Table of Contents

Next topic

repast4py

This Page

Show Source

Quick search

Go

Repast4Py API Documentation

Contents:

Indices and tables

- Index
- Module Index
- Search Page

Quick search

Go

Index

Quick search

Go

Python Module Index

r

repast4py

repast4py package

Submodules

Module contents

Table of Contents

Previous topic

repast4py package

Next topic

repast4py.core module

This Page

Show Source

Quick search

Go

repast4py.context module

class `repast4py.context.`**`SharedContext`**`(`*`comm`*`)`

Bases: **`object`**

Encapsulates a population of agents on a single process rank.

A SharedContext may have one or more projections associated with it to impose a relational structure on the agents in the context. It also provides functionality for synchronizing agents across processes, moving agents from one process to another and managing any ghosting strategy.

Parameters:
comm (*mpi4py.MPI.Intracomm*) – the communicator used to communicate among SharedContexts in the distributed model

`add``(`*`agent`*`)`

Adds the specified agent to this SharedContext.

The agent will also be added to any projections currently in this SharedContext

Parameters:
agent (*Agent*) – the agent to add

`add_projection``(`*`projection`*`)`

Adds the specified projection to this SharedContext.

Any agents currently in this context will be added to the projection.

Parameters:
projection (*SharedProjection*) – the projection add

`add_value_layer``(`*`value_layer`*`)`

Adds the specified value_layer to the this context.

Parameters:

value_layer (*SharedValueLayer*) – the value layer to add.

agent(*agent_id*)

Gets the specified agent from the collection of local agents in this context.

Parameters:
agent_id – the unique id tuple of the agent to return
Returns:
The agent with the specified id or None if no such agent is found.
Return type:
Agent

Examples

```
>>> ctx = SharedContext(comm)
>>> # .. Agents Added
>>> agent1 = ctx.agent((1, 0, 1))
```

agents(*agent_type=None*, *count=None*, *shuffle=False*)

Gets the agents in this SharedContext, optionally of the specified type, count or shuffled.

Parameters:

- **agent_type** (*int*) – the type id of the agent, defaults to None.
- **count** (*Optional[int]*) – the number of agents to return, defaults to None, meaning return all the agents.
- **shuffle** (*bool*) – whether or not the iteration order is shuffled. If true, the order is shuffled. If false, the iteration order is the order of insertion.

Returns:
An iterable over all the agents in the context. If the agent_type is not None then an iterable over agents of that type will be returned.
Return type:
iterable

Examples

```
>>> PERSON_AGENT_TYPE = 1
>>> for agent in
ctx.agents(PERSON_AGENT_TYPE,
shuffle=True):
        ...
```

contains_type(*agent_type*)

Get whether or not this SharedContexts contains any agents of the specified type

Parameters:
agent_type (*int*) – the agent type id
Returns:
True if this SharedContext does contains agents of the specified type, otherwise False.
Return type:
bool

get_projection(*projection_name*)

Gets the named projection.

Parameters:
projection_name (*str*) – the name of the projection to get
Returns:
The named projection.
Raises:
KeyError – If the collection of projections in this SharedContext does not include the named projection.
Return type:
SharedProjection

ghost_agent(*agent_id*)

Gets the specified agent from the collection of ghost agents in this context.

Parameters:
agent_id – the unique id tuple of the agent to return
Returns:
The ghost agent with the specified id or None if no such agent is found.
Return type:
Agent

move_agents(*agents_to_move*,

create_agent)

Moves agents from this rank to another rank where it becomes a local agent.

The list of agents to move must be agents currently local to this rank. This performs a synchronize after moving the agents in order synchronize the new location of the agents. This is a collective operation and all ranks must call it, regardless of whether agents are being moved to or from that rank.

Parameters:

- **agents_to_move** (*List*) – A list of tuples specifying agents to move and the rank to move the agent to. Each tuple must contain the agents unique id tuple and the rank, for example `((id, type, rank), rank_to_move_to)`.
- **create_agent** (*Callable*) – a Callable that can take the result of an agent :samp: *save()* and return an agent.

remove(*agent*)

Removes the specified agent from this SharedContext

This agent is also removed from any projections associated with this SharedContext. If the agent is shared as a ghost on any other ranks it will be removed from those ranks during the next synchronization.

Parameters:
agent (*Agent*) – the agent to remove.

request_agents(*requested_agents*, *create_agent*)

Requests agents from other ranks to be copied to this rank as ghosts.

This is a collective operation and all ranks must call it, regardless of whether agents are being requested by that rank. The requested agents will be automatically added as ghosts to this rank.

Parameters:

- **requested_agents** (*List*) – A list of tuples specifying requested agents and the rank to request from. Each tuple must contain the agents unique id tuple and the rank, for example `((id, type, rank), requested_rank)`.
- **create_agent** (*Callable*) – a Callable that can take the result of an agent `save()` and return an agent.

Returns:
The list of requested agents.

Return type:
List[*Agent*]

size(*agent_type_ids=None*)

Gets the number of local agents in this SharedContext, optionally by type.

Parameters:
agent_type_ids (*Optional[List[int]]*) – a list of the agent type ids identifying the agent types to count. If this is None then the total size is returned with an agent type id of -1.

Returns:
A dictionary containing the counts (the dict values) by agent type (the dict keys).

Return type:
dict

synchronize(*restore_agent, sync_ghosts=True*)

Synchronizes the model state across processes by moving agents, filling projection buffers with ghosts, updating ghosted state and so forth.

Parameters:

- **restore_agent** (*Callable*) – a callable that takes agent state data and returns an agent instance from that data. The data is a tuple whose first element is the agent's unique id tuple, and the second element is the agent's state, as returned by that agent's type's `save()` method.
- **sync_ghosts** (*bool*) – if True, the ghosts in any SharedProjections and value layers associated with this SharedContext are also

synchronized. Defaults to True.

Table of Contents

Previous topic

repast4py.context module

Next topic

repast4py.geometry module

This Page

repast4py.core module

This module implements core functionality used by both contexts and projections.

class `repast4py.core.`**`Agent`**`(`*`id, type, rank`*`)`

Bases: **`object`**

Parent class of all agents in a repast4py simulation.

Each agent must have an id that is unique among all agents over all the ranks of a simulation. This id is composed of an integer id, an agent type id, and the integer rank on which the agent is created. These components are the arguments to the Agent constructor

Parameters:
- **id** (*int*) – an integer that uniquely identifies this agent from among those of the same type and created on the same rank. Consequently, agents created on different ranks, or of different types but created on the same rank may have the same id.
- **type** (*int*) – an integer that specifies the type of this agent.
- **rank** (*int*) – the rank on which this agent is created.

`id`

Gets the id component from this agent's unique id

Type:
int

`local_rank`

Gets and sets the current local rank of this agent. Users should **NOT** need to access this value.

Type:
int

`type`

Gets the type component from this agent's unique id

Type:
int

uid

Gets this agent’s unique id tuple (id, type, rank)

Type:
Tuple(int, int, int)

uid_rank

Gets the rank component from this agent’s unique id

Type:
int

class repast4py.core.**AgentManager**(*rank*, *world_size*)

Bases: `object`

Manages local and non-local (ghost) agents as they move between processes.

This is class is internal to the repast4py implementation and is NOT for users.

Parameters:
- **rank** (*int*) – the local process rank
- **world_size** (*int*) – the total number of ranks in the model

add_ghost(*ghosted_rank*, *agent*, *incr=1*)

Adds the specified agent to the ghost collection from the specified rank.

Parameters:
- **ghosted_rank** (*int*) – the rank the agent was received from
- **agent** (*Agent*) – the agent to add
- **incr** (*int*) – the amount to increment the reference count

add_ghosts_to_projection(*projection*)

Adds all the ghost agents to the specified projection.

Parameters:

projection – The projection to add the ghosts to

add_local(*agent*)

Adds the specified agent as a local agent

Parameters:
agent (*Agent*) – the agent to add

add_req_ghost(*agent_id*)

Adds the specified agent to the set of requested agents that are ghosts on this rank.

Parameters:
agent_id (*Tuple*) – the id of the ghost requested agent

delete_ghost(*agent_id*)

Deletes the specified ghost agent. This is used to clear the ghost when the non ghost has been removed from the simulation

Parameters:
agent_id (*Tuple*) – the unique id tuple of the ghost.

delete_ghosted(*agent_id*)

Removes the specified agent from the collection of agents ghosted from this rank and returns the ranks it is ghosted to.

This is used when the ghosted agent moves off of a rank on to another rank.

Parameters:
agent_id (*Tuple*) – the unique id tuple of the agent to remove from the ghosted collection
Returns:
A dictionary where the key is the rank the agent is ghosted to, and the value is the projection reference count for the agent on that rank.
Return type:
Dict[int, int]

delete_local(*agent_id*, *ghosted_deleted*)

Deletes the specified agent from the collection of local agents, and adds data any ghosts to be deleted.

Parameters:

- **agent_id** (*Tuple*) – the id of the agent to remove
- **ghosted_deleted** (*List*) – appended with a GhostedAgent if the agent to be removed is ghosted on another rank.

Returns:

The deleted agent or None if the agent does not exist

Return type:

Agent

`get_ghost(agent_uid, incr=1)`

Gets the agent with the specified id from the collection of ghost agents, or None if the agent does not exist in the ghost agent collection. If the agent exists, its projection reference count is incremented by the specified amount

Parameters:

- **agent_uid** (*agent uid tuple*) – the uid of the agent to get
- **incr** (*int*) – the amount to increment the reference count

Returns:

The specified agent or None if the agent is not in the ghost agent collection. If the agent exists, its projection reference count is incremented.

Return type:

Agent

`get_local(agent_id)`

Gets the specified agent from the collection of local agents.

Parameters:

agent_id (*Tuple*) – the unique id of the agent to get.

Returns:

The agent with the specified id or None if no such agent is in the local collection.

Return type:

Agent

`is_ghosted_to`(*ghost_rank*, *agent_id*)

Gets whether or not the specified agent is ghosted to the specified rank.

Parameters:

- **ghost_rank** (*int*) – the rank the agent is being sent to as a ghost
- **agent_id** (*Tuple*) – the id of the agent to get

Returns:

True if the agent is ghosted to the rank, otherwise False.

Return type:

bool

`is_requested`(*agent_id*)

Gets whether or not the specified agent is requested as a ghost on this rank.

Parameters:

agent_id (*Tuple*) – the id of the agent to check

Returns:

True if the agent is requested as a ghost on this rank, otherwise False.

Return type:

bool

`remove_ghost`(*agent*)

Decrements the ghost agents reference count and removes it from this rank, if its reference count is 0.

Parameters:

agent (*Agent*) – the agent to remove

`remove_local`(*agent_id*)

Removes the specified agent from the collection of local agents.

Parameters:

agent_id (*Tuple*) – the id of the agent to remove

Returns:

The removed agent or None if the agent does not exist

Return type:

Agent

set_as_ghosted(*ghost_ranks*, *agent_id*)

Sets the specified agent as ghosted from this rank to the specified ranks.

