

Cryptocurrency Orderbook Analysis Tool

Ivan E. Perez

February 16, 2021
v1.0.2

Contents

1	Introduction	2
2	crobot Features	3
2.1	Orderbook updating	3
3	Getting Started and Installation	4
3.1	CoPrA	4
3.2	Installation	4
4	Time-series Data Structure Layout	5
4.1	Introduction	5
4.2	Understanding The Raw Order Book Data	5
4.2.1	Order Book Snapshots	5
4.2.2	Signed Order Book	6
4.3	Final notes on Orderbook interpretation	7
5	Context: Where does crobat operate?	8
6	Modules	10
6.1	recorder_full	10
6.1.1	Description	10
6.1.2	Class <code>L2_Update</code>	10
6.2	LOBF_funcs	15
6.2.1	class <code>history()</code>	15
6.3	history_funcs.py	31
7	Credits	38
7.1	Development Lead	38
7.2	Contributors	38
8	License	39
9	Appendix	40
10	To do list	40

1 Introduction

This project is an extension of my thesis, ja A Study of CUSUM Statistics on Bitcoin Transactions, where I was tasked with implementing CUSUM statistic processes to identify price actions periods in bitcoin markets. After developing a tool for market orders, the natural extension was to find relationships from activities in the limit order book. I started developing this tool to record instances of the limit order book in order to record Limit Order insertions (LO), cancellations (CO), and Market Orders (MO).

As the project grew I wanted to make a tool that could be used by academics looking to apply and develop market microstructure models in live markets. As a result, the styles in which the limit orderbook and orderbook events are recorded are being developed in accordance to the conventions presented in recent market microstructure papers correspond to the following papers:

1. Huang W., Lehalle C.A. and Rosenbaum M. - Simulating and analyzing order book data:The queue-reactive model[1]
2. Cont R., Stoikov S. and Talreja R. - A stochastic model for order book dynamics
3. Cont R., Kukanov A. and Stoikov S. - The price impact of order book events
4. Cartea. A, Jaimungal S. and Wang Y. - Spoofing and Price Manipulation in Order Driven Markets
5. Silyantev, E. - Order flow analysis of cryptocurrency markets¹

¹This paper shows a working model implementing Order Flow Imbalance (OFI) and Trade Flow Imbalance(TFI) to BTC-USD trades was done by Ed Silyantev. He developed a tool to assess OFI and TFI of XBT-USD pair.

2 crobat Features

2.1 Orderbook updating

- Order book recording module that maintains a time-series of:
 1. Limit order insertions, cancellations and market orders,
 2. volumes at n^{th} best limits, and
 3. prices at n^{th} best limits.

3 Getting Started and Installation

3.1 CoPrA

3.2 Installation

Given that this is still very much a work in progress, it may make more sense to fork the project, or download the project as a compressed folder, and build `CSV_out_test.py` with your preferred settings.

Note: depending on the popularity of the asset and the computational power of your PC, you may run into errors arising from the computer not being able to keep up with the market (especially BTC-USD). I would suggest experimenting with an unpopular pair, (e.g., XRP-USD), or a crypto-crypto pair (e.g., XRP-BTC), and timing your queries outside of NYSE, and London Stock Exchange trading hours as they tend to have less activity.

however if you want an easy installation:

“pip3 install crobat“

4 Time-series Data Structure Layout

4.1 Introduction

Since this is an orderbook `ju recorder / u`, my use until now has been to record the orderbook. However there are accessors in the “LOB_funcs.py” file, under in the `*history*` class. In the `/test` folder there is a small usecase if you would like to see it but documentation is pending.

1. For now we only have the full orderbook, with no regard for ticksize, and we call that “recorder_full.py”.
2. We change the “settings” variable in the “CSV_out_test.py” file that has arguments for:

Parameter	Function Arg	Type/Format	Description
Recording Duration	<code>duration</code>	<code>int</code>	recording time in seconds
Position Range	<code>position_range</code>	<code>int</code>	ordinal distance from the best bid(ask)
Currency Pair ²	<code>currency_pair</code>	<code>str</code>	

3. When you are ready, you can start the build. When it finishes you should get a message “Connection Closed” from “CoPrA”. And the files for the limit orderbook for each side should be created with a timestamp:

4.2 Understanding The Raw Order Book Data

The coinbase exchange operates using the double auction model, the Coinbase Pro API, and by extension the CoPrA API makes it relatively easy to get still images of an instance of the orderbook as **snapshots** and it sends updates in real time of the volume at a particular price level as **12.update** messages. If you would like to know more, the cited papers do a great job introducing the double auction model for the purposes of defining the types of orders, and how they record events and make sense of them.

4.2.1 Order Book Snapshots

Below there is a graph of the snapshot where bids (green) show open limit orders to buy the 1 unit of the cryptocurrency below \$7085.930, and asks (red) show open limit orders to buy 1 unit above \$7085.930. The x-axis shows the price points, and the y-axis is the aggregate size at the price level. Note that the signed order book calls volume on the bid side negative.

Early and current works relied on exchanges and private data providers (e.g., NASDAQ - BookViewer,LOBSTER to provide reconstructions of order books. Earlier works were limited to taking snapshots and inferring the possible sequence of order book events between states. Coinbase and by extension crobat update the levels on the instance of a update message from the exchange so there is no guess as to what happened between states of the order book. The current format of the order book snapshot is not aggregated. The format of the order book snapshot for a single side is shown below

Item	Description/Format
Timestamp	YYYY-MM-DDTHH:MM:SS.fffff
1	total BTC at position 1
2	total BTC at position 1
...	...
<i>n</i>	total BTC at position <i>n</i>

Incl. sample output of an entry

The associated price quote (price quote (USD per XTC))snapshot is also generated, to make generation of market depth feasible.

Item	Description/Format
Timestamp	YYYY-MM-DDTHH:MM:SS.fffff
1	price quote at position 1
2	price quote at position 1
...	...
<i>n</i>	price quote at position <i>n</i>

Event recording are a timeseries of MO, LO, CO's as afforded from the **12.update** messages which are used to update the price, volume pair size at each price level. The format of the Event recorder is as follows:

Item	Description	format
Timestamp	Timestamp of when the event occurred	YYYY-MM-DDTHH:MM:SS.fffff
order type	MO, LO, CO	$\text{str} \in \{\text{'market'}, \text{'limit'}, \text{'cancellation'}\}$
price level	price of event occurrence in quote currency	float64
event size	size of event in base currency ³	float64
position	signed position (– for bids, + for asks)	int
mid price	(best-ask + best-bid)/2	float64
spread	best-ask + best-bid	float64

4.2.2 Signed Order Book

4.2.2.1 Signed Order Book Snapshot Prices The signed orderbook takes a different approach to position labelling so please keep that in mind. (note: I should shift the position index to start at 1, for single side order book snapshot time series). The signed orderbook snapshot is generated in a similar fashion with a volume, and price at each position. However, it uses the convention established in [3] for the signed order book. where positions on the bid are negative, with negative volume (XTC). I'll show the default setting that displays the 5 best bids and asks on each side.

