YFrake

Release 0.3.4

Mattias Aabmets

May 19, 2022

YFRAKE

CHAPTER

ONE

DESCRIPTION

YFrake is a fast and flexible stock market, forex and cryptocurrencies data scraper and server[1]. It enables developers to **build powerful apps** without having to worry about the details of session management or maximizing throughput[2].

YFrake has caching built in to speed up requests even more and to reduce load on the source servers. The cache and other YFrake options are fully customizable through the configuration file.

YFrake can be used as a client to directly return market data to the current program or as a **programmatically controllable server** to provide market data to other applications.

In addition, all network requests by the client in **both** sync and async modes are **non-blocking**, which means that your program can continue executing your code while network requests are in progress.

The best part about YFrake is its **built-in swagger API documentation** which you can use to perform test queries and examine the returned responses straight in your web browser.

YFrake is built upon the widely used **aiohttp** package and its plugins.

[1] Stock market data is sourced from Yahoo Finance.
[2] The limits of YFrake are configurable and depend on the capabilities of your system.

CHAPTER

TWO

GETTING STARTED

Install the package by executing:

```
pip install yfrake
```

Import the public objects with:

```
from yfrake import client, server, config
```

The `client`, `server`, and `config` objects are singletons, which have been instantiated internally beforehand to provide the user with lower-case object name identifiers.

NB! The minimum required Python version for YFrake is **Python 3.10**. From YFrake version **2.0.0** forward, trying to import YFrake in lower Python versions will raise a **RuntimeError**.

CHAPTER

THREE

ENDPOINTS

Here is the full list of all available endpoints.
You can perform test queries to these endpoints from the built-in Swagger documentation.

Count	Endpoints	Symbols
1	historical_prices	stocks, forex, crypto
2	quotes_overview	stocks, forex, crypto
3	quote_type	stocks, forex, crypto
4	news	stocks, forex, crypto
5	recommendations	stocks, forex, crypto
6	validate_symbols	stocks, forex, crypto
7	price_overview	stocks, forex, crypto
8	detailed_summary	stocks, forex, crypto
9	options	stocks only
10	insights	stocks only
11	esg_chart	stocks only
12	shares_outstanding	stocks only
13	esg_scores	stocks only
14	purchase_activity	stocks only
15	earnings	stocks only
16	calendar_events	stocks only
17	company_overview	stocks only
18	sec_filings	stocks only
19	financials	stocks only
20	recommendation_trend	stocks only
21	ratings_history	stocks only
22	earnings_history	stocks only
23	earnings_trend	stocks only
24	key_statistics	stocks only
25	income_statements	stocks only
26	cashflow_statements	stocks only
27	balance_statements	stocks only
28	institution_ownership	stocks only
29	fund_ownership	stocks only
30	major_holders	stocks only
31	insider_transactions	stocks only
32	insider_holders	stocks only
33	market_summary	none

continues on next page

Table 1 – continued from previous page

Count	Endpoints	Symbols
34	trending_symbols	none
35	currencies	none

CHAPTER

FOUR

CACHING

YFrake includes a fast in-memory **TLRU** cache for the client and the server objects to speed up consecutive identical requests to the same endpoints over a period of time. The default time-to-live *(TTL)* values have been found to be optimal through testing.

Caching can be disabled either individually for each endpoint by setting their TTL value to zero or in groups by enabling the group override setting and leaving the relevant group TTL value to zero.

This cache does not persist over program restarts. If the user desires to use something more permanent, it is suggested to use a library like diskcache.

CHAPTER

FIVE

OVERVIEW

Contents

5.1 Client Object

5.1.1 Methods

The `client` singleton is the main object which is used to request data from the Yahoo Finance API servers. It has three methods: the `get` method, which is used to make a single request, the `batch_get` helper method, which is used to schedule multiple requests with one call, and the `get_all` helper method, which requests data about a single symbol from all symbol-specific endpoints at once.

