Mesa is a built-in package in Python that creates ABM using the built-in components.
Refer:

- https://mesa.readthedocs.io/en/stable/#:~:text=Mesa%20allows%20users%20to%20quickly,using%20Python%27s%20data%20analysis%20tools.
- https://buildmedia.readthedocs.org/media/pdf/mesa/pyconsprints/mesa.pdf
- https://mesa.readthedocs.io/en/stable/apis/space.html

Get started with $pip install mesa

Import the Agent and Model classes of mesa
```
from mesa import Agent, Model
```

Mesa.time handles the time component of the model. I contains schedulers that handles agent activations. RandomActivation is a scheduler that activates each agent once per step, in random order with order reshuffled every step
```
from mesa.time import RandomActivation
```

DataCollector is a standard way to collect data generated by a Mesa model. They can be model-level data, agent-level data and tables
```
from mesa.datacollection import DataCollector
```

To add spatial component to the model. Under Multigrid, each cell can contain a set of agents
```
from mesa.space import MultiGrid
```

Random model in python defines a series of functions to generate and manipulate random integers
```
import random
```

Numpy helps in creating multidimensional arrays in Python. Np is an alias to numpy so we can use np later on in the codes instead of numpy
```
import numpy as np
```

Sys stands for system specific parameter and functions
```
import sys
```

matplotlib is the plotting library used for data visualizations and plotting the results and data
```
from matplotlib.pyplot import *
```

Pandas has a data analysis library. And pd is alias for panda
```
import pandas as pd
```

Itertools helps in creating iterations. Product returns the cartesian product from each iteration
```
from itertools import product
```

Assigning arrays for the average of all 10 iterations:

Percentage sellers indicate that out of 500 agents 10% or 20% are sellers: 10% sellers = 50 sellers and 450 buyers
```
perc_sellers=[10,20,30,40,50,60,70,80,90]
```

Percentage Value Increased: Was initially being considered in place of GFT: Not a part of the model now

```
pvi_all=[]
```

Percentage of agents trading: E.g. if 75 agents have traded then (75/500)*100

```
trade_percent_all=[]
```

Quantity of water held by sellers

```
water_held_all_s=[]
```

Quantity of water held by buyers

```
water_held_all_b=[]
```

Allowable water to divert by the sellers

```
water_allow_all_s=[]
```

Allowable water to divert by the buyers

```
water_allow_all_b=[]
```

Average value of water held by sellers

```
av_all_s=[]
```

Average value of water held by buyers

```
av_all_b=[]
```

Overall GFT after 10 iterations:

```
gft_all_old=[]
gft_all_new=[]
```

For each round of iteration:

Range 10 indicates that the model will run 10 times for each drought number chosen below

```
for ppp in range(10):
```

If drought number is 50 it means agents 1 to 50 are sellers and agents 51 to 500 are buyers

```
    dr_num=[50,100,150,200,250,300,350,400,450]
```

Random seeds keep the characteristics of agents intact. They will not vary over rounds

```
    rand_seed=[11,13,15,17,19,21,23,25,27]
    rand_seed3=[10,12,14,16,18,20,22,24,26]
    rand_seed2=[1,2,3,4,5,6,7,8,9]
```

Explained earlier:

```
    gft_tot_old=[]
    gft_tot_new=[]
    trade_percent=[]
    pvi=[]
    water_held_tot_b=[]
```

```
water_allow_tot_b=[]
water_held_tot_s=[]
water_allow_tot_s=[]
av_tot_s=[]
av_tot_b=[]
```

len stands for length of an object:

```
for jj in range(len(dr_num)):
```

Number of agents = 500

```
a_once=500
for ii in range(500):
```

Main simulation portion:

```
print('Try number: ',ii+1)
```

Initialise the class of an agent:

```
class MyAgent(Agent):
```

Initialization method:

Unique id is assigned to each agent, AV stands for average value, endow = c, slope and intercept stands for the same from the yield equation in the ABM, c_b = $\bar{c}$, w_b = $\bar{w}$, allow_h2o = allowable water to divert for each water right, exo_price = exogenous price, conur = consumptive use rate, h20 =actual water withdrawn, ret = return flow, distrib_combo = the array that determines the exact location of the diversion of a water right along a river, techno = technology, senior = seniority rank, field = land acreage, tot_h2o = h20 =actual water withdrawn * land acreage, river_m = river miles, yield is the production of the crop portfolio, revenue is the revenue generated from selling or purchasing a water right, pr_wtp =Willingness to Pay, pr_wta = Willingness to accept, pr_bid =Bid Price and pr_ask = Ask Price

```
def __init__(self, unique_id,
model,AV,endow,slope,intercept,c_b,w_b,allow_h2o,conu,exo_price,conur,h2o,ret,distrib_
comb,techno,senior,field,tot_h2o,river_m,yield_agents,revenue,pr_wtp,pr_wta,pr_bid,pr_a
sk):
        super().__init__(unique_id, model)
        self.AV = AV
        self.endow = endow
        self.slope = slope
        self.intercept = intercept
        self.river_m=river_m
        self.c_b=c_b
        self.w_b=w_b
        self.allow_h2o=allow_h2o
        self.conu=conu
        self.exo_price=exo_price
        self.conur=conur
        self.h2o=h2o
```

self.ret=ret
self.distrib_comb=distrib_comb
self.techno=techno
self.senior=senior
self.field=field
self.tot_h2o=tot_h2o
self.yield_agents=yield_agents
self.revenue=revenue
self.pr_wtp=pr_wtp
self.pr_wta=pr_wta
self.pr_bid=pr_bid
self.pr_ask=pr_askType equation here.
self.price1=0.0 (Buying Price)
self.price2=0.0 (Selling Price)
self.gain=0.0

Self defines instance of a class. We check for if trade can happen:
def trade(self):

#CASE_1: self=Seller and self.other_agent=Buyer (the self-agent picked is a seller and the other agent picked is a buyer)
if self.senior <= dr_no and self.other_agent.senior > dr_no:

Buyer's bid price >= Sellers Ask price
if self.other_agent.pr_bid >= self.pr_ask:
Then Market price = bid price
self.price2 = self.other_agent.pr_bid updating selling price of the Seller
self.other_agent.price1 = self.other_agent.pr_bid updating buying price of the Buyer
ss_s[self.unique_id]=self.AV*self.c_b
abc=self.AV*self.c_b
self.endow = self.endow - self.c_b updating the consumptive use of Seller (initial endowment = $\bar{c}$ – amount sold (= $\bar{c}$) because sellers can sell everything or nothing at all
self.other_agent.endow = self.other_agent.endow + self.c_b updating the consumptive use of Buyer
pp[self.other_agent.unique_id]=self.other_agent.price1 storing Buying price
qq[self.unique_id]=self.price2 storing Selling price
self.gain = (self.other_agent.pr_wtp-self.pr_wta)*self.other_agent.endow Gain from one transaction = (WTP -WTA)*amount sold
self.other_agent.gain = (self.other_agent.pr_wtp-self.pr_wta)*self.other_agent.endow
rr[self.unique_id]=self.gain storing gain of Seller
rr[self.other_agent.unique_id]=self.other_agent.gain storing gain of Buyer
self.AV = -self.slope*self.endow + self.intercept updating the AV of Seller
self.other_agent.AV = -self.other_agent.slope*self.other_agent.endow + self.other_agent.intercept updating the AV of Buyer
(Refer to the equations in the ABM for the AV for $\bar{c}$ amount of water
cba=self.other_agent.AV*(self.other_agent.endow)
print(self.senior,self.other_agent.senior,cba,abc)
ss_b[self.other_agent.unique_id]=self.other_agent.AV*(self.other_agent.endow)

Removing the buyer from trade if the buyer has received $\bar{c}$

```
if self.other_agent.endow == self.other_agent.c_b:
self.model.schedule.remove(self.other_agent)
```

Removing the seller from trade if seller has already sold entire $\bar{c}$

```
if self.endow ==0:
self.model.schedule.remove(self)
```

If Bid Price of buyers < Ask Price of sellers (we check for if Ask Price <= WTP) Check model for equations. The rest of the part is same.

```
elif self.other_agent.pr_bid < self.pr_ask:
```

