LiSE Documentation

Release 0.9dev

Zachary Spector

Oct 30, 2018

Contents:

1 allegedb **1**
- 1.1 usage . . . 1
- 1.2 ORM . . . 2
- 1.3 cache . . . 5
- 1.4 graph . . . 8
- 1.5 query . . . 11
- 1.6 wrap . . . 13

2 LiSE **15**
- 2.1 engine . . . 15
- 2.2 character . . . 24
- 2.3 node . . . 32
- 2.4 place . . . 34
- 2.5 thing . . . 35
- 2.6 portal . . . 36
- 2.7 rule . . . 36
- 2.8 query . . . 38

3 ELiDE **41**
- 3.1 game . . . 42
- 3.2 board . . . 42
- 3.3 screen . . . 42
- 3.4 card . . . 42
- 3.5 charmenu . . . 42
- 3.6 charsview . . . 42
- 3.7 dialog . . . 42
- 3.8 dummy . . . 42
- 3.9 menu . . . 42
- 3.10 pallet . . . 42
- 3.11 rulesview . . . 42
- 3.12 spritebuilder . . . 42
- 3.13 statcfg . . . 42
- 3.14 statlist . . . 42
- 3.15 stores . . . 42
- 3.16 util . . . 42
- 3.17 app . . . 42

4 Indices and tables 43

Python Module Index 45

CHAPTER 1

allegedb

Object relational mapper for graphs with in-built revision control.

allegedb serves its own special variants on the networkx graph classes: Graph, DiGraph, MultiGraph, and MultiDiGraph. Every change to them is stored in an SQL database.

This means you can keep multiple versions of one set of graphs and switch between them without the need to save, load, or run git-checkout. Just point the ORM at the correct branch and revision, and all of the graphs in the program will change. All the different branches and revisions remain in the database to be brought back when needed.

1.1 usage

```
>>> from allegedb import ORM
>>> orm = ORM('sqlite:///test.db')
>>> orm.initdb()  # only necessary the first time you use a particular database
>>> g = orm.new_graph('test')  # also new_digraph, new_multigraph, new_multidigraph
>>> g.add_nodes_from(['spam', 'eggs', 'ham'])
>>> g.add_edge('spam', 'eggs')
>>> g.edge  # strings become unicode because that's the way sqlite3 rolls
{u'eggs': {u'ham': {}, u'spam': {}}, u'ham': {u'eggs': {}}, u'spam': {u'eggs': {}}}
>>> del g
>>> orm.close()
>>> del orm
>>> orm = ORM('sqlite:///test.db')
>>> g = orm.get_graph('test')  # returns whatever graph type you stored by that name
>>> g.edge
{u'eggs': {u'ham': {}, u'spam': {}}, u'ham': {u'eggs': {}}, u'spam': {u'eggs': {}}}
>>> import networkx as nx
>>> red = nx.random_lobster(10,0.9,0.9)
>>> blue = orm.new_graph('red', red)  # initialize with data from the given graph
>>> red.edge == blue.edge
True
>>> orm.rev = 1
```

(continues on next page)

(continued from previous page)

```
>>> blue.add_edge(17, 15)
>>> red.edge = blue.edge
False
>>> orm.rev = 0  # undoing what I did when rev-1
>>> red.edge == blue.edge
True
>>> orm.rev = 0
>>> orm.branch = 'test'    # navigating to a branch for the first time creates that␣
↪branch
>>> orm.rev = 1
>>> red.edge == blue.edge
True
>>> orm.branch = 'trunk'
>>> red.edge == blue.edge
False
```

1.2 ORM

The main interface to the allegedb ORM, and some supporting functions and classes

exception allegedb.**GraphNameError**
For errors involving graphs' names

class allegedb.**ORM**(*dbstring*, *alchemy=True*, *connect_args={}*, *validate=False*)
Instantiate this with the same string argument you'd use for a SQLAlchemy `create_engine` call. This will be your interface to allegedb.

advancing()
A context manager for when time is moving forward one turn at a time.

When used in LiSE, this means that the game is being simulated. It changes how the caching works, making it more efficient.

batch()
A context manager for when you're creating lots of state.

Reads will be much slower in a batch, but writes will be faster.

You *can* combine this with `advancing` but it isn't any faster.

btt()
Return the branch, turn, and tick.

close()
Write changes to database and close the connection

commit()
Write the state of all graphs to the database and commit the transaction.

Also saves the current branch, turn, and tick.

contextmanager()
@contextmanager decorator.

Typical usage:

@contextmanager def some_generator(<arguments>):

<setup> try:

yield <value>

finally: <cleanup>

This makes this:

with some_generator(<arguments>) as <variable>: <body>

equivalent to this:

<setup> try:

<variable> = <value> <body>

finally: <cleanup>

del_graph(*name*)
Remove all traces of a graph's existence from the database

edge_cls
alias of *allegedb.graph.Edge*

get_delta(*branch*, *turn_from*, *tick_from*, *turn_to*, *tick_to*)
Get a dictionary describing changes to all graphs.

The keys are graph names. Their values are dictionaries of the graphs' attributes' new values, with `None` for deleted keys. Also in those graph dictionaries are special keys 'node_val' and 'edge_val' describing changes to node and edge attributes, and 'nodes' and 'edges' full of booleans indicating whether a node or edge exists.

get_graph(*name*)
Return a graph previously created with `new_graph`, `new_digraph`, `new_multigraph`, or `new_multidigraph`

get_turn_delta(*branch=None*, *turn=None*, *tick_from=0*, *tick_to=None*)
Get a dictionary describing changes made on a given turn.

If `tick_to` is not supplied, report all changes after `tick_from` (default 0).

The keys are graph names. Their values are dictionaries of the graphs' attributes' new values, with `None` for deleted keys. Also in those graph dictionaries are special keys 'node_val' and 'edge_val' describing changes to node and edge attributes, and 'nodes' and 'edges' full of booleans indicating whether a node or edge exists.

initdb()
Alias of `self.query.initdb`

is_parent_of(*parent*, *child*)
Return whether `child` is a branch descended from `parent` at any remove.

nbtt()
Increment the tick and return branch, turn, tick

Unless we're viewing the past, in which case raise HistoryError.

