ABCE Documentation

Release 0.2alpha

Davoud Taghawi-Nejad

July 23, 2012

CONTENTS

ABCE is a Python Agent-Based Complete Economy Protocol. Written by Davoud Taghawi-Nejad. The walk through, explains you how to set up a simulation. In the agent model you learn how to set up an agent and how to trade with other agents. The households and firms classes allow you to produce with different production functions and consume with according production functions. In order to get result you need to use the logging and panel data creation tools described in Obserning Agents.

CHAPTER

ONE

INSTALLATION

The installation has two parts. Installing the necessary software packages. Retrieving ABCE with git or as a zip file.

In terminal (necessary software) [1]

```
sudo apt-get install python-pyparsing python-numpy python-scipy python-zmq
```

Alternative 1 as a zip (EASY):

1. download the zip file from: https://github.com/DavoudTaghawiNejad/abce
2. extract zip file

Alternative 2 via git [2] in terminal (RECOMMENDED):

```
[register with git and automatize a secure connection with your computer]
sudo apt-get install git
mkdir abce
cd abce
git init
git pull git@github.com:DavoudTaghawiNejad/abce.git
```

Optional for development you can install sphinx and sphinx-apidoc. sphinx-apidoc currently needs the newest version of sphinx, they system that created this documentation

[1] possible you have to install sqlite3 and the according python bindings

[2] Git is a a version controll system. It is higly recommended to use it to make your development process more efficient and less error prone. http://gitimmersion.com/

CHAPTER

TWO

WALK THROUGH

This tutorial is a step by step guide to create an agent-based model with ABCE. In the following two text boxes, we the two concepts that make ABCE special are explaind: The explict modelling of trade able goods. And the physical closedness of the economy. (Which with some care leads to a general model - al in general equilibrum. (Of cause without equilibrum)

Objects the other ontological object of agent-based models.

Objects have a special stance in agent-based modeling:

- objects can be recovered (resources)
- exchanged (trade)
- transformed (production)
- consumed (transformed :-))
- destroyed (not really) and time depreciated

ABCE, takes automatically care of Trade, production/transformation and consumption of goods. Good categories can also be made to perish or regrow. Services and renting can so be modeled.

Closed economy When we impose that goods can only be transformed. The economy is physically closed (the economy is stock an flow consistent). When the markets are in a complete network our economy is complete. Think "general" in equilibrum economics.
Caveats: If agents horde without taking their stock into account its like destruction.

With this basic understanding you can now start writing your own model.

To create a model you basically have to follow three steps:

1. **Specify endowments that repletish every round and goods/services** that perish
2. Write the timeline
3. Write the agents with their actions

There is of course a littly bit of admin you have to do:

1. import agents in the model
2. specify parameters

2.1 Have a look on the *abce/examples/* folder

It is instructive to look an a simple example, for example the 2x2 economy. Than you can make a working copy of the templote or a copy of an example. (you need to change the *abce/lib* path in *start.py*)

2.2 Make a working copy

1. copy abce/example to your_model_path:

```
cd your_model_path
cp ../abce/template/* .
```

2. In line 3 of *start.py*: adjust sys.path.append('your_path_to/abce/lib')

2.3 start.py

In start.py the simulation, thus the parameters, objects, agents and time line are set up. Further it is declared, what is observed and written to the database. [1]

```
from Firm import Firm
from Household import Household
```

Here the Agent class Firm is imported from the file Firm.py. Likewise the Household class.

ABCE, reads the model parameter from a spreed sheet, every line is one simulation:

```
for parameters in simulation.read_parameters('simulation_parameters.csv'):

    ...
```

With the parameters ABCE loops over the intended line, to create the simulation and than runs the simulation. (after that it reads the next line an loops again). The variable parameters contains all parmeters from 'simulation_parameters.csv'. See *simulation_parameters and agent_parameters* for details.

To set up a new model, you create a class a that will comprise your model:

```
s = Simulation(parameters)

...
```

After this the timeline, agents and objects are added in arbitrary order to the model.

```
action_list = [
('Household', 'offer_capital'),
('Firm', 'buy_capital'),
('Household', 'search_work'),
('Firm', 'hire_labor'),
('Firm', 'production'),
'after_sales_before_consumption'
('Household', 'consumption')
]
s.add_action_list(action_list)
```

This establishes the order of the simulation. It can also be read from file .. _add_action_list_from_csv_file

In order to add an agent which was imported before we simply build this agents:

```
s.build_agents(Firm, 'number_of_firms')
s.build_agents(Household, 10)
```

[1] from __future__ import division, instructs python to handel division always as a floating point divison. Use this in all your python code. If you do not use this `3 / 2 = 1` instead of `3 / 2 = 1.5` (floor division).

The number of firms to be build is read from the column in simulation_parameters.csv called number_of_firms. The number of households on the other side is fixed at 10.

Goods are declared by goods classes. A normal good needs not to be declared. Below in the text you will see how to declare, perishable goods and periodically renewed endowments (resources) [The goods]

Or you can create panal data from agoup of agents:

```
s.panel_db('Firm', command='after_sales_before_consumption')
s.panel_db('all')  # at the beginning
```

For both follow_agent and panal_db you can controll what they log see [todo]

...

s.run()

In the remainder of this Walk through, we will see how the timeline, the goods and the agents are created.

2.3.1 The timeline: The order of actions within a round

Every agents-based model is caracterized by the order of which the actions are executed. In ABCE, there are rounds, every round is composed of subrounds, in which all agents, a group of agents or a single agent, act in parallel. In the code below you see a typical subround.

You have to declare an action_list, that is made of tuples telling ABCE which agent or agent group, should execute which method:

```
action_list = [
repeat([
    ('Household', 'offer_capital'),
    ('Firm', 'buy_capital'),
],
repetitions=10),
('Household', 'search_work'),
('Firm', 'hire_labor'),
('Firm', 'production'),
'after_sales_before_consumption',
('Household', 'consumption')
]
s.add_action_list(action_list)
```

The first tuple for example tells all Household agents to execute the methode "offer_capital". The 'after_sales_before_consumption' is a database command. see [panel_db].