Parameters:

- **ghost_ranks** (*Dict*) – the ranks where the agent is a ghost on
- **agent_id** (*Tuple*) – the id of the agent that is ghosted from this rank

tag_as_ghosted(*ghost_rank*, *agent_id*)

Gets the specified agent from the local collection and marks it as ghosted on the specified rank.

Parameters:

- **ghost_rank** (*int*) – the rank the agent is being sent to as a ghost
- **agent_id** (*Tuple*) – the id of the agent to tag

Returns:

The specified agent.

Return type:

Agent

untag_as_ghosted(*ghost_rank*, *agent_id*)

Decrements the ghosted reference count for the specified agent on the specifed rank.

If the reference count goes to 0, then the agent will no longer be ghosted to that rank.

Parameters:

- **ghost_rank** (*int*) – the rank the agent is being sent to as a ghost
- **agent_id** (*Tuple*) – the id of the agent to tag

class repast4py.core.**BoundedProjection**(**args*, ***kwargs*)

Bases: `Protocol`

Protocol class for projections that are bounded such that an agent can move beyond the bounds of one instance and into another on another rank.

class repast4py.core.**GhostAgent**(*agent*, *ref_count*)

Bases: `object`

A non-local agent copied from another rank.

GhostAgent is used by the AgentManager to track and manage ghosts agents on a rank.

This is class is internal to the repast4py implementation and is NOT for users.

Parameters:

- **agent** (*Agent*) –
- **ref_count** (*int*) –

`agent`

the ghost agent

Type:
Agent

`ref_count`

a reference count tracking the number of projections on this rank that refer to the agent

Type:
int

`agent`*: Agent*

`ref_count`*: int*

class `repast4py.core.`**`GhostedAgent`**`(`*agent*`,` *ghost_ranks*`)`

Bases: `object`

An agent local to this rank that has been copied or “ghosted” to another rank.

GhostedAgent is used by the AgentManager to track and manage agents that have been ghosted from this rank to other ranks.

This is class is internal to the repast4py implementation and is NOT for users.

Parameters:

- **agent** (*Agent*) –
- **ghost_ranks** (*Dict*) –

`agent`

the ghosted agent

Type:
Agent

ghost_ranks

maps the ghosted to rank to the number references the agent has on that rank.

Type:
Dict

agent*: Agent*

ghost_ranks*: Dict*

class repast4py.core.**SharedProjection**(**args*, ***kwargs*)

Bases: **Protocol**

Protocol class that defines the API for projections that are shared across multiple processes

Previous topic

repast4py.core module

Next topic

repast4py.logging module

This Page

Show Source

Quick search

Go

repast4py.geometry module

class `repast4py.geometry.`**`BoundingBox`**(*xmin, xextent, ymin, yextent, zmin=0, zextent=0*)

Bases: **`tuple`**

A BoundingBox defines an up to 3 dimensional space in terms of the minimum value and extent along each dimension.

Create new instance of BoundingBox(xmin, xextent, ymin, yextent, zmin, zextent)

`xextent`

the size of the x dimension

Type:
int

`xmin`

the minimun x coordinate value.

Type:
int

`yextent`

the size of the y dimension

Type:
int

`ymin`

the minimun y coordinate value.

Type:
int

`zextent`

the size of the z dimension

Type:
int

`zmin`

the minimun z coordinate value.

Type:
int

repast4py.geometry.**find_1d_nghs_periodic**(*pt*, *min_max*)

Finds the neighboring 1D points of the specified point within the min_max inclusive range, using "periodic" semantics. See **`repast4py.space.BorderType`** and **`repast4py.space.GridPeriodicBorders`** for more on periodic border semantics.

Parameters:
- **pt** (*array*) – A numpy scalar array with at least one element
- **min_max** (*array*) – A numpy array of format [min_x, max_x]

Returns:
A 1d numpy array with the neighboring points according to periodic border semantics, including the source point.

Examples

Find the neighboring points of pt(4) within the bounds of 4 and 10.

```
>>> from repast4py.space import
DiscretePoint
>>> import numpy as np
>>> pt = DiscretePoint(4, 0, 0)
>>> min_max = np.array([4, 10])
>>> nghs =
find_1d_nghs_periodic(pt.coordinates,
min_max)
>>> nghs
array([10, 4, 5])
```

repast4py.geometry.**find_1d_nghs_sticky**(*pt*, *min_max*)

Finds the neighboring 1D points of the specified point within the min_max inclusive range, using "sticky" semantics. See **`repast4py.space.BorderType`** and **`repast4py.space.GridStickyBorders`** for more on sticky border semantics.

Parameters:
- **pt** (*array*) – A numpy scalar array with at least

one element

- **min_max** (*array*) – A numpy array of format [min_x, max_x]

Returns:

A 1d numpy array with the neighboring points according to sticky border semantics, including the source point.

Examples

Find the neighboring points of pt(4) within the bounds of 2 and 10.

```
>>> from repast4py.space import
DiscretePoint
>>> import numpy as np
>>> pt = DiscretePoint(4, 0, 0)
>>> min_max = np.array([2, 10])
>>> nghs =
find_1d_nghs_sticky(pt.coordinates,
min_max)
>>> nghs
array([3, 4, 5])
```

`repast4py.geometry.`**`find_2d_nghs_periodic`**`(`*`pt, min_max, pytorch_order=False`*`)`

Finds the neighboring 2D points of the specified point within the min_max inclusive range, using “periodic” semantics. See **`repast4py.space.BorderType`** and **`repast4py.space.GridPeriodicBorders`** for more on periodic border semantics.

Parameters:

- **pt** (*array*) – A numpy scalar array with at least two elements
- **min_max** (*array*) – A numpy array of format [min_x, max_x, min_y, max_y]
- **pytorch_order** – determines the order in which the neighbors are returned. If True, the y coordinates are returned first, otherwise, the x coordinates are returned first.

Returns:

A 2d numpy array with the neighboring points according to periodic border semantics, in the order determined by the pytorch_order argument, including the source point.

`repast4py.geometry.`**`find_2d_nghs_sticky`**`(`*`pt,`* *min_max*, *pytorch_order=False*)

Finds the neighboring 2D points of the specifed point within the min_max inclusive range, using "sticky" semantics. See **`repast4py.space.BorderType`** and **`repast4py.space.GridStickyBorders`** for more on sticky border semantics.

Parameters:

- **pt** (*array*) – A numpy scalar array with at least two elements
- **min_max** (*array*) – A numpy array of format [min_x, max_x, min_y, max_y]
- **pytorch_order** (*bool*) – determines the order in which the neighbors are returned. If True, the y coordinates are returned first, otherwise, the x coordinates are returned first.

Returns:

A 2d numpy array with the neighboring points according to sticky border semantics, in the order determined by the pytorch_order argument, including the source point

`repast4py.geometry.`**`find_3d_nghs_periodic`**(*pt*, *min_max*, *pytorch_order=False*)

Finds the neighboring 3D points of the specified point within the min_max inclusive range, using "periodic" semantics. See **`repast4py.space.BorderType`** and **`repast4py.space.GridPeriodicBorders`** for more on periodic border semantics.

Parameters:

- **pt** (*array*) – A numpy scalar array with at least three elements
- **min_max** (*array*) – A numpy array of format [min_x, max_x, min_y, max_y, min_z, max_z]
- **pytorch_order** – determines the order in which the neighbors are returned. If True, the y coordinates are returned first, otherwise, the x coordinates are returned first.

Returns:

A 3d numpy array with the neighboring points according to periodic border semantics, in the order determined by the pytorch_order argument, including the source point.

`repast4py.geometry.`**`find_3d_nghs_sticky`**(*pt*, *min_max*, *pytorch_order=False*)

Finds the neighboring 3D point of the specifed point within the min_max inclusive range, using “sticky” semantics. See **`repast4py.space.BorderType`** and **`repast4py.space.GridStickyBorders`** for more on sticky border semantics.

Parameters:

- **pt** (*array*) – A numpy scalar array with at least three elements
- **min_max** (*array*) – A numpy array of format [min_x, max_x, min_y, max_y, min_z, max_z]
- **pytorch_order** – determines the order in which the neighbors are returned. If True, the y coordinates are returned first, otherwise, the x coordinates are returned first.

Returns:

A 3d numpy array with the neighboring points according to sticky border semantics, in the order determined by the pytorch_order argument, including the source point.

`repast4py.geometry.`**`get_num_dims`**(*box*)

Gets the BoundingBox’s number of dimensions.

A dimension with an extent of 0 is considered not to exist.

Parameters:

box (*BoundingBox*) – the bounding box

Returns:

the number of dimensions.

Return type:

int

Examples

1D BoundingBox

```
>>> bb = BoundingBox(xmin=0,
xextent=10, ymin=0, yextent=0, zmin=0,
zextent=0)
>>> get_num_dims(bb)
1
```

2D BoundingBox

```
>>> bb = BoundingBox(xmin=0,
xextent=10, ymin=0, yextent=20, zmin=0,
zextent=0)
>>> get_num_dims(bb)
2
```

3D BoundingBox

```
>>> bb = BoundingBox(xmin=0,
xextent=10, ymin=0, yextent=10, zmin=0,
zextent=12)
>>> get_num_dims(bb)
3
```

Table of Contents

Previous topic

repast4py.geometry module

Next topic

repast4py.network module

This Page

Show Source

Quick search

Go

repast4py.logging module

The Logging module contains classes and functions for logging data produced by a repast4py simulation to a file.

class `repast4py.logging.`**`DCDataSource`**(*data_class*, *field_name*, *ds_name=None*)

Bases: **`object`**

A DCDataSource implements the **`repast4py.logging.DataSource`** protocol for Python **`dataclasses.dataclass`** objects. Each DCDataSource gets its data to log from a dataclass field.

The constructor creates a DCDataSource that will log the specified field of the specified dataclass. By default, the field name will be used as the data source name, but an optional data source name can be supplied. The data source name will become the column header in the logged tabular data.

Parameters:

- **data_class** (*dataclass*) – the dataclass containing the values to log
- **field_name** (*str*) – the name of the field in the dataclass to log
- **ds_name** (*str*) – an optional name that will be used as the column header if supplied, otherwise the field_name will be used.

property **`dtype`**

Gets the numpy dtype of this DCDataSource.

Returns:
The numpy dtype of this DCDataSource.

property **`name`***: str*

Gets the name of this DCDataSource.

Returns:
The name of this DataSource.

property **`value`***: float*

Gets the value of this DCDataSource.

Returns:
The value of this DataSource.

class repast4py.logging.**DataSource**(**args*, ***kwargs*)

Bases: **`Protocol`**

Protocol class for objects that can be used during logging as a source of data. Such objects must have a name, a type, and be able return the data to log via a "value" property.

property **dtype***: dtype*

Gets the numpy dtype of this DataSource.

Returns:
The numpy dtype of this DataSource.

property **name***: str*

Gets the name of this DataSource.

Returns:
The name of this DataSource.

property **value***: float*

Gets the value (i.e., the data to log) of this DataSource.

