**Polaris project**

# Market class

The market class manages the interface with the ***Binance API***, principally:

1. Sockets management
   * start
   * stop
   * calllbacks
2. Socket callbacks
   * symbol ticker socket callback
   * user socket callback
3. Other queries:
   * get asset balance
   * get trade fee
4. Data sent to Binance API:
   * create order

The lass is like a bridge between the Binance API and the Trade Manager class where all the logic is done, together with Perfect Trade and Order classes.

A picture containing text, whiteboard

Description automatically generated

## Properties:

|  |  |
| --- | --- |
| \_tm | Pointer to the trade manager |
| \_client | Pointer to the Binance API |
| \_is\_running | Flag to mark whether the market is receiving the data through the sockets or not.  The app allows a fake mode, where two buttons on the dashboard simulate the market price variations |

## Private methods

|  |  |
| --- | --- |
| \_user\_socket\_callback() |  |
| \_symbol\_ticker\_socket\_callback() |  |
| \_start\_sockets() |  |
| \_get\_balance() |  |
| \_get\_fees\_d() |  |
| \_get\_client\_keys() |  |

## Getters & Setters

|  |  |
| --- | --- |
| isRunning() |  |

## Public methods

|  |  |
| --- | --- |
| start() |  |
| stop() |  |
| place\_order() |  |

# TradeManager class

The TradeManager class manages the perfect tradings initiation, launching a new one every time certain conditions are accomplished.

This class is placed between the Market class and the DashboardManager class, using the PerfectTrade class.

Its main functions are:

1. Start/stop the global session. When the session is active, he market price is updated ‘life’ from Binance through the socket.
2. Process the sockets callback functions, that from market redirect to methods in the TradeManager class.
3. Manage the variables that store all the aggregated balance data:
   * net and gross balance
   * perfect trades count
   * orders placed count
   * orders canceled count
   * compensations count

Diagram

Description automatically generated

## Properties:

|  |  |
| --- | --- |
| \_dbm | Pointer to the dashboard manager |
| \_market | Pointer to the Market |
| \_mp\_f | Current market price |
| \_trades | Trades list |
| \_is\_session\_active | Flag to mark the session active or finished |
| \_ticket\_count | counter that increases at every symbol\_ticker update (approximately every second) and used as a kind of ‘timer’ to create new perfect trades |
| \_net\_balance\_f |  |
| \_gross\_balance\_f |  |
| \_perfect\_trade\_count |  |
| \_order\_count |  |
| \_canceled\_count |  |
| \_compensation\_count |  |

## Private methods

|  |  |
| --- | --- |
| \_start\_session() | starts the session and inits all the variables related to balances, counters, etc… |
| \_stop\_session() | finishes the session |
| \_cycle() | called from the redirected symbol ticker socket callback method one\_loop() to iterate through the trades list and update orders status from MONITOR to ACTIVE when needed |
| \_create\_new\_pt() | if conditions are reached, creates a new perfect trade |
| \_new\_pt\_allowed() | checks whether the timer reaches zero or the trades list is empty |
| \_fake\_trades() | in simulator/fake mode, it checks whether the market price reaches the order price, and if it is true then in fakes an order trade |

## Getters & Setters

|  |  |
| --- | --- |
| setMarket()() |  |

## Public methods

|  |  |
| --- | --- |
| tm\_order\_traded() | called from the user socket callback function when an order has been traded |
| tm\_symbol\_ticker\_updated () | called from the symbol ticker socket callback function every time the market price is updated |

