Algorithmic platform for Nordnet nExtAPI (APNn)

Table of Content

Introduction

The Algorithmic platform for Nordnet nExtAPI (APNn) is a trading platform that can be used to trade multiple assets on Nordnet's nExtapi. It is developed to be flexible for change and is currently set to run in Nordnet's development/test environment. The software is built so that it can easily be setup to work with the production environment as well.

This document aims to describe the software architecture and to briefly describe all the different functions and files being used. To get a deeper understanding, additional comments can be found in each of the different files.

Use this software at your own risk, and if you want to trade in the live Nordnet environment you will still need to certify yourself with your own implementation. This platform is built as an inspiration for future development. I'm not a professional developer nor a security expert and do not take any responsibility for the software not working or having security flaws.

Questions, requests, and suggestions can be sent to @ottovelander on twitter (checked sporadically).

Setup (unix / OSX)

To setup APNn you will need to have python3 and pip3 installed

1. **Installing modules**
   After downloading the APNn, install the modules in the “requirements.txt” file by running:

   pip3 install –r requirements.txt

   (you might need to install additional modules as well).

2. **Certificates**
   Download your certificate files from nordnet. Download the “.pem”-file and put it in the lib folder (replacing the txt placeholder).

3. **Config settings**
   Edit config.json by entering the location of your pem-file, add your login credentials to Nordnet, your e-mail information etc. More introduction to the config file is presented later.

4. **Database**
   The asset history data collected by APNn is stored in a SQL database. APNn already comes with a created database for the test environment, but in order to create a new database (e.g. to trade different/multiple instruments) you will need to edit the “setup_database.py” and then create a new database by running

