ABCE Documentation

Release 0.1

Davoud Taghawi-Nejad

April 26, 2012

CONTENTS

1 start Module 3

2 agent Module 7

3 world Module 17

4 write Module 19

5 Indices and tables 21

Python Module Index 23

Index 25

ABCE is a Python Agent-Based Complete Economy Protocol. Written by Davoud Taghawi-Nejad.

To write a ABCE model there are three steps:

1. define agent in AgentName.py using the 'Agent.py' prototype
2. modify this file

1. import agents
2. define action_list below [('which_agent', 'does_what'), ...]
3. define parameter suchs as the number_of_each_agent_type in parameter.csv
4. build_agents
5. declare some goods as resources

Further instructions contained in the files.

Contents:

CHAPTER

ONE

START MODULE

To write a ABCE model there are three steps:

```
(1) define agent in AgentName.py using the 'Agent.py' prototype
(2) modify this file
   (a) import agents
   (b) define action_list below [('which_agent', 'does_what'), ...]
   (c) define parameter suchs as the number_of_each_agent_type in parameter.csv
   (d) build_agents
   (e) declare some goods as resources
```

Further instructions contained in the files.

Example:

```
import sys
sys.path.append('/home/taghawi/Dropbox/workspace/eABMmaker/0SLAPP')
import world
from firm import Firm
from household import Household
from nature import Nature

for parameter in world.read_parameter('parameter.csv'):
   action_list = [('nature', 'assign'), ('household', 'recieve_connections'),
   ('household',  'report'), ('household', 'offer_capital'),
   ('firm', 'buy_capital'), ('household', 'search_work'),
   ('firm', 'hire_labor'), ('firm', 'report'), ('firm', 'production'),
   ('firm', 'report') ]
   w = world.World(parameter)
   w.add_action_list(action_list)
   w.build_agents(Firm, 'firm', 'number_of_firms')
   w.build_agents(Household, 'household', 'number_of_households')
   w.build_agents(Nature, 'nature', 1)
   w.declare_resource(resource='labor_endowment', productivity=1, product='labor')
   w.declare_resource(resource='capital_endowment', productivity=1, product='capital')
   w.run()
```

class `worldengine.`**`WorldEngine`**(*parameter*)

`add_action_list_from_file`(*parameter_name='action_list'*)
reads the action_list from the parameter file NOT YET IMPLEMENTED

`ask_agent`(*agent*, *command*)
This is only relevant when you derive your custom world/swarm not in start.py applying a method to a single agent

Args:

```
agent_name as string or using agent_name('group_name', number)
method: as string
```

ask_each_agent_in(*group_name*, *command*)

This is only relevant when you derive your custom world/swarm not in start.py applying a method to a group of agents group_name, method.

Args:

```
agent_group: using group_address('group_name', number)
method: as string
```

build_agents(*Agent*, *group_name*, *parameter_num_agents*)

This method creates agents, the first parameter is the agent class, the second parameter is a string with the group name of the agents the third parameter gives the name of the variable in parameter.csv

parameter_num_agents: number of agents to be created either a number or a string that is the header of a column in parameter.csv

declare_resource(*resource*, *productivity*, *product*, *command='default_resource'*, *group='all'*)

every resource you declare here produces productivity units of the product per round. For example, 'gold_mine' produces productivity units of 'gold', 'land' produces productivity units of 'harvest' and 'labor_endowment' produces productivity units of 'labor'. By default the resource is replentished at the begin of the round. You can change this. Insert the command string you chose it self.action_list. One command can be associated with several resources.

resources can be goupe specific, that means that when somebody except this group holds them they do not produce. The default is all and its better to keep the default 'all' except you have a very good reason to do so or if your model is running and you are optimizing. We recommend not to optimize before the simulation is working perfectly.

follow_agent(*group_name*, *number*)

This logs a particular agents variables after every subround. By default the agent's goods are tracked. You can change this by writing a custom follow(self) method that returns a dictionary in the agent. (details under agent.follow())

start_db(*group*, *variables='goods'*, *typ='FLOAT'*, *command='round_end'*)

writes variables of a group of agents into the database, by default the db write is at the end of the round. You can also specify a command and insert the command you choose in the action_list. If you choose a custom command, you can declace a function with the name command that returns the variable you want to track. Details are in agent.follow.