Returns:
The value (i.e., the data to log) of this DataSource.

class repast4py.logging.**ReducingDataLogger**(*data_source*, *op*, *rank*)

Bases: **`object`**

Creates a ReducingDataRecorder that gets its data from the specified source, reduces that data using the specified op and runs on the specified rank.

Parameters:

- **data_source** (*DataSource*) – the source of the data to reduce.
- **op** – an mpi reduction operator, e.g. MPI.SUM
- **rank** (*int*) – the rank of this ReducingDataLogger

property **dtype**

Gets the numpy dtype of this ReducingDataLogger. This is forwarded from the data source.

Returns:
The numpy dtype of this ReducingDataLogger.

log()

Logs the current value of this ReducingDataRecorder’s data source.

property **name***: str*

Gets the name of this ReducingDataLogger.

This is forwarded from the data source.

Returns:
The name of this ReducingDataLogger.

reduce(*comm*)

Reduces the values on all processes in the specified communicator to single values using the op specified in the constructor. The reduction is performed on each logged value at which point, the logged values are discarded.

Parameters:
comm (*Intracomm*) – the communicator over whose ranks the reduction is performed.
Returns:
A numpy array containing the reduced values.
Return type:
array

property **size***: int*

Gets the number of values this logger currently holds. This is set back to 0 on a reduce.

Returns:
The number of values this logger currently holds.

class repast4py.logging.**ReducingDataSet**(*data_loggers*, *comm*, *fpath*, *delimiter=','*, *buffer_size=1000*)

Bases: `object`

A ReducingDataSet represents a tabular data set where each column is produced by a ReducingDataLogger and where the name of each logger is the name of the column. The produced tabular data is periodically written to a file.

The constructor creates a ReducingDataSet whose columns are produced from the specified data loggers which are reduced across the specified communicator. The data_loggers can be created using the **`repast4py.logging.create_loggers()`** function.

Parameters:

- **data_loggers** (*List[ReducingDataLogger]*) – a list of ReducingDataLoggers that will produce the tabular data, one data_logger per column.
- **comm** (*Comm*) – the communicator to reduce over
- **fpath** (*str*) – the file to write the data to
- **delimiter** (*str*) – the delimiter to use to separate the column values
- **buffer_size** (*int*) – the number of values to log before writing to a file.

`close()`

Closes this ReducingDataSet, writing any remaining data to the file.

`log(`*`tick`***`)`**

Logs the data for the specified tick, by calling log on each data logger contained in this ReducingDataSet.

Parameters:
tick (*float*) – the tick (timestamp) at which the data is logged.

`write()`

Writes any currently logged, but not yet reduced data to this ReducingDataSet’s file, by reducing and then writing the resulting data to the file.

class `repast4py.logging.`**`TabularLogger`**`(`*`comm, fpath, headers, delimiter=','`*`)`

Bases: **`object`**

Logs arbitrary values by row in a delimited tabular format. All the rows logged by each rank are concatenated on a write into multiple rows.

Parameters:

- **comm** (*Comm*) – the communicator to reduce over
- **fpath** (*str*) – the file to write the data to
- **headers** (*List[str]*) – the header values for each column
- **delimiter** (*str*) – the seperator to use to seperate the column values

`close()`

Closes this TabularLogger, writing any rows of data to the file.

`log_row(*args)`

Logs the specified values as a row in the tabular output.

Each value in the argument list is written to a column in the order they appear in the argument list.

Parameters:
args – variable length argument list containing the values to log. The order of the values should correspond to the headers argument in the constructor.

Examples

The following will log the value of the tick, person_id variables, and 12, and 24 as a row in the tabular data.

```
>>> logger.log_row(tick, person_id,
12, 24)
```

`write()`

Writes all the currently logged rows to a file by gathering all the rows from all ranks, concatenating them, and writing the total collection of rows to the file specified in the constructor. This is a collective operation and must be called by all ranks in the communicator.

`repast4py.logging.create_loggers(data_class, op, rank, names=None)`

Creates ReducingDataLogger-s from the fields in a **`dataclasses.dataclass`**, optionally constraining the loggers to log only from specified fields. By default the names argument is None and all the dataclass fields will be logged.

Parameters:

- **data_class** (*dataclass*) – the dataclass providing the values to log
- **op** – an mpi reduction operator (e.g., MPI.SUM)
- **rank** (*int*) – the rank of this process
- **names** (*Optional[Dict[str, str]]*) – a Python dict where the keys are the names of the dataclass fields to log, and the values are the names of the column header in the output tabular data. If value is None, then the dataclass field name will be used as the data source column name.

Returns:

A list of ReducingDataLoggers that can be added to a ReducingDataSet for logging.

Return type:

List[*ReducingDataLogger*]

Examples

Creating multiple different loggers from the same data source, assigning each a different reduction operation. We append the new loggers to the original list with `+=`.

```
>>> meet_log = MeetLog()
>>> meet_log
MeetLog(total_meets=0, min_meets=0,
max_meets=0)
>>> loggers =
logging.create_loggers(meet_log,
op=MPI.SUM, names={'total_meets':
'total'}, rank=rank)
>>> loggers +=
logging.create_loggers(meet_log,
op=MPI.MIN, names={'min_meets': 'min'},
rank=rank)
>>> loggers +=
logging.create_loggers(meet_log,
op=MPI.MAX, names={'max_meets': 'max'},
rank=rank)
```

Table of Contents

Previous topic

repast4py.logging module

Next topic

repast4py.parameters module

This Page

Show Source

Quick search

Go

repast4py.network module

The network module contains classes and functions for simulating networks (i.e., graphs) where agents are the nodes in the network. The network code is built-on the networkx python package.

class `repast4py.network.`**`DirectedSharedNetwork`**`(`*`name`*`, `*`comm`*`)`

Bases: **`SharedNetwork`**

Encapsulates a directed network shared over multiple process ranks.

This wraps a networkx DiGraph object and delegates all the network related operations to it. That Graph is exposed as the *graph* attribute. See the networkx Graph documentation for more information on the network functionality that it provides. **The network structure must NOT be changed using the networkx functions and methods (adding and removing nodes, for example)**. Use this class' methods for manipulating the network structure.

`graph`

a networkx graph object wrapped by this class.

Type:
networkx.DiGraph

`comm`

the communicator over which the network is shared.

Type:
MPI.Comm

`names`

the name of this network.

Type:
str

Parameters:

- **name** (*str*) – the name of the SharedNetwork
- **comm** (*Comm*) – the communicator over which this DirectedSharedNetwork is distributed

add_edge(*u_agent*, *v_agent*, ***kwattr*)

Adds an edge beteen u_agent and v_agent.

If the u and v agents are not existing nodes in the network, they will be added. Edge attributes can be added using keyword arguments.

Parameters:

- **u_agent** (*Agent*) – The u agent
- **v_agent** (*Agent*) – The v agent
- **kwattr** – optional keyword arguments for assigning edge data.

Examples

Add an edge with a weight attribute

```
>>> g.add_edge(agent1, agent2,
weight=3.1)
```

property **is_directed***: bool*

Gets whether or not this network is directed.

Returns:
True

num_edges(*agent*)

Gets the number of edges that contain the specified agent.

Returns:
The number of edges that contain the specified agent

Parameters:
agent (*Agent*) –

Return type:
int

class repast4py.network.**SharedNetwork**(*name*, *comm*, *graph*)

Bases: `object`

A network that can be shared across multiple process ranks through ghost nodes and edges. This is the base class for the Directed and Undirected SharedNetwork classes. This class should **NOT** be instantiated directly by users.

This wraps a networkx Graph object and delegates all the network related operations to it. That Graph is exposed as the *graph* attribute. See the networkx Graph documentation for more information on the network functionality that it provides. **The network structure should _NOT_ be changed using the networkx functions and methods (adding and removing nodes, for example)**. Use this class' methods for manipulating the network structure.

`graph`

a networkx graph object that provides the network functionality.

Type:
networkx Graph

Parameters:

- **name** (*str*) – the name of the SharedNetwork
- **comm** (*Comm*) – the communicator over which this SharedNetwork is distributed
- **graph** (*networkx Graph*) – the networkx graph object that provides the network functionality.

`add(agent)`

Adds the specified agent as a node in the network.

Parameters:
agent (*Agent*) –

`add_nodes(agents)`

Adds each agent in the specified list of agents as nodes in the network.

Parameters:
agents (*List[Agent]*) – the list of agents to add

`clear_edges()`

Removes all the edges.

`contains_edge(u_agent, v_agent)`

Gets whether or not an edge exists between the u_agent and v_agent.

Parameters:

- **u_agent** (*Agent*) – the u node of the edge.
- **v_agent** (*Agent*) – the v node of the edge.

Returns:

True if an edge exists, otherwise False.

Return type:

bool

property `edge_count`*: int*

Gets the number of edges in this SharedNework.

property `node_count`*: int*

Gets the number of nodes in this SharedNetwork.

`remove(agent)`

Removes the specified agent from this SharedNetwork.

Parameters:

agent (*Agent*) – the agent to remove

Raises:

NetworkXError – if the agent is not a node in this SharedNetwork.

`remove_edge(u_agent, v_agent)`

Removes the edge between u_agent and v_agent.

Parameters:

- **u_agent** (*Agent*) – the u node of the edge.
- **v_agent** (*Agent*) – the v node of the edge.

Raises:

NetworkXError – if there is no edge between u_agent and v_agent.

`update_edge(u_agent, v_agent, **kwattr)`

Updates the edge between u_agent and v_agent with the specified attributes.

Parameters:

- **u_agent** (*Agent*) – the u node of the edge

- **v_agent** (*Agent*) – the v node of the edge
- **kwattr** – keyword arguments containting the edge attribute updates

Examples

Update the weight attribute for the edge between agent1 and agent2.

```
>>> g.update_edge(agent1, agent2,
weight=10.1)
```

class repast4py.network.**UndirectedSharedNetwork**(*name*, *comm*)

Bases: `SharedNetwork`

Encapsulates an undirected network shared over multiple processes.

This wraps a networkx Graph object and delegates all the network related operations to it. That Graph is exposed as the *graph* attribute. See the networkx Graph documentation for more information on the network functionality that it provides. **The network structure should NOT be changed using the networkx functions and methods (adding and removing nodes, for example).** Use this class' methods for manipulating the network structure.

graph

a network graph object responsible for network operations.

Type:
networkx.Graph

comm

the communicator over which the network is shared.

Type:
MPI.Comm

name

the name of this network.

Type:

str

Parameters:

- **name** (*str*) – the name of the SharedNetwork
- **comm** (*Comm*) – the communicator over which this SharedNetwork is distributed
- **directed** – specifies whether this SharedNetwork is directed or not

add_edge(*u_agent*, *v_agent*, ***kwattr*)

Adds an edge between u_agent and v_agent.

If the u and v agents are not existing nodes in the network, they will be added. Edge attributes can be added using keyword arguments.

Parameters:

- **u_agent** (*Agent*) – The u agent
- **v_agent** (*Agent*) – The v agent
- **kwattr** – optional keyword arguments for assigning edge data.