Text, letter

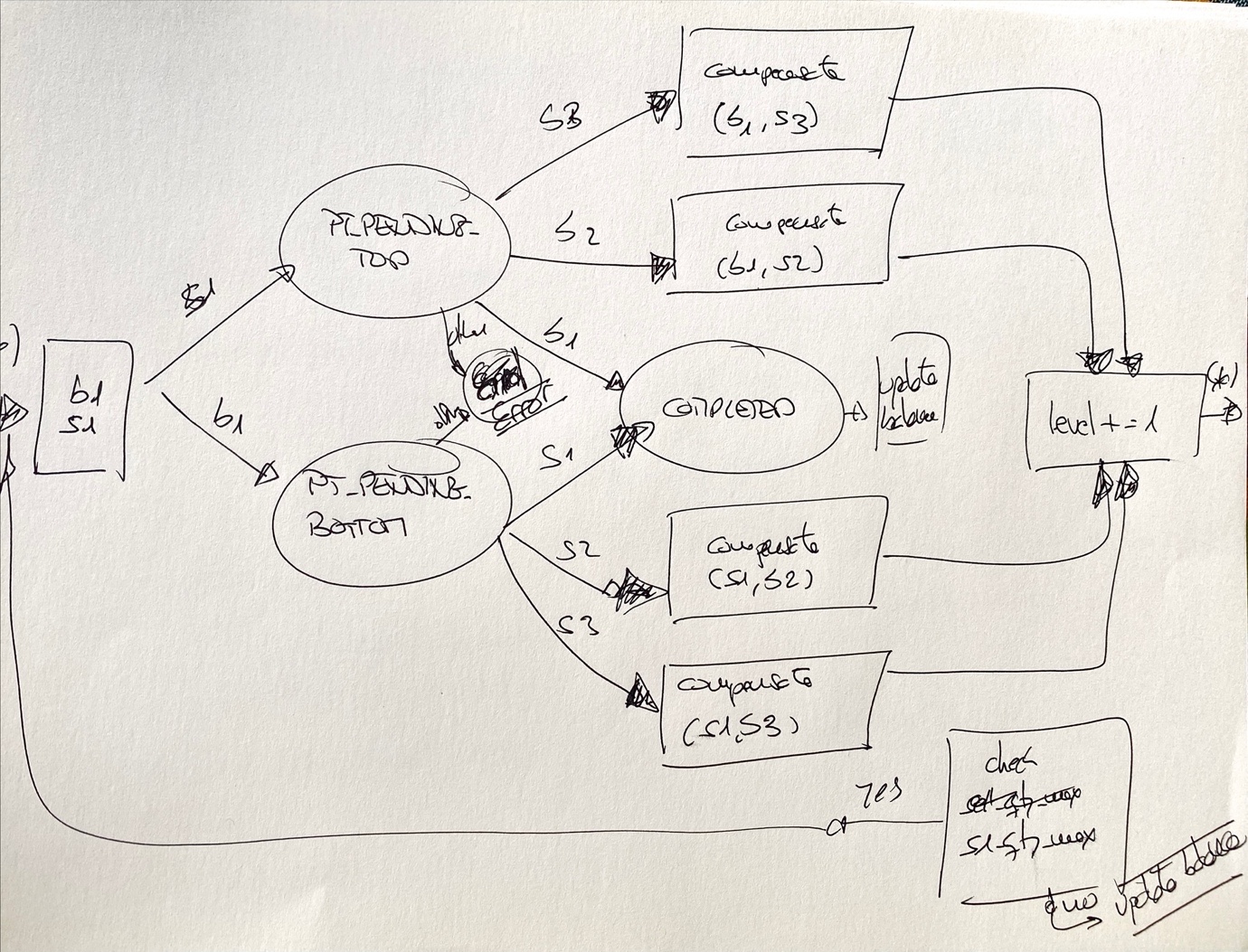
Description automatically generated

# Perfect Trade class

The perfect trade class manages the core functions of each trade, performing the strategy about which orders to create, when to create them and which prices and quantities should have.

Once the trade is initiated with b1 and s1, it remains in standby and only changes when info from one of the two sockets is received:

1. **Symbol ticker socket**: through this socket the updates of the market price are received, and it is sent to each order to change its status from MONITOR to ACTIVE when the activation value is reached.
2. **User socket**: through this socket the info of an order traded is received. It will trigger a change in status according to the strategy shown in the following diagram.



When b1 and s1 are created, they are both at MONITOR status.

## PerfectTrade creation

This class has to know the following data:

* How to create the first pair of orders: b1 and s1
* How to create the second and third pair of orders:
  + b2 and s2 will be STOP\_LOSS\_LIMIT or TAKE\_PROFIT\_LIMIT
  + b3 and s3 will be MARKET orders
* The activation values of all orders

Therefore, the following parameters have to be passed to \_\_ini\_\_():

* **mp\_f:** Current market price. Used to calculate first pt values
* **symbol**: ‘BTCEUR’
* **min net balance**: The minimum gain in BTC (or symbol\_1) once the perfect trade is correctly completed
* **s1\_qty**: The amount in BTC (or symbol\_1) to place in s1
* **s1\_qty\_max**: After each compensation, the new s1\_qty value has to be check against this max quantity and, if it is higher, the perfect trade will result as uncomplete (**not clear what to do in this case yet**)
* **buy\_fee**
* sell\_fee
* **kp2**: Where to exactly place b2 and s2
* **kp3**: Where to exactly place b3 and s3
* **ka1**: Market place value that, when reached, b1 or s1 have to be activated (M->A)
* **ka2**: Market place value that, when reached, b2 or s2 have to be activated (M->A)
* **ka3**: Market place value that, when reached, b3 or s3 have to be activated (M->A)
* **shift**: The difference between stop price and limit price of b2 and s2

To create the first pair of orders b1 and s1, the four main parameters (b1q, b1p, s1p and g) are obtained, using the **PerfectTradeCalculator** class, from:

* mp\_f
* s1\_qty
* min net balance
* buy fee
* sell fee

Then the pt1 is created through the following steps:

* set activation values for b1 and s1
* create b1 and s1
* empty orders list
* append b1 and s1 to the orders list
* increase level by 1 (in \_\_init\_\_() it means from 0 to 1)
* log
* set pt status to PT1\_PENDING

## Properties:

|  |  |
| --- | --- |
| \_symbol | Symbol ‘BTCEUR’ |
| \_mnb | Minimum net balance |
| \_s1\_qty\_ref | The s1 quantity used as reference |
| \_s1\_qty\_max |  |
| \_buy\_fee |  |
| \_sell\_fee |  |
| \_trade\_id |  |
| \_kp2 |  |
| \_kp3 |  |
| \_ka1 |  |
| \_ka2 |  |
| \_ka3 |  |
| \_shift |  |
| \_reference\_mp\_f | The first mp, which created pt1, and the one used together with \_kp2 and\_kp3 to calculate the second and third order pairs |
| \_level |  |
| \_timestamp\_active | when the perfect trade was created |
| \_timestamp\_completed | when the perfect trade was completed |
| \_orders | orders list |
| \_g | reference gap, used to create the first pair of orders: b1 and s1 |
| \_status | perfect trade status:   * PT1\_PENDING * PT1\_COMPLETED\_TOP * PT1\_COMPLETED\_BOTTOM * COMPLETED |

Properties can be classified in four groups:

|  |  |  |  |
| --- | --- | --- | --- |
| overall | to create pt1 | references/consts. | orders list |
| *\_symbol* | *\_reference\_mp\_f* | *\_kp2* | *\_orders* |
| *\_trade\_id* | *\_mnb* | *\_kp3* |  |
| *\_level* | *\_s1\_qty\_ref* | *\_ka1* |  |
| *\_status* | *\_buy\_fee* | *\_ka2* |  |
| *\_timestamp\_active* | *\_sell\_fee* | *\_ka3* |  |
| *\_timestamp\_completed* | *\_g* |  |  |

