DEVELOPER MANUAL
FMCLIENT V3.0.0

PYTHON API FOR FLEX-E-MARKETS

Abhijeet ANAND,
Nitin YADAV,
Peter BOSSAERTS

Brain Mind and Markets Lab. Department of Finance.
University of Melbourne.

AUGUST 4, 2020 VERSION:V3.0

CONTENTS

1 INTRODUCTION

Flex-E-Markets[1] is a software-as-a-service (SaaS) platform that allows creation of experimental markets. The platform is equipped to serve both manual trading and algorithmic trading mechanisms. The FM-Client python package provides a library that eases the development of automated trading algorithms.

1.1 OVERVIEW OF FLEX-E-MARKETS

Flex-E-Markets consists of *marketplaces* (e.g., a fruit marketplace where different farmers come to sell their produce) where each marketplace contains one or more *markets* (e.g., a fruit marketplace may contain market for apples and bananas). Users of the Flex-E-Markets platform can be of two types, namely, traders (who buy and sell in a marketplace) and managers (who manage a marketplace). A trader can submit two kinds of orders, a *limit order* to buy or sell in a market, or a *cancel* order to cancel an unfulfilled order. A manager can open, pause or close a marketplace and also update the traders' holdings. A manager is also allowed to trade in their marketplaces.

1.2 FMCLIENT PACKAGE

The FMClient package provides a Python client to the Flex-E-Markets platform. It provides a high level functionality to interact with the platform as a trader as well as a manager using Python programming language. The package provides an abstract class called *Agent* that has a pre-implemented capability to connect with the Flex-E-Markets platform, send and cancel orders, receive current order book, new and updated orders, holdings, and marketplace updates. However, the reasoning behind how to respond to the changes in order book and holdings, and which orders to be submitted (or cancelled) is left to the developer (i.e., it is the responsibility of an Agent's subclass).

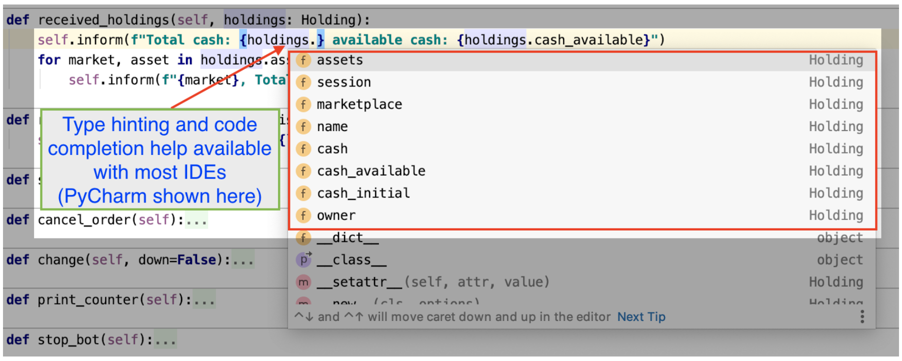
Note: Since v2.0.4, FMClient package is now a unified client for both FM (Flex-E-Markets) and AHM (AdHocMarkets) servers. In order to select an appropriate server (robot must have correct credentials), an environment variable `FM_HOST` must be set. Value for this environment variable could be either `'FM'` or `'AHM'`. The default host is set to `'FM'`.

1.3 FMCLIENT IDE INTEGRATION

The FMClient package has some helpful integration with most of the popular IDEs, such as PyCharm. Most classes within fmclient have type hinting and in-built documentation for methods and attributes.

[1] https://adhocmarkets.com

When writing code in PyCharm, for instance, the IDE shows code completion and help as shown in the figure 1 and 2.



