crypto$_{h}istory$

Release 1.2.b14

May 14, 2022

Contents:

CHAPTER 1

Welcome to crypto-history

This is a wrapper on binance and other exchange APIs to aggregate historical information in structured tabular formats (such as xarray.DataArray and SQLite).

Source code https://github.com/vikramaditya91/crypto_history

Documentation https://crypto-history.readthedocs.io/en/latest/

1.1 Features

- Cleans the data ticker-wise if incomplete
- Sets the correct type on the data obtained
- Is able to join data from various chunks of time in a single DataArray
- Candles of varying intervals can be obtained in a single DataArray
- Fetches information about all tickers available on Binance asynchronously
- Delays requests if it is close to the limit prescribed by Binance
- Retries when the requests have exceeded the performance limit of the machine
- Obtains the history of each/all tickers in the xarray.DataArray format

- Easily extendable to other exchanges and other data formats
- It does not require an API key from Binance
- null values can be dropped either timestamp-wise and/or coin-wise
- Can export data in SQLite format and xr.DataArray
- Chunks of time can be aggregated into a single data object

1.2 Quick Start

```
pip install crypto-history
```

See a basic example at :examples/binance_basic.py

```
exchange_factory = class_builders.get("market").get("binance")()

desired_fields = ["open_ts", "open"]

binance_homogenizer = exchange_factory.create_data_homogenizer()
base_assets = await binance_homogenizer.get_all_base_assets()
print(f"All the base assets available on the Binance exchange are {base_assets}")

time_range = {("25 Jan 2020", "27 May 2020"): "1d",
              ("26 Aug 2020", "now"):         "1h"}
time_aggregated_data_container = data_container_intra.TimeAggregatedDataContainer(
    exchange_factory,
    base_assets=["NANO", "IOST", "XRP"],
    reference_assets=["BTC"],
    ohlcv_fields=desired_fields,
    time_range_dict=time_range
)
xdataarray_of_coins = await time_aggregated_data_container.get_time_aggregated_data_
↪container()
pprint(xdataarray_of_coins)
```

For more check out the documentation.

1.2.1 Market Data

The following provide information on obtaining market data from crypto-currency exchanges and market APIs.

class crypto_history.stock_market.stock_market_factory.**StockMarketFactory**
Abstract factory to generate factories per exchange

static create_data_homogenizer() → crypto_history.stock_market.stock_market_factory.AbstractMarketHomogeni
Create an instance of a market homogenizer which is the interface to the market from the outside. It should ensure that the market data from various markets/exchanges which might have different low-level APIs is available in a uniform format.

static create_market_operations() → crypto_history.stock_market.stock_market_factory.AbstractMarketOperation
Create the instance of the class to channel the right requests to the low level market requester.

static create_market_requester() → crypto_history.stock_market.request.AbstractMarketRequester
Create the low-level market requester instance per exchange

static create_ohlcv_field_types() → crypto_history.stock_market.stock_market_factory.AbstractOHLCVFieldTy

static create_time_interval_chunks() → crypto_history.stock_market.stock_market_factory.AbstractTimeInterva

class crypto_history.stock_market.stock_market_factory.**ConcreteBinanceFactory**
Binance's factory for creating factories

create_data_homogenizer() → crypto_history.stock_market.stock_market_factory.BinanceHomogenizer
Creates the instance of the Binance Market Homogenizer

Returns Instance of BinanceHomogenizer

Return type *BinanceHomogenizer*

create_market_operations() → crypto_history.stock_market.stock_market_factory.BinanceMarketOperations
Creates the instance of the Binance Market Operator

Returns Instance of BinanceMarketOperator

Return type *BinanceMarketOperations*

class crypto_history.stock_market.stock_market_factory.**AbstractMarketHomogenizer**
Synthesizes the information obtained from various different market operator to a consistent format

OHLCVFields = None

get_all_base_assets()

get_all_coins_ticker_objects() → crypto_history.stock_market.tickers.TickerPool
Generates the standard/uniform TickerPool object which should be consistent no matter which exchange it is coming from

Returns TickerPool which contains all the different tickers and holds it in one

Return type TickerPool

get_all_raw_tickers()

get_all_reference_assets()

get_history_for_ticker(**args*, ***kwargs*) → map
Gets the history of the ticker symbol