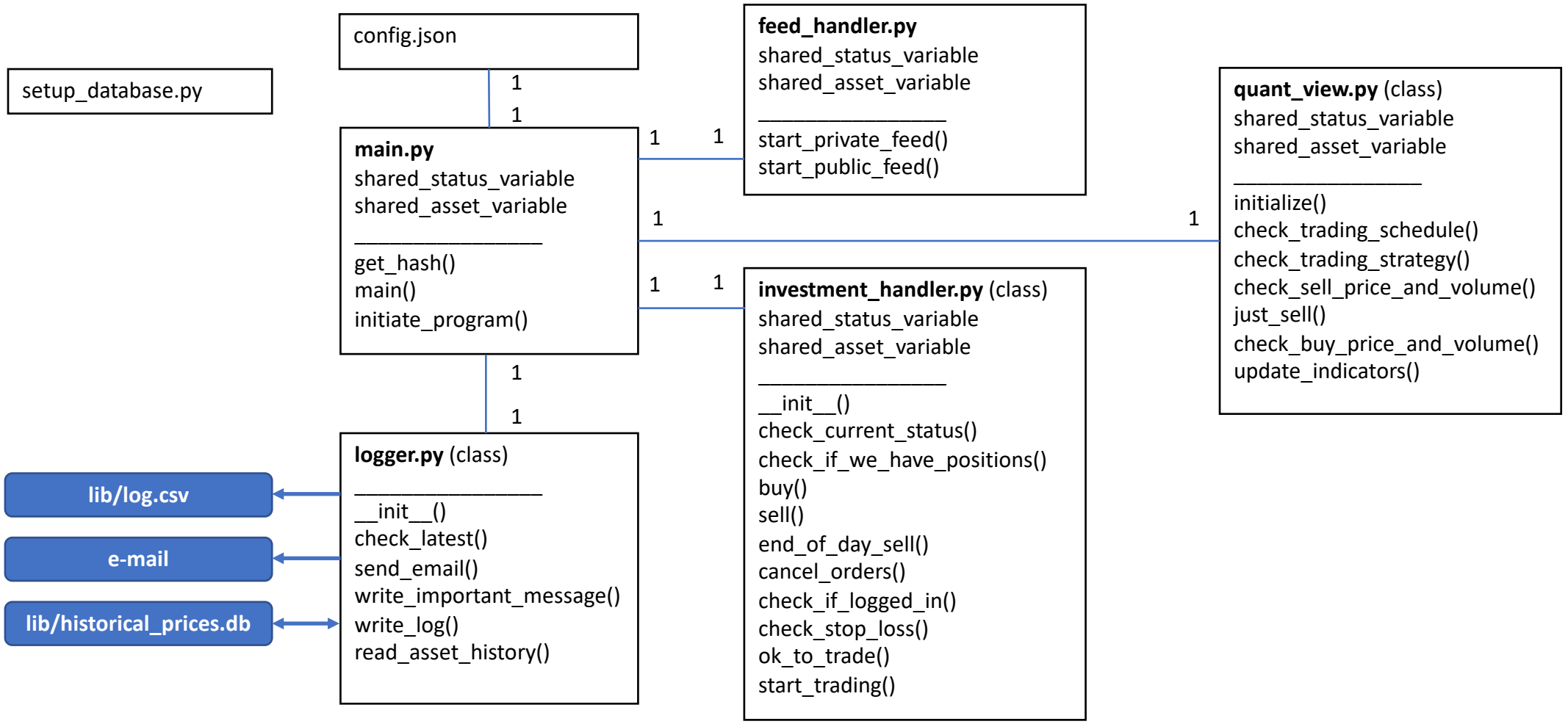
   python3 setup_database.py

5. **Start**
   To start APNn, run (from the main folder):

   python3 main.py

Software architecture

Overview of the different files, classes, functions, and most important variables



Files and functions

(overview and brief explinations)

config.json

```
{
    "login":{
        "username":"nordnetusername",
        "password":"nordnetpassword",
        "certificate":"lib/NEXTAPI_TEST_public.pem",
        "accountnumber": 0
    },
    "api-info":{
        "url":"api.test.nordnet.se",
        "service": "NEXTAPI",
        "api_version": "2",
        "headers": {"Accept": "application/json"}
    },
    "e-mail":{
        "from_addr_username":"email_send_account@gmail.com",
        "from_addr_password":"password_send_account",
        "from_addr_smtp":"smtp.gmail.com:587",
        "to_addr":"email_to_send_to@gmail.com"
    },
    "history":{
        "database":"lib/historical_prices.db",
        "log":"lib/log.csv"
    },
    "hours":{
        "reset_loss_safety":"08:30",
        "cancel_morning_orders":"09:05",
        "trade_hours_open":"09:20",
        "trade_hours_close":"16:45",
        "cancel_evening_orders":"17:15",
        "start_logging":"09:00",
        "stop_logging":"17:30",
        "saturday":5,
        "sunday":6
    }
}
```

This config file is loaded in main.py and includes the basic software configuration.

login

Edit the login credentials to Nordnet's nExtapi. This is also where you specify where your certificate file is. There is also the possibility to change the account you trade from (in the live environment you might have multiple accounts). The account is by default set to 0.

api-info

Only change this information if you are going into the live environment.

e-mail

APNn uses a gmail account to alert the user if certain events happen (e.g. a trade / error has occurred). For this to work, it is recommended that you create a new gmail account for this specific purpose.

The to_addr e-mail is used when important messages are being sent, for example to e-mail accounts that are checked on a more regular basis.

history

Specifies where the database of historical prices is, as there might be multiple being used between different environments. A log, created as a csv-file is used to log events that are happening.

hours

Specifies certain hours when APNn should conduct trading. This section also specifies certain events that can be useful, e.g. when to start cancelling the orders and liquidated the assets that are long.