5.1.2 Decorators

The `client` object has a single decorator named `session`, which opens a session to the Yahoo Finance API servers and inspects the concurrency mode of your program to adjust its behaviour accordingly. This enables YFrake to work in async and sync (threaded) modes out-of-the-box.

A function or a coroutine must be decorated with this decorator before any calls to the `client` methods are made. Calls to the `client` methods do not have to take place inside the same function or coroutine which was decorated.

For simplicity's sake, it is recommended to decorate the `main` function or coroutine of your program, so the session is opened on program start and closed when the program ends, but in essence any function or a coroutine can be used, as long as the before-mentioned considerations are taken into account.

The best practice is to have your program activate the decorator only once, because repeatedly opening and closing the session will kill your performance.

Note: On Windows machines, the decorator automatically sets the asyncio event loop policy to *WindowsSelectorEventLoopPolicy*, because the default *WindowsProactorEventLoopPolicy* does not work correctly. This automatic selection

works only when the decorated coroutine of your program is the `main` coroutine, which gets passed into the `asyncio.run()` function.

5.2 ClientResponse Object

Instances of this object are returned by the `client.get` method. It handles the request and contains the response from the Yahoo Finance API servers in three properties: `endpoint`, `error` and `data`.

The `endpoint` is a string, while the `error` and `data` can be either dictionaries or None. If the request returned with an error, the `error` property is a dictionary and the `data` property is None. If the request returned with data, then the `data` property is a dictionary and the `error` property is None. This allows the developer to easily check for response status by writing `if resp.error is None:`.

It has methods to (a)wait for the response and to check its completion status and also two properties, `event` and `future`, to access the low-level internals of the `ClientResponse` object.

5.3 Async- and ThreadResults Object

Instances of these objects, which are returned by the `client.batch_get` and the `client.get_all` methods, are a list-like containers of `ClientResponse` objects with additional functionality attached on top.

There are two kinds of results objects: `AsyncResults` and `ThreadResults`. Which one is returned depends on the concurrency mode of the program. `AsyncResults` is returned when the program is running in async mode and the `ThreadResults` is returned when the program is running in sync (threaded) mode.

The results objects can be used with the `len()` and `list()` functions and the subscript operator `[]`. They have methods to (a)wait for the requests and to check their completion statuses and also generators to iterate over the `ClientResponse` objects in a `for` or an `async for` loop. These generators guarantee that the objects which they yield into the `for` loop have finished their request to the servers.

You can also loop over a results object with `for resp in results`, but the returned objects are not guaranteed to be in a finished state, unless you have specifically (a)waited the results object beforehand.

CHAPTER

SIX

REFERENCE

6.1 Client Reference

Contents

6.1.1 Public Decorators

@session

Manages the network connection to the Yahoo Finance API servers.
Needs to be active only when the client methods are being called.
Used internally by the YFrake server process.

Raises **RuntimeError** – if a configuration is already active.

6.1.2 Public Methods

classmethod get(*endpoint*, ***kwargs*)

Schedules a request to be made to the Yahoo Finance servers.
Returns immediately with the pending response object.

Parameters

- **endpoint** (*str*) – The name of the endpoint from which to request data.
- **kwargs** (*unpacked dict*) – Variable keyword arguments, which depend on the endpoint requirements. Values can be either *str*, *int* or *bool*.

Raises

- **RuntimeError** – if the session decorator is not in use.

- **NameError** – if an invalid endpoint name has been provided.
- **KeyError** – if an invalid query parameter has been provided.
- **TypeError** – if the datatype of a query parameter is invalid.

Returns Response object

Return type ClientResponse

classmethod batch_get(*queries*)

Helper method which schedules multiple queries at once.
Returns immediately with the pending results object.

Parameters **queries** (*list*) – Collection of query dicts.

Raises

- **RuntimeError** – if the session decorator is not in use.
- **NameError** – if an invalid endpoint name has been provided.
- **KeyError** – if an invalid query parameter has been provided.
- **TypeError** – if the datatype of a query parameter is invalid.

