



# Writing an Algorithm in Python

The explanation below assumes familiarity with the basics of Object-Oriented Programming in Python. To refresh this knowledge you can have a look at the source provided on the .

[The Challenge](#)

[Overview of the `Trader` class](#)

[Overview of the `TradingState` class](#)

[Trade class](#)

[OrderDepth class](#)

[Observation class](#)

[How to send orders using the `Order` class](#)

[Position Limits](#)

[Example of Trading](#)

[Technical Notes](#)

[Available Resources to Help Build the Algorithm](#)

[Appendix A: `Trader` Class Example](#)

[Appendix B: `datamodel.py` file](#)

[Appendix C: Supported libraries](#)

## The Challenge

For the algorithmic trading challenge, you will be writing and uploading a trading algorithm class in Python, which will then be set loose on the island exchange. On this exchange, the algorithm will trade against a number of bots, with the aim of earning as many SeaShells (the currency of the archipelago) as possible. The algorithmic trading challenge consists of several rounds, that take place on different days of the challenge. At the beginning of each round, it is disclosed which new products will be available for trading on that day. Sample data for these products is provided that players can use to get a better understanding of the

price dynamics of these products, and consequently build a better algorithm for trading them. While most days will feature new products, the old products will also still be tradable in the rounds after which they are introduced. This means that based on the result of the previous rounds, players also have the opportunity to analyse and optimise their trading strategies for these “old” products.

The format for the trading algorithm will be a predefined `Trader` class, which has a single method called `run` which contains all the trading logic coded up by the trader. Once the algorithm is uploaded it will be run in the simulation environment. The simulation consists of a large number of iterations. During each iteration the `run` method will be called and provided with a `TradingState` object. This object contains an overview of all the trades that have happened since the last iteration, both the algorithm's own trades as well as trades that happened between other market participants. Even more importantly, the `TradingState` will contain a per product overview of all the outstanding buy and sell orders (also called “quotes”) originating from the bots. Based on the logic in the `run` method the algorithm can then decide to either send orders that will fully or partially match with the existing orders, e.g. sending a buy (sell) order with a price equal to or higher (lower) than one of the outstanding bot quotes, which will result in a trade. If the algorithm sends a buy (sell) order with an associated quantity that is larger than the bot sell (buy) quote that it is matched to, the remaining quantity will be left as an outstanding buy (sell) quote with which the trading bots will then potentially trade. When the next iteration begins, the `TradingState` will then reveal whether any of the bots decided to “trade on” the player's outstanding quote. If none of the bots trade on an outstanding player quote, the quote is automatically cancelled at the end of the iteration.

Every trade done by the algorithm in a certain product changes the “position” of the algorithm in that product. E.g. if the initial position in product X was 2 and the algorithm buys an additional quantity of 3, the position in product X is then 5. If the algorithm then subsequently sells a quantity of 7, the position in product X will be -2, called “short 2”. Like in the real world, the algorithms are restricted by per product position limits, which define the absolute position (long or short) that the algorithm is not allowed to exceed. If the aggregated quantity of all the buy (sell) orders an algorithm sends during a certain iteration would, if all fully matched, result in the algorithm obtaining a long (short) position exceeding the position limit, all the orders are cancelled by the exchange.

In the first section, the general outline of the `Trader` class that the player will be creating is outlined.

## Overview of the `Trader` class

Below an abstract representation of what the trader class should look like is shown. The class only requires a single method called `run`, which is called by the simulation every time a new `TraderState` is available. The logic within this `run` method is written by the player and determines the behaviour of the algorithm. The output of the method is a dictionary named `result`, which contains all the orders that the algorithm decides to send based on this logic.

```
from datamodel import OrderDepth, UserId, TradingState, Order
from typing import List
import string

class Trader:

    def run(self, state: TradingState):
        print("traderData: " + state.traderData)
        print("Observations: " + str(state.observations))

        # Orders to be placed on exchange matching engine
        result = {}
        for product in state.order_depths:
            order_depth: OrderDepth = state.order_depths[product]
            orders: List[Order] = []
            acceptable_price = 10 # Participant should calculate this value
            print("Acceptable price : " + str(acceptable_price))
            print("Buy Order depth : " + str(len(order_depth.buy_orders)) + ", Sell order depth : " + str(len(order_depth.sell_orders)))

            if len(order_depth.sell_orders) != 0:
                best_ask, best_ask_amount = list(order_depth.sell_orders.items())[0]
                if int(best_ask) < acceptable_price:
```

```

        print("BUY", str(-best_ask_amount) + "x", best_ask)
        orders.append(Order(product, best_ask, -best_ask_amount))

    if len(order_depth.buy_orders) != 0:
        best_bid, best_bid_amount = list(order_depth.buy_orders.items())[0]
        if int(best_bid) > acceptable_price:
            print("SELL", str(best_bid_amount) + "x", best_bid)
            orders.append(Order(product, best_bid, -best_bid_amount))

    result[product] = orders

    # String value holding Trader state data required.
    # It will be delivered as TradingState.traderData on next execution.
    traderData = "SAMPLE"

    # Sample conversion request. Check more details below.
    conversions = 1
    return result, conversions, traderData

