mTree

Release 1.0

CSN

Apr 20, 2022

CONTENTS

Welcome to the **mTree** documentation! **mTree** is a Agent-Based Modelling software in python. If this is your first time interacting with **mTree** follow the **Novice path** mentioned below in order to properly install and test the software.

CHAPTER

ONE

NOVICE PATH

Do this step by step

1. *Installing Docker Desktop*
2. *Installing mTree*
3. *Quick Start Guide*
4. *Cloning mTree_auction_examples*
5. *Running mTree_auction_examples*
    a. *mTree_auction_examples container setup*
        - *mTree Container Setup*
    b. *Running mTree_auction_examples container*
        - *Start*
        - *Open Shell*
    c. *Common Value Auction*
    d. *File Structure*
    e. *Running common_value_auction simulation*
    f. *How to know your simulation has finished running?*
    g. *Simulation Results*
    h. *Checking for Errors*
    i. *Quitting*
    j. *Conclusion*

CHAPTER

TWO

CONTENTS

2.1 Installation

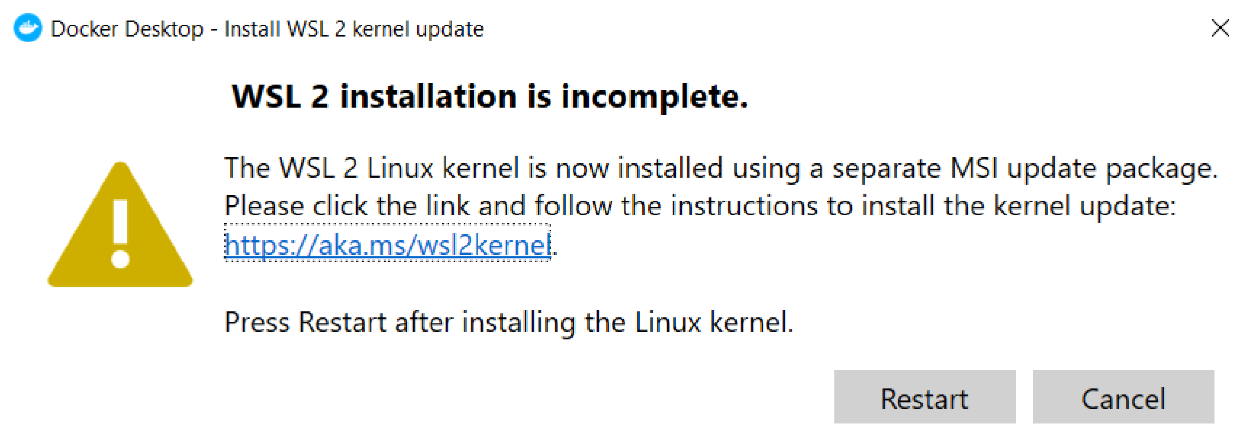
In order to run **mTree** we need to install Docker Desktop first.

2.1.1 Installing Docker Desktop

The links for **Docker Desktop** installation for different **os** can be found below.

- Download Docker Desktop

Tip: If you have a **Windows** machine and end up at the following prompt after installing **Docker Desktop** -



Docker Desktop - Install WSL 2 kernel update

WSL 2 installation is incomplete.

The WSL 2 Linux kernel is now installed using a separate MSI update package. Please click the link and follow the instructions to install the kernel update: https://aka.ms/wsl2kernel.

Press Restart after installing the Linux kernel.

Fig. 1: WLS 2 Installation Incomplete

Visit the link and only complete **Step 4** from the webpage

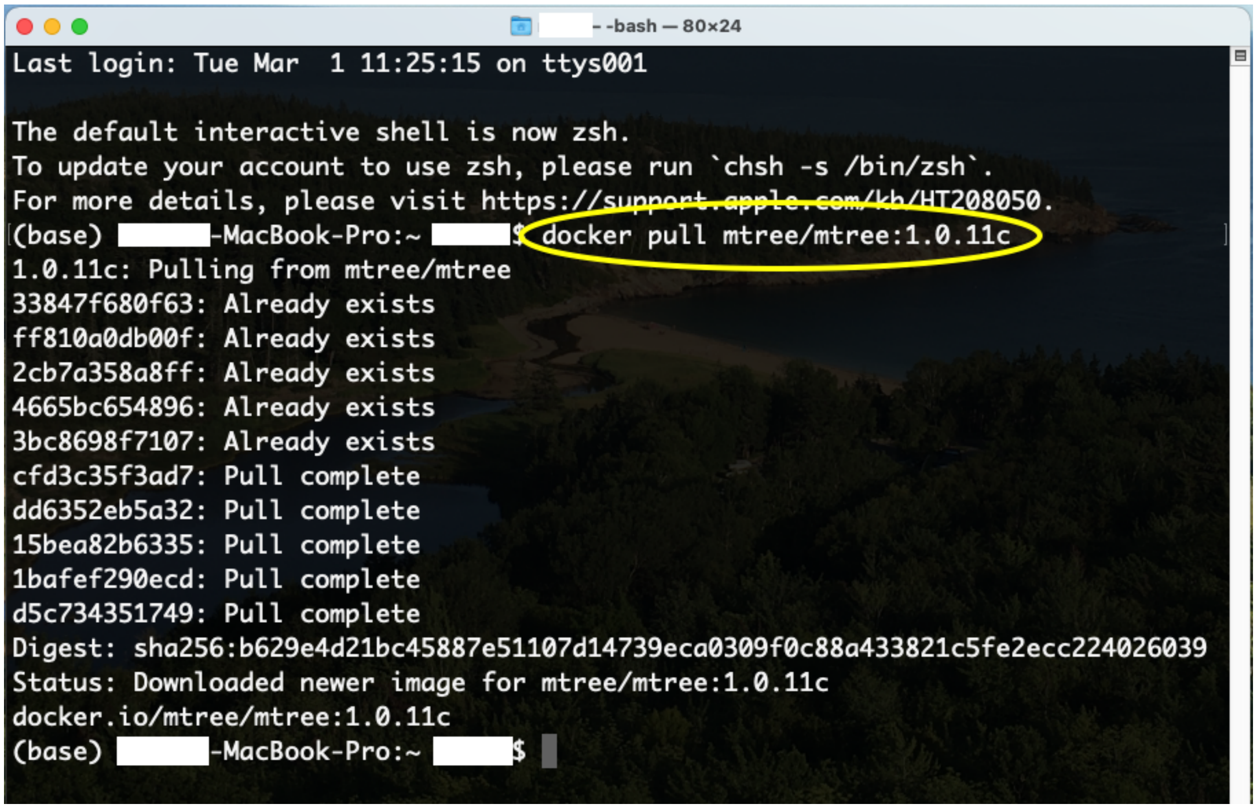
After completing the **Docker Desktop** installation, we can start installing **mTree**

2.1.2 Installing mTree

You can install **mTree** by pasting the following code in your Command Prompt, PowerShell or Terminal. However, make sure to check for the latest version of **mTree** here, in case the one below is out of date.

```
docker pull mtree/mtree:1.0.11c
```

After pasting and running the command in your Command Prompt, your screen should look like something like this



```
Last login: Tue Mar  1 11:25:15 on ttys001

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) [redacted]-MacBook-Pro:~ [redacted]$ docker pull mtree/mtree:1.0.11c
1.0.11c: Pulling from mtree/mtree
33847f680f63: Already exists
ff810a0db00f: Already exists
2cb7a358a8ff: Already exists
4665bc654896: Already exists
3bc8698f7107: Already exists
cfd3c35f3ad7: Pull complete
dd6352eb5a32: Pull complete
15bea82b6335: Pull complete
1bafef290ecd: Pull complete
d5c734351749: Pull complete
Digest: sha256:b629e4d21bc45887e51107d14739eca0309f0c88a433821c5fe2ecc224026039
Status: Downloaded newer image for mtree/mtree:1.0.11c
docker.io/mtree/mtree:1.0.11c
(base) [redacted]-MacBook-Pro:~ [redacted]$
```

Fig. 2: Your Command Prompt/Terminal/PowerShell after docker pull command

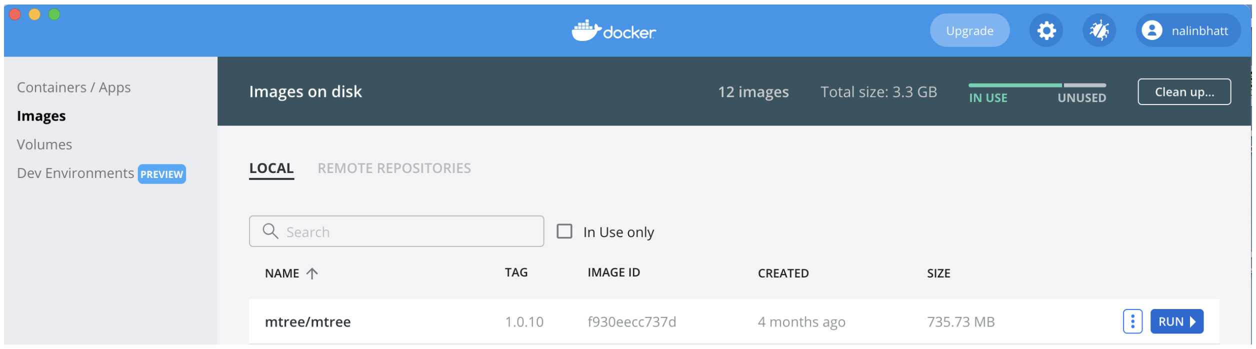
Note: If you get an **ERROR** message in your **Command Prompt** try running the command after starting the **Docker Desktop** App and see if that helps.

Note: It is important that your **Command Prompt** is based in the same virtual environment where you have **Docker Desktop** installed in order for the docker pull command to work. If you don't understand what this means, you don't have to worry about this.

2.1.3 mTree Container Setup

Tip: If you don't have an **mTree** simulation that is ready to run or you are new to **mTree**, visit the *Quick Start Guide* before you do this next step.

Open **Docker Desktop** app on your computer and click **Images** on the sidebar.



NAME	TAG	IMAGE ID	CREATED	SIZE
mtree/mtree	1.0.10	f930eecc737d	4 months ago	735.73 MB

Fig. 3: **Images** Section on the **Docker Desktop** App

You should see the **mTree** image we just downloaded through docker hub in the previous step. In the next step, we are going to run this image within a small virtualization of the **os** called a **container**. We can create our docker **container** by clicking **RUN** on the **mTree** image.

After that you should see the following window. Follow all the steps, in the image below, before moving on to the next step.

Once all the instructions in the above image are completed, you should click **Containers/Apps** on the sidebar. After hitting **Containers/Apps**, you should see the following container -

Container Options

Your container comes with several options that can be executed to change its state.

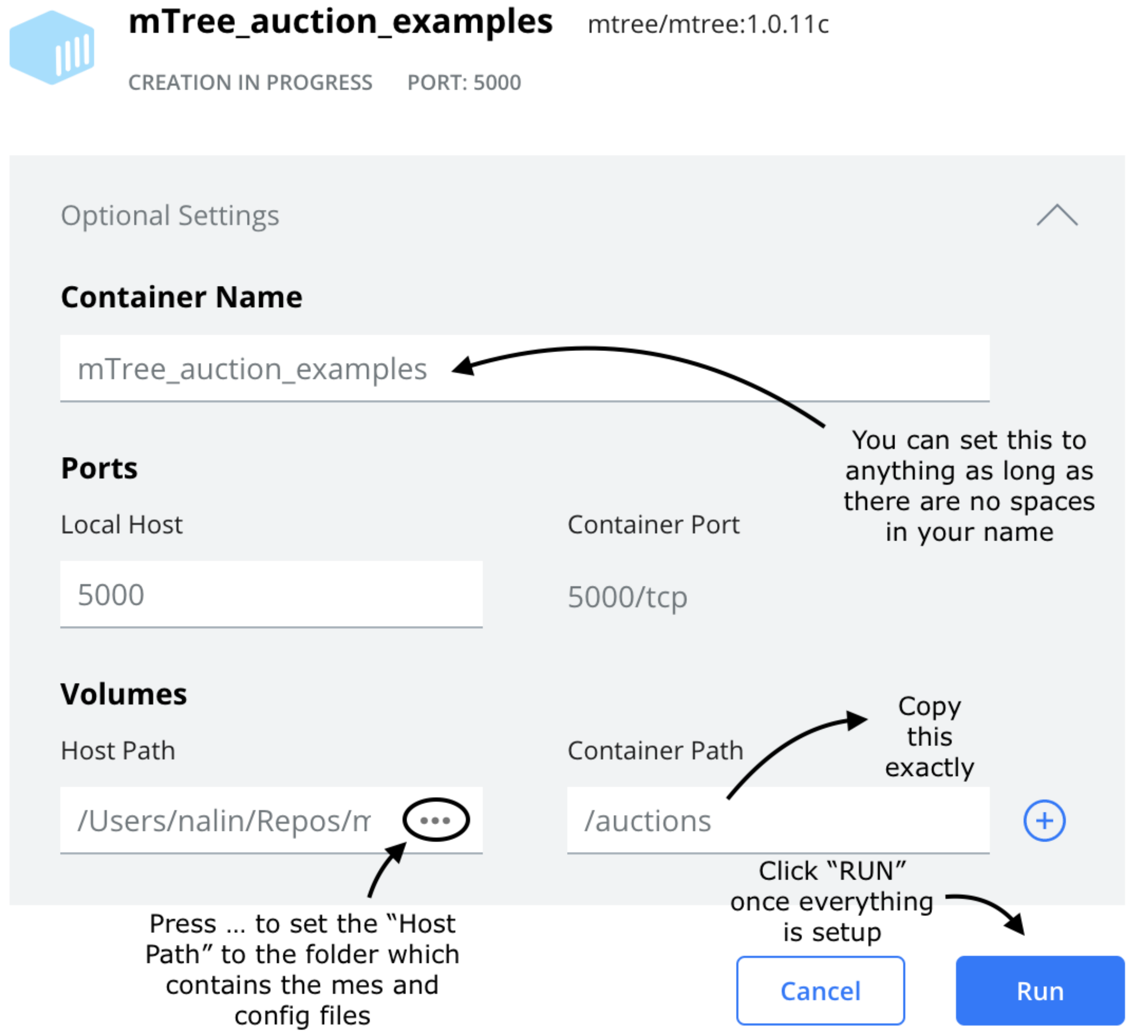
Start

Click `START` to start your container.

A running docker container should have a green symbol on the left side.

Stop

You can stop running your container by pressing `STOP` button



mTree_auction_examples mtree/mtree:1.0.11c

CREATION IN PROGRESS PORT: 5000

Optional Settings

Container Name

mTree_auction_examples

You can set this to anything as long as there are no spaces in your name

Ports

Local Host	Container Port
5000	5000/tcp

Volumes

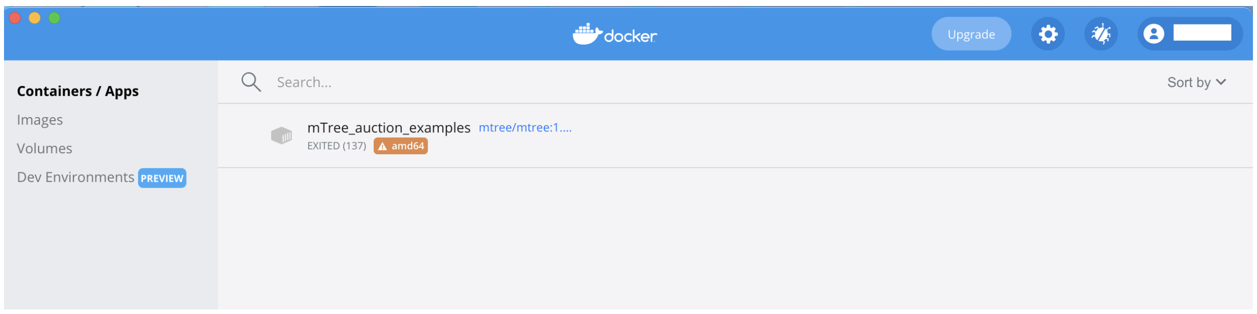
Host Path	Container Path
/Users/nalin/Repos/m	/auctions

Copy this exactly

Press ... to set the "Host Path" to the folder which contains the mes and config files

Click "RUN" once everything is setup

Fig. 4: **mTree** container setup



mTree_auction_examples mtree/mtree:1....
EXITED (137) amd64

Fig. 5: **Container/Apps** Section on the **Docker Desktop** App

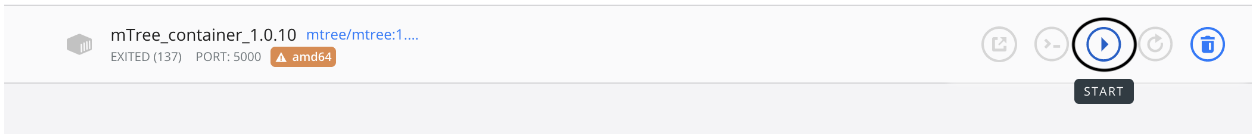

mTree_container_1.0.10 mtree/mtree:1....
EXITED (137) PORT: 5000 amd64
START

Fig. 6: **START** button

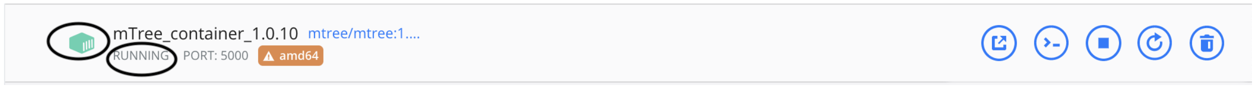


Fig. 7: A running docker container

Restart

You can restart your container by pressing the `RESTART` button

Delete

If you want to delete the image, you can press the `DELETE` button

Open Shell

Once your container is running, you should click `CLI` button to open the **Command Prompt/ shell** linked to your container.