Item	Description/Format
Timestamp	YYYY-MM-DDTHH:MM:SS.fffff
$-n$	price quote at the n^{th} best bid (i.e., worst bid)
$-n + 1$	aggregate XTC limit buys at the second to worst bid
...	...
-2	aggregate XTC being bid for at the 2^{nd} best bid
-1	aggregate XTC limit being bid for at the best bid
1	aggregate XTC offered at the best ask
2	aggregate XTC offered at the 2^{nd} best ask
...	...
$n - 1$	aggregate XTC offered at the second worst ask
n	aggregate XTC offered at the worst ask

Similar to the single side implementation, there is an associated price quote (e.g., USD per XTC) snapshot generated at each timepoint. The default format is given below:

Item	Description/Format
Timestamp	YYYY-MM-DDTHH:MM:SS.fffff
$-n$	price quote at the n^{th} best bid (i.e., worst bid)
$-n + 1$	price quote at the second to worst bid
...	...
-2	price quote at the 2^{nd} best bid
-1	price quote at the best bid
1	price quote at the best ask
2	price quote at the 2^{nd} best ask
...	...
$n - 1$	price quote at the second worst ask
n	price quote at the worst ask

4.2.2.2 Signed Events Signed event recordings follow the convention from The Price impact of Orderbook events, where positive order flow is due to MO's on the buy side, CO on the sell side, and LO on the buy side. Conversely, negative order flow is due to MO's on the sell side, CO on the buy side, and LO on the buy side. The format is similar to the single side order book events time-series, but the order volume is signed based on the aforementioned construction.

Item	Description	format
Timestamp	Timestamp of when the event occurred	YYYY-MM-DDTHH:MM:SS.fffff
order type	MO, LO, CO	$\text{str} \in \{\text{'market'}, \text{'limit'}, \text{'cancellation'}\}$
price level	price of event occurrence in quote currency	float64
event size	size of event in base currency ⁴	float64
position	signed position (– for bids, + for asks)	int
side	bid/ask side where the event occurs	$\text{str} \in \{\text{'buy'}, \text{'sell'}\}$
mid price	(best-ask + best-bid)/2	float64
spread	best-ask + best-bid	float64

4.3 Final notes on Orderbook interpretation

write about how the program object sees the orderbook with an example of code.

Start

5 Context: Where does crobat operate?

Following CoPrA's example as to how to set up a web socket connectin we provide context as to where crobat operates. Lets begin with copras heartbeat.py example:

```

1 import asyncio
2
3 from copra.websocket import Channel, Client
4
5 loop = asyncio.get_event_loop()
6
7 ws = Client(loop, Channel('heartbeat', 'BTC-USD'))
8
9 try:
10 loop.run_forever()
11 except KeyboardInterrupt:
12 loop.run_until_complete(ws.close())
13 loop.close()

```

If we were to run it we would execute methods in the instance of the class Client based on the arrival of messages on the heartbeat channel for the pair BTC-USD. crobat primarily overwrites this Client class and inserts its new methods. Below we introduce our own version of this loop.

```

1 import recorder_full as rec
2 import asyncio, time
3 from datetime import datetime
4 import copra.rest
5 from copra.websocket import Channel, Client
6
7 class input_args(object):
8     def __init__(self, currency_pair='ETH-USD', position_range=5, recording_duration=5):
9         self.currency_pair = currency_pair
10        self.position_range = position_range
11        self.recording_duration = recording_duration
12
13 def main():
14     settings_1 = input_args()
15
16     loop = asyncio.get_event_loop()
17
18     channel1 = Channel('level2', settings_1.currency_pair)
19     channel2 = Channel('ticker', settings_1.currency_pair)
20
21     ws_1 = rec.L2_Update(loop, channel1, settings_1)
22     ws_1.subscribe(channel2)
23
24     timestart=datetime.utcnow()
25
26     try:
27         loop.run_forever()
28     except KeyboardInterrupt:
29         loop.run_until_complete(ws_1.close())
30         loop.close()
31
32 if __name__ == '__main__':
33     main()

```

In this modified event loop we have the context where the bulk of crobat sits, the class l2.update. we can the loop object that is passed to ws_1, the instance of the class L2_Update, receives msg objects through the async methods inherited from Client. Here we give an example of what L2_Update looks like:


```

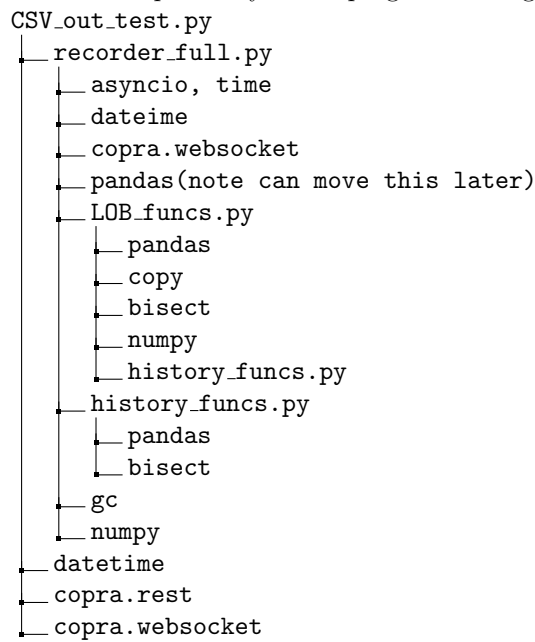
1 import asyncio, time
2 from datetime import datetime
3 import copra.rest
4 from copra.websocket import Channel, Client
5
6 class L2_Update(Client):
7     def __init__(self, loop, channel, input_args):
8         self.time_now = datetime.utcnow() #initial start time
9         self.position_range = input_args.position_range
10        self.recording_duration = input_args.recording_duration
11        self.snap_received = False
12        super().__init__(loop, channel)
13
14    def on_open(self):
15        print("Let's count the L2 messages!", self.time_now)
16        super().on_open() # inheriting things from the parent class who really knows
17
18    def on_message(self, msg):
19        if msg['type'] in ['snapshot']:
20            print("received the snapshot")
21            time = self.time_now
22            self.snap_received = True
23            if msg['type'] in ['ticker']:
24                print("received ticker message")
25                if self.snap_received:
26                    # Do Something Here
27            else:
28                print("market order arrived but no snapshot received yet")
29
30            if msg['type'] in ['l2update']:
31                print("received an l2update message")
32            else:
33                print("unknown message")
34
35            if (datetime.utcnow() - self.time_now).total_seconds() > self.recording_duration:
36                self.loop.create_task(self.close())
37
38    def on_close(self, was_clean, code, reason):
39        print("Connection to server is closed")
40        print(was_clean)
41        print(code)
42        print(reason)
43
44    def main():
45        pass
46
47 if __name__ == '__main__':
48     main()

```

Here we can see that we can create an instance of this class, and on its initialization we can create light weight arrays that can hold our changes and order book states. the logic tree for when each type of message arrives can dictate what kinds of records, and changes to the snapshot are made the arrival of a message. In this current version, the history module contains methods to both update the order book and record changes, but in future versions the processes should be separated as other may have better ideas as to what to do with order book changes.