```

Example implementation above presents placing order idea as well.

When you send the Trader implementation there is always submission identifier generated. It's UUID value similar to "59f81e67-f6c6-4254-b61e-39661eac6141". Should any questions arise on the results, feel free to communicate on Discord channels. Identifier is absolutely essential to answer questions. Please put it in the message.

Technical implementation for the trading container is based on Amazon Web Services Lambda function. Based on the fact that Lambda is stateless AWS can not guarantee any class or global variables will stay in place on subsequent calls. We provide possibility of defining a traderData string value as an opportunity to keep the state details. Any Python variable could be serialised into string with jsonpickle library and deserialised on the next call based on TradingState.traderData property. Container will not interfere with the content.

To get a better feel for what this `TradingState` object is exactly and how players can use it, a description of the class is provided below.

## Overview of the `TradingState` class

The `TradingState` class holds all the important market information that an algorithm needs to make decisions about which orders to send. Below the definition is provided for the `TradingState` class:

```
Time = int
Symbol = str
Product = str
Position = int

class TradingState(object):
    def __init__(self,
                  traderData: str,
                  timestamp: Time,
                  listings: Dict[Symbol, Listing],
                  order_depths: Dict[Symbol, OrderDepth],
                  own_trades: Dict[Symbol, List[Trade]],
                  market_trades: Dict[Symbol, List[Trade]],
                  position: Dict[Product, Position],
                  observations: Observation):
        self.traderData = traderData
        self.timestamp = timestamp
        self.listings = listings
        self.order_depths = order_depths
        self.own_trades = own_trades
        self.market_trades = market_trades
        self.position = position
        self.observations = observations

    def toJSON(self):
        return json.dumps(self, default=lambda o: o.__dict__, sort_keys=True)
```

The most important properties

- `own_trades`: the trades the algorithm itself has done since the last `TradingState` came in. This property is a dictionary of `Trade` objects with key being a product name. The definition of the `Trade` class is provided in the subsections below.
- `market_trades`: the trades that other market participants have done since the last `TradingState` came in. This property is also a dictionary of `Trade` objects with key being a product name.
- `position`: the long or short position that the player holds in every tradable product. This property is a dictionary with the product as the key for which the value is a signed integer denoting the position.
- `order_depths`: all the buy and sell orders per product that other market participants have sent and that the algorithm is able to trade with. This property is a dict where the keys are the products and the corresponding values are instances of the `OrderDepth` class. This `OrderDepth` class then contains all the buy and sell orders. An overview of the `OrderDepth` class is also provided in the subsections below.

## `Trade` class

Both the `own_trades` property and the `market_trades` property provide the traders with a list of trades per products. Every individual trade in each of these lists is an instance of the `Trade` class.

```
Symbol = str
```

```
UserId = str
```

```
class Trade:
```

```
    def __init__(self, symbol: Symbol, price: int, quantity: int, buyer: UserId = None,
                  self.symbol = symbol
                  self.price: int = price
```

```

        self.quantity: int = quantity
        self.buyer = buyer
        self.seller = seller
        self.timestamp = timestamp

    def __str__(self) → str:
        return "(" + self.symbol + ", " + self.buyer + " << " + self.seller + ", " + str(se

    def __repr__(self) → str:
        return "(" + self.symbol + ", " + self.buyer + " << " + self.seller + ", " + str(se

```

These trades have five distinct properties:

1. The symbol/product that the trade corresponds to (i.e. are we exchanging apples or oranges)
2. The price at which the product was exchanged
3. The quantity that was exchanged
4. The identity of the buyer in the transaction
5. The identity of the seller in this transaction

On the island exchange, like on most real-world exchanges, counterparty information is typically not disclosed. Therefore properties 4 and 5 will only be non-empty strings if the algorithm itself is the buyer (4 will be "SUBMISSION") or the seller (5 will be "SUBMISSION").