Parameters

- ***args** – unknown, as it depends on the exchange
- ****kwargs** – unknown, as it depends on the exchange

Returns map object of history of the ticker

Return type map

get_named_ohlcv_tuple()

get_ticker_instance(**args*, ***kwargs*)
Gets the standard ticker dataclass object. Note that the fields may be dependent on the exchange that it is coming from

Parameters

- ***args** – unknown, as it depends on the exchange
- ****kwargs** – unknown, as it depends on the exchange

Returns dataclass instance of the symbol info

Return type dataclass

class crypto_history.stock_market.stock_market_factory.**BinanceHomogenizer**

get_all_base_assets() → List[T]
Obtains the list of all base assets

get_all_coins_ticker_objects
Obtains the exchange-independent TickerPool from all the tickers/symbols that are available on the exchange

Returns All tickers and their information stored in the TickerPool defined in

Return type TickerPool

TODO

get_all_raw_tickers() → List[T]
Obtains the list of all raw tickers available on binance

get_all_reference_assets() → List[T]
Obtains the list of all reference assets

get_exchange_assets(*type_of_asset: str*) → Generator[T_co, T_contra, V_co]
Obtains the type of asset from the exchange. :param type_of_asset: string identifier that the binance API uses to identify the key in each symbol :type type_of_asset: str

Returns The generator of items that are available in the exchange

Return type Generator

get_exchange_info()

get_history_for_ticker(**args*, ***kwargs*) → map
Gets the history of the ticker for the desired duration,

TODO Probably yank the doc from MarketOperator to here

Parameters

- ***args** – See the `BinanceMarketOperator.get_raw_history_for_ticker` for arguments
- ****kwargs** – See the `BinanceMarketOperator.get_raw_history_for_ticker` for arguments

Returns map of the history of the ticker mapped to the OHLCVFields namedtuple

Return type map

get_set_of_ticker_attributes(*attribute: str*)
Aggregates the set items of the required attribute across the whole history

Parameters **attribute** – The attribute/key which is the common term in the history whose values are to be identified

Returns set of values whose attributes are common

Return type set

get_ticker_instance(*ticker_name: str*)
Obtains the TickerDataclass based on the string of the ticker name provided

Parameters **ticker_name** (*str*) – ticker name whose standard Ticker dataclass object is desired

Returns instance of the dataclass of the ticker

Return type dataclass

class crypto_history.stock_market.stock_market_factory.**AbstractMarketOperations**
Abstract Base Class to serve as the parent for all market operators.

Seeing that the market requester may respond with different formats, it is not possible to know the signature of the method. The arguments and the return values are unknown by the AbstractMarketOperator

get_all_raw_tickers(**args*, ***kwargs*)
Obtain all the tickers from the low-level market requester. Seeing that the market requester may respond with different formats, it is not possible to know the signature of the method

get_exchange_info(**args*, ***kwargs*)
Gets general properties of the exchange

get_raw_history_for_ticker(**args*, ***kwargs*)
Obtain the raw history of a particular ticker.

get_raw_symbol_info(**args*, ***kwargs*)
Obtains the information on the particular ticker. It is not known what are the exact information that is going to be received because the information is coming from various exchanges

class crypto_history.stock_market.stock_market_factory.**BinanceMarketOperations**
Binance's market operator. Implements methods to get market knowledge

get_all_raw_tickers()
Gets all the tickers available on the binance exchange.

Returns All the raw tickers obtained from the python-binance (`python-binance`) In binance, the format of each raw ticker is {"symbol": <>, "price": <>}

Return type list

get_exchange_info()
Obtains the complete information available at the exchange Returns:

get_raw_history_for_ticker(*ticker: str*, *interval: str*, *start_time: int*, *end_time: int*) → List[T]
Gets the kline history of the ticker from binance exchange

Parameters

- **ticker** (*str*) – ticker whose history has to be pulled
- **interval** (*str*) – interval of the history (eg. 1d, 3m, etc). See `binance.enums()` in `python-binance`
- **start_time** (*int*) – Start date string in exchange-format
- **end_time** (*int*) – End date string in exchange-format

Returns List of snapshots of history. Each snapshot is a list of collection of OHLCV values. See details in `BinanceHomogenizer.OHLCVFields`