Idea is you use this when you want to advance time, which you can only do once per branch, turn, tick.

new_digraph(*name*, *data=None*, ***attr*)
Return a new instance of type DiGraph, initialized with the given data if provided.

new_graph(*name*, *data=None*, ***attr*)
Return a new instance of type Graph, initialized with the given data if provided.

new_multidigraph(*name*, *data=None*, ***attr*)
Return a new instance of type MultiDiGraph, initialized with the given data if provided.

new_multigraph(*name*, *data=None*, ***attr*)
Return a new instance of type MultiGraph, initialized with the given data if provided.

node_cls
alias of *allegedb.graph.Node*

plan()
A context manager for 'hypothetical' edits.

Start a block of code like:

‘‘‘ with orm.plan():

    ...

‘‘‘

and any changes you make to the world state within that block will be 'plans,' meaning that they are used as defaults. The world will obey your plan unless you make changes to the same entities outside of the plan, in which case the world will obey those, and cancel any future plan.

New branches cannot be started within plans.

query_engine_cls
alias of *allegedb.query.QueryEngine*

class allegedb.**PlanningContext**(*orm*)
A context manager for 'hypothetical' edits.

Start a block of code like:

‘‘‘ with orm.plan():

    ...

‘‘‘

and any changes you make to the world state within that block will be 'plans,' meaning that they are used as defaults. The world will obey your plan unless you make changes to the same entities outside of the plan, in which case the world will obey those, and cancel any future plan.

New branches cannot be started within plans.

class allegedb.**TimeSignal**(*engine*)
Acts like a list of `[branch, turn]` for the most part.

You can set these to new values, or even replace them with a whole new `[branch, turn]` if you wish. It's even possible to use the strings `'branch'` or `'turn'` in the place of indices, but at that point you might prefer to set `engine.branch` or `engine.turn` directly.

This is a Signal, so pass a function to the connect(...) method and it will be called whenever the time changes. Not when the tick changes, though. If you really need something done whenever the tick changes, override the _set_tick method of *allegedb.ORM*.

class allegedb.**TimeSignalDescriptor**
Acts like a list of `[branch, turn]` for the most part.

You can set these to new values, or even replace them with a whole new `[branch, turn]` if you wish. It's even possible to use the strings `'branch'` or `'turn'` in the place of indices, but at that point you might prefer to set `engine.branch` or `engine.turn` directly.

This is a Signal, so pass a function to the connect(...) method and it will be called whenever the time changes. Not when the tick changes, though. If you really need something done whenever the tick changes, override the _set_tick method of *allegedb.ORM*.

allegedb.**setedge**(*delta*, *is_multigraph*, *graph*, *orig*, *dest*, *idx*, *exists*)
Change a delta to say that an edge was created or deleted

allegedb.**setedgeval**(*delta*, *is_multigraph*, *graph*, *orig*, *dest*, *idx*, *key*, *value*)
Change a delta to say that an edge stat was set to a certain value

allegedb.**setgraphval**(*delta*, *graph*, *key*, *val*)
Change a delta to say that a graph stat was set to a certain value

allegedb.**setnode**(*delta*, *graph*, *node*, *exists*)
Change a delta to say that a node was created or deleted

allegedb.**setnodeval**(*delta*, *graph*, *node*, *key*, *value*)
Change a delta to say that a node stat was set to a certain value

1.3 cache

Classes for in-memory storage and retrieval of historical graph data.

class allegedb.cache.**Cache**(*db*)
A data store that's useful for tracking graph revisions.

branches = None
A less structured alternative to `keys`.

For when you already know the entity and the key within it, but still need to iterate through history to find the value.

contains_entity(**args*)
Check if an entity has a key at the given time, if entity specified.

Otherwise check if the entity exists.

contains_entity_key(**args*)
Check if an entity has a key at the given time, if entity specified.

Otherwise check if the entity exists.

contains_entity_or_key(**args*)
Check if an entity has a key at the given time, if entity specified.

Otherwise check if the entity exists.

contains_key(**args*)
Check if an entity has a key at the given time, if entity specified.

Otherwise check if the entity exists.

count_entities(**args*, *forward=None*)
Return the number of keys an entity has, if you specify an entity.

Otherwise return the number of entities.

count_entities_or_keys(**args*, *forward=None*)
Return the number of keys an entity has, if you specify an entity.

Otherwise return the number of entities.

count_entity_keys(**args*, *forward=None*)
Return the number of keys an entity has, if you specify an entity.

Otherwise return the number of entities.

count_keys(**args*, *forward=None*)
Return the number of keys an entity has, if you specify an entity.

Otherwise return the number of entities.

iter_entities(**args*, *forward=None*)
Iterate over the keys an entity has, if you specify an entity.

Otherwise iterate over the entities themselves, or at any rate the tuple specifying which entity.

iter_entities_or_keys(**args*, *forward=None*)
Iterate over the keys an entity has, if you specify an entity.

Otherwise iterate over the entities themselves, or at any rate the tuple specifying which entity.

iter_entity_keys(**args*, *forward=None*)
Iterate over the keys an entity has, if you specify an entity.

Otherwise iterate over the entities themselves, or at any rate the tuple specifying which entity.

iter_keys(**args*, *forward=None*)
Iterate over the keys an entity has, if you specify an entity.

Otherwise iterate over the entities themselves, or at any rate the tuple specifying which entity.

keycache = None
Keys an entity has at a given turn and tick.

keys = None
Cache of entity data keyed by the entities themselves.

That means the whole tuple identifying the entity is the top-level key in this cache here. The second-to-top level is the key within the entity.

Deeper layers of this cache are keyed by branch, turn, and tick.

load(*data*, *validate=False*, *cb=None*)
Add a bunch of data. It doesn't need to be in chronological order.

With `validate=True`, raise ValueError if this results in an incoherent cache.

If a callable `cb` is provided, it will be called with each row. It will also be passed my `validate` argument.

parents = None
Entity data keyed by the entities' parents.

An entity's parent is what it's contained in. When speaking of a node, this is its graph. When speaking of an edge, the parent is usually the graph and the origin in a pair, though for multigraphs the destination might be part of the parent as well.

