```

Gets a `bool`-type series whether the candlestick of a period is of `style` style

- **style** `'bullish' | 'bearish'`

```
stock['style:bullish']
```

6.4 repeat

```
repeat:(<bool_directive>),<repeat>
```

The `repeat` command first gets the result of directive `bool_directive`, and detect whether `True` is repeated for `repeat` times

- **bool_directive** `str` the directive which should returns a series of `bools`. PAY ATTENTION, that the directive should be wrapped with parantheses as a parameter.
- **repeat?** `int=1` which should be larger than `0`

```
# Whether the bullish candlestick repeats for 3 periods (maybe 3 days)
stock['repeat:(style:bullish),3']
```

6.5 change

```
change:<on>,<period>
```

Percentage change between the current and a prior element on a certain series

Computes the percentage change from the immediately previous element by default. This is useful in comparing the percentage of change in a time series of prices.

- **on** `str` the directive which returns a series of numbers, and the calculation will based on the series.
- **period?** `int=2` `2` means we computes with the start value and the end value of a 2-period window.

```
# Percentage change of 20-period simple moving average
stock['change:(ma:20)']
```

CHAPTER 7

Operators

```
left operator right
```

7.1 Operator: /

whether `left` crosses through `right` from the down side of `right` to the upper side which we call it as "cross up".

7.2 Operator: \

whether `left` crosses down `right`.

```
# Which we call them "dead crosses"
stock['macd \\ macd.signal']
```

PAY ATTENTION, in the example above, we should escape the backslash, so we've got double backslashes `'\\'`

7.3 Operator: ><

whether `left` crosses `right`, either up or down.

7.4 Operator: < | <= | == | >= | >

For a certain record of the same time, whether the value of `left` is less than / less than or equal to / equal to / larger than or equal to / larger than the value of `right`.

CHAPTER 8

Errors

```
from stock_pandas import (
    DirectiveSyntaxError,
    DirectiveValueError
)
```

8.1 DirectiveSyntaxError

Raises if there is a syntax error in the given directive.

```
stock['''
repeat
    :
        (
            column:close >> boll.upper
        ),
        5
''']
```

`DirectiveSyntaxError` might print some messages like this:

```
File "<string>", line 5, column 26

   repeat
       :
           (
>              column:close >> boll.upper
           ),
           5

                             ^
DirectiveSyntaxError: ">>" is an invalid operator
```

8.2 DirectiveValueError

Raises if

- there is an unknown command name
- something is wrong about the command arguments
- etc.

CHAPTER 9

Advanced Sections

How to extend stock-pandas and support more indicators,

This section is only recommended for contributors, but not for normal users, for that the definition of `COMMANDS` might change in the future.

```
from stock_pandas import COMMANDS, CommandPreset
```

To add a new indicator to stock-pandas, you could update the `COMMANDS` dict.

```
# The value of 'new-indicator' is a tuple
COMMANDS['new-indicator'] = (
    # The first item of the tuple is a CommandPreset instance
    CommandPreset(
        formula,
        args_setting
    ),
    sub_commands_dict,
    aliases_of_sub_commands
)
```

You could check here to figure out the typings for `COMMANDS`.

For a simplest indicator, such as simple moving average, you could check the implementation here.

9.1 formula(df, s, *args) -> Tuple[np.ndarray, int]

`formula` is a `Callable[[StockDataFrame, slice, ...], [ndarray, int]]`.

- **df** `StockDataFrame` the first argument of `formula` is the stock dataframe itself
- **s** `slice` sometimes, we don't need to calculate the whole dataframe but only part of it. This argument is passed into the formula by `stock_pandas` and should not be changed manually.
- **args** `Tuple[Any]` the args of the indicator which is defined by `args_setting`

The Callable returns a tuple:

- The first item of the tuple is the calculated result which is a numpy ndarray.
- The second item of the tuple is the mininum periods to calculate the indicator.

9.2 args_setting: [(default, validate_and_coerce), ...]

`args_setting` is a list of tuples.

- The first item of each tuple is the default value of the parameter, and it could be `None` which implies it has no default value and is required.
- The second item is a raisable callable which receives user input, validates it, coerces the type of the value and returns it. If the parameter has a default value and user don't specified a value, the function will be skipped.

9.3 sub_commands_dict: Dict[str, CommandPreset]

A dict to declare sub commands, such as `boll.upper`.

`sub_commands_dict` could be `None` which indicates the indicator has no sub commands

9.4 aliases_of_sub_commands: Dict[str, Optional[str]]

Which declares the shortcut or alias of the commands, such as `boll.u`

```
dict(
    u='upper'
)
```

If the value of an alias is `None`, which means it is an alias of the main command, such as `macd.dif`

```
dict(
    dif=None
)
```

CHAPTER 10

Development

First, install conda (recommended), and generate a conda environment for this project

```
conda create -n stock-pandas python=3.11

conda activate stock-pandas

# Install requirements
make install

# Build python ext (C++)
make build-ext

# Run unit tests
make test
```