Indra Classes

There are layers of generality in these classes. I'll try to organize them.

FIRST IN GENERALITY

NODE

The Node class is the most abstract class of the Indra modules; or at least, what Gene calls "the core of the system." From it, all other classes are derived. Everything is a node, making everything relatable under a common general structure.

INITIALIZATION

(1) It is named.
(2) It's node type name is stored in ntype.
(3) Initially the object is not created as a graph in itself, though it can be changed to be one.
(4) The entire class hierarchy of objects is held in a graph, to which an individual node is added as long as it isn't part of the graph already.
(5) Looks like it adds the generic class to the class graph.

METHODS

draw(self) - shows object's structure
display(self) - object shows up on screen
debug_info(self) - returns name and ntype as str.
get_type(self) - returns ntype
to_json - returns a set: {"name", self.name}

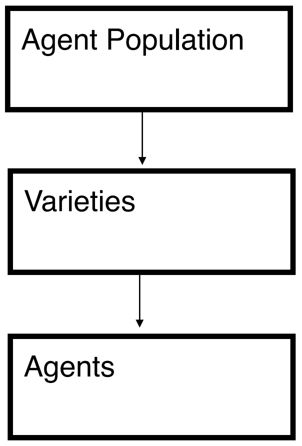
UNIVERSAL

Not implemented at the moment.

SECOND IN GENERALITY

AGENTPOP (INHERITS NODE)

"Holds the collection of agents and a sub-graph of their relationships." To implement list functions. E Right now most of the data is stored in the following way.



Varieties are the elements of the Agent Population list.

Agents are listed under a variety.

This way the relationships are clearly delineated.

INITIALIZATION

(1) Created as a node named after the agent population in question.
(2) Is has an ordered dictionary attribute which contains the varieties of agents in the population. NOT *the agents* of the population. Varieties are dictionaries themselves which have the following entries:
    - a list of agents
    - pop_data
    - my_periods
    - disp_color
(3) It has a graph attribute which is a network x graph.

METHODS

__iter__ (self) - "Creates an iterator that chains the lists [of agents] together as if one."

__len__(self) - Returns the number *of agents* in the agent population.

__reversed__(self) - Returns a reversed iterator.

varieties_iter(self) - Returns an iterator over the varieties of agents.

all_agents_list(self) - Returns a list of all agents in the population.

agent_random_iter(self) - Returns an iterator over the agents in a random order.

element_at(self, i) - Treats the agent population like a big list. Finds the agent in the i-th place.

get_var_color(self, var) - Returns the varieties display color.

add_variety(self, var) - adds a variety to the agent population. No agents are created for it.

append(self, agent, v=None) - Adds an agent to the pop. If the agent's variety is not present, we add this variety and connect the variety to the the agent_pop node. Creates an "edge" connection between the agent and it's variety.

remove(self, agent, v=None) - Removes an individual agent from the agent population. Also removes it's connection to the agent population node. [Not sure how this works in the networkx module.]

join_agents(self, a1, a2) - Adds an "edge" graph connection between agent 1 and agent 2.

contains(self, var) - If a variety is in the agent population, return true. Otherwise false.

get_agents_of_var(self, var) - Returns the list of the agents of this variety. If there are no agents, return "None."

get_randagent_of_var(self, var) - chooses a random agent of variety var get_pop(self, var) - Returns the number of agents of a variety.

get_total_pop(self) - Returns the total number of agents in the entire population.

get_pop_hist(self) - “Make a list containing the population history for each var in vars. We should merge this with display_methods assemble data when we can.”

get_pop_data(self, vars) - “Returns the value of pop_data for ‘var.’
change_pop_data(self, var, change) - “Changes the value of pop_data by ‘change.’”

change_agent_type(self, agent, old_type, new_type) - Removes the agent of the old type (cleaning up all the graphs stuff along the way.). Then appends the agent as a new type.

census(self) - “Takes a census of agents by variety. Returns a string of results for possible display.”

jsondump(self, obj) - If the obj is a graph, returns “Graph.” ….

record_results(self, file_nm) - uses the previous method …

ENTITY (Inherits Node)

INITIALIZATION

It’s a node given a name. The entity initially does not have an environment, but environment is a basic attribute.

*** env is basic to the entity class ***

METHODS

add_env(self, env) - here’s where the environment comes in. Why is this separate from the initialization? Environment-less entities?

ENVIRONMENT (INHERITS Node)

prev_period = 0 - “In case we need to restore state.” period appears to be a record of how many times agents have acted, or of how many times the whole environment has acted.

This general environment class is the space that allows agents to ______________.

INITIALIZATION

(1) Creates a Node of the given name.
(2) Networkx graph attribute created called graph.
(3) The Environment population is named after the model if possible.
(4) Agent population added with this name.
(5) Edge is added to the graph from the Environment to the Population.