## OrderDepth class

Provided by the `TradingState` class is also the `OrderDepth` per symbol. This object contains the collection of all outstanding buy and sell orders, or "quotes" that were sent by the trading bots, for a certain symbol.

```

class OrderDepth:
    def __init__(self):
        self.buy_orders: Dict[int, int] = {}
        self.sell_orders: Dict[int, int] = {}

```

All the orders on a single side (buy or sell) are aggregated in a dict, where the keys indicate the price associated with the order, and the corresponding values indicate the total volume on that price level. For example, if the `buy_orders` property would look like this for a certain product `{9: 5, 10: 4}`. That would mean that there is a total buy order quantity of 5 at the price level of 9, and a total buy order quantity of 4 at a price level of 10. Players should note that in the `sell_orders` property, the quantities specified will be negative. E.g., `{12: -3, 11: -2}` would mean that the aggregated sell order volume at price level 12 is 3, and 2 at price level 11.

Every price level at which there are buy orders should always be strictly lower than all the levels at which there are sell orders. If not, then there is a potential match between buy and sell orders, and a trade between the bots should have happened.

## Observation class

Observation details help to decide on eventual orders or conversion requests. There are two items delivered inside the `TradingState` instance:

1. Simple product to value dictionary inside `plainValueObservations`
2. Dictionary of complex **ConversionObservation** values for respective products. Used to place conversion requests from `Trader` class. Structure visible below.

```
class ConversionObservation:
```

```
    def __init__(self, bidPrice: float, askPrice: float, transportFees: float, exportTariff: float, importTariff: float, sugarPrice: float, sunlightIndex: float):
        self.bidPrice = bidPrice
        self.askPrice = askPrice
        self.transportFees = transportFees
        self.exportTariff = exportTariff
        self.importTariff = importTariff
        self.sugarPrice = sugarPrice
        self.sunlightIndex = sunlightIndex
```



In case you decide to place conversion request on product listed integer number should be returned as "conversions" value from run() method. Based on logic defined inside Prosperity container it will convert positions acquired by submitted code. There is a number of conditions for conversion to happen:

- You need to obtain either long or short position earlier.
- Conversion request cannot exceed the minimum of possessed items count and conversion limit.
- In case you have 10 items short (-10) and conversion limit is 5, you can only request from 1 to 5. Request for 6 or more will be fully ignored.
- While conversion happens you will need to cover transportation and import/export tariff.
- Conversion request is not mandatory. You can send 0 or None as value.

## How to send orders using the **Order** class

After performing logic on the incoming order state, the **run** method defined by the player should output a dictionary containing the orders that the algorithm wants to send. The keys of this dictionary should be all the products that the algorithm wishes to send orders for. These orders should be instances of the **Order** class. Each order has three important properties. These are:

1. The symbol of the product for which the order is sent.
2. The price of the order: the maximum price at which the algorithm wants to buy in case of a BUY order, or the minimum price at which the algorithm wants to sell in case of a SELL order.
3. The quantity of the order: the maximum quantity that the algorithm wishes to buy or sell. If the sign of the quantity is positive, the order is a buy order, if the sign of the quantity is negative it is a sell order.

```
Symbol = str
```

```
class Order:
```

```

def __init__(self, symbol: Symbol, price: int, quantity: int) → None:
    self.symbol = symbol
    self.price = price
    self.quantity = quantity

def __str__(self) → str:
    return "(" + self.symbol + ", " + str(self.price) + ", " + str(self.quantity) + ")"

def __repr__(self) → str:
    return "(" + self.symbol + ", " + str(self.price) + ", " + str(self.quantity) + ")"

```

If there are active orders from counterparties for the same product against which the algorithms' orders can be matched, the algorithms' order will be (partially) executed right away. If no immediate or partial execution is possible, the remaining order quantity will be visible for the bots in the market, and it might be that one of them sees it as a good trading opportunity and will trade against it. If none of the bots decides to trade against the remaining order quantity, it is cancelled. Note that after cancellation of the algorithm's orders but before the next `Tradingstate` comes in, bots might also trade with each other.