main.py

```python
import traceback

#Load config (has information about login, path to api, and certificate file location)
with open('lib/config.json') as json_data_file:
    config = json.load(json_data_file)

USERNAME = config['login']['username']
...


#Program tracking variable
global_number_of_fails = 0

#get_hash: Generate authhash
def get_hash(username, password):
...


#main: Start of the program
def main():
...

#initiate_program: Using the login information, initiate feeds and investment class
def initiate_program(auth_hash):
    ...


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

The main file is responsible for the login to Nordnet's API platform. After login, it starts the loops defined in the feed_handler and creates the objects Logger, InvestmentHandler and Quant_View. It also setups three important variables that are shared between the different files/objects:

- shared_status_variable
- stop_loss_variable
- shared_asset_variable

These are referenced in all of the different files to share important information, like if a trade has been executed, the latest price of the assets, and if a stop loss order needs to be executed.

get_hash(username, password)

This function is creating the hash needed to authenticate against the Nordnet API. It is called from the main() function.

main()

This function is where APNn is started, it creates a hash and initiates the program.

initiate_program(auth_hash)

This function is used to setup all the shared variables that are mentioned above. It also holds the main try and catch statement. It keeps track of all the feeds (public and private), and is responsible for initiating the logging of data that is collected in the MySql-database.

In this function three different threads are initiated to keep track of what is happening in the public feed, private feed, and the trading strategy. If one of these threads fail, APNn will try to restart.

feed_handler.py

```python
import json
import socket, ssl, pprint
import http.client
from time import sleep
import investment_handler as investment_handler
import logger as logger
import traceback
import quant_view as quant_view


#start_private_feed: Looping, checking the socket of the private feed
def start_private_feed(session_key, private_feed_hostname, private_feed_port,
investment_handler, shared_status_variable, shared_asset_variable):

    try:
        ...

    except Exception as e:
        shared_status_variable['rprf'] = False
        shared_status_variable['exception'] = str(traceback.format_exc())


#start_public_feed: Keeps track of the stocks we subscribe to
def start_public_feed(session_key, public_feed_hostname, public_feed_port,
investment_handler, quant_view, shared_status_variable, shared_asset_variable):

    try:
        ...

    except Exception as e:
        shared_status_variable['rpuf'] = False
        shared_status_variable['exception'] = str(traceback.format_exc())
```

The feed_handler is not a class, but holds two types of functions that are used to setup the two threads mentioned in the main.py file.

start_private_feed(session_key, private_feed_hostname, private_feed_port, investment_handler, shared_status_variable, shared_asset_variable)

The function keeps track of the private feed, i.e. the updates that occur regarding the trades and orders that are made in the API. Some changes will need to be made in this function if multiple assets are traded.

start_public_feed(session_key, public_feed_hostname, public_feed_port, investment_handler, quant_view, shared_status_variable, shared_asset_variable)

The public feed keep track of everything regarding the assets that we are subscribing too (these are defined in quant_view.py). If there is an update regarding price or volume, on any of the bid or ask levels, this will be recorded in the shared_asset_variable.

Potentially, there is a possibility to check your trading strategy in the Quant_View object every time there is an asset update. Depending on the processing power of your trading machine, you could potentially enable this by uncommenting one of the last lines of code in this function.

quant_view.py

```
class Quant_View:
    ...

    #initialize: Setup initial trade values
    def initialize(self, shared_asset_variable, loggervariable):
        ...


    #check_trading_schedule: Runs every 5 second (set in investment_handler)
    def check_trading_schedule(self, investment_handler, shared_status_variable,
    shared_asset_variable, current_datetime):

        ...


    #check_sell_price_and_volume: Recursive selling strategy that will loop through
    the order book
    def check_sell_price_and_volume(self, investment_handler, shared_status_variable,
    shared_asset_variable, key, accumulated_vol, level):

        ...


    #check_sell_price_and_volume: Recursive selling strategy that will loop through
    the order book
    def just_sell(self, investment_handler, shared_status_variable,
    shared_asset_variable, key, accumulated_vol, level):

        ...


    #check_buy_price_and_volume: Recursive buying strategy that will loop through the
    order book
    def check_buy_price_and_volume(self, investment_handler, shared_status_variable,
    shared_asset_variable, key, accumulated_vol, level):

        ...