(6) A list called “womb” for unborn agents.
(7) Preact??? Postact??? line_graph???
(8) Will be display the census for the user? True=Yes, False=No
(9) Something with PropArgs
(10)…
(11) Defines the user.
(12) Adds the user, menu, props, and universals to it’s graph.

METHODS

add_agent(self, agent) - appends agent to the agent population
remove agent(self, agent) - removes “ “ “
join_agents(self, a1, a2) - adds graph connection between agent 1 and agent 2
add_child(self, agent) - adds child to the womb (yet to be born into the population)
find_agent(self, name) - returns the named agent
get_agents_of_var(self, var) - Returns the list of the agents of this variety.
get_randagent_of_var(self, var) - Does the same as the previous, but randomly.

***
run(self, resume=False, loops=0) - MAIN MENU LOOP FOR ALL MODELS
(1) The program resumes its place at wherever we left off, or begins by saying the number of periods exercised on this run is 0.
(2) IF loops>0, while we have undergone less periods than we specify by “loops,” we run the program by gathering and reporting the population census.
    (1) Otherwise we run the program introduction.
    (2) … (more to document later perhaps if need be)
(3)
***

step(self) - takes census of population, increases period [What does disp=self… mean?]
- empties womb by adding agents to the environment
- Then runs program: preacts, acts, postacts for all agents (preacting and postacting are optional)

act_loop(self) - we randomly select all agents to act, then we act
preact_loop(self) - we (not randomly!?!?!) iterate through the agent pop
postact_loop(self) - same as act_loop
graph_agents(self) - “draws a graph of the agent relationships”
graph_env(self) - “draws a graph of the env’s relationships”
graph_unv(self) - “draws a graph of the universal relationship”
keep_running(self) - Returns True, presumably to tell the system to keep running.
view_pop(self) - draws a line graph if more than four periods of action
line_data(self) - returns period and population history as an ordered pair

add(self) -

MENU

The end goal is a menu that is GUI or text based depending on the users environment.

INITITIALIZATION (self, name, env, level=0)

(1) Makes a node with it's name.
(2) Records which environment we're in.
(3) The selectable menu items are recorded in an ordered dictionary.
(4) def_act = None ???
(5) The depth this menu in the submenu scheme is saved.

METHODS

add_menu_item(self, letter, item, default=False) - adds menu item, with letter representing the item
act(self) - displays itself as a node by calling the following function [EVERYTHING ACTS?]
display(self) - menu displayed, tells the user what his options are, asks the user what to do, if the choice is in the menu, we act. If no default? action is specified, we pass to a later instantiation of Menu to see what to do.

MENULEAF (Inherits Node)

INITIALIZATION (self, name, func)

(1) Node with name.
(2) Saves a function to perform on itself.

METHOD

act(self) - acts on itself using a function.

PROPARGS

"This class holds sets of named properties for program-wide values. It enables getting properties from a file, in-program, or from the user, either via the command line or a prompt."

INITIALIZATION

(1) It is a node with a given name.
(2) The model name is saved.
(3) Stores itself? in a dictionary under key after the model name.
(4) It has a graph property which is a networkx graph.
(5) Props? logfile?
(6) Makes a Logger object to record log information.
(7) …

METHODS

…

LOGGER

INITIALIZATION

…

METHODS

…

THIRD IN GENERALITY

AGENT (INHERITS ENTITY)

An agent is an entity with a goal. (I've seen this attribute only used as a string.)

INITIALIZATION

Adds a goal attribute to the entity object.

METHODS

act(self) - ABSTRACT - "What agents do."

preact(self) - ABSTRACT - "What agents do before they act." ??? Isn't this just an action???

postact(self) - ABSTRACT - "What agents do after they act." ???

survey_env(self, universal) - NOT USED - Returns None.

debug_info(self) - Returns name and ntype and goal as str

to_json(self) - Returns a "dictionary of "safe_fields" to output to a json file." LOOK UP JSON FILES.

MAINMENU (INHERITS MENU)

INITIALIZATION (self, name, env)

(1) It is a menu with a name and specific environment.
(2) e = the environment [WHY this labeling?]
(3) It is given a graph from networkx.
(4) The file menu is defined, it's one level down. It has menu items "examine log," "quit," and "write props" these act on the environment by prompt of the user. …
(5) The edit, graph, and tools submenus are similarly defined. …

USER (INHERITS ENTITY)

The user himself is a node in the scheme!

INITIALIZATION

(1) Names the user.
(2) User is given a type. Because no GUI environment is made, the utype is TERMINAL.

METHODS

tell(self, msg, type=INFO, indnt=0) - If the user is using a terminal, ipython, or “ipython_nb,” we return a message to the user with indentation. The coloring of the text is appropriate to the type of message.
ask_for_ltr(self, msg) - asks for user input after a message. Strips pre and post whitespace.
ask(self, msg) - Mistake here? It seem that the user type is checked twice (or thrice if ask_for_ltr is called) before the message is returned.