The repeat function allows to repeat actions within the brackets a determinate amount of time.

Interactions happen between subrounds. An agent, sends a message in one round. The recieving agent, recieves the message the following subround. A trade is finished in three rounds: (1) an agend sends an offer. (footnote: the offered good is blocked, so it can not be sold twice or used before delivery.) (2) the other agent accepts (or rejects it). (3) The good is automatically delivered. (footnote: if the trade was rejected: the blocked good is unblocked.)

2.3.2 The goods

A normal good can be traded and transformed = used for production or consumption. You do not have to do anything, about normal goods it is sufficient to create with `abceagent.Agent.create()` them in the agent's function.

If an agent receives and endowment every round this can be automatically handled:

`abce.Simulation.declare_round_endowment()`

Also goods that last only one round, can automatically perish. `abce.Simulation.declare_perishable()`

This for example is usefull for labor service, when it is not used in a production simply disappears at the end of the round.

One important remark, for a logically consistent **macro-model** it is best to not create any goods during the simulation, but only in `abceagent.Agent.__init__()`. During the simulation the only new goods should be created by declare_round_endowment. In this way the economy is physically closed. An exception is of course money.

2.4 The agents

Agents are modeled in a separate file. In the template directory, you will find three agents: agent.py, firm.py and household.py.

An agent has to import the :module:abceagent module and some helpers:

```
import abceagent
from abcetools import is_zero, is_positive, is_negative, NotEnoughGoods
```

This imports the base classes: abceagent, Household and Firm.

An agent is a class and must at least inherit abceagent.Agent. *Trade*, *Messaging* and *Database* are automatically inherited:

```
class Agent(abceagent.Agent):
```

To create an agent that can also consume:

```
class Household(abceagent.Agent, abceagent.Household):
```

You see our Household agent inherits from abceagent, which is compulsary and Household. Household an the other-hand are a set of methods that are unique for Household agents. (there is also a Firm class)

2.4.1 The __init__ method

```
def __init__(self, simulation_parameters, own_parameters, _pass_to_engine):
    abceagent.__init__(self, *_pass_to_engine)
    self.create('labor_endowment', 1)
    self.create('capital_endowment', 1)
    self.set_cobb_douglas_utility_function({"MLK": 0.300, "BRD": 0.700})
    self.prices = {}
    self.prices['labor'] = 1
    self.number_of_firms = simulation_parameters['number_of_firms']
    self.renter = random.randint(0, 100)
    self.last_utility = None
```

The __init__ method is the method that is called when the agents are created (by the the :func:simulationengine:build_agents are :func:simulationengine.build_agents_from_file method.) In this method agents can access the simulation_parameters from the 'simulation_parameters.csv'. If the agents are build using :func:simulationengine.build_agents_from_file, own_parameters grants you access to the row for this agent in 'agents_parameters.csv'.

Line 2 is compulsory to pass the parameters to the abceagent.

With self.create the agent creates out of nothing the good 'labor_endowment'. Any good can be created. Generally speaking. The __init__ method is the only place where it is consistent to creat a good. (except for money, if you simulate a naive central bank).

This agent class inherited :meth:self.set_cobb_douglas_utility_function from :class:abceagent.Household. With :func:self.set_cobb_douglas_utility_function you can create a cobb-douglas function. Other functional forms are also available.

self.prices is a user dictionary. That saves prices for specific goods. Here the price for labor is set to 1.

In order to let the agent remember a simulation_parameter it has to be saved in the self domaine of the agent. (self.number_of_firms = simulation_parameters['number_of_firms'])

There is a random number assigned to self.renter and self.last_utility is intialized with None. It is often necessary to intialize variable in the __init__ method to avaid errors in the first round.

2.4.2 The action methods and a consuming Household

All the other methods of the agent are executed when the corresponding subround is called in start.py. [2]

so when in the action list *('household', 'eat')* s called the eat methode is executed:

```
class Agent(abceagent.Agent, abceagent.Household)
    def __init__(self):
        self.set_cobb_douglas_utility_function({'cookies': 0.9', 'bread': 0.1})
        self.create('cookies', 1)
        self.create('bread', 5)

...
def eat(self):
    utility = self.consume_everything()
    self.log('utility', {'a': utility})
```

In the above example we see how a utility function is declared and and how and agent cosumes. The utility is logged and con be retrieved see *Retrival of the simulation results*

2.4.3 Firms and Production functions

Firms do two things they produce (transform) and trade. [3] The following code shows you how to declare a technology and produce brad from labor and yeast.

```
class Agent(abceagent.Agent, abceagent.
def init(self):
   set_cobb_douglas('BRD', 1.890, {"yeast": 0.333, "LAB": 0.667})
    ..
def productionround(self):
    self.produce_use_everything()
```

More details in `abceagent.Firm`. `abceagent.FirmMultiFirm` offers a more advanced interface for firms with complicated technologies.

2.4.4 Trade

ABCE handles trade fully automatically. That means, that goods are automatically blocked and exchange. The modeler has only to decide when the agent offers a trade and set the criteria to accept the trade:

[2] With the exception of methods, whose names start with a '_' underscore.underscoring methods that the agent uses only internaly can speed up your code.

[3] We are aware that this is not entirely accurate, they also lobby to maximize their profit.

```
# Agent 1
def selling(self):
    offerid = self.sell(buyer, 'BRD', 1, 2.5)
    self.checkorders.append(offerid)

# Agent 2
def buying(self):
    offers = self.get_offers('cookies')
    for offer in offers:
       try:
          self.accept(offer)
       except NotEnoughGoods:
          self.reject(offer)
```

You can find a detailed explenation how trade works is `abceagent.Trade`

2.4.5 Dataproduction

There are three different ways of observing your agents:

Trade Logging

ABCE by default logs all trade and creates a SAM or IO matrix.