    #update_indicators: Function that will be run from the main loop. Indicator that
    can influence buying/selling
    def update_indicators(self, investment_handler, shared_status_variable,
    shared_asset_variable, current_time):

        ...
```

The Quant_View class is used to formulize the trading strategy to be implemented. The object is initiated in main.py file.

initialize(self, shared_asset_variable, loggervariable)

The initialize function is used to setup the asset variables that are supposed to be traded.*

check_trading_schedule(self, investment_handler, shared_status_variable, shared_asset_variable, current_datetime)

The trading schedule is used to execute certain actions at certain points in time. As an example, you might want to liquidate your assets by the end of the day. The function can also be used to setup certain stop loss functions.

check_trading_strategy(self, investment_handler, shared_status_variable, shared_asset_variable, key)

The check trading strategy function is called from the update_indicators function each minute. It is in this function that initiates the trade strategy.

check_sell_price_and_volume(self, investment_handler, shared_status_variable, shared_asset_variable, key, accumulated_vol, level)

This recursive function will check for the optimal sell price (with given trade strategy) that will ensure that the entire order can be filled by looping over the different bid levels.

just_sell(self, investment_handler, shared_status_variable, shared_asset_variable, key, accumulated_vol, level)

The just_sell function is written to execute a quick sale for the best price. It is for example called to execute a stop-loss.

check_buy_price_and_volume(self, investment_handler, shared_status_variable, shared_asset_variable, key, accumulated_vol, level)

This recursive function will check for the optimal buy price (with given trade strategy) that will ensure that the entire order can be filled by looping over the different ask levels.

update_indicators(self, investment_handler, shared_status_variable, shared_asset_variable, current_time)

The update indicator function is called from the main.py file after the database has been updated with new information.

*It is important to note that in order to log the data recorded from these assets, updates need to be made in the database to accommodate this (e.g. if multiple assets are supposed to be traded).

investment_handler.py (1/2)

```
class Investment_Handler:

    #init: Constructor of class. Will set the key variables to perform trading
    def __init__(self, session_key, account_number, quant_view, shared_status_variable, shared_asset_variable, loggervariable, config):

    #check_current_status: Run at initialization
    def check_current_status(self,shared_status_variable, shared_asset_variable):


    #check_if_we_have_positions: Checking if we have positions (used to check whenever there is a sale)
    def check_if_we_have_positions(self, shared_status_variable, shared_asset_variable):


    #buy: Buy function, will be triggered from quant view depending on strategy
    def buy(self, shared_status_variable, shared_asset_variable, key, price, asked_volume, MAXIMUM_TRADE_VOLUME):


    #sell: Will be used when it is time to sell (end of day or when trading strategy indicates)
    def sell(self, shared_status_variable, shared_asset_variable, key, price):


    #end_of_day_sell: Will be used to try to sell remaining positions we have in the instruments.
    def end_of_day_sell(self, shared_status_variable, shared_asset_variable, key, why):


    #cancel_orders: Will be used in the end of the day to cancel pending orders...
    def cancel_orders(self, shared_status_variable):


    #check_if_logged_in: Housekeeping function called from schedule in quant_view
    def check_if_logged_in(self, shared_status_variable):


    #check_stop_loss: Housekeeping function called from schedule in quant_view
    def check_stop_loss(self, shared_status_variable):


    #ok_to_trade: Returns a boolean to check if it is ok to trade
    def ok_to_trade(self, shared_status_variable, side, shared_asset_variable, key):


    #start_trading: Main loop of system, called from the "main.py file"
    def start_trading(self, shared_status_variable, shared_asset_variable):
```

The investment_class is closely related to the Quant_View but handles the actual API queries to the nExtapi.

__init__(self, session_key, account_number, quant_view, shared_status_variable, shared_asset_variable, loggervariable, config):

The constructor is initializing the object with information about the trading hours and API information that is needed for the queries etc.

check_current_status(self,shared_status_variable, shared_asset_variable):

This function is called whenever the object is initialized and is used to get an understanding of the current status on the account (e.g. how many orders that are outstanding etc.)

check_if_we_have_positions(self, shared_status_variable, shared_asset_variable):

This is a helper function to check if we have any positions in the assets that we aim to trade.

buy(self, shared_status_variable, shared_asset_variable, key, price, asked_volume, MAXIMUM_TRADE_VOLUME):

The buy function is used to send a buy order to the API according to the price that was calculated in the Quant_View.

sell(self, shared_status_variable, shared_asset_variable, key, price):

The sell function is used to initialize send a sell order to the API according to the price that was calculated in the Quant_View.

end_of_day_sell(self, shared_status_variable, shared_asset_variable, key, why):

The function is used to trigger a sale by the end of the day. It is called from the Quant_View.

cancel_orders(self, shared_status_variable):

The cancel_orders function loops through all pending orders and cancels them.

investment_handler.py (2/2)

```
class Investment_Handler:

    ...