The **shell** produced by **Docker** should look similar to the following -

2.2 Quick Start Guide

In this **Quick Start Guide**, we are going to run a simple **mTree simulation** while giving an overview of key components that are necessary for **mTree** to execute without error. The goal of this guide is to show you how to run an **mTree simulation** from start to finish and point out the vital indicators that convey a simulation has run properly.

In order to complete this **Quick Start Guide**, you would need to the following installed on your computer before you can begin

1. **Docker Desktop** - The quick start guide assumes that you have finished *Installing Docker Desktop*
2. **Latest mTree Image** - This should be covered in the *Installing mTree* section.
3. **Git - We are going to use git to run a simple mTree simulation later in this section. A simple way to check if you have git i**

   - ***Mac Users***

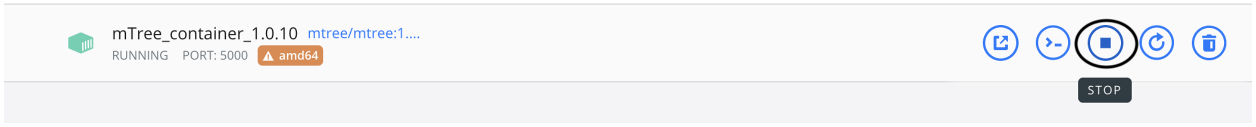


Fig. 8: **STOP** button

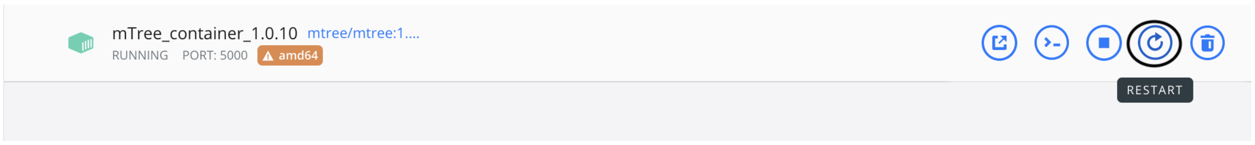


Fig. 9: **RESTART** button

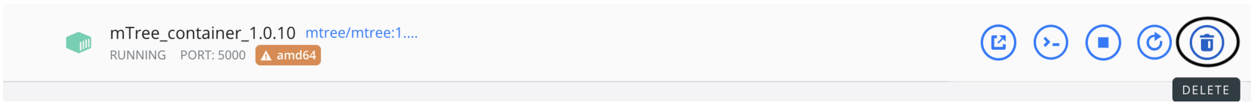


Fig. 10: **DELETE** button

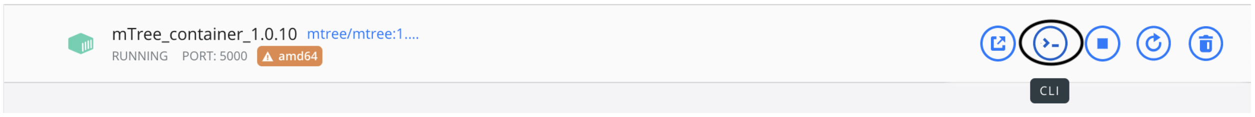
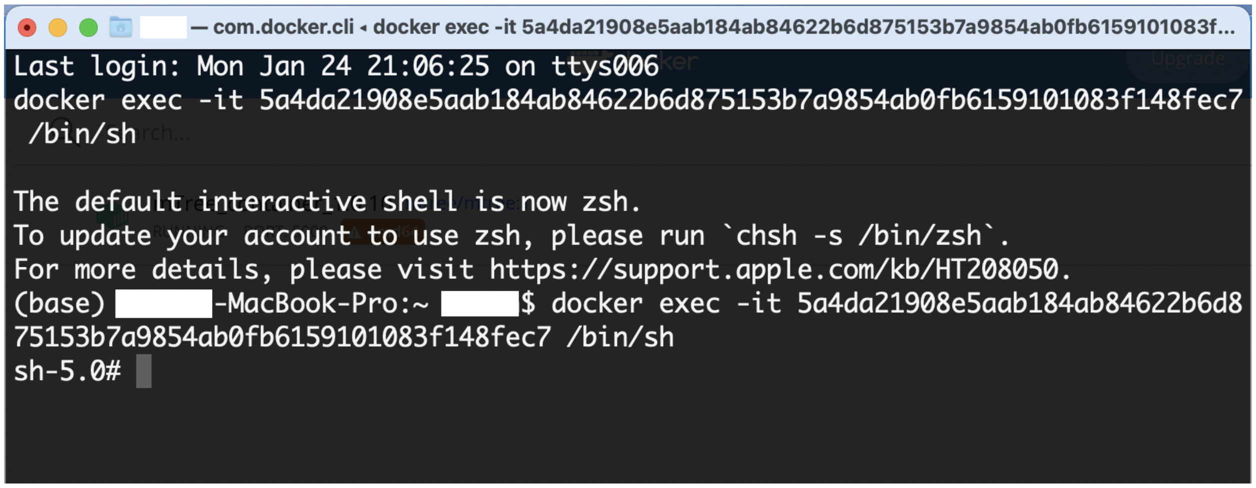


Fig. 11: **CLI** button



```
[redacted] — com.docker.cli ◂ docker exec -it 5a4da21908e5aab184ab84622b6d875153b7a9854ab0fb6159101083f...
Last login: Mon Jan 24 21:06:25 on ttys006
docker exec -it 5a4da21908e5aab184ab84622b6d875153b7a9854ab0fb6159101083f148fec7
 /bin/sh

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) [redacted]-MacBook-Pro:~ [redacted]$ docker exec -it 5a4da21908e5aab184ab84622b6d8
75153b7a9854ab0fb6159101083f148fec7 /bin/sh
sh-5.0#
```

Fig. 12: **docker shell**

  - If your Terminal says it doesn't recognize the command, which is very unlikely, visit the Git Download for macOS website to download git.
  - You'll have several options on how to install **git**, however, it is recommended to use **homebrew** route, check out this git homebrew download video on how to do this.
- ***Windows Users***
  - If it doesn't recognize the command, suggesting you don't have git, visit the Git Download for Windows website and follow the directions highlighted in this windows git download video .

4. **VSCode** - We recommend using an Integrated Development Environment (**IDE**) to edit and view **mTree** simulation code. Although, VSCode is versatile and great, however, any **IDE** of your choice should also work.

2.2.1 Cloning mTree_auction_examples

We are going to clone the mTree_auction_examples repository and run one of the examples to make sure **mTree** is running properly.

Open your Command Prompt and navigate to an apporpriate place within your file system using the `cd` command and run the following code. If you have not used **Command Line** before you can check out the **tip** below or you could simply run the following code and it will create an **mTree_auction_examples** folder in your **home directory** where the **Desktop** folder exists.

```
git clone https://github.com/nalinbhatt/mTree_auction_examples.git
```

This will create an **mTree_auction_examples** folder at your specified location.

Tip: If you are new to **command line** you can check out the following links on how to navigate your file system -

Terminal for Beginners (Macs)
A Beginner's Guide to the Windows Command Prompt (Windows)

Although, the following resources provide a great background which might be helpful later, nevertheless, for these next few steps, you only need to know how the `cd` command works.

2.2.2 Running mTree_auction_examples

In order to run this simulation we need to create a **docker container** using the **Docker Desktop** app that we downloaded in *Installation* section.

mTree_auction_examples container setup

Follow all the steps highlighted in the *mTree Container Setup* section and set the **Host Path** to the **mTree_auction_example folder** (which you cloned in the previous step).

- If you installed **mTree_auction_example** by navigating to somewhere in your file system, you are going to have to locate your folder in finder window by reviewing the steps you took.
- If you did a simple **git clone** without ever using the `cd` command then you need navigate to your **home folder** (the folder which contains your Desktop) and select the **mTree_auction_examples** folder.

After finishing the setup process, click **Container/Apps** on the sidebar of **Docker Desktop**. There should be a container by the name **mTree_auction_examples** present.

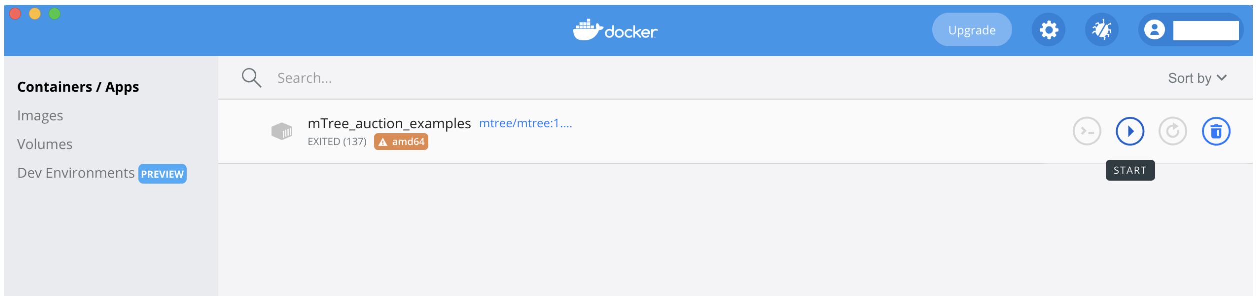
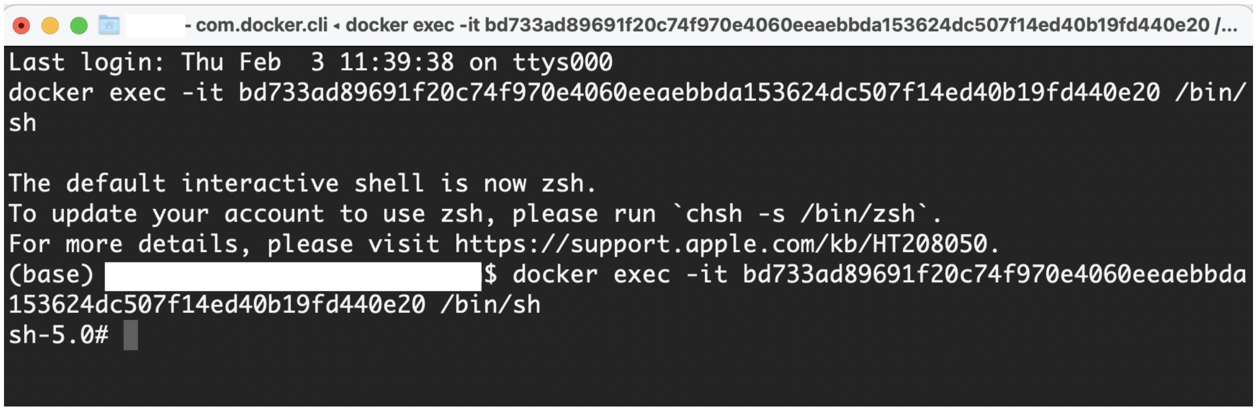


Fig. 13: Your **Containers/Apps** section should display a container similar to this with the name you chose

Running mTree_auction_examples container

Start the container and open the **shell**. More details on how to do this are covered in *Container Options* under *Start* and *Open Shell*.

Your **shell** should look some version of this -



```
[redacted] - com.docker.cli ◂ docker exec -it bd733ad89691f20c74f970e4060eeaebbda153624dc507f14ed40b19fd440e20 /...
Last login: Thu Feb  3 11:39:38 on ttys000
docker exec -it bd733ad89691f20c74f970e4060eeaebbda153624dc507f14ed40b19fd440e20 /bin/
sh

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) [redacted]$ docker exec -it bd733ad89691f20c74f970e4060eeaebbda
153624dc507f14ed40b19fd440e20 /bin/sh
sh-5.0#
```

Fig. 14: mTree_auction_examples shell produced by clicking the CLI button

Run the following commands to view the underlying files in the folder.

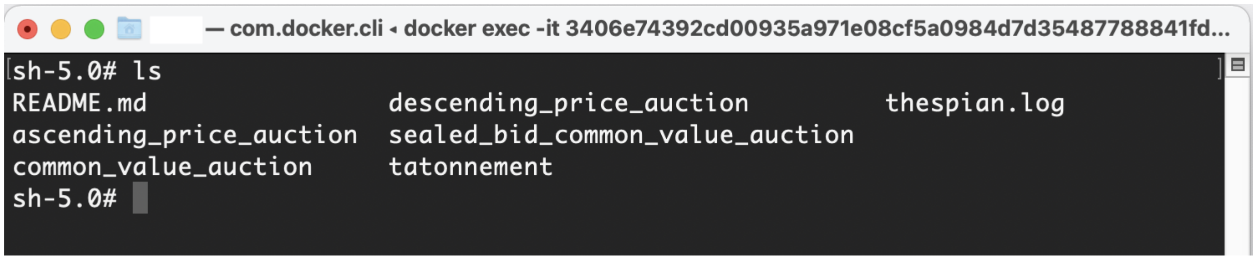
Mac

```
ls
```

Windows

```
dir
```

You should see the following subfolders-



— com.docker.cli ◂ docker exec -it 3406e74392cd00935a971e08cf5a0984d7d35487788841fd...

```
sh-5.0# ls
README.md                descending_price_auction        thespian.log
ascending_price_auction  sealed_bid_common_value_auction
common_value_auction     tatonnement
sh-5.0# 
```

Fig. 15: Folders inside mTree_auction_examples

Common Value Auction

One of the subfolders present should have the name **common_value_auction**. Further information about the auction style and description can be found in the *Common Value Auction* section of *Learning Paths*.

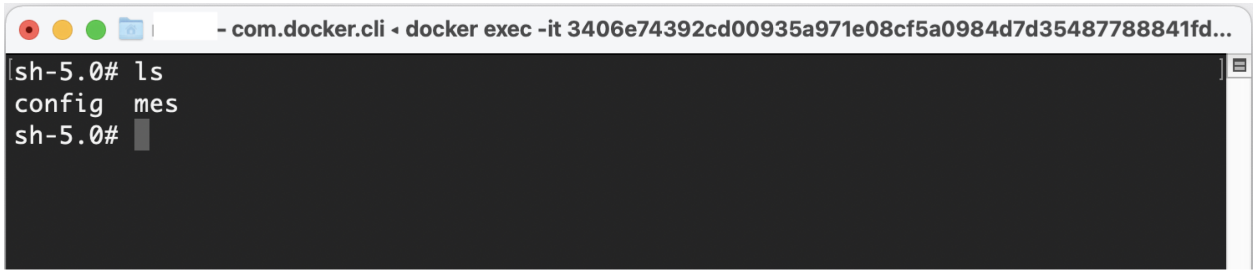
In your **mTree_auction_examples** container **shell** type in the following command to set the current directory to **common_value_auction**.

```
cd common_value_auction
```

File Structure

After setting **common_value_auction** as the current directory, run **ls** or **dir** and you should see the following folders.

1. *config*
2. *mes*
3. *logs*



- com.docker.cli ◂ docker exec -it 3406e74392cd00935a971e08cf5a0984d7d35487788841fd...

```
sh-5.0# ls
config  mes
sh-5.0# 
```

Fig. 16: Folders inside common_value_auction

Note: In order to properly run an **mTree simulation** you need to set the current directory to the folder which contains a **config**, **mes**, and a **logs** folder. **mTree** looks for these particular folders to run the simulation. For our example, this is the **common_value_auction** folder inside **mTree_auction_examples**.

Tip: In the future, when designing your own container, you can set the **Host Path** directly to the folder containing the **config** and **mes** folder. That way you don't have to navigate to the desired directory within the docker **shell**.

The *config folder* folder (short for configurations) contains your **JSON config files** which are used to instantiate **mTree** *Actors* defined in the **mes** folder.

The **mes** folder (short for Microeconomic System) containes the python files where you define the different *Actor* classes, namely - the *Environment* , *Institution* and *Agent*.