Manual in agent logging

An agent can log a variable, `abceagent.Agent.possessions()`, :meth:abceagent.Agent.possessions_all‘ and must other methods such as `abceagent.Firm.produce()` with `log()` or a change in a variable with `log_change()`:

```
self.log('possessions', self.possesions_all())
self.log('custom', {'price_setting': 5: 'production_value': 12})
prod = self.production_use_everything()
self.log('current_production', prod)
```

Panel Data

`panel_data()` creates panel data for all agents in a specific agent group at a specific point in every round. It is set in start.py:

```
s.paneldb('Household', command='aftersalesbeforeconsumption')
```

The command has to be inserted in the action_list.

How to retrieve the Simulation results is explained in *simulation_results*

CHAPTER

THREE

AGENTS

The abceagent.Agent class is the basic class for creating your agent. It automatically handles the possession of goods of an agent. In order to produce/transforme goods you need to also subclass the abceagent.Firm [1] or to create a consumer the abceagent.Household.

For detailed documentation on:

Trading: see abceagent.Trade

Logging and data creation: see abceagent.Database and *Retrival of the simulation results*

Messaging between agents: see abceagent.Messaging.

exception abcetools.**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()
```

class abceagent.**Agent**(*idn*, *group*, *_Agent__command_addresse*, *trade_logging*)

Bases: abceagent.Database, abceagent.Trade, abceagent.Messaging, multiprocessing.process.Process

Every agent has to inherit this class. It connects the agent to the simulation and to other agent. The abceagent.Trade, abceagent.Database and abceagent.Messaging classes are include. You can enhance an agent, by also inheriting from abceagent.Firm.:class:*abceagent.FirmMultiTechnologies* or abceagent.Household.

For example:

```
class Household(abceagent.Agent, abceagent.Household):
    def __init__(self, simulation_parameters, agent_parameters, _pass_to_engine):
    abceagent.Agent.__init__(self, *_pass_to_engine)
```

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

[1] or abceagent.FirmMultiTechnologies for simulations with complex technologies.

destroy(*good*, *quantity*)
destroys quantity of the good,

Args:

```
'good': is the name of the good
quantity: number
```

Raises:

```
NotEnoughGoods: when goods are insufficient
```

destroy_all(*good*)
destroys all of the good, returns how much

Args:

```
'good': is the name of the good
```

possession(*good*)
returns how much of good an agent possesses (0 when unknown)

Returns: A number.

possession does not return a dictionary for self.log(...), you can use self.possessions([...]) (plural) with self.log.

Example:

if self.possession('money') < 1: self.financial_crisis = True

if not(is_positive(self.possession('money')): self.bancrupcy = True

possessions(*list_of_goods*)
returns a dictionary of goods and the correstponding amount an agent owns

Argument: A list with good names. Can be a list with a single element.

Returns: A dictionary, that can be used with self.log(..)

Examples:

```
self.log('buget', self.possesions(['money']))

self.log('goods', self.possesions(['gold', 'wood', 'grass']))

have = self.possessions(['gold', 'wood', 'grass']))
for good in have:
    if have[good] > 5:
        rich = True
```

possessions_all()
returns all possessions

possessions_filter(*goods=None*, *but=None*, *match=None*, *typ=None*)
returns a subset of the goods an agent owns:

Args:

goods (list, optional): a list of goods to return

but(list, optional): all goods but the list of goods here

match(string, optional TODO): goods that match pattern

begin_with(string, optional TODO): all goods that begin with string

end_with(string, optional TODO) all goods that end with string

is(string, optional TODO)

'resources': return only goods that are endowments

'perishable': return only goods that are perishable

'resources+perishable': goods that are both

'produced_by_resources': goods which can be produced by resources

Example:

```
self.consume(self.possessions_filter(but=['money']))
# This is redundant if money is not in the utility function
```

round_begin()

If you declare a round_begin methode in your agent, it will be called at the begin of each round

run()

CHAPTER

FOUR

TRADE

class abceagent.**Trade**

Agents can trade with each other. The clearing of the trade is taken care of fully by ABCE. Selling a good works in the following way:

1. An agent sends an offer. `sell()`

   The good offered is blocked and self.possession(...) does not account for it.

2. **Next subround:** An agent receives the offer `get_offers()`, and can `accept()`, `reject()` or partially accept it. `accept_partial()`

   The good is credited and the price is deducted from the agent's possesions.

3. **Next subround:**

   - in case of acceptal *the money is automatically credited.*
   - in case of partial acceptance *the money is credited and part of the blocked good is unblocked.*
   - in case of rejection *the good is unblocked.*

Analogously for buying. (`buy()`)

Example:

```
# Agent 1
def sales(self):
    self.remember_trade = self.sell('Household', 0, 'cookies', quantity=5, price=self.price)

# Agent 2
def receive_sale(self):
    oo = self.get_offers('cookies')
    for offer in oo:
        if offer['price'] < 0.3:
            try:
                self.accept(offer)
            except NotEnoughGoods:
                self.accept_partial(offer, self.possession('money') / offer['price'])
        else:
            self.reject(offer)

# Agent 1, subround 3
def learning(self):
    offer = self.info(self.remember_trade)
    if offer['status'] == 'reject':
        self.price *= .9
```