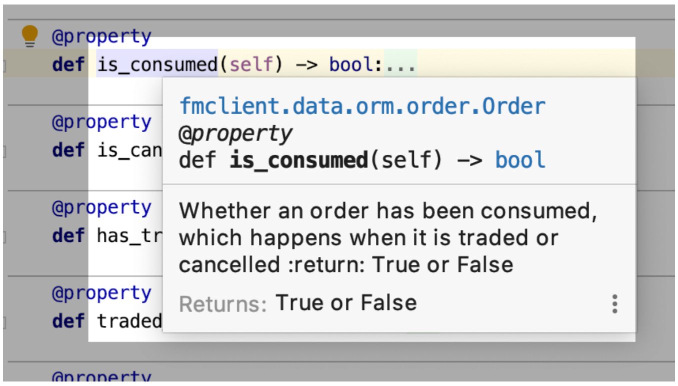
```
def received_holdings(self, holdings: Holding):
    self.inform(f"Total cash: {holdings.} available cash: {holdings.cash_available}")
    for market, asset in holdings.as
        self.inform(f"{market}, Tota

def cancel_order(self):...

def change(self, down=False):...

def print_counter(self):...

def stop_bot(self):...
```

```
f assets               Holding
f session              Holding
f marketplace          Holding
f name                 Holding
f cash                 Holding
f cash_available       Holding
f cash_initial         Holding
f owner                Holding
f __dict__             object
p __class__            object
m __setattr__(self, attr, value)   Holding
```

Type hinting and code completion help available with most IDEs (PyCharm shown here)

^↓ and ^↑ will move caret down and up in the editor Next Tip

Figure 1: FMClient integration with PyCharm



```
@property
def is_consumed(self) -> bool:...

@property
def is_can

@property
def has_tr

@property
def traded
```

```
fmclient.data.orm.order.Order
@property
def is_consumed(self) -> bool
```

Whether an order has been consumed, which happens when it is traded or cancelled :return: True or False

Returns: True or False

Figure 2: FMClient method documentation popup in PyCharm

2 INSTALLATION

The FMClient (v3.0.0 or higher) package requires Python 3.8.0 or higher. It depends on these packages: `aiohttp`, `asyncio_extras`, `autobahn`, `deprecated` and `pytz`. If these are not available, they will be automatically installed at the time of installing this package. Currently, the FMClient package is in active development and therefore not publicly distributed. To obtain the current version of this package please contact Nitin Yadav or Peter Bossaerts.

2.1 PYTHON INSTALLATION

Considering there are a variety of operating systems found on computers these days, it is beyond the scope of this document to provide instructions on how to install Python. However, a good place to find detailed instructions, as well as an installation binary, for a given operating system is the official website of Python programming language.

First, download the specific installation package for Python 3.8 and install it on your computer.

macOS

If the target OS is macOS, then usually an additional step is required to ensure Python is correctly installed. Please read the "Important Information" displayed during installation for information about SSL/TLS certificate validation and the running the `Install Certificates.command`. Once the installation is complete, ensure that you install SSL certificates required by Python for establishing SSL network connections. On most installations, this command could be found in `/Applications/Python 3.8/` directory. Simply navigate to this directory in Finder and double click to execute the script.

Once Python 3.8 has been installed on your computer, create a Python virtual environment[1] specific to your project. A virtual environment isolates all package dependencies for a project, without affecting rest of the system or other projects. Once a virtual environment has been created, activate the environment in the project directory where you are working.

Windows

```
C:\path\to\project> tutorial-env\Scripts\activate.bat
```

[1] On *nix and macOS computers, it is perhaps more convenient (and recommended) to use pyenv, a Python Version Manager.

***nix/macOS**

```
$ source tutorial-env/bin/activate
```

This is an important step if you develop from the command line and also for the next step, when you have to install the fmclient package (see section 2.2).

2.2 FMCLIENT INSTALLATION

To install this package use the following command (we assume pip is available for Python 3.8.0+):

```
$ pip3 install <path-to>/fmclient-<version>-py3-none-any.whl
```

If the package was successfully installed you should see something similar as following:

```
Installing collected packages: fmclient
Successfully installed fmclient-3.0.0rc1
```

Please note that the actual name and version of the FMClient package may be different to the one shown above. The core classes of this package can now be imported from the fmclient package:

```
from fmclient import Agent
from fmclient import Holding
from fmclient import Market, Asset
from fmclient import Marketplace
from fmclient import Order, OrderSide, OrderType
from fmclient import Session, SessionState
from fmclient import User, UserRole
from fmclient import MessageType
```

3 TRADING AGENT

In order to create a trading agent, one has to sub-class the `class Agent` and implement its abstract methods.