Examples

Add an edge with a weight attribute

```
>>> g.add_edge(agent1, agent2,
weight=3.1)
```

property **is_directed***: bool*

Gets whether or not this network is directed. Returns False

Returns:
False

num_edges(*agent*)

Gets the number of edges that contain the specified agent.

Parameters:
agent (*Agent*) – agent whose edge will be counted
Returns:
The number of edges that contain the specified agent
Return type:
int

`repast4py.network.`**`read_network`**`(`*`fpath, ctx, create_agent, restore_agent`*`)`

Creates and initializes the network described in the specified file.

The network will be created and added to the specified context as a projection. The nodes defined in the file will be created as agents. Those agents with a local rank will be added to the specified context, and those remote agents that participate in an edge with a local agent with be added as ghost agents.

The format of the file is as follows. The first line consists of a network id (name) followed by a space followed by 0 or 1, where 0 indicates that the network is undirected and 1 that it is directed. This first line is followed the node descriptions. Each node description line defines a node and consists at least 3 space separated elements. These 3 are the node id, the agent type, and the rank on which the agent should be created. These 3 are optionally followed by a json dictionary string containing any attributes of that agent. The node descriptions should be followed by "EDGES" to indicate that the edge descriptions follow in the remaining lines. Each edge description line defines an edge and consists of at least 2 space separated elements. These two are the node ids of the u and v components of the edge. An optional 3rd element, a json dictionary, defines any attributes (e.g., weight) to associate with that edge. For example:

```
friend_network 0
1 0 1 {"age": 23}
2 0 1 {"age" : 24}
3 0 0 {"age" : 30}
EDGES
1 2 {"weight": 0.5}
3 1 {"weight": 0.75}
```

Without node or edge attributes::

```
friend_network 0
1 0 1
2 0 1
3 0 0
EDGES
1 2
```

```
3 1
```

The node id, the type and the rank elements are passed to a user specified Callable to create the agents. In that Callable, these 3 elements must be used to create the agents' unique ids

Parameters:

- **fpath** (*str*) – the path to the network description file.
- **ctx** (*repast4py.context.SharedContext*) – the context to add the agents to
- **create_agent** (*Callable*) – a Callable used to create an agent from a node description. The Callable must have the following signature `(node_id: int, agent_type: int, rank: int, **agent_attributes)` and return an agent.
- **restore_agent** (*Callable*) – a Callable used to deserialize agents from agent data sent from another rank, that is, one that takes the tuple returned by an agent's save() method and returns an agent.

`repast4py.network.`**`write_network`**`(`*`graph, network_name, fpath, n_ranks, **partition_args`*`)`

Partitions a specified network over the specified process ranks and writes that network to a file in a format that can be read by the **`repast4py.network.read_network()`** function. The intention is that *write_network* is used to create a distributed partitioned graph file outside of a running simulation, and once created that file can then be read during simulation initialization to create a repast4py network.

The format of the file is as follows. The first line consists of a network id (name) followed by a space followed by 0 or 1, where 0 indicates that the network is undirected and 1 that it is directed. This first line is followed the node descriptions. Each node description line defines a node and consists at least 3 space separated elements. These 3 are the node id, the agent type, and the rank on which the agent should be created. These 3 are optionally followed by a json dictionary string containing any attributes of that agent. The node descriptions should be followed by “EDGES” to indicate that the edge descriptions follow in the remaining lines. Each edge description line defines an edge and consists of at least 2 space separated elements. These two are the node ids of the u and v components of the edge. An optional 3rd element, a json dictionary, defines any attributes (e.g., weight) to associate with that edge. If a node in the graph contains an ‘agent_type’ attribute then that will written as the agent type id for that node, otherwise the type is 0.

The network is partitioned across process ranks by assigning each node to a rank. A *partition_method* can be specified in the *partition_args* and can be one of ‘random’ or ‘metis’. If no *partition_method* is defined, then the method defaults to random where the nodes will be uniformly distributed among the process ranks without any load balancing. A partition method of *metis* will use the metis load balancer to allocate nodes to ranks. Metis is not included with repast4py and must be installed separately. See the networkx metis docs for more information on installation and nxmetis. Note that installing from source or github may be necessary.

Parameters:

- **graph** (*Graph*) – the graph to partition and write out.
- **network_name** (*str*) – the id / name to assign to the partitioned network
- **fpath** (*str*) – the path of the file to write the partitioned network to
- **n_ranks** (*int*) – the number of ranks to partition the network over
- **partition_args** – keyword arguments that determine how the graph will be partitioned.
  - The *partition_method* keyword is used to determine the partition method (‘random’ or ‘metis’).

  If *partition_method* is ‘random’:

    - *rng* specifies the random number generator to use in the random partitioning. Valid values are any numpy.random.Generator instance or 'default' to use the repast4py.random.default_rng. If no 'rng' is specified then by default the repast4py.random.default_rng is used.

If *partition_method* is 'metis':

- The partition_args are forwarded to the nxmetis' partition function. The various arguments to that can be found in the networkx metis reference.

Examples

Random Patitioning

```
>>> import networkx as nx
>>> g =
nx.generators.dual_barabasi_albert_graph(
 2, 1, 0.25)
>>> fname =
'./test_data/gen_net_test.txt'
>>> write_network(g, "test", fname, 3,
partition_method='random',
rng='default')
```

Metis Partitioning

```
>>> g = nx.complete_graph(60)
>>> options =
nxmetis.types.MetisOptions(seed=1)
>>> write_network(g, "metis_test",
fname, 3, partition_method='metis',
options=options)
```

repast4py.parameters module

The parameters module contains functions for working with model input parameters.

repast4py.parameters.**create_args_parser**()

Creates an argparse parser with two arguments: 1) a yaml format file containing model parameter input, and 2) an optional json dictionary string that can override that input.

This function is intended to work in concert with the **`repast4py.parameters.init_params()`** function where the results of the argparse argument parsing are passed as arguments to that function.

Examples

```
>>> parser = create_args_parser()
...
>>> args = parser.parse_args()
>>> params =
init_params(args.parameters_file,
args.parameters)
```

repast4py.parameters.**init_params**(*parameters_file*, *parameters*, *dump_file=None*)

Initializes the **`repast4py.parameters.params`** dictionary with the model input parameters.

This reads the parameters file, overrides any of those parameter values with those in the parameters string. This will automatically set the random number generator's seed if the parameters file or the parameters string contain a 'random.seed' parameter.

Parameters:
- **parameters_file** (*str*) – yaml format file containing model parameters as key value pairs.
- **parameters** (*str*) – json map format string that

overrides those in the file.
- **dump_file** (*Optional[str]*) – optional file name to dump the resolved parameters to.

Returns:
A dictionary containing the final model parameters.

Return type:
Dict

`repast4py.parameters.params = {}`

After calling **`repast4py.parameters.init_params()`**, this dictionary will contain the model parameters.

Type:
Dict

Table of Contents

Previous topic

repast4py.parameters module

Next topic

repast4py.schedule module

This Page

Show Source

Quick search

Go

repast4py.random module

Random numbers for repast4py. When this module is imported, **`repast4py.random.default_rng`** is created using the current epoch time as the random seed, and **`repast4py.random.seed`** is set to that value. The default random number generator is a numpy.random.Generator. See that API documentation for more information on the available distributions and sampling functions.

`repast4py.random.`**`default_rng`**`: Generator = Generator(PCG64) at 0xFFFF8E9475A0`

repast4py's default random generator created using init. See the Generator API documentation for more information on the available distributions and sampling functions.

Type:
numpy.random.Generator

`repast4py.random.`**`init`**`(rng_seed=None)`

Initializes the default random number generator using the specified seed.

Parameters:
rng_seed (*Optional[int]*) – the random number seed. Defaults to None in which case, the current time as returned by `time.time()` is used as the seed.

`repast4py.random.`**`seed`**`: int = 1676491545`

The current random seed used by **`repast4py.random.default_rng`**

Table of Contents

Previous topic

repast4py.random module

Next topic

repast4py.space module

repast4py.schedule module

This module includes classes and functions for scheduling events in a Repast4py simulation. Users will typically only use the **`repast4py.schedule.SharedScheduleRunner`**.

class `repast4py.schedule.`**`OneTimeEvent`**(*at, evt, priority_type=PriorityType.RANDOM, priority=nan*)

Bases: **`ScheduledEvent`**

Scheduled event that executes only once.

Parameters:
- **at** (*float*) – the time of the event.
- **evt** (*Callable*) – the callable to execute when this event is executed.
- **priority_type** (*PriorityType*) – a **`repast4py.schedule.PriorityType`** specifying the type of priority used to order events that occur at the same tick.
- **priority** (*float*) – when priority_type is PriorityType.BY_PRIORITY, the priority value is used to order all the events assigned a BY_PRIORITY priority type. Otherwise, this value is ignored. Lower values have a higher priority and will execute before those with a higher value.

`reschedule`(*queue*)

Null-op as this OneTimeEvent only occurs once.

`void`()

Voids this ScheduledEvent so that it will not execute

class `repast4py.schedule.`**`PriorityType`**(*value*)

Bases: **`IntEnum`**

Enums used to specify the priority type for scheduled events.

`BY_PRIORITY` *= 3*

FIRST *= 0*

LAST *= 1*

RANDOM *= 2*

class `repast4py.schedule.`**RepeatingEvent**(*at, interval, evt, priority_type=PriorityType.RANDOM, priority=nan*)

Bases: `ScheduledEvent`

Scheduled event that repeats at some specified interval.

Parameters:
- **at** (*float*) – the time of the event.
- **interval** (*float*) – the interval at which to repeat.
- **evt** (*Callable*) – the callable to execute when this event is executed.
- **priority_type** (*PriorityType*) – a `repast4py.schedule.PriorityType` specifying the type of priority used to order events that occur at the same tick.
- **priority** (*float*) – when priority_type is PriorityType.BY_PRIORITY, the priority value is used to order all the events assigned a BY_PRIORITY priority type. Otherwise, this value is ignored. Lower values have a higher priority and will execute before those with a higher value.

reschedule(*queue*)

Reschedules this event to occur after its interval has passed.

Parameters:
- **queue** (*List*) – the priority queue list to schedule this event on
- **sequence_count** – the original sequeuce count for this event. The sequence count is used to order events scheduled for the same tick.

void()

Voids this ScheduledEvent so that it will not execute *class* `repast4py.schedule.`**`Schedule`**

Bases: **`object`**

Encapsulates a dynamic schedule of events and a method for iterating through that schedule and exectuting those events.

Events are added to the schedule for execution at a particular *tick*, and with a particular priority. The first valid tick is 0. Events will be executed in tick order, earliest before latest. When multiple events are scheduled for the same tick, the events' priorities will be used to determine the order of execution within that tick. If an event is scheduled before the current executing tick (i.e., scheduled to occur in the past) then that event is ignored.

`execute``()`

Executes this schedule by repeatedly popping the next scheduled events off the queue and executing them.

`next_tick``()`

Gets the tick of the next scheduled event.