6 Modules

The module dependency of the program are organized as follows:



6.1 recorder_full

6.1.1 Description

The `recorder_full` module contains the class `L2_Update`. `L2_Update` is responsible for:

- initializing instances of the order book history class, `history`.
- interpreting the snapshot, ticker and l2update messages that come from the websocket.
- calling the appropriate functions and classes to carry out the orderly update to the limit order book, and order book history arrays.

6.1.2 Class `L2_Update`

Function list:

- `__init__(self, loop, channel, input_args)`
- `on_open(self)`
- `on_message(self, msg)`
- `on_close(self, was_clean, code, reason)`
- `__init__(self, loop, channel, input_args)`

Description: Inheriting attributes `loop`, and `channel`, from `Client`, it

1. initializes the class `history` imported from `LOB_funcs` and
2. passes on the settings from the class `input_args` and
3. uses functions `on_message(self, msg)`,
4. `on_close(self, was_clean, code, reason)` to manage incoming messages.

6.1.2.1 function `__init__(self, loop, channel, input_args)`

Description: Initializes the class, using the attributes `loop`, `channel` from `Client` and attributes `position_range`, `recording_duration` from the class `input_args`. It also creates an instance of the class `history`.

parameters: loop: object
 Comes from CoPrA
 channel: object
 comes from CoPrA
 input_args: class
 passes on arguments for recording duration, and position range

returns: None

Example: None

6.1.2.2 *function* `on_open(self)`

Description: From class `Client` uses method `on_open()` to set things up. See ACoPrA `on_open()` for more information.

restated from CoPrA:

`on_open` is called as soon as the initial WebSocket opening handshake is complete. The connection is open, but the client is not yet subscribed.

If you override this method it is important that you still call it from your subclass' `on_open` method, since the parent method sends the initial subscription request to the WebSocket server. Somewhere in your `on_open` method you should have `super().on_open()`.

In addition to sending the subscription request, this method also logs that the connection was opened.

parameters: None: None

returns: None: None

Example: None

6.1.2.3 function `on_message`(self, msg),

Description: After matching `msg['type']` to 'snapshot', 'ticker', 'l2update' do one of three actions,

<code>msg['type'] ==</code>	action
'snapshot'	initialize the limit order book using the <code>initialize_snap_events</code>
'ticker'	parse a market order,
'l2update'	parse a limit order insertion or cancellation to the limit order book.

restated from CoPrA:

`on_message` is called every time a message is received. message is a dict representing the message. Its content will depend on the type of message, the channels subscribed to, etc. Please read Coinbase Pro's WebSocket API documentation to learn about these message formats.

Note that with the exception of errors, every other message triggers this method including things like subscription confirmations. Your code should be prepared to handle unexpected messages.

This default method just prints the message received. If you override this method, there is no need to call the parent method from your subclass' method.

parameters: msg: dict

returns: None: None

Example:

```

1 # Let our module be imported and our channels be assigned as follows:
2 import recorder_full as rec
3 channel1= Channel('level2', 'BTC-USD')
4 channel2= Channel('ticker', 'BTC-USD')
5
6
7 #Let the instance of the class L2_Update be ws_1
8 ws_1 = rec.L2_Update(loop, channel1, settings_1)
9 ws_1.subscribe(channel2)
10
11 #Let the message received from the websocket be:
12 msg = {
13     "type": "l2update",
14     "product_id": "BTC-USD",
15     "time": "2019-08-14T20:42:27.265Z",
16     "changes": [
17         [
18             "buy",
19             "10101.80000000",
20             "0.162567"
21         ]
22     ]
23 }
24
25 # let the current state of the orderbook be defined as follows:
26 ws_1.orderbook_instance.snapshot_bid = [
27     [10101.00, 5.23], [10101.50, 1.11], [10101.80, 0.5]
28 ]
29
30 ws_1.orderbook_instance.snapshot_signed = [
31     [10101.00, -5.23], [10101.50, -1.11], [10101.80, -0.5],
32     [10101.90, 0.4], [10102.00, 1.3], [10102.10, 5.00]
33 ]
34
35 ws_1.orderbook_instance.bid_events = []
36 ws_1.orderbook_instance.signed_events = []
37 #Suppose on loop you this message was passed to on_message(msg)
38 on_message(msg)
39
40 # the new values for the the snapshots, and events would be:
41 >>print(ws_1.orderbook_instance.snapshot_bid)
42 >>[[10101.00, 5.23], [10101.50, 1.11], [10101.80, 0.162567]]
43
44 >>print(ws_1.orderbook_instance.snapshot_signed)
45 >>[[10101.00, -5.23], [10101.50, -1.11], [10101.80, -0.162567], [10101.90, 0.4],
46     [10102.00, 1.3], [10102.10, 5.00]]
47
48 >>print(ws_1.orderbook_instance.bid_events)
49 >>[[2019-08-14T20:42:27.265Z, 10101.80, cancellation, 0.337433, 1, 10101.85, 0.10]]
50
51 >>print(ws_1.orderbook_instance.signed_events)
52 >>[[2019-08-14T20:42:27.265Z, 10101.80, cancellation, 0.337433, -1, 10101.85, 0.10]]

```

6.1.2.4 function `on_close`(self, was_clean, code, reason)

description: The function called when the `close()` task is executed. Begins the post processing of accumulated time-series using the functions:

- `hf.convert_array_to_list_dict`
- `hf.convert_array_to_list_dict_sob`
- `hf.pd_excel.save`

and `gc.collect()` to clear memory after storing `.xlsx` files of the processed data.

restated from CoPrA:

`on_close` is called whenever the connection between the client and server is closed. `was_clean` is a boolean indicating whether or not the connection was cleanly closed. `code`, an integer, and `reason`, a string, are sent by the end that initiated closing the connection.

If the client did not initiate this closure and `client.auto_reconnect` is set to `True`, the client will attempt to reconnect to the server and resubscribe to the channels it was subscribed to when the connection was closed. This method also logs the closure.

If your subclass overrides this method, it is important that the subclass method calls the parent method if you want to preserve the auto reconnect functionality. This can be done by including `super().on_close(was_clean, code, reason)` in your subclass method.

parameters: `was_clean`: bool
 `code`: int or None
 `reason`: str or None

returns: None: None

52 `# Needs an Example`

Listing 1: Python example

6.2 LOBF_funcs

Description: module that houses the class `history` and the methods that outline how to update snapshots.