3.1 CONSTRUCTOR

The `Agent` class constructor takes 4 mandatory arguments and one optional argument. The mandatory arguments are:

1. **account**: The name of the Flex-E-Markets account;
2. **email**: Email address of the user;
3. **password**: Password for the account and email address;
4. **marketplace_id**: *id* of the Flex-E-Markets marketplace;
5. **name**: Optional name for the trader agent.

An *account* on Flex-E-Markets is identified by an account name, email address and password. A trader agent needs to know the marketplace in which it will trade, and therefore requires the *id* of that marketplace. The marketplaces that are available for a trader to interact in (either manually or algorithmically) can be seen after logging in at the Flex-E-Markets website. The *id* of these marketplaces can be known from the marketplace url or by asking the marketplace manager.

The code below creates a sub-class called `class FMBot` (see lines 5–12) and instantiates it with with Flex-E-Markets account *durable-guard*, with email address *test@at* and password *aLgoTraDer*, and for the marketplace with *id 1* (lines 15 and 16).

```
1   from fmclient import Agent
2   from fmclient import Order, OrderSide, OrderType
3   from fmclient import Holding
4
5   class FMBot(Agent):
6       """
7       This is a very simple "Hello World" agent!
8       """
9       def __init__(self, account, email, password,
        ↪   marketplace_id):
10          name = "SimpleBot"
11          super().__init__(account, email, password,
            ↪   marketplace_id, name=name)
12          self.description = "This is a very simple bot!"
13
14  if __name__ == "__main__":
15      marketplace_id = 1
16      fm_bot = FMBot("durable-guard", "test@at",
        ↪   "aLgoTraDer", marketplace_id)
```

Note: The code above will not run as the `class FMBot` definition is incomplete.

There are multiple ways of creating a sub-class of `Agent` (e.g., hard coding the account details and marketplace *id* inside the sub-class constructor). However, we suggest the structure in the code shown above for two reasons. First, the same class definition can be used across different accounts and marketplaces. Second, the algorithm hosting platform (described later in part 6) relies on it.

3.2 ABSTRACT METHODS

The sub-class needs to provide an implementation to all the abstract methods of the `class Agent`. Importantly, the parent `Agent` class has built in functionality to monitor the assigned marketplace. It calls these methods *automatically* when it retrieves (or receives) relevant information from the Flex-E-Markets server. It is the *responsibility of the `Agent` class to monitor these events*, however, it is the *responsibility of a sub-class to respond to them* (that is, the sub-class should reason on the information available at the time of the event and decide what action to take, if any). These methods allow implementation of the trading logic and enable implementation of both *re-active* and *pro-active strategies*.

1. **Marketplace events**

   - `def received_session_info(self, session: Session)`

     This method is called when the status of the associated marketplace switches between active (i.e., opens for trading) and inactive (i.e., closed for trading). The object `session` is an instance of `class Session` which provides various information on current session state. Objects of this class can be queried for session id (`fm_id`) and current state using properties (`is_open`, `is_closed` and `is_paused`). The code below prints the new status of the marketplace along with the current session id (when the market is active).

     ```
     def received_session_info(self, session: Session):
         session_id = session.fm_id
         if session.is_open:
             self.inform(f"Marketplace is now open. New
              ↪  session is {session_id}")
         else:
             self.inform("Marketplace is now closed.")
     ```

     Tip: Depending upon the task, an agent might want to check if their holdings changed when a marketplace re-opens. The marketplace manager may decide to allocate dividends (or take away some cash!) when the marketplace is closed. An agent gets a notification when holdings are updated, via a different method as described later on page 9.

2. **Market information**

   - `def received_orders(self, orders: List[Order])`

     This method is called each time Flex-E-Markets server *pushes* updates to clients, while they are connected, and provides the order book for a market in two ways. First time a client connects to Flex-E-Markets server, they receive the current orders in the order book, including current trades. Subsequently, clients receive only updates on changes to the order book.

     For e.g. , when an order is cancelled, or partially or fully matched. The list of orders would contain all such orders for all markets in the given marketplace. It is up to an agent to determine their course of action with the knowledge of available orders and trades.