```
elif offer['status'] = 'partial':
    self.price *= offer['counter_quantity'] / offer['quantity']
```

Quotes on the other hand allow you to ask a trade partner to send you a not committed price quote. The modeller has to implement a response mechanism. For convenience `accept_quote()` and `accept_quote_partial()`, send a committed offer that its the uncommitted price quote.

accept (*offer*)
The offer is accepted and cleared

Args:

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

Return: Returns a dictionary with the good's quantity and the amount paid.

accept_partial (*offer*, *quantity*)
TODO The offer is partly accepted and cleared

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

Return: Returns a dictionary with the good's quantity and the amount paid.

accept_quote (*quote*)
makes a commited buy or sell out of the counterparties quote

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

accept_quote_partial (*quote*, *quantity*)
makes a commited buy or sell out of the counterparties quote

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

buy (*receiver_group*, *receiver_idn*, *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_group: 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

get_offers (*good*, *descending=False*)
returns all open offers of the 'good' ordered by price.

Offers that are not accepted in the same subround (def block) are automatically rejected. However you can also manualy reject.

Args:

```
good: the good which should be retrieved
descending(=False): is a bool. False for descending True for
                    ascending by price
```

Example:

```
offers = get_offers('books')
for offer in offers:
    if offer['price'] < 50:
```

```
            self.accept(offer)
        elif offer['price'] < 100:
            self.accept_partial(offer, 1)
        else:
            self.reject(offer)  # optional
```

get_offers_all(*descending=False*)

returns all offers in a dictionary, with goods as key. The in each goods-category the goods are ordert by price. The order can be reverse by setting descending=True

Offers that are not accepted in the same subround (def block) are automatically rejected. However you can also manualy reject.

Args:

```
descending(optional): is a bool. False for descending True for
                      ascending by price
```

Example2:

```
oo = get_offers_all(descending=False)
for good_category in oo:
   print('The cheapest good of category' + good_category
   + ' is ' + good_category[0])
#sorted list of beer prices and seller
for offer in oo['beer']:
   print(offer['price'], offer['sender'])
```

Lists can only efficiently pop the last item. Therefore it is more efficient to order backward and buy the last good first:

```
def buy_input_good(self):
   offers = self.get_offers_all(descending=True)
   while offers:
       if offers[good][-1]['quantity'] == self.prices_for_which_buy[good]:
           self.accept(offers[good].pop())
```

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.

give(*receiver_group*, *receiver_idn*, *good*, *quantity*)

gives a good to another agent

Args:

receiver_group: Group name of the receiver

receiver_idn: id number of the receiver

good: the good to be transfered

quantity: amount to be transfered

Raises:

AssertionError, when good smaller than 0.

Return: Dictionary, with the transfere, which can be used by self.log(...).

Example:

```
self.log('taxes', self.give('money': 0.05 * self.possession('money'))
```

info(*offer_idn*)

lets you access all fields of a **given** offer. This allows you to check whether an offer was accepted, partially accepted or rejected and retrieve the quantity actually traded.

Returns a dictionary; Fields:

['status']:

'accepted': trade fully accepted

'partial': ['counter_quantity'] and self.offer_partial_status_percentage(...) for the quantities actually accepted

'rejected': trade rejected

['quantity']: the quantity of the original quote.

['counter_quantity']:

This returns the accutal quantity bought or sold. (Equal to quantity if the offer was accepted fully)

Raises:

KeyError: If the offer was answered more than one round ago.

KeyError - Dictionary: The dictionary has no status and raises a KeyError, iff the offer was not yet answered. see *Example Pending* below.

Example Pending:

```
try:
    status = info(self.offer_idn)['status']
except KeyError:
    print('offer has not yet been answered')
```

Example:

```
from pybrain.rl.learners.valuebased import ActionValueTable
from pybrain.rl.agents import LearningAgent
from pybrain.rl.learners import Q

def __init__(self):
    controller = ActionValueTable(dimState=1, numActions=1)
    learner = Q()
    rl_price = LearningAgent(controller, learner)
    self.car_cost = 500

def sales(self):
    price = reinforcement_learner.getAction():
```

```
        self.offer = self.sell('Household', 1, 'car', 1, price)

    def learn(self):
        reinforcement_learner.integrateObservation([offer_status(self.offer)])
        reinforcement_learner.giveReward([offer_status(self.offer) * price - self.car_cost])
```

partial_status_percentage(*offer_idn*)
returns the percentage of a partial accept

Args:

offer_idn: on offer as returned by self.sell(...) ord self.buy(...)

Returns: A value between [0, 1]

Raises: KeyError, when no answer has not been given or received more than one round before

quote_buy(*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)
quotes a price to buy quantity of 'good' a receiver

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

Args:

receiver_group: agent group name of the agent

receiver_idn: the agent's id number

'good': name of the good

quantity: maximum units disposed to buy at this price

price: price per unit

quote_sell(*receiver_group*, *receiver_idn*, *good*, *quantity*, *price*)
quotes a price to sell quantity of 'good' to a receiver

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

Args:

receiver_group: agent group name of the agent

receiver_idn: the agent's id number

'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_idn*)
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. However 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!)

sell(*receiver_group*, *receiver_idn*, *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_group: 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

Returns: A reference to the offer. The offer and the offer status can be accessed with *self.info(offer_reference)*.

Example:

```
def subround_1(self):
    self.offer = self.sell('household', 1, 'cookies', quantity=5, price=0.1)

def subround_2(self):
    offer = self.info(self.offer)
    if offer['status'] == 'partial':
        print(offer['counter_quantity'] , 'cookies have be bougth')
    elif:
        offer['status'] == 'accepted':
        print('Cookie monster bougth them all')
    elif:
        offer['status'] == 'rejected':
        print('On diet')
```

CHAPTER

FIVE

FIRM AND PRODUCTION

class abceagent.**Firm**

Bases: abceagent.FirmMultiTechnologies

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)
```

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}
```

produce(*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:

{'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.

Example:

```
self.set_cobb_douglas_production_function('car' ..)
car = {'tire': 4, 'metal': 2000, 'plastic':  40}
try:
    self.produce(car)
except NotEnoughGoods:
    print('today no cars')
```

produce_use_everything()

Produces output goods from all input goods.

Example:

```
self.produce_use_everything()
```

set_cobb_douglas(*output*, *multiplier*, *exponents*)

sets the firm to use 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.

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

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

set_leontief(*output*, *utilization_quantities*, *multiplier=1*, *isinteger='int'*)

sets the firm to use a Leontief production function.

A production function is a production process that produces the given input given input goods according to the formula to the output good.

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.

Example: self.car_technology = self.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)

set_production_function(*formula*, *typ='from_formula'*)

sets the firm to use a Cobb-Douglas production function from a 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 to 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 ;

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

//exponential is ** not ^

set_production_function_fast (*formula*, *output_goods*, *input_goods*, *typ='from_formula'*)
sets the firm to use a Cobb-Douglas production function from a formula, with given outputs

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 to 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)

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

//exponential is ** not ^

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

CHAPTER

SIX

HOUSEHOLD AND CONSUMPTION

class abceagent.**Household**

buy_max_utility_cobb_douglas(*offers=None*, *budget=None*, *assumed_possessions=None*)

Buys the optimal amount of goods given the possessions or alternativly a given bundle. If no budget is given all money is spend.

Args:

```
offers(optional):
    a dict with goods as keys and a list of corresponding
    offers as produced by get_offers_all. Must be ordered
    ascending by prices! If not specified get_offers_all
    is envoked.
budget(optional):
    int with maximum amount spend. If not specified all available
    money is spend.
assumed_possessions(optional):
    a dictionary of goods that are assumed to be already in the
    possession of the agent. If this is not set the current
    possessions are assumed.
```

Returns: 'no_offers': if there are no offers

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.

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
py:meth:`create_utility_function`,
py:meth:`set_cobb_douglas_utility_function` or
```

Example:

```
self.consume_everything()
```

predict_utility(*input_goods*)
Calculates the utility of a production (but does not consume)

Predicts the utility of consume_with_utility(utility_function, input_goods)

Args:

```
{'input_good1': amount1, 'input_good2': amount2 ...}: dictionary
containing the amount of input good used for the production.
```

Returns:

```
utility: Number
```

Example:

```
print(A.predict_utility(self._utility_function, {'ball': 2, 'paint': 1}))
```

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

//exponential is ** not ^

utility_function()

the utility function should be created with: set_cobb_douglas_utility_function, create_utility_function or create_utility_function_fast

CHAPTER

SEVEN

MESSAGING

class abceagent.**Messaging**

get_messages(*topic='m'*)
self.messages() returns all new messages send before this step (topic='m'). The order is randomized self.messages(topic) returns all messages with a particular non standard topice topic e.G. 'n'. The order of the messages is randomized.

Example:

```
... agent_01 ...
self.send_messages('firm_01', 'm', 'hello message')

... firm_01 - one subround later ...
potential_buyers = get_messages('m')
for buyer in potential_buyers:
   print(buyer.content)
```

Example2: see send_message

get_messages_all()
returns all messages irregardless of the topic, in a dictionary by topic

get_messages_biased(*topic='m'*)
like self.messages(topic), 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

message(*receiver_group*, *receiver_idn*, *topic*, *content*)
sends a message to agent, agent_group or 'all'. Agents receive it at the beginning of next round with self.get_messages()

Args:

```
receiver_group: agent, agent_group or 'all'
topic: string, with which this message can be received
content: string, dictionary or class, that is send.
```

Example:

```
... household_01 ...
self.send_message('firm_01', 'quote_sell', {'good':'BRD', 'quantity': 5})

... firm_01 - one subround later ...
requests = self.get_messages('quote_sell')
for req in requests:
    self.sell(req.sender, req.good, reg.quantity, self.price[req.good])
```

Example2:

```
self.send_message('firm_01', 'm', "hello my message")
```

CHAPTER

EIGHT

OBSERVING AGENTS

There are three different ways of observing your agents:

Trade Logging ABCE by default logs all trade and creates a SAM or IO matrix.

Manual in agent logging An agent is instructed to log a variable with `log()` or a change in a variable with `log_change()`.

Panel Data `panel_data()` creates panel data for all agents in a specific agent group at a specific point in every round. It is set in start.py

How to retrieve the Simulation results is explained in *Retrival of the simulation results*

8.1 Trade Logging

By default ABCE logs all trade and creates a social accounting matrix or input output matrix. Because the creation of the trade log is very time consuming you can change the default behavior in world_parameter.csv. In the column 'trade_logging' you can choose 'individual', 'group' or 'off'. (Without the apostrophs!).

8.2 Manual logging

All functions except the trade related functions can be logged. The following code logs the production function and the change of the production from last year:

```
output = self.produce(self.inputs)
self.log('production', output)
self.log_change('production', output)
```

Log loggs dictionaries. To log your own variable:

```
self.log('price', {'input': 0.8, 'output': 1})
```

Further you can write the change of a varibale between a start and an end point with: `observe_begin()` and `observe_end()`.

class abceagent.**Database**

The database class

log(*action_name*, *data_to_log*)

This command puts in a database called log, what ever values you want values need to be delivered as a dictionary:

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Example:

```
self.log('production', {, self.sector: self.final_product[self.sector]})

... different method ...

self.log('employment_and_rent', {'employment': self.possession('LAB'),
                                 'rent': self.possession('CAP'), 'composite': self.composite

for i in range(self.num_households):
    self.log('give%i' % i, self.give('Household', i, 'money', payout / self.num_households))
```

log_change(*action_name*, *data_to_log*)

This command logs the change in the variable from the round before. Important, use only once with the same action_name.

