dxlib

Release 1.0.13

Rafael Zimmer

May 09, 2024

CONTENTS

dxlib is a quantitative trading and analysis library for python. It aims to provide a flexible, easy-to-use, and high-performance trading framework that can be used to develop and backtest trading strategies.

Perform backtests on historical data, and analyze the results, as well as live trading with multiple brokers. Develop and test your own trading strategies, and deploy them as a fully automated trading system.

Check out the *Getting Started* section for further information.

Note: This project is under active development. The API is subject to change

---

CHAPTER

ONE

DOCUMENTATION

The core modules are the main modules that are used to build the application. They are the main building blocks of the application.

1.1 Core

The core modules are the main modules that are used to build the application. They are the main building blocks of the application.

1.1.1 Components

History

History

class dxlib.history.**History**(*df: DataFrame | dict | None = None*, *schema:* Schema *| None = None*)

Bases: `object`

add(*data:* History *| DataFrame | Series | tuple | dict*)

Add historical data to history

Parameters

data – pandas DataFrame or History object

Examples

```
>>> bars = {
        ('2024-01-01', 'AAPL'): Bar(close=155, open=150, high=160, low=140,␣
↪volume=1000000, vwap=150),
        ('2024-01-01', 'MSFT'): Bar(close=255, open=250, high=260, low=240,␣
↪volume=2000000, vwap=250)
    }
>>> history = History(data)
>>> history.add({
        ('2024-01-02', 'AAPL'): Bar(close=160, open=155, high=165, low=145,␣
↪volume=1000000, vwap=155),
        ('2024-01-02', 'MSFT'): Bar(close=260, open=255, high=265, low=245,␣
↪volume=2000000, vwap=255)
```

(continues on next page)

(continued from previous page)

```
    })
>>> history.get(securities='AAPL', fields='close', dates='2024-01-02')
# Output:
# date        security
# 2024-01-02  AAPL      160
# Name: close, dtype: int64
```

apply(*func: Dict[*SchemaLevel*, callable] | callable*, *schema:* Schema | *None = None*, **args*, ***kwargs*)

apply_df(*func: Dict[*SchemaLevel*, callable] | callable*, **args*, ***kwargs*)

apply_on(*other:* History, *func: callable*, *schema:* Schema | *None = None*)

apply_on_df(*other: DataFrame*, *func: callable*)

convert_index(*index: MultiIndex*) → MultiIndex

copy()

classmethod from_df(*df: DataFrame*, *schema:* Schema | *None = None*)

classmethod from_dict(*serialized=False*, ***kwargs*)

classmethod from_list(*history: List[Series]*, *schema:* Schema | *None = None*)

classmethod from_tuple(*history: tuple*, *schema:* Schema | *None = None*)

get(*levels: Dict[*SchemaLevel*, list] | None = None*, *fields: List[str] | None = None*) → *History*

Get historical data for a given security, field and date

Args:

Returns

pandas DataFrame with multi-index and fields as columns

get_df(*levels: Dict[*SchemaLevel*, list] | None = None*, *fields: List[str] | None = None*) → DataFrame

level_unique(*level:* SchemaLevel = *SchemaLevel.SECURITY*)

levels_unique(*levels: List[*SchemaLevel*] | None = None*) → Dict[*SchemaLevel*, list]

property schema

set(*fields: List[str] | None = None*, *values: DataFrame | dict | None = None*)

Set historical data for a given security, field and date

Parameters

- **fields** – list of bar fields
- **values** – pandas DataFrame or dict with multi-index and bar fields as columns

Examples

```
>>> history = History()
>>> history.set(
        fields=['close'],
        values={
            ('2024-01-01', 'AAPL'): 155,
            ('2024-01-01', 'MSFT'): 255
        }
    )
>>> history.get(securities='AAPL', fields='close', dates='2024-01-01')
date        security
2024-01-01  AAPL      155
Name: close, dtype: int64
```

set_df(*levels: Dict[SchemaLevel, list] | None = None*, *fields: List[str] | None = None*, *values: DataFrame | dict | None = None*)

property shape

to_dict(*serializable=False*)

Schema

class dxlib.history.**Schema**(*levels: List[SchemaLevel] | None = None*, *fields: List[str] | None = None*, *security_manager: SecurityManager | None = None*)

Bases: `object`

apply_deserialize(*df: DataFrame*)

copy()

classmethod deserialize(*obj: any*)

extend(*other:* Schema) → *Schema*

fields: List[str]

classmethod from_dict(***kwargs*) → *Schema*

levels: List[*SchemaLevel*]

security_manager: SecurityManager

classmethod serialize(*obj: any*)

to_dict() → dict

SchemaLevel

class dxlib.history.**SchemaLevel**(*value*)

Bases: Enum

An enumeration.

DATE = 'date'

SECURITY = 'security'

classmethod from_dict(***kwargs*)

to_dict()

1.2 Interfaces

The core modules are the main modules that are used to build the application. They are the main building blocks of the application.