     The parameter `orders` is a list of objects of `class Order`. An order object has following commonly used properties:

     - `price`: The price each unit in cents;
     - `units`: Number of units;
     - `order_side`: `OrderSide.SELL` if the order is to sell and `OrderSide.BUY` if the order is to buy in the respective market;
     - `order_type`: This is normally `OrderType.LIMIT` for limit orders or `OrderType.CANCEL` when an existing order has to be cancelled;
     - `ref`: A string reference for this order that can be used by an agent;
     - `market`: An object of `class Market` to which this order belongs (see page 13 for more info);
     - `date_created`: The date and time when the order was received at the exchange;
     - `date_last_modified`: The date and time when the order was last updated at the exchange (for instance, when an order is split, cancelled or traded);
     - `mine`: `True` if the order is from the same account as the agent, `False` otherwise (readonly attribute, set by FM server);
     - `owner_or_target`: If this is a private order, then this property contains private trader ID to whom the order was sent (`mine is True`) or from whom it was received (`mine is False`);
     - `is_private`: `True` if the order is in a private market, `False` otherwise.

     In addition to the properties above, there are several other properties in an `Order` object that can be used for various purposes:

- `is_balance`: `True` if an order was created as a result of partial trade of a previously split order for remaining units, `False` otherwise;
- `is_cancelled`: `True` if an order has been cancelled (there will be a matching order with `OrderType.CANCEL`), `False` otherwise;
- `is_consumed`: `True` if an order has been consumed, which happens when it is traded or cancelled, `False` otherwise;
- `is_partial`: `True` if an order has been split for partial trades or balance order, `False` otherwise;
- `is_partial_trade`: `True` if an order has been split for partial trades or balance order, `False` otherwise;
- `is_pending`: `True` if an order is active and hasn't been consumed or split, `False`;
- `is_split`: `True` if an order has been split for partial trades, `False` otherwise;
- `has_traded`: `True` if an order has traded with another matching order, `False` otherwise;
- `consumer`: an order that has consumed this order, `None` if not consumed;
- `traded_order`: an order that has traded with this order, `None` if not traded;

Note: Firstly, this method is called each time there is an update in the marketplace, such as, when a new order arrives, an order is cancelled or a trade happens. Secondly, the order book will contain both `LIMIT` and `CANCEL`led orders, therefore, one must be careful when reasoning on the order book. Thirdly, an `Agent` or its sub-classes do not have to maintain a separate order book. The full order book can be accessed anytime by calling a `@classmethod Order.all() -> Dict[int, Order]`. In order to access only the active order book, one can call a `@classmethod Order.current() -> Dict[int, Order]`. However, note that the order book is cleared each time a market opens after a close, but not a pause.

There are other methods in the `Order` class that can be used to query an order or the order book. Such as,

- `def traded_with(self, other: Order) -> bool`: can be used to check whether a given order was 'traded' with another order;
- `@classmethod Order.trades_dict(cls) -> Dict[Order, Order]`: returns a dictionary of traded order pairs, where keys are original orders and values are fulfilling orders;
- `@classmethod Order.trades(cls, order_by: str = 'date_created', rev: bool = True) -> List[Tuple[Order, Order]]`: returns a list of traded order pairs (tuple), where first ele-

ment is original order and second element is fulfilling order. This list can be ordered by certain attributes.

- `def received_holdings(self, holdings: Holding)`

  The received holdings method is called each time there is a change in a user's holding and provides the information regarding the account's current holdings. The `holdings` parameter is an object of `class Holding` which provides information on its various properties. A holding object has following commonly used properties:

  - `assets`: is a dictionary of objects of `class Asset` associated with a `Market` object as key. `Asset` class has following properties that can be accessed:
    * `units`: Number of units of this asset currently held
    * `units_available`: Number of units that are currently available for trading (could be less than settled units if there are units committed in current submitted sell orders)
    * `units_initial`: Number of units that were initially held at the beginning of a session
    * `units_initial_short`: Number of short units held at the beginning of a session
    * `units_granted`: Number of units that were granted by manager
    * `units_granted_short`: Number of short units granted by manager
    * `can_buy`: `True` if this user can buy units of this asset, `False` otherwise
    * `can_sell`: `True` if this user can sell units of this asset, `False` otherwise
    * `market`: an object of `class Market` associated with this asset
  - `cash`: amount of cash (in cents) currently held
  - `cash_available`: amount of cash (in cents) that is available for trading (could be less if there are cash committed in current submitted buy orders)
  - `cash_initial`: initial amount of cash allocated at the start of session
  - `name`: a name of this holding, usually associated with private trader name
  - `owner`: an object of `class User` with which this holding is associates, which is normally the current user
  - `session`: an object of `class Session` for which this holding is applicable