## Private methods

|  |  |
| --- | --- |
| \_setup\_pt1() |  |
| \_create\_pt1() |  |
| \_compensate() |  |
| \_get\_order\_by\_name() |  |
| \_cancel\_pending\_orders() |  |
| \_completed() |  |
| \_log\_error() |  |

## Getters & Setters

|  |  |
| --- | --- |
| getOrders() |  |
| getStatus() |  |

## Public methods

|  |  |
| --- | --- |
| order\_traded() | called from the user socket callback function when an order has been traded |
| symbol\_ticker\_updated () | called from the symbol ticker socket callback function every time the market price is updated |

## pt\_symbol\_ticker\_update()

Iterates through all the orders in the orders list and call the update\_status() method, passing as parameters:

* The current market price.
* A pointer to the market, needed when an order has to be placed (M->A). It happens when the market price reaches the order activation value/price.

1. def pt\_symbol\_ticker\_updated(self, mp\_f, market):
2. for order in self.\_orders:
3. order.update\_status(mp\_f, market)

## pt\_order\_traded()

This is the **core method** of dawning app.

# Order class

The order is placed in the Binance market through the **self.place()** method, called from **self.set\_status()** method/setter, called from **self.update\_status()** method.

## Properties:

|  |  |
| --- | --- |
| \_market | the pointer to market is needed when placing an order to call the method in the market class |
| \_name | Name of the order: b1, s1, b2, s2, b3 or s3 |
| \_symbol | ‘BTCEUR’ |
| \_side |  |
| \_type |  |
| \_quantity\_f |  |
| \_price\_f |  |
| \_stop\_price\_f |  |
| \_trade\_id |  |
| \_fee |  |
| \_commission\_amount\_f |  |
| \_activation\_mp |  |
| \_activation\_type |  |
| \_status |  |
| \_time\_in\_force |  |
| \_order\_id |  |
| \_creation\_timestamp |  |
|  |  |
|  |  |
|  |  |

## Private methods

|  |  |
| --- | --- |
| \_place\_to\_market() | It calls the method **place\_order()** in the market class to place the order in the Binance market. That is why the pointer to market is needed in the order class (stored in **self.\_market**) |

## Getters & Setters

|  |  |
| --- | --- |
| setStatus() | It sets the new status. If it is ACTIVE, then it calls the **place()** method |
| setPrice() | Used in **tm\_order\_traded()** method after getting the actual price of a MARKET order |
| setCommissionAmount\_f() |  |
| getStatus() |  |
| getSide() |  |
| getType() |  |
| getTimeInForce() |  |
| getPriceStr() |  |
| getStopPriceStr() |  |
| getPriceFloat() |  |
| getCommissionAmount\_f() |  |
| getQty() |  |
| get\_name() |  |
| getSymbol() |  |
| getTradeId() |  |
| getOrderId() |  |
| get\_distance() | This is a calculated getter: (mp – price). Check whether it is used or not |

## Public methods

|  |  |
| --- | --- |
| update\_status() | Called from **pt\_user\_socket\_updated()** method. It checks whether an order status has to be changed from MONITOR to ACTIVE. If so, it calls the **set\_status()** method/setter |
| format\_for\_list() | Called from the **update()** method of the dashboard manager to properly format the order string presentation in the dashboard lists |

Assess whether set\_status() and place() methods could be private

# Simulated market class

When the app is started, it has to be passed a mandatory command line argument *- - mode*, with two possible values:

* **binance**
* **simulated**

It will determine the mode in which the app will operate: using the actual binance api (**binance**) or using a simulated market (**simulated**), with the following considerations:

* Once the app is started its mode cannot be changed.
* In binance mode, the market can be stopped, prior its dashboard window closing (this is required to stop **reactor** before closing the window).
* In binance mode the self.\_client property points to the **SimulatedMarket** object.

# Exceptions

# Binance API

Check order types available for a symbol:

1. info\_d = client.get\_exchange\_info()

It returns a dictionary with the following interesting items:

1. info\_d = client.get\_exchange\_info()
3. for symbol in info\_d[‘symbols’]:
4. if symbol[‘symbol’] == ‘BTCEUR’:
5. print(symbol[‘orderTypes’]

For the ‘BTCEUR’ symbol, the following order types are allowed:

* LIMIT
* LIMIT\_MAKER
* MARKET
* STOP\_LOSS\_LIMIT
* TAKE\_PROFIT\_LIMIT

The signature for **STOP\_LOSS\_LIMIT** and **TAKE\_PROFIT\_LIMIT** is different than for LIMIT, because it must include the ***stopPrice*** parameter

# Exchange info

**get\_exchange\_info()[[source]](https://python-binance.readthedocs.io/en/latest/_modules/binance/client.html" \l "Client.get_exchange_info)**

Return rate limits and list of symbols

|  |  |
| --- | --- |
| **Returns:** | list - List of product dictionaries |

{

"timezone": "UTC",

"serverTime": 1508631584636,

"rateLimits": [

{

"rateLimitType": "REQUESTS",

"interval": "MINUTE",

"limit": 1200

},

{

"rateLimitType": "ORDERS",

"interval": "SECOND",

"limit": 10

},

{

"rateLimitType": "ORDERS",

"interval": "DAY",

"limit": 100000

}

],

"exchangeFilters": [],

"symbols": [

{

"symbol": "ETHBTC",

"status": "TRADING",

"baseAsset": "ETH",

"baseAssetPrecision": 8,

"quoteAsset": "BTC",

"quotePrecision": 8,

"orderTypes": ["LIMIT", "MARKET"],

"icebergAllowed": false,

"filters": [

{

"filterType": "PRICE\_FILTER",

"minPrice": "0.00000100",

"maxPrice": "100000.00000000",

"tickSize": "0.00000100"

