

# Luna

## Backtesting Framework – Documentation

---

### Overview

**Luna** is a modular, high-performance backtesting and research framework designed for option and futures strategies in the Indian derivatives market (NSE NFO segment).

It provides fast access to precomputed datasets, remote tick-level market data, and a clean object-oriented design for developing and testing trading strategies.

Luna's design emphasizes:

- **Ease of experimentation** – simple APIs for strategy definition and backtesting.
  - **Data modularity** – clear separation between pre-fetched data, market data fetching, and strategy logic.
  - **Extensibility** – new data sources, indicators, or order logic can be added easily.
- 
- 

### About This Document

This document serves as a **formal API reference** for Luna's internal modules and functions.

It is structured to provide developers with clear, implementation-level understanding of:

1. **precomputed.py** – pre-fetched and cached datasets, instrument and expiry lookups.
2. **fetchers.py** – remote SFTP data fetching and time-based price access utilities.
3. **core.py** – main framework components: Marketdata, Position, and Strategy.

Each function and class is described with:

- Purpose / Summary
- Parameters and Return Types
- Exceptions Raised
- Usage Examples

# Module: fetchers.py

## Purpose

The fetchers.py module is responsible for **remote data retrieval** and **time-indexed price access** used in Luna's intraday backtesting framework.

It connects to a remote SFTP data server to fetch tick-level CSV files for futures and options, processes them into pandas DataFrames, and provides utilities for locating prices at specific intraday timestamps.

All data operations are designed to be deterministic and compatible with the framework's event-driven backtesting model.

## Functions

### 1. `fetch_data(date: str, stock_name: str, data_type: str, future_number: int = 1)`

#### Description:

Fetches tick-level CSV data for a given instrument and trading date directly from the remote SFTP server.

Supports both **futures** and **options** instruments.

#### Parameters:

Name	Type	Description
date	str	Trading date in the format DDMMYYYY.
stock_name	str	Name of the symbol or instrument.
data_type	str	Either "future" or "option".
future_number	int, optional	Specifies the contract (1 for current month, 2 for next, 3 for far month). Defaults to 1.

**Returns:**

pandas.DataFrame – The parsed CSV data as a DataFrame.  
None – If the file could not be found or any error occurs.

**Raises:**

- ValueError – If date format or data\_type is invalid.
- FileNotFoundError – If the file does not exist on the remote server.

**Example Usage:**

```
df = fetch_data("05072024", "BANKNIFTY", "future", future_number=1)

print(df.head())
```

## 2. get\_closest\_option(csv\_files: list, target\_price: int, underlying: str)

**Description:**

Given a list of available option filenames, returns the **strike price** closest to a given target price for a specific underlying.

**Parameters:**

Name	Type	Description
csv_files	list	List of CSV filenames retrieved from the data directory.
target_price	int	Desired strike price to match.
underlying	str	Underlying symbol name (e.g., "NIFTY", "BANKNIFTY").

**Returns:**

str – Closest matching strike price.

**Example Usage:**

```
closest = get_closest_option(file_list, target_price=46000, underlying="BANKNIFTY")

print(f"Closest strike: {closest}")
```

### 3. `read_remote_csv(file_path: str)`

#### Description:

Reads a remote CSV file from the SFTP data server and converts it into a pandas DataFrame.

#### Parameters:

Name	Type	Description
file_path	str	Full remote file path.

#### Returns:

pandas.DataFrame – The loaded CSV as a DataFrame.

#### Example Usage:

```
df =  
read_remote_csv("/home/quant/data/quant_data/NFO_TICK/2024/GFDLINFO_TICK_05072024/Options/  
BANKNIFTY46000CE.NFO.csv")
```

### 4. `getOptionPrice(Year: str, Date: str, Symbol: str)`

#### Description:

Fetches option price data for a given symbol and trading date directly from the remote server.

#### Parameters:

Name	Type	Description
Year	str	Four-digit year (e.g., "2024").
Date	str	Trading date in the format DDMMYYYY.
Symbol	str	Option symbol (e.g., "BANKNIFTY46000CE").

#### Returns:

pandas.DataFrame – The corresponding tick data.

### **Example Usage:**

```
option_df = getOptionPrice("2024", "05072024", "BANKNIFTY46000CE")
```

## **5. get\_price\_at\_time(data: pd.DataFrame, target\_time\_str: str)**

### **Description:**

Retrieves a valid tick price for a specific time from a tick-level DataFrame.

The function validates bid/ask data and returns the LTP if it lies between BuyPrice and SellPrice, or otherwise computes a midpoint.

### **Parameters:**

Name	Type	Description
data	pandas.DataFrame	Data containing columns Time, BuyPrice, SellPrice, and LTP.
target_time_str	str	Target time in 'HH:MM:SS' format.

### **Returns:**

float – The computed or fetched price for the given time.

None – If no valid data is found.