6.2.1 `class history()`

Description: Contains attributes for the time series array of the limit order book, the current snapshot of the limit order book and the time-series of the events in the limit order book. We detail the role of each attribute in the Table whatever. The class contains methods that operate of these attributes, in order to update the order book and record changes. The functions `UpdateSnapshot_bid Seq` and `UpdateSnapshot_ask Seq` found outside of these methods, dictate the sequence in which the methods contained in `history` are executed.

6.2.1.1 `function __init__(self)`

Description: Initializes the lists and variables that will be operated on by other class methods. Each side, bid, ask and the combined signed has a list demarked by `_history`, that will be populated by timestamped states of the limit order book. The lists prefixed by `snapshot` are populated by the state of limit order book demarked by the suffix `bid`, `ask`, and `signed`. The variables `order_type`, `position` and `event_size` denote the type, position and size of the event based on the change in the order book as a result of an `l2_update` message. The `token` variable determines whether the change in the order book is recorded in the events list.

parameters: None: None

returns: None: None

Example: None

6.2.1.2 *function initialize_snap_events*(self, msg, time)

Description: Method that parses the msg payload if msg['type'] == snapshot . Adds the first entry to the collection of _history variables. Uses np.round and hf.min_dec to calculate the smallest amount of the quote currency is needed to make the ticks 0.01 units of the base currency.⁵

function dependency:

- np.round
- hf.min_dec

parameters: msg: dict
time: datetime object

returns: None: None

Example:

```

53 # let the instance of our L2_Update class be defined as follows:
54 channel1= Channel('level2', 'BTC-USD')
55 ws_1 = L2_Update(loop, channel1, settings_1)
56
57 # from our __init__ method, we initialized the arrays in history:
58 >>print(vars(ws_1.hist))
59 >>{bid_history:[],ask_history:[], signed_history:[], snpashot_bid:[],snapshot_ask:[],
    snapshot_signed:[], bid_events:[], ask_events:[], signed_events:[], order_type:None,
    token:False, position:0, event_size:0}
60
61 #after receiving the snapshot from our event loop we will run initialize_snap_events
62 #to populate the variables created in __init__.
63
64 #Let our message be
65 ws_1.time_now = "2019-08-14T20:42:27.265Z"
66 msg = {
67     "type": "snapshot",
68     "product_id": "BTC-USD",
69     "bids": [["10101.10", "0.450541"],["10101.20", "0.44100"], ["10101.55", "0.013400"]],
70     "asks": [["10102.55", "0.577535"],["10102.58", "0.63219"], ["10102.60", "0.803200"]]
71 }
72
73 #note there is not a timestamp for the snapshot that we receive from coinbase so it is
    passed from the datetime.utcnow() from the init method of the L2_Update class.
74
75 ws_1.hist.initialize_snapevents(msg, ws_1.time_now)
76
77 >>print(vars(ws_1.hist))
78 >>{bid_history:[[2019-08-14T20:42:27.265Z, [[10101.10, 0.450541], [10101.20, 0.44100],
    [10101.55, 0.013400]]]],
79    ask_history:[[2019-08-14T20:42:27.265Z, [[10102.55, 0.577535], [10102.58, 0.63219],
    [10102.60, 0.803200]]]],
80    signed_history:[2019-08-14T20:42:27.265Z, [[10101.10, -0.450541], [10101.20,
    -0.44100], [10101.55, -0.013400], [10102.55, 0.577535], [10102.58, 0.63219],
    [10102.60, 0.803200]]]],
81    snapshot_bid:[[10101.10, 0.450541],[10101.20, 0.44100], [10101.55, 0.013400]],
82    snapshot_ask:[[10102.55, 0.577535],[10102.58, 0.63219], [10102.60, 0.803200]],
83    snapshot_signed:[[10101.10, -0.450541],[10101.20, -0.44100], [10101.55, -0.013400],
    [10102.55, 0.577535],[10102.58, 0.63219], [10102.60, 0.803200]],
84    bid_events:[], ask_events:[], signed_events:[], order_type:None, token:False, position
    :0, event_size:0}

```

⁵For example if BTC-USD = 10101 USD/BTC, then min_dec would return the number of BTC needed for 0.01 USD in BTC and np.round truncates the volume returns to the decimal places returned by hf.min_dec. Doing so resolves the issue with artifacts from floating point arithmetic.

6.2.1.3 add_market_order_message

Description: Parses a message object, aggregates for duplicate timestamps, and appends them to the events lists.

parameters: message: array
events: array

returns: events: array

Example:

```

1 # let our events be take the form
2 # some_event = [time, order_type, float(msg['price']), size, position, mid_price, spread
3 # for 4 events we have
4
5 bid_events = [
6     [2019-08-14T20:42:27.265Z, "market", 10101.80, 0.123030, 0, 10101.85, 0.10],
7     [2019-08-14T20:42:27.560Z, "insertion", 10100.00, 0.123030, 2, 10101.85, 0.10],
8     [2019-08-14T20:42:28.123Z, "insertion", 10100.00, 0.256000, 2, 10101.85, 0.10],
9     [2019-08-14T20:42:27.560Z, "cancellation", 10100.00, 0.256000, 2, 10101.85, 0.10]
10 ]
11
12 # we receive a 'ticker' message from our event loop
13
14 msg = {
15     "type": "ticker",
16     "trade_id": 20153558,
17     "sequence": 3262786978,
18     "time": "2019-08-14T20:42:27.966Z",
19     "product_id": "BTC-USD",
20     "price": "10101.8000000",
21     "side": "sell", // Taker side
22     "last_size": "0.03000000",
23     "best_bid": "10101.8000000",
24     "best_ask": "10101.9000000",
25 }
26
27 # on message conventiently converts the message into an array that can be used by
28     add_market_order_message
29
30 message = [2019-08-14T20:42:27.966Z, "market", 10101.80, 0.030000, 0, 10101.85, 0.10]
31
32 bid_events = add_market_order_message(message, bid_events)
33
34 # we would see that the event would be appended to the list of bid events
35 >>print(bid_events)
36 >>[[2019-08-14T20:42:27.265Z, "market", 10101.80, 0.123030, 0, 10101.85, 0.10],
37 [2019-08-14T20:42:27.560Z, "insertion", 10100.00, 0.123030, 2, 10101.85, 0.10],
38 [2019-08-14T20:42:28.123Z, "insertion", 10100.00, 0.256000, 2, 10101.85, 0.10],
39 [2019-08-14T20:42:27.560Z, "cancellation", 10100.00, 0.256000, 2, 10101.85, 0.10],
40 [2019-08-14T20:42:27.966Z, "market", 10101.80, 0.030000, 0, 10101.85, 0.10]] #new msg!