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Examples:

```
self.log_change('profit', {'money': self.possession('money')]})
self.log_change('inputs', {'money': self.possessions(['money', 'gold', 'CAP', 'LAB')]})
```

observe_begin(*action_name*, *data_to_observe*)

observe_start and observe_end, observe the change of a variable. observe_start(...), takes a list of variables to be observed. observe_end(...) writes the change in this vars into the log file

you can use nested observe_start / observe_end combinations

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Example:

```
self.log('production', {'composite': self.composite,
                        self.sector: self.final_product[self.sector]})

... different method ...

self.log('employment_and_rent', {'employment': self.possession('LAB'),
                                 'rent': self.possession('CAP')})
```

observe_end(*action_name*, *data_to_observe*)

This command puts in a database called log, what ever values you want values need to be delivered as a dictionary:

Args:

'name'(string): the name of the current action/method the agent executes

data_to_log: a dictianary with data for the database

Example:

```
        self.log('production', {'composite': self.composite,
                                self.sector: self.final_product[self.sector]})

        ... different method ...

        self.log('employment_and_rent', {'employment': self.possession('LAB'),
                                         'rent':self.possession('CAP')})
```

8.3 Panel Data

`Simulation.`**`panel_data`**(*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 declare a method that returns the variable you want to track. This function in the class of the agent must have the same name as the command.

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

Args:

agentgroup: can be either a group or 'all' for all 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 in start.py:

```
w.panel_data(group='Firm', command='after_production')

or

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

Optional in the agent:

```
class Firm(AgentEngine):

...
def after_production(self):
    track = {}
    track['t'] = 'yes'
    for key in self.prices:
        track['p_' + key] = self.prices[key]
    track.update(self.product[key])
    return track
```

CHAPTER

NINE

FIRMMULTITECHNOLOGIES

class abceagent.**FirmMultiTechnologies**

buy_optimal_production(*production_function*, *output*, *offers=None*, *budget=None*, *assumed_possessions=None*)

Buys the optimal bundle of goods to produce the given output.

Args:

```
output:
    output objective
offers(optional):
    a dict with goods as keys and a list of corresponding
    offers as produced by get_offers_all. Must be ordered
    ascending by prices! If not specified get_offers_all
    is envoked.
budget(optional):
    int with maximum amount spend. If not specified all available
    money is spend.
assumed_possessions(optional):
    a dictionary of goods that are assumed to be already in the
    possession of the agent. If this is not set the current
    possessions are assumed.
```

Returns (Not raise!):

```
'output_not_reached':
    if there was not enough money to produce the specified output.
    The maximum attainable output is was reached.

'no_offers':
    if no offers where insterted
```

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

create_leontief (*output*, *utilization_quantities*, *isinteger=''*)
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

utilization_quantities: a dictionary containing good names and correstponding exponents

isinteger='int' or isinteger='': When 'int' produce only integer amounts of the good. When '', produces floating amounts. (default)

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

Example: self.car_technology = self.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)

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 = self.create_production_function(formula) self.produce(self.production_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

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 = self.create_production_function(formula, 'golf', ['waste', 'paint']) self.produce(self.production_function, {'ball' : 1, 'paint' : 2}

//exponential is ** not ^

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)
```

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)
```

predict_produce_output (*production_function*, *input_goods*)
Predicts the output of a certain input vector and for a given production function

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.
```

Returns:

```
A dictionary of the predicted output
```

Example:

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

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 py:meth:.*create_production_function*, py:meth:.*create_cobb_douglas* or py:meth:.*create_leontief*

input goods {dictionary}: dictionary containing the amount of input good used for the production.

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

Example:

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

Example:

```
self.produce_use_everything(car_production_function)
```

sufficient_goods (*input_goods*)

checks whether the agent has all the goods in the vector input

CHAPTER

TEN

THE SIMULATION IN START.PY

The best way to start creating a simulation is by copying the start.py file and other files from 'abce/template'. Than you have to change the *sys.path.append('../abce/lib')* path so that it leads to the folder in your system.

To see how to create a simulation read *Walk through*. In this module you will find the explenation for the command.

This is a minimal template for a start.py:

```
from __future__ import division  # makes / division work correct in python !
import sys
sys.path.append('../abce/lib')  # <--- ADJUST
from agent import Agent
from abce import *


for parameters in read_parameters('simulation_parameters.csv'):
    s = Simulation(parameters)
    action_list = [
    ('all', 'one'),
    ('all', 'two'),
    ('all', 'three'),
    ]
    s.add_action_list(action_list)
    s.build_agents(Agent, 2)
    s.run()
```

class abce.**Simulation**(*simulation_parameters*)

This class in which the simulation is run. It takes the simulation_parameters to set up the simulation. Actions and agents have to be added. databases and resource declarations can be added. Than run runs the simulation.

Usually the parameters are specified in a tab separated csv file. The first line are column headers.

Args:

```
simulation_parameters: a dictionary with all parameters. "name" and
"num_rounds" are mandatory.
```

Example::

for simulation_parameters in read_parameters('simulation_parameters.csv'): action_list = [('household', 'recieve_connections'), ('household', 'offer_capital'), ('firm', 'buy_capital'), ('firm', 'production'), ('household', 'buy_product') 'after_sales_before_consumption' ('household', 'consume')] w = Simulation(simulation_parameters) w.add_action_list(action_list) w.build_agents(Firm, 'firm', 'num_firms') w.build_agents(Household, 'household', 'num_households')

w.declare_round_endowment(resource='labor_endowment', productivity=1, product='labor')
w.declare_round_endowment(resource='capital_endowment', productivity=1, product='capital')

w.panel_data('firm', command='after_sales_before_consumption')

w.run()

add_action_list (*action_list*)

add an *action_list*, which is a list of either:

- tuples of an goup_names and and action
- a single command string for panel_data or follow_agent
- [tuples of an agent name and and action, currently not unit tested!]