### **Logic Summary:**

- If BuyPrice or SellPrice is zero, use the last row's LTP.
- If LTP lies between BuyPrice and SellPrice, return LTP.
- Otherwise, return the midpoint of BuyPrice and SellPrice.

### **Example Usage:**

```
price = get_price_at_time(option_df, "10:15:00")
```

```
print("Price at 10:15:00 =", price)
```

## 6. get\_ltp\_at\_time(df: pd.DataFrame, time\_str: str)

### Description:

Returns the LTP at a specific time, or the closest earlier available value.

### Parameters:

Name	Type	Description
df	pandas.DataFrame	Tick data containing a Time column and LTP.
time_str	str	Time string in 'HH:MM:SS' format.

### Returns:

float – Last Traded Price (LTP) at or before the given time.

None – If no data exists.

### Example Usage:

```
ltp = get_ltp_at_time(option_df, "10:30:00")
```

## 7. fetch\_data\_and\_get\_price(date, option\_strike, time, default=None)

### Description:

Convenience wrapper that fetches option data remotely and immediately retrieves its price at a given time.

### Parameters:

Name	Type	Description
date	str	Trading date in DDMMYYYY format.
option_strike	str	Option symbol including strike (e.g., "BANKNIFTY46000CE").
time	str	Target time in 'HH:MM:SS' format.
default	float, optional	Default return value if data retrieval fails.

**Returns:**

float – Price at the specified time.

default – If data fetch or lookup fails.

**Example Usage:**

```
price = fetch_data_and_get_price("05072024", "BANKNIFTY46000CE", "10:00:00", default=0.0)
```

---

## Module: precomputed.py

### Purpose

The precomputed.py module provides high-speed, in-memory access to preprocessed market reference datasets required by Luna's intraday backtesting framework.

It loads precomputed pickled (.pkl) files containing option chains, strike mappings, futures instruments, lot sizes, expiry schedules, and trading calendars.

These data files are loaded once at initialization, allowing constant-time dictionary lookups during strategy execution without repeated disk I/O.

All lookups are performed through nested Python dictionaries, providing **average O(1)** access complexity.

### Functions

#### 1. `get_futures(date: str, symbol: str = None)`

##### Description:

Returns futures instrument data for a given trading date.

If a symbol is provided, returns only data for that symbol.

##### Parameters:

Name	Type	Description
date	str	Trading date in DDMMYYYY format.
symbol	str, optional	Specific underlying symbol. Defaults to None.

##### Returns:

- dict – Futures data for the specified date.
- None – If no matching entry is found.

## Example Usage:

```
data = get_futures("04012021")  
  
symbol_data = get_futures("04012021", "RELIANCE")
```

## 2. **get\_strikes(date: str, symbol: str, option\_type: str)**

### Description:

Retrieves available strike-level information for the specified date, underlying, and option type ("CE" or "PE").

### Parameters:

Name	Type	Description
date	str	Trading date in DDMMYYYY format.
symbol	str	Underlying symbol (e.g., "RELIANCE").
option_type	str	Either "CE" or "PE".

### Returns:

- dict – Strike mapping dictionary for the given parameters.
- None – If no data is available.

## Example Usage:

```
strikes = get_strikes("04012021", "RELIANCE", "CE")
```

## 3. **get\_options(date: str, symbol: str, option\_type: str)**

### Description:

Fetches the option mapping for the given date, symbol, and option type ("CE" or "PE"). Useful for retrieving the full instrument name or token mapping.

### Parameters:

Name	Type	Description
date	str	Trading date in DDMMYYYY format.
symbol	str	Underlying symbol.
option_type	str	"CE" or "PE".

**Returns:**

- dict – Mapping of options for the given parameters.
- None – If no entry exists.

**Example Usage:**

```
options = get_options("04012021", "RELIANCE", "PE")
```

**4. get\_trading\_dates()****Description:**

Returns the complete list of available trading dates loaded from precomputed data.

**Parameters:**

*None.*

**Returns:**

- list[str] – Chronologically ordered list of trading date strings.

**Example Usage:**

```
dates = get_trading_dates()
```

**5. get\_lot\_size(date: str, symbol: str)****Description:**

Fetches the lot size applicable to a given symbol on a specific trading date.

**Parameters:**

Name	Type	Description
date	str	Trading date in DDMMYYYY format.
symbol	str	Underlying symbol.

## Returns:

- int – Lot size for the symbol.
- None – If no data exists.

## Example Usage:

```
lot = get_lot_size("01012021", "RELIANCE")
```

## 6. get\_expiry(date: str, symbol: str)

### Description:

Retrieves expiry date information for a specific symbol on a given trading day.

### Parameters:

Name	Type	Description
date	str	Trading date in DDMMYYYY format.
symbol	str	Underlying symbol.

## Returns:

- str – Expiry date string.
- None – If no matching record is found.

## Example Usage:

```
expiry = get_expiry("01012021", "RELIANCE")
```

## Design Notes

- All .pkl datasets are loaded once into memory at module import time.
- Data is stored as nested Python dictionaries, enabling **average O(1)** lookup performance.
- The module assumes all pickle files are version-aligned and schema-consistent.
- It serves as the static reference layer for all instrument metadata, expiry details, and lot configurations used across the Luna framework.