    #init: Constructor of class. Will set the key variables to perform trading
    def __init__(self, session_key, account_number, quant_view, shared_status_variable, shared_asset_variable, loggervariable, config):
    ...

    #check_current_status: Run at initialization
    def check_current_status(self,shared_status_variable, shared_asset_variable):
    ...


    #check_if_we_have_positions: Checking if we have positions (used to check whenever there is a sale)
    def check_if_we_have_positions(self, shared_status_variable, shared_asset_variable):
    ...


    #buy: Buy function, will be triggered from quant view depending on strategy
    def buy(self, shared_status_variable, shared_asset_variable, key, price, asked_volume, MAXIMUM_TRADE_VOLUME):
    ...


    #sell: Will be used when it is time to sell (end of day or when trading strategy indicates)
    def sell(self, shared_status_variable, shared_asset_variable, key, price):
    ...


    #end_of_day_sell: Will be used to try to sell remaining positions we have in the instruments.
    def end_of_day_sell(self, shared_status_variable, shared_asset_variable, key, why):
    ...


    #cancel_orders: Will be used in the end of the day to cancel pending orders...
    def cancel_orders(self, shared_status_variable):
    ...


    #check_if_logged_in: Housekeeping function called from schedule in quant_view
    def check_if_logged_in(self, shared_status_variable):
    ...


    #check_stop_loss: Housekeeping function called from schedule in quant_view
    def check_stop_loss(self, shared_status_variable):
    ...


    #ok_to_trade: Returns a boolean to check if it is ok to trade
    def ok_to_trade(self, shared_status_variable, side, shared_asset_variable, key):
    ...


    #start_trading: Main loop of system, called from the "main.py file"
    def start_trading(self, shared_status_variable, shared_asset_variable):
    ...
```

check_if_logged_in(self, shared_status_variable):

This function is called from the check_trading_schedule function in the Quant_View. It is used to keep the current login session active.

check_stop_loss(self, shared_status_variable):

The check loss function will query the Nordnet API to see if there are any large changes to the portfolio value that would trigger a warning to stop the trading.

ok_to_trade(self, shared_status_variable, side, shared_asset_variable, key):

The ok_to_trade function checks if it is ok to trade by reviewing the trading hours and that there hasn't been a big loss that would halt the trading.

start_trading(self, shared_status_variable, shared_asset_variable):

This function is a loop that will will continuously check the trading schedule in the Quant_View. It is called by the main.py file to initiate the trading. It is also the place where where the status of the shared_status_variable and shared_asset_variable are printed out to the console/terminal.

logger.py

```
class Logger:


    #Constructor for setting up database location, e-mail address to use etc.
    def __init__(self, config):
...


    #check_latest: Helper function to check if we already logged data for this time
    def check_latest(self, shared_asset_variable, key):
...


    #send_email: Sends email to check what has happened
    def send_email(self, message, receiver):
...


    #write_important_message: message that deserves extra attentions
    def write_important_message(self, message):
...


    #write_log: Send e-mail and write to log, called when major events happen
    def write_log(self, message):
...


    #write_history: logging values, called from the main.py loop
    def write_history(self, shared_asset_variable):
...