}, {

"filterType": "LOT\_SIZE",

"minQty": "0.00100000",

"maxQty": "100000.00000000",

"stepSize": "0.00100000"

}, {

"filterType": "MIN\_NOTIONAL",

"minNotional": "0.00100000"

}

]

}

]

}

|  |  |
| --- | --- |
| **Raises:** | BinanceRequestException, BinanceAPIException |

# Get symbol info

**get\_symbol\_info(*symbol*)[[source]](https://python-binance.readthedocs.io/en/latest/_modules/binance/client.html" \l "Client.get_symbol_info)**

Return information about a symbol

|  |  |
| --- | --- |
| **Parameters:** | **symbol** (*str*) – required e.g BNBBTC |
| **Returns:** | Dict if found, None if not |

{

"symbol": "ETHBTC",

"status": "TRADING",

"baseAsset": "ETH",

"baseAssetPrecision": 8,

"quoteAsset": "BTC",

"quotePrecision": 8,

"orderTypes": ["LIMIT", "MARKET"],

"icebergAllowed": false,

"filters": [

{

"filterType": "PRICE\_FILTER",

"minPrice": "0.00000100",

"maxPrice": "100000.00000000",

"tickSize": "0.00000100"

}, {

"filterType": "LOT\_SIZE",

"minQty": "0.00100000",

"maxQty": "100000.00000000",

"stepSize": "0.00100000"

}, {

"filterType": "MIN\_NOTIONAL",

"minNotional": "0.00100000"

}

]

}

|  |  |
| --- | --- |
| **Raises:** | BinanceRequestException, BinanceAPIException |

# Place order

1. >>> from src.market import Market, MarketMode
2. >>> mode = MarketMode.BINANCE
3. >>> market = Market(None, mode)
4. >>> order = market.\_client.order\_limit\_buy(symbol='BTCEUR', quantity=0.001, price='50000.0', newClientOrderId=’OR000001af3c’)
5. >>> order
6. {'symbol': 'BTCEUR', 'orderId': 555472500, 'orderListId': -1, 'clientOrderId': 'OR000001af3c', 'transactTime': 1618521179140, 'price': '50000.00000000', 'origQty': '0.00100000', 'executedQty': '0.00000000', 'cummulativeQuoteQty': '0.00000000', 'status': 'NEW', 'timeInForce': 'GTC', 'type': 'LIMIT', 'side': 'BUY', 'fills': []}

To cancel the previous order we can use either the orderId or the clientOrderId:

1. >>> result = market.\_client.cancel\_order(symbol='BTCEUR', orderId=555472500)
2. >>> result
3. >>> {'symbol': 'BTCEUR', 'origClientOrderId': ' OR000001af3c', 'orderId': 555472500, 'orderListId': -1, 'clientOrderId': 'jDBLGYh2g3a5urkkpwNwou', 'price': '50000.00000000', 'origQty': '0.00100000', 'executedQty': '0.00000000', 'cummulativeQuoteQty': '0.00000000', 'status': 'CANCELED', 'timeInForce': 'GTC', 'type': 'LIMIT', 'side': 'BUY'}

Once canceled, the result message shows the original clientOrderId as origClientOrderId

# Cancel order

python-binance:

Graphical user interface, text, application, email

Description automatically generated

Binance API:

Graphical user interface, application

Description automatically generated

**Response:** (Binance API)

{

"symbol": "LTCBTC",

**"origClientOrderId": "myOrder1"**,

"orderId": 4,

"orderListId": -1, //Unless part of an OCO, the value will always be -1.

"clientOrderId": "cancelMyOrder1",

"price": "2.00000000",

"origQty": "1.00000000",

"executedQty": "0.00000000",

"cummulativeQuoteQty": "0.00000000",

"status": "CANCELED",

"timeInForce": "GTC",

"type": "LIMIT",

"side": "BUY"

}

# Order traded

Received through the user socket (Order update) with **[‘e’] = ‘executionReport’**.

Other interesting items:

1. [‘x’] Execution type:
   1. NEW
   2. CANCELED
   3. REJECTED
   4. REPLACED
   5. TRADE
   6. EXPIRED
2. [‘X’] Order status:

Table

Description automatically generated with medium confidence

# Websockets

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generated

Steps:

1. Create a listen key
2. Ping/keep alive the listen key
3. Close the listen key
4. def \_start\_sockets(self):
5. # init socket manager
6. self.\_bsm = BinanceSocketManager(self.\_client)
8. # symbol ticker socket and get the **listen key**
9. self.\_symbol\_ticker\_s = self.\_bsm.start\_symbol\_ticker\_socket(
10. 'BTCEUR',
11. self.\_symbol\_ticker\_socket\_callback)
13. # user socket
14. self.\_user\_s = self.\_bsm.start\_user\_socket(
15. self.\_user\_socket\_callback)
17. # start sockets
18. self.\_bsm.start()
20. self.\_bsm.stop\_socket(self.\_symbol\_ticker\_s)
21. self.\_bsm.stop\_socket(self.\_user\_s)
23. # properly close the WebSocket, only if it is running
24. # trying to stop it when it is not running, will raise an error
25. if reactor.running:
26. reactor.stop()