# Module: core.py

## Purpose

core.py provides the central classes and logic for Luna's intraday backtesting and strategy simulation. It handles market data caching, single-leg positions, multi-leg strategy tracking, tick management, live and realized PnL computation, and graceful strategy termination.

The module supports intraday logics and is fully compatible with event-driven backtesting.

## Classes and Methods

### Class: MarketData

#### Purpose:

Handles fetching, caching, and preprocessing of tick-level data for a given date and contract. It allows efficient repeated access to intraday data.

Method	Parameters	Returns	Description
<code>__init__(data_loader=None)</code>	<code>data_loader: Optional[Callable]</code>	<code>None</code>	Initializes a <code>MarketData</code> object. Uses <code>fetch_data</code> by default if no loader is provided.
<code>get_data(date, contract, data_type="option")</code>	<code>date: str, contract: str, data_type: str</code>	<code>pandas.DataFrame</code>	Fetches data for a contract on a specific date. Caches result for repeated access. Returns empty DataFrame if no data is found.

#### Example Usage:

```
from core import MarketData  
  
md = MarketData()  
  
df = md.get_data("05072024", "BANKNIFTY46000CE", "option")
```

## Class: Position

### Purpose:

Represents a single trading leg, manages its tick data, and calculates live and finalized PnL.

Method	Parameters	Returns	Description
<code>__init__(contract, qty)</code>	<code>contract: str, qty: int</code>	<code>None</code>	Initializes a single-leg position with contract name and quantity.
<code>load_data(market_data, date, data_type="option")</code>	<code>market_data: MarketData, date: str, data_type: str</code>	<code>None</code>	Loads tick data for the position using a MarketData instance.
<code>next_tick_time()</code>	<code>None</code>	<code>datetime.time or None</code>	Returns the time of the next tick, or <code>None</code> if no ticks remain.
<code>update_to_next_tick()</code>	<code>None</code>	<code>bool</code>	Advances to the next tick and updates last LTP. Returns <code>False</code> if at the end of data.
<code>live_pnl()</code>	<code>None</code>	<code>float</code>	Computes current unrealized PnL: $(\text{last_ltp} - \text{entry\_price}) * \text{qty}$ .
<code>finalize()</code>	<code>None</code>	<code>None</code>	Finalizes position by locking exit price.

## Class: Strategy

### Purpose:

Manages multiple positions, computes total and realized PnL, handles tick-level updates, and enforces intraday targets or stop-losses.

Method	Parameters	Returns	Description
<code>__init__(name, market_data, target=None, sl=None)</code>	<code>name: str, market_data: MarketData, target: float, sl: float</code>	<code>None</code>	Initializes strategy with name, optional target PnL, stop-loss, and market data source.
<code>open_leg(contract, qty, date, data_type="option")</code>	<code>contract: str, qty: int, date: str, data_type: str</code>	<code>None</code>	Registers a position to the strategy but does not start PnL tracking yet.
<code>start_tracking_leg(contract)</code>	<code>contract: str</code>	<code>None</code>	Begins PnL tracking for a subscribed leg. Aligns leg to the current strategy tick.
<code>stop_tracking_leg(contract)</code>	<code>contract: str</code>	<code>None</code>	Stops tracking a leg and books realized PnL.
<code>close_all()</code>	<code>None</code>	<code>None</code>	Closes all active legs and finalizes realized PnL. Marks strategy inactive.
<code>get_latest_ltp(contract=None)</code>	<code>contract: Optional[str]</code>	<code>float or dict</code>	Returns latest LTP for a specific leg or all subscribed legs.
<code>live_total_pnl()</code>	<code>None</code>	<code>float</code>	Computes total unrealized PnL for all active legs.
<code>total_pnl()</code>	<code>None</code>	<code>float</code>	Returns total PnL: <code>realized_pnl + live_total_pnl()</code> .
<code>next_tick_time()</code>	<code>None</code>	<code>datetime.time or None</code>	Returns the next tick time across all legs, or <code>None</code> if all ticks finished.

update_tick()	None	bool	Advances all legs by one tick, updates total PnL, checks target/SL, and returns strategy active status.
is_active()	None	bool	Returns True if the strategy is currently active.
terminate(reason="Manual termination")	reason: str	None	Gracefully stops strategy without closing positions. Active positions remain open.
terminate_and_square_off(reason="Manual termination and square-off")	reason: str	None	Stops the strategy and closes all positions immediately. Updates realized PnL.

**Note:** For example strategies and usage patterns, please refer to the sample.py files included in the repository. They demonstrate multi-leg strategies, live PnL tracking, tick updates, and intraday backtesting using the core.py framework.

**Important:** Luna is under active development. Always pull the latest code from the Git repository before running any strategies to ensure you are using the most recent builds, features, and bug fixes.