Returns List-like collection object

Return type AsyncResults or ThreadResults

classmethod get_all(*symbol*)

Helper method which schedules a request to all symbol-specific
endpoints for a given symbol at once. A single call results in
32 simultaneous requests to the Yahoo Finance API servers.
Size of the returned data can vary from 1 to 1.5 megabytes.
Returns immediately with the pending results object.

Parameters **symbol** (*str*) – Security identifier.

Raises

- **RuntimeError** – if the session decorator is not in use.
- **NameError** – if an invalid endpoint name has been provided.
- **KeyError** – if an invalid query parameter has been provided.
- **TypeError** – if the datatype of a query parameter is invalid.

Returns List-like collection object

Return type AsyncResults or ThreadResults

6.2 ClientResponse Reference

Contents

6.2.1 Public Methods

pending()

Checks if the request has completed by calling the `is_set()` method on the internal event object. Returns `True` if the request is still in progress.

Returns Request completion status

Return type bool

wait()

In async mode, returns the `wait()` coroutine of the internal asyncio.Event object.
In sync (threaded) mode, calls the `wait()` method on the internal threading.Event object.

Returns Awaitable coroutine or None

Return type Coroutine or None

6.2.2 API Response Properties

property endpoint

Provides access to the endpoint name of the response.

Raises **RuntimeError** – on property modification or deletion.

Returns Name of the endpoint.

Return type str

property error

Provides access to the error dictionary of the response.

Raises **RuntimeError** – on property modification or deletion.

Returns Error dict, if there was an error, or None.

Return type dict or None

`property data`

Provides access to the data dictionary of the response.

Raises `RuntimeError` – on property modification or deletion.

Returns Data dict, if there weren't any errors, or None.

Return type dict or None

6.2.3 Internal Request Properties

`property event`

Provides access to the internal request completion event object.
Return type depends on the concurrency mode of the program.
In most cases, manual usage of this object is unnecessary.

Disclaimer: Incorrect usage of this object can break things.

Raises `RuntimeError` – on property modification or deletion.

Returns Reference to the internal event object.

Return type asyncio.Event in async mode

Return type threading.Event in sync (threaded) mode

`property future`

Provides access to the internal future-like request object.
Return type depends on the concurrency mode of the program.
In most cases, manual usage of this object is unnecessary.

Disclaimer: Incorrect usage of this object can break things.

Raises `RuntimeError` – on property modification or deletion.

Returns Reference to the internal future-like object.

Return type asyncio.Task in async mode

Return type concurrent.futures.Future in sync (threaded) mode

6.3 AsyncResults Reference

Contents

6.3.1 Public Methods

`pending()`

Function which checks the completion statuses of all its requests by calling the `pending()` method on each `ClientResponse`. Returns `True` if at least one request is still in progress.

Returns Request completion status

Return type bool

6.3.2 Public Coroutines

`async wait()`

Awaits until all its requests have completed.

Returns None

`async gather()`

Asynchronous generator which can be used in the `async for` loop.
Awaits and starts yielding results when all requests have completed.

Returns Request response objects

Return type ClientResponse

`async as_completed()`

Asynchronous generator which can be used in the `async for` loop.
Awaits and starts yielding results immediately as they become available.

Returns Request response objects

Return type ClientResponse

6.4 ThreadResults Reference

Contents

6.4.1 Public Methods

`pending()`

Function which checks the completion statuses of all its requests by calling the `pending()` method on each `ClientResponse`. Returns `True` if at least one request is still in progress.

Returns Request completion status

Return type bool

`wait()`

Waits until all its requests have completed.

Returns None

`gather()`

Synchronous generator which can be used in the `for` loop.
Waits for and starts yielding results when all requests have completed.

Returns Request response objects

Return type ClientResponse

`as_completed()`

Synchronous generator which can be used in the `for` loop.
Waits for and starts yielding results immediately as they become available.