Returns:
the tick at which the next scheduled event will occur or -1 if nothing is scheduled.
Return type:
float

`schedule_event``(`*`at, evt, priority_type=PriorityType.RANDOM, priority=nan`*`)`

Schedules the specified event to execute at the specified tick with the specified priority. By default, events are scheduled with a random priority type.

An event's priority_type and priority determines when it will execute with respect to other events scheduled for the same tick. The priority types are:

- PriorityType.FIRST - events will execute before those with other PriorityTypes.
  All events with a FIRST priority type

will execute in the order in which they are scheduled with respect to other FIRST priority type events.

- PriorityType.RANDOM - events will execute in a random order, after the FIRST

    priority type events, and before the LAST priority type events. If there are BY_PRIORITY events scheduled for the same tick as RANDOM events, the RANDOM events will be shuffled at random into the ordered BY_PRIORITY events.

- PriorityType.BY_PRIORITY - events will execute in the order specified by the

    priority parameter (lower values are higher priority), and after any FIRST priority events and before any LAST priority events. If there are RANDOM priority events scheduled for the same tick as BY_PRIORITY events, those will be shuffled at random into the ordered BY_PRIORITY events.

- PriorityType.LAST - events will execute after those with other priority types.

    All events with a LAST priority type will execute in the order in which they are scheduled with respect to other LAST priority type events.

Parameters:

- **at** (*float*) – the time of the event.
- **evt** (*Callable*) – the Callable to execute when the event occurs.
- **priority_type** (*PriorityType*) – a `repast4py.schedule.PriorityType` specifying the type of priority used to order events that occur at the same tick.
- **priority** (*float*) – when priority_type is PriorityType.BY_PRIORITY, the priority value is used to order all the events assigned a BY_PRIORITY priority type. Otherwise, this value is ignored. Lower values have a higher priority and will execute

before those with a higher value.

Returns:
The ScheduledEvent instance that was scheduled for execution.

Return type:
ScheduledEvent

schedule_repeating_event(*at*, *interval*, *evt*, *priority_type=PriorityType.RANDOM*, *priority=nan*)

Schedules the specified event to execute at the specified tick with the specified priority, and to repeat at the specified interval. By default, events are scheduled with a random priority type.

An event's priority_type and priority determines when it will execute with respect to other events scheduled for the same tick. The priority types are:

- PriorityType.FIRST - events will execute before those with other PriorityTypes.

  All events with a FIRST priority type will execute in the order in which they are scheduled with respect to other FIRST priority type events.

- PriorityType.RANDOM - events will execute in a random order, after the FIRST

  priority type events, and before the LAST priority type events. If there are BY_PRIORITY events scheduled for the same tick as RANDOM events, the RANDOM events will be shuffled at random into the ordered BY_PRIORITY events.

- PriorityType.BY_PRIORITY - events will execute in the order specified by the

  priority parameter (lower values are higher priority), and after any FIRST priority events and before any LAST priority events. If there are RANDOM priority events scheduled for the same tick as BY_PRIORITY events, those will be shuffled at random into the ordered BY_PRIORITY events.

- PriorityType.LAST - events will execute after those with other priority types.

  All events with a LAST priority type will execute in the order in which they are scheduled with respect to other LAST priority type events.

Parameters:

- **at** (*float*) – the time of the event.
- **interval** (*float*) – the interval at which to repeat event execution.
- **evt** (*Callable*) – the Callable to execute when the event occurs.
- **priority_type** (*PriorityType*) – a **`repast4py.schedule.PriorityType`** specifying the type of priority used to order events that occur at the same tick.
- **priority** (*float*) – when priority_type is PriorityType.BY_PRIORITY, the priority value is used to order all the events assigned a BY_PRIORITY priority type. Otherwise, this value is ignored. Lower values have a higher priority and will execute before those with a higher value.

Returns:

The ScheduledEvent instance that was scheduled for execution.

Return type:

ScheduledEvent

class `repast4py.schedule.`**`ScheduleGroup`**

Bases: **`object`**

A ScheduleGroup is used internally by repast4py to order events for execution.

`add_evt`(*evt*)

Parameters:

evt (*ScheduledEvent*) –

`execute`(*queue*)

Parameters:

queue (*List*) –

execute_evts(*evts*, *queue*)

Parameters:

- **evts** (*List*) –
- **queue** (*List*) –

Return type:
bool

reset()

property **size***: int*

sort()

class repast4py.schedule.**ScheduledEvent**(*at*, *evt*, *priority_type*, *priority*)

Bases: `object`

A callable base class for all scheduled events. Calling instances of this class will execute the Callable evt.

Parameters:

- **at** (*float*) – the time of the event.
- **evt** (*Callable*) – the callable to execute when this event is executed.
- **priority_type** (*PriorityType*) – a `repast4py.schedule.PriorityType` specifying the type of priority used to order events that occur at the same tick.
- **priority** (*float*) – when priority_type is PriorityType.BY_PRIORITY, the priority value is used to order all the events assigned a BY_PRIORITY priority type. Otherwise, this value is ignored. Lower values have a higher priority and will execute before those with a higher value.

reschedule(*queue*)

Implemented by subclasses.

void()

Voids this ScheduledEvent so that it will not execute

class repast4py.schedule.**SharedScheduleRunner**(*comm*)

Bases: `object`

Encapsulates a dynamic schedule of executable events shared and synchronized across processes.

Events are added to the schedule for execution at a particular *tick*, and with a particular priority. The first valid tick is 0. Events will be executed in tick order, earliest before latest. When multiple events are scheduled for the same tick, the events' priorities will be used to determine the order of execution within that tick. If an event is scheduled before the current executing tick (i.e., scheduled to occur in the past) then that event is ignored. The scheduled is synchronized across process ranks by determining the global cross-process minimum next scheduled event time, and executing only the events schedule for that time. In this way, no schedule runs ahead of any other.

Parameters:
comm (*Intracomm*) – the communicator over which this schedule is shared.

`execute`()

Executes this SharedSchedule by repeatedly popping the next scheduled events off the queue and executing them.

`schedule_end_event`(*evt*)

Schedules the specified event (a Callable) for execution when the schedule terminates and the simulation ends.

Parameters:
evt (*Callable*) – the Callable to execute when simulation ends.
Returns:
The ScheduledEvent instance that was scheduled to execute at the end.
Return type:
ScheduledEvent

`schedule_event`(*at*, *evt*, *priority_type=PriorityType.RANDOM*, *priority=nan*)

Schedules the specified event to execute at the specified tick with the specified priority. By default, events are scheduled with a random priority type.

An event’s priority_type and priority determines when it will execute with respect to other events scheduled for the same tick. The priority types are:

- PriorityType.FIRST - events will execute before those with other PriorityTypes.

    All events with a FIRST priority type will execute in the order in which they are scheduled with respect to other FIRST priority type events.

- PriorityType.RANDOM - events will execute in a random order, after the FIRST

    priority type events, and before the LAST priority type events. If there are BY_PRIORITY events scheduled for the same tick as RANDOM events, the RANDOM events will be shuffled at random into the ordered BY_PRIORITY events.

- PriorityType.BY_PRIORITY - events will execute in the order specified by the

    priority parameter (lower values are higher priority), and after any FIRST priority events and before any LAST priority events. If there are RANDOM priority events scheduled for the same tick as BY_PRIORITY events, those will be shuffled at random into the ordered BY_PRIORITY events.

- PriorityType.LAST - events will execute after those with other priority types.

    All events with a LAST priority type will execute in the order in which they are scheduled with respect to other LAST priority type events.

Parameters:

- **at** (*float*) – the time of the event.
- **evt** (*Callable*) – the Callable to execute when the event occurs.

priority_type (*PriorityType*) – a **`repast4py.schedule.PriorityType`** specifying the type of priority used to order events that occur at the same tick.

- **priority** (*float*) – when priority_type is PriorityType.BY_PRIORITY, the priority value is used to order all the events assigned a BY_PRIORITY priority type. Otherwise, this value is ignored. Lower values have a higher priority and will execute before those with a higher value.

Returns:
The ScheduledEvent instance that was scheduled for execution.

Return type:
ScheduledEvent

`schedule_repeating_event`(*at*, *interval*, *evt*, *priority_type=PriorityType.RANDOM*, *priority=nan*)

Schedules the specified event to execute at the specified tick with the specified priority, and to repeat at the specified interval. By default, events are scheduled with a random priority type.

An event's priority_type and priority determines when it will execute with respect to other events scheduled for the same tick. The priority types are:

- PriorityType.FIRST - events will execute before those with other PriorityTypes.

  All events with a FIRST priority type will execute in the order in which they are scheduled with respect to other FIRST priority type events.

- PriorityType.RANDOM - events will execute in a random order, after the FIRST

  priority type events, and before the LAST priority type events. If there are BY_PRIORITY events scheduled for the same tick as RANDOM events, the RANDOM events will be shuffled at random into the ordered

BY_PRIORITY events.

- PriorityType.BY_PRIORITY - events will execute in the order specified by the
  priority parameter (lower values are higher priority), and after any FIRST priority events and before any LAST priority events. If there are RANDOM priority events scheduled for the same tick as BY_PRIORITY events, those will be shuffled at random into the ordered BY_PRIORITY events.
- PriorityType.LAST - events will execute after those with other priority types.
  All events with a LAST priority type will execute in the order in which they are scheduled with respect to other LAST priority type events.

Parameters:

- **at** (*float*) – the time of the event.
- **interval** (*float*) – the interval at which to repeat event execution.
- **evt** (*Callable*) – the Callable to execute when the event occurs.
- **priority_type** (*PriorityType*) – a **`repast4py.schedule.PriorityType`** specifying the type of priority used to order events that occur at the same tick.
- **priority** (*float*) – when priority_type is PriorityType.BY_PRIORITY, the priority value is used to order all the events assigned a BY_PRIORITY priority type. Otherwise, this value is ignored. Lower values have a higher priority and will execute before those with a higher value.

Returns:
The ScheduledEvent instance that was scheduled for execution.

Return type:
ScheduledEvent

`schedule_stop`(*`at`*)

Schedules the execution of this schedule to stop at the specified tick.

Parameters:
at (*float*) – the tick at which the schedule will stop.
Returns:
The ScheduledEvent instance that executes the stop event.
Return type:
ScheduledEvent

`stop()`

Stops schedule execution. All events scheduled for the current tick will execute and then schedule execution will stop.

`tick()`

Gets the current tick.

Returns:
the currently executing tick.
Return type:
float

`repast4py.schedule.create_arg_evt(evt, *args, **kwargs)`

Creates a new Callable that will call the specified Callable, passing it the specified arguments when it is executed as part of a schedule. The returned Callable can be scheduled using the SharedScheduleRunner's schedule_* methods.

Parameters:
- **evt** (*Callable*) – the Callable to execute
- **args** – the positional arguments to pass to the evt Callable
- **kwargs** – the keyword arguments to pass to the evt Callable.