Deeper layers of this cache are keyed by branch and revision.

presettings = None
The values prior to `entity[key] = value` operations performed on some turn

retrieve(**args*)
Get a value previously .store(...)'d.

Needs at least five arguments. The -1th is the tick within the turn you want, the -2th is that turn, the -3th is the branch, and the -4th is the key. All other arguments identify the entity that the key is in.

settings = None
All the `entity[key] = value` operations that were performed on some turn

shallowest = None
A dictionary for plain, unstructured hinting.

store(**args*, *planning=None*, *forward=None*)
Put a value in various dictionaries for later .retrieve(...).

Needs at least five arguments, of which the -1th is the value to store, the -2th is the tick to store it at, the -3th is the turn to store it in, the -4th is the branch the revision is in, the -5th is the key the value is for, and the remaining arguments identify the entity that has the key, eg. a graph, node, or edge.

With `planning=True`, you will be permitted to alter "history" that takes place after the last non-planning moment of time, without much regard to consistency. Otherwise, contradictions will be handled by deleting everything after the present moment.

class allegedb.cache.**EdgesCache**(*db*)
A cache for remembering whether edges exist at a given time.

count_predecessors(*graph*, *dest*, *branch*, *turn*, *tick*, ***, *forward=None*)
Return the number of predecessors from a given destination node at a given time.

count_successors(*graph*, *orig*, *branch*, *turn*, *tick*, ***, *forward=None*)
Return the number of successors to a given origin node at a given time.

has_predecessor(*graph*, *dest*, *orig*, *branch*, *turn*, *tick*)
Return whether an edge connects the destination to the origin at the given time.

has_successor(*graph*, *orig*, *dest*, *branch*, *turn*, *tick*)
Return whether an edge connects the origin to the destination at the given time.

iter_predecessors(*graph*, *dest*, *branch*, *turn*, *tick*, ***, *forward=None*)
Iterate over predecessors to a given destination node at a given time.

iter_successors(*graph*, *orig*, *branch*, *turn*, *tick*, ***, *forward=None*)
Iterate over successors of a given origin node at a given time.

class allegedb.cache.**FuturistWindowDict**(*data=None*)
A WindowDict that does not let you rewrite the past.

class allegedb.cache.**NodesCache**(*db*)
A cache for remembering whether nodes exist at a given time.

class allegedb.cache.**PickyDefaultDict**(*type=<class 'object'>*, *args_munger=<function _default_args_munger>*, *kwargs_munger=<function _default_kwargs_munger>*)
A `defaultdict` alternative that requires values of a specific type.

Pass some type object (such as a class) to the constructor to specify what type to use by default, which is the only type I will accept.

Default values are constructed with no arguments by default; supply `args_munger` and/or `kwargs_munger` to override this. They take arguments `self` and the unused key being looked up.

class allegedb.cache.**SettingsTurnDict**(*data=None*)

cls
alias of `allegedb.window.WindowDict`

class allegedb.cache.**StructuredDefaultDict**(*layers*, *type=<class 'object'>*, *args_munger=<function _default_args_munger>*, *kwargs_munger=<function _default_kwargs_munger>*)

A `defaultdict`-like class that expects values stored at a specific depth.

Requires an integer to tell it how many layers deep to go. The innermost layer will be `PickyDefaultDict`, which will take the `type`, `args_munger`, and `kwargs_munger` arguments supplied to my constructor.

class allegedb.cache.**TurnDict**(*data=None*)

cls
alias of *FuturistWindowDict*

1.4 graph

allegedb's special implementations of the NetworkX graph objects

class allegedb.graph.**AbstractEntityMapping**

class allegedb.graph.**AbstractSuccessors**(*container*, *orig*)

clear()
Delete every edge with origin at my orig

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class allegedb.graph.**AllegedGraph**(*db*, *name*, *data=None*, ***attr*)
Class giving the graphs those methods they share in common.

clear()
Remove all nodes and edges from the graph.

Unlike the regular networkx implementation, this does *not* remove the graph's name. But all the other graph, node, and edge attributes go away.

graph_map_cls
alias of *GraphMapping*

node_map_cls
alias of *GraphNodeMapping*

class allegedb.graph.**AllegedMapping**
Common amenities for mappings

clear()
Delete everything

connect(*func*)
Arrange to call this function whenever something changes here.

The arguments will be this object, the key changed, and the value set.

disconnect(*func*)
No longer call the function when something changes here.

send(*sender*, ***kwargs*)
Internal. Call connected functions.

update(*other*, ***kwargs*)
Version of `update` that doesn't clobber the database so much

class allegedb.graph.**DiGraph**(*db*, *name*, *data=None*, ***attr*)
A version of the networkx.DiGraph class that stores its state in a database.

add_edge(*u*, *v*, *attr_dict=None*, ***attr*)
Version of add_edge that only writes to the database once

add_edges_from(*ebunch*, *attr_dict=None*, ***attr*)
Version of add_edges_from that only writes to the database once

adj_cls
alias of *DiGraphSuccessorsMapping*

pred_cls
alias of *DiGraphPredecessorsMapping*

remove_edge(*u*, *v*)
Version of remove_edge that's much like normal networkx but only deletes once, since the database doesn't keep separate adj and succ mappings

remove_edges_from(*ebunch*)
Version of remove_edges_from that's much like normal networkx but only deletes once, since the database doesn't keep separate adj and succ mappings

class allegedb.graph.**DiGraphPredecessorsMapping**(*graph*)
Mapping for Predecessors instances, which map to Edges that end at the dest provided to this

class Predecessors(*container*, *dest*)
Mapping of Edges that end at a particular node

class allegedb.graph.**DiGraphSuccessorsMapping**(*graph*)

class Successors(*container*, *orig*)

class allegedb.graph.**Edge**(*graph*, *orig*, *dest*, *idx=0*)
Mapping for edge attributes

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

exception allegedb.graph.**EntityCollisionError**
For when there's a discrepancy between the kind of entity you're creating and the one by the same name

class allegedb.graph.**Graph**(*db*, *name*, *data=None*, ***attr*)
A version of the networkx.Graph class that stores its state in a database.

adj_cls
alias of *GraphSuccessorsMapping*

class allegedb.graph.**GraphEdgeMapping**(*graph*)
Provides an adjacency mapping and possibly a predecessor mapping for a graph.