Warning: It is critical that your **simulation folder** contains a **config** folder, with a **JSON config file** inside, and a separate **mes** folder with python files inside, which contain *Environment* , *Institution* and *Agent* code. **In the absence of any of these your mTree simulation will not run.**

Inside the **config** folder in the **common_value_auction** auction example, you should see a **basic_simulation.json** file. This is the config file which we will run.

For the next step we want to make sure that our current directory is **common_value_auction** so if you used the **cd** command to change the directory to **config** and view its contents, we want to go up a directory using the following command to make sure we are in the right directory.

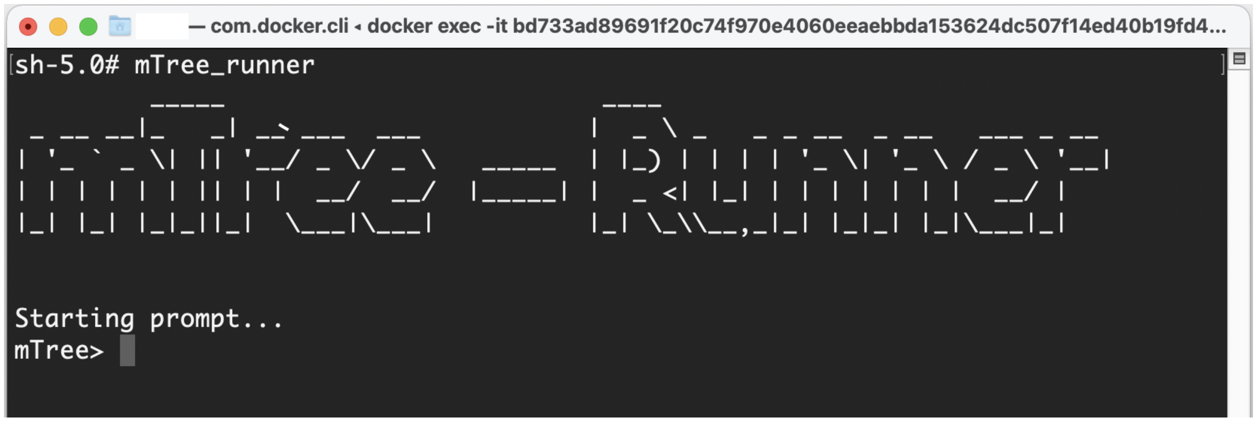
```
cd ..
```

Running common_value_auction simulation

We can type the following command into the **shell** to start **mTree**.

```
mTree_runner
```

You should see something similar to this.



— com.docker.cli ◂ docker exec -it bd733ad89691f20c74f970e4060eeaebbda153624dc507f14ed40b19fd4...

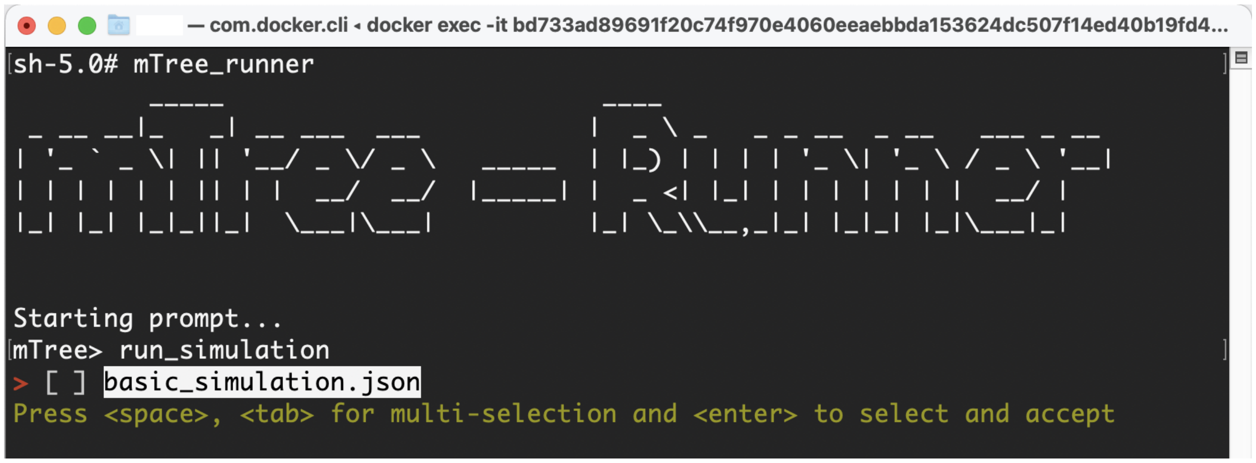
```
sh-5.0# mTree_runner
           _____                          ____
 _ __ ___ |_   _| __ ___  ___            |  _ \ _   _ _ __  _ __   ___ _ __
| '_ ` _ \  | || '__/ _ \/ _ \   _____   | |_) | | | | '_ \| '_ \ / _ \ '__|
| | | | | | | || | |  __/  __/  |_____|  |  _ <| |_| | | | | | | |  __/ |
|_| |_| |_| |_||_|  \___|\___|           |_| \_\\__,_|_| |_|_| |_|\___|_|


Starting prompt...
mTree>
```

Fig. 17: mTree_runner window

Enter the following to start the selection process for the config file.

```
run_simulation
```

Your window should look like this.



```
— com.docker.cli ◂ docker exec -it bd733ad89691f20c74f970e4060eeaebbda153624dc507f14ed40b19fd4...
sh-5.0# mTree_runner
            _____                       ____
  _ __ ___ |_   _| __ ___  ___         |  _ \ _   _ _ __  _ __   ___ _ __
 | '_ ` _ \  | || '__/ _ \/ _ \  _____ | |_) | | | | '_ \| '_ \ / _ \ '__|
 | | | | | | | || | |  __/  __/ |_____||  _ <| |_| | | | | | | |  __/ |
 |_| |_| |_| |_||_|  \___|\___|        |_| \_\\__,_|_| |_|_| |_|\___|_|


Starting prompt...
mTree> run_simulation
> [ ] basic_simulation.json
Press <space>, <tab> for multi-selection and <enter> to select and accept
```

Fig. 18: run_simulation window

Click **<enter>** to select and run the **basic_simulation.json** file. Your output should look something similar to this.

How to know your simulation has finished running?

mTree provides a `check_status` command that allows you to inquire the state of the simulation from the **shell** or **console**. Run the following command in your **shell** to see the state of the simulation. If you wish to know more about this command visit *mTree Simulation State* section.

```
check_status
```

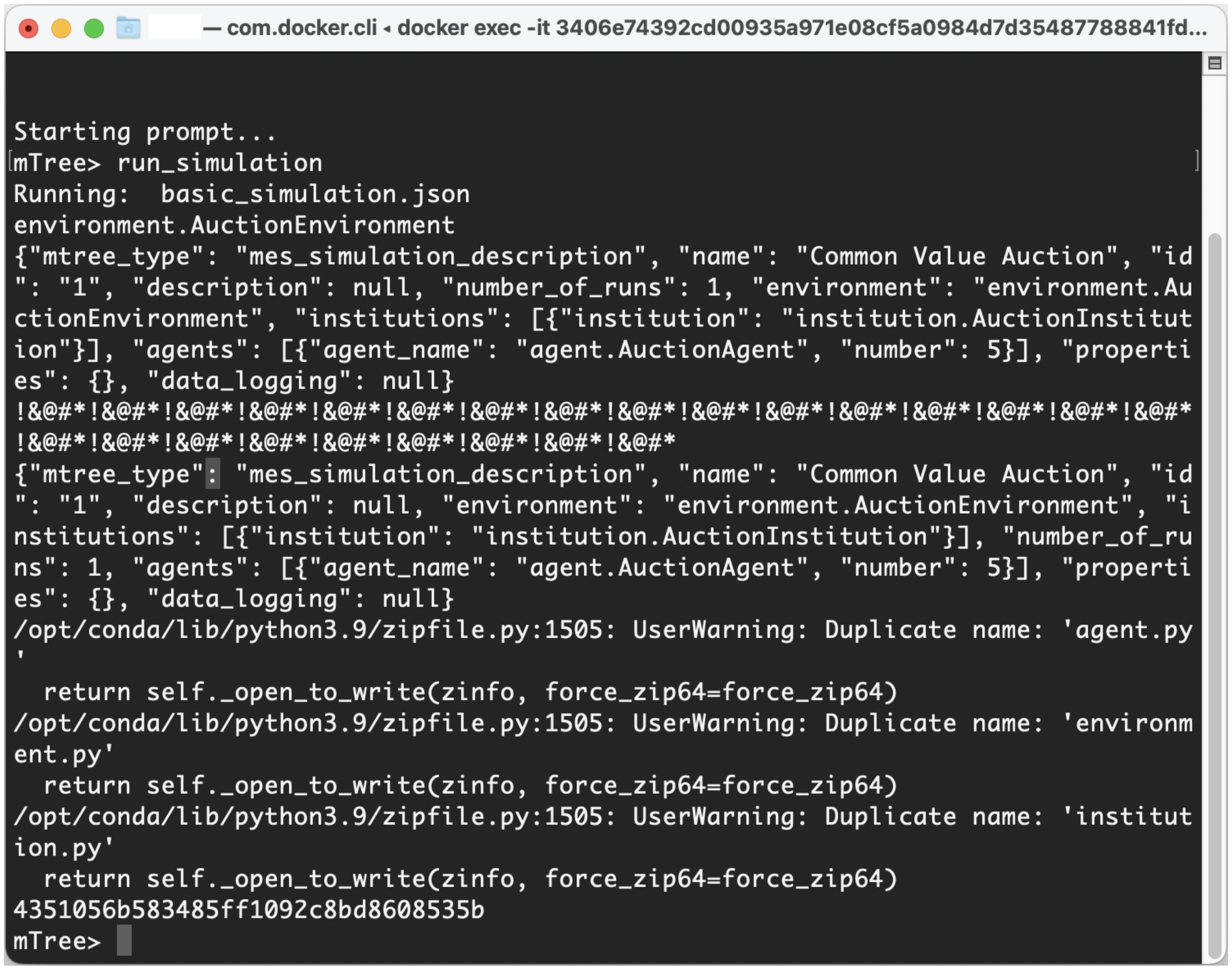
Note: You can enter the `check_status` command multiple times to view the state of your simulation.

Depending on the when you entered the `check_status` command, you should see any one of the following screens.

Once we have identified that our simulation has finished we can move on to the next step which involves

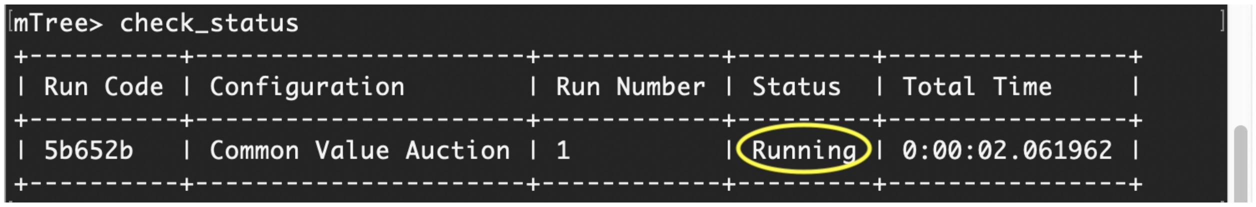
Simulation Results

Ideally when a simulation is run, you should setup *Actors* in such a way that they constantly *log* their states to *.log* and *.data* files. This allows us to analyze how Actors behaved in our system, what decisions they made, and what effects those decisions had on the system as whole.



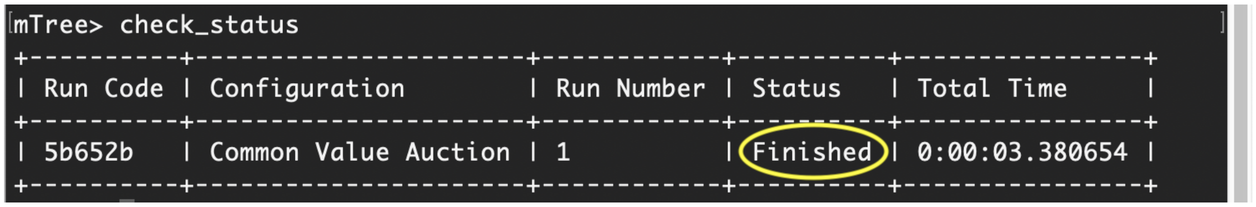
```
— com.docker.cli ‹ docker exec -it 3406e74392cd00935a971e08cf5a0984d7d35487788841fd...

Starting prompt...
mTree> run_simulation
Running:  basic_simulation.json
environment.AuctionEnvironment
{"mtree_type": "mes_simulation_description", "name": "Common Value Auction", "id
": "1", "description": null, "number_of_runs": 1, "environment": "environment.Au
ctionEnvironment", "institutions": [{"institution": "institution.AuctionInstitut
ion"}], "agents": [{"agent_name": "agent.AuctionAgent", "number": 5}], "properti
es": {}, "data_logging": null}
!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*
!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*!&@#*
{"mtree_type": "mes_simulation_description", "name": "Common Value Auction", "id
": "1", "description": null, "environment": "environment.AuctionEnvironment", "i
nstitutions": [{"institution": "institution.AuctionInstitution"}], "number_of_ru
ns": 1, "agents": [{"agent_name": "agent.AuctionAgent", "number": 5}], "properti
es": {}, "data_logging": null}
/opt/conda/lib/python3.9/zipfile.py:1505: UserWarning: Duplicate name: 'agent.py
'
  return self._open_to_write(zinfo, force_zip64=force_zip64)
/opt/conda/lib/python3.9/zipfile.py:1505: UserWarning: Duplicate name: 'environm
ent.py'
  return self._open_to_write(zinfo, force_zip64=force_zip64)
/opt/conda/lib/python3.9/zipfile.py:1505: UserWarning: Duplicate name: 'institut
ion.py'
  return self._open_to_write(zinfo, force_zip64=force_zip64)
4351056b583485ff1092c8bd8608535b
mTree>
```

Fig. 19: Running basic_simulation.json file



```
mTree> check_status
+----------+----------------------+------------+---------+----------------+
| Run Code | Configuration        | Run Number | Status  | Total Time     |
+----------+----------------------+------------+---------+----------------+
| 5b652b   | Common Value Auction | 1          | Running | 0:00:02.061962 |
+----------+----------------------+------------+---------+----------------+
```

Fig. 20: This indicates our simulation is still running



```
mTree> check_status
+----------+----------------------+------------+----------+----------------+
| Run Code | Configuration        | Run Number | Status   | Total Time     |
+----------+----------------------+------------+----------+----------------+
| 5b652b   | Common Value Auction | 1          | Finished | 0:00:03.380654 |
+----------+----------------------+------------+----------+----------------+
```

Fig. 21: This indicates our simulation has finished running and we can move to the next step and view our simulation results.

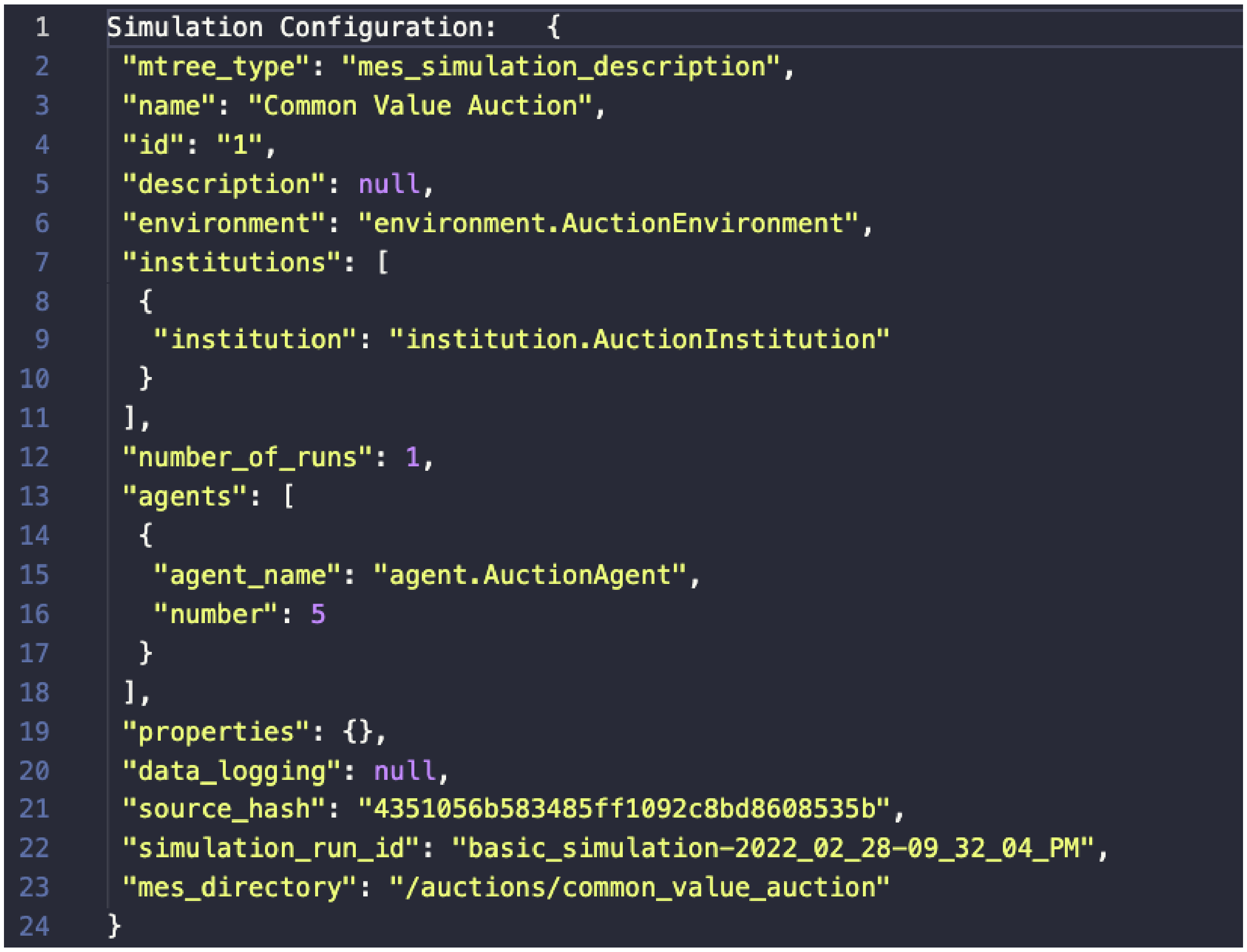
logs

The **logs** folder, inside your simulation folder (which in our case is **common_value_auction**), is where the output from your simulation gets stored. You should see a file ending in `.log` and a file ending in `.data`.