Returns Request response objects

Return type ClientResponse

CHAPTER

SEVEN

EXAMPLES

7.1 Async Mode Examples

Contents

7.1.1 Client.get() Examples

The following example loops at line 4 `while` the response has not yet arrived:

```
1 @client.session
2 async def main():
3     resp = client.get('quote_type', symbol='msft')
4     while resp.pending():
5         # do some other stuff
```

The following example blocks at line 4 until the response has arrived:

```
1 @client.session
2 async def main():
3     resp = client.get('quote_type', symbol='msft')
4     await resp.wait()
5     # do some other stuff
```

7.1.2 Client.batch_get() Examples

The following example waits until all of the responses have arrived before running the `async for` loop:

```
@client.session
async def main():
    queries = [
        dict(endpoint='quote_type', symbol='msft'),
        dict(endpoint='price_overview', symbol='aapl'),
        dict(endpoint='key_statistics', symbol='tsla')
    ]
    results = client.batch_get(queries)
    async for resp in results.gather():
        # do some stuff with the resp
```

The following example starts yielding the responses into the `async for` loop as soon as they become available:

```
@client.session
async def main():
    queries = [
        dict(endpoint='quote_type', symbol='msft'),
        dict(endpoint='price_overview', symbol='aapl'),
        dict(endpoint='key_statistics', symbol='tsla')
    ]
    results = client.batch_get(queries)
    async for resp in results.as_completed():
        # do some stuff with the resp
```

7.1.3 Client.get_all() Examples

The following example loops while all the available data about a symbol is being retrieved:

```
@client.session
async def main():
    results = client.get_all(symbol='msft')
    while results.pending():
        # do some other stuff
```

The following example blocks while all the available data about a symbol is being retrieved:

```
@client.session
async def main():
    results = client.get_all(symbol='aapl')
    await results.wait()
    # do some other stuff
```

WARNING: A single call to `get_all()` creates 32 simultaneous network requests and can return up to 1.5 megabytes of data, so uncontrolled usage of this method *may* deplete the memory of your system and *may* get your IP blacklisted by Yahoo.

7.2 Sync (Threaded) Mode Examples

Contents

7.2.1 Client.get() Examples

The following example loops at line 4 `while` the response has not yet arrived:

```
1 @client.session
2 def main():
3     resp = client.get('quote_type', symbol='msft')
4     while resp.pending():
5         # do some other stuff
```

The following example blocks at line 4 until the response has arrived:

```
1 @client.session
2 def main():
3     resp = client.get('quote_type', symbol='msft')
4     resp.wait()
5     # do some other stuff
```

7.2.2 Client.batch_get() Examples

The following example waits until all of the responses have arrived before running the `for` loop:

```
1  @client.session
2  def main():
3      queries = [
4          dict(endpoint='quote_type', symbol='msft'),
5          dict(endpoint='price_overview', symbol='aapl'),
6          dict(endpoint='key_statistics', symbol='tsla')
7      ]
8      results = client.batch_get(queries)
9      for resp in results.gather():
10         # do some stuff with the resp
```

The following example starts yielding the responses into the `for` loop as soon as they become available:

```
1 @client.session
2 def main():
3     queries = [
4         dict(endpoint='quote_type', symbol='msft'),
```

(continues on next page)

(continued from previous page)

```
        dict(endpoint='price_overview', symbol='aapl'),
        dict(endpoint='key_statistics', symbol='tsla')
    ]
    results = client.batch_get(queries)
    for resp in results.as_completed():
        # do some stuff with the resp
```

7.2.3 Client.get_all() Examples

The following example loops while all the available data about a symbol is being retrieved:

```
@client.session
def main():
    results = client.get_all(symbol='msft')
    while results.pending():
        # do some other stuff
```

The following example blocks while all the available data about a symbol is being retrieved:

```
@client.session
def main():
    results = client.get_all(symbol='aapl')
    results.wait()
    # do some other stuff
```

WARNING: A single call to `get_all()` creates 32 simultaneous network requests and can return up to 1.5 megabytes of data, so uncontrolled usage of this method *may* deplete the memory of your system and *may* get your IP blacklisted by Yahoo.