Return type list

get_raw_symbol_info(*symbol: str*)
Obtains the information for the symbol/ticker requested

Parameters **symbol** (*str*) – symbol of the ticker whose information is desired

Returns The raw information of the ticker desired with information where the keys are the baseAsset, precision, quoteAsset, etc. See `binance.AsyncClient.get_symbol_info()` in `python-binance`

Return type dict

class crypto_history.stock_market.request.**AbstractMarketRequester**
AbstractBaseClass for the low-level market requester

request(*method_name: str*, **args*, ***kwargs*)
Interface to the user of the low level request made to the API server

Parameters

- **method_name** – name of the method to be called on the client object.
- ***args** – arguments transferred by the market operator
- ****kwargs** – arguments transferred by the market operator

Returns response from the market/exchange

class crypto_history.stock_market.request.**BinanceRequester**
Low level requester for the Binance API. api_key and api_secret are not required to get market information. Limit for requests are officially set at 2400 per minute. However, it is throttled to 500 per minute

class crypto_history.stock_market.stock_market_factory.**AbstractOHLCVFieldTypes**

class OHLCVFields

get_dict_name_type()

get_named_tuple()

class crypto_history.stock_market.stock_market_factory.**BinanceRequester**
Low level requester for the Binance API. api_key and api_secret are not required to get market information. Limit for requests are officially set at 2400 per minute. However, it is throttled to 500 per minute

class crypto_history.stock_market.stock_market_factory.**AbstractTimeIntervalChunks**
Abstract class to handle the chunking of dates as the exchanges have limits on the length of the interval of history

get_exchange_specific_sub_chunks(*sub_chunks: List[tuple]*) → List[tuple]
Gets the exchange specific sub-chunk from default sub-chunks :param sub_chunks: default sub-chunks to be converted to exchange specific :type sub_chunks: list

Returns List of exchange-specific formatted sub-chunks

get_time_range_for_historical_calls(*raw_time_range_dict: Dict[KT, VT]*) → List[tuple]
Obtains the time ranges for making the historical calls.

Parameters

- **raw_time_range_dict** – Dictionary of the
- **(start** – str, end:str): (type_of_interval: str)

Returns List of the tuples of the exchange-specific format. ((start_time, end_time), type_of_interval)

limit = inf

static sanitize_datetime_to_exchange_specific(*datetime_obj: datetime.datetime*)
Converts the datetime object to exchange specific format

static sanitize_item_to_datetime_object(*item_to_parse: str*) → datetime.datetime
Converts/sanitizes the string to the datetime.datetime object .. rubric:: Notes

Timezone is not handled. The local timezone is considered by dateutil's parser

Parameters **item_to_parse** (*str*) – the item that has to be converted

Returns datetime.datetime object from the string

url = ''

class crypto_history.stock_market.stock_market_factory.**BinanceTimeIntervalChunks**
Binance specific information for the interval generation

limit = 1000

static sanitize_datetime_to_exchange_specific(*datetime_obj: datetime.datetime*) → int
Converts the datetime object to binance specific format for making the requests :param datetime_obj: datetime.datetime object which needs to be converted

Returns binance specific format

url = 'https://github.com/binance-exchange/binance- official-api-docs/blob/master/rest

1.2.2 Data Container Pre-process

The following provide information on how the primitive data array is built

class crypto_history.data_container.data_container_pre.**PrimitiveCoinHistoryObtainer**
Abstract class to serve as the parent to generating histories obtainer classes

classmethod create_primitive_coin_history_obtainer(*exchange_factory: crypto_history.stock_market.stock_market_factory.St interval: str, start_time: Union[str, datetime.datetime, int], end_time: Union[str, datetime.datetime, int]*) → crypto_history.data_container.data_container_pre.Pri
Generates the coin history obtainer

Parameters

- **exchange_factory** (StockMarketFactory) – instance of the exchange_factory which is responsible for setting the market_homogenizer
- **interval** (*str*) – Length of the history of the klines per item
- **start_time** (*str/datetime/int*) – duration from which history is necessary
- **end_time** (*str/datetime/int*) – duration up to which history is necessary

Yields PrimitiveCoinHistoryObtainer instance from the above arguments

get_all_raw_tickers() → crypto_history.stock_market.tickers.TickerPool
Obtains the ticker pool by accessing the market homogenizer