  For example, the code snippet shown below prints the status of current holdings.

```
def received_holdings(self, holdings: Holding):
    self.inform(f"Total cash: {holdings.cash}
     ↪  available cash:
     ↪  {holdings.cash_available}")
    for market, asset in holdings.assets.items():
        self.inform(f"{market}, Total units:
         ↪  {asset.units}, available units:
         ↪  {asset.units_available}")
```

Note: This method is called only on change in holdings.

- `def order_accepted(self, order: Order)`

  The method is called when a submitted order is accepted by the Flex-E-Markets server. The parameter `order` is an object of `class Order` (described on page 7) that was sent to the server (please see page 14 for details on how to send an order).

- `def order_rejected(self, info: dict, order: Order)`

  This method is called when the server rejects a submitted order. The parameter `info` contains the error information from the server and the parameter `order` is the order that was submitted to the server.

  Two common reasons for order rejection are sending an *invalid* order (returned error is "ORDER_INVALID") and having *insufficient funds* for an order (returned error is "ORDER_INSUFFICIENT_ASSETS").

  An order could be invalid when it has an invalid price (e.g., price is not within the allowed range or the tick size is invalid). Some of these validations are performed within the `class Order`, however, the sub-class of `Agent` must do this before sending an order. Such pre-emptive validations save an unnecessary round trip to the servers. In order to perform these validations an agent can query the `class Market` for various market settings, described on page 13.

3. **Running a trading agent**

   When running a trading agent (as described in section 3.3), there are certain methods that a sub-class of `class Agent` can optionally implement. These methods allow further customisation.

   - `def initialised(self)`

     This method is called when an agent has successfully initialised (see Section 3.3 for details).

   - `def pre_start_tasks(self)`

     This method is called by the `run` method to execute any sub-class specific tasks. This method should be used for any post-initialisation and pre-start tasks, such as scheduling any recurrent task by using various methods that are named like `def execute_periodically_*`.

3.3 ROBOT EXECUTION

To run an agent one needs to call `def run()` from the parent class.

1. `def run(self)`

   The `run()` method is used to start execution of the sub-class of `Agent`. This method is of relevance when an agent is hosted on third party platforms (e.g., the hosting platform provided). To start execution of an agent the third party platform first needs to create an object of the sub-class and then call its run method (see Section 3.3 for details). This method performs three tasks: calls `def initialise()`, `def pre_start_tasks()` and finally `def start()`.

   Note: Though an agent can be executed by calling appropriate methods from the parent class outside of the `run()` method, we advice against it.

2. `def initialise(self)`

   The `initialise()` method executes 4 key tasks:

   - Log into the Flex-E-Markets platform using given account details;
   - Connect to the updates channel on Flex-E-Markets platform;
   - Retrieve details of available markets;
   - Connect to the hosting platform (if available).

   Upon successful completion of these tasks the agent then calls the `initialised()` method. The agent sub-class can perform any post initialisation configuration of the agent in the implementation of `initialised()` method (e.g., getting available markets).

3. `def start(self)`

   The `start()` method causes the Agent to start any long-running tasks and updates on order acceptance and rejection. In addition, it establishes a communication channel with the hosting platform (if available).

4.
   ```
   def execute_periodically(self, func: callable,
   ↪  sleep_time: int)
   ```

   This method can be used to schedule any custom methods that a trading agent may have. For e.g., to run periodically (every `sleep_time` seconds) any reasoning methods, execution timers, etc. (see example bot)

5.
   ```
   def execute_periodically_variably(self, func: callable,
   ↪  sleep_time_func: callable):
   ```

This method can be used to schedule any custom methods that a trading agent may have, based on varying sleep time. For e.g., to run periodically (every `x` seconds, as returned by `sleep_time_func`) any reasoning methods, execution timers, etc. (see example bot)

6.
   ```
   def execute_periodically_conditionally(self, func:
   ↪ callable, sleep_time: int, condition: callable):
   ```

   This method can be used to schedule any custom methods that a trading agent may have, if the `condition` evaluates to `True`. For e.g., to run periodically (every `sleep_time` seconds) if `condition` is `True`, any reasoning methods, execution timers, etc. (see example bot)

7.
   ```
   def execute_periodically_variably_conditionally(self,
   ↪ func: callable, sleep_time_func: callable,
   ↪ condition: callable):
   ```

   This method can be used to schedule tasks that run every `x` seconds, as returned by `sleep_time_func`, if the `condition` evaluates to `True`

Note: All these methods are fully implemented in the `Agent` class and the respective sub-class merely needs to implement any optional methods for customisation.