7.3 Various Examples

The following example prints out the names of all the endpoints queried:

```
from yfrake import client
import asyncio

@client.session
async def main():
    results = client.get_all(symbol='msft')
    async for resp in results.gather():
        print(f'Endpoint: {resp.endpoint}')

if __name__ == '__main__':
    asyncio.run(main())
```

The following example prints out either the `error` or the `data` property of the `ClientResponse` objects:

```
from yfrake import client
import asyncio

```

(continues on next page)

(continued from previous page)

```
@client.session
async def main():
    queries = [
        dict(endpoint='quote_type', symbol='msft'),
        dict(endpoint='price_overview', symbol='gme_to_the_moon'),
        dict(endpoint='key_statistics', symbol='tsla')
    ]
    results = client.batch_get(queries)
    await results.wait()
    for resp in results:
        if resp.error:
            print(f'Error: {resp.error}')
        else:
            print(f'Data: {resp.data}')

if __name__ == '__main__':
    asyncio.run(main())
```

The following example creates a batch request of 3 endpoints for 3 symbols:

```
from yfrake import client

@client.session
def main():
    all_queries = list()
    for symbol in ['msft', 'aapl', 'tsla']:
        queries = [
            dict(endpoint='quote_type', symbol=symbol),
            dict(endpoint='price_overview', symbol=symbol),
            dict(endpoint='key_statistics', symbol=symbol)
        ]
        all_queries.extend(queries)

    results = client.batch_get(all_queries)
    results.wait()

    count = len(results)
    print(f'ClientResponse objects: {count}')  # 9

if __name__ == '__main__':
    main()
```

The following example demonstrates the usage of the `get` method inside a non-decorated function (or coroutine):

```
from yfrake import client

def make_the_request(symbol):
    resp = client.get('quote_type', symbol=symbol)
    resp.wait()
    return resp

@client.session
```

(continues on next page)

(continued from previous page)

```
def main():
    resp = make_the_request('msft')
    print(f'Data: {resp.data}')

if __name__ == '__main__':
    main()
```

CHAPTER

EIGHT

OVERVIEW

The standardized interface of the YFrake server simplifies the process of acquiring stock market data for other applications, which can use their own networking libraries to make web requests to the YFrake server.

There are two ways how you can run the server: you can either control it from within your Python program through the `server` singleton or you can directly call the YFrake module in the terminal with `python -m yfrake args`. When running the server from the terminal without any args, then nothing will happen. The optional args are `--run-server` and `--config-file /path`, which can be used independently from each other.

The arg `--config-file` accepts as its only parameter either a full path to the config file or the special keyword `here`, which will have the server look for the config file in the **Current Working Directory**. When using the keyword `here`, if the file does not exist, it will be created with the default settings. If the parameter is a full path to a config file, then the file must exist, otherwise an exception will be thrown. In all cases, the config file must be named `yfrake_settings.ini`.

When `--run-server` is used without the `--config-file` arg, then the server is run with the default settings. Using `--config-file here` without the `--run-server` arg is useful for getting a copy of the config file with the default settings to the **CWD**.

You can access the built-in Swagger documentation by running the server and navigating to the servers root address in your web browser (default: `http://localhost:8888`).

You can perform queries to the endpoints either directly through the Swagger Docs UI, or by navigating to the appropriate URL-s in the address bar of your web browser.

When accessing endpoints through their URL-s, each endpoint has a path name like `/market_summary`. To request data from that endpoint, in your address bar you would write: `http://localhost:8888/market_summary`.

If an endpoint like `/company_overview` requires query parameters, then you would write in your address bar: `http://localhost:8888/company_overview?symbol=msft`.

CHAPTER

NINE

REFERENCE

classmethod server.**start**()

Starts the YFrake server. Only one server can be active *per process* at any time.