Note that on the island exchange players' execution is infinitely fast, which means that all their orders arrive in the exchange matching engine without any delay. Therefore, all the orders that a player sends that can be immediately matched with an order from one of the bots, will be matched and result in a trade. In other words, none of the bots can send an order that is faster than the player's order and get the opportunity instead.

See

[Trading Glossary](#) for a more elaborate explanation of order execution in financial markets.

## Position Limits

Just like in the real world of trading, there are position limits, i.e. limits to the size of the position that the algorithm can trade into in a single product. These position limits are defined on a per-product basis, and refer to the absolute allowable

position size. So for a hypothetical position limit of 10, the position can neither be greater than 10 (long) nor less than -10 (short). On the Prosperity Island exchange, this position limit is enforced by the exchange. If at any iteration, the player's algorithm tries to send buy (sell) orders for a product with an aggregated quantity that would cause the player to go over long (short) position limits if all orders would be fully executed, all orders will be rejected automatically. For example, the position limit in product X is 30 and the current position is -5, then any aggregated buy order volume exceeding  $30 - (-5) = 35$  would result in an order rejection.

For an overview of the per-product position limit players are referred to the 'Rounds' section on [Prosperity 3 Wiki](#).

Below two example iterations are provided to give an idea what the simulation behaviour looks like.

## Example of Trading

For the following example we assume a situation with two products:

- PRODUCT1 with position limit 10
- PRODUCT2 with position limit 20

At the start of the first iteration the run method is called with the `TradingState` generated by the below code. Note: the `datamodel.py` file from which the classes are imported is provided in Appendix B. The code can also be used to test algorithms locally.

```
from datamodel import Listing, OrderDepth, Trade, TradingState

timestamp = 1000

listings = {
    "PRODUCT1": Listing(
        symbol="PRODUCT1",
        product="PRODUCT1",
        denomination= "SEASHELLS"
    ),
    "PRODUCT2": Listing(
```

```

        symbol="PRODUCT2",
        product="PRODUCT2",
        denomination= "SEASHELLS"
    ),
}

order_depths = {
    "PRODUCT1": OrderDepth(
        buy_orders={10: 7, 9: 5},
        sell_orders={11: -4, 12: -8}
    ),
    "PRODUCT2": OrderDepth(
        buy_orders={142: 3, 141: 5},
        sell_orders={144: -5, 145: -8}
    ),
}

own_trades = {
    "PRODUCT1": [],
    "PRODUCT2": []
}

market_trades = {
    "PRODUCT1": [
        Trade(
            symbol="PRODUCT1",
            price=11,
            quantity=4,
            buyer="",
            seller="",
            timestamp=900
        )
    ],
    "PRODUCT2": []
}

```

```

position = {
    "PRODUCT1": 3,
    "PRODUCT2": -5
}

observations = {}
traderData = ""

state = TradingState(
    traderData,
    timestamp,
    listings,
    order_depths,
    own_trades,
    market_trades,
    position,
    observations
)

```

Let's say that at this point in the simulation, the algorithm has the following convictions:

1. PRODUCT1 is worth 13
2. PRODUCT2 is worth 142

It could then potentially decide on the following

1. Since the sell orders in PRODUCT1 at price 11 (qty 4) and price 12 (qty 8) are both below the algorithms calculated fair value, it would like to send a buy order to trade with these sell orders. Given that the position in PRODUCT2 is already 3 long and the position limit is set at 10, sending one or more buy orders with an aggregated quantity of >7 would result in rejection of all buy orders, the algorithm sends a buy order with quantity 7 and a price of 12.
2. Versus the fair value of 142 neither the sell orders nor the buy orders look profitable. The algorithm therefore decides to see if any of the bots is willing to buy at 143, and sends a sell order of quantity 5 at that price level.