To stop the execution of an agent one needs to set its `@property` `stop` to `True`. For example, the line `fm_bot.stop = True` will stop the execution of the `fm_bot` agent. Note that the agent may take some time before stopping. The `Agent` class runs multiple concurrent tasks and it waits for the pending tasks to complete before stopping its execution.

4 MARKET INTERACTION

An agent interacts in a market by sending orders. Orders are submitted specific to each market, and for this the sub-class needs to know the respective market.

4.1 MARKET INFORMATION

The agent constructor takes the id of the marketplace and retrieves the available markets during initialisation. The sub-class agent can extract the ids of the respective markets inside the `initialised` method. The market attribute of the Agent class is a dictionary with market id as its key and information about the market (such as its name, the item's name, its description as its value).

```
def initialised(self):
    for market_id, market in self.markets.items():
        self.inform(f"I can trade {market.item} by sending
         ↪  orders with market id {market_id}")
```

An object of `class Market` has following commonly used properties:

- `fm_id`: ID of this market;
- `item`: item/symbol name of this market;
- `name`: general name of this market;
- `description`: a description of this market
- `min_price`: minimum price (in cents) per unit of an order submitted in this market;
- `max_price`: maximum price (in cents) per unit of an order submitted in this market;
- `min_units`: minimum number of units in a single order submitted in this market;
- `max_units`: maximum number of units in a single order submitted in this market;
- `price_tick`: step size (in cents) for price of an order submitted in this market (price is a multiple of this value);
- `unit_tick`: step size for number of units in an order submitted in this market (number of units must be a multiple of this value);
- `private_market`: `True` if this is a private market and `False` otherwise;

- private_traders: a list of names of private traders in a private market;

The class Market can also be queried directly. For e.g., to get a Market object, if we only know the market item name, by using @classmethod Market.get_by_item(cls, item_) -> Optional[Market]. Or if we know the market id, @classmethod Market.get_by_id(cls, id_) -> Optional[Market]. These methods can only be called after the agent has finished initialisation.

4.2 ORDER MANAGEMENT

- **Sending an order**: To send an order one first creates an object of the class Order and sets the required values. Then, one passes this object to the send_order() method. The example below sends a LIMIT order with price of $1 per unit, to sell 2 units, for market with id 50.

```
def create_new_order():
    market = Market(50)
    new_order = Order.create_new()
    new_order.market = market
    new_order.price = 10
    new_order.units = 1
    new_order.order_side = OrderSide.SELL
    new_order.order_type = OrderType.LIMIT
    new_order.ref = 'Test Order; Sell 2'
    self.send_order(new_order)
```

The ref argument is to track order acceptance and rejection as these acknowledgements arrive asynchronously.

```
def order_accepted(self, order):
    if order.ref == 'Test Order; Sell 2':
        self.inform("Sell order was accepted!")
```

Tip: It is a good practice to keep a track of the orders that were accepted. Perhaps store references in a list

If an agent wants to send a private order (in a private market), then a 'target' needs to be specified. For example,

```
import random  # Normally this would be at the top of
↪  a python module
...
# While creating a new order (see above code)
if market.private_market:
    new_order.owner_or_target =
    ↪  random.choice(market.private_traders)  # Just
    ↪  an example for choosing a trader randomly
...
```

- **Cancelling an order**: To cancel an accepted order one needs to know the id of the order that was set by the server. When the server accepts an order it assigns an id to it and returns this information in its response. The Agent class extracts the id from the server's response and assigns it to the order object that was passed in the `send_order(...)` method. A copy of this order object is also passed in the `order_accepted(...)` method, and the sub-class can use these order objects to cancel orders.

  ```
  if order.mine and not order.is_consumed:
      cancel_order = copy.copy(order)
      cancel_order.type = OrderType.CANCEL
      cancel_order.ref = "sell2_cancel"
      self.send_order(cancel_order)
  ```

  Note: It is best practice to check whether an order is 'mine' before trying to cancel it.