More on how these files are named can be found *here*.

Note: In the figure below, we use VSCode to open the generated **log files**. However, no **IDE** is necessary to open these files and your notepad should also work. That being said, we still advise using an **IDE**, like **VSCode**, to interact with an **mTree simulation**, since they make viewing and editing files of different formats more intuitive.

The first few lines of you `.log` file document the config file parameters which were used to run the simulation



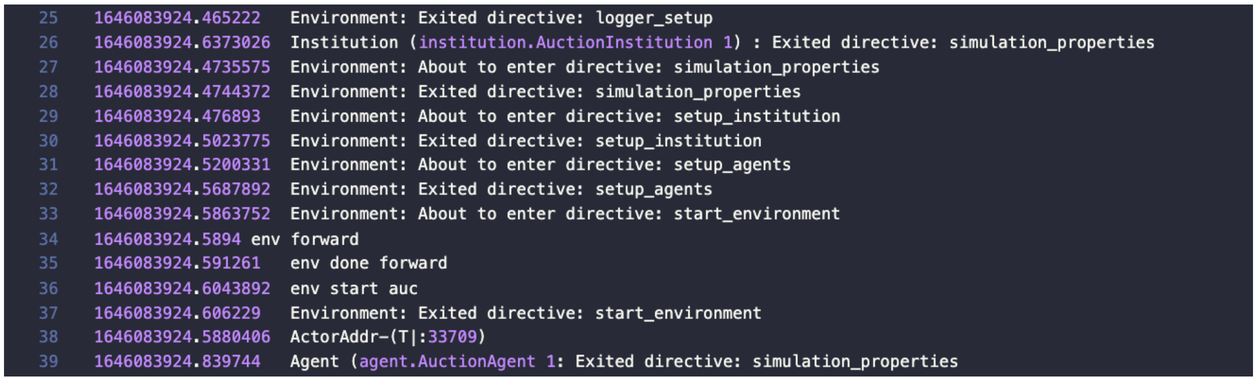
```
Simulation Configuration:   {
 "mtree_type": "mes_simulation_description",
 "name": "Common Value Auction",
 "id": "1",
 "description": null,
 "environment": "environment.AuctionEnvironment",
 "institutions": [
  {
   "institution": "institution.AuctionInstitution"
  }
 ],
 "number_of_runs": 1,
 "agents": [
  {
   "agent_name": "agent.AuctionAgent",
   "number": 5
  }
 ],
 "properties": {},
 "data_logging": null,
 "source_hash": "4351056b583485ff1092c8bd8608535b",
 "simulation_run_id": "basic_simulation-2022_02_28-09_32_04_PM",
 "mes_directory": "/auctions/common_value_auction"
}
```

Fig. 22: basic_simulation-2022_02_28-09_32_04_PM-R1-experiment.log

The rest of your `.log` file should look as follows.

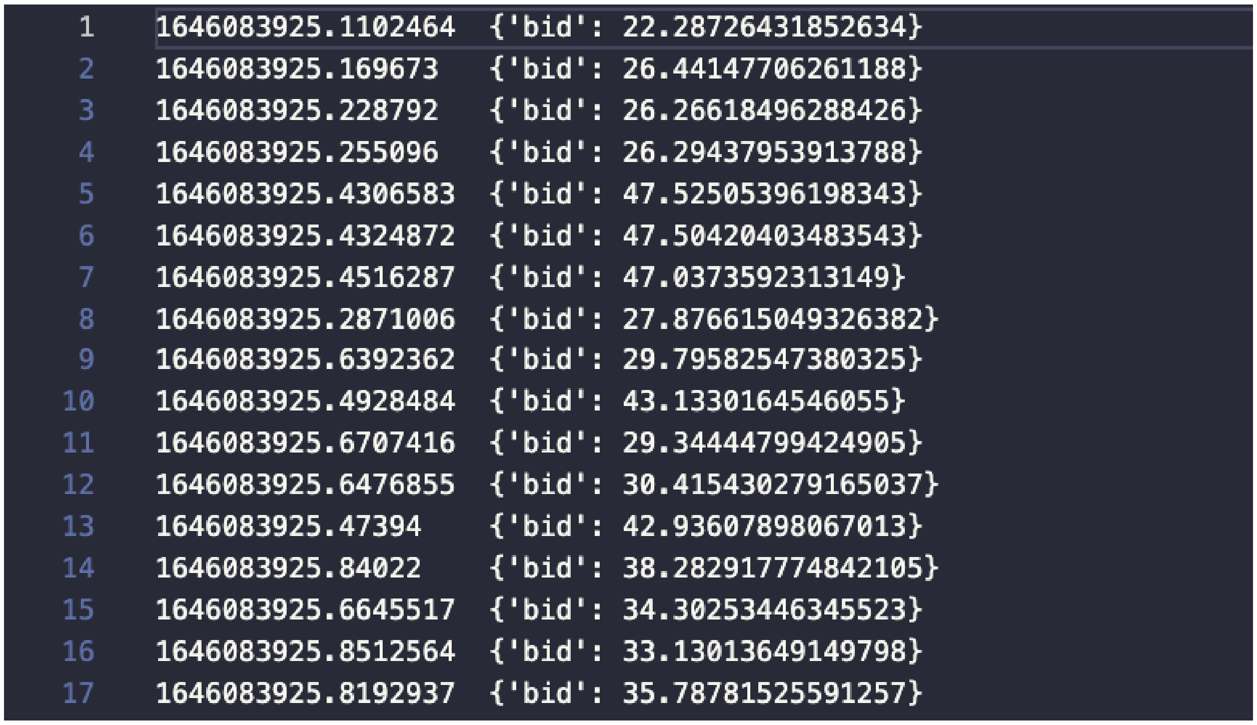
Your `.data` file should look something like this -

Note: Don't worry if the log files on your end don't match the ones shown here word for word. Since **mTree** is a concurrent Agent-Based Modelling software, it is common for different *Actors* to log asynchronously to the same `.log` and `.data` files, giving them an out of order look.



```
1646083924.465222   Environment: Exited directive: logger_setup
1646083924.6373026  Institution (institution.AuctionInstitution 1) : Exited directive: simulation_properties
1646083924.4735575  Environment: About to enter directive: simulation_properties
1646083924.4744372  Environment: Exited directive: simulation_properties
1646083924.476893   Environment: About to enter directive: setup_institution
1646083924.5023775  Environment: Exited directive: setup_institution
1646083924.5200331  Environment: About to enter directive: setup_agents
1646083924.5687892  Environment: Exited directive: setup_agents
1646083924.5863752  Environment: About to enter directive: start_environment
1646083924.5894 env forward
1646083924.591261   env done forward
1646083924.6043892  env start auc
1646083924.606229   Environment: Exited directive: start_environment
1646083924.5880406  ActorAddr-(T|:33709)
1646083924.839744   Agent (agent.AuctionAgent 1: Exited directive: simulation_properties
```

Fig. 23: basic_simulation-2022_02_28-09_32_04_PM-R1-experiment.log



```
1646083925.1102464  {'bid': 22.28726431852634}
1646083925.169673   {'bid': 26.44147706261188}
1646083925.228792   {'bid': 26.26618496288426}
1646083925.255096   {'bid': 26.29437953913788}
1646083925.4306583  {'bid': 47.52505396198343}
1646083925.4324872  {'bid': 47.50420403483543}
1646083925.4516287  {'bid': 47.0373592313149}
1646083925.2871006  {'bid': 27.876615049326382}
1646083925.6392362  {'bid': 29.79582547380325}
1646083925.4928484  {'bid': 43.1330164546055}
1646083925.6707416  {'bid': 29.34444799424905}
1646083925.6476855  {'bid': 30.415430279165037}
1646083925.47394    {'bid': 42.93607898067013}
1646083925.84022    {'bid': 38.282917774842105}
1646083925.6645517  {'bid': 34.30253446345523}
1646083925.8512564  {'bid': 33.13013649149798}
1646083925.8192937  {'bid': 35.78781525591257}
```

Fig. 24: basic_simulation-2022_02_28-09_32_04_PM-R1-experiment.data

Checking for Errors

You can use the `ctrl F` (Windows) or `cmd F` (Mac) command to search for `Error` messages in the `.log` file. If there are no results then it is likely that your simulation has run properly. If there are instances of `Error` messages then check out the *Error Handling* section.

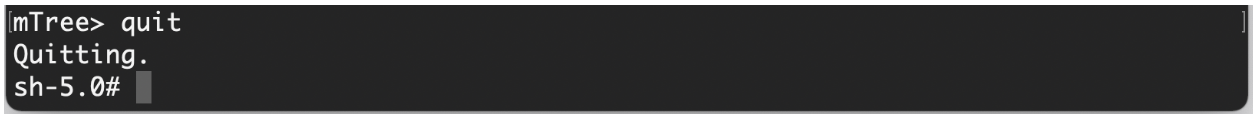
Warning: If you see no results for `Error` but your mTree log stops logging in the middle of the simulation, then it is still possible you have logic errors that don't terminate the process. Luckily, you don't have to worry about that in the **common_value_auction** auction example.

Quitting

Once the simulation has ended, you can run `quit` command in the **docker shell** to kill mTree. The `quit` command is used to kill all mTree processes as well as **delete** all *Actor* instances previously created to run the simulation.

```
quit
```

Your console should look like some version of this -



```
mTree> quit
Quitting.
sh-5.0#
```

Fig. 25: Quitting **mTree**

Conclusion

Congratulations on successfully running your first mTree simulation! If you want to know how this example was built or you want to find more projects like this, checkout *Common Value Auction* or *Learning Paths* sections. If you want to view a more in-depth case which builds an mTree project from scratch, checkout *Quick Build*.

2.3 Quick Build

2.4 Reference

Several sections are under development…

2.4.1 Theory of Operations

- Contains a description and background of Microeconomic Systems and how mTree allows you to define different actors
- Goes in depth as to why mTree.
- Need to define why messages are necessary

2.4.2 messages

In the *Actor* system, Actors only have access to their personal states. As a result, the only way Actors can change their state is through some constant design or by recieving new information from a different Actor.

In **mTree**, Actors send **messages** using the `Message` class which needs to be *imported* at the top of each file that includes the code for your **mTree** *Actor*.

The `Message` class is used to create a `Message` object, which is then used to send a **message** to another Actor. The following code snippet shows how crafting and sending a basic **message** looks like. To know more about the neccessary contents of messages check out *How to send a message*.

```
new_message = Message() #creates a message object
new_message.set_sender(self.myAddress) #self.myAddress is the agent's personal mTree␣
↪Actor address
new_message.set_directive("institution_message") #directives are used by message␣
↪receiving agents to recieve specific messages
message_payload = "any_python_data_type_would_do"
new_message.set_payload(message_payload) #you can set the payload to any python data type

self.send(reciever_address, new_message) # This method is used to finally send your␣
↪message
```

start_environment

The `start_environment` message is the very first message that gets sent by the **mTree_runner** to the *Environment* Actor (specified in the *config folder* file) after **mTree** initializes everything.

```
#inside simulation_folder/mes/environment_file.py

@directive_enabled_class
class EnvironmentClass(Environment):
    def __init__(self):
        pass

    @directive_decorator("start_environment")
    def start_environment(self, message:Message):
        pass
```

Tip: The `start_environment` directive can be viewed as the genisis message which gets the ball rolling for all other subsequent messages. Therefore, it is recommended that the directive is used to initialize the environment state as well as send important state information to other Actors.

Warning: All mTree simulation need to have a `start_environment` *directive* specified in the Environment Actor in order to start their simulation. However, messages sent in the `start_environment` directive as well as other directives can be based on your design.

How to send a message

In order to send a message, the Actor must first receive a message in a *directive* first. Once in a **directive**, the key elements for a message are -

- **Sending Actor's address**: Usually accessed by `self.myAddress`
- **Content**: This could be any python data type message (None types also work) that you want the other Actor to recieve.
- **Receiving Actor's address**: This could be accessed several ways, see code example in *directive* or checkout *address book*

Here is how you can define and send a message-

```
new_message = Message() #creates a message object
new_message.set_sender(self.myAddress) #self.myAddress is the agent's personal mTree␣
↪Actor address
new_message.set_directive("institution_message") #directives are used by message␣
↪receiving agents to recieve specific messages
new_message.set_payload("any_python_data_type_would_do") #you can set the payload to any␣
↪python data type

self.send(reciever_address, new_message) # This method is used to finally send your␣
↪message
```

In the example below, we continue the `start_messsage` directive method in the Environment and send a message to the Institution.

```
@directive_enabled_class
class EnvironmentClass(Environment):
    def __init__(self):
        pass

    @directive_decorator("start_environment")
    def start_environment(self, message:Message):

        your_message = Message() #create a message object
        your_message.set_sender(self.myAddress) #self.myAddress is the agent's personal␣
↪mTree Actor address
        your_message.set_directive("institution_message") #directives are used by␣
↪message receiving agents to recieve specific messages
        your_message.set_payload("any_python_data_type_would_do") #you can set the␣
↪payload to any python data type

        #checkout the <address_book> section in References to find how different Actors␣
↪access each other's addresses
        receiver_address = self.address_book.select_addresses({"short_name":"institution_
↪file.InstitutionClass 1"})
```

(continues on next page)

(continued from previous page)

```
        self.send(receiver_address, your_message) # This method is used to finally send␣
↪your message
```

Directives / Receiving Messages

Directives are special class methods defined in Actor classes (contained in .py files inside your **mes** folder). They are used to view messages sent to the Actor.

Actors need to have the following in their classes to recieve a particular message.

```
@directive_decorator("directive_name")
def directive_name(self, message: Message):

    message_payload = message.get_payload() #accesses the message payload
    message_sender_address = message.get_sender() #access the sender agent's address
```

Warning: In order to recieve a messsage your directive name and your method name need to be the same, otherwise, mTree throws the following *error*.

Note: For the following example our Actor is set as the *Institution* type, however, the message receiving process is applicable for any type.

In this example below, the institution receives a message sent by the Environment in *send message*.

```
@directive_enabled_class
class InstitutionClass(Institution):
    def __init__(self):
        pass

    @directive_decorator("institution_message")
    def institution_message(self, message:Message):#The method name needs to be the same␣
↪as the directive name set in quotes above

        message_payload = message.get_payload() #accesses the message payload
        message_sender_address = message.get_sender() #access the sender agent's address

        #You can find more on logging in the <logs> section in References
        self.log_message(f"message_payload = {message_payload}\n"
                         f"message_sender_address = {message_sender_address}\n")
```

Your *log* file should produce the following output -

```
1645122024.0900638  message_payload = any_python_data_type_would_do

1645122024.0937853  message_sender_address = ActorAddr-(T|:43253)
```

2.4.3 Actor Description

Imports

While coding **mTree** Actors, there are several features that **mTree** provides Actor classes the ability to interact with within the Actor world.