Based on the above the `run` method output would then be generated as shown below:

```
result["PRODUCT1"] = [Order("PRODUCT1", 12, 7)]
result["PRODUCT2"] = [Order("PRODUCT2", 143, -5)]
```

An example of what the next `TradingState` will look like is generated by the code below.

```
from datamodel import Listing, OrderDepth, Trade, TradingState

timestamp = 1100

listings = {
    "PRODUCT1": Listing(
        symbol="PRODUCT1",
        product="PRODUCT1",
        denomination: "SEASHELLS"
    ),
    "PRODUCT2": Listing(
        symbol="PRODUCT2",
        product="PRODUCT2",
        denomination: "SEASHELLS"
    ),
}

order_depths = {
    "PRODUCT1": OrderDepth(
        buy_orders={10: 7, 9: 5},
        sell_orders={12: -5, 13: -3}
    ),
    "PRODUCT2": OrderDepth(
        buy_orders={142: 3, 141: 5},
        sell_orders={144: -5, 145: -8}
    ),
}
```

```

own_trades = {
    "PRODUCT1": [
        Trade(
            symbol="PRODUCT1",
            price=11,
            quantity=4,
            buyer="SUBMISSION",
            seller="",
            timestamp=1000
        ),
        Trade(
            symbol="PRODUCT1",
            price=12,
            quantity=3,
            buyer="SUBMISSION",
            seller="",
            timestamp=1000
        )
    ],
    "PRODUCT2": [
        Trade(
            symbol="PRODUCT2",
            price=143,
            quantity=2,
            buyer="",
            seller="SUBMISSION",
            timestamp=1000
        ),
    ]
}

market_trades = {
    "PRODUCT1": [],
    "PRODUCT2": []
}

```

```

position = {
    "PRODUCT1": 10,
    "PRODUCT2": -7
}

observations = {}
traderData = ""

state = TradingState(
    traderData,
    timestamp,
    listings,
    order_depths,
    own_trades,
    market_trades,
    position,
    observations
)

```

A few observations can be made from this `TradingState` 's properties:

1. The algorithm's buy orders for "PRODUCT1" matched first with the full quantity of the sell order at price 11. As a result the order is now gone from the `order_depths` and a corresponding `own_trade` is created.
2. The remaining order quantity of 3 then matched with part of the order at price 12, resulting in a second trade at price 12. As can be seen in the `order_depths` the quantity of the corresponding sell order is now reduced by 3 as well.
3. For "PRODUCT2" a trade at price 143 with a quantity 2 can be observed, which indicates that one of the bots decided to send a buy order of quantity 2 as a reaction to the player's sell order at that price level. For the player's initial order this means that a quantity of 3 of the 5 total remains unexecuted. None of the bots decides to trade against this order, and the full order quantity is automatically cancelled.



# Technical Notes

There are a few technicalities that players should take into account when writing an algorithms:

1. Only the libraries noted at the bottom of this page under “Supported Libraries” are allowed to be used by the algorithm. These are listed in Appendix C.
2. Each time the “run” method is called, it should generate a response in <900ms, otherwise the function call will time out. Players should make sure that their algorithms are sufficiently lightweight to make sure this requirement is met.

## Available Resources to Help Build the Algorithm


To aid players in building the algorithm several resources are made available:

1. For every new product introduced several days of sample data are provided. For each of these days two .csv's are available, one containing a list of all the trades done on that day, and one showing the market orders at every time step. Examples of the file formats:
  - a. .csv file with trade example
  - b. .csv file with market orders example
2. When players upload their algorithms on the Prosperity platform, the algorithm is tested for 1000 iterations using data from a sample day (different than the actual day that will be used for the challenge). After the run a log file is provided which can aid players in debugging their algorithms. To aid debugging, the log file also contains the output of any print statements that players put within the `run` method of their trading class.

## Appendix A: `Trader` Class Example

Below an example of an implementation of the `Trader` class is provided. While very simple and likely not profitable, this example algorithm does include all the

necessary logic to send orders for the first day's "PEARLS" product (see the page that describes the first round).

 Download

[example-program.zip](#)

```
from datamodel import OrderDepth, UserId, TradingState, Order
from typing import List
import string

class Trader:

    def run(self, state: TradingState):
        # Only method required. It takes all buy and sell orders for all symbols as an
        print("traderData: " + state.traderData)
        print("Observations: " + str(state.observations))
        result = {}
        for product in state.order_depths:
            order_depth: OrderDepth = state.order_depths[product]
            orders: List[Order] = []
            acceptable_price = 10; # Participant should calculate this value
            print("Acceptable price : " + str(acceptable_price))
            print("Buy Order depth : " + str(len(order_depth.buy_orders)) + ", Sell orde

            if len(order_depth.sell_orders) != 0:
                best_ask, best_ask_amount = list(order_depth.sell_orders.items())[0]
                if int(best_ask) < acceptable_price:
                    print("BUY", str(-best_ask_amount) + "x", best_ask)
                    orders.append(Order(product, best_ask, -best_ask_amount))

            if len(order_depth.buy_orders) != 0:
                best_bid, best_bid_amount = list(order_depth.buy_orders.items())[0]
                if int(best_bid) > acceptable_price:
```

```

        print("SELL", str(best_bid_amount) + "x", best_bid)
        orders.append(Order(product, best_bid, -best_bid_amount))

    result[product] = orders

    traderData = "SAMPLE" # String value holding Trader state data required. It

    conversions = 1
    return result, conversions, traderData