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class allegedb.graph.**GraphMapping**(*graph*)
Mapping for graph attributes

clear()
Delete everything

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

unwrap()
Return a deep copy of myself as a dict, and unwrap any wrapper objects in me.

class allegedb.graph.**GraphNodeMapping**(*graph*)
Mapping for nodes in a graph

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class allegedb.graph.**GraphSuccessorsMapping**(*graph*)
Mapping for Successors (itself a MutableMapping)

class Successors(*container*, *orig*)

class allegedb.graph.**MultiDiGraph**(*db*, *name*, *data=None*, ***attr*)
A version of the networkx.MultiDiGraph class that stores its state in a database.

add_edge(*u*, *v*, *key=None*, *attr_dict=None*, ***attr*)
Version of add_edge that only writes to the database once.

adj_cls
alias of *MultiDiGraphSuccessorsMapping*

pred_cls
alias of *MultiDiGraphPredecessorsMapping*

remove_edge(*u*, *v*, *key=None*)
Version of remove_edge that's much like normal networkx but only deletes once, since the database doesn't keep separate adj and succ mappings

remove_edges_from(*ebunch*)
Version of remove_edges_from that's much like normal networkx but only deletes once, since the database doesn't keep separate adj and succ mappings

class allegedb.graph.**MultiDiGraphPredecessorsMapping**(*graph*)
Version of DiGraphPredecessorsMapping for multigraphs

class Predecessors(*container*, *dest*)
Predecessor edges from a given node

class allegedb.graph.**MultiDiGraphSuccessorsMapping**(*graph*)

class Successors(*container*, *orig*)

class allegedb.graph.**MultiEdges**(*graph*, *orig*, *dest*)
Mapping of Edges between two nodes

clear()
Delete all edges between these nodes

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class allegedb.graph.**MultiGraph**(*db*, *name*, *data=None*, ***attr*)
A version of the networkx.MultiGraph class that stores its state in a database.

adj_cls
alias of *MultiGraphSuccessorsMapping*

class allegedb.graph.**MultiGraphSuccessorsMapping**(*graph*)
Mapping of Successors that map to MultiEdges

class Successors(*container*, *orig*)
Edges succeeding a given node in a multigraph

class allegedb.graph.**Node**(*graph*, *node*)
Mapping for node attributes

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

allegedb.graph.**convert_to_networkx_graph**(*data*, *create_using=None*, *multigraph_input=False*)
Convert an AllegedGraph to the corresponding NetworkX graph type.

allegedb.graph.**getatt**(*attribute_name*)
An easy way to make an alias

1.5 query

Wrapper to run SQL queries in a lightly abstracted way, such that code that's more to do with the queries than with the data per se doesn't pollute the other files so much.

class allegedb.query.**GlobalKeyValueStore**(*qe*)
A dict-like object that keeps its contents in a table.

Mostly this is for holding the current branch and revision.

class allegedb.query.**QueryEngine**(*dbstring*, *connect_args*, *alchemy*, *pack=None*, *unpack=None*)
Wrapper around either a DBAPI2.0 connection or an Alchemist. Provides methods to run queries using either.

all_branches()
Return all the branch data in tuples of (branch, parent, parent_turn).

close()
Commit the transaction, then close the connection

commit()
Commit the transaction

del_graph(*graph*)
Delete all records to do with the graph

edge_val_del(*graph*, *orig*, *dest*, *idx*, *key*, *branch*, *turn*, *tick*)
Declare that the key no longer applies to this edge, as of this branch and revision.

edge_val_dump()
Yield the entire contents of the edge_val table.

edge_val_set(*graph*, *orig*, *dest*, *idx*, *key*, *branch*, *turn*, *tick*, *value*)
Set this key of this edge to this value.

edges_dump()
Dump the entire contents of the edges table.

exist_edge(*graph*, *orig*, *dest*, *idx*, *branch*, *turn*, *tick*, *extant*)
Declare whether or not this edge exists.

exist_node(*graph*, *node*, *branch*, *turn*, *tick*, *extant*)
Declare that the node exists or doesn't.

Inserts a new record or updates an old one, as needed.

flush()
Put all pending changes into the SQL transaction.

global_del(*key*)
Delete the global record for the key.

global_get(*key*)
Return the value for the given key in the `globals` table.

global_items()
Iterate over (key, value) pairs in the `globals` table.

global_set(*key*, *value*)
Set `key` to `value` globally (not at any particular branch or revision)

graph_type(*graph*)
What type of graph is this?

graph_val_del(*graph*, *key*, *branch*, *turn*, *tick*)
Indicate that the key is unset.

graph_val_dump()
Yield the entire contents of the graph_val table.

have_branch(*branch*)
Return whether the branch thus named exists in the database.

have_graph(*graph*)
Return whether I have a graph by this name.

initdb()
Create tables and indices as needed.

new_branch(*branch*, *parent*, *parent_turn*, *parent_tick*)
Declare that the `branch` is descended from `parent` at `parent_turn`, `parent_tick`

new_graph(*graph*, *typ*)
Declare a new graph by this name of this type.

node_val_del(*graph*, *node*, *key*, *branch*, *turn*, *tick*)
Delete a key from a node at a specific branch and revision

node_val_dump()
Yield the entire contents of the node_val table.

node_val_set(*graph*, *node*, *key*, *branch*, *turn*, *tick*, *value*)
Set a key-value pair on a node at a specific branch and revision

nodes_dump()
Dump the entire contents of the nodes table.

sql(*stringname*, **args*, ***kwargs*)
Wrapper for the various prewritten or compiled SQL calls.

First argument is the name of the query, either a key in `sqlite.json` or a method name in `allegedb.alchemy.Alchemist`. The rest of the arguments are parameters to the query.

sqlmany(*stringname*, **args*)
Wrapper for executing many SQL calls on my connection.

First arg is the name of a query, either a key in the precompiled JSON or a method name in `allegedb.alchemy.Alchemist`. Remaining arguments should be tuples of argument sequences to be passed to the query.

exception allegedb.query.**TimeError**
Exception class for problems with the time model

1.6 wrap

Wrapper classes to let you store mutable data types in the allegedb ORM

class allegedb.wrap.**DictWrapper**(*getter*, *setter*, *outer*, *key*)
A dictionary synchronized with a serialized field.