You can use the same command for several groups, that report at the same time.

Args:

```
agentgroup: can be either a group or a list of agents
variables: default='goods' monitors all the goods the agent owns
you can insert any variable your agent possesses. For
self.knows_latin you insert 'knows_latin'. If your agent
has self.technology you can use 'technology['formula']'
(typ='CHAR(50)'.
typ: the type of the sql variable (FLOAT, INT, CHAR(length))
command
```

Example:

```
w.start_trade_db(group='firm')
```

```
or

w.start_trade_db(agents_list=[agent_name('firm', 5), agent_name('household', 10))
```

CHAPTER

TWO

AGENT MODULE

The Agent class contains and registers an agents actions. Actions need to be registerd in order to be accessable from outside the class. When actions are in this class and registered they need to be called in the ModelSwarm class.

Example:

```
class Agent(AgentEngine):
   def __init__(self, arguments):
       AgentEngine.__init__(self, *arguments)
       self.create(’red_ball’, self.idn)
       self.create(’money’, 10)
       self.red_to_blue = create_production_function(’blue_ball=red_ball’)
       if self.idn == 0:
           self.painter = True
       else:
           self.painter = False
       self.report()
               HARDCODEDNUMBEROFAGENTS = 3  #hardcoding only for a simple example

   def give(self):
       # gives self.some_agent_name 1 ball, if he has one
       reciever = self.next_agent()
       if self.count(’red_ball’) > 0:
           self.sell(reciever, "red_ball", self.count(’red_ball’), 1)

   def get(self):
       # accepts all balls #
       offers = self.received_offers

       if offers:
           offer = offers.values()[0]
           try:
               self.accept(offer)
           except NotEnoughGoods:
               pass

       def report(self):
       print(self.idn, ’:’, "have", self.count(’blue_ball’), "blue balls", self.count(’red_ball’), "

   def next_agent(self):
       return agent_name(’agent’, (1 + self.idn) % HARDCODEDNUMBEROFAGENTS)
```

Start methods that start are not meant to be in the self.action with an underscore(‘_’), this makes the execution faster as the system does not have to keep track whether this methods are externally called.

class agent.**Agent** (*parameter*, *arguments*)

Bases: `agentengine.AgentEngine`, `agentengine.Firm`, `agentengine.Household`

The Agent class contains agents actions. if your agent is a Firm, meaning it uses production functions it must inherit from firm:

```
class Agent(AgentEngine, Firm):
```

if it is a utility maximizing household:

```
class Agent(AgentEngine, Household):
```

accept (*offer*)

The offer is accepted and cleared

Args:

```
offer: the offer the other party made
(offer not quote!)
```

accept_partial (*offer*, *percentage*)

TODO The offer is partly accepted and cleared

Args: offer: the offer the other party made (offer not quote!)

accept_quote (*quote*)

makes a commited buy or sell out of the counterparties quote

Args:: quote: buy or sell quote that is acceped

accept_quote_partial (*quote*, *quantity*)

makes a commited buy or sell out of the counterparties quote

Args:: quote: buy or sell quote that is acceped quantity: the quantity that is offered/requested it should be less than propsed in the quote, but this is not enforced.

assert_empty_messages ()

this method can be used to make sure that at the end of a round no recieved messages are in the que

buy (*receiver*, *good*, *quantity*, *price*)

commits to sell the quantity of good at price

The goods are not in haves or self.count(). When the offer is rejected they are automatically reacreditated. When the offer is accepted the money amount is accreditated. (partial acceptance accordingly)

Args: receiver: an agent name NEVER a group or 'all'!!! (its an error but with a confusing warning) 'good': name of the good quantity: maximum units disposed to buy at this price price: price per unit

consume (*input_goods*)

consumes input_goods returns utility according consumption

Args: {'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good consumed.

Raises: NotEnoughGoods: This is raised when the goods are insufficient. GoodDoesNotExist: This is raised when unknown goods are used.

Example:

```
self.consumption_set = {'car': 1, 'ball': 2000, 'bike':  2}
try:
    self.consume(utility_function, self.consumption_set)
except NotEnoughGoods:
    self.consume(utility_function, self.smaller_consumption_set)
```

consume_everything()

consumes everything that is in the utility function returns utility according consumption

Args:

```
utility_function: A utility_function produced with
:ref:‘create_utility_function‘,
:ref:‘create_cobb_douglas_utility_function‘ or
```

Raises:: GoodDoesNotExist: This is raised when unknown goods are used.