Selling Price = Ask Price

```
self.price2 = self.pr_ask
```

Buying Price = Bid Price

```
self.other_agent.price1 = self.pr_ask
ss_s[self.unique_id]=self.AV*self.c_b
abc=self.AV*self.c_b
self.endow = self.endow - self.c_b
self.other_agent.endow = self.other_agent.endow + self.c_b
pp[self.other_agent.unique_id]=self.other_agent.price1
qq[self.unique_id]=self.price2
self.gain = (self.other_agent.pr_wtp-self.pr_wta)*self.other_agent.endow
self.other_agent.gain = (self.other_agent.pr_wtp-self.pr_wta)*self.other_agent.endow
rr[self.unique_id]=self.gain
rr[self.other_agent.unique_id]=self.other_agent.gain
self.AV = -self.slope*self.endow + self.intercept
self.other_agent.AV = -self.other_agent.slope*self.other_agent.endow +
self.other_agent.intercept
cba=self.other_agent.AV*(self.other_agent.endow)
print(self.senior,self.other_agent.senior,cba,abc)
ss_b[self.other_agent.unique_id]=self.other_agent.AV*(self.other_agent.endow)
if self.other_agent.endow == self.other_agent.c_b:
self.model.schedule.remove(self.other_agent
if self.endow ==0:
self.model.schedule.remove(self)
```

`elif self.price2 > 0.0`: if Seller has already traded in this iteration, then its Selling price should not go to 0

```
qq[self.unique_id]=self.price2 + 0.0
```

#CASE_2: self=Buyer and self.other_agent=Seller

Absolutely similar to case 1 except the self would be buyer and other agent would be seller

There are two other cases: one in which both the agents picked are buyers and one in which both the agents picked are sellers. In both cases we have to ensure that the trade does not take place and the buying and selling prices are fixed at 0

```
elif self.senior > dr_no and self.other_agent.senior > dr_no:
pp[self.unique_id] = self.price1 + 0.0
qq[self.other_agent.unique_id] = self.other_agent.price2 + 0.0
qq[self.unique_id] = self.price2 + 0.0
pp[self.other_agent.unique_id] = self.other_agent.price1 + 0.0
elif self.senior <= dr_no and self.other_agent.senior <= dr_no:
pp[self.unique_id] = self.price1 + 0.0
qq[self.other_agent.unique_id] = self.other_agent.price2 + 0.0
qq[self.unique_id] = self.price2 + 0.0
pp[self.other_agent.unique_id] = self.other_agent.price1 + 0.0
```

To check if agents can trade at all:

```
def step(self):
if self.AV > 0.0: If AV <0 then the agent will not trade
self.trade() #function for trading
elif self.AV < 0.0:
pp[self.unique_id]=0.0 #Buying price set to 0
qq[self.unique_id]=0.0 #Selling price set to 0
else:
pass
```

We can assume the market price from either price 1 or price 2 as defined earlier because both them are same in case of a transaction (The buying price and selling price is same)

```
def compute_price(model):
agent_price = [agent.price1 for agent in model.schedule.agents]
return agent_price
```

Use Trade model to initiate interaction and assign the values to the parameter of the model:

```
class Trade_Model(Model):
def__init__(self,N,cu,ex_price,trib_comb,riv_m,cur,water,ret_fl,tech,sen_id,land,tot_water,
seed=None):
```

Global variable is that outside of a function

`global gamma` (gamma refers to drought severity = 1 at the moment from the model)

Number of agents

```
self.num_agents = N
```

Activate the iteraation

```
self.schedule = RandomActivation(self)  #activation of the interaction
self.running = True
```