# User socket

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

After placing an order, through ‘outboundAccountPosition’ we receive the updated balance of our cryptos:

* BTC
* BNB
* EUR

Each one would have to be updated in the dashboard in two ways:

* total
* for each active perfect trade

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

Text

Description automatically generated with low confidence

Graphical user interface, text, application, Teams

Description automatically generated

Text

Description automatically generated with medium confidence

# API errors

## Messages for -1010 ERROR\_MSG\_RECEIVED, -2010 NEW\_ORDER\_REJECTED, and -2011 CANCEL\_REJECTED

This code is sent when an error has been returned by the matching engine. The following messages which will indicate the specific error:

| **Error message** | **Description** |
| --- | --- |
| "Unknown order sent." | The order (by either orderId, clOrdId, origClOrdId) could not be found |
| "Duplicate order sent." | The clOrdId is already in use |
| "Market is closed." | The symbol is not trading |
| "Account has insufficient balance for requested action." | Not enough funds to complete the action |
| "Market orders are not supported for this symbol." | MARKET is not enabled on the symbol |
| "Iceberg orders are not supported for this symbol." | icebergQty is not enabled on the symbol |
| "Stop loss orders are not supported for this symbol." | STOP\_LOSS is not enabled on the symbol |
| "Stop loss limit orders are not supported for this symbol." | STOP\_LOSS\_LIMIT is not enabled on the symbol |
| "Take profit orders are not supported for this symbol." | TAKE\_PROFIT is not enabled on the symbol |
| "Take profit limit orders are not supported for this symbol." | TAKE\_PROFIT\_LIMIT is not enabled on the symbol |
| "Price \* QTY is zero or less." | price \* quantity is too low |
| "IcebergQty exceeds QTY." | icebergQty must be less than the order quantity |
| "This action disabled is on this account." | Contact customer support; some actions have been disabled on the account. |
| "Unsupported order combination" | The orderType, timeInForce, stopPrice, and/or icebergQty combination isn't allowed. |
| "Order would trigger immediately." | The order's stop price is not valid when compared to the last traded price. |
| "Cancel order is invalid. Check origClOrdId and orderId." | No origClOrdId or orderId was sent in. |
| "Order would immediately match and take." | LIMIT\_MAKER order type would immediately match and trade, and not be a pure maker order. |
| "The relationship of the prices for the orders is not correct." | The prices set in the OCO is breaking the Price rules. The rules are: SELL Orders: Limit Price > Last Price > Stop Price BUY Orders: Limit Price < Last Price < Stop Price |
| "OCO orders are not supported for this symbol" | OCO is not enabled on the symbol |
| "Quote order qty market orders are not support for this symbol." | MARKET orders using the parameter quoteOrderQty are not enabled on the symbol. |

## -9xxx Filter failures

| **Error message** | **Description** |
| --- | --- |
| "Filter failure: PRICE\_FILTER" | price is too high, too low, and/or not following the tick size rule for the symbol. |
| "Filter failure: PERCENT\_PRICE" | price is X% too high or X% too low from the average weighted price over the last Y minutes. |
| "Filter failure: LOT\_SIZE" | quantity is too high, too low, and/or not following the step size rule for the symbol. |
| "Filter failure: MIN\_NOTIONAL" | price \* quantity is too low to be a valid order for the symbol. |
| "Filter failure: ICEBERG\_PARTS" | ICEBERG order would break into too many parts; icebergQty is too small. |
| "Filter failure: MARKET\_LOT\_SIZE" | MARKET order's quantity is too high, too low, and/or not following the step size rule for the symbol. |
| "Filter failure: MAX\_POSITION" | The account's position has reached the maximum defined limit. This is composed of the sum of the balance of the base asset, and the sum of the quantity of all open BUY orders. |
| "Filter failure: MAX\_NUM\_ORDERS" | Account has too many open orders on the symbol. |
| "Filter failure: MAX\_ALGO\_ORDERS" | Account has too many open stop loss and/or take profit orders on the symbol. |
| "Filter failure: MAX\_NUM\_ICEBERG\_ORDERS" | Account has too many open iceberg orders on the symbol. |
| "Filter failure: EXCHANGE\_MAX\_NUM\_ORDERS" | Account has too many open orders on the exchange. |
| "Filter failure: EXCHANGE\_MAX\_ALGO\_ORDERS" | Account has too many open stop loss and/or take profit orders on the exchange. |

# Python

## Command line arguments

1. pip install argparse
2. # Import the library  
   import argparse
3. # Create the parser  
   parser = argparse.ArgumentParser()
4. # Add an argument  
   parser.add\_argument('--name', type=str, required=True)
5. # Parse the argument  
   args = parser.parse\_args()
6. # Print "Hello" + the user input argument  
   print('Hello,', args.name)

# Developer log

01/04/2021

## TODO#01:

To solve the problem with negative quantities in b1 and s1 after second compensation. Binance rise an exception when trying to place b1(level 3) with a negative quantity.

One problem detected when matching with Excel:

* When creating s2 and b2 in **\_create\_s2\_and\_b2()** method, the **mp** passed as parameter is not the original market price used to create the first perfect pair (b1, s1) in level 1. Therefore we have to keep a reference to the original value, that is **self.\_reference\_mp\_f** and use it to create the perfect trade pair (b2, s2).