Returns:
A new Callable that is compatible with SharedScheduleRunner's schedule_* methods arguments.
Return type:
Callable

`repast4py.schedule.init_schedule_runner(comm)`

Initializes the default schedule runner, a dynamic schedule of executable events shared and synchronized across processes.

Events are added to the scheduled for execution at a particular *tick*. The first valid tick is 0. Events will be executed in tick order, earliest before latest. Events scheduled for the same tick will be executed in the order in which they were added. If during the execution of a tick, an event is scheduled before the executing tick (i.e., scheduled to occur in the past) then that event is ignored. The scheduled is synchronized across process ranks by determining the global cross-process minimum next scheduled event time, and executing only the events schedule for that time. In this way, no schedule runs ahead of any other.

Parameters:
comm (*Intracomm*) – the communicator over which this scheduled is shared.
Returns:
The default SharedScheduledRunner instance that can be used to schedule events.
Return type:
SharedScheduleRunner

`repast4py.schedule.`**`runner`**`()`

Gets the default schedule runner, a dynamic schedule of executable events shared and synchronized across processes.

Returns:
The default SharedScheduledRunner instance that can be used to schedule events.
Return type:
SharedScheduleRunner

repast4py.space module

class `repast4py.space.`**`BorderType`**

Bases: **`object`**

An enum defining the border types that can be used with a space or grid.

The border type determines an agent's new location when the assigned location is beyond a grid or space's bounds. For example, during agent movement, when that movement carries the agent beyond the borders of a grid or space. Valid values are **`Sticky`**, and **`Periodic`**.

`Periodic` *= 1*

Wraps point coordinates when the point coordinate is less than or greater than the grid or spaces maximum or minimum value. For example, if the minimum grid x location is 0, the maximum is 20, and an agent moves to an x of -2, then the new x coordinate is 19.

`Sticky` *= 0*

Clips any point coordinate to the maximum or minimum value when the coordinate is less than or greater than the grid or spaces maximum or minimum value. For example, if the minimum grid x location is 0, and an agent moves to an x of -1, then the new coordinate is 0.

class `repast4py.space.`**`CartesianTopology`**

Bases: **`object`**

CartesianTopolgy(comm, global_bounds, periodic)

A CartesianTopology is used by a SharedGrid or SharedContinuousSpace to create an efficient communicator over which the grid or space is distributed and to compute the buffer synchronization metadata used to synchronize the buffers between grid and space neighbors.

This class should **not** typically be created by users, but is rather part of the internal synchronization mechanism used by the shared cartesian spaces. More info on MPI topologies can be found here.

Parameters:

- **comm** (*mpi4py.MPI.Intracomm*) – the communicator to create the cartesian topology communicator from.
- **global_bounds** (*repast4py.geometry.BoundingBox*) – the global size of the shared space or grid that will use this topology.
- **periodic** (*bool*) – whether or not the shared space or grid that will use this topology is periodic.

`comm`

Gets the mpi communicator created by this CartesianTopology

Type:
mpi4py.MPI.Intracomm

`compute_buffer_nghs`(*`buffer_size`*)

Gets an iterator over the collection of buffer synchronization meta data for the current rank for the specified buffer_size.

This data contains information for each cartesian neighbor of the current rank specifying what subsection of this grid or space should be sent what rank when synchronizing buffers. This method should not typically be called by users, but is rather part of the internal synchronization mechanism.

Parameters:
buffer_size (*int*) – the size of the buffer.
Returns:
an iterator over tuples of buffer synchronization meta data: `(rank, (xmin, xmax, ymin, ymax, zmin, zmax))` where rank is the neighbor's rank, and the nested tuple specifies what part of this space or grid to send.
Return type:
iterator

`coordinates`

Gets the cartesian coordinates of the current rank within this CartesianTopology

Type:
tuple(int)

`local_bounds`

Gets the local bounds of the current rank within this CartesianTopology

Type:
repast4py.geometry.BoundingBox

`procs_per_dim`

Gets the number of ranks per dimension in x,y,z order

Type:
tuple(int)

class `repast4py.space.`**`ContinuousPoint`**(*x*, *y*, *z=0*)

Bases: **`object`**

A 3D point with continuous (floating point) coordinates.

Parameters:

- **x** (*float*) – the x coordinate.
- **y** (*float*) – the y coordinate.
- **z** (*float, optional*) – the z coordinate. Defaults to 0.0

`_reset_from_array`(*np_array*)

Resets the coordinate values of this ContinuousPoint from the specified array's elements. The array must have a single dimension and have at least one element. The x coordinate is set from the first element, y from the second, and z from the 3rd.

This method should ONLY be used in code fully responsible for the point, that is, the point was not returned from any repast4py method or function.

Parameters:
np_array (*numpy.array*) – the array to reset

from.

`coordinates`

Gets this ContinuousPoint's coordinates as 3 element numpy array.

Type:
numpy.array

`x`

Gets this ContinuousPoint's x coordinate.

Type:
float

`y`

Gets this ContinuousPoint's y coordinate.

Type:
float

`z`

Gets this ContinuousPoint's z coordinate.

Type:
float

class `repast4py.space.`**`ContinuousSpace`**(*name*, *bounds*, *borders*, *occupancy*, *tree_threshold*)

Bases: **`object`**

An N-dimensional cartesian continuous space where agents can occupy locations defined by a continuous floating point coordinate.

The ContinuousSpace uses a tree (quad or oct depending on the number of dimensions) to optimize spatial queries. The tree can be tuned using the tree threshold parameter.

Parameters:

- **name** (*str*) – the name of the grid.
- **bounds** (*repast4py.geometry.BoundingBox*) – the dimensions of the grid.
- **borders** (*repast4py.space.BorderType*) – the border semantics - **`BorderType.Sticky`** or **`BorderType.Periodic`**
- **occupancy** (*repast4py.space.OccupancyType*) – the type of occupancy in each cell -

`OccupancyType.Multiple` or `OccupancyType.Single`
- **tree_threshold** (*int*) – the space’s tree cell maximum capacity. When this capacity is reached, the cell splits.

add(*agent*)

Adds the specified agent to this continuous space projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to add.

contains(*agent*)

Gets whether or not this continuous space projection contains the specified agent.

Parameters:
agent (*repast4py.core.Agent*) – the agent to check.
Returns:
true if this continuous space contains the specified agent, otherwise false
Return type:
bool

get_agent(*pt*)

Gets the agent at the specified location. If more than one agent exists at the specified location, the first agent to have moved to that location from among those currently at that location will be returned.

Parameters:
pt (*repast4py.space.ContinuousPoint*) – the location to get the agent at.
Returns:
the agent at that location, or None if the location is empty
Return type:
repast4py.core.Agent

get_agents(*pt*)

Gets an iterator over all the agents at the specified location.

Parameters:
pt (*repast4py.space.ContinuousPoint*) – the location to get the agents at.
Returns:
an iterator over all the agents at the specified location.
Return type:
iterator

`get_agents_within(bbox)`

Gets an iterator over all the agents within the specified bounding box.

Parameters:
box (*repast4py.geometry.BoundingBox*) – the bounding box to get the agents within.
Returns:
an iterator over all the agents within the specified bounding box.
Return type:
iterator

`get_location(agent)`

Gets the location of the specified agent

Parameters:
agent (*repast4py.core.Agent*) – the agent whose location we want to get.
Returns:
the location of the specified agent or None if the agent is not in the space.
Return type:
repast4py.space.ContinuousPoint

`move(agent, pt)`

Moves the specified agent to the specified location, returning the moved to location.

If the agent does not move beyond the continuous space’s bounds, then the returned location will be be the same as the argument location. If the agent does move out of bounds, then the location is determined by the continuous space’s border’s semantics (e.g., a location on the border if using `BorderType.Sticky` borders.

Parameters:

- **agent** (*repast4py.core.Agent*) – the agent to move.
- **pt** (*repast4py.space.ContinuousPoint*) – the location to move to.

Returns:
the location the agent has moved to or None if the agent cannot move to the specified location (e.g., if the occupancy type is **`OccupancyType.Single`** and the location is occupied.)

Return type:
repast4py.space.ContinuousPoint

`name`

Gets the name of this continuous space.

Type:
str

`remove`(*agent*)

Removes the specified agent from this continuous space projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to remove.

class repast4py.space.**DiscretePoint**(*x*, *y*, *z=0*)

Bases: **`object`**

A 3D point with discrete (integer) coordinates.

Parameters:

- **x** (*int*) – the x coordinate.
- **y** (*int*) – the y coordinate.
- **z** (*int, optional*) – the z coordinate. Defaults to 0

`_reset_from_array`(*np_array*)

Resets the coordinate values of this DiscretePoint from the specified array's elements. The array must be of the integer type, have a single dimension and have at least one element. The x coordinate is set from the first element, y from the second, and z from the 3rd.

This method should ONLY be used in code fully responsible for the point, that is, the

point was not returned from any repast4py method or function.

Parameters:
np_array (*numpy.array*) – the array to reset from.

`coordinates`

Gets this DiscretePoint's coordinates as 3 element numpy array.

Type:
numpy.array

`x`

Gets this DiscretePoint's x coordinate.

Type:
int

`y`

Gets this DiscretePoint's y coordinate.

Type:
int

`z`

Gets this DiscretePoint's z coordinate.

Type:
int

class `repast4py.space.`**`Grid`**`(`*`name, bounds, borders, occupancy`*`)`

Bases: `object`

An N-dimensional cartesian discrete space where agents can occupy locations at a discrete integer coordinate.

Parameters:

- **name** (*str*) – the name of the grid.
- **bounds** (*repast4py.geometry.BoundingBox*) – the dimensions of the grid.
- **borders** (*repast4py.space.BorderType*) – the border semantics - **`BorderType.Sticky`** or **`BorderType.Periodic`**.
- **occupancy** (*repast4py.space.OccupancyType*) –

the type of occupancy in each cell - **`OccupancyType.Multiple`** or **`OccupancyType.Single`**.

`add`(*agent*)

Adds the specified agent to this grid projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to add.

`contains`(*agent*)

Gets whether or not this grid projection contains the specified agent.

Parameters:
agent (*repast4py.core.Agent*) – the agent to check.
Returns:
true if this grid contains the specified agent, otherwise false
Return type:
bool

`get_agent`(*pt*)

Gets the agent at the specified location. If more than one agent exists at the specified location, the first agent to have moved to that location from among those currently at that location will be returned.

Parameters:
pt (*repast4py.space.DiscretePoint*) – the location to get the agent at.
Returns:
the agent at that location, or None if the location is empty
Return type:
repast4py.core.Agent

`get_agents`(*pt*)

Gets an iterator over all the agents at the specified location.

Parameters:
pt (*repast4py.space.DiscretePoint*) – the location to get the agents at.
Returns:

an iterator over all the agents at the specified location.

Return type:

iterator

`get_location(agent)`

Gets the location of the specified agent

Parameters:

agent (*repast4py.core.Agent*) – the agent whose location we want to get.

Returns:

the location of the specified agent or None if the agent is not in the space.