Raises **RuntimeError** – if the server is already running.

Returns None

classmethod server.**stop**()

Stops the YFrake server.

Raises **RuntimeError** – if the server is already stopped.

Returns None

classmethod server.**is_running**()

Checks if the server is running.

Returns Server status

Return type bool

CHAPTER

TEN

EXAMPLES

Running the server programmatically:

```
from yfrake import server

if not server.is_running()
    server.start()

# do other stuff

if server.is_running()
    server.stop()
```

Creating the 'yfrake_settings.ini' file to the *CWD* if it doesn't exist, without running the server:
`$ python -m yfrake --config-file here`

Running the server from the terminal:

1) With the default configuration:
`$ python -m yfrake --run-server`

2) With 'yfrake_settings.ini' in the *CWD*:
`$ python -m yfrake --run-server --config-file here`

3) With the config file in a custom directory:
`$ python -m yfrake --run-server --config-file "/path/to/'yfrake_settings.ini"`

CHAPTER

ELEVEN

OVERVIEW

Configuration settings for YFrake are stored in a file named `yfrake_settings.ini`. The `config` singleton reads the settings from that file and configures the `client` and the `server` objects. It is not necessary to use the `config` object, if you want to run YFrake with the default settings.

The config has two properties named `file` and `settings` and one method named `is_locked`, which is used to check if the configuration is **locked**, i.e., the `client.session` decorator is in use (active).

All the properties of the `config` object can be **read** at any time, but the `file` property can be modified **only** when the `client.session` decorator is **not** in use (active). The `file` property can accept either a pathlib.Path or a string object, which contains a full path to a config file. Modifying the `file` property after the `server` has started has undefined behaviour and is therefore **not recommended**.

Accessing the `settings` property will return a dictionary of the currently loaded configuration. Modifying this dictionary does not modify the currently loaded configuration.

The `config` object also has an attribute named `HERE`, which points to an abstract config file in the **Current Working Directory**. Assigning the `HERE` attribute to the `file` property will create the config file in the **CWD** with the default settings, if it doesn't exist.

CHAPTER

TWELVE

REFERENCE

Contents

12.1 Public Methods

classmethod is_locked()

Helper method which is used to check if the configuration is being used by the `client.session` decorator. Any attempt to change the configuration while the session is open will cause a `RuntimeError` to be thrown.

Returns Value of the config lock status.

Return type bool

12.2 Public Properties

class property file

The full path to the configuration file which should be used by the client and the server objects. Can be assigned either a `pathlib.Path` or a `str` object.

Raises `TypeError` – on attempt to delete the property.

Returns Full path to the config file to be used.

Return type pathlib.Path

`class property settings`

Deep copied dictionary of the currently loaded configuration.
This property is *READ ONLY*.

Raises

- **TypeError** – on attempt to modify the property.
- **TypeError** – on attempt to delete the property.

Return type dict

CHAPTER

THIRTEEN

EXAMPLES

Contents

13.1 Correct Usage Examples

No config object usage is required to use the default settings:

```
from yfrake import client

@client.session
def main():
    # do stuff

main()
```

Assigning a custom config file in the **Current Working Directory**.
If the file doesn't exist, it will be created with the default settings.

```
from yfrake import client, config

config.file = config.HERE

@client.session
def main():
    # do stuff

main()
```

Assigning a custom config file in the specified path:

```
from yfrake import client, config

config.file = "C:/Users/username/Projects/Project Name/yfrake_settings.ini"

@client.session
def main():
    # do stuff

main()
```

Reading the currently loaded configuration settings:

```
from yfrake import client, config

settings = config.settings  # correct

@client.session
def main():
    settings = config.settings  # also correct

main()
```

Assigning a custom config file before the server is started:

```
from yfrake import server, config

config.file = Path("C:/Users/username/Projects/Project Name/yfrake_settings.ini")
server.start()

# defined behaviour

server.stop()
```