- **Active orders**: To track active orders, one needs to implement the `received_orders(...)` method and check for the `mine` attribute of the order objects.

  ```
  def received_orders(self, orders: List[Order]):
      for order in orders:
          if order.mine:
              self.inform("This is my order!")
  ```

4.3 HOLDINGS

A trader's holdings will change when their order is submitted, is cancelled, gets executed, or when the marketplace manager decides to alter their holdings (such as giving dividends). The Agent class only stores the last available holding information in its `self.holdings` attribute; it does not track any change in holdings. If one needs to track the changes in holdings, this should be implemented in the sub-class. For details on the holdings please refer the details of the `received_holdings(...)` method on page 9.

5 INFORMATION LOGGING

The Agent class creates a log file when it starts execution in folder called `logs`. Currently the agent outputs minimal log information, in particular, on sending the initial marketplace information request and then on receiving it.

```
[2017-08-15 09:11:10: agent.FMBot]: INFO - Requesting initial
↪  marketplace information.
[2017-08-15 09:11:15: agent.FMBot]: INFO - Received initial
↪  marketplace information.
```

Additionally, if also logs when the marketplace status changes (i.e., when it opens or closes).

```
[2017-08-15 10:32:32: agent.FMBot]: INFO - The session status
↪  has changed. Is it open?True
```

The Agent class provides an `inform()` method to log custom information messages. The method logs the message and also sends it to the algorithm hosting display (if it is available). For example, the following snippet logs when an order gets accepted.

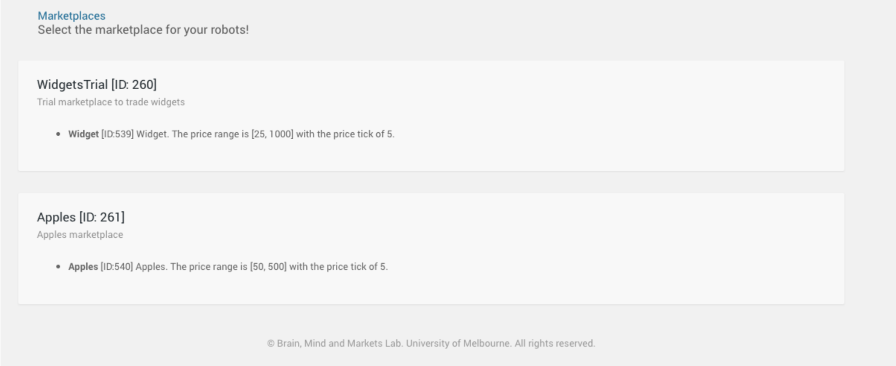
```python
def order_accepted(self, order: Order):
    self.inform(f"Order accepted: {order}")
```

Tip: We advise using the inform method instead of the print method to output information.

Note that the Agent class also provides methods to log `warning`, `error`, and `debug`. However, their output is dependent on the logging level of the Agent class (by default it is set to `INFO`).

6 HOSTING OF ALGORITHMS

We provide a hosting platform (`https://algohost.bmmlab.org`) where users can upload their robots and run them online. After signing in to the hosting platform (it accepts the same credentials as Flex-E-Markets) one is presented with the available marketplaces (see Figure 3 for an example). Note that the hosting platform displays the marketplaces differently to the Flex-E-Markets interface. In particular, it displays the marketplace id to be used when creating trading agents for the relevant marketplace.



Marketplaces
Select the marketplace for your robots!

WidgetsTrial [ID: 260]
Trial marketplace to trade widgets

- **Widget** [ID:539] Widget. The price range is [25, 1000] with the price tick of 5.

Apples [ID: 261]
Apples marketplace

- **Apples** [ID:540] Apples. The price range is [50, 500] with the price tick of 5.

© Brain, Mind and Markets Lab. University of Melbourne. All rights reserved.

Figure 3: List of available marketplaces on the hosting platform.