Another problem detected:

* At each level, when creating the second perfect trade pair (b2, s2) or (s2, b2), the reference values to use are different from the ones at level 1. Therefore, after the compensation the following reference values have to be updated:
  + **\_reference\_mp\_f**
  + **\_reference\_g**
  + **\_reference\_nmb**

The variables to play with at level 2 and beyond, and fine-tune are the following:

* **\_reference\_mnb**: it should be zero because the net profit is taken into account at level 1, and at higher levels the goal is to complete the perfect trade as soon as possible, rather than increasing the net profit. Increasing the reference nmb would create a much greater g.
* **Minimum net profit** at stage three, values after compensation. Initially it is considered a value of zero, because a greater value would imply greater quantities after compensation, but a small amount could be assessed and finally considered.
* **\_kg**: being the value to calculate the gap after compensation in reference to **\_ref\_g**.

## TODO #02

How to calculate the initial values **b1**, **s1**, **reference mp**, **reference g** and **reference nmb** at each level:

* At level one, (b1, s1) are calculated creating a perfect trade around the current market price (reference\_mp), and reference g and reference nmb are parameters defined in the app.
* At the following levels, (b1, s1) are the outcome of the compensation, reference mp is the current market price, same as the price of the last traded order, and reference g is the g considered when performing the last compensation.

02/04/2021

## TODO #02 (cont.)

Modified the perfect trade class with the following:

* In \_compensate() method:
  + delete active orders
  + setup reference values
* In several methods, changes made to use the reference values

03/04/2021

## TODO #03

In trade manager added the following initial parameters:

* IP\_MAX\_LEVEL\_ALLOWED: once a pt reaches this level it does not perform more compensations and it remains with the pair (b1n, s1n), where n is the maximum level defined.
* IP\_MAX\_PT\_CREATED: it establishes the maximum number of perfect trades created during the session.

In perfect trade added the following parameter:

* PT\_MP\_COMPENSATION\_SHIFT: this factor shifts the mp value to the “internal side” to favour the trading of the furthest new (b1, s1) pair. In the case of SIDE-TOP, this is a left-shift and in the case of SIDE-BOTTOM a right-shift.

In perfect trade added the following property and public getters:

* \_trades\_history: a list containing all the orders traded in this perfect trade object. They are sorted according to trade-time.
* getTradesStr(): to allow the logging/printing of the historical orders for this perfect trade.
* getLastOrderTradedName():

04/04/2021

## TODO #04 (pending)

Save pending orders to close perfect trades in a file, no matter what status they have (MONITOR or ACTIVE).

This is important when the app is closed, intentionally or not, to know the pending orders to compensate somehow.

## TODO #05

Fixed getting and setting the commission once the order is traded.

To do that we have to get first the **binance order id** for this particular order.

Processed in the trade manager **tm\_order\_traded()** method.

## TODO #06 (pending)

Create an orders list called **desired\_pending\_orders** to assess compensations at session level, rather than at perfect trade level.

05/04/2021

## TODO #07

Once reached the maximum allowed level, since there will not be a compensation, both orders of the first pair (b1, s1) have to be placed at creation of the new level and changed its status to ACTIVE.

To simplify the implementation, instead of placing the order when the new level is created, what it has been done is setting activation values of 0 and 1\_000\_000.00.

# Annex I

## Virtual environment

Create a new folder with the name of the project, create the virtual environment and activate it

1. > mkdir Polaris
2. > cd Polaris
3. > python3 -m venv .venv
4. > source .venv/bin/actívate
5. …
6. > deactivate

Copy into the new folder the file **requirements,txt** and install all packages:

1. *(.venv) crypto\_apps/dawning\_0.0.1*> cp requirements.txt ../Polaris
2. *(.venv) crypto\_apps/dawning\_0.0.1*> cd ../Polaris
3. *(.venv) crypto\_apps/Polaris*> pip install -r requirements.txt

To make a copy of the project in the new folder:

* From github download the .zip of the required project branch.
* Unzip the .zip into the new folder.
* Open atom and set this project as the project folder.
* Create new repository and publish to github.

Useful commands:

1. > pip freeze > requirements.txt
2. > pip list –outdated
3. > pip install –upgrade pip

Extra feature:

* In OS X Finder, **CMD + SHIFT + .** to toggle hidden files visibility

# Annex II

## PyCharm

Shortcuts:

* Open keymap: [CMD] + [,]

Setup log files visualization:

* Under Run>Edit configurations -> Logs, add the log file Polaris.log and an alias.
* This alias will appear in the running/debugging window when running from PyCharm (not if it is run from terminal).
* There is the option to select the log level live.

# Annex III

## SQLITE

## SQLite and Python types

### **Introduction**

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem:

| **Python type** | **SQLite type** |
| --- | --- |
| [None](https://docs.python.org/3/library/constants.html#None) | NULL |
| [int](https://docs.python.org/3/library/functions.html#int) | INTEGER |
| [float](https://docs.python.org/3/library/functions.html#float) | REAL |
| [str](https://docs.python.org/3/library/stdtypes.html#str) | TEXT |
| [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) | BLOB |

This is how SQLite types are converted to Python types by default:

| **SQLite type** | **Python type** |
| --- | --- |
| NULL | [None](https://docs.python.org/3/library/constants.html#None) |
| INTEGER | [int](https://docs.python.org/3/library/functions.html#int) |
| REAL | [float](https://docs.python.org/3/library/functions.html#float) |
| TEXT | depends on [text\_factory](https://docs.python.org/3/library/sqlite3.html" \l "sqlite3.Connection.text_factory" \o "sqlite3.Connection.text_factory), [str](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "str) by default |
| BLOB | [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) |

The type system of the [sqlite3](https://docs.python.org/3/library/sqlite3.html" \l "module-sqlite3" \o "sqlite3: A DB-API 2.0 implementation using SQLite 3.x.) module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the [sqlite3](https://docs.python.org/3/library/sqlite3.html" \l "module-sqlite3" \o "sqlite3: A DB-API 2.0 implementation using SQLite 3.x.) module convert SQLite types to different Python types via converters.