This is meant to be used in allegedb entities (graph, node, or edge), for when the user stores a dictionary in them.

class allegedb.wrap.**ListWrapper**(*getter*, *setter*, *outer*, *key*)
A list synchronized with a serialized field.

This is meant to be used in allegedb entities (graph, node, or edge), for when the user stores a list in them.

append(*v*)
S.append(value) – append value to the end of the sequence

insert(*i*, *v*)
S.insert(index, value) – insert value before index

unwrap()
Return a deep copy of myself as a list, and unwrap any wrapper objects in me.

class allegedb.wrap.**MutableMappingWrapper**

unwrap()
Return a deep copy of myself as a dict, and unwrap any wrapper objects in me.

class allegedb.wrap.**MutableSequenceWrapper**

unwrap()
Return a deep copy of myself as a list, and unwrap any wrapper objects in me.

class allegedb.wrap.**MutableWrapper**

class allegedb.wrap.**MutableWrapperDictList**

class allegedb.wrap.**MutableWrapperSet**

add(*element*)
Add an element.

discard(*element*)
Remove an element. Do not raise an exception if absent.

pop()
Return the popped value. Raise KeyError if empty.

remove(*element*)
Remove an element. If not a member, raise a KeyError.

unwrap()
Return a deep copy of myself as a set, and unwrap any wrapper objects in me.

class allegedb.wrap.**SetWrapper**(*getter*, *setter*, *outer*, *key*)
A set synchronized with a serialized field.

This is meant to be used in allegedb entities (graph, node, or edge), for when the user stores a set in them.

class allegedb.wrap.**SubDictWrapper**(*getter*, *setter*)

class allegedb.wrap.**SubListWrapper**(*getter*, *setter*)

append(*object*)
S.append(value) – append value to the end of the sequence

insert(*index*, *object*)
S.insert(index, value) – insert value before index

class allegedb.wrap.**SubSetWrapper**(*getter*, *setter*)

class allegedb.wrap.**UnwrappingDict**
Dict that stores the data from the wrapper classes but won't store those objects themselves.

CHAPTER 2

LiSE

2.1 engine

The "engine" of LiSE is an object relational mapper with special stores for game data and entities, as well as properties for manipulating the flow of time.

class LiSE.engine.**AbstractEngine**

Parent class to the real Engine as well as EngineProxy.

Implements serialization methods and the __getattr__ for stored methods.

By default, the deserializers will refuse to create LiSE entities. If you want them to, use my `loading` property to open a `with` block, in which deserialized entities will be created as needed.

coinflip()

Return True or False with equal probability.

contextmanager()

@contextmanager decorator.

Typical usage:

@contextmanager def some_generator(<arguments>):

<setup> try:

yield <value>

finally: <cleanup>

This makes this:

with some_generator(<arguments>) as <variable>: <body>

equivalent to this:

<setup> try:

<variable> = <value> <body>

finally: <cleanup>

dice(*n*, *d*)
Roll `n` dice with `d` faces, and yield the results.

This is an iterator. You'll get the result of each die in succesion.

dice_check(*n*, *d*, *target*, *comparator='<='*)
Roll `n` dice with `d` sides, sum them, and return whether they are <= `target`.

If `comparator` is provided, use it instead of <=. You may use a string like '<' or '>='.

loading()
Context manager for when you need to instantiate entities upon unpacking

percent_chance(*pct*)
Given a `pct``% chance of something happening right now, decide at random whether it actually happens, and return ``True` or `False` as appropriate.

Values not between 0 and 100 are treated as though they were 0 or 100, whichever is nearer.

roll_die(*d*)
Roll a die with `d` faces. Return the result.

class LiSE.engine.**DummyEntity**(*engine*)
Something to use in place of a node or edge

class LiSE.engine.**Engine**(*worlddb*, ***, *string='strings.json'*, *function='function.py'*, *method='method.py'*, *trigger='trigger.py'*, *prereq='prereq.py'*, *action='action.py'*, *connect_args={}*, *alchemy=False*, *commit_modulus=None*, *random_seed=None*, *logfun=None*, *validate=False*, *clear_code=False*, *clear_world=False*)
LiSE, the Life Simulator Engine.

Each instance of LiSE maintains a connection to a database representing the state of a simulated world. Simulation rules within this world are described by lists of Python functions, some of which make changes to the world.

The top-level data structure within LiSE is the character. Most data within the world model is kept in some character or other; these will quite frequently represent people, but can be readily adapted to represent any kind of data that can be comfortably described as a graph or a JSON object. Every change to a character will be written to the database.

LiSE tracks history as a series of turns. In each turn, each simulation rule is evaluated once for each of the simulated entities it's been applied to. World changes in a given turn are remembered together, such that the whole world state can be rewound: simply set the properties `branch` and `turn` back to what they were just before the change you want to undo.

Properties:

- `branch`: The fork of the timestream that we're on.
- `turn`: Units of time that have passed since the sim started.
- `time`: `(branch, turn)`
- `tick`: A counter of how many changes have occurred this turn
- `character`: A mapping of *`Character`* objects by name.
- `rule`: A mapping of all rules that have been made.
- `rulebook`: A mapping of lists of rules. They are followed in their order. A whole rulebook full of rules may be assigned to an entity at once.

- `trigger`: A mapping of functions that might trigger a rule.
- `prereq`: A mapping of functions a rule might require to return `True` for it to run.
- `action`: A mapping of functions that might manipulate the world state as a result of a rule running.
- `function`: A mapping of generic functions.
- `string`: A mapping of strings, probably shown to the player at some point.
- `eternal`: Mapping of arbitrary serializable objects. It isn't sensitive to sim-time. A good place to keep game settings.
- `universal`: Another mapping of arbitrary serializable objects, but this one *is* sensitive to sim-time. Each turn, the state of the randomizer is saved here under the key `'rando_state'`.

class Character(*engine*, *name*, *data=None*, *, *init_rulebooks=True*, **attr*)
A graph that follows game rules and has a containment hierarchy.

Nodes in a Character are subcategorized into Things and Places. Things have locations, and those locations may be Places or other Things.

