Mesa Space Module

Objects used to add a spatial component to a model.

Grid: base grid, which creates a rectangular grid. SingleGrid: extension to Grid which strictly enforces one agent per cell. MultiGrid: extension to Grid where each cell can contain a set of agents. HexGrid: extension to Grid to handle hexagonal neighbors. ContinuousSpace: a two-dimensional space where each agent has an arbitrary

position of *float*'s.

NetworkGrid: a network where each node contains zero or more agents.

accept_tuple_argument(*wrapped_function: F*) → F [source]

Decorator to allow grid methods that take a list of (x, y) coord tuples to also handle a single position, by automatically wrapping tuple in single-item list rather than forcing user to do it.

class **SingleGrid(*width: int*, *height: int*, *torus: bool*)** [source]

Rectangular grid where each cell contains exactly at most one agent.

Grid cells are indexed by [x, y], where [0, 0] is assumed to be the bottom-left and [width-1, height-1] is the top-right. If a grid is toroidal, the top and bottom, and left and right, edges wrap to each other.

Properties:

width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus.

Create a new grid.

Args:

width, height: The width and height of the grid torus: Boolean whether the grid wraps or not.

position_agent(*agent: Agent*, *x: int | str = 'random'*, *y: int | str = 'random'*) → None [source]

Position an agent on the grid. This is used when first placing agents! Setting either x or y

to “random” gives the same behavior as ‘move_to_empty()’ to get a random position. If x or y are positive, they are used. Use ‘swap_pos()’ to swap agents positions.

place_agent(*agent: Agent*, *pos: Tuple[int, int]*) → None [source]

Place the agent at the specified location, and set its pos variable.

remove_agent(*agent: Agent*) → None [source]

Remove the agent from the grid and set its pos attribute to None.

coord_iter() → Iterator[tuple[GridContent, int, int]]

An iterator that returns coordinates as well as cell contents.

static **default_val() → None**

Default value for new cell elements.

exists_empty_cells() → bool

Return True if any cells empty else False.

find_empty() → Coordinate | None

Pick a random empty cell.

get_neighborhood(*pos: Coordinate*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[Coordinate]

Return a list of cells that are in the neighborhood of a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals) If False, return Von Neumann neighborhood (exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

A list of coordinate tuples representing the neighborhood; With radius 1, at most 9 if

Moore, 5 if Von Neumann (8 and 4 if not including the center).

get_neighbors(*pos: Coordinate*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Return a list of neighbors to a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood

(exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

A list of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

is_cell_empty(*pos: Tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

iter_neighborhood(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood

(exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

iter_neighbors(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent]

Return an iterator over neighbors to a certain point.

Args:

pos: Coordinates for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood

(exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

move_agent(*agent: Agent*, *pos: Tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location

stored in a ‘pos’ tuple.

pos: Tuple of new position to move the agent to.

move_to_empty(*agent: Agent*, *cutoff: float = 0.998*, *num_agents: int | None = None*) → None

Moves agent to a random empty cell, vacating agent's old cell.

neighbor_iter(*pos: Tuple[int, int]*, *moore: bool = True*) → Iterator[Agent]

Iterate over position neighbors.

Args:

pos: (x,y) coords tuple for the position to get the neighbors of. moore: Boolean for whether to use Moore neighborhood (including

diagonals) or Von Neumann (only up/down/left/right).

out_of_bounds(*pos: Tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

swap_pos(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

torus_adj(*pos: Tuple[int, int]*) → Tuple[int, int]

Convert coordinate, handling torus looping.

---

class **MultiGrid(*width: int*, *height: int*, *torus: bool*)** [source]

Rectangular grid where each cell can contain more than one agent.

Grid cells are indexed by [x, y], where [0, 0] is assumed to be at bottom-left and [width-1, height-1] is the top-right. If a grid is toroidal, the top and bottom, and left and right, edges wrap to each other.

Properties:

width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus.

Create a new grid.

Args:

width, height: The width and height of the grid torus: Boolean whether the grid wraps or not.

static **default_val() → List[Agent]** [source]

Default value for new cell elements.

place_agent(*agent: Agent*, *pos: Tuple[int, int]*) → None [source]

Place the agent at the specified location, and set its pos variable.

remove_agent(*agent: Agent*) → None [source]

Remove the agent from the given location and set its pos attribute to None.

coord_iter() → Iterator[tuple[GridContent, int, int]]

An iterator that returns coordinates as well as cell contents.

exists_empty_cells() → bool

Return True if any cells empty else False.

find_empty() → Coordinate | None

Pick a random empty cell.

get_neighborhood(*pos: Coordinate*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[Coordinate]

Return a list of cells that are in the neighborhood of a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals) If False, return Von Neumann neighborhood (exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

A list of coordinate tuples representing the neighborhood; With radius 1, at most 9 if Moore, 5 if Von Neumann (8 and 4 if not including the center).

get_neighbors(*pos: Coordinate*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[Agent]

Return a list of neighbors to a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood

(exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

A list of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

is_cell_empty(*pos: Tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

iter_neighborhood(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood

(exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4)

if Von Neumann (if not including the center).

iter_neighbors(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent]

Return an iterator over neighbors to a certain point.