Example:

```
action_list = [
   repeat([
            ('Firm', 'sell'),
            ('Household', 'buy')
        ],
        repetitions=10
   ),
   ('Household_03', 'dance')
   'panel_data_end_of_round_befor consumption',
   ('Household', 'consume'),
   ]
w.add_action_list(action_list)
```

add_action_list_from_table (*parameter*)

The action list can also be declared in the simulation_parameters.csv file. Which allows you to run a batch of simulations with different orders. In simulation_parameters.csv there must be a column with which contains the a declaration of the action lists:

num_rounds	num_agents	action_list	endowment
1000,	10,	[('firm', 'sell'), ('household', 'buy')],	(5,5)
1000,	10,	[('firm', 'buf'), ('household', 'sell')],	(5,5)
1000,	10,	[('firm', 'sell'), ('household', 'calculate_net_wealth'), ('household', 'buy')],	(5,5)

The command:

```
self.add_action_list_from_table('parameters['action_list'])
```

Args:

```
parameter
    a string that contains the action_list. The string can be read
    from the simulation_parameters file: parameters['action_list'], where action_list
    is the column header in simulation_parameters.
```

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(*AgentClass*, *number=None*, *group_name=None*, *agents_parameters=None*)

This method creates agents, the first parameter is the agent class. "num_agent_class" (e.G. "num_firm") should be difined in simulation_parameters.csv. Alternatively you can also specify number = 1.s

Args:

```
AgentClass: is the name of the AgentClass that you imported
number (optional): number of agents to be created. or the colum name
of the row in simulation_parameters.csv that contains this number. If not
specified the colum name is assumed to be 'num_' + agent_name
(all lowercase). For example num_firm, if the class is called
Firm or name = Firm.
[group_name (optional): to give the group a different name than the
class_name. (do not use this if you have not a specific reason]
```

Example:

```
w.build_agents(Firm, number='num_firms')
# 'num_firms' is a column in simulation_parameters.csv
w.build_agents(Bank, 1)
w.build_agents(CentralBank, number=1)
```

build_agents_from_file(*AgentClass*, *parameters_file=None*, *multiply=1*, *delimiter='t'*, *quotechar=""*)

This command builds agents of the class AgentClass from an csv file. This way you can build agents and give every single one different parameters.

The file must be tab separated. The first line contains the column headers. The first column "agent_class" specifies the agent_class. The second column "number" (optional) allows you to create more than one agent of this type. The other columns are parameters that you can access in own_parameters the __init__ function of the agent.

Agent created from a csv-file:

```
class Agent(AgentEngine):
   def __init__(self, simulation_parameter, own_parameters, _pass_to_engine):
       AgentEngine.__init__(self, *_pass_to_engine)
       self.size = own_parameters['firm_size']
```

declare_perishable(*good*, *command='perish_at_the_round_end'*)

This good only lasts one round and than disappers. For example labor, if the labor is not used today todays labor is lost. In combination with resoure this is useful to model labor or capital.

In the example below a worker has an endowment of labor and capital. Every round he can sell his labor service and rent his capital. If he does not the labor service for this round and the rent is lost.

Args:

```
good: the good that perishes
[command: In order to perish at another point in time you can choose
a commmand and insert that command in the action list.
```

Example::

```
w.declare_round_endowment(resource='LAB_endowment', productivity=1000, product='LAB')
w.declare_round_endowment(resource='CAP_endowment', productivity=1000, product='CAP')
w.declare_perishable(good='LAB')
w.declare_perishable(good='CAP')
```

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

Every round the agent gets 'productivity' units of good 'product' for every 'resource' he possesses.

By default the this happens at the beginning 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.

Round endowments can be goup specific, that means that when somebody except this group holds them they do not produce. The default is 'all'. Restricting this to a group could have small speed gains.

panel_data(*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 declare a method that returns the variable you want to track. This function in the class of the agent must have the same name as the command.

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

Args:

agentgroup: can be either a group or 'all' for all 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 in start.py:

```
w.panel_data(group='Firm', command='after_production')

or

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

Optional in the agent:

```
class Firm(AgentEngine):

...
def after_production(self):
    track = {}
    track['t'] = 'yes'
    for key in self.prices:
        track['p_' + key] = self.prices[key]
    track.update(self.product[key])
    return track
```

run()

This runs the simulation

abce.**read_parameters**(*parameters_file='simulation_parameters.csv'*, *delimiter='\t'*, *quotechar=""*)

reads a parameter file line by line and gives a list. Where each line contains all parameters for a particular run of the simulation.

Args:

parameters_file (optional): filename of the csv file. (default:*simulation_parameters.csv*)

delimiter (optional): delimiter of the csv file. (default: tabs)

quotechar (optional): for single entries that contain the delimiter. (default: ") See python csv lib http://docs.python.org/library/csv.html

This code reads the file and runs a simulation for every line:

```
for parameter in read_parameters('simulation_parameters.csv'):
    w = Simulation(parameter)
    w.build_agents(Agent, 'agent', 'num_agents')
    w.run()
```

class abce.**repeat**(*action_list*, *repetitions*)

Repeats the contained list of actions several times

Args:

action_list: action_list that is repeated

repetitions: the number of times that an actionlist is repeated or the name of the corresponding parameter in simulation_parameters.csv

Example with number of repetitions in simulation_parameters.csv:

```
action_list = [
    repeat([
            ('firm', 'sell'),
            ('household', 'buy')
        ],
        repetitions=parameter['number_of_trade_repetitions']
    ),
    ('household_03', 'dance')
    'panel_data_end_of_round_before consumption',
    ('household', 'consume'),
    ]
s.add_action_list(action_list)
```

class abce.**repeat_while**(*action_list*, *repetitions=None*)

NOT IMPLEMENTED Repeats the contained list of actions until all agents_risponed done. Optional a maximum can be set.

Args:

```
action_list: action_list that is repeated
repetitions: the number of times that an actionlist is repeated or the name of
the corresponding parameter in simulation_parameters.csv
```

CHAPTER

ELEVEN

SIMULATION_PARAMETERS AND AGENT_PARAMETERS

In the file *simulation_parameters.csv*, parameters can be specified that either govern the simulation or are accessable to all agents. The file *agent_parameters.csv* is used to create agents from a file with :method:abce.build_agents_from_file. The agents created can access the agent_parameters for their agent group. For every simulation all agents are specified in the same file. Eventhough, they are build seperately.

We will first expose the compulsory columns in simulation_parameters.csv and agent_parameters respectively and than show how agents can access the parameters.

There are a few conventions conventions: - The files must be tab separated - First row has column headings - All lower case - *num_* indicates number of

11.1 simulation_parameters.csv

compulsory fields:

name: name of the simulation

num_rounds: number of rounds of this simulation

random_seed (optional): random seed 0 or missing chooses a random_seed at random

trade_logging: Can be set to *group* (fast) or *individual* (slow, default) or *off*)

11.2 agent_parameters.csv

This table it self does not create the agents. Rather :methode:abce.build_agents_from_file creates the agent. build_agents_from_file, searches for the line(s) with the agent_class, specified. It than creates the number of agents of this class, specified in the number column. There can be several lines with the same agent class, in this case for each line the the number of agents are create. These agents get the *agent_parameters* specified in the particular line.

compulsory fields:

agent_class:

nome of the agent's agent class.

number: number of agents for this class.

11.3 Accessing parameters in agents

Agents can only access the parameters in the __init__ method. You can store parameter seperatly or store all of them:

storing single parametern:

```
class Firm(abceagent.Agent, abceagent.Firm):
   def __init__(self, simulation_parameters, agent_parameters, _pass_to_engine):
       abceagent.Agent.__init__(self, *_pass_to_engine)

       self.sector = agents_pameters['sector']
       # saves the value or string from the column sector, for this class of agents

       self.gravity = simulation_parameters['gravity']
       # saves the simulation parameter gravity

       self.num_households = world_parameters['num_household']
       # saves the auto generated number of agents of the type household
```

As you can see above there is an **autogenerated simulation parameter 'num_agent_class'** for every agent class that is created.

You can also store all parameters as dictionary, but that is not recommended:

```
def __init__(self, simulation_parameters, agent_parameters, _pass_to_engine):
    abceagent.Agent.__init__(self, *_pass_to_engine)
    self.simulation_parameters = simulation_parameters
    self.agent_parameters = agent_parameters
```

CHAPTER

TWELVE

RETRIVAL OF THE SIMULATION RESULTS

Agents can log their internal states and the simulation can create panel data. `abceagent.Database`.

the results are stored in a subfolder of the ./results/ folder.

The tables are stored as '.csv' files which can be opened with excel and libreoffice. Further you can import the files with R, which also gives you a social accounting matix:

1. **start a in the subfolder of ./results/ that contains your simulation** results
2. start R
3. *source('../../sam.R')*
4. sam(t=0)

The same data is also as a sqlite database 'database.db' available. It can be opened by 'sqlitebrowser' in ubuntu.

CHAPTER

THIRTEEN

TOOLS AND FLOATING POINT ISSUES: ABCETOOLS MODULE

This file contains functions to compare floating point variables to 0. All variables in this simulation as in every computer programm are floating point variables. Floating point variables are not exact. Therefore var_a == var_b has no meaning. Further a variable that is var_c = 9.999999999999966e-30 is for our purpose equal to zero, but var_c == 0 would lead to False. `is_zero()`, `is_positive()` and `is_negative()` work around this problem by defining float epsilon and determine whether the variable is sufficiently close to zero or not.

This file also defines the `abcetools.NotEnoughGoods`

exception `abcetools.`**`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()
```

`abcetools.`**`agent_name`**(*group_name*, *idn*)

Given a group name and a id-number it returns the agent_name of the individual agent with the number idn. A message send to the agent_name, will be received by this individual agent

`abcetools.`**`is_negative`**(*x*)

see is positive

`abcetools.`**`is_positive`**(*x*)

checks whether a number is positive and sufficiently different from zero. All variables in ABCE are floating point numbers. Do to the workings of floating point arithmetic. If x is 1.0*e^-100 so really close to 0, x > 0 will be true, eventhough it is very very small; is_zero will be true.

`abcetools.`**`is_zero`**(*x*)

checks whether a number is sufficiently close to zero. All variables in ABCE are floating point numbers. Do to the workings of floating point arithmetic. If x is 1.0*e^-100 so really close to 0, x == 0 will be false; is_zero will be true.

CHAPTER

FOURTEEN

FILES IN THIS PACKAGE

abce/lib: the actual ABCE engine, you have to import the modules in this directory

abce/template: a template from which you can start writing your own simulation including start.py, agents and parmeter files

abce/examples: a series of example simulations

abce/unitest: Unit testing is a way to ensure that your software works as specified. The unit test simulation ensures, that all functions of the simulation and the agent's base classes work correctely.

abce/doc: The documentation source code.

CHAPTER

FIFTEEN

USING AMAZON ELASTIC CLOUD SERVER

1. Create an amazon ec2 account
2. Launch an ubuntu instance
3. Open two bash / command lines. On for your local computer one to login to the server
4. In the management console find your server address

(replace the addesses below with ubuntu@your_server_string.compute.amazonaws.com)

Log on the amazon server:

```
ssh -i ./ec2/ec2.pem ubuntu@ec2-79-125-32-99.eu-west-1.compute.amazonaws.com
```

on amazon ec2 ubuntu server:

```
mkdir cce
mkdir abce
mkdir abce/lib
```

on local computer:

```
SPATH="ubuntu@ec2-79-125-32-99.eu-west-1.compute.amazonaws.com"
scp -r -i ./ec2/ec2.pem ./abce/lib/*.py  $SPATH:~/abce/lib/
scp -r -i ./ec2/ec2.pem ./cce/*.py  $SPATH:~/cce/
scp -r -i ./ec2/ec2.pem ./cce/*.csv  $SPATH:~/cce/
```

To shut down your ubuntu instance from the command line:

```
sudo shutdown -P now
```

CHAPTER

SIXTEEN

INDICES AND TABLES

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

PYTHON MODULE INDEX

a

INDEX

N

O

P

Q

R

S

T

U