```

Example 2: When two ticker messages arrive at the same time

```

1 # starting from our last entry, lets examine what would happen if another ticker message
2     containing a market order with an identical timestamp were to be received from the
3     event loop.
4
5 bid_events = [
6     [2019-08-14T20:42:27.265Z, "market", 10101.80, 0.123030, 0, 10101.85, 0.10],
7     [2019-08-14T20:42:27.560Z, "insertion", 10100.00, 0.123030, 2, 10101.85, 0.10],
8     [2019-08-14T20:42:28.123Z, "insertion", 10100.00, 0.256000, 2, 10101.85, 0.10],
9     [2019-08-14T20:42:27.560Z, "cancellation", 10100.00, 0.256000, 2, 10101.85, 0.10],
10    [2019-08-14T20:42:27.966Z, "market", 10101.80, 0.030000, 0, 10101.85, 0.10]
11 ]
12
13 # we receive a new message
14
15 msg = {
16     "type": "ticker",
17     "trade_id": 20153559,
18     "sequence": 3262786979,
19     "time": "2019-08-14T20:42:27.966Z",
20     "product_id": "BTC-USD",
21     "price": "10101.8000000",
22     "side": "sell", // Taker side
23     "last_size": "0.15000000",
24     "best_bid": "10101.8000000",
25     "best_ask": "10101.9000000",
26 }
27

```

```
24 # converting with on_message
25 message = [2019-08-14T20:42:27.966Z, "market", 10101.80, 0.150000, 0, 10101.85, 0.10]
26
27 #running bid events again
28 bid_events = add_market_order_message(message, bid_events)
29
30 # we would observe a concatenated form
31 >>print(bid_events)
32 >>[[2019-08-14T20:42:27.265Z, "market", 10101.80, 0.123030, 0, 10101.85, 0.10],
33 [2019-08-14T20:42:27.560Z, "insertion", 10100.00, 0.123030, 2, 10101.85, 0.10],
34 [2019-08-14T20:42:28.123Z, "insertion", 10100.00, 0.256000, 2, 10101.85, 0.10],
35 [2019-08-14T20:42:27.560Z, "cancellation", 10100.00, 0.256000, 2, 10101.85, 0.10],
36 [2019-08-14T20:42:27.966Z, "market", 10101.80, 0.180000, 0, 10101.85, 0.10]]
```

6.2.1.4 *function* `remove_price_level`(snap_array, level_depth, match_index)

Description: Checks the level depth of an `l2update` message, removes the existing price level from the snapshot array if the level depth is 0. Counts as a change to the limit order book therefore `self.token` is set to `True`.

parameters: `snap_array` array
 `level_depth` float64
 `match_index` int

returns: `snap_array:` array

Example:

```

1  # let our snapshot_bid be:
2  snapshot_bid = [[10101.80, 0.013400],[10101.20, 0.44100], [10101.10, 0.450541]]
3
4  # we receive a 'l2update' message from our event loop
5
6  msg = {
7      "type": "l2update",
8      "product_id": "BTC-USD",
9      "time": "2019-08-14T20:42:27.265Z",
10     "changes": [
11         [
12             "buy",
13             "10101.80000000",
14             "0.0"
15         ]
16     ]
17 }
18
19 # on_message() conveniently parses the message to create relevant local variables
20 # time, changes, side, price_level, level_depth, pre_level_depth, hist.token
21 >>print(time, side, price_level, level_depth, prelevel_depth, hist.token)
22 >>2019-08-14T20:42:27.265Z "buy" 10101.80000000 0.0 0 False
23
24 # in on_message(), the variable price_match_index is assigned by finding the index
25 # with the matching price, in the snapshot. In this case:
26 price_match_index = [0]
27
28 snapshot_bid = remove_price_level(snapshot_bid, level_depth, price_match_index)
29
30 >>print(snapshot_bid, hist.token)
31 >>[[10101.20, 0.44100], [10101.10, 0.450541]] True

```

6.2.1.5 *function* `update_level_depth`(snap_array, level_depth, match_index, pre_level_depth)

Description: updates the existing price level to a new size. Sets `self.order_type` to insertion if the new size is larger than the old size, or cancellation if the new size is smaller than the old size. sets the position to the `self.position` variable to the index where the event occurred.

parameters:

- `snap_array`: list
The snapshot array that will be modified.
- `level_depth`: float64
The new level depth that will be checked against the old level depth in the snapshot array.
- `match_index`: list
single item list that contains the index where the price in the snapshot array matches that of the `l2update` message.

returns:

- `snap_array`: list
The modified snapshot array.
- `pre_level_depth`: float64
An updated `pre_level_depth`⁶

Example:

```

1  # let our snapshot_bid be:
2  snapshot_bid = [[10101.80, 0.013400],[10101.20, 0.44100], [10101.10, 0.450541]]
3
4  # we receive a 'l2update' message from our event loop
5
6  msg = {
7      "type": "l2update",
8      "product_id": "BTC-USD",
9      "time": "2019-08-14T20:42:27.265Z",
10     "changes": [
11         [
12             "buy",
13             "10101.80000000",
14             "0.500000"
15         ]
16     ]
17 }
18
19 # on_message() conveniently parses the message to create relevant local variables
20 # time, changes, side, price_level, level_depth, pre_level_depth, hist.token
21 >>print(time, side, price_level, level_depth, prelevel_depth, hist.token)
22 >>2019-08-14T20:42:27.265Z "buy" 10101.80000000 0.500000 0 False
23
24 # in on_message(), the variable price_match_index is assigned by finding the index
25 # with the matching price, in the snapshot. In this case:
26 price_match_index = [0]
27
28 snapshot_bid, pre_level_depth = updatedate_price_level(snapshot_bid, level_depth,
29     price_match_index, pre_level_depth)
30
31 >>print(price)
32
33 >>print(snapshot_bid, hist.token)
>>[[10101.20, 0.44100], [10101.10, 0.450541]] True

```

6.2.1.6 *function* `update_price_index.buy`(self, level_depth, price_level, pre_level_depth)

Description: If there is a new price level introduced, this function determines its location and appropriately inserts it into the snapshot bid array. It will also set the position, and token depending on where the change occurs.

parameters:	level_depth:	float64	level depth received from the l2update message
	price_level:	float64	The price level being introduced
	pre_level_depth:	float64	some pre_level_depth that is reset to 0, idk this feature
returns:	snapshot_bid	list	The modified snapshot_bid.
	bid_range:	list	The modified range of bids available in the snapshot.
	pre_level_depth	float64	again the pre level depth that idk why i still have this.
latent changes:	token	boolean	bool that informs whether a significant change has occurred in the order book.
	order_type		