Returns The TickerPool filled with all the ticker available to the market/exchange

Return type TickerPool

get_historical_data_from_base_and_reference_assets(*base_assets: List[T], reference_assets: List[T]*) → Dict[KT, VT]
Get the historical data for the combination of base and reference assets :param base_assets: list of all base assets :type base_assets: List[‘str’] :param reference_assets: list of all reference assets :type reference_assets: List[‘str’]

Returns The kline historical data of the combinations

initialize_example() → List[T]
Initializes an example of the raw history and stores it example_raw_history

Returns instance of an example of a raw history of ETHvsBTC

Return type list

class crypto_history.data_container.data_container_pre.**PrimitiveDimensionsManager**
Class responsible for managing the dimensions/coordinates of the data container

class Dimensions(*base_assets: List[T], reference_assets: List[T], ohlcv_fields: List[T], index_number: List[T]*)
Class for keeping track of the dimensions of the XDataArray

classmethod create_primitive_dimensions_manager(*history_obtainer: crypto_history.data_container.data_container_pre.Primit*) → crypto_history.data_container.data_container_pre.Primitiv
Initializes the dimensions manager with the coin_history_obtainer

get_depth_of_indices() → List[T]

Obtains the depth of the indices to identify how many time-stamp values each history has available

Returns

The list of indices from 0..n-1 where n corresponds to the number of time-stamps in the history

Return type list

get_dimension_coordinates_from_fields_and_assets(*ohlcv_fields: List[T], base_assets: List[T], reference_assets: List[T]*)
Gets the dataclass containing the coordinates required for creating the xr.DataArray :param ohlcv_fields: fields related to open_ts, close_ts, and so on :param base_assets: list of base coins :param reference_assets: list of reference coins

Returns DataClass of the coordinates necessary to build the DataArray

static set_coords_dimensions_in_empty_container(*dataclass_dimensions_coordinates*) → xarray.core.dataarray.DataArray
Initialize the xarray container with None values but with the coordinates as it helps in the memory allocation :param dataclass_dimensions_coordinates: dataclass of coordinates/dimensions as the framework for generating the XDataArray

class crypto_history.data_container.data_container_pre.**PrimitiveDataArrayOperations**
Abstract Base Class for generating the data operations

static append_column_to_df(*df, column_name, column_value*) → pandas.core.frame.DataFrame
Appends a new column to a df with constant value :param df: dataframe to which column is to be appended :type df: DataFrame :param column_name: name of the column :type column_name: str :param column_value: value of the column :type column_value: str

Returns new column added DataFrame

classmethod create_primitive_data_array_operations(*exchange_factory: crypto_history.stock_market.stock_market_factory.St base_assets: List[T], reference_assets: List[T], ohlcv_fields: List[T], interval: str, start_time: Union[str, datetime.datetime, int], end_time: Union[str, datetime.datetime, int]*)

Initializes the DataContainerOperations which is the user end-point for the basic data building :param exchange_factory: factory of the exchange :type exchange_factory: StockMarketFactory :param base_assets: list of base coins to be accumulated :type base_assets: List :param reference_assets: list of reference/quote-against coins to be accumulated :type reference_assets: List :param ohlcv_fields: list of fields for the various fields :type ohlcv_fields: List :param interval: data capture interval :type interval: str :param start_time: date from which data collection should start :type start_time: str/datetime/int :param end_time: date up to which data collection should be made :type end_time: str/datetime/int

Yields PrimitiveDataArrayOperations instance from the above arguments

get_example_history() → List[T]

A standard example on which the gaps are padded

get_populated_primitive_container() → xarray.core.dataarray.DataArray

Populates the container and returns it to the use

Returns the filled container of information

Return type xr.DataArray

populate_container(*data_container: xarray.core.dataarray.DataArray*, *coord_dimension_dataclass*) → xarray.core.dataarray.DataArray

Populates the xarray.DataArray with the historical data of all the coins

class crypto_history.data_container.utilities.**DataFrameOperations**

Operations purely dedicated to dataframe

static add_extra_rows_to_bottom(*df: pandas.core.frame.DataFrame*, *empty_rows_to_add: int*)

Adds extra rows to the pd.DataFrame

Parameters

- **df** (*pd.DataFrame*) – to which extra rows are to be added
- **empty_rows_to_add** (*int*) – Number of extra rows that have to be added