Return type:

repast4py.space.DiscretePoint

`move(agent, pt)`

Moves the specified agent to the specified location, returning the moved to location.

If the agent does not move beyond the grid's bounds, then the returned location will be be the same as the argument location. If the agent does move out of bounds, then the location is determined by the grid border's semantics (e.g., a location on the border if using `BorderType.Sticky` borders).

Parameters:

- **agent** (*repast4py.core.Agent*) – the agent to move.
- **pt** (*repast4py.space.DiscretePoint*) – the location to move to.

Returns:

the location the agent has moved to or None if the agent cannot move to the specified location (e.g., if the occupancy type is `OccupancyType.Single` and the location is occupied.)

Return type:

repast4py.space.DiscretePoint

`name`

Gets the name of this grid.

Type:

str

remove(*agent*)

Removes the specified agent from this grid projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to remove.

class repast4py.space.**OccupancyType**

Bases: **`object`**

An enum defining the occupancy type of a location in a space or grid. The occupancy type determines how many agents are allowed at a single location.

Valid values are: **`Multiple`**, **`Single`**.

Multiple *= 0*

Any number of agents can inhabit inhabit a location.

Single *= 1*

Only a single agent can inhabit a location.

class repast4py.space.**SharedCSpace**(*name*, *bounds*, *borders*, *occupancy*, *buffer_size*, *comm*, *tree_threshold*)

Bases: **`SharedContinuousSpace`**

An N-dimensional cartesian space where agents can occupy locations defined by a continuous floating point coordinate.

The space is shared over all the ranks in the specified communicator by sub-dividing the global bounds into some number of smaller spaces, one for each rank. For example, given a global 2D space size of 100 x 25 and 2 ranks, the global space will be split along the x dimension such that the SharedCSpace in the first MPI rank covers 0-50 x 0-25 and the second rank 50-100 x 0-25.

Each rank's SharedCSpace contains a buffer of a specified size that duplicates or "ghosts" an adjacent area of the neighboring rank's SharedCSpace. In the above example, the rank 1 space buffers the area from 50-52 x 0-25 in rank 2, and rank 2 buffers 48-50 x 0-25 in rank 1. **Be sure to specify a buffer size appropriate to any agent behavior**. For example, if an agent can “see” 3 units away and take some action based on what it perceives, then the buffer size should be at least 3, insuring that an agent can properly see beyond the borders of its own local SharedCSpace. When an agent moves beyond the borders of its current SharedCSpace, it will be transferred from its current rank, and into that containing the section of the global space that it has moved into. The SharedCSpace uses a tree (quad or oct depending on the number of dimensions) to optimize spatial queries. The tree can be tuned using the tree threshold parameter.

Parameters:

- **name** (*str*) – the name of the space.
- **bounds** (*BoundingBox*) – the global dimensions of the space.
- **borders** (*BorderType*) – the border semantics - `BorderType.Sticky` or `BorderType.Periodic`.
- **occupancy** (*OccupancyType*) – the type of occupancy in each cell - `OccupancyType.Single` or `OccupancyType.Multiple`.
- **buffer_size** (*int*) – the size of this SharedCSpace’s buffered area. This single value is used for all dimensions.
- **comm** (*Intracomm*) – the communicator containing all the ranks over which this SharedCSpace is shared.
- **tree_threshold** (*int*) – the space’s tree cell maximum capacity. When this capacity is reached, the cell splits.

add(*agent*)

Adds the specified agent to this shared continuous space projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to add.

contains(*agent*)

Gets whether or not the **local** bounds of this shared continuous space projection contains the specified agent.

Parameters:
agent (*repast4py.core.Agent*) – the agent to check.
Returns:
true if this shared continuous space contains the specified agent, otherwise false
Return type:
bool

`get_agent(pt)`

Gets the agent at the specified location. If more than one agent exists at the specified location, the first agent to have moved to that location from among those currently at that location will be returned.

Parameters:
pt (*repast4py.space.ContinuousPoint*) – the location to get the agent at.
Returns:
the agent at that location, or None if the location is empty
Return type:
repast4py.core.Agent

`get_agents(pt)`

Gets an iterator over all the agents at the specified location.

Parameters:
pt (*repast4py.space.ContinuousPoint*) – the location to get the agents at.
Returns:
an iterator over all the agents at the specified location.
Return type:
iterator

`get_agents_within(bbox)`

Gets an iterator over all the agents within the specified bounding box.

The bounding box is assumed to be within the local bounds of this SharedCSpace.

Parameters:
box (*repast4py.geometry.BoundingBox*) – the

bounding box to get the agents within.

Returns:
an iterator over all the agents within the specified bounding box.

Return type:
iterator

`get_local_bounds()`

Gets the local bounds of this shared continuous space.

The local bounds are the bounds of this shared continuous space on the current rank. For example, if the global bounds are 100 in the x dimension and 100 in the y dimension, and there are 4 ranks, then the local bounds will be some quadrant of those global bounds, 0 - 50 x 0 - 50 for example.

Returns:
the local bounds as a BoundingBox.

Return type:
repast4py.geometry.BoundingBox

`get_location(agent)`

Gets the location of the specified agent

Parameters:
agent (*repast4py.core.Agent*) – the agent whose location we want to get.

Returns:
the location of the specified agent or None if the agent is not in the space.

Return type:
repast4py.space.ContinuousPoint

`get_num_agents(pt, agent_type=None)`

Gets number of agents at the specified location, optionally fitered by agent type.

Parameters:
- **pt** (*repast4py.space.ContinuousPoint*) – the location to get the agents at.
- **agent_type** (*int*) – the type id of the agents to get the number of.

Returns:
the number of agents at the specified location.

Return type:

int

`get_random_local_pt`(*rng*)

Gets a random location within the local bounds of this SharedCSpace.

Parameters:
rng (*Generator*) – the random number generator to use to select the point.
Returns:
the random point
Return type:
ContinuousPoint

`move`(*agent, pt*)

Moves the specified agent to the specified location, returning the moved to location.

If the agent does not move beyond the shared continuous space's global bounds, then the returned location will be be the same as the argument location. If the agent does move out of the global bounds, then the location is determined by the shared continuous space's border semantics (e.g., a location on the border if using `BorderType.Sticky` borders).

Parameters:
- **agent** (*repast4py.core.Agent*) – the agent to move.
- **pt** (*repast4py.space.ContinuousPoint*) – the location to move to.

Returns:
the location the agent has moved to or None if the agent cannot move to the specified location (e.g., if the occupancy type is `OccupancyType.Single` and the location is occupied.)
Return type:
repast4py.space.ContinuousPoint

`name`

Gets the name of this shared continuous space.

Type:
str

`remove`(*agent*)

Removes the specified agent from this shared continuous space projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to remove.

`class repast4py.space.SharedGrid(name, bounds, borders, occupancy, buffer_size, comm)`

Bases: `SharedGrid`

An N-dimensional cartesian discrete space shared across ranks, where agents can occupy locations defined by a discretete integer coordinate.

The grid is shared over all the ranks in the specified communicator by sub-dividing the global bounds into some number of smaller grids, one for each rank. For example, given a global grid size of (100 x 25) and 2 ranks, the global grid will be split along the x dimension such that the SharedGrid in the first MPI rank covers (0-50 x 0-25) and the second rank (50-100 x 0-25). Each rank's SharedGrid contains buffers of a specified size that duplicate or "ghosts" an adjacent area of the neighboring rank's SharedGrid. In the above example, the rank 1 grid buffers the area from (50-52 x 0-25) in rank 2, and rank 2 buffers (48-50 x 0-25) in rank 1. Be sure to specify a buffer size appropriate to any agent behavior. For example, if an agent can "see" 3 units away and take some action based on what it perceives, then the buffer size should be at least 3, insuring that an agent can properly see beyond the borders of its own local SharedGrid. When an agent moves beyond the borders of its current SharedGrid, it will be transferred from its current rank, and into that containing the section of the global grid that it has moved into.

Parameters:

- **name** (*str*) – the name of the grid.
- **bounds** (*BoundingBox*) – the global dimensions of the grid.
- **borders** (*BorderType*) – the border semantics: BorderType.Sticky or BorderType.Periodic
- **occupancy** (*OccupancyType*) – the type of

occupancy in each cell: OccupancyType.Multiple.
- **buffer_size** (*int*) – the size of this SharedGrid buffered area. This single value is used for all dimensions.
- **comm** (*Intracomm*) – the communicator containing all the ranks over which this SharedGrid is shared.

`add(agent)`

Adds the specified agent to this shared grid projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to add.

`contains(agent)`

Gets whether or not the **local** bounds of this shared grid projection contains the specified agent.

Parameters:
agent (*repast4py.core.Agent*) – the agent to check.
Returns:
true if this shared grid contains the specified agent, otherwise false
Return type:
bool

`get_agent(pt)`

Gets the agent at the specified location. If more than one agent exists at the specified location, the first agent to have moved to that location from among those currently at that location will be returned.

Parameters:
pt (*repast4py.space.DiscretePoint*) – the location to get the agent at.
Returns:
the agent at that location, or None if the location is empty
Return type:
repast4py.core.Agent

`get_agents(pt)`

Gets an iterator over all the agents at the specified location.

Parameters:
pt (*repast4py.space.DiscretePoint*) – the location to get the agents at.
Returns:
an iterator over all the agents at the specified location.
Return type:
iterator

`get_local_bounds()`

Gets the local bounds of this shared grid.

The local bounds are the bounds of this shared grid on the current rank. For example, if the global bounds are 100 in the x dimension and 100 in the y dimension, and there are 4 ranks, then the local bounds will be some quadrant of those global bounds 0 - 50 x 0 - 50 for example.

Returns:
the local bounds as a BoundingBox.
Return type:
repast4py.geometry.BoundingBox

`get_location(agent)`

Gets the location of the specified agent

Parameters:
agent (*repast4py.core.Agent*) – the agent whose location we want to get.
Returns:
the location of the specified agent or None if the agent is not in the space.
Return type:
repast4py.space.DiscretePoint

`get_num_agents(pt, agent_type=None)`

Gets number of agents at the specified location, optionally fitered by agent type.

Parameters:

- **pt** (*repast4py.space.DiscretePoint*) – the location to get the agents at.
- **agent_type** (*int*) – the type id of the agents to get the number of.

Returns:
the number of agents at the specified location.
Return type:
int

get_random_local_pt(*rng*)

Gets a random location within the local bounds of this SharedGrid.

Parameters:
rng (*Generator*) – the random number generator to use to select the point.
Returns:
the random point
Return type:
DiscretePoint

move(*agent*, *pt*)

Moves the specified agent to the specified location, returning the moved to location.

If the agent does not move beyond the shared grid's global bounds, then the returned location will be be the same as the argument location. If the agent does move out of the global bounds, then the location is determined by the shared grid's border semantics (e.g., a location on the border if using `BorderType.Sticky` borders.

Parameters:
- **agent** (*repast4py.core.Agent*) – the agent to move.
- **pt** (*repast4py.space.DiscretePoint*) – the location to move to.