Characters may have avatars in other Characters. These are just nodes. You can apply rules to a Character's avatars, and thus to any collection of nodes you want, perhaps in many different Characters. But you may want a Character to have exactly one avatar, representing their location in physical space – the Character named 'physical'. So when a Character has only one avatar, you can treat the `avatar` property as an alias of the avatar.

class AvatarGraphMapping(*char*)
A mapping of other characters in which one has an avatar.

Maps to a mapping of the avatars themselves, unless there's only one other character you have avatars in, in which case this maps to those.

If you have only one avatar anywhere, you can pretend this is that entity.

class CharacterAvatarMapping(*outer*, *graphn*)
Mapping of avatars of one Character in another Character.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

node
If I have avatars in only one graph, return a map of them.

Otherwise, raise AttributeError.

only
If I have only one avatar, return it.

Otherwise, raise AttributeError.

class PlaceMapping(*character*)
`Place` objects that are in a `Character`

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class PortalPredecessorsMapping(*graph*)
Mapping of nodes that have at least one incoming edge.

Maps to another mapping keyed by the origin nodes, which maps to Portal objects.

class Predecessors(*container*, *dest*)
Mapping of possible origins from some destination.

class PortalSuccessorsMapping(*graph*)
Mapping of nodes that have at least one outgoing edge.

Maps them to another mapping, keyed by the destination nodes, which maps to Portal objects.

class Successors(*container*, *orig*)
Mapping for possible destinations from some node.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

static send(*self*, ***kwargs*)
Call all listeners to `dest` and to my `orig`.

character
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class ThingMapping(*character*)
Thing objects that are in a Character

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class ThingPlaceMapping(*character*)
GraphNodeMapping but for Place and Thing

character
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

add_avatar(*a*, *b=None*)
Start keeping track of a Thing or Place in a different Character.

add_places_from(*seq*, ***attrs*)
Take a series of place names and add the lot.

add_portal(*origin*, *destination*, *symmetrical=False*, ***kwargs*)
Connect the origin to the destination with a Portal.

Keyword arguments are the Portal's attributes. Exception: if keyword symmetrical == True, a mirror-Portal will be placed in the opposite direction between the same nodes. It will always appear to have the placed Portal's stats, and any change to the mirror Portal's stats will affect the placed Portal.

add_portals_from(*seq*, *symmetrical=False*)
Take a sequence of (origin, destination) pairs and make a Portal for each.

Actually, triples are acceptable too, in which case the third item is a dictionary of stats for the new `Portal`.

If optional argument `symmetrical` is set to `True`, all the `Portal` instances will have a mirror portal going in the opposite direction, which will always have the same stats.

add_thing(*name*, *location*, ***kwargs*)
Create a Thing, set its location, and set its initial attributes from the keyword arguments (if any).

adj_cls
alias of `Character.PortalSuccessorsMapping`

avatars()
Iterate over all my avatars, regardless of what character they are in.

del_avatar(*a*, *b=None*)
This is no longer my avatar, though it still exists on its own.

node_map_cls
alias of `Character.ThingPlaceMapping`

place2thing(*name*, *location*)
Turn a Place into a Thing with the given location.

It will keep all its attached Portals.

portals()
Iterate over all portals.

pred_cls
alias of `Character.PortalPredecessorsMapping`

thing2place(*name*)
Unset a Thing's location, and thus turn it into a Place.

class Place(*character*, *name*)
The kind of node where a thing might ultimately be located.

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

delete()
Remove myself from the world model immediately.

class Portal(*graph*, *orig*, *dest*, *idx=0*)
Connection between two Places that Things may travel along.

Portals are one-way, but you can make one appear two-way by setting the `symmetrical` key to `True`, eg. `character.add_portal(orig, dest, symmetrical=True)`. The portal going the other way will appear to have all the stats of this one, and attempting to set a stat on it will set it here instead.

character
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

delete()
Remove myself from my `Character`.

For symmetry with `Thing` and :class‘Place‘.

destination
Return the Place object at which I end

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter(‘name’), the call f(r) returns r.name. After g = attrgetter(‘name’, ‘date’), the call g(r) returns (r.name, r.date). After h = attrgetter(‘name.first’, ‘name.last’), the call h(r) returns (r.name.first, r.name.last).

origin
Return the Place object that is where I begin

reciprocal
If there’s another Portal connecting the same origin and destination that I do, but going the opposite way, return it. Else raise KeyError.

unwrap()
Return a deep copy of myself as a dict, and unwrap any wrapper objects in me.

update(*d*)
Works like regular update, but only actually updates when the new value and the old value differ. This is necessary to prevent certain infinite loops.

class QueryEngine(*dbstring*, *connect_args*, *alchemy*, *pack=None*, *unpack=None*)

exception IntegrityError

exception OperationalError

exist_edge(*character*, *orig*, *dest*, *idx*, *branch*, *turn*, *tick*, *extant=None*)
Declare whether or not this edge exists.

exist_node(*character*, *node*, *branch*, *turn*, *tick*, *extant*)
Declare that the node exists or doesn’t.

Inserts a new record or updates an old one, as needed.

initdb()
Set up the database schema, both for allegedb and the special extensions for LiSE

class Thing(*character*, *name*)
The sort of item that has a particular location at any given time.

If a Thing is in a Place, it is standing still. If it is in a Portal, it is moving through that Portal however fast it must in order to arrive at the other end when it is scheduled to. If it is in another Thing, then it is wherever that is, and moving the same.

clear()
Unset everything.

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter(‘name’), the call f(r) returns r.name. After g = attrgetter(‘name’, ‘date’), the call g(r) returns

(r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

delete()
Get rid of this, starting now.

Apart from deleting the node, this also informs all its users that it doesn't exist and therefore can't be their avatar anymore.

follow_path(*path*, *weight=None*)
Go to several `Place`s in succession, deciding how long to spend in each by consulting the ``weight` stat of the `Portal` connecting the one `Place` to the next.

Return the total number of turns the travel will take. Raise `TravelException` if I can't follow the whole path, either because some of its nodes don't exist, or because I'm scheduled to be somewhere else.

go_to_place(*place*, *weight=''*)
Assuming I'm in a `Place` that has a `Portal` direct to the given `Place`, schedule myself to travel to the given `Place`, taking an amount of time indicated by the `weight` stat on the `Portal`, if given; else 1 turn.