### **Using adapters to store additional Python types in SQLite databases**

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must **adapt** them to one of the sqlite3 module’s supported types for SQLite: one of NoneType, int, float, str, bytes.

There are two ways to enable the [sqlite3](https://docs.python.org/3/library/sqlite3.html" \l "module-sqlite3" \o "sqlite3: A DB-API 2.0 implementation using SQLite 3.x.) module to adapt a custom Python type to one of the supported ones.

#### **Letting your object adapt itself**

This is a good approach if you write the class yourself. Let’s suppose you have a class like this:

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

Now you want to store the point in a single SQLite column. First you’ll have to choose one of the supported types to be used for representing the point. Let’s just use str and separate the coordinates using a semicolon. Then you need to give your class a method \_\_conform\_\_(self, protocol) which must return the converted value. The parameter protocol will be PrepareProtocol.

**import** **sqlite3**

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

**def** \_\_conform\_\_(self, protocol):

**if** protocol **is** sqlite3.PrepareProtocol:

**return** "*%f*;*%f*" % (self.x, self.y)

con = sqlite3.connect(":memory:")

cur = con.cursor()

p = Point(4.0, -3.2)

cur.execute("select ?", (p,))

print(cur.fetchone()[0])

con.close()

#### **Registering an adapter callable**

The other possibility is to create a function that converts the type to the string representation and register the function with [register\_adapter()](https://docs.python.org/3/library/sqlite3.html" \l "sqlite3.register_adapter" \o "sqlite3.register_adapter).

**import** **sqlite3**

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

**def** adapt\_point(point):

**return** "*%f*;*%f*" % (point.x, point.y)

sqlite3.register\_adapter(Point, adapt\_point)

con = sqlite3.connect(":memory:")

cur = con.cursor()

p = Point(4.0, -3.2)

cur.execute("select ?", (p,))

print(cur.fetchone()[0])

con.close()

The [sqlite3](https://docs.python.org/3/library/sqlite3.html" \l "module-sqlite3" \o "sqlite3: A DB-API 2.0 implementation using SQLite 3.x.) module has two default adapters for Python’s built-in [datetime.date](https://docs.python.org/3/library/datetime.html" \l "datetime.date" \o "datetime.date) and [datetime.datetime](https://docs.python.org/3/library/datetime.html" \l "datetime.datetime" \o "datetime.datetime) types. Now let’s suppose we want to store [datetime.datetime](https://docs.python.org/3/library/datetime.html" \l "datetime.datetime" \o "datetime.datetime) objects not in ISO representation, but as a Unix timestamp.

**import** **sqlite3**

**import** **datetime**

**import** **time**

**def** adapt\_datetime(ts):

**return** time.mktime(ts.timetuple())

sqlite3.register\_adapter(datetime.datetime, adapt\_datetime)

con = sqlite3.connect(":memory:")

cur = con.cursor()

now = datetime.datetime.now()

cur.execute("select ?", (now,))

print(cur.fetchone()[0])

con.close()

### **Converting SQLite values to custom Python types**

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let’s go back to the Point class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we’ll define a converter function that accepts the string as a parameter and constructs a Point object from it.

**Note**

Converter functions **always** get called with a [bytes](https://docs.python.org/3/library/stdtypes.html" \l "bytes" \o "bytes) object, no matter under which data type you sent the value to SQLite.

**def** convert\_point(s):

x, y = map(float, s.split(b";"))

**return** Point(x, y)

Now you need to make the [sqlite3](https://docs.python.org/3/library/sqlite3.html" \l "module-sqlite3" \o "sqlite3: A DB-API 2.0 implementation using SQLite 3.x.) module know that what you select from the database is actually a point. There are two ways of doing this:

* Implicitly via the declared type
* Explicitly via the column name

Both ways are described in section [Module functions and constants](https://docs.python.org/3/library/sqlite3.html" \l "sqlite3-module-contents), in the entries for the constants [PARSE\_DECLTYPES](https://docs.python.org/3/library/sqlite3.html" \l "sqlite3.PARSE_DECLTYPES" \o "sqlite3.PARSE_DECLTYPES) and [PARSE\_COLNAMES](https://docs.python.org/3/library/sqlite3.html" \l "sqlite3.PARSE_COLNAMES" \o "sqlite3.PARSE_COLNAMES).

The following example illustrates both approaches.

**import** **sqlite3**

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

**def** \_\_repr\_\_(self):

**return** "(*%f*;*%f*)" % (self.x, self.y)

**def** adapt\_point(point):

**return** ("*%f*;*%f*" % (point.x, point.y)).encode('ascii')

**def** convert\_point(s):

x, y = list(map(float, s.split(b";")))

**return** Point(x, y)

*# Register the adapter*

sqlite3.register\_adapter(Point, adapt\_point)

*# Register the converter*

sqlite3.register\_converter("point", convert\_point)

p = Point(4.0, -3.2)

*#########################*

*# 1) Using declared types*

con = sqlite3.connect(":memory:", detect\_types=sqlite3.PARSE\_DECLTYPES)

cur = con.cursor()

cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))

cur.execute("select p from test")

print("with declared types:", cur.fetchone()[0])

cur.close()

con.close()

*#######################*

*# 1) Using column names*

con = sqlite3.connect(":memory:", detect\_types=sqlite3.PARSE\_COLNAMES)

cur = con.cursor()

cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))

cur.execute('select p as "p [point]" from test')

print("with column names:", cur.fetchone()[0])

cur.close()

con.close()

### **Default adapters and converters**

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name “date” for [datetime.date](https://docs.python.org/3/library/datetime.html" \l "datetime.date" \o "datetime.date) and under the name “timestamp” for [datetime.datetime](https://docs.python.org/3/library/datetime.html" \l "datetime.datetime" \o "datetime.datetime).