Example: None

```

1  # let our snapshot_bid be:
2  snapshot_bid = [[10101.80, 0.013400],[10101.20, 0.44100], [10101.10, 0.450541]]
3
4  # we receive a 'l2update' message from our event loop
5
6  msg = {
7      "type": "l2update",
8      "product_id": "BTC-USD",
9      "time": "2019-08-14T20:42:27.265Z",
10     "changes": [
11         [
12             "buy",
13             "10101.90000000",
14             "0.500000"
15         ]
16     ]
17 }
18
19 # on_message() conveniently parses the message to create relevant local variables
20 # time, changes, side, price_level, level_depth, pre_level_depth, hist.token
21 >>print(time, side, price_level, level_depth, prelevel_depth, hist.token)
22 >>2019-08-14T20:42:27.265Z "buy" 10101.90000000 0.500000 0 False
23
24 # in on_message(), the variable price_match_index is assigned by finding the index
25 # with the matching price, in the snapshot. In this case:
26 price_match_index = []
27
28 snapshot_bid, pre_level_depth = update_price_index_buy(level_depth, price_level,
29 pre_level_depth)
30
31 >>print(snapshot_bid, hist.token)
>[[[10101.90, 0.500000], [10101.80, 0.013400],[10101.20, 0.44100], [10101.10,
0.450541]] True

```

6.2.1.7 *function* `update_price_index_sell`(self, level_depth, price_level, pre_level_depth)

Description: If there is a new price level introduced, this function determines its location and appropriately inserts it into the snapshot ask array. It will also set the position, and token depending on where the change occurs.

parameters:	level_depth:	float64	level depth received from the <code>l2update</code> message
	price_level:	float64	The price level being introduced
	pre_level_depth:	float64	some pre_level_depth that is reset to 0, idk this feature
returns:	snapshot_ask:	list	The modified snapshot_ask.
	bid_range:	list	The modified range of bids available in the snapshot.
	pre_level_depth	float64	again the pre level depth that idk why i still have this.
latent changes:	token:	boolean	bool that informs whether a significant change has occurred in the order book.
	order_type:	string	type of order in ['market', 'insertion', 'cancelation']

Example:

```

1  # let our snapshot_bid be:
2  snapshot_ask = [[10101.90, 0.013400],[10102.00, 0.52100], [10102.11, 0.89041]]
3
4  # we receive a 'l2update' message from our event loop
5
6  msg = {
7      "type": "l2update",
8      "product_id": "BTC-USD",
9      "time": "2019-08-14T20:42:27.265Z",
10     "changes": [
11         [
12             "sell",
13             "10101.95000000",
14             "0.500000"
15         ]
16     ]
17 }
18
19 # on_message() conveniently parses the message to create relevant local variables
20 # time, changes, side, price_level, level_depth, pre_level_depth, hist.token
21 >>print(time, side, price_level, level_depth, prelevel_depth, hist.token)
22 >>2019-08-14T20:42:27.265Z "buy" 10101.90000000 0.500000 0 False
23
24 # in on_message(), the variable price_match_index is assigned by finding the index
25 # with the matching price, in the snapshot. In this case:
26 price_match_index = []
27
28 snapshot_ask, pre_level_depth = update_price_index_sell(level_depth, price_level,
29 pre_level_depth)
30
31 >>print(snapshot_bid, hist.token)
> [[10101.90, 0.013400],[10101.95, 0.500000],[10102.00, 0.52100], [10102.11, 0.89041]]
True

```

6.2.1.8 *function* `update_snapshot_bid`(self)

Description: Updates the variable `bid_range`. I'm not sure why I need to do this but it does keep update snapshot bid/ask from making mistakes when doing bisect/insert.

parameters: None: None
 None

returns: snapshot_bid: list
 The snapshot on the bid side.
 bid_range: list
 ordered range of prices for the bid side

latent changes: None None
 None

Example: None

```

1  x = __my_function__(1.234, True, 3)
2  >>print(x)
3  >>-1.23
4
5  x = __my_function__(1.234, False, 3)
6  >>print(x)
7  >>1.23

```

6.2.1.9 *function* `update_snapshot_ask(self)`

Description: Updates the variable `ask_range`. I'm not sure why I need to do this but it does keep update snapshot bid/ask from making mistakes when doing bisect/insert.

parameters: None: None
 None
returns: snapshot_ask: list
 The snapshot on the ask side.
 ask_range: list
 ordered range of prices for the ask side
latent changes: None None
 None

Example: None

1 ??????????

6.2.1.10 *function* `trim.coordinator`(self, position, bound)

Description: Sets the tokenvariable to False if the change to the orderbook was outside of thte position bound.

parameters: position: int
 position where the order book was updated.
 bound: int
 the largest position where the function can trigger

returns: token boolean
 The boolean argument that determines whether the change in the order book is recorded.

latent changes: None None.

Example: None

1 ???????

6.2.1.11 *function* `append_snapshot_bid`(self, time, price_level)

Description: Appends the current form of the snapshot to `bid_history` and computes the event and appends the event record to `bid_events`.

parameters:

	:	float64	some input that will be passed to the function
param_2:		boolean	a conditional that will be checked by the function
param_3:		int	some int that is needed for the function

returns: trans_number float64
The number with the correct significant digits and sign.

latent changes: trans_number float64
The number with the correct significant digits and sign.

Example: None

```
1 x = __my_function__(1.234, True, 3)
2 >>print(x)
3 >>-1.23
4
5 x = __my_function__(1.234, False, 3)
6 >>print(x)
7 >>1.23
```

6.2.1.12 *function* `append_snapshot_ask`(self, time, price_level)

Description: Appends the current form of the snapshot to `ask_history` and computes the event and appends the event record to `ask_events`.

parameters:

time:	datetime object
	timestamp that when the message was received
price_level:	float64
	price level where the change occurred

returns:

snapshot_ask	list
	The snapshot on the ask side.
ask_range	list
	the range of prices in <code>snapshot_ask</code> .

latent changes: None: None

Example: None

```
1 x = __my_function__(1.234, True, 3)
2 >>print(x)
3 >>-1.23
4
5 x = __my_function__(1.234, False, 3)
6 >>print(x)
7 >>1.23
```

6.2.1.13 *function* `append_signed_book`(self, time, price_level)

Description: Appends the current form of the snapshot to `ask_history` and computes the event and appends the event record to `ask_events`.

parameters:

time:	datetime object
	timestamp that when the message was received
price_level:	float64
	price level where the change occurred

returns:

snapshot_ask	list
	The snapshot on the ask side.
ask_range	list
	the range of prices in <code>snapshot_ask</code> .

latent changes: None: None

Example: None

```
1 x = __my_function__(1.234, True, 3)
2 >>print(x)
3 >>-1.23
4
5 x = __my_function__(1.234, False, 3)
6 >>print(x)
7 >>1.23
```

6.2.1.14 *function* `chk_mkt_can_overlap`(self, events, order_type)

Description: passes the order_type either market order or canceled order. If it was a market order checks for a recent canceled order of the same size and deletes the canceled order. likewise, if the ordertype passed was a canceled order it looks for a recent market order and deletes the canceled order.

The reason this is done this way is because the l2update channel only updates queue sizes while the ticker channel only announces market order arrivals.

parameters: events: list
 current event list to be modified
 order_type: string
 type of order observed ['market','cancellation']

returns: events list
 modified event list

latent changes: None: None

Example: None

```

1  x = __my_function__(1.234, True, 3)
2  >>print(x)
3  >>-1.23
4
5  x = __my_function__(1.234, False, 3)
6  >>print(x)
7  >>1.23

```

Accessors: The following functions, prefaced with `last_` are the currently named accessors. They cover the last order of a named type, the last state of the order book, and the market depth for a the last given state of the order book. If I come up with other relevant accessors or refine how each function works their changes will likely be noted in the versions, or at least a git entry.