```

## Appendix B: datamodel.py file

```

import json
from typing import Dict, List
from json import JSONEncoder
import jsonpickle

Time = int
Symbol = str
Product = str
Position = int
UserId = str
ObservationValue = int

class Listing:

    def __init__(self, symbol: Symbol, product: Product, denomination: Product):
        self.symbol = symbol
        self.product = product
        self.denomination = denomination

```

```
class ConversionObservation:
```

```
    def __init__(self, bidPrice: float, askPrice: float, transportFees: float, exportTariff: float, importTariff: float, sugarPrice: float, sunlightIndex: float):
        self.bidPrice = bidPrice
        self.askPrice = askPrice
        self.transportFees = transportFees
        self.exportTariff = exportTariff
        self.importTariff = importTariff
        self.sugarPrice = sugarPrice
        self.sunlightIndex = sunlightIndex
```

```
class Observation:
```

```
    def __init__(self, plainValueObservations: Dict[Product, ObservationValue], conversionObservations: Dict[Product, ConversionObservation]):
        self.plainValueObservations = plainValueObservations
        self.conversionObservations = conversionObservations
```

```
    def __str__(self) → str:
        return "(plainValueObservations: " + jsonpickle.encode(self.plainValueObservations) + ", conversionObservations: " + jsonpickle.encode(self.conversionObservations) + ")"
```

```
class Order:
```

```
    def __init__(self, symbol: Symbol, price: int, quantity: int) → None:
        self.symbol = symbol
        self.price = price
        self.quantity = quantity
```

```
    def __str__(self) → str:
        return "(" + self.symbol + ", " + str(self.price) + ", " + str(self.quantity) + ")"
```

```
    def __repr__(self) → str:
        return "(" + self.symbol + ", " + str(self.price) + ", " + str(self.quantity) + ")"
```

```
class OrderDepth:
```

```
    def __init__(self):
        self.buy_orders: Dict[int, int] = {}
        self.sell_orders: Dict[int, int] = {}
```

```
class Trade:
```

```
    def __init__(self, symbol: Symbol, price: int, quantity: int, buyer: UserId=None, seller: UserId=None, timestamp: Time):
        self.symbol = symbol
        self.price: int = price
        self.quantity: int = quantity
        self.buyer = buyer
        self.seller = seller
        self.timestamp = timestamp
```

```
    def __str__(self) → str:
        return "(" + self.symbol + ", " + self.buyer + " << " + self.seller + ", " + str(self.timestamp) + ")"
```

```
    def __repr__(self) → str:
        return "(" + self.symbol + ", " + self.buyer + " << " + self.seller + ", " + str(self.timestamp) + ")"
```

```
class TradingState(object):
```

```
    def __init__(self,
        traderData: str,
        timestamp: Time,
        listings: Dict[Symbol, Listing],
        order_depths: Dict[Symbol, OrderDepth],
        own_trades: Dict[Symbol, List[Trade]],
        market_trades: Dict[Symbol, List[Trade]],
        position: Dict[Product, Position],
        observations: Observation):
        self.traderData = traderData
```

```
self.timestamp = timestamp
self.listings = listings
self.order_depths = order_depths
self.own_trades = own_trades
self.market_trades = market_trades
self.position = position
self.observations = observations

def toJSON(self):
    return json.dumps(self, default=lambda o: o.__dict__, sort_keys=True)

class ProsperityEncoder(JSONEncoder):

    def default(self, o):
        return o.__dict__
```

## Appendix C: Supported libraries

The following libraries are supported in the simulation. Importing other external libraries is not supported.

pandas

NumPy

statistics

math

typing

jsonpickle