CHAPTER

TWO

GETTING STARTED:

2.1 Getting Started

Welcome to the getting started guide for dxlib! This guide will walk you through the initial steps to install, set up, and start using dxlib in your Python projects.

2.1.1 What is dxlib?

dxlib is a powerful Python library designed specifically for quantitative finance, offering a comprehensive suite of tools for financial modeling, analysis, and algorithmic trading. Whether you're a quantitative analyst, algorithmic trader, or financial researcher, dxlib provides the building blocks needed to develop sophisticated trading strategies, optimize portfolios, and analyze financial data with ease.

- Core Components: dxlib's core includes essential components such as history objects for data storage and retrieval, strategy executors for executing trading strategies, inventory management for tracking positions, and security models for managing financial instruments.
- Indicators: Leveraging a wide range of technical indicators, time series analysis tools, and statistical functions, dxlib empowers users to perform in-depth market analysis, generate trading signals, and gain insights into market trends and patterns.
- Portfolio Management: dxlib simplifies portfolio management by offering tools for creating and managing portfolios, loading and processing data, and applying portfolio transformations to optimize performance and manage risk.
- Financial Structures: From options contracts and option models to fixed income rates and financial terms, dxlib provides a flexible framework for modeling a variety of financial instruments and structures.
- Trading Tools: With dxlib's trading tools, users can manage orders, generate trading signals based on predefined strategies, record transactions, and streamline the execution of trading algorithms.
- Interface Integration: dxlib seamlessly integrates with external data sources such as YFinance, Alpaca Markets, and IBKR, allowing users to access real-time market data and historical information. Additionally, internal interfaces facilitate communication between different components within the library, enabling efficient data flow and system integration.

2.1.2 Who is dxlib for?

- Backtesting Strategies: Use dxlib to backtest trading strategies against historical data, evaluate performance metrics, and refine strategies for live trading.
- Algorithmic Trading: Develop custom trading algorithms using technical indicators, market signals, and risk management tools provided by dxlib to automate trading decisions and execute trades.
- Portfolio Optimization: Utilize dxlib's portfolio management capabilities to construct diversified portfolios, rebalance asset allocations, and optimize portfolio performance based on risk-return objectives.
- Real-Time Analysis: Leverage dxlib for real-time data analysis, decision-making, and trade execution in fast-paced financial markets, gaining a competitive edge in algorithmic trading.
- Financial Research: Conduct research and development of new financial instruments, derivative models, and trading strategies using dxlib's flexible framework and extensive toolset.

2.2 Installation

To use **dxlib**, first install it using pip:

```
pip install dxlib
```

2.3 Quickstart

2.3.1 Usage

Defining the financial instruments and formats You can use the dxlib.Schema and dxlib.SecurityManager classes to define the financial instruments and formats.

```
import dxlib as dx

tickers = ['AAPL', 'GOOG', 'MSFT']

security_manager = dx.SecurityManager.from_list(tickers)
print(security_manager)

schema = dx.Schema(
    levels=[dx.SchemaLevel.SECURITY],
    fields=['price'],
    security_manager=security_manager
)
print(schema)
```

```
SecurityManager(3)
Schema(levels=[<SchemaLevel.SECURITY: 'security'>], fields=['price'], security_
↪manager=SecurityManager(3))
```

Creating a history of prices You can use the `dxlib.History` class to store the history of a stock.

```
import dxlib as dx
import datetime

data = {
    (datetime.datetime(2015, 1, 1), 'AAPL'): {'open': 100, 'high': 105, 'low': 95, 'close
↪': 100, 'volume': 1000000},
    (datetime.datetime(2015, 1, 2), 'AAPL'): {'open': 100, 'high': 105, 'low': 95, 'close
↪': 100, 'volume': 1000000},
}

schema = dx.Schema(
    levels=[dx.SchemaLevel.DATE, dx.SchemaLevel.SECURITY],
    fields=['open', 'high', 'low', 'close', 'volume'],
    security_manager=dx.SecurityManager.from_list(['AAPL'])
)

history = dx.History(data, schema)
print(history)
```

```
                          open  high  low  close   volume
date       security
2015-01-01 AAPL (equity)   100   105   95    100  1000000
2015-01-02 AAPL (equity)   100   105   95    100  1000000
```

2.3.2 Executing a strategy

```
import dxlib as dx

strategy = dx.strategies.RsiStrategy()

tickers = ['AAPL', 'GOOG', 'MSFT']
security_manager = dx.SecurityManager.from_list(tickers)

schema = dx.Schema(
    levels=[dx.SchemaLevel.DATE, dx.SchemaLevel.SECURITY],
    fields=['open', 'high', 'low', 'close', 'volume'],
    security_manager=security_manager
)
```

CHAPTER

THREE

TUTORIALS

3.1 Basic Tutorials

3.1.1 Understanding the basic Components

3.2 Advanced Tutorials

3.2.1 External APIs

3.2.2 Defining External and Internal interfaces

INDEX