Args:

pos: Coordinates for the neighborhood to get. moore: If True, return Moore neighborhood

(including diagonals)

If False, return Von Neumann neighborhood

(exclude diagonals)

include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of non-None objects in the given neighborhood; at most 9 if Moore, 5 if Von-Neumann (8 and 4 if not including the center).

move_agent(*agent: Agent*, *pos: Tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location

stored in a 'pos' tuple.

pos: Tuple of new position to move the agent to.

move_to_empty(*agent: Agent*, *cutoff: float = 0.998*, *num_agents: int | None = None*) → None

Moves agent to a random empty cell, vacating agent's old cell.

neighbor_iter(*pos: Tuple[int, int]*, *moore: bool = True*) → Iterator[Agent]

Iterate over position neighbors.

Args:

pos: (x,y) coords tuple for the position to get the neighbors of. moore: Boolean for whether to use Moore neighborhood (including

diagonals) or Von Neumann (only up/down/left/right).

out_of_bounds(*pos: Tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

swap_pos(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

torus_adj(*pos: Tuple[int, int]*) → Tuple[int, int]

Convert coordinate, handling torus looping.

---

class **HexGrid(*width: int*, *height: int*, *torus: bool*)** [source]

Hexagonal Grid: Extends SingleGrid to handle hexagonal neighbors.

Functions according to odd-q rules. See http://www.redblobgames.com/grids/hexagons/#coordinates for more.

Properties:

width, height: The grid's width and height. torus: Boolean which determines whether to treat the grid as a torus.

Methods:

get_neighbors: Returns the objects surrounding a given cell. get_neighborhood: Returns the cells surrounding a given cell. iter_neighbors: Iterates over position neighbors. iter_neighborhood: Returns an iterator over cell coordinates that are

in the neighborhood of a certain point.

Create a new grid.

Args:

width, height: The width and height of the grid torus: Boolean whether the grid wraps or not.

get_neighborhood(*pos: Coordinate*, *include_center: bool = False*, *radius: int = 1*) → list[Coordinate] [source]

Return a list of coordinates that are in the neighborhood of a certain point. To calculate the neighborhood for a HexGrid the parity of the x coordinate of the point is important,

the neighborhood can be sketched as:

Always: (0,-), (0,+) When x is even: (-,+), (-,0), (+,+), (+,0) When x is odd: (-,0), (-,-), (+,0), (+,-)

Args:

pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

A list of coordinate tuples representing the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

neighbor_iter(*pos: Tuple[int, int]*) → Iterator[Agent] [source]

Iterate over position neighbors.

Args:

pos: (x,y) coords tuple for the position to get the neighbors of.

iter_neighborhood(*pos: Tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Tuple[int, int]] [source]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of coordinate tuples representing the neighborhood.

iter_neighbors(*pos: Tuple[int, int]*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Agent] [source]

Return an iterator over neighbors to a certain point.

Args:

pos: Coordinates for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

An iterator of non-None objects in the given neighborhood

get_neighbors(*pos: Coordinate*, *include_center: bool = False*, *radius: int = 1*) → list[Agent] [source]

Return a list of neighbors to a certain point.

Args:

pos: Coordinate tuple for the neighborhood to get. include_center: If True, return the (x, y) cell as well.

Otherwise, return surrounding cells only.

radius: radius, in cells, of neighborhood to get.

Returns:

A list of non-None objects in the given neighborhood

coord_iter() → Iterator[tuple[GridContent, int, int]]

An iterator that returns coordinates as well as cell contents.

static **default_val() → None**

Default value for new cell elements.

exists_empty_cells() → bool

Return True if any cells empty else False.

find_empty() → Coordinate | None

Pick a random empty cell.

is_cell_empty(*pos: Tuple[int, int]*) → bool

Returns a bool of the contents of a cell.

move_agent(*agent: Agent*, *pos: Tuple[int, int]*) → None

Move an agent from its current position to a new position.

Args:

agent: Agent object to move. Assumed to have its current location

stored in a ‘pos’ tuple.

pos: Tuple of new position to move the agent to.

move_to_empty(*agent: Agent*, *cutoff: float = 0.998*, *num_agents: int | None = None*) → None

Moves agent to a random empty cell, vacating agent’s old cell.

out_of_bounds(*pos: Tuple[int, int]*) → bool

Determines whether position is off the grid, returns the out of bounds coordinate.

place_agent(*agent: Agent*, *pos: Tuple[int, int]*) → None

Place the agent at the specified location, and set its pos variable.

position_agent(*agent: Agent*, *x: int | str = 'random'*, *y: int | str = 'random'*) → None

Position an agent on the grid. This is used when first placing agents! Setting either x or y to “random” gives the same behavior as ‘move_to_empty()’ to get a random position. If x or y are positive, they are used. Use ‘swap_pos()’ to swap agents positions.

remove_agent(*agent: Agent*) → None

Remove the agent from the grid and set its pos attribute to None.

swap_pos(*agent_a: Agent*, *agent_b: Agent*) → None

Swap agents positions

torus_adj(*pos: Tuple[int, int]*) → Tuple[int, int]

Convert coordinate, handling torus looping.