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

**import** **sqlite3**

**import** **datetime**

con = sqlite3.connect(":memory:", detect\_types=sqlite3.PARSE\_DECLTYPES|sqlite3.PARSE\_COLNAMES)

cur = con.cursor()

cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()

now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))

cur.execute("select d, ts from test")

row = cur.fetchone()

print(today, "=>", row[0], type(row[0]))

print(now, "=>", row[1], type(row[1]))

cur.execute('select current\_date as "d [date]", current\_timestamp as "ts [timestamp]"')

row = cur.fetchone()

print("current\_date", row[0], type(row[0]))

print("current\_timestamp", row[1], type(row[1]))

con.close()

If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.

# Annex IV

## Logging

A picture containing text, newspaper, document

Description automatically generated

# Annex V

## Digital Ocean

Account:

Log in with google

* Ip: 167.71.62.77
* Usr: root
* Pwd: digi\*\*\*\*

1. \*\*\*\*\*\*\*\*\*\* Connect to Digital Ocean as root \*\*\*\*\*\*\*\*\*\*
2. $ ssh root@167.71.62.77
3. $ password:
5. root@droplet-002:~#
6. apt-get update
7. apt-get upgrade
8. \*\*\*\*\*\*\*\*\*\* Create user (first time) \*\*\*\*\*\*\*\*\*\*
9. # create user (pwd: pola\*\*\*\*)
10. adduser xavi
11. # add user to sudo
12. adduser xavi sudo
13. Exit
14. \*\*\*\*\*\*\*\*\*\* Connect to Digital Ocean as xavi \*\*\*\*\*\*\*\*\*\*
15. # connect with new user
16. ssh [xavi@167.71.62.77](mailto:xavi@167.71.62.77)
17. \*\*\*\*\*\*\*\*\*\* First time setup \*\*\*\*\*\*\*\*\*\*\*
18. # check Python versión
19. python3 –version
20. Python 3.8.5
21. # update to 3.9
22. sudo apt install python3.9
23. # install pip
24. sudo apt install pyton3-pip
25. # create Project directory
26. mkdir polaris\_plus\_project
28. # create virtual environment
29. cd polaris\_plus\_project
30. python3 -m venv .venv
31. # activate venv
32. source .venv/bin/actívate
33. # upload Project fron github
34. (.venv) git clone <https://github.com/xavibenavent/polaris_plus.git>
35. ls
36. cd polaris\_plus
37. # install packages needed
38. (.venv) pip install -r requirements.txt
39. # create required directories not in git
40. cd src
41. mkdir log
42. mkdir database
43. \*\*\*\*\*\*\*\*\*\* Run the app (test) \*\*\*\*\*\*\*\*\*\*
44. # run the app from the working directory
45. (.venv) python src/main.py --client\_mode=simulated --new\_master\_session=True

Setup the gunicorn server (WGSI), that will substitute the flask server

1. pip install gunicorn
2. # modify the app
3. # 1.run from cli and launch the dash module: pp\_dash.py
4. # 2.after app = dash.Dash(\_\_name\_\_, …) set the variable used in the cli
5. # server = app.server
6. # 3. Run
7. gunicorn src.pp\_dash:server

Setup the web server (nginx)

1. # install nginx
2. sudo apt install nginx

“Let's start by installing it. From the command line, while logged in to your server, run the following command:

sudo apt install nginx

We now want to create a configuration file for our app. Installing nginx does several things, one of which is creating a sites-enabled folder. We want to create our configuration file there, with basic options. We can use any text editor for that; a simple one to use that you can usually find on Linux machines is the nano editor. Running it as a command followed by a filename opens that file for editing (or creates one if it doesn't exist).

From the command line, run the following command to open and edit our file:

sudo nano /etc/nginx/sites-enabled/dash\_app

You should get an empty file, and you can copy and paste the following code, but make sure to replace the IP address after server\_name with your own IP address:

server {

    listen 80;

    server\_name 172.105.72.121;

    location / {

        proxy\_pass http://127.0.0.1:8000;

        proxy\_set\_header Host $host;

        proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

    }

  }

”

Excerpt From: Elias Dabbas. “Interactive Dashboards and Data Apps with  Plotly and Dash”. Apple Books.

“This code contains configuration for the server context, as you can see. It tells it to listen on port 80, which is the default port for web servers. It also defines the server\_name as the IP address. Later on, you can use this to define your own domain name.

It then defines the behavior of the server for location / under another block. The most important thing for us is that we are making nginx a proxy server with the proxy\_pass directive and telling it to listen to the URL and port that Gunicorn is listening to. So, now, the cycle should be complete. Our web server will be sending and receiving requests and responses through the correct URL and port, where the interface with our Python code is handled by Gunicorn.

Installing nginx creates a default configuration file, which we need to unlink with the following command:

sudo unlink /etc/nginx/sites-enabled/default

We just need to reload nginx after making this change. Keep this in mind when you make any changes in the future. You should reload nginx for any changes to take place, which you can do with the following command:

sudo nginx -s reload

Now[…]”

Excerpt From: Elias Dabbas. “Interactive Dashboards and Data Apps with  Plotly and Dash”. Apple Books.