    #read_history: read history from database. Only for subscribed assets
    def read_asset_history(self, shared_asset_variable, key, minutes):
...
```

The logger class Is initiated and called from main.py to update the database with new information about the different assets. It is also where the software gets historical data to update the different trading indicators. Additionally, it holds the logging and e-mail messaging services of important messages that are being sent.

__init__(self, config) :

Constructor to setup the Logger object. It passes information such as the e-mail account information that is needed to setup the e-mail notification service.

check_latest(self, shared_asset_variable, key):

This function is used to make sure that the software does not log the same information multiple times.

send_email(self, message, receiver):

send_email is used to send an e-mail about important events that are happening (e.g. when a trade has been executed).

write_important_message(self, message):

This function is used to log an important events and send an e-mail to an additional preferred e-mail account (e.g. one that is more often checked).

write_log(self, message):

This function is used to log information in a CSV file about what is happening in the system.

write_history(self, shared_asset_variable):

The write_history function is used to record asset related data in the SQL database. This (minute) data can later be used to calculate the trading indicators.

read_asset_history(self, shared_asset_variable, key, minutes):

The read_asset_history is called from the Quant_View to update the trading indicators.

setup_database.py

```python
import sqlite3

def create_table(db_name, sql):
    with sqlite3.connect(db_name) as db:
        cursor = db.cursor()
        cursor.execute(sql)
        db.commit()

if __name__ == "__main__":
    db_name = "lib/historical_prices.db"

    #ISIN = Tablename.
    sql = """create table SE0000108656
            (AID integer,
            isin text,
            identifier text,
            market_id integer,
            bid float,
            bid_volume float,
            ask float,
            ask_volume float,
            last float,
            bid1 float,
            bid_volume1 float,
            bid2 float,
            bid_volume2 float,
            bid3 float,
            bid_volume3 float,
            bid4 float,
            bid_volume4 float,
            bid5 float,
            bid_volume5 float,
            ask1 float,
            ask_volume1 float,
            ask2 float,
            ask_volume2 float,
            ask3 float,
            ask_volume3 float,
            ask4 float,
            ask_volume4 float,
            ask5 float,
            ask_volume5 float,
            time datetime,
            primary key(AID)
            )"""
    create_table(db_name, sql)
```

The setup database is a helper file to create a database structure used for the trading. In order to be able to trade certain assets, one will need to update this file so that the right tables for those assets are created in the database (called historical_prices.db).

In the example to the left, a table for an Ericsson stock is created according to the structure used in the software. Multiple tables can be created by modifying the code.

When running the setup_database.py, ensure that it is being run from inside the main folder so that the new database is placed in the correct lib-folder. Additionally, before creating a new database file, you should remove the old one.

Screenshot

Screenshot of APNn in action – Output from terminal

Printed information from the **shared_status_variable**

Printed information from the **shared_asset_variable**

```
{   'available_funds': 1031049.12,
    'current_orders': False,
    'exception': 'none',
    'have_positions': False,
    'rivh': True,
    'rprf': True,
    'rpuf': True,
    'stop_loss': {   'initial_total_portfolio_value': 1031049.12,
                     'last_order': datetime.datetime(2000, 1, 1, 1, 1, 1, 1),
                     'no_big_loss': True,
                     'total_portfolio_value_morning': 1000000.0}}
{   '10111': {   'ask1': 0.0,
                 'ask2': 0.0,
                 'ask3': 0.0,
                 'ask4': 0.0,
                 'ask5': 0.0,
                 'ask_volume1': 0,
                 'ask_volume2': 0,
                 'ask_volume3': 0,
                 'ask_volume4': 0,
                 'ask_volume5': 0,
                 'bid1': 71.64,
                 'bid2': 0.0,
                 'bid3': 0.0,
                 'bid4': 0.0,
                 'bid5': 0.0,
                 'bid_volume1': 248849,
                 'bid_volume2': 0,
                 'bid_volume3': 0,
                 'bid_volume4': 0,
                 'bid_volume5': 0,
                 'current_ask': 0.0,
                 'current_ask_volume': 0,
                 'current_bid': 71.64,
                 'current_bid_volume': 248849,
                 'current_last': 71.08,
                 'identifier': '101',
                 'indicator': {   'SMA10': 0.0,
                                  'SMA5': 0.0,
                                  'updated_at': datetime.datetime(2000, 1, 1, 1, 1, 1, 1)},
                 'isin': 'SE0000108656',
                 'market_id': 11,
                 'position_acq_price': 0,
                 'position_qty': 0}}
```