---

class **ContinuousSpace(*x_max: float*, *y_max: float*, *torus: bool*, *x_min: float = 0*, *y_min: float = 0*)** [source]

Continuous space where each agent can have an arbitrary position.

Assumes that all agents have a pos property storing their position as an (x, y) tuple.

This class uses a numpy array internally to store agents in order to speed up neighborhood lookups. This array is calculated on the first neighborhood lookup, and is updated if agents are added or removed.

Create a new continuous space.

> **Args:**
>
> x_max, y_max: Maximum x and y coordinates for the space. torus: Boolean for whether the edges loop around. x_min, y_min: (default 0) If provided, set the minimum x and y coordinates for the space. Below them, values loop to the other edge (if torus=True) or raise an exception.

> **place_agent(*agent:* *Agent*, *pos:* *Tuple*[*float*, *float*] | *ndarray*[*Any*, *dtype*[*float*]]) → None** [source]
>
> Place a new agent in the space.
>
> > **Args:**
> >
> > agent: Agent object to place. pos: Coordinate tuple for where to place the agent.

> **move_agent(*agent:* *Agent*, *pos:* *Tuple*[*float*, *float*] | *ndarray*[*Any*, *dtype*[*float*]]) → None** [source]
>
> Move an agent from its current position to a new position.
>
> > **Args:**
> >
> > agent: The agent object to move. pos: Coordinate tuple to move the agent to.

> **remove_agent(*agent:* *Agent*) → None** [source]
>
> Remove an agent from the space.
>
> > **Args:**
> >
> > agent: The agent object to remove

> **get_neighbors(*pos:* *FloatCoordinate*, *radius:* *float*, *include_center:* *bool* = *True*) → list[Agent]** [source]
>
> Get all agents within a certain radius.
>
> > **Args:**
> >
> > pos: (x,y) coordinate tuple to center the search at. radius: Get all the objects within this distance of the center. include_center: If True, include an object at the *exact* provided coordinates. i.e. if you are searching for the neighbors of a given agent, True will include that agent in the results.

get_heading(*pos_1: Tuple[float, float] | ndarray[Any, dtype[float]]*, *pos_2: Tuple[float, float] | ndarray[Any, dtype[float]]*) → Tuple[float, float] | ndarray[Any, dtype[float]] [source]

Get the heading vector between two points, accounting for toroidal space. It is possible to calculate the heading angle by applying the atan2 function to the result.

Args:

pos_1, pos_2: Coordinate tuples for both points.

get_distance(*pos_1: Tuple[float, float] | ndarray[Any, dtype[float]]*, *pos_2: Tuple[float, float] | ndarray[Any, dtype[float]]*) → float [source]

Get the distance between two point, accounting for toroidal space.

Args:

pos_1, pos_2: Coordinate tuples for both points.

torus_adj(*pos: Tuple[float, float] | ndarray[Any, dtype[float]]*) → Tuple[float, float] | ndarray[Any, dtype[float]] [source]

Adjust coordinates to handle torus looping.

If the coordinate is out-of-bounds and the space is toroidal, return the corresponding point within the space. If the space is not toroidal, raise an exception.

Args:

pos: Coordinate tuple to convert.

out_of_bounds(*pos: Tuple[float, float] | ndarray[Any, dtype[float]]*) → bool [source]

Check if a point is out of bounds.

---

class **NetworkGrid(*g: Any*)** [source]

Network Grid where each node contains zero or more agents.

Create a new network.

Args:

G: a NetworkX graph instance.

static **default_val() → list** [source]

Default value for a new node.

place_agent(*agent: Agent*, *node_id: int*) → None [source]

Place an agent in a node.

get_neighbors(*node_id: int*, *include_center: bool = False*, *radius: int = 1*) → list[int] [source]

Get all adjacent nodes within a certain radius

move_agent(*agent: Agent*, *node_id: int*) → None [source]

Move an agent from its current node to a new node.

remove_agent(*agent: Agent*) → None [source]

Remove the agent from the network and set its pos attribute to None.

is_cell_empty(*node_id: int*) → bool [source]

Returns a bool of the contents of a cell.

get_cell_list_contents(*cell_list: list[int]*) → list[Agent] [source]

Returns a list of the agents contained in the nodes identified in *cell_list*; nodes with empty content are excluded.

get_all_cell_contents() → list[Agent] [source]

Returns a list of all the agents in the network.

iter_cell_list_contents(*cell_list: list[int]*) → Iterator[Agent] [source]

Returns an iterator of the agents contained in the nodes identified in *cell_list*; nodes with empty content are excluded.