Necessary Imports

Each file that contains the code for your mTree Actors (Environment/Institution/Agent) **needs** to have the following imports in order to work properly. These imports provide the Actors with a range of capabilities including but not limited to communicating via messages.

```
from mTree.microeconomic_system.environment import Environment #Parent class for␣
↪Environment Actors
from mTree.microeconomic_system.institution import Institution #Parent class for␣
↪Institutoin Actors
from mTree.microeconomic_system.agent import Agent #Parent class for Agent Actors
from mTree.microeconomic_system.directive_decorators import *
from mTree.microeconomic_system.message import Message #Message class allows you to␣
↪create and send messages
import logging #Allows you to log messages to log files
```

Additional Imports

mTree also provides the following additional imports when running **mTree** in a container.

```
import math
import random

import time
import datetime
import sympy
```

General Methods and Capabilities (better name under way)

Each Actor comes with a general set of capabilities, often represented in the form of class variables and methods available to the Actor. On top of that, there are Actor specific class variables and methods that **mTree** reserves for the **Environment Actor**, which might not be available to **Institution** and **Agent Actors**. Some of these methods have individual sections such as **Message Sending**, **address_boook**, and **logging**, however, some Actor specific methods have been listed under the different Actor sections.

The four main capabilities have been listed below -

1. *Message Sending*- covers how and what Actors can send to each other
2. *Address Book*- covers how to keep track of other Actor's address(without which you can't send messages)

3. *Logging*- covers how to output interactions taking place inside a simulation
4. *short_name*- unique identifier of the Actor, used to navigate the **address_book** and keep track of Actors.

short_name

The **short_name** is a simple unique identifier created by **mTree** for each Actor within the system. The **short_name** can be used for identifying which Actor *logged the data* as well as for navigating the *address_book*.

Note: **short_name** was created to distinguish between multiple instances of the same **Actor Class** (an example of an Actor Class can be **InstitutionClass** from the message sending example above). Therefore, currently `self.short_name` is not accessible to the **Environment Actor** because there can only be one and doesn't need distinguising. However, newer versions of **mTree** plan on instilling **short_name** identifier in all Actors for uniformity purposes.

The Actors can access their individual **short_name** the following way -

```
self.short_name #since it is a class variable, it can be called anywhere in the Actor␣
↪class
```

if we use the *self.log_message(content)* method to log this variable we should observe the following output -

```
self.log_message(self.short_name) #more about this method can be found in the log_
↪message section
```

Output

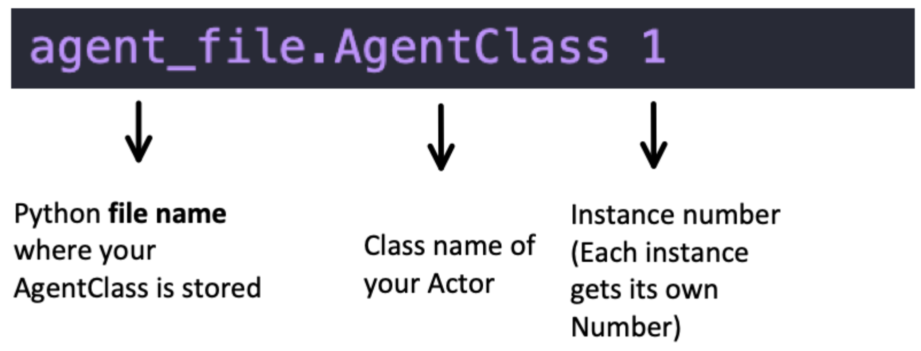
The **short_name** can identify where the Actor code is located in the **mes** folder, which **Actor Class** within that file was used to create the Actor, and, finally, which instance of the **Actor Class** is the current Actor. The last part is useful because there can be multiple instances of the same **Actor Class** and the **short_name** allows use to differentiate among them.

Environment

Here is a code snippet that you can modify to construct your **mTree** `Environment` Actor.

```
#NOTE: this python file needs to be inside the /mes folder

#These imports can also be found in the Imports section above
from mTree.microeconomic_system.environment import Environment
from mTree.microeconomic_system.institution import Institution
from mTree.microeconomic_system.agent import Agent
from mTree.microeconomic_system.directive_decorators import *
from mTree.microeconomic_system.message import Message
import math
import random
import logging
import time
import datetime

#In the config, the class below
#should be referenced as  "<.py filename>.EnvironmentClass",
#Example -  environment_file.InstitutionClass (assuming the filename is set to␣
↪environment_file.py)
```

(continues on next page)


agent_file.AgentClass 1
Python **file name** where your AgentClass is stored
Class name of your Actor
Instance number (Each instance gets its own Number)

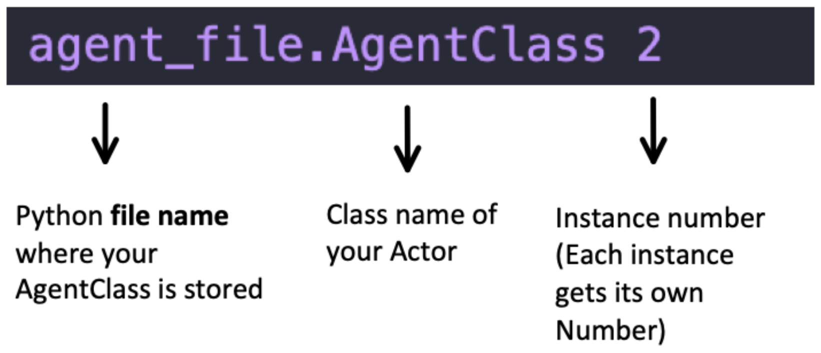

agent_file.AgentClass 2
Python **file name** where your AgentClass is stored
Class name of your Actor
Instance number (Each instance gets its own Number)

Fig. 26: If you log the `self.short_name` in an *Agent Actor* you would can see any one of the following outputs.

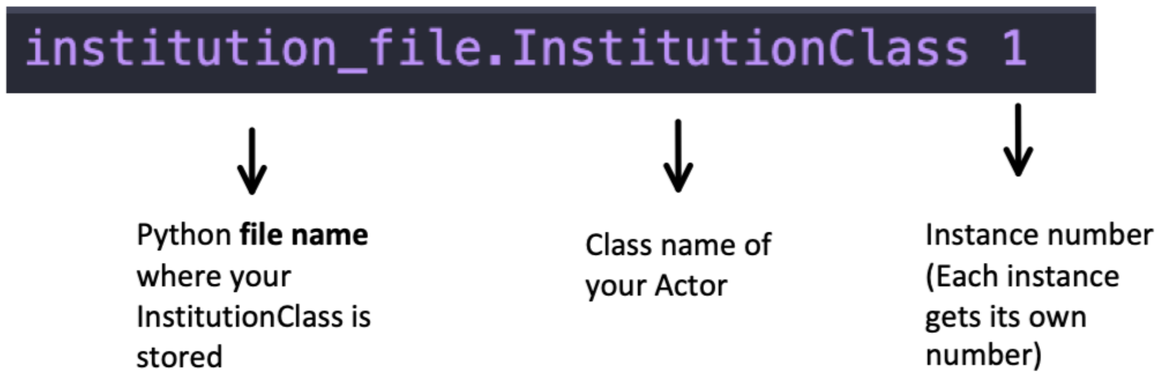

institution_file.InstitutionClass 1
Python **file name** where your InstitutionClass is stored
Class name of your Actor
Instance number (Each instance gets its own number)

Fig. 27: If you log the `self.short_name` in an *Institution Actor* you would can see any one of the following outputs.

(continued from previous page)

```
@directive_enabled_class
class EnvironmentClass(Environment): #you can change the class name to anything, as long␣
↪as the parent class (Environment) stays same
    def __init__(self):
        pass

    @directive_decorator("start_environment")
    def start_environment(self, message: Message): # The first message sent by mTree_
↪runner, check messages section to find out more

        pass
```

Tip: You can change the class name of the above Actor `EnvironmnetClass` to anything as long as the parent class `Environment` stays the same.

Institution

Here is a code snippet that you can modify to construct your **mTree** `Institution` Actor.

```
#NOTE: this python file needs to be inside the /mes folder

#These imports can also be found in the Imports section above
from mTree.microeconomic_system.environment import Environment
from mTree.microeconomic_system.institution import Institution
from mTree.microeconomic_system.agent import Agent
from mTree.microeconomic_system.directive_decorators import *
from mTree.microeconomic_system.message import Message
import math
import random
import logging
import time
import datetime

#In the config, the class below
#should be referenced as  "<.py filename>.InstitutionClass",
#Example -  institution_file.InstitutionClass (assuming the filename is set to␣
↪institution_file.py)
@directive_enabled_class
class InstitutionClass(Institution): #you can change the class name to anything, as long␣
↪as the parent class (Institution) stays same
    def __init__(self):
        pass
```

Tip: You can change the class name of the above Actor `InstitutionClass` to anything as long as the parent class `Institution` stays the same.

Agent

Here is a code snippet that you can modify to construct your **mTree** `Agent` Actor.

```python
#NOTE: this python file needs to be inside the /mes folder

#These imports can also be found in the Imports section above
from mTree.microeconomic_system.environment import Environment
from mTree.microeconomic_system.institution import Institution
from mTree.microeconomic_system.agent import Agent
from mTree.microeconomic_system.directive_decorators import *
from mTree.microeconomic_system.message import Message
import math
import random
import logging
import time
import datetime


#In the config, the class below
#should be referenced as "<.py filename>.AgentClass" ,
#Example -  "institution_file.InstitutionClass" (assuming the filename is set to␣
↪institution_file.py)
@directive_enabled_class
class AgentClass(Agent):  #you can change the class name to anything, as long as the␣
↪parent class (Agent) stays same
    def __init__(self):
        pass
```

Tip: You can change the class name of the above Actor `AgentClass` to anything as long as the parent class `Agent` stays the same.

2.4.4 config folder

There needs to be a **config** folder inside each **mTree simulation folder**. Within the **config** folder there needs to be a **.json** file that contains your simulation configurations. Although, the name of the **config** folder cannot be changed, nevertheless, your **.json** config file, can have any name.

config file

Your config file is a **.json** file containing a **json dictionary**. Inside this **json dictionary** we define the key parameters that **mTree** uses to instantiate the various *Actors* as well as any simulation specific variables that our Actors might need.

```json
{"mtree_type": "mes_simulation_description",
"name":"Basic Simulation Run",
"id": "1",
"environment": "environment_file.EnvironmentClass",
"institution": "institution_file.InstitutionClass",
"number_of_runs": 1,
```

(continues on next page)

(continued from previous page)

```
"data_logging": "json",
"agents": [{"agent_name": "agent_file.AgentClass", "number": 5}],
"properties": {"this_a_property":"this_is_a_property"}
}
```

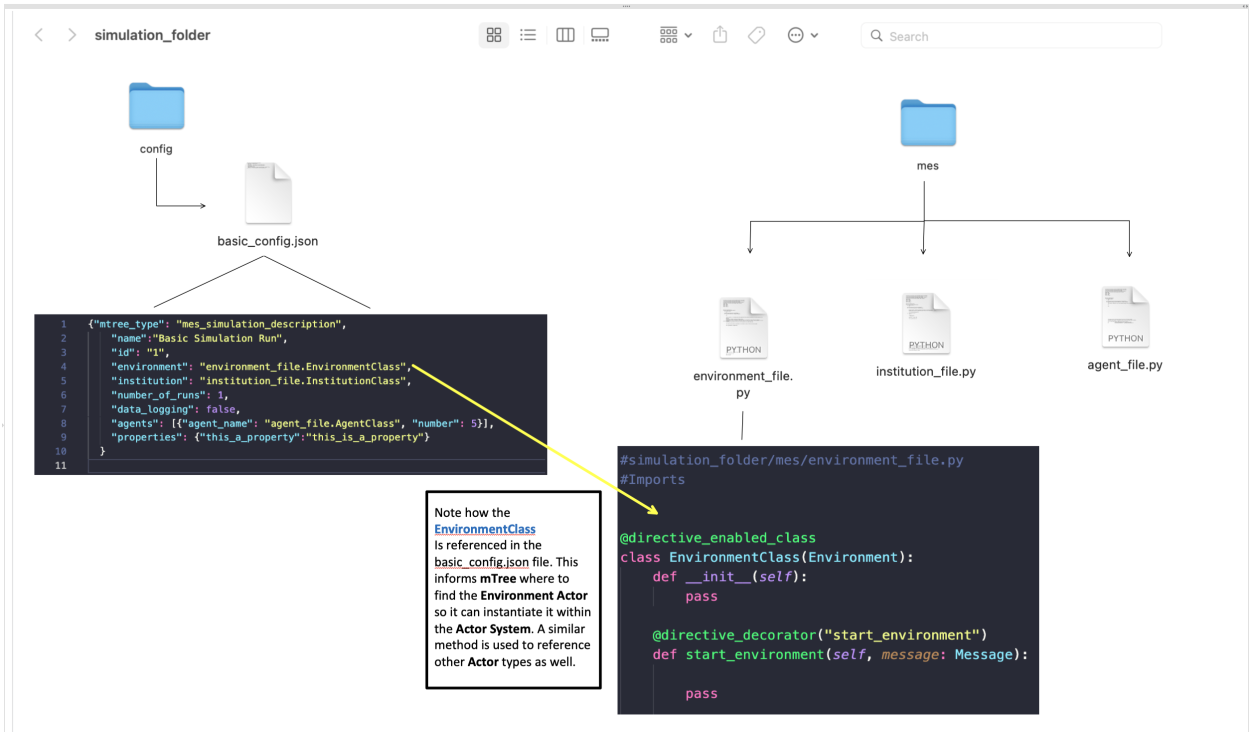
mTree use

```
{"mtree_type": "mes_simulation_description",
 "name": "any name should do",
 "id": "1"
 }
```

Although, the first three keys are used by **mTree** on a systemic level, however, even if you don't include the three keys, **mTree** assigns default values for them. More importantly, it is still highly recommended that you pass some values for them, even the ones suggested above.

Referencing different actors

Within the **config file** we inform **mTree** which code we want to use to spawn Actors.



```
{"mtree_type": "mes_simulation_description",
    "name":"Basic Simulation Run",
    "id": "1",
    "environment": "environment_file.EnvironmentClass",
    "institution": "institution_file.InstitutionClass",
    "number_of_runs": 1,
    "data_logging": false,
    "agents": [{"agent_name": "agent_file.AgentClass", "number": 5}],
    "properties": {"this_a_property":"this_is_a_property"}
  }
```

Note how the EnvironmentClass Is referenced in the basic_config.json file. This informs **mTree** where to find the **Environment Actor** so it can instantiate it within the **Actor System**. A similar method is used to reference other **Actor** types as well.

```
#simulation_folder/mes/environment_file.py
#Imports

@directive_enabled_class
class EnvironmentClass(Environment):
    def __init__(self):
        pass

    @directive_decorator("start_environment")
    def start_environment(self, message: Message):

        pass
```

Fig. 28: The figure shows how an Environmnet Actor(EnvironmentClass) is referenced within a config file.

Environment

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass" }
```

After selecting and running a **config file**, the **mTree_runner** looks for the **Environment Actor** code inside the **mes** folder. The value of the `"environment"` key - "environment_file.EnvironmentClass" informs **mTree** to spawn the **Environment Actor** using the *EnvironmentClass* class present inside the *environment_file.py* file, which in turn should be located inside the **mes** folder.

Note: Unlike **Institutions** and **agents**, **mTree** only allows for a single **Environment** per simulation. Also, each simulation **needs** to have an **Environment Actor** because the very first message that gets sent by the system is the `start_environment` message.

Institution

Single Instance

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass",
 "institution": "institutin_file.InstitutionClass"
 }
```

After selecting and running a **config file**, the **mTree_runner** looks for the **Institution Actor(s)** code inside the **mes** folder. The value of the `"institution"` key - "institution_file.InstitutionClass" informs **mTree** to spawn the **Institution Actor(s)** using the *InstitutionClass* class present inside the *institution_file.py* file inside the **mes** folder.

Multiple Instances

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass",
 "institutions": [{"institution": "institution_file.InstitutionClass", "number": 2}]
 }
```

For multiple instances of the same **InstitutionClass** Actor we use the above format where the key changes from `"institution"` to `"institutions"`, and the corresponding value is a list of dictionaries. Within the institution dictionary, the value of the `"institution"` key specifies where the **Institution Actor** code is and the value of the `"number"` key specifies - how many to spawn.

To sum it all up, the above code should create 2 Institution Actors using the same code present inside mes/institution_file.py with the class name - InstitutionClass.

Multiple Institutions

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
```

(continues on next page)

(continued from previous page)

```
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass",
 "institutions": [{"institution": "institution_file.InstitutionClass", "number": 1},
                  {"institution": "institution_file.DifferentInstitutionClass", "number":␣
↪1}]
}
```

Notice that the "institutions" key has a list as its corresponding value. Inside this list, you can insert the different types of **Institution Actor** you want to create as separate dictionaries. This is useful if you have two separate coded institution classes that serve different roles in your microeconomic system.

You can also control the number of instances of each particular **Institution Actor** using the "number" key.

Agents

The reference for **Agents** works exactly like references for **Institutions**.