Example:

```
self.consume_everything()
```

count(*good*)

returns how much of good an agent has (0 when unknown)

create(*good*, *quantity*)

creates quantity of the good out of nothing

Use this create with care, as long as you use it only for labor and natural resources your model is macroeconomally complete.

Args: ‘good’: is the name of the good quantity: number

daemon

Return whether process is a daemon

exitcode

Return exit code of process or *None* if it has yet to stop

follow()

You can implement this fuction in your agent to track the changes of an agents variables or goods after each subround. If you do not implement this function the goods an agent owns are tracked. You track an agent by declaring in start.py: w.follow_agent(‘agent’, id_number)

You have to write a function that returns a dictionary. This dictionary will be tracked in the database.

Example:

```
start.py:
   ...
   w.follow_agent('firm', i)
   w.run
```

agent.py:

def follow(self): track = {} # track is a dictionary track.update(self._haves) # adds all goods of the agent to the dict track['knows_neighbor'] = self.knows_neighbor track['last_round_price'] = self.last_round_price return track # essential line

Because follow tracks from the beginnig on it might be necessary to set variable to None or any value in the __init__(...) method. E.G. self.last_round_price = 0; self.knows_neighbor = None

get_quotes()

self.quotes() returns all new quotes and removes them. The order is randomized.

Example:

```
quotes = self.get_quotes()
```

Returns:: list of quotes

get_quotes_biased()

like self.quotes(), but the order is not randomized, so its faster.

self.quotes() returns all new quotes and removes them. The order is randomized.

Use whenever you are sure that the way you process messages is not affected by the order.

ident

Return identifier (PID) of process or *None* if it has yet to start

is_alive()

Return whether process is alive

join(*timeout=None*)

Wait until child process terminates

mail(*receiver*, *message*)

sends a message to agent, agent_group or ‘all’. Agents receive it at the beginning of next round in self.messages()

messages(*typ=’m’*)

self.messages() returns all new messages send before this step (typ=’m’). The order is randomized self.messages(typ) returns all messages with a particular non standard type typ e.G. ‘n’. The order of the messages is randomized.

Example:

```
potential_buyers = self.messages('address')
for p_buyer in potential_buyers:
   print(p_buyer)
```

messages_biased(*typ=’m’*)

like self.messages(typ), but the order is not properly randomized, but its faster. use whenever you are sure that the way you process messages is not affected by the order

open_offers(*good*)

returns all open offers of the ‘good’, without deleting them

open_offers_all()

returns all open offers, without deleting them

pid

Return identifier (PID) of process or *None* if it has yet to start

produce(*production_function*, *input_goods*)

Produces output goods given the specified amount of inputs.

Transforms the Agent’s goods specified in input goods according to a given production_function to output goods. Automatically changes the agent’s belonging. Raises an exception, when the agent does not have sufficient resources.

Args: production_function: A production_function produced with *create_production_function*, *create_cobb_douglas* or *create_leontief* {‘input_good1’: amount1, ‘input_good2’: amount2 ...}: dictionary containing the amount of input good used for the production.

Raises: NotEnoughGoods: This is raised when the goods are insufficient. GoodDoesNotExist: This is raised when unknown goods are used.

Example:

```
self.car = {'tire': 4, 'metal': 2000, 'plastic':  40}
self.bike = {'tire': 2, 'metal': 400, 'plastic':  20}
try:
    self.produce(car_production_function, self.car)
except NotEnoughGoods:
    A.produce(bike_production_function, self.bike)
```

produce_use_everything(*production_function*)

Produces output goods from all input goods, used in this production_function, the agent owns.