Initialising empty arrays

`return_water=[]` – Return Flow

avg_val=[] – Average Value
endowment=[] – initial endowment of agents which is $\bar{c}$
sl=[] - Slope
interc=[] – Intercept
c_bar=[] – The amount of water that would provide maximum yields
w_bar=[] – $\bar{w}=\bar{c}$/Consumptive use rate
allow_water=[] – Amount of water that is allowed to be diverted
yield_all=[] – total yield
rev=[] – Revenue generated from sell and purchase of water rights
P_wtp=[] - Willingness to Pay
P_wta=[] - Willingness to Accept
P_bid=[] - Bid Price
P_ask=[] – Ask Price
gamma=1.0
eps_s=(dr_no)/self.num_agents - number of sellers is (dr_no)
eps_b=(self.num_agents-(dr_no))/self.num_agents - number of buyers is (num_agents-dr_no+1)

for i in range(self.num_agents):
if sen_list[i] <= dr_no: checking if sen_id is less than dr_no
sl.append(random.uniform(0,2)) – The slope ranges from 0 to 2 and is allotted randomly
interc.append(random.uniform(20,40)) – intercept ranges from 20 to 40 and is assigned randomly
c_bar.append(interc[i]/(2*sl[i])) – Obtained from the yield equation
endowment.append(c_bar[i])
avg_val.append(interc[i]-sl[i]*endowment[i])
P_wtp.append('NA')
P_wta.append(avg_val[i]*c_bar[i])
P_bid.append('NA')
P_ask.append(P_wta[i]*(1+gamma*eps_b))
w_bar.append(c_bar[i]/cur[i])
return_water.append(w_bar[i]-c_bar[i])
allow_water.append(w_bar[i]*land[i])
yield_all.append(interc[i]*c_bar[i]-sl[i]*(c_bar[i])**2)
rev.append(ex_price[i]*yield_all[i])

elif sen_list[i] > dr_no: checking if sen_id is more than dr_no
sl.append(random.uniform(0,2))
interc.append(random.uniform(20,40))#(2*sl[i]*cu[i])
c_bar.append(interc[i]/(2*sl[i]))
endowment.append(0)
avg_val.append(interc[i]-sl[i]*endowment[i])
P_wtp.append(avg_val[i]*c_bar[i])
P_wta.append('NA')
P_bid.append(P_wtp[i]*(1-gamma*eps_s))
P_ask.append('NA')

```
w_bar.append(c_bar[i]/cur[i])
return_water.append(w_bar[i]-c_bar[i])
allow_water.append(w_bar[i]*land[i])
yield_all.append(interc[i]*c_bar[i]-sl[i]*(c_bar[i])**2)
rev.append(ex_price[i]*yield_all[i])
```

`for i in range(self.num_agents):` runs over all the agents (one at a time)

Initialising each agent

```
a =
MyAgent(i,self,avg_val[i],endowment[i],sl[i],interc[i],c_bar[i],w_bar[i],allow_water[i],cu[i],ex_price[i],cur[i],water[i],ret_fl[i],trib_comb[i],tech[i],sen_list[i],land[i],tot_water[i],riv_m[i],yield_all[i],rev[i],P_wtp[i],P_wta[i],P_bid[i],P_ask[i])
```

Adding to the scheduler

```
self.schedule.add(a)
```

For collecting the data

```
self.datacollector = DataCollector(
model_reporters = {"Agent Price": compute_price},
agent_reporters={"Price1": "price1","Price2": "price2","Endowment": "endow","AV": "AV","Slope":"slope","Intercept":"intercept","c_bar":"c_b","w_bar":"w_b","Allowable_water":"allow_h2o","gft":"gain","Con_use":"conu","Exo_price":"exo_price","Con_use_r":"conur","Water":"h2o","Return":"ret","Trib_comb":"distrib_comb","Technology":"techno","Seniority":"senior","Land":"field","Total_water":"tot_h2o","River_m":"river_m","Yield":"yield_agents","Revenue":"revenue","Pr_wtp":"pr_wtp","Pr_wta":"pr_wta","Pr_bid":"pr_bid","Pr_ask": "pr_ask"})
```

Function for starting the interaction between agents

```
def mystep(self):
self.datacollector.collect(self)
self.schedule.step() #randomly starts trading
```

`a = a_once` - input("How many agents? : ")

`b = 1000` - input("How many steps? : ")

```
num_agents = int(a)
iterations = int(b)
num_of_agents=num_agents
dr_no =int(dr_num[jj]) #int(num_of_agents/2 +1 )
print('Random drought number is: ',dr_no)
```