Return the number of turns the travel will take.

location
The `Thing` or `Place` I'm in.

travel_to(*dest*, *weight=None*, *graph=None*)
Find the shortest path to the given `Place` from where I am now, and follow it.

If supplied, the `weight` stat of the :class:'Portal's along the path will be used in pathfinding, and for deciding how long to stay in each Place along the way.

The `graph` argument may be any NetworkX-style graph. It will be used for pathfinding if supplied, otherwise I'll use my `Character`. In either case, however, I will attempt to actually follow the path using my `Character`, which might not be possible if the supplied `graph` and my `Character` are too different. If it's not possible, I'll raise a `TravelException`, whose `subpath` attribute holds the part of the path that I *can* follow. To make me follow it, pass it to my `follow_path` method.

Return value is the number of turns the travel will take.

add_character(*name*, *data=None*, ***kwargs*)
Create a new character.

You'll be able to access it as a *Character* object by looking up `name` in my `character` property.

`data`, if provided, should be a networkx-compatible graph object. Your new character will be a copy of it.

Any keyword arguments will be set as stats of the new character.

advance()
Follow the next rule if available.

If we've run out of rules, reset the rules iterator.

char_cls
alias of *LiSE.character.Character*

close()
Commit changes and close the database.

critical(*msg*)
Log a message at level 'critical'

debug(*msg*)
Log a message at level 'debug'

del_character(*name*)
Remove the Character from the database entirely.

This also deletes all its history. You'd better be sure.

edge_cls
alias of *LiSE.portal.Portal*

error(*msg*)
Log a message at level 'error'

get_delta(*branch*, *turn_from*, *tick_from*, *turn_to*, *tick_to*)
Get a dictionary describing changes to the world.

Most keys will be character names, and their values will be dictionaries of the character's stats' new values, with `None` for deleted keys. Characters' dictionaries have special keys 'nodes' and 'edges' which contain booleans indicating whether the node or edge exists at the moment, and 'node_val' and 'edge_val' for the stats of those entities. For edges (also called portals) these dictionaries are two layers deep, keyed first by the origin, then by the destination.

Characters also have special keys for the various rulebooks they have:

- 'character_rulebook'
- 'avatar_rulebook'
- 'character_thing_rulebook'
- 'character_place_rulebook'
- 'character_portal_rulebook'

And each node and edge may have a 'rulebook' stat of its own. If a node is a thing, it gets a 'location'; when the 'location' is deleted, that means it's back to being a place.

Keys at the top level that are not character names:

- 'rulebooks', a dictionary keyed by the name of each changed rulebook, the value

being a list of rule names * 'rules', a dictionary keyed by the name of each changed rule, containing any of the lists 'triggers', 'prereqs', and 'actions'

get_turn_delta(*branch=None*, *turn=None*, *tick=None*, *start_tick=0*)
Get a dictionary describing changes to the world within a given turn

Defaults to the present turn, and stops at the present tick unless specified.

See the documentation for `get_delta` for a detailed description of the delta format.

info(*msg*)
Log a message at level 'info'

new_character(*name*, *data=None*, ***kwargs*)
Create and return a new *Character*.

node_cls
alias of *LiSE.place.Place*

place_cls
alias of *LiSE.place.Place*

portal_cls
alias of *LiSE.portal.Portal*

query_engine_cls
alias of *LiSE.query.QueryEngine*

thing_cls
alias of *LiSE.thing.Thing*

warning(*msg*)
Log a message at level 'warning'

class LiSE.engine.**FinalRule**
A singleton sentinel for the rule iterator

exception LiSE.engine.**InnerStopIteration**

class LiSE.engine.**NextTurn**(*engine*)
Make time move forward in the simulation.

Calls `advance` repeatedly, returning a list of the rules' return values.

I am also a `Signal`, so you can register functions to be called when the simulation runs. Pass them to my `connect` method.

2.2 character

The top level of the LiSE world model, the Character.

Based on NetworkX DiGraph objects with various additions and conveniences.

A Character is a graph that follows rules. Its rules may be assigned to run on only some portion of it: just edges (called Portals), just nodes, or just nodes of the kind that have a location in another node (called Places and Things, respectively). Each Character has a `stat` property that acts very much like a dictionary, in which you can store game-relevant data for the rules to use.

You can designate some nodes in one Character as avatars of another, and then assign a rule to run on all of a Character's avatars. This is useful for the common case where someone in your game has a location in the physical world (here, a Character, called 'physical') but also has a behavior flowchart, or a skill tree, that isn't part of the physical world. In that case the flowchart is the person's Character, and their node in the physical world is an avatar of it.

class LiSE.character.**AbstractCharacter**
The Character API, with all requisite mappings and graph generators.

Mappings resemble those of a NetworkX digraph:

- `thing` and `place` are subsets of `node`
- `edge`, `adj`, and `succ` are aliases of `portal`
- `pred` is an alias to `preportal`
- `stat` is a dict-like mapping of data that changes over game-time,

to be used in place of graph attributes

become(*g*)
Erase all my nodes and edges. Replace them with a copy of the graph provided.

Return myself.

copy_from(*g*)
Copy all nodes and edges from the given graph into this.

Return myself.

cull_edges(*stat*, *threshold=0.5*, *comparator=<built-in function ge>*)
Delete edges whose stat >= `threshold` (default 0.5).

Optional argument `comparator` will replace >= as the test for whether to cull. You can use the name of a stored function.

cull_nodes(*stat*, *threshold=0.5*, *comparator=<built-in function ge>*)
Delete nodes whose stat >= `threshold` (default 0.5).

Optional argument `comparator` will replace >= as the test for whether to cull. You can use the name of a stored function.

cull_portals(*stat*, *threshold=0.5*, *comparator=<built-in function ge>*)
Delete portals whose stat >= `threshold` (default 0.5).

Optional argument `comparator` will replace >= as the test for whether to cull. You can use the name of a stored function.

do(*func*, **args*, ***kwargs*)
Apply the function to myself, and return myself.

Look up the function in the database if needed. Pass it any arguments given, keyword or positional.