Single Instances

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass",
 "institution": "institutin_file.InstitutionClass",
 "agent": "agent_file.AgentClass"
 }
```

After selecting and running a **config file**, the **mTree_runner** looks for the **Agent Actor(s)** code inside the **mes** folder. The value of the "agent" key - “agent_file.AgentClass” informs **mTree** to spawn the **Agent Actor(s)** using the *AgentClass* class present inside the *agent_file.py* file inside the **mes** folder.

Multiple Instances

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass",
 "institution": "institutin_file.InstitutionClass",
 "agents": [{"agent": "agent_file.AgentClass", "number": 2}]
 }
```

For multiple instances of the same **AgentClass** Actor we use the above format where the key changes from "agent" to "agents", and the corresponding value is a list of dictionaries. Within the agent dictionary, the value of the "agent" key specifies where the **Agent Actor** code is and the value of the "number" key specifies - how many to spawn.

To sum it all up, the above code should create 2 Agent Actors using the same code present inside mes/agent_file.py with the class name - AgentClass.

Multiple Agents

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass",
```

(continues on next page)

(continued from previous page)

```
 "institution": "institutin_file.InstitutionClass",
 "agents": [{"agent": "agent_file.AgentClass", "number": 1},
                  {"agent": "agent_file.DifferentAgentClass", "number": 1}]
}
```

Notice that the "agents" key has a list as its corresponding value. Inside this list, you can insert the different types of **Agent Actor** you want to create as separate dictionaries. This is useful if you have two separate coded agent classes that serve different roles in your microeconomic system.

You can also control the number of instances of each particular **Agent Actor** using the "number" key.

Simulation Properties/ self.get_properties()

Users are allowed to specify additional information to the "properties" dictionary. This dictionary is reserved for including information that is simulation specific and can be used to initialize different agent types, initialize different institutions, and much more. Check out one of the *Learning Paths* to view how properties can be used to prevent hard coding Actors.

```
{"mtree_type": "mes_simulation_description",
 "name": "Basic Simulation",
 "id": "1" ,
 "environment": "environment_file.EnvironmentClass",
 "institution": "institutin_file.InstitutionClass",
 "agents": [{"agent": "agent_file.AgentClass", "number": 1},
                  {"agent": "agent_file.DifferentAgentClass", "number": 1}],
 "properties": {"agent_types": ["buyer", "seller"],
                 "agent_endowment": 30,
                 "institution_type": ["sealed_bid_auction", "common_value"]
                  }
}
```

Accessing Properties

Information mentioned in the "properties" dictionary can be accessed by the **Environment** Actor using the following code.

```
self.get_properties() # this should return the entire properties dictionary.
```

Example

If we wanted to access the properties mentioned above, we could use the following code.

```
agent_type_list = self.get_properties()["agent_types"] #list, accessing the different␣
↪agent types in the system
agent_endowment = self.get_properties()['agent_endowment'] #int, accesing the agent␣
↪endowment
institution_type_list = self.get_properties()['institution_type'] #list, institution_
↪type_list
```

Note: Only the **Environment** Actor has access to the self.get_properties() method and can choose to pass relevant information (defined in the config) regarding the an Actor's initial states to them.

2.4.5 address book

The **address_book** is an **mTree** object that stores and manages addresses of all the Actors that are initialized in the *config folder* file. Each Actor in the system has an **address_book** object instantiated when they are spawned. However, at the beginning, only the *Environment* Actor's **address_book** has the complete list of Actor addresses in the system.

The **Environment Actor** can then choose to pass the addresses to different **Institution** and **Agent** Actors across the system. We have listed below the different methods that this **address_book** object has and how to access them.

How to access the address_book

The **address_book** object can be accessed by the Actors in the following ways

```
self.address_book()
```

`self.address_book` is a class variable that gets set by **mTree** for each Actor prior to sending the *start_environment* directive message and points to the Actor's own **address_book** object. Since `self.address_book` is a class variable, it can be accessed everywhere.

Structure

Below we evaluate one of the key **address_book** methods and explore how addresses are stored.

```
all_addresses = self.address_book.get_addresses() #This code should return a dictionary␣
↪of the following format
self.log_message(all_addresses) #since mTree suppresses print statements, logging is the␣
↪only way to get info out
```

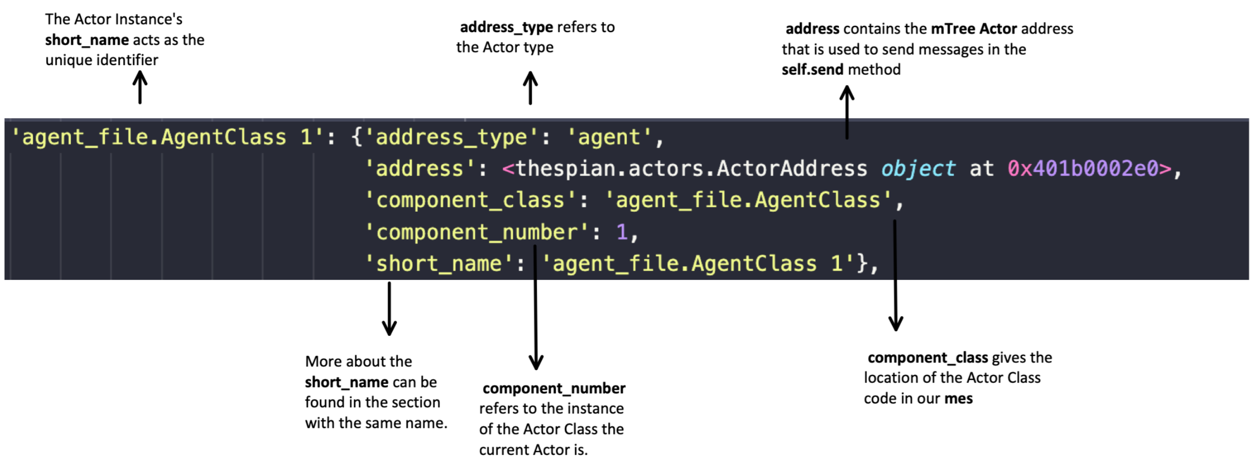
```
#The above message should output the following dictionary
#Notice all the keys are the different actor's short_names and the value of each
#key is another dictionary containing other important distinguishing information about␣
↪the Actor
    {'institution_file.InstitutionClass 1': {'address_type': 'institution', #The Actor's␣
↪type
                                            'address': <thespian.actors.ActorAddress␣
↪object at 0x401aff5c70>, #The Actor's address
                                            'component_class': 'institution_file.
↪InstitutionClass', #Where the code for ActorClass is located
                                            'component_number': 1, #instance number of␣
↪the Actor
                                            'short_name': 'institution_file.
↪InstitutionClass 1'}, #Actor short_name
    'agent_file.AgentClass 1': {'address_type': 'agent',
                                'address': <thespian.actors.ActorAddress object at␣
↪0x401b00002e0>,
                                'component_class': 'agent_file.AgentClass',
                                'component_number': 1,
                                'short_name': 'agent_file.AgentClass 1'},
```

(continues on next page)

(continued from previous page)

```
    'agent_file.AgentClass 2': {'address_type': 'agent',
                                'address': <thespian.actors.ActorAddress object at␣
↪0x401b000460>,
                                'component_class': 'agent_file.AgentClass',
                                'component_number': 2,
                                'short_name': 'agent_file.AgentClass 2'},
    'agent_file.AgentClass 3': {'address_type': 'agent',
                                'address': <thespian.actors.ActorAddress object at␣
↪0x401b0004f0>,
                                'component_class': 'agent_file.AgentClass',
                                'component_number': 3,
                                'short_name': 'agent_file.AgentClass 3'},
    'agent_file.AgentClass 4': {'address_type': 'agent',
                                'address': <thespian.actors.ActorAddress object at␣
↪0x401b000580>,
                                'component_class': 'agent_file.AgentClass',
                                'component_number': 4,
                                'short_name': 'agent_file.AgentClass 4'},
    'agent_file.AgentClass 5': {'address_type': 'agent',
                                'address': <thespian.actors.ActorAddress object at␣
↪0x401b000610>,
                                'component_class': 'agent_file.AgentClass',
                                'component_number': 5,
                                'short_name': 'agent_file.AgentClass 5'}
                                }
```

We are going to evaluate a single entry in this **address_book** dictionary and explore what each information means in the figure below.



```
'agent_file.AgentClass 1': {'address_type': 'agent',
                            'address': <thespian.actors.ActorAddress object at 0x401b0002e0>,
                            'component_class': 'agent_file.AgentClass',
                            'component_number': 1,
                            'short_name': 'agent_file.AgentClass 1'},
```

The Actor Instance's **short_name** acts as the unique identifier
address_type refers to the Actor type
address contains the **mTree Actor** address that is used to send messages in the **self.send** method
More about the **short_name** can be found in the section with the same name.
component_number refers to the instance of the Actor Class the current Actor is.
component_class gives the location of the Actor Class code in our **mes**

Fig. 29: More about **short_name** can be found in the *short_name* section.

Note: For the rest of the **address_book** section, we will refer to keys in the dictionary above as **entries** and their corresponding value, which is another dictionary, as the **description dictionary**.

Warning: Currently all Actor Instances except the **Environment Actor** have an entry in the **address_book**. As a result, the only way to get the **Environment Actor's** address is to receive a message from it and access the **address** using `message.get_sender()` method inside the **directive** you receive a message from the **Environment Actor**.

Methods

The **address_book** object provides several methods.

self.address_book.get_addresses()

The following returns a dictionary with all **address_book** elements exactly like the one explored in *Structure* section above.

```
self.address_book.get_addresses() #This code should return a dictionary of the following␣
↪format

#the get_addresses() method returns all the elements stored in the .addresses variable␣
↪inside the address_book object
#another way to access the same dictionary can be
#self.address_book.addresses
```

self.address_book.merge_addresses(addresses)

This method allows you to merge your **address_book** with another **address_book**. The goal of this method is to append your personal **address_book** using the **addresses** provided as input.

Input: dict

The **addresses** argument in `self.address_book.merge_addresses(addresses)` needs follow the **address_book** dictionary structure as shown in the *Structure* section.

Output: None

Although,``self.address_book.merge_addresses(addresses`` method does not return anything, nevertheless, it updates the Actor's personal **address_book** object to include the new **entries** mentioned in the **addresses** input dictionary.

Tip: Since, at first, only the **Environment** Actor has a complete **address_book** with **entries** of all the **Actors** in the system. Consequentially, the **Environment** can access the **address_book dictionary** using *self.address_book.get_addresses()* and pass this to other Actors by setting it as the *message* **payload**. The Actor receiving the **address_book dictionary** can then add those addresses to its personal **address_book** using `self.address_book.merge_addresses(address_book_dictionary)`

Example: Environment sends Institution the address_book

self.address_book.get_agents()

The following returns a dictionary similar to the one in `self.address_book.get_addresses()`, however, only includes **entries** whose **description dictionary "address_type"** key has the value - **"agent"**

```
self.address_book.get_agents()#Only returns the addresses of Agent Actors
```

Output: float

The code above should return the following dictionary -

```
{'agent_file.AgentClass 1': {'address_type': 'agent',  # all elements are 'agents'
                             'address': <thespian.actors.ActorAddress object at␣
↪0x401b0002e0>,
                             'component_class': 'agent_file.AgentClass',
                             'component_number': 1,
                             'short_name': 'agent_file.AgentClass 1'},
'agent_file.AgentClass 2': {'address_type': 'agent',
                            'address': <thespian.actors.ActorAddress object at␣
↪0x401b000460>,
                            'component_class': 'agent_file.AgentClass',
                            'component_number': 2,
                            'short_name': 'agent_file.AgentClass 2'},

                            ... }
```

self.address_book.get_institutions()

The following returns a dictionary similar to the one in `self.address_book.get_addresses()`, however, only includes entries whose **"address_type"** key has the value - **"institution"**

```
self.address_book.get_institutions()#Only returns the addresses of Agent Actors
```

Output: dict

The code above should return the following dictionary

```
{'institution_file.InstitutionClass 1': {'address_type': 'institution',
                                         'address': <thespian.actors.ActorAddress object␣
↪at 0x401aff5c70>,
                                         'component_class': 'institution_file.
↪InstitutionClass',
                                         'component_number': 1,
                                         'short_name': 'institution_file.
↪InstitutionClass 1'},
                                         ...
                                         }
```

self.address_book.num_agents()

The following sums up the number of entries with `{"address_type":"agent"}` in their description. So if there are 5 Agent Actors in our simulation, the following code should output-

```
self.address_book.num_agents()
```

Output: float

```
5
```

self.address_book.num_institutions()

The following sums up the number of entries with `{"address_type":"institution"}` in their description. So if there is a single **Institution Actor** in our simulation, the following code should output-

```
self.address_book.num_institutions()
```

Output: float

```
1
```

self.address_book.select_addresses(selector)

The `self.address_book.select_addresses(selector)` outputs a list of **mTree addresses** based on the **selector** that is provided.

Input: selector(dict)

The **selector** is a dictionary that can only have one of the following key and value pairs.

Table 1: Selector

key	value
"address_type"	"agent"/"institution"
"short_name"	"file_name.ActorClass instance(int)"

The purpose of the **selector** is to help **address_book** object select specific **mTree addresses** from the **entries** that have the same **value** as the **selector** inside their description dictionaries.

```
#address_type selectors
agent_addresses_selector = {"address_type": "agent"}
institution_address_selector = {"address_type": "institution"}

#short_name selectors
agent_short_name_selector = {"short_name": "agent_file.AgentClass 1"}

#if you pass any of these as an input to
self.address_book.select_addresses(agent_addresses_selector)
#the above code would output either a list of addresses or a single address
```

Output: list or address

Depending on the number of **entries** in the **address_book** that agree with the **selector**, `self.address_book.select_addresses(selector)` returns either a list of **mTree_addresses** or a single **mTree_address**.

Example: List of Addresses Returned

```
#we want to select all mTree Actor addresses of those that have "address_type" as "agent
↪" in their
#description dictionaries
selector = {"address_type": "agent"} #we create a selector dictionary
agent_addresses = self.address_book.select_addresses(selector)

self.log_message(agent_addresses) #more about self.log_message can be found in the self.
↪log_message section
```

Output

```
#Assuming we use the same config for all examples
#the above code should produce a similar list in the log file
#since we are using a config file with 5 Agent Actors, we get a list with 5 elements
[<thespian.actors.ActorAddress object at 0x40180f5a30>, <thespian.actors.ActorAddress␣
↪object at 0x40180f5cd0>, <thespian.actors.ActorAddress object at 0x40180f6e80>,
↪<thespian.actors.ActorAddress object at 0x40180f6e20>, <thespian.actors.ActorAddress␣
↪object at 0x40180f6d90>]
```

Example: Single Address Returned

```
#we want to select all mTree Actor addresses of those that have "address_type" as
↪"institution" in their
#description dictionaries
#since we are using a config file with 1 Institution Actor, we get a single address␣
↪returned

selector = {"address_type": "institution"}
institution_address = self.address_book.select_addresses(selector)

self.log_message(institution_address) #more about self.log_message can be found in the␣
↪self.log_message section
```

Output

```
#Assuming we use the same config for all examples,
#we know that there is only 1 Institution Actor in our system.
#Therefore the above code should produce a the following in the log file

ActorAddr-LocalAddr.1 #This is mTree address for the institution and can be used in the␣
↪message sending proceess
```

Example: Using short_name selector

The *short_name* selector is useful when the **Actor** wants to send a message specifically to another Actor. Since, no two Actors, share a common **short_name**, `self.address_book.select_addresses(selector)` should return a single address.