Returns:
the location the agent has moved to or None if the agent cannot move to the specified location (e.g., if the occupancy type is `OccupancyType.Single` and the location is occupied.)
Return type:
repast4py.space.DiscretePoint

name

Gets the name of this shared grid.

Type:
str

remove(*agent*)

Removes the specified agent from this shared grid projection.

Parameters:
agent (*repast4py.core.Agent*) – the agent to remove.

repast4py.util module

`repast4py.util.find_free_filename(file_path)`

Given a file path, check if that file exists, and if so, repeatedly add a numeric infix to that file path until the file does not exist.

For example, if output/counts.csv, exists check if counts_1.csv, counts_2.csv, and so on exists until finding one that doesn't exist.

Parameters:
file_path (*str*) – the path to the file to check
Returns:
the path to the unused file
Return type:
Path

`repast4py.util.is_empty(lst)`

Returns whether or not the specified list of lists is empty.

Parameters:
lst (*List[List]*) – the list of lists
Returns:
True if the list is empty or all of its nested lists are empty, otherwise False.
Return type:
bool

Previous topic

repast4py.util module

This Page

Show Source

Quick search

Go

repast4py.value_layer module

class `repast4py.value_layer.`**`ReadWriteValueLayer`**(*comm, bounds, borders, buffer_size, init_value, dtype=torch.float64*)

Bases: **`object`**

Encapsulates two **`repast4py.value_layer.SharedValueLayer`**, one of which functions as a read layer, and the other as the write layer. A ValueLayer is cross-process shared N-dimensional raster type matrix where each discrete integer coordinate contains a numeric value. A SharedValueLayer is shared over all the ranks in the specified communicator.

All write operations will be performed on the write layer, and all read operations on the read layer. The two can be swapped using the **`swap_layers()`** method. The intention is to present all agents with the equivalent value layer state, such that they all read from the read layer, but when making changes to the value layer, these changes are not available via a read until the layers are swapped. This removes any ordering effects such as a “first mover advantage.”

A SharedValueLayer is shared over all the ranks in the specified communicator by sub-dividing the global bounds into some number of smaller value layers, one for each rank. For example, given a global size of 100 x 25 and 2 ranks, the global space will be split along the x dimension such that the SharedValueLayer in the first MPI rank covers 0-50 x 0-25 and the second rank 50-100 x 0-25. Each rank’s SharedValueLayer contains a buffer of the specified size that duplicates or “ghosts” an adjacent area of the neighboring rank’s SharedValueLayer. In the above example, the rank 1 space buffers the area from 50-52 x 0-25 in rank 2, and rank 2 buffers 48-50 x 0-25 in rank 1. **Be sure to specify a buffer size**

appropriate to any agent behavior. For example, if an agent can “see” 3 units away and take some action based on what it perceives, then the buffer size should be at least 3, insuring that an agent can properly see beyond the local borders of its own SharedValueLayer.

The read and write SharedValueLayers delegate their matrix storage to a pytorch tensor, accessible via the `ValueLayer.grid` property. The read and write ValueLayers can be initialized with a specified value or with ‘random’ to initialize the matrix with a random value.

Note: 3D SharedValueLayers are not yet supported and will raise an Exception.

Parameters:

- **comm** (*mpi4py.MPI.Intracomm*) – the communicator containing all the ranks over which this SharedValueLayer is shared.
- **bounds** (*BoundingBox*) – the dimensions of the ValueLayer
- **borders** (*BorderType*) – the border semantics - `BorderType.Sticky` or `BorderType.Periodic`.
- **buffer_size** (*int*) – the size of the ValueLayers’ buffered areas. This single value is used for all dimensions.
- **init_value** – the initial value of each cell in the read and write ValueLayers: either a numeric value or `'random'` to specify a random initialization.
- **dtype** (*torch.dtype*) – the numeric type of this ValueLayer. Defaults to torch.float64.

property **bounds**

Gets the dimensions of the read and write layers.

Type:
BoundingBox

get(*pt*)

Gets the value at the specified point from the read layer.

Parameters:
pt (*DiscretePoint*) – the location to get the

value of

Returns:

The value at the specified location.

`get_nghs(pt)`

Gets the neighboring values and locations around the specified point.

Parameters:

pt – the point whose neighbors to get

Returns:

A two elelment tuple consisting of the neighboring values as a pytorch tensor, and the neighboring locations as a np.array: (values, ngh_locations). The first value in the values tensor is the value of the first location in the ngh_locations array, and so forth.

Return type:

Tuple

property `read_grid`

Gets the pytorch tensor that stores the values in the read layer. Note that the tensor is not addressed in x, y, z order so see the pytorch docs when using the tensor directly.

Type:

pytorch.tensor

`set(pt, val)`

Sets the value at the specified location on the write layer.

Parameters:

- **pt** (*DiscretePoint*) – the location to set the value of
- **val** – the value to set the location to

`swap_layers()`

Swaps the two layers. The write layer becomes the read and the read becomes the write

property `write_grid`

Gets the pytorch tensor that stores the values in the write layer. Note that the tensor is not addressed in x, y, z order so see the pytorch docs when using the tensor directly.

Type:
pytorch.tensor

class repast4py.value_layer.**SharedValueLayer**(*comm*, *bounds*, *borders*, *buffer_size*, *init_value*, *dtype=torch.float64*)

Bases: `ValueLayer`

A cross-process shared N-dimensional raster type matrix where each discrete integer coordinate contains a numeric value.

A SharedValueLayer is shared over all the ranks in the specified communicator by sub-dividing the global bounds into some number of smaller value layers, one for each rank. For example, given a global size of 100 x 25 and 2 ranks, the global space will be split along the x dimension such that the SharedValueLayer in the first MPI rank covers 0-50 x 0-25 and the second rank 50-100 x 0-25. Each rank's SharedValueLayer contains a buffer of the specified size that duplicates or "ghosts" an adjacent area of the neighboring rank's SharedValueLayer. In the above example, the rank 1 space buffers the area from 50-52 x 0-25 in rank 2, and rank 2 buffers 48-50 x 0-25 in rank 1. **Be sure to specify a buffer size appropriate to any agent behavior.** For example, if an agent can "see" 3 units away and take some action based on what it perceives, then the buffer size should be at least 3, insuring that an agent can properly see beyond the local borders of its own SharedValueLayer.

A SharedValueLayer delegates its matrix storage to a pytorch tensor, accessible via the `ValueLayer.grid` property.

Note: 3D SharedValueLayers are not yet supported and will raise an Exception.

Parameters:

- **comm** (*mpi4py.MPI.Intracomm*) – the communicator containing all the ranks over which this SharedValueLayer is shared.
- **bounds** (*BoundingBox*) – the dimensions of the ValueLayer
- **borders** (*BorderType*) – the border semantics:

BorderType.Sticky or BorderType.Periodic.
- **buffer_size** (*int*) – the size of this ValueLayers's buffered area. This single value is used for all dimensions.
- **init_value** – the initial value of each cell in this ValueLayer: either a numeric value or 'random' to specify a random initialization.
- **dtype** (*torch.dtype*) – the numeric type of this ValueLayer. Defaults to torch.float64.

class repast4py.value_layer.**ValueLayer**(*bounds*, *borders*, *init_value*, *dtype=torch.float64*)

Bases: `object`

N-dimensional raster type matrix where each discrete integer coordinate contains a numeric value.

A ValueLayer delegates its matrix storage to a pytorch tensor. The `grid` property provides access to this tensor. A ValueLayer can be initialized with a specified value or with `'random'` to initialize the matrix with a random value.

Parameters:
- **bounds** (*BoundingBox*) – the dimensions of the ValueLayer
- **borders** (*BorderType*) – the border semantics: BorderType.Sticky or BorderType.Periodic.
- **init_value** – the initial value of each cell in this ValueLayer: either a numeric value or `'random'` to specify a random initialization.
- **dtype** (*torch.dtype*) – the numeric type of this ValueLayer. Defaults to torch.float64.

property **bounds**

Gets the dimensions of this ValueLayer.

Type:
BoundingBox

get(*pt*)

Gets the value at the specified point

Parameters:
pt (*DiscretePoint*) – the location to get the value of
Returns:

The value at the specified location.

get_nghs(*pt*)

Gets the neighboring values and locations around the specified point.

Parameters:
pt (*DiscretePoint*) – the point whose neighbors to get
Returns:
A two elelment tuple consisting of the neighboring values as a pytorch tensor, and the neighboring locations as a np.array: (values, ngh_locations). The first value in the values tensor is the value of the first location in the ngh_locations array, and so forth.
Return type:
Tuple

property **grid**

Gets the pytorch tensor that stores the values in this ValueLayer. Note that the tensor is not addressed in x, y, z order so see the pytorch docs when using the tensor directly.

Type:
pytorch.tensor

set(*pt, val*)

Sets the value at the specified location

Parameters:
- **pt** (*DiscretePoint*) – the location to set the value of
- **val** – the value to set the location to

Search

Searching for multiple words only shows matches that contain all words.

search

```
.. repast4py documentation master file, created by
   sphinx-quickstart on Thu Apr 22 13:56:50 2021.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Repast4Py API Documentation
=====================================

.. toctree::
   :maxdepth: 4
   :caption: Contents:

   source/modules.rst




Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Search Go

Welcome

On this page

Site navigation

Sphinx makes it easy to create intelligent and beautiful documentation.

Here are some of Sphinx's major features:

- **Output formats:** HTML (including Windows HTML Help), LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text
- **Extensive cross-references:** semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information
- **Hierarchical structure:** easy definition of a document tree, with automatic links to siblings, parents and children
- **Automatic indices:** general index as well as a language-specific module indices
- **Code handling:** automatic highlighting using the Pygments highlighter
- **Extensions:** automatic testing of code snippets, inclusion of docstrings from Python modules (API docs) via built-in extensions, and much more functionality via third-party extensions.
- **Themes:** modify the look and feel of outputs via creating themes, and reuse many third-party themes.
- **Contributed extensions:** dozens of extensions contributed by users; most of them installable from PyPI.

Sphinx uses the reStructuredText markup language by default, and can read MyST markdown via third-party extensions. Both of these are powerful and straightforward to use, and have functionality for complex documentation and publishing workflows. They both build upon Docutils to parse and write documents.

See below for how to navigate Sphinx's documentation.

See also
The Sphinx documentation Table of Contents has a full list of this site's pages.

Get started

These sections cover the basics of getting started with Sphinx, including creating and building your own documentation from scratch.

Get started

User Guides

These sections cover various topics in using and extending Sphinx for various use-cases. They are a comprehensive guide to using Sphinx in many contexts and assume more knowledge of Sphinx. If you are new to Sphinx, we recommend starting with Get started.

User Guides

Community guide

Sphinx is community supported and welcomes contributions from anybody. The sections below should help you get started joining the Sphinx community as well as contributing.

See the Sphinx contributors' guide if you would like to contribute to the project.

Community

Reference guide

Reference documentation is more complete and programmatic in nature, it is a collection of information that can be quickly referenced. If you would like usecase-driven documentation, see Get started or User Guides.

Reference