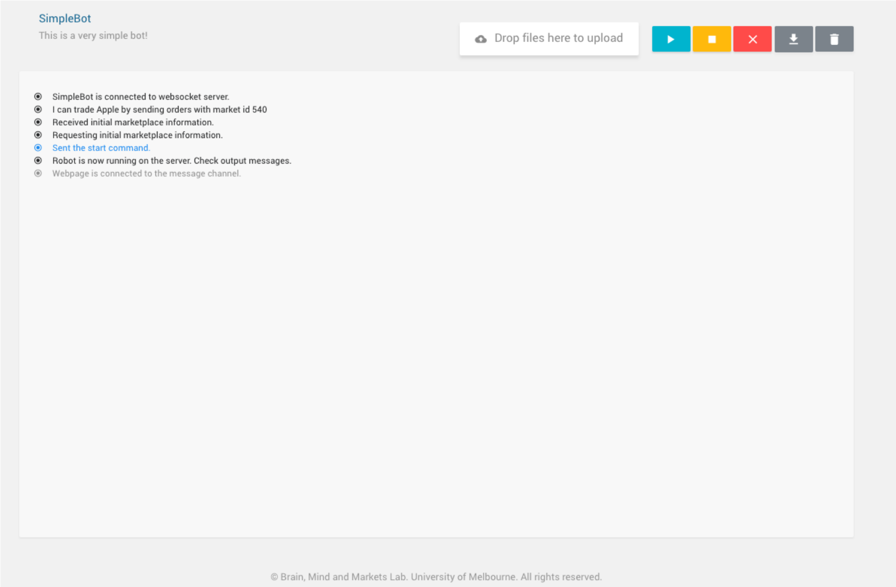
The control interface to upload and manage the agents is available on clicking the name of a marketplace. The upload functionality *only accepts a single Python file* that contains the sub-class trader agent definition and the main method to run the agent (please see skeleton agent in the appendix for a minimal example). Any subsequent uploads will override the previously uploaded file.

The server performs a check on the uploaded file to ensure that it contains the definition of an Agent sub-class. Please ensure that the constructor of the uploaded agent definition takes 4 mandatory arguments, namely, the account name, the email, the password, and the marketplace id (see details on the constructor on page 5).

The name and description of the uploaded robot are visible on the interface. These parameters are picked from the attributes `name` and `description`.

The display area on the right side displays the messages that are logged by the trading agent (see logging section for details). These messages are displayed in real time and for older messages one needs to consult the log file. The log file can be downloaded using the download button on the interface.

The stop button sends a stop message to the agent, which when receives the message sets `self.stop=True`. In case the trader agent does not stop after a while, it can be forcefully terminated by using



SimpleBot

This is a very simple bot!

- SimpleBot is connected to websocket server.
- I can trade Apple by sending orders with market id 540
- Received initial marketplace information.
- Requesting initial marketplace information.
- Sent the start command.
- Robot is now running on the server. Check output messages.
- Webpage is connected to the message channel.

© Brain, Mind and Markets Lab. University of Melbourne. All rights reserved.

Figure 4: Robot control and message display interface on the hosting platform.

the kill button.

APPENDIX A

A.1 SKELETON AGENT

```python
from copy import copy
from typing import List

from fmclient import Agent
from fmclient import Holding
from fmclient import Market, Asset
from fmclient import Order, OrderSide, OrderType
from fmclient import Session, SessionState


class FMBot(Agent):
    """ This is a very simple "Hello World" agent! """

    def __init__(self, account, email, password, marketplace_id):
        name = "SimpleBot"
        super().__init__(account, email, password, marketplace_id,
        ↪  name=name)
        self.description = "This is a very simple bot!"

    def initialised(self):
        pass

    def pre_start_tasks(self):
        pass

    def received_orders(self, orders: List[Order]):
        pass

    def received_holdings(self, holdings: Holding):
        pass

    def received_session_info(self, session: Session):
        pass

    def order_accepted(self, order: Order):
        pass

    def order_rejected(self, info: dict, order: Order):
        pass


    if __name__ == "__main__":
        marketplace_id = 1
        fm_bot = FMBot("durable-guard", "test@at", "aLgoTraDer",
        ↪  marketplace_id)
        fm_bot.run()
```