Returns Re-indexed pd.DataFrame which has the compatible number of rows

Return type pd.DataFrame

static calculate_rows_to_add(*df: pandas.core.frame.DataFrame, list_of_standard_history: List[T]*) → int

Calculates the additional number of rows that might have to be added to get the df in the same shape

Parameters

- **df** (*pd.DataFrame*) – pandas dataframe which is obtained for the coin's history
- **list_of_standard_history** – expected standard history which has the complete history

Returns number of rows that have to be added to the df

Return type int

static drop_unnecessary_columns_from_df(*df, necessary_columns: List[T]*) → pandas.core.frame.DataFrame

Drop all columns which are not necessary from the df :param df: from which the unnecessary columns are to be dropped :type df: pd.DataFrame :param necessary_columns: list of columns which are to be stored :type necessary_columns: list

Returns pd.DataFrame where the unnecessary columns are dropped

get_compatible_df(*standard_example: List[T], ticker_history: Iterable[T_co]*) → pandas.core.frame.DataFrame

Makes the ticker history compatible to the standard pd.DataFrame by extending the shape to add null values

Parameters

- **standard_example** (*DataFrame*) – standard example to know how many rows to pad
- **ticker_history** – history of the current ticker

Returns compatible history of the df adjusted for same rows as expected

Return type pd.DataFrame

pad_extra_rows_if_necessary(*standard_example: List[T], history_df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Add extra rows on the bottom of the DF is necessary. i.e when the history is incomplete. # FixMe . Probably needs to be reevaluated :param standard_example: standard example on which it is based :type standard_example: DataFrame :param history_df: history of the dataframe that has not been padded yet

Returns that has been padded

Return type pd.DataFrame

1.2.3 Data Container Access

The following provide information on how the high level operations on the data container are performed

class crypto_history.data_container.data_container_access.**TimeStampIndexedDataContainer**

Responsible for transforming the Primitive DataArray to a DataArray indexed by the timestamp of choice with possibility to approximate for easier handling

calculate_tolerance_value_from_ratio(*ratio: float = 0.001*) → float

Calculates the tolerance value based on the tolerance ratio :param ratio: ratio of the tolerance :type ratio: float

Returns float, value of the tolerance value

classmethod create_time_stamp_indexed_data_container(*exchange_factory: crypto_history.stock_market.stock_market_factory base_assets: List[T], reference_assets: List[T], reference_ticker: tuple, aggregate_coordinate_by: str, ohlcv_fields: List[T], weight: str, start_time: Union[str, datetime.datetime, int], end_time: Union[str, datetime.datetime, int]*)

The factory for creating the time stamp indexed data container :param exchange_factory: The exchange factory :type exchange_factory: StockMarketFactory :param base_assets: List of base assets :type base_assets: List :param reference_assets: List of reference assets :type reference_assets: List :param reference_ticker: ('xxx', 'yyy') where xxx is the base_asset and yyy is the reference asset for indexing the timestamp :type reference_ticker: tuple :param aggregate_coordinate_by: The direction in which the coordinates should be aggregated by :type aggregate_coordinate_by: str :param ohlcv_fields: list of ohlcv-fields necessary to capture :type ohlcv_fields: List :param weight: weight/interval of the kline/candle :type weight: str :param start_time: start time/date of the candles :type start_time: str/datetime.datetime/int :param end_time: end time/date of the candles :type end_time: str/datetime.datetime/int

Yields TimeStampIndexedDataContainer constructed with above details

static generate_empty_dataarray_indexed_by_timestamp(*coords: Dict[KT, VT]*) → xarray.core.dataarray.DataArray

Generates the new dataarray from the coordinates provided :param coords: the coordinates for the new dataarray :type coords: Dict

Returns xr.DataArray the new empty dataarray

generate_empty_df_with_new_timestamps(*do_approximation: bool*, *old_dataarray: xarray.core.dataarray.DataArray*) → xarray.core.dataarray.DataArray

Generates an empty dataframe with new timestamps in the coordinates :param do_approximation: Check if the timestamps can be approximated :type do_approximation: bool :param old_dataarray: The old dataarray used for reference :type old_dataarray: xr.DataArray :param for constructing new dataarray:

Returns xr.DataArray The new empty dataarray which has the coordinates set

static get_all_unique_values(*dataarray: xarray.core.dataarray.DataArray*, *coord_name: str*, *field_in_coord: str*) → List[T]

Gets all the unique values in the xr.DataArray in the particular item of the coordinate :param dataarray: whose data is to be selected :type dataarray: xr.DataArray :param coord_name: name of the coordinate which is the primary selector :type coord_name: str :param field_in_coord: name of the item in the coordinate which is the second level of select :type field_in_coord: str

Returns list of all the items in the selected coordinate, field sorted increasingly

get_coords_for_timestamp_indexed_datarray(*old_dataarray: xarray.core.dataarray.DataArray, reference_ts: List[int]*) → Dict[KT, VT]

Gets the coordinates for the timestamp index dataarray :param old_dataarray: The old dataarray taken as the building block for the new dataarray :type old_dataarray: xr.DataArray :param reference_ts: The new timestamps which serve as the references instead of index_number :type reference_ts: list

Returns dictionary of coordinates for the new dataarray

static get_df_to_insert(*sub_dataarray: xarray.core.dataarray.DataArray, index_of_integrating_ts: int, reference_ts: List[T], tolerance: float*) → pandas.core.frame.DataFrame

Calculates the dataframe from the dataarray with one reduced shape size :param sub_dataarray: the sub dataarray whose dataframe is to be extracted :type sub_dataarray: xr.DataArray :param index_of_integrating_ts: index of the integrating field in the coordinate :type index_of_integrating_ts: int :param reference_ts: list of the indexes according to what the dataframe needs to be adjusted :type reference_ts: List :param tolerance: the value of the tolerance which can be considered while matching indices :type tolerance: float

Returns pd.DataFrame the reindexed dataframe which contains the data in the required indices

Raises EmptyDataFrameException if the sub_dataarray contains only na values

static get_index_of_field(*data_array: xarray.core.dataarray.DataArray*, *coord_name: str*, *item_in_coord: str*) → int

Gets the index of the particular field in the coordinate :param data_array: in which the index is to be identified :type data_array: xr.DataArray :param coord_name: name of the coordinate :type coord_name: str :param item_in_coord: name of the field in the coordinate :type item_in_coord: str

Returns int, the index number which can be used to identify the item

get_primitive_full_xr_dataarray() → xarray.core.dataarray.DataArray

Gets the full data container :returns: the complete data container :rtype: xr.DataArray

get_primitive_reference_xr_dataarray() → xarray.core.dataarray.DataArray

Gets the reference data container :returns: the reference data container (eg. ETHBTC history) :rtype: xr.DataArray

static get_primitive_xr_dataarray(*data_container_object: crypto_history.data_container.data_container_pre.PrimitiveDataArrayOperat*) → xarray.core.dataarray.DataArray

Gets the primitive xr.DataArray from the data container object :returns: the actual xr.DataArray :rtype: xr.DataArray

get_timestamps_for_new_dataarray(*do_approximation: bool*, *dataarray: xarray.core.dataarray.DataArray*) → List[T]

Obtains the time stamp for the new dataarray. It may either select it from the reference dataarray or get it from the actual dataarray if approximation is not desired :param do_approximation: if True, timestamps are approximated and taken from the reference dataarray. if False, timestamps are not approximated. All timestamps from all various tickers will be the coordinate for the timestamp :type do_approximation: bool :param dataarray: the original dataarray which contains all the timestamps in it :type dataarray: xr.DataArray

Returns list of the timestamps for the new dataarray

get_timestamps_of_reference_dataarray(*field_in_coord: str*) → List[T]

Obtains the time stamp list from the reference dataarray :param field_in_coord: the item in the ohlcv_fields which is of interest :type field_in_coord: str

Returns list of the time stamps from the reference dataarray

get_value_of_tolerance(*do_approximation: bool*, *tolerance_ratio: float = 0.001*)

Gets the value of tolerance used for reindexing :param do_approximation: Check if approximation can be made :type do_approximation: bool :param tolerance_ratio: The ratio of maximum tolerance of time difference for generating coordinates of timestamp :type tolerance_ratio: float

Returns float, value of the tolerance

get_xr_dataarray_indexed_by_timestamps(*do_approximation: bool = True*, *tolerance_ratio: float = 0.001*) → xarray.core.dataarray.DataArray