```
#we want to select the mTree address of the Actor with the short_name - "institution_
↪file.InstitutionClass 1"
```

(continues on next page)

(continued from previous page)

```
#since short_names are unique to each Actor instance, the following should return a␣
↪singe Actor address

selector = {"short_name": "institution_file.InstitutionClass 1"} # more on how short_
↪names get assigned can be found in the short_name section
institution_address = self.address_book.selector(selector)

self.log_message(institution_address) #more about self.log_message can be found in the␣
↪self.log_message section
```

Output

```
#Note this is should be the same as the output produced in the Example where our␣
↪selector was {"address_type": "institution"}
ActorAddr-LocalAddr.1 #This is mTree address for the institution and can be used in the␣
↪message sending proceess
```

Note: In most cases, `self.address_book.select_addresses(selector)` produces a `for loop` compatible list of addresses(if there is more than one entry for which the **selector** applies to). Consequently, this method becomes useful when you want to send *message* to multiple Actors with varying **payloads**. However, if you want to send the same message(with no change in directive or payload) to a group of Actor types, you might want to consider using *self.address_book.broadcast_message(selector, message)* instead.

Example: Combining .select_addresses(selector) and Message()

In the code example below, we try to send each Agent Actor slightly different **value estimates** for a common value good. More about this code can be found in the *Common Value Auction* section in *Learning Paths*.

Institution Code Here the institution sends slightly different **value estimates** of a **common value good** to all the Agent Actors in its **address_book**

```
#This is an imagined directive that our institution finds itself in

@directive_decorator("institution_directive")
def institution_directive(self, message: Message):

    #We are assuming that the Institution Actor has already received a copy of the
    #Environment Actor's address_book which it has merged with its own using the
    # .merge_addresses(addresses) method.
    #We also assume we are using the same config with 5 Agent Actors.
    agent_address_list = self.address_book.select_addresses({"address_type": "agent"}) #␣
↪produces a list of 5 Agent Actor addresses

    for agent_address in agent_address_list: #we iterate over the addresses

        #Assume self.common_value and self.error are float values set in
        #a previous directive
        lower_bound = self.common_value - self.error
        upper_bound = self.common_value + self.error

        #random.uniform will produce a different value_estimate b/w [lower_bound, upper_
↪bound] for each iteration of the for loop
```

(continues on next page)

(continued from previous page)

```
        value_estimate = random.uniform(lower_bound, self.common_value + self.error)
↪#The random function should return a number between () and 20 with uniform probability

        #The dictionary we send to each agent
        payload_dict = {"value_estimate": value_estimate, "error": self.error}

        agent_message =  Message()#create a message object
        agent_message.set_directive("receive_value_estimate") #this is the directive␣
↪where each Agent can receive messages
        agent_message.set_sender(self.myAddress)#set the sender to the Actor's personal␣
↪address
        agent_message.set_payload(payload_dict) #pass the payload dict we just defined

        self.send(agent_address, agent_message) #the agent_address would be new each␣
↪time the for loop is run
```

Agent Message Receiving Code

Here the Agent Actor receives the unique `value_estimate` that the **Institution** sent along with the ubiquitous `error` key.

```
@directive_decorator("receive_value_estimate")
def receive_value_estimate(self, message:Message):

    payload_dict = message.get_payload()#returns the payload dictionary that was set by␣
↪the sender
    #we define class variables using the keys of the payload dictionary
    self.value_estimate = payload_dict["value_estimate"]
    self.error  = payload_dict["error"]
```

Note: *self.address_book.select_addresses(selector)* is very useful method when sending a slightly unique message to all Actors of one type (Institution/Agent). Notice, the only aspect of the message that changed for each iteration of the `for loop` was the **value_estimate**. Moreover, in this example, we assume that all **Agent Actors** have a *@directive_decorator("receive_value_estimate")* in their AgentClass. Although, this shouldn't be a problem if all Actors belong to the same **AgentClass**, however, if there is more than one **AgentClass** defined in the **mes** folder as well as referenced in the *config file*, each AgentClass would need to have a *@directive_decorator("receive_value_estimate")* method. Otherwise, **mTree** would through an **ERROR**.

self.address_book.broadcast_message(selector, message)

This method broadcasts or sends a **message** to all **entries** in the Actor's personal **address_book** that the **selector** agrees with. Unlike *self.address_book.select_addresses(selector)*, which returns a list of addresses, the `self.address_book.broadcast_message(selector, message)` method does the message sending for the Actor.

Input: selector(dict), message

The **selector** is a dictionary that can only have one of the following key and value pairs.

Table 2: Selector

key	value
“address_type”	“agent”/”institution”
“short_name”	“file_name.ActorClass instance(int)”

The purpose of the **selector** is to help **address_book** object select specific **mTree addresses** from the **entries** that have the same **value** as the **selector** inside their **description dictionaries**.

The **message** argument takes in the *message* object that needs to be passed to all entries that the **selector** applies to.

Output: None

This method doesn’t return anything, however, performs the important function of sending the **message**, that is submitted as an argument, to all the **address_book entries** that the **selector** applies to.

Example

The following code should send a message to all Agent Actor **entries** present in the **address_book**

```
new_message = Message()#we create a new message object
new_message.set_sender(self.myAddress)
new_message.set_directive("receive_message") #the directive_method where this message␣
↪will be received
payload = None #you can choose this to be anything
new_message.set_payload(payload)

selector = {"address_type": "agent"} # you can change the selector

self.address_book.broadcast_message(selector, new_message)#take note of how we pass the␣
↪selector and the message object
```

Note: The *self.address_book.broadcast_message(selector, message)* is useful when you want to send the same message (no variation) to all Actors of one type (Institution/Agent).

Example: Common Value Auction

The code example below is from *Common Value Auction* section in *Learning Paths*. In this portion, the *Environment* Actor sends the *Agent* Actor their **endowment** which is constant for all Agents in the system. As a result, *self.address_book.broadcast_message(selector, message)* becomes valuable because we are sending the same *message* to all Agent Actors.

Environment Code

```
#self.provide_endowment is a method that gets run in the start_environment method

def provide_endowment(self):
    endowment = 30 #Agent endowment

    #Defining a Message
    new_message = Message()  #declare message
    new_message.set_sender(self.myAddress)  # set the sender of message to this actor
    new_message.set_directive("set_endowment")  #set the directive
    payload_dict  = {"endowment": endowment}
    new_message.set_payload(payload_dict) #set the payload as the payload_dict we␣
↪defined in the line above
```

(continues on next page)

(continued from previous page)

```
    #Broadcasting the message using the AddressBook
    selector = {"address_type": "agent"}
    self.address_book.broadcast_message(selector, new_message) #this will send a message␣
↪to all Agent Actors

    #Or the following should also work
    #self.address_book.broadcast_message({"address_type": "agent"}, new_message)
```

Agent Message Receiving Code

```
#this is a directive inside the AgentClass
@directive_decorator("set_endowment")
def set_endowment(self, message: Message):

    payload_dict = message.get_payload() #we access the payload/content of the message␣
↪that was sent
    environment_address = message.get_sender() #we access the environment's address

    self.endowment = payload_dict["endowment"]#we create a class variable self.endowment␣
↪and set it equal to the amount sent by the environment
```

2.4.6 Logs

Logging is a way to output important information from a simulation in order to keep track of what the code is doing at various steps, as well as collect data for analysis.

There are 2 types of logging that **mTree** allows -

1. *Experiment Logging* - Appears in the `.log` files.
2. *Data Logging*- Appears in the `.data` files.

Each run of an **mTree** *config file* creates a `.log` file which gets placed inside the **logs** folder inside your **simulation_folder** (where your *config folder* and **mes** folders are stored).

Note: If you have already created a **logs** folder prior to running your **mTree** simulation, your `.log` and `.data` files should appear inside it. Nevertheless, even if you haven't created one, **mTree** will generate a **logs** folder for you and place those files inside it.

Note: Although, each run of a *config file* creates a `.log` file, however, a `.data` file only gets created when one of the **ActorClasses** uses the *self.log_data(content)* method somewhere in one of its methods.

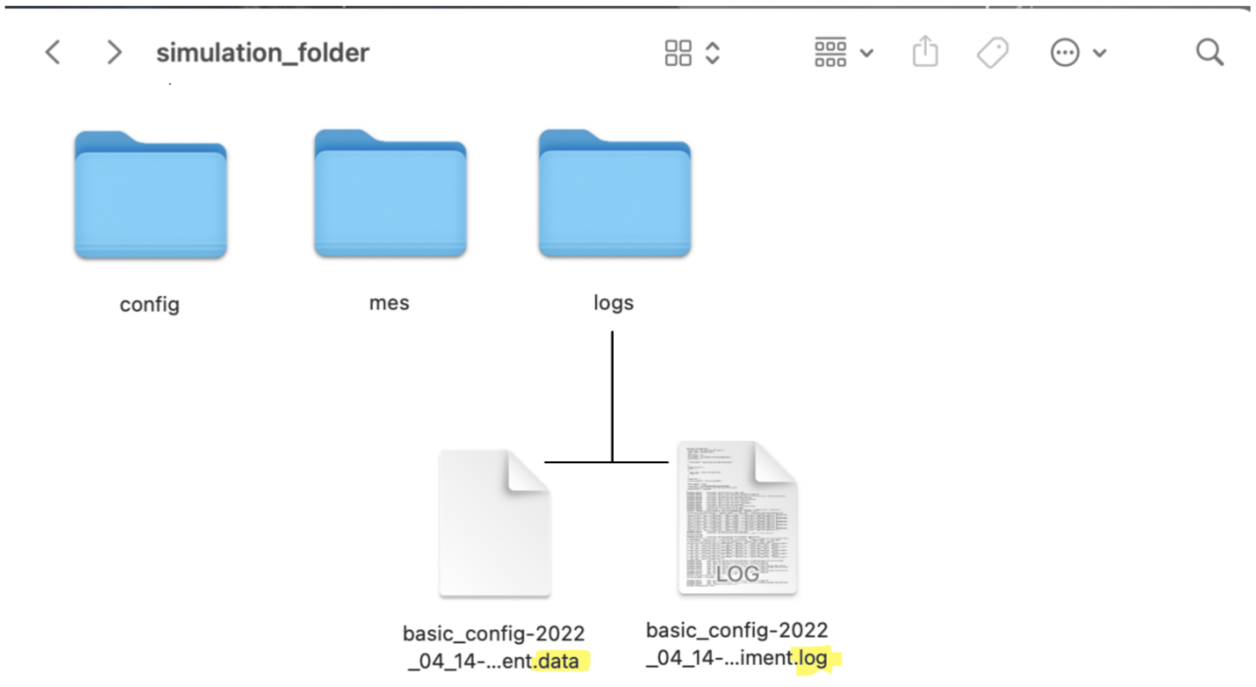

simulation_folder
config
mes
logs
basic_config-2022_04_14-...ent.data
basic_config-2022_04_14-...iment.log
LOG

Fig. 30: Inside the **logs** folder

Naming

Each `.log` and `.data` file associated with a *config file* get named the following way -

The following figure shows how different runs of the same *config file* are named -

Experiment Logging

Experiment Logging can be used to keep track of

1. The messages Actors send or receive.
2. How received messages change the internal state of the Actors.

As a result, **Experiment Logging** can be used to **monitor** whether our simulation/experiment is behaving according to our **microeconomic system** design.

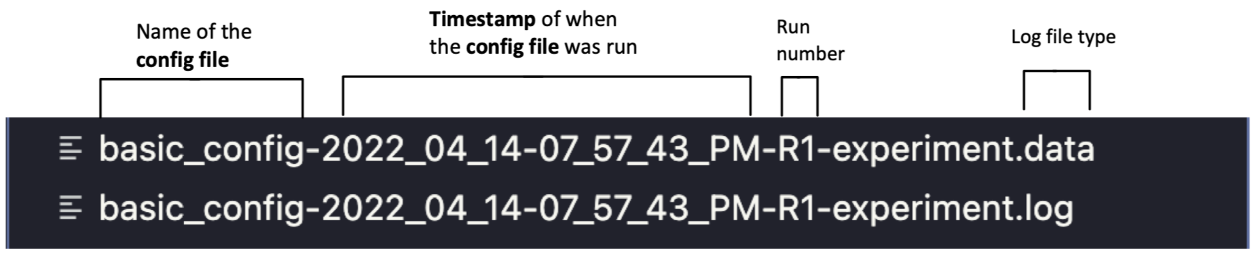
self.log_message(content)

The `self.log_message(content)` is a method that gets setup for each Actor during initialization and allows them to output information from various points within the **ActorClass** to the *.log*.

Since **mTree** suppresses `print()` statements, `self.log_message(content)` is the closest method we have to monitor how the internal state of the Actors is changing in response to received messages.

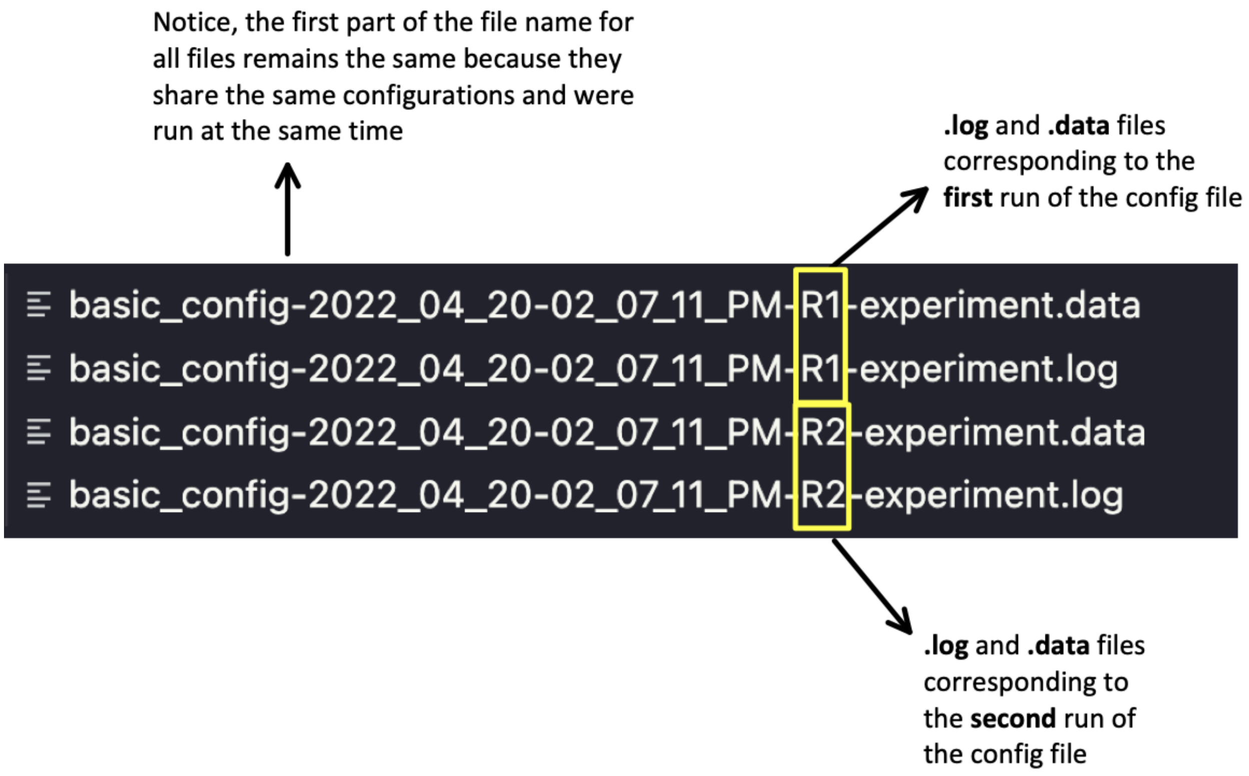
The `self.log_message(content)` logs to the *.log* file present in the *logs* folder.

Input: content(any python data type)


Name of the **config file**
Timestamp of when the **config file** was run
Run number
Log file type

```
basic_config-2022_04_14-07_57_43_PM-R1-experiment.data
basic_config-2022_04_14-07_57_43_PM-R1-experiment.log
```

Fig. 31: Each run of the same *config file* should generate its own set of *.log* and *.data* files


Notice, the first part of the file name for all files remains the same because they share the same configurations and were run at the same time
.log and **.data** files corresponding to the **first** run of the config file
.log and **.data** files corresponding to the **second** run of the config file

```
basic_config-2022_04_20-02_07_11_PM-R1-experiment.data
basic_config-2022_04_20-02_07_11_PM-R1-experiment.log
basic_config-2022_04_20-02_07_11_PM-R2-experiment.data
basic_config-2022_04_20-02_07_11_PM-R2-experiment.log
```

Fig. 32: Snapshot of the files that get generated inside the **logs** folder.

`self.log_message(content)` method can take in all arguments types that a `print()` function is equipped to handle.

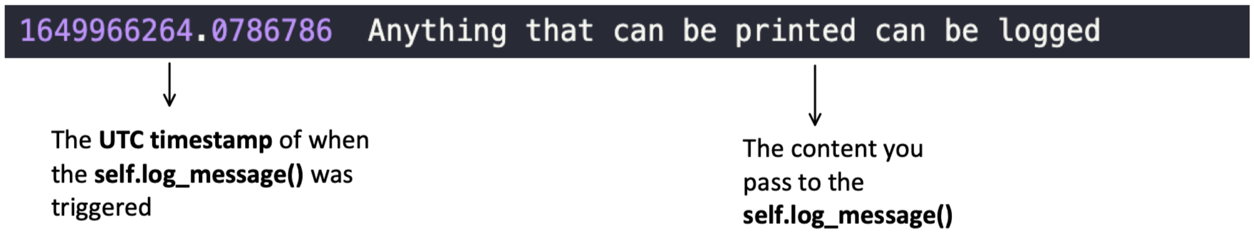
> **Warning:** One of the common mistakes people make while using `self.log_message(content)` is treating it exactly like a `print()` function and passing more than one argument to it. For instance, if you pass 2 arguments to a `print("a", "b")` function, `print()` treats the comma in between "a" and "b" as space and outputs - `a b`. However, if you run `self.log_message("a", "b")` you will get an **ERROR** because the method only takes in a single argument and you have tried to pass in two.