Consumptive use amounts – Not used at the moment

```
num_list=[0.625,1.25,1.875,2.5,3.125,3.75,4.375,5
ex_price_list=[1,2,3,4,5,6,7,8] #crop list
tech_list=[1,2,3,4,5,6] #technology list
```

`sen_list=random.sample(range(1,num_of_agents+1),num_of_agents)` -list of seniority id

`low=0.001` -lower bound of consumptive use rate

`high=0.999` -upper bound of consumptive use rate

land_low=2 -lower bound of amount of land
land_high=172 -upper bound of amount of land

cu=[] -consumptive use
ex_price=[] -exogenous price of crop
cur=[] -consumptive use rate
water=[] -amount of water withdrawn
ret_fl=[] -return flow
tech=[] -technology
land=[] -amount of land
tot_water=[] -water*land
trib_comb=[] -tributary position
trib_comb_1=[] -temporary tributary position
trib_comb_list=[] -all possible tributary positions
riv_m_start=[] -start river miles range
riv_m_end=[] -end of river miles range
riv_m=[] -actual river miles for each agent

If you look at the tributary combination as depicted in that model document, you'll notice that the combination goes as follows--
1, 10, 11, 110, 111, 100, 101, 1100, 1101, 1110, 1111, 1000, 1001, 1010, 1011 .... [here 1 is main stream and then tributary to the right is 1 and to the left is 0]. So, if one looks closely one can see that these numbers are nothing but all the possible combinations of 0 and 1 of a given length (1,2,3,...) and let us denote the length of the number by 'k'. But, the series does not have any number that starts with 0 (because main stream is denoted as 1). So, we just have to write a function that creates combinations of 0 and 1 of all the desired lengths (say, k=5) and stores them in an array. Then from that array only pick those numbers that start with a 1. Two functions have been defined 'printAllKLength' and 'printAllKLengthRec' which serve this purpose. The 'printAllKLengthRec' function recursively creates all those combinations and stores them in the array 'trib_comb_1'. We actually call the function to create the array 'trib_comb_1'. Then the for loop we pick the combinations starting with 1 and store them in the array 'trib_comb_list'. In the current code, we pick k=4 but k can be chosen to whatever number we want. Once we have the array of all possible combinations, then we go ahead and assign a combination randomly to each of the agents. One thing to note here is that this combination of numbers is symmetric. This implies that there is tributary to the right if there is a tributary to the left. Real life scenarios are not always symmetric. So, to introduce asymmetry we can randomly choose some number combinations from the trib_comb_list array and remove them. This way we can model real life scenarios more closely. Next, we need the 'river_miles' for each agent. An agent with tributary combination '11101' should not have river miles more than an agent with tributary combination '101'. To make sure this does not happen, we fix a range of river miles depending on the length of the tributary combination number. So, in the code, if k=1 (i.e. the length of combination is 1) the range of river miles has to be in between 1 to 1000. If k=2 the range of river miles has to be between 1000 to 2000. Similarly, if k=3 the range of river miles has to be between 2000 to 3000 and so on. So, depending on the length of tributary combination (k) assigned to a particular agent, we generate two numbers and store them in 'riv_m_start' (corresponding to the start value of river miles) and 'riv_m_end' (corresponding to the end value of river miles). Then, for each agent we randomly choose a number within that range and store them in the array 'riv_m'.

```
def printAllKLength(set, k):

    n = len(set)
    printAllKLengthRec(set, "", n, k)

def printAllKLengthRec(set, prefix, n, k):
    if (k == 0) :
        trib_comb_1.append(prefix)
        return
    for i in range(n):
        newPrefix = prefix + set[i]
        printAllKLengthRec(set, newPrefix, n, k - 1)

set1=['0','1']
comb_no=5
for i in range(comb_no):
    k=i+1
    printAllKLength(set1,k)
trib_comb_1=np.array(trib_comb_1)
for i in range(len(trib_comb_1)):
    if trib_comb_1[i][0]=='1':
        trib_comb_list.append(trib_comb_1[i])
#print(trib_comb_list)
```