Gets the xr.DataSet of the coin histories of the particular chunk. Obtains the data and then transforms it to the xr.DataSet :param do_approximation: check if the timestamps can be approximated to the reference datarray :type do_approximation: bool :param tolerance_ratio: the ratio of the maximum tolerance for :type tolerance_ratio: float :param approximations of timestamps:

Returns xr.DataSet data of the coin history

class crypto_history.data_container.data_container_access.**TimeAggregatedDataContainer**

Aggregates various time-ranges into the data container

static concatenate_dataarray_in_coord(**dataarray*, *coordinate='timestamp'*) → xarray.core.dataarray.DataArray

Concatenates two histories in the desired coordinate/dimension :param dataarray List[xr.DataArray]:

List/Tuple of xr.DataArrays that need to be concatednated :param coordinate: coordinate direction to concat by

Returns Concatenated xr.DataArray consisting of all the data concatenated

classmethod create_instance(*exchange_factory: crypto_history.stock_market.stock_market_factory.StockMarketFact base_assets: List[str], reference_assets: List[str], ohlcv_fields: List[str], time_range_dict: Dict[Tuple[Union[datetime.timedelta, str], Union[datetime.timedelta, str]], str], reference_ticker: Tuple = ('ETH', 'BTC'), aggregate_coordinate_by: str = 'open_ts'*) → TypeVarPlaceHolder

Creates the time-aggregator from time-deltas which go back from the current time

Parameters

- **exchange_factory** (StockMarketFactory) – The exchange factory
- **base_assets** (*List*) – base-asset coins
- **reference_assets** (*List*) – reference-asset coins
- **ohlcv_fields** (*List*) – ohlcv-fields to calculate
- **time_range_dict** (*Dict*) – dictionary of time-ranges where the keys are tuples of timedeltas
- **reference_ticker** (*Tuple*) – reference-ticker for reference timestamps
- **aggregate_coordinate_by** (*str*) – Timestamp indexed by this value

Returns TimeAggregatedDataContainer

get_chunk_history(*kline_width: str, start_time: Union[str, datetime.datetime, int], end_time: Union[str, datetime.datetime, int]*) → xarray.core.dataarray.DataArray

Gets the history of the particular chunk whose start and end times are well defined :param kline_width: The exchanges interval :type kline_width: str :param start_time: the start-time of the chunk of history :type start_time: datetime/str/int :param end_time: (datetime/str/int): the end-time of the chunk of history

Returns xr.DataArray the history of the chunk

static get_sorted_dataarray(*dataarray: xarray.core.dataarray.DataArray*, *coordinate: str*) → xarray.core.dataarray.DataArray

Sorts the given dataarray in the given coordinate direction :param dataarray: The dataarray which has to be sorted :type dataarray: xr.DataArray :param coordinate: sort the dataarray in the particular direction of the coordinate

Returns xr.DataArray The sorted dataarray

get_time_aggregated_data_container(*sort: bool = True*) → xarray.core.dataarray.DataArray

Gets time aggregated data container by splitting the containers as required

Parameters **sort** (*bool*) – to check if the data is to be sorted in an increasing order

Returns xr.DataArray The complete history of the interval

get_time_interval_chunks(*time_range: Dict[KT, VT]*) → List[tuple]

Gets the time interval chunks from the raw dict provided :param time_range: Dict with key=tuple of start and end time value=type of klines/intervals

Returns

List of tuples with the individual requests for histories ((start time, end time), interval)

1.2.4 Post Processing Data

The following provide information on how the historical data obtained in the form of xarray.DataArray as described here can be post-processed.

You can find an example on post-processing at :examples/coin_history_post_process.py

The currently offered post-processing capabilities are:

Type Conversion

The original data obtained from the exchange may or may not be set with the correct type. An example of this is Binance which provides the *open* (the opening value of the ticker) as a string. The type converter stores the same value as a float.