Useful chiefly when chaining.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

grid_2d_8graph(*m*, *n*)
Make a 2d graph that's connected 8 ways, enabling diagonal movement

perlin(*stat='perlin'*)
Apply Perlin noise to my nodes, and return myself.

I'll try to use the name of the node as its spatial position for this purpose, or use its stats 'x', 'y', and 'z', or skip the node if neither are available. z is assumed 0 if not provided for a node.

Result will be stored in a node stat named 'perlin' by default. Supply the name of another stat to use it instead.

stat
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class LiSE.character.**CharRuleMapping**(*character*, *rulebook*, *booktyp*)
Wraps one of a character's rulebooks so you can get its rules by name.

You can access the rules in this either dictionary-style or as attributes. This is for convenience if you want to get at a rule's decorators, eg. to add an Action to the rule.

Using this as a decorator will create a new rule, named for the decorated function, and using the decorated function as the initial Action.

Using this like a dictionary will let you create new rules, appending them onto the underlying `RuleBook`; replace one rule with another, where the new one will have the same index in the `RuleBook` as the old one; and activate or deactivate rules. The name of a rule may be used in place of the actual rule, so long as the rule already exists.

You can also set a rule active or inactive by setting it to `True` or `False`, respectively. Inactive rules are still in the rulebook, but won't be followed.

class LiSE.character.**Character**(*engine*, *name*, *data=None*, *, *init_rulebooks=True*, ***attr*)

A graph that follows game rules and has a containment hierarchy.

Nodes in a Character are subcategorized into Things and Places. Things have locations, and those locations may be Places or other Things.

Characters may have avatars in other Characters. These are just nodes. You can apply rules to a Character's avatars, and thus to any collection of nodes you want, perhaps in many different Characters. But you may want a Character to have exactly one avatar, representing their location in physical space – the Character named 'physical'. So when a Character has only one avatar, you can treat the `avatar` property as an alias of the avatar.

class AvatarGraphMapping(*char*)

A mapping of other characters in which one has an avatar.

Maps to a mapping of the avatars themselves, unless there's only one other character you have avatars in, in which case this maps to those.

If you have only one avatar anywhere, you can pretend this is that entity.

class CharacterAvatarMapping(*outer*, *graphn*)

Mapping of avatars of one Character in another Character.

engine

attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name

attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

node

If I have avatars in only one graph, return a map of them.

Otherwise, raise AttributeError.

only

If I have only one avatar, return it.

Otherwise, raise AttributeError.

class PlaceMapping(*character*)

`Place` objects that are in a *Character*

engine

attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class PortalPredecessorsMapping(*graph*)
Mapping of nodes that have at least one incoming edge.

Maps to another mapping keyed by the origin nodes, which maps to Portal objects.

class Predecessors(*container*, *dest*)
Mapping of possible origins from some destination.

class PortalSuccessorsMapping(*graph*)
Mapping of nodes that have at least one outgoing edge.

Maps them to another mapping, keyed by the destination nodes, which maps to Portal objects.

class Successors(*container*, *orig*)
Mapping for possible destinations from some node.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

static send(*self*, ***kwargs*)
Call all listeners to `dest` and to my `orig`.

character
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class ThingMapping(*character*)
`Thing` objects that are in a *Character*

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class ThingPlaceMapping(*character*)
GraphNodeMapping but for Place and Thing

character
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

add_avatar(*a*, *b=None*)
Start keeping track of a `Thing` or `Place` in a different *Character*.

add_places_from(*seq*, ***attrs*)
Take a series of place names and add the lot.

add_portal(*origin*, *destination*, *symmetrical=False*, ***kwargs*)
Connect the origin to the destination with a `Portal`.

Keyword arguments are the `Portal`'s attributes. Exception: if keyword `symmetrical` == `True`, a mirror-`Portal` will be placed in the opposite direction between the same nodes. It will always appear to have the placed `Portal`'s stats, and any change to the mirror `Portal`'s stats will affect the placed `Portal`.

add_portals_from(*seq*, *symmetrical=False*)
Take a sequence of (origin, destination) pairs and make a `Portal` for each.

Actually, triples are acceptable too, in which case the third item is a dictionary of stats for the new `Portal`.

If optional argument `symmetrical` is set to `True`, all the `Portal` instances will have a mirror portal going in the opposite direction, which will always have the same stats.

add_thing(*name*, *location*, ***kwargs*)
Create a Thing, set its location, and set its initial attributes from the keyword arguments (if any).

adj_cls
alias of *Character.PortalSuccessorsMapping*

avatars()
Iterate over all my avatars, regardless of what character they are in.

del_avatar(*a*, *b=None*)
This is no longer my avatar, though it still exists on its own.

node_map_cls
alias of *Character.ThingPlaceMapping*

place2thing(*name*, *location*)
Turn a Place into a Thing with the given location.

It will keep all its attached Portals.

portals()
Iterate over all portals.

pred_cls
alias of *Character.PortalPredecessorsMapping*

thing2place(*name*)
Unset a Thing's location, and thus turn it into a Place.

class LiSE.character.**CharacterSense**(*container*, *sensename*)
Mapping for when you've selected a sense for a character to use but haven't yet specified what character to look at

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

func
Return the function most recently associated with this sense.

observer
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class LiSE.character.**CharacterSenseMapping**(*character*)
Used to view other Characters as seen by one, via a particular sense.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class LiSE.character.**Facade**(*character=None*)

class PlaceMapping(*facade*)

facadecls
alias of *FacadePlace*

innercls
alias of *LiSE.place.Place*

class PortalPredecessorsMapping(*facade*)

cls
alias of *FacadePortalPredecessors*

class PortalSuccessorsMapping(*facade*)

cls
alias of *FacadePortalSuccessors*

class StatMapping(*facade*)

class ThingMapping(*facade*)

facadecls
alias of *FacadeThing*

innercls
alias of *LiSE.thing.Thing*

add_edge(*orig*, *dest*, ***kwargs*)
Add an edge between u and v.

The nodes u and v will be automatically added if they are not already in the graph.

Edge attributes can be specified with keywords or by directly accessing the edge's attribute dictionary. See examples below.

u, v [nodes] Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

attr [keyword arguments, optional] Edge data (or labels or objects) can be assigned using