Args:

```
production_function: A production_function produced with
:ref:`create_production_function`, :ref:`create_cobb_douglas` or
:ref:`create_leontief`
```

Raises:

```
GoodDoesNotExist: This is raised when unknown goods are used.
```

Example:

```
self.produce(car_production_function)
```

quote_buy(*receiver*, *good*, *quantity*, *price*)

quotes a price to buy quantity of 'good' to receiver

price (money) per unit offers a deal without checking or committing resources

Args: receiver: an agent name a group of agents on 'all' (names and group names can be generated with agent_name(group_name, id) and group_address(group_name)) 'good': name of the good quantity: maximum units disposed to buy at this price price: price per unit

quote_sell(*receiver*, *good*, *quantity*, *price*)

quotes a price to sell quantity of 'good' to receiver

price (money) per unit offers a deal without checking or committing resources

Args: receiver: an agent name a group of agents on 'all' (names and group names can be generated with agent_name(group_name, id) and group_address(group_name)) 'good': name of the good quantity: maximum units disposed to sell at this price price: price per unit

reject(*offer*)

The offer is rejected

Args: offer: the offer the other party made (offer not quote!)

retract(*offer*)

The agent who made a buy or sell offer can retract it

The offer an agent made is deleted at the end of the subround and the committeg good reapears in the haves. Howevery if another agent accepts in the same round the trade will be cleared and not retracted.

Args: offer: the offer he made with buy or sell (offer not quote!)

run()

internal

sell(*receiver*, *good*, *quantity*, *price*)

commits to sell the quantity of good at price

The goods are not in haves or self.count(). When the offer is rejected they are automatically reacreditated. When the offer is accepted the money amount is accreditated. (partial acceptance accordingly)

Args: receiver: an agent name NEVER a group or 'all'!!! (its an error but with a confusing warning) 'good': name of the good quantity: maximum units disposed to buy at this price price: price per unit

send_address (*receiver*, *name*)
Sends some agent's name ore agents group's address under which these agents recieve messages and objects to reciever (agent or group)

agents can receive addresses with:

potential_buyers = self.messages('address')

send_my_address (*receiver*, *typ='address'*)
Sends the agents own name under which the agents recieves messages and objects to reciever (agent or group)

agents can receive addresses with:

potential_buyers = self.messages('address')

set_cobb_douglas_utility_function (*multiplier*, *exponents*)
creates a Cobb-Douglas utility function

Utility_functions are than used as an argument in consume_with_utility, predict_utility and predict_utility_and_consumption.

Args: {'input1': exponent1, 'input2': exponent2 ...}: dictionary containing good names 'input' and correstponding exponents

Returns: A utility_function that can be used in consume_with_utility etc.

Example: self._utility_function = create_cobb_douglas({'bread' : 10, 'milk' : 1}) self.produce(self.plastic_utility_function, {'bread' : 20, 'milk' : 1})

set_utility_function (*formula*, *typ='from_formula'*)
creates a utility function from formula

Utility_functions are than used as an argument in consume_with_utility, predict_utility and predict_utility_and_consumption.

create_utility_function_fast is faster but more complicatedutility_function

Args: "formula": equation or set of equations that describe the utility function. (string) needs to start with 'utility = ...'

Returns: A utility_function

Example: formula = 'utility = ball + paint' self._utility_function = create_utility_function(formula) self.consume_with_utility(self._utility_function, {'ball' : 1, 'paint' : 2})

//exponential is ** not ^

set_utility_function_fast (*formula*, *input_goods*, *typ='from_formula'*)
creates a utility function from formula

Utility_functions are than used as an argument in consume_with_utility, predict_utility and predict_utility_and_consumption.

create_utility_function_fast is faster but more complicated

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are seperated by a ; [output]: list of all output goods (left hand sides of the equations)

Returns: A utility_function that can be used in produce etc.

Example: formula = 'utility = ball + paint'

self._utility_function = create_utility_function(formula, ['ball', 'paint']) self.consume_with_utility(self._utility_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

start()
Start child process

sufficient_goods(*input_goods*)
checks whether the agent has all the goods in the vector input

terminate()
Terminate process; sends SIGTERM signal or uses TerminateProcess()

utility_function()
the utility function should be created with: create_cobb_douglas_utility_function, create_utility_function or create_utility_function_fast

agentengine.**net_value**(*goods_vector*, *price_vector*)
Calculates the net_value of a goods_vector given a price_vector

goods_vectors are vector, where the input goods are negative and the output goods are positive. When we multiply every good with its according price we can calculate the net_value of the correstponding production. goods_vectors are produced by predict_produce(.)