13.2 Incorrect Usage Examples

Trying to assign a custom config file in the **Current Working Directory**.

```
from yfrake import client, config

@client.session
def main():
    config.file = config.HERE

    # will raise an exception

main()
```

Trying to assign a custom custom config file in the specified path:

```
from yfrake import client, config

```

(continues on next page)

(continued from previous page)

```
@client.session
def main():
    config.file = "C:/Users/username/Projects/Project Name/yfrake_settings.ini"

    # will raise an exception

main()
```

Assigning a custom config file after the server has started:

```
from yfrake import server, config

server.start()
config.file = Path("C:/Users/username/Projects/Project Name/yfrake_settings.ini")

# undefined behaviour

server.stop()
```

CHAPTER

FOURTEEN

CONFIG FILE

Contents

14.1 Description

TTL time values are **integer seconds**. All settings in the config file affect the client and the server behaviour both, except those in the **SERVER** section, which affect only the behaviour of the server.

14.2 Sections

14.2.1 CLIENT

limit: *integer - default:* **64**
The amount of active concurrent requests to Yahoo servers.

timeout: *integer - default:* **2**
The amount of time in seconds to wait for each response.

14.2.2 SERVER

host: *string - default:* **localhost**
The host name on which the YFrake server listens on.

port: *integer - default:* **8888**
The port number on which the YFrake server listens on.

backlog: *integer - default:* **128**
The number of unaccepted connections that the system will allow before refusing new connections.

14.2.3 CACHE_SIZE

max_entries: *integer - default:* **1024**
The max number of entries in the cache before the cache begins to evict LRU entries.

max_entry_size: *integer - default:* **1**
The max memory usage for a single cache entry in megabytes.
A request is not cached if the response is larger than this value.

max_memory: *integer - default:* **64**
The max memory usage of entries in megabytes before the cache begins to evict LRU entries.

14.2.4 CACHE_TTL_GROUPS

override: *string - default:* **false**
If **false**, the individual TTL value of each endpoint is used.
If **true**, the group TTL value of the endpoints is used.

short_ttl: *integer - default:* **0**
Defines the group TTL value for the *CACHE_TTL_SHORT* section.

long_ttl: *integer - default:* **0**
Defines the group TTL value for the *CACHE_TTL_LONG* section.

14.2.5 CACHE_TTL_SHORT

historical_prices: *integer - default:* **60**
detailed_summary: *integer - default:* **60**
financials: *integer - default:* **60**
insights: *integer - default:* **60**
key_statistics: *integer - default:* **60**
market_summary: *integer - default:* **60**
news: *integer - default:* **60**
options: *integer - default:* **60**
price_overview: *integer - default:* **60**

quotes_overview: *integer - default:* **60**
trending_symbols: *integer - default:* **60**

14.2.6 CACHE_TTL_LONG

balance_statements: *integer - default:* **3600**
calendar_events: *integer - default:* **3600**
cashflow_statements: *integer - default:* **3600**
company_overview: *integer - default:* **3600**
currencies: *integer - default:* **3600**
earnings: *integer - default:* **3600**
earnings_history: *integer - default:* **3600**
earnings_trend: *integer - default:* **3600**
esg_chart: *integer - default:* **3600**
esg_scores: *integer - default:* **3600**
fund_ownership: *integer - default:* **3600**
income_statements: *integer - default:* **3600**
insider_holders: *integer - default:* **3600**
insider_transactions: *integer - default:* **3600**
institution_ownership: *integer - default:* **3600**
major_holders: *integer - default:* **3600**
purchase_activity: *integer - default:* **3600**
quote_type: *integer - default:* **3600**
ratings_history: *integer - default:* **3600**
recommendation_trend: *integer - default:* **3600**
recommendations: *integer - default:* **3600**
sec_filings: *integer - default:* **3600**
shares_outstanding: *integer - default:* **3600**
validate_symbols: *integer - default:* **3600**

INDEX