```
self.log_message(f"Anything that can be printed can be logged")
```

Output

The output generated in your *.log file* should look like the following.

```
1649966264.0786786  Anything that can be printed can be logged
```



```
1649966264.0786786  Anything that can be printed can be logged
```

The **UTC timestamp** of when the **self.log_message()** was triggered
The content you pass to the **self.log_message()**

Fig. 33: A `.log` file entry

Each time `self.log_message(content)` gets called by one of the Actors, there is a new entry in the `.log` file on a new line. The first part of the entry is the **UTC timestamp** of when the `self.log_message(content)` was called, the second part (separated by a space) is the **content** you passed into the method.

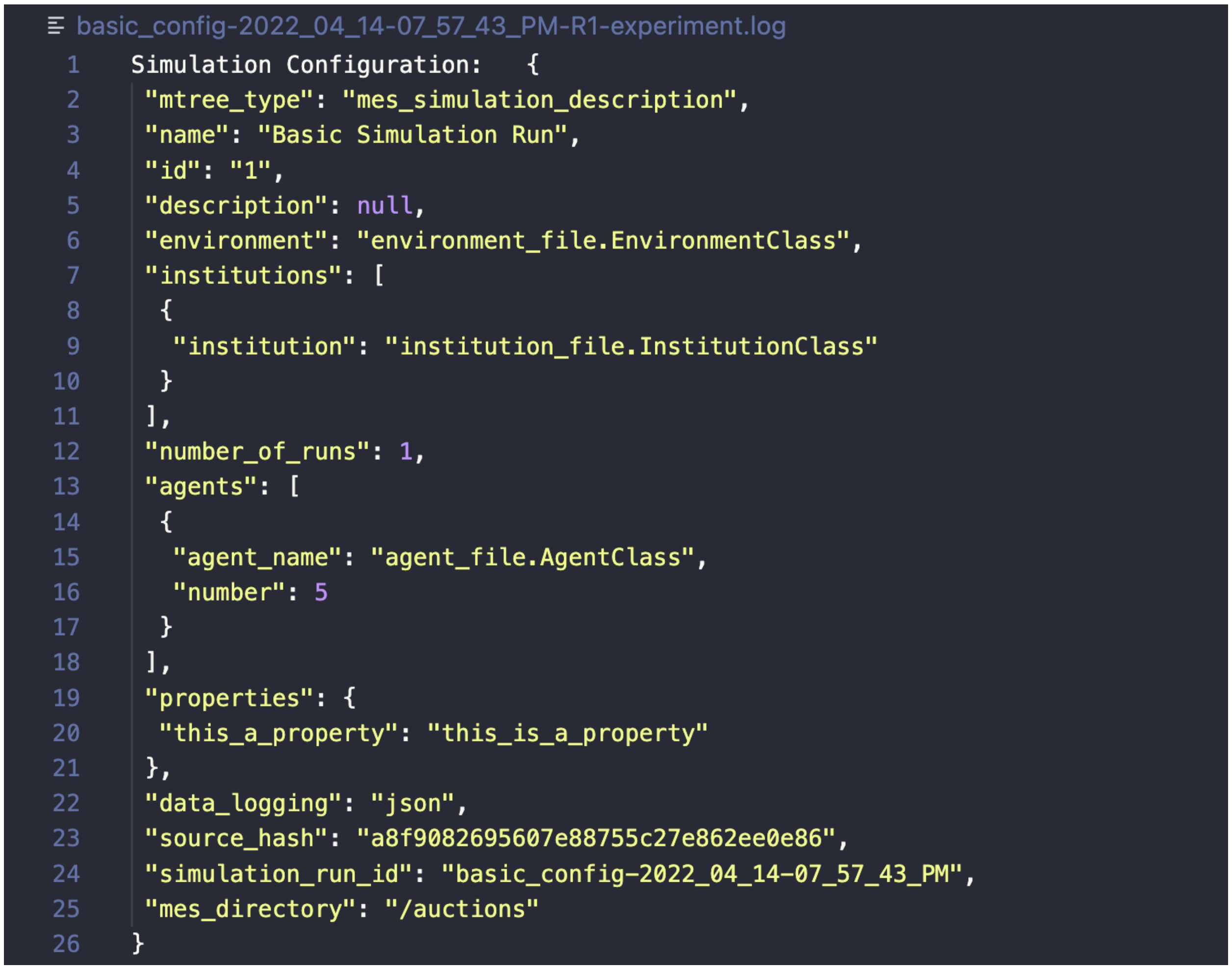
Log File

The `.log` file for each run of an **mTree** *config file* should appear in the **logs** folder.

Structure

The first few lines of the `.log` file that gets generated from the simulation run consists of the **json dictionary** that was used as the *configurations dictionary* for the run.

The second part of the `.log` file consists of a series of **timestamped** log outputs made by both **mTree** and *self.log_message(content)* calls (inside different message passing/receiving **Actors**).



basic_config-2022_04_14-07_57_43_PM-R1-experiment.log

```
Simulation Configuration:   {
 "mtree_type": "mes_simulation_description",
 "name": "Basic Simulation Run",
 "id": "1",
 "description": null,
 "environment": "environment_file.EnvironmentClass",
 "institutions": [
  {
   "institution": "institution_file.InstitutionClass"
  }
 ],
 "number_of_runs": 1,
 "agents": [
  {
   "agent_name": "agent_file.AgentClass",
   "number": 5
  }
 ],
 "properties": {
  "this_a_property": "this_is_a_property"
 },
 "data_logging": "json",
 "source_hash": "a8f9082695607e88755c27e862ee0e86",
 "simulation_run_id": "basic_config-2022_04_14-07_57_43_PM",
 "mes_directory": "/auctions"
}
```

Fig. 34: This is the same as the **json dictionary** inside the *config file* you ran.

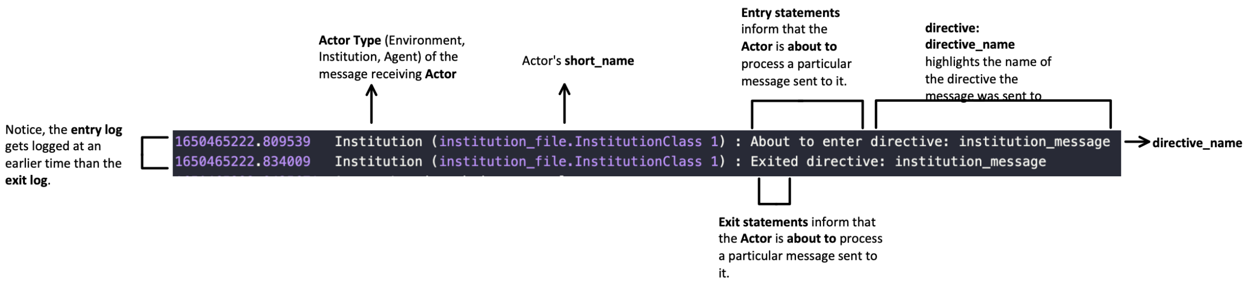
mTree logging

mTree logs each time an **Actor** enters and exits a *directive method*. In other words, **mTree** logs each time an **Actor** -

1. Receives a message - denoted by **entering** the *directive method* where the message was sent to.
2. Is finished processing the message - denotend by **exiting** the *directive method* where the message was sent to.

mTree logging also informs us **who** (*short_name*) received the message as well.

The following figure evaluates how an **Institution Actor** logs when a message is sent to it by the **Environment**.



```
1650465222.809539   Institution (institution_file.InstitutionClass 1) : About to enter directive: institution_message
1650465222.834009   Institution (institution_file.InstitutionClass 1) : Exited directive: institution_message
```

Actor Type (Environment, Institution, Agent) of the message receiving Actor
Actor's short_name
Entry statements inform that the Actor is about to process a particular message sent to it.
directive: directive_name highlights the name of the directive the message was sent to
Notice, the entry log gets logged at an earlier time than the exit log.
directive_name
Exit statements inform that the Actor is about to process a particular message sent to it.

The **entry** and **exit** method can be a good way to keep track of what messages are being sent and whether they were processes or not. For instance, if a message is never received by an **Actor**, then there should be not trace of an **entry log** similar to the once above inside the `.log` file. Similarly, if a message is received, but somewhere inside the directive a **python error** occurs, then, firstly, **mTree** should log the error, secondly, there should be no trace of the **exit log**.

Note: For clarity purposes, we have shown the **entry** and **exit** statements by **mTree** one after the other. However, since **mTree** has a lot of concurrent actors logging to the `.log` file, it is possible that these statements are far apart and have multiple log **entries** made by other **Actors** in between. Regardless, the order of the statements should not change, meaning, the **entry** statement should still be observed before the **exit** statement.

Actor logging

Actor logging refers to the logging done by different Actors using the *self.log_message(content)* method inside the Actor. This can be used in conjunction with that **mTree** does automatically for both debugging the code and tracking whether internal states of **Actors** is changing according to our design.

Example

The following code shows the log output generated by the **Institution Actor** when it receives a copy of the *addresses* from the **Environment** and adds it to its own version of address_book. During the process, we try to track the following values -

- *self.address_book.get_addresses()* - before and after *self.address_book.merge_addresses(addresses)* is called.
- message.payload() - The content of the message that gets sent.
- message.get_sender() - The **mTree** address of the message sending Actor.

We include the **Environment Actor** code that does the message sending but we only display the log statements made by the **Institution Actor** for clarity.

Environment Code Environment sends the **addresses dictionary** to Institution in start_environment directiv.

```
from mTree.microeconomic_system.environment import Environment
from mTree.microeconomic_system.institution import Institution
from mTree.microeconomic_system.agent import Agent
from mTree.microeconomic_system.directive_decorators import *
from mTree.microeconomic_system.message import Message
import math
import random
import logging
import time
import datetime


@directive_enabled_class
class EnvironmentClass(Environment):
    def __init__(self):
        pass

    @directive_decorator("start_environment")
    def start_environment(self, message: Message):

        selector = {"short_name": "institution_file.InstitutionClass 1"}#short_name␣
↪selector, helps in the select_addresses() method
        institution_address = self.address_book.select_addresses(selector) #extract the␣
↪institution address from the address_book object
        #self.log_message(f"institution_address = {institution_address}")
        address_dictionary = self.address_book.get_addresses() #extract all the address_
↪book entries

        new_message =  Message() #define a message object
        new_message.set_sender(self.myAddress) #set sender to the Actor's personal address
        new_message.set_directive("institution_message")#set directive
        new_message.set_payload(address_dictionary) #set the payload to the address_
↪dictionary we define above
        self.send(institution_address, new_message)
```

Institution Code Institution receives the **addresses dictionary** and adds it to its *address_book* object using *self.address_book.merge_addresses(addresses)*. We track the message attributes that are received and more specifically the

```
from mTree.microeconomic_system.environment import Environment
from mTree.microeconomic_system.institution import Institution
from mTree.microeconomic_system.agent import Agent
from mTree.microeconomic_system.directive_decorators import *
from mTree.microeconomic_system.message import Message
import math
import random
import logging
import time
import datetime
```

(continues on next page)

(continued from previous page)

```
@directive_enabled_class
class InstitutionClass(Institution):
    def __init__(self):
        pass

    @directive_decorator("institution_message")
    def institution_message(self, message:Message):

        environment_address = message.get_sender() #since environment is the message␣
↪sender, we can extract its address this way.
        address_dictionary = message.get_payload() #the Environment sends the address_
↪dictionary (obtained using self.address_book.get_addresses())

        #Logging Initial State
        self.log_message(f"Institution: environment_address = {environment_address}\n" #␣
↪the environment's address
                         f"Institution: address_dictionary = {address_dictionary}\n" #␣
↪the address_dictionary it received from the Environment
                         f"Institution: Initial self.address_book.get_addresses = {self.
↪address_book.get_addresses()} ") #logs the current addresses it has in its possession

        self.address_book.merge_addresses(address_dictionary)#we merge the addresses we␣
↪received with our own address_book object

        #Logging Final State
        self.log_message(f"Institution: Final self.address_book.get_addresses = {self.
↪address_book.get_addresses()}")
```

.log Output

We have listed the log outputs generated as a result of the **institution_message** directive.

```
1650482151.553274   Institution (institution_file.InstitutionClass 1) : About to enter␣
↪directive: institution_message
1650482151.556913   Institution: environment_address = ActorAddr-(T|:41851)
Institution: address_dictionary = {'institution_file.InstitutionClass 1': {'address_type
↪': 'institution', 'address': <thespian.actors.ActorAddress object at 0x4018103c10>,
↪'component_class': 'institution_file.InstitutionClass', 'component_number': 1, 'short_
↪name': 'institution_file.InstitutionClass 1'}, 'agent_file.AgentClass 1': {'address_
↪type': 'agent', 'address': <thespian.actors.ActorAddress object at 0x4018103b20>,
↪'component_class': 'agent_file.AgentClass', 'component_number': 1, 'short_name':
↪'agent_file.AgentClass 1'}, 'agent_file.AgentClass 2': {'address_type': 'agent',
↪'address': <thespian.actors.ActorAddress object at 0x4018103310>, 'component_class':
↪'agent_file.AgentClass', 'component_number': 2, 'short_name': 'agent_file.AgentClass 2
↪'}, 'agent_file.AgentClass 3': {'address_type': 'agent', 'address': <thespian.actors.
↪ActorAddress object at 0x4018103340>, 'component_class': 'agent_file.AgentClass',
↪'component_number': 3, 'short_name': 'agent_file.AgentClass 3'}, 'agent_file.
↪AgentClass 4': {'address_type': 'agent', 'address': <thespian.actors.ActorAddress␣
↪object at 0x4018101880>, 'component_class': 'agent_file.AgentClass', 'component_number
↪': 4, 'short_name': 'agent_file.AgentClass 4'}, 'agent_file.AgentClass 5': {'address_
↪type': 'agent', 'address': <thespian.actors.ActorAddress object at 0x4018101130>,
↪'component_class': 'agent_file.AgentClass', 'component_number': 5, 'short_name':
↪'agent_file.AgentClass 5'}}
```

(continues on next page)

(continued from previous page)

```
Institution: Initial self.address_book.get_addresses = {}
1650482151.5645256  Institution: Final self.address_book.get_addresses = {'institution_
↪file.InstitutionClass 1': {'address_type': 'institution', 'address': <thespian.actors.
↪ActorAddress object at 0x4018103c10>, 'component_class': 'institution_file.
↪InstitutionClass', 'component_number': 1, 'short_name': 'institution_file.
↪InstitutionClass 1'}, 'agent_file.AgentClass 1': {'address_type': 'agent', 'address':
↪<thespian.actors.ActorAddress object at 0x4018103b20>, 'component_class': 'agent_file.
↪AgentClass', 'component_number': 1, 'short_name': 'agent_file.AgentClass 1'}, 'agent_
↪file.AgentClass 2': {'address_type': 'agent', 'address': <thespian.actors.ActorAddress␣
↪object at 0x4018103310>, 'component_class': 'agent_file.AgentClass', 'component_number
↪': 2, 'short_name': 'agent_file.AgentClass 2'}, 'agent_file.AgentClass 3': {'address_
↪type': 'agent', 'address': <thespian.actors.ActorAddress object at 0x4018103340>,
↪'component_class': 'agent_file.AgentClass', 'component_number': 3, 'short_name':
↪'agent_file.AgentClass 3'}, 'agent_file.AgentClass 4': {'address_type': 'agent',
↪'address': <thespian.actors.ActorAddress object at 0x4018101880>, 'component_class':
↪'agent_file.AgentClass', 'component_number': 4, 'short_name': 'agent_file.AgentClass 4
↪'}, 'agent_file.AgentClass 5': {'address_type': 'agent', 'address': <thespian.actors.
↪ActorAddress object at 0x4018101130>, 'component_class': 'agent_file.AgentClass',
↪'component_number': 5, 'short_name': 'agent_file.AgentClass 5'}}
1650482151.5661018  Institution (institution_file.InstitutionClass 1) : Exited␣
↪directive: institution_message
```

Based on the output we can see that the message was properly processed according to our code because of the **entry logging** and **exit logging**(first and last lines respectively). We were also able to check that the payload (line number 3 of the code-block) was not **None**. Finally, we were able to monitor the state change of the **self.address_book.get_addresses()** in line 4 and the change that takes place in line 5.

Data Logging

self.log_data(content)

Data File

Interpret into Jupyter notebook

Simple suggestions on how to log data using dictionaries and little code on how pandas could be used to read the dataframe.

2.4.7 Error Handling

Directive name error

2.4.8 mTree Simulation State

2.5 Contribute

under development …

2.6 Learning Paths

Under development…

2.6.1 Tatonnement

2.6.2 Sealed Bid Auction

2.6.3 Ascending Price Auction

2.6.4 Descending Price Auction

2.6.5 Common Value Auction

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search