Args: goods_vector: a dictionary with goods and quantities e.G. {'car': 1, 'metal': -1200, 'tire': -4, 'plastic': -21} price_vector: a dictionary with goods and prices (see example)

Example:

```
prices = {'car': 50000, 'tire': 100, 'metal': 10, 'plastic':  0.5}
value_one_car = net_value(predict_produce(car_production_function, one_car), prices)
value_two_cars = net_value(predict_produce(car_production_function, two_cars), prices)
if value_one_car > value_two_cars:
   produce(car_production_function, one_car)
else:
   produce(car_production_function, two_cars)
```

agentengine.**create_production_function**(*formula*, *typ='from_formula'*)
creates a production function from formula

A production function is a produceation process that produces the given input given input goods according to the formula to the output goods. Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

create_production_function_fast is faster but more complicated

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are seperated by a ;

Returns: A production_function that can be used in produce etc.

Example: formula = 'golf_ball = (ball) * (paint / 2); waste = 0.1 * paint' self.production_function = create_production_function(formula, 'golf', 'waste') self.produce(self.production_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

agentengine.**create_production_function_fast**(*formula*, *output_goods*, *input_goods*, *typ='from_formula'*)
creates a production function from formula, with given outputs

A production function is a producetion process that produces the given input given input goods according to the formula to the output goods. Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

Args: "formula": equation or set of equations that describe the production process. (string) Several equation are seperated by a ; [output]: list of all output goods (left hand sides of the equations)

Returns: A production_function that can be used in produce etc.

Example: formula = 'golf_ball = (ball) * (paint / 2); waste = 0.1 * paint' self.production_function = create_production_function(formula, 'golf', 'waste') self.produce(self.production_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

`agentengine.`**`create_cobb_douglas`**`(`*output*, *multiplier*, *exponents*`)`

creates a Cobb-Douglas production function

A production function is a produceation process that produces the given input given input goods according to the formula to the output good. Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

Args: 'output': Name of the output good multiplier: Cobb-Douglas multiplier {'input1': exponent1, 'input2': exponent2 ...}: dictionary containing good names 'input' and correstponding exponents

Returns: A production_function that can be used in produce etc.

Example: self.plastic_production_function = create_cobb_douglas('plastic', {'oil' : 10, 'labor' : 1}, 0.000001) self.produce(self.plastic_production_function, {'oil' : 20, 'labor' : 1})

`agentengine.`**`create_leontief`**`(`*output*, *utilization_quantities*, *multiplier=1*, *isinteger='int'*`)`

creates a Leontief production function

A production function is a produceation process that produces the given input given input goods according to the formula to the output good. Production_functions are than used as an argument in produce, predict_vector_produce and predict_output_produce.

Warning, when you produce with a Leontief production_function all goods you put in the produce(...) function are used up. Regardless whether it is an efficient or wastefull bundle

Args: 'output': Name of the output good {'input1': utilization_quantity1, 'input2': utilization_quantity2 ...}: dictionary containing good names 'input' and correstponding exponents multiplier: multipler isinteger='int' or isinteger='': When 'int' produce only integer amounts of the good. When '', produces floating amounts.

, str(input_quantity)

Returns: A production_function that can be used in produce etc.

Example: self.car_technology = create_leontief('car', {'tire' : 4, 'metal' : 1000, 'plastic' : 20}, 1) two_cars = {'tire': 8, 'metal': 2000, 'plastic': 40} self.produce(self.car_technology, two_cars)

`agentengine.`**`predict_produce`**`(`*production_function*, *input_goods*`)`

Returns a vector with input (negative) and output (positive) goods

Predicts the production of produce(production_function, input_goods) and the use of input goods. net_value(.) uses a price_vector (dictionary) to calculate the net value of this production.

Args: production_function: A production_function produced with create_production_function, create_cobb_douglas or create_leontief {'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good used for the production.

Example:

```
prices = {'car': 50000, 'tire': 100, 'metal': 10, 'plastic':  0.5}
value_one_car = net_value(predict_produce(car_production_function, one_car), prices)
value_two_cars = net_value(predict_produce(car_production_function, two_cars), prices)
if value_one_car > value_two_cars:
   A.produce(car_production_function, one_car)
else:
   A.produce(car_production_function, two_cars)
```

agentengine.**predict_produce_output** (*production_function*, *input_goods*)

Calculates the output of a production (but does not preduce)

> Predicts the production of produce(production_function, input_goods) see also: Predict_produce(.) as it returns a calculatable vector

Args: production_function: A production_function produced with create_production_function, create_cobb_douglas or create_leontief {'input_good1': amount1, 'input_good2': amount2 ...}: dictionary containing the amount of input good used for the production.