The types are stored in *the OHLCVFields*

class crypto_history.data_container.data_container_post.**TypeConvertedData**

Type converts the data in the dataarray/dataset

get_ohlcv_field_type_dict() → Dict[KT, VT]

Gets the field types of the OHLCV Fields converted to the numpy/pandas format. This is done to be able to handle nan values in Int/String types

Returns (dict): Dictionary of the map from the ohlcv-field to the pd/np type

set_type_on_dataarray(*dataarray: xarray.core.dataarray.DataArray*) → xarray.core.dataarray.DataArray

Sets the type on the xr.DataArray according to the ohlcv field type :param dataarray: The DataArray on which the type has to be set :type dataarray: xr.DataArray

Returns xr.DataArray which has the type set on it

set_type_on_dataset(*dataset: xarray.core.dataset.Dataset*) → xarray.core.dataset.Dataset

Sets the type on the xr.DataSet according to the ohlcv field type

Parameters **dataset** (*xr.DataSet*) – The dataset on which the type has to be set

Returns xr.DataSet which has the type set on it

type_mapping = {<class 'int'>: <class 'pandas.core.arrays.integer.Int64Dtype'>, <clas

Incomplete Data Deletion

Incomplete data from the xarray.DataArray or xarray.DataSet may have to be removed to avoid unexpected behaviour and to save memory. It offers removal of incomplete data in two ways. If all the data corresponding to a particular base or reference asset is not available, it can remove that coin from the xarray item. If one of the values corresponding to a particular ticker is nan, it can make the entire ticker contents nan.

class crypto_history.data_container.data_container_post.**HandleIncompleteData**

Responsible for handling missing data: 1. If a certain coin has to be dropped if it is null 2. If a ticker has to be nulliifed as it has incomplete data

drop_xarray_coins_with_entire_na(*data_item: Union[xarray.core.dataarray.DataArray, xarray.core.dataset.Dataset]*) → Union[xarray.core.dataarray.DataArray, xarray.core.dataset.Dataset]

Drops the coins from the base/reference asset if all its corresponding values are nan :param data_item: which contains information of the coin histories :type data_item: xr.DataArray/xr.DataSet

Returns xr.DataArray/xr.DataSet where the coins have been dropped

get_all_coord_combinations(*data_item: Union[xarray.core.dataarray.DataArray, xarray.core.dataset.Dataset]*)

Gets all the various combinations to iterate according to the coordinates to drop

Parameters **data_item** (*xr.DataArray/xr.DataSet*) – data_item whose combinations need to be iterated over

Yields A dict with various combinations

nullify_incomplete_data_from_dataarray(*dataarray: xarray.core.dataarray.DataArray*) → xarray.core.dataarray.DataArray

Nullifies incomplete data from the xr.DataArray :param dataarray: dataarray whose coordinates are to be nullified :type dataarray: xr.DataArray

Returns xr.DataArray whose data has been nullified if incomplete

nullify_incomplete_data_from_dataset(*dataset: xarray.core.dataset.Dataset*) → xarray.core.dataset.Dataset

Nullifies the incomplete data of datasets

Notes

Using indexing to assign values to a subset of dataset (e.g., ds[dict(space=0)] = 1) is not yet supported. http://xarray.pydata.org/en/stable/indexing.html

Parameters **dataset** (*xr.DataSet*) – dataset whose data is to be nullified

Returns xr.DataSet whose incomplete items are nullified

1.2.5 Utilities

The following provide information on general utilities used

class crypto_history.utilities.general_utilities.**TokenBucket**

Controls the number of requests that can be made to the API. All times are written in micro-seconds for a good level of accuracy

hold_if_exceeded()

Interface to the outside. It pauses the request until the request is allowed

class crypto_history.utilities.general_utilities.**RetryModel**

Provides the ability for the method to be retried

consume_available_retry()

Triggered after an unsuccessful attempt.

Returns whose attempts to retry have been reduced by 1

Return type *RetryModel*

retries

class crypto_history.utilities.general_utilities.**AbstractFactory**
Abstract Factory for all the factories which is responsible for registering classes

classmethod get_builders()

classmethod register_builder(*factory_type*, *identifier*, *class_type*)
Registers the factory to be used later by the user

crypto_history.utilities.general_utilities.**register_factory**()

CHAPTER 2

Indices and tables

- genindex
- search

CHAPTER 3

Examples

- A basic example which interacts with Binance exchange, gets information on assets available and stores a basic *xarray.DataArray* is available at binance-basic
- A second example which displays how the data is type corrected, incomplete data is expunged is available coin-history-post-processing

Index

D

G

H

I

L

N

O

P

R

S

T

U