6.2.1.15 *function* `last_inserted_order`(self, side='signed')

Description: Returns the latest limit order insertion from the bid, ask, or signed events. Defaults to signed.

[illegible]

latent changes: None: None

Example: None

```
1 x = __my_function__(1.234, True, 3)
2 >> print(x)
3 >> -1.23
4
5 x = __my_function__(1.234, False, 3)
6 >> print(x)
7 >> 1.23
```

list to create

1. accessors
 - (a) last CO, MO
 - (b) last image
 - (c) last event
 - (d) last market depth
2. price_match
3. update bid seq
4. update ask seq

6.3 history_funcs.py

Description: Helper functions that operate in recorder_full.py and LOBfuncs.py. It contains simple methods that are agnostic to side (e.g., bid/ask/signed) for the most part.

6.3.0.1 *function* `check_order`(snapshot, side)

Description: Checks that the snapshot is sorted correctly. Used in debugging when snapshots do not look well ordered. Only checks bid and ask side, because signed snapshot is a concatenated form of a bid and ask snapshot. `pass` placeholder can be changed to a print statement.

parameters:	snapshot:	list
		snpshot to check
	side:	string
		side bid, ask

returns: snapshot list

latent changes: None: None

Example: None

```
1 # let our snapshot_bid be:
2 snapshot_bid = [[10101.80, 0.013400],[10101.20, 0.44100], [10101.10, 0.450541]]
3
4 #I will say that if the snapshot is not ordered I can ask the function to return a
   boolean, True if ordered, else False
5 >>x = check_order(snapshot_bid, "bid")
6 >>print(x)
```

6.3.0.2 *function* `convert_array_to_list_dict`(history, pos_limit=5)**Description:** converts the time-series of order book states into a list of dictionaries for a single side.

parameters: history: list
 time series of order book states
 pos_limit: int, default := 5
 ordinal distance from the best bid/ask that is worth saving.

returns: volm_list: list
 list of dictionaries for volume sizes and their respective position.
 price_list: list
 list of dictionaries for prices and their respective position.

latent changes: None: None**Example:** None

```

1  # let our snapshot_bid be:
2  bid_history = [
3  [2019-08-14T20:42:27.265Z, [[10101.80, 0.013400], [10101.20, 0.44100], [10101.10,
4  0.450541], [10100.55, 5.24501], [10099.00, 10.24511], [10090.11, 24.21395]]],
5  [2019-08-14T20:42:27.500Z, [[10101.20, 0.44100], [10101.10, 0.450541], [10100.55,
6  5.24501], [10099.00, 10.24511], [10090.11, 24.21395], [10090.05]] ],
7  [2019-08-14T20:42:27.963Z, [[10101.20, 0.55200], [10101.10, 0.450541], [10100.55,
8  5.24501], [10099.00, 10.24511], [10090.11, 24.21395]]]
9  ]
10
11 ordinal_volumes, ordinal_prices = convert_array_list_to_dict(snapshot)
12
13 >>print(ordinal_volumes)
14 >>[{"time":2019-08-14T20:42:27.265Z, "1":0.013400, "2":0.44100, "3":0.450541, "4":
15 :5.24501, "5":10.24511},
16 {"time":2019-08-14T20:42:27.500Z, "1":0.44100, "2":0.450541, "3":5.24501, "4":
17 :10.24511, "5":24.21395},
18 {"time":2019-08-14T20:42:27.963Z, "1":0.55200, "2":0.450541, "3":5.24501, "4":
:10.24511, "5":24.21395}]

```


6.3.0.3 *function* `conver_array_to_list_dict_sob`(history, events, pos_limit=5)

Description: converts the time-series of order book states into a list of dictionaries for the signed order book. This has some nuance because the signed book format has negative volumes, and the ordinal scale is prefaced by a negative sign. Works similarly to `conver_array_to_list_dict` but uses the mid-price from `events` to find where to separate the bid and ask sides.

parameters:

- `history:` list
time series of order book states
- `events:` list
The events list where the function will extract the mid price from.
- `pos_limit:` int, default := 5
Ordinal distance from the best bid/ask that is worth saving.

returns:

- `volm_list:` list
list of dictionaries for volume sizes and their respective position.
- `price_list:` list
list of dictionaries for prices and their respective position.

latent changes: None: None

Example: None

```

1  # let our snapshot_bid be:
2  bid_history = [
3  [2019-08-14T20:42:27.265Z, [[10101.80, 0.013400], [10101.20, 0.44100], [10101.10,
4  0.450541], [10100.55, 5.24501], [10099.00, 10.24511], [10090.11, 24.21395]]],
5  [2019-08-14T20:42:27.500Z, [[10101.20, 0.44100], [10101.10, 0.450541], [10100.55,
6  5.24501], [10099.00, 10.24511], [10090.11, 24.21395], [10090.05]] ],
7  [2019-08-14T20:42:27.963Z, [[10101.20, 0.55200], [10101.10, 0.450541], [10100.55,
8  5.24501], [10099.00, 10.24511], [10090.11, 24.21395]]]
9  ]
10
11 ordinal_volumes, ordinal_prices = convert_array_list_to_dict(snapshot)
12
13 >>print(ordinal_volumes)
14 >>[{"time":2019-08-14T20:42:27.265Z, "1":0.013400, "2":0.44100, "3":0.450541, "4":
15 :5.24501, "5":10.24511},
16 {"time":2019-08-14T20:42:27.500Z, "1":0.44100, "2":0.450541, "3":5.24501, "4":
17 :10.24511, "5":24.21395},
18 {"time":2019-08-14T20:42:27.963Z, "1":0.52200, "2":0.450541, "3":5.24501, "4":
:10.24511, "5":2421395}]

```

```

15 >>print(ordinal_prices)
16 >>[{"time":2019-08-14T20:42:27.265Z, "1":10101.80, "2":10101.20, "3":10101.10, "4":
17 :10100.55, "5":10099.00},
18 {"time":2019-08-14T20:42:27.500Z, "1":10101.20, "2":10101.10, "3":10100.55, "4":
:10099.00, "5":10090.11},
{"time":2019-08-14T20:42:27.963Z, "1":10101.20, "2":10101.10, "3":10100.55, "4":
:10099.00, "5":10090.11}]

```

6.3.0.4 function `pd_excel_save`(title, hist_obj_dict)

Description: Convert the list of dictionary objects into a pandas dataframe object. Excises the first entry, because it is usually empty, saves the dataframe object as a .xlsx file with the title given as an argument.

parameters:

- `title:` string
Title of the output .xlsx file
- `hist_obj_dict:` list
The list of dictionaries to be converted to a pandas dataframe object.

returns: None: None

latent changes: None: None

Example: None

```

1 ordinal_volumes = [{"time":2019-08-14T20:42:27.265Z, "1":0.013400, "2":0.44100, "3":
   :0.450541, "4":5.24501, "5":10.24511},
2 {"time":2019-08-14T20:42:27.500Z, "1":0.44100, "2":0.450541, "3":5.24501, "4":10.24511,
   "5":24.21395},
3 {"time":2019-08-14T20:42:27.963Z, "1":0.52200, "2":0.450541, "3":5.24501, "4":10.24511,
   "5":24.21395}]
4
5 volm_title = "L2_orderbook_volm_bid"+str(hist_obj_dict[-1]['time'])+".xlsx"
6
7 >>pd_excel_save(volm_title, ordinal_volumes)

```

Below a simple table generated from L2_orderbook_volm_bid2019-08-14T20:42:27.963Z.xlsx is shown.

time	1	2	3	4	5
2019-08-14T20:42:27.265	0.013400	0.44100	0.450541	5.24501	10.24511
2019-08-14T20:42:27.500	0.44100	0.450541	5.24501	10.24511	24.21395
2019-08-14T20:42:27.963	0.52200	0.450541	5.24501	10.24511	24.21395

6.3.0.5 *function* `set_sign(event_size, side, order_type)`

Description: Sets the multiplicative sign, either 1 or -1 depending on the order type and event side.

parameters:

<code>event_size:</code>	<code>float64</code>	Size of the event, artifact from when I returned the signed event
<code>side:</code>	<code>string</code>	Event side either "bid" or "ask"
<code>order_type:</code>	<code>str</code>	the type of order either "insertion", "cancellation" or "market".

returns: `sign` `int`
The correct sign given the event's order type and side.

latent changes: `None: None`

Example: `None`

```
1 x = __my_function__(1.234, True, 3)
2 >>print(x)
3 >>-1.23
4
5 x = __my_function__(1.234, False, 3)
6 >>print(x)
7 >>1.23
```

6.3.0.6 *function* `set_signed_positiion`(position, side)

Description: Simple conditional that sets the position for the bid side.... I'm tired...

parameters: param_1: float64
 some input that will be passed to the function
 param_2: boolean
 a conditional that will be checked by the function
 param_3: int
 some int that is needed for the function
returns: trans_number float64
 The number with the correct significant digits and sign.
latent changes: trans_number float64
 The number with the correct significant digits and sign.

Example: None

```
1 x = __my_function__(1.234, True, 3)
2 >>print(x)
3 >>-1.23
4
5 x = __my_function__(1.234, False, 3)
6 >>print(x)
7 >>1.23
```

6.3.0.7 *function* `get_min_dec`(min_currency_denom, min_asset_value)

Description: determines the decimal place for the smallest amount of base currency needed for the minimal denomination of the float currency.

Example: Suppose you have an exchange rate of $10010.99USD/BTC$, (i.e., you need $10010.99USD$ to buy $1BTC$) then the amount of BTC in 1 cent or 0.01 USD which is the smallest denomination of the USD is $0.000000999BTC$. We see that this is 10^{-7} , `get_min_dec` would return 7, to pass onto `numpy.around`.

parameters: min_currency_denom : float64
 the minimum currency denomination (e.g., 0.01 USD)
 min_asset_value: float64
 the smallest current valuation of the asset.

returns: min_dec_out int

The number of decimal places out for the smallest movement of the asset for a minimal

latent changes: None: None

Example: None

1 ??????

7 Credits

7.1 Development Lead

- Ivan E. Perez - perez.ivan.e@gmail.com

7.2 Contributors

- BTC-DAVE - jewmansown69@gmail.com

8 License

GNU GPLv3 License

Copyright (c) 2020, Ivan E. Perez

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

9 Appendix

10 To do list

1. daily goals
- (a) write 2 whole function definitions with description, examples, i/o,

(b) help format or rewrite one unique section.

10.0.0.1 *function* `__my_function__(self, param_1, param_2=False, param_3)`

Description: Put your description here. In the next table we will state the parameters, on the left, with the object type on the right. just below it on the next line, we will give a brief description of the input parameter. After that we will use the returns portion in the exact same way. The final section is an example that uses `lstlisting` package to write snippets of python code for us. The filled out template is then included in the class .tex file and the class .tex file is included in the master manual .tex file.

For example this function will convert `param_1` as a float to the nearest significant digits as defined by `param_3` and will multiply by -1 if `param_2:=True`.

parameters:	<code>param_1:</code>	<code>float64</code>	some input that will be passed to the function
	<code>param_2:</code>	<code>boolean</code>	a conditional that will be checked by the function
	<code>param_3:</code>	<code>int</code>	some int that is needed for the function
returns:	<code>trans_number</code>	<code>float64</code>	The number with the correct significant digits and sign.
latent changes:	<code>trans_number</code>	<code>float64</code>	The number with the correct significant digits and sign.

Example: None

```
1 x = __my_function__(1.234, True, 3)
2 >>print(x)
3 >>-1.23
4
5 x = __my_function__(1.234, False, 3)
6 >>print(x)
7 >>1.23
8
```

variable name	type
	definition/description
<code>snapshot_bid:</code>	<code>list</code>
	The modified <code>snapshot_bid</code> .
<code>bid_range:</code>	<code>list</code>
	The modified range of bids available in the snapshot.
<code>pre_level.depth:</code>	<code>float64</code>
	again the pre level depth that idk why i still have this.

10.0.0.2 function `__my_function__`(self, param_1, param_2=False, param_3)

Description: Put your description here. In the next table we will state the parameters, on the left, with the object type on the right. just below it on the next line, we will give a brief description of the input parameter. After that we will use the returns portion in the exact same way. The final section is an example that uses `lstlisting` package to write snippets of python code for us. The filled out template is then included in the class .tex file and the class .tex file is included in the master manual .tex file.

For example this function will convert `param_1` as a float to the nearest significant digits as defined by `param_3` and will multiply by -1 if `param_2:=True`.

parameters:	<code>param_1:</code>	<code>float64</code>	some input that will be passed to the function
	<code>param_2:</code>	<code>boolean</code>	a conditional that will be checked by the function
	<code>param_3:</code>	<code>int</code>	some int that is needed for the function
returns:	<code>trans_number</code>	<code>float64</code>	The number with the correct significant digits and sign.
latent changes:	<code>trans_number</code>	<code>float64</code>	The number with the correct significant digits and sign.

Example: None

```

1  x = __my_function__(1.234, True, 3)
2  >>print(x)
3  >>-1.23
4
5  x = __my_function__(1.234, False, 3)
6  >>print(x)
7  >>1.23
8

```

References

- [1] Weibing Huang, Charles-Albert Lehalle, and Mathieu Rosenbaum. Simulating and analyzing order book data: The queue-reactive model. *Journal of the American Statistical Association*, 110(509):107–122, 2015.