Example:

```
print(A.predict_output_produce(car_production_function, two_cars))
>>> {'car': 2}
```

exception agent.**NotEnoughGoods** (*agent_name*, *good*, *amount_missing*)

Methods raise this exception when the agent has less goods than needed

This functions (self.produce, self.offer, self.sell, self.buy) should be encapsulated by a try except block:

```
try:
   self.produce(...)
except NotEnoughGoods:
   alternative_statements()
```

exception agent.**GoodDoesNotExist** (*agent_name*, *good*)

The good 'self.good' does not exist for this agent (usually a programming error)

CHAPTER

THREE

WORLD MODULE

Using this file is not necessary. However if you want to create special action groups you can copy this file in your working directory and extent it. If it is not in your working directory everything will work anyway.

The World class defines special ActionGroups that replace the (‘which_agent’, ‘does_what’) entries in the action_list.

In this example every round first all agents raise there hands, then one random agent jumps.

ACTIONLIST:

self.actionList, is the sequence in which the actions are executed each round. It must be a list contain a string. The names must be the same as in the “def” function in this class below. self.actionList must be declared in the “def __init__(...):”. the indentation must by two tabs (or 8 spaces).

For example:

```
self.actionList = [('which_agent', 'does_what'),
        "raise_hands", "one_agent_jumps"].
```

ACTIONGROUPS:

For example:

from worldengine import *

class World(WorldEngine): # Calls the ActionGroups in the order specified in self.actionList. def __init__(self, parameter_file):

WorldEngine.__init__(self, parameter_file) self.action_list = [”give”, “get”, “report”]

def give(self): self.ask_each_agent_in(“agent”, “give”)

def get(self): self.ask_each_agent_in(“agent”, “get”)

def report(self): self.ask_agent(“agent_0”, “report”) time.sleep(0.1) self.ask_agent(“agent_1”, “report”) time.sleep(0.1) self.ask_agent(“agent_2”, “report”)

Start methods that start are not meant to be in the self.action with an underscore(‘_’), this makes the execution faster as the system does not have to keep track whether this methods are externally called.

ATTENTION: if you write mylist = self.agent_list and you change mylist. (for example: mylist.remove(3)) self agent_list will be changed. In order avoid this make a copy: write mylist = self.agent_list[:]

class world.**World**(*parameter_file*)

Bases: worldengine.WorldEngine

Calls the ActionGroups in the order specified in self.actionList.

add_action_list_from_file(*parameter_name='action_list'*)

reads the action_list from the parameter file NOT YET IMPLEMENTED

ask_agent (*agent*, *command*)
applying a method to an instance of a class agent, method, dict. of the parameter (may be empty)

ask_each_agent_in (*group_name*, *command*)
applying a method to a collection of instances collection, method, dict. of the parameter (may be empty)

build_agents (*Agent*, *group_name*, *parameter_num_agents*)
This method creates agents, the first parameter is the agent class, the second parameter is a string with the group name of the agents the third parameter gives the name of the variable in parameter.csv

parameter_num_agents: number of agents to be created either a number or a string that is the header of a column in parameter.csv

declare_resource (*resource*, *productivity*, *product*, *command='default_resource'*, *group='all'*)
every resource you declare here produces productivity units of the product per round. For example, 'gold_mine' produces productivity units of 'gold', 'land' produces productivity units of 'harvest' and 'labor_endowment' produces productivity units of 'labor'. By default the resource is replentished at the begin of the round. You can change this. Insert the command string you chose it self.action_list. One command can be associated with several resources.

resources can be goupe specific, that means that when somebody except this group holds them they do not produce. The default is all and its better to keep the default 'all' except you have a very good reason to do so or if your model is running and you are optimizing. We recommend not to optimize before the simulation is working perfectly.

parameter (*parameter*)
returns the value or string of the parameter form the current line/simulation in parameter.csv or parameter_file

start_db (*group*, *variables='goods'*, *typ='FLOAT'*, *command='round_end'*)
writes variables of a group of agents into the database, by default the db write is at the end of the round. You can also specify a command and insert the command you choose in the action_list.

You can use the same command for several groups, that report at the some time

Args:

```
agentgroup: can be either a group or a list of agents
variables: default='goods' monitors all the goods the agent owns
you can insert any variable your agent possesses. For
self.knows_latin you insert 'knows_latin'. If your agent
has self.technology you can use 'technology['formula']'
(typ='CHAR(50)'.
typ: the type of the sql variable (FLOAT, INT, CHAR(length))
command
```

Example:

```
w.start_trade_db(group='firm')

or

w.start_trade_db(agents_list=[agent_name('firm', 5), agent_name('household', 10))
```

CHAPTER

FOUR

WRITE MODULE

write.**line**(*sign=’*’*, *text=’‘*, *length=39*)

write.**write**(**k*)

writes prints without carrier feed:

```
write(’text’, ’’) => text
write(’text’) => <text>
write(’text’,’{’) => {text}
write(’text’, ’<msg) => <msg:text>
write(’text’,’msg:’) => msg:
```

text ===

write.**writeln**(**k*)

writes prints:

```
write(’text’, ’’) => text
write(’text’) => <text>
write(’text’,’{’) => {text}
write(’text’, ’<msg) => <msg:text>
write(’text’,’msg:’) => msg:
```

text ===

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

a

agent, 7

w

world, 17
write, 19

INDEX

A

accept() (agent.Agent method), 8
accept_partial() (agent.Agent method), 8
accept_quote() (agent.Agent method), 8
accept_quote_partial() (agent.Agent method), 8
add_action_list_from_file() (world.World method), 17
add_action_list_from_file() (worldengine.WorldEngine method), 3
Agent (class in agent), 7
agent (module), 7
ask_agent() (world.World method), 17
ask_agent() (worldengine.WorldEngine method), 3
ask_each_agent_in() (world.World method), 18
ask_each_agent_in() (worldengine.WorldEngine method), 4
assert_empty_messages() (agent.Agent method), 8

B

build_agents() (world.World method), 18
build_agents() (worldengine.WorldEngine method), 4
buy() (agent.Agent method), 8

C

consume() (agent.Agent method), 8
consume_everything() (agent.Agent method), 8
count() (agent.Agent method), 9
create() (agent.Agent method), 9
create_cobb_douglas() (in module agentengine), 14
create_leontief() (in module agentengine), 14
create_production_function() (in module agentengine), 13
create_production_function_fast() (in module agentengine), 13

D

daemon (agent.Agent attribute), 9
declare_resource() (world.World method), 18
declare_resource() (worldengine.WorldEngine method), 4

E

exitcode (agent.Agent attribute), 9

F

follow() (agent.Agent method), 9
follow_agent() (worldengine.WorldEngine method), 4

G

get_quotes() (agent.Agent method), 9
get_quotes_biased() (agent.Agent method), 10
GoodDoesNotExist, 15

I

ident (agent.Agent attribute), 10
is_alive() (agent.Agent method), 10

J

join() (agent.Agent method), 10

L

line() (in module write), 19

M

mail() (agent.Agent method), 10
messages() (agent.Agent method), 10
messages_biased() (agent.Agent method), 10

N

net_value() (in module agentengine), 13
NotEnoughGoods, 15

O

open_offers() (agent.Agent method), 10
open_offers_all() (agent.Agent method), 10

P

parameter() (world.World method), 18
pid (agent.Agent attribute), 10
predict_produce() (in module agentengine), 14
predict_produce_output() (in module agentengine), 15
produce() (agent.Agent method), 10
produce_use_everything() (agent.Agent method), 11

Q

quote_buy() (agent.Agent method), 11
quote_sell() (agent.Agent method), 11

R

reject() (agent.Agent method), 11
retract() (agent.Agent method), 11
run() (agent.Agent method), 11

S

sell() (agent.Agent method), 11
send_address() (agent.Agent method), 12
send_my_address() (agent.Agent method), 12
set_cobb_douglas_utility_function() (agent.Agent method), 12
set_utility_function() (agent.Agent method), 12
set_utility_function_fast() (agent.Agent method), 12
start() (agent.Agent method), 13
start_db() (world.World method), 18
start_db() (worldengine.WorldEngine method), 4
sufficient_goods() (agent.Agent method), 13

T

terminate() (agent.Agent method), 13

U

utility_function() (agent.Agent method), 13

W

World (class in world), 17
world (module), 17
WorldEngine (class in worldengine), 3
write (module), 19
write() (in module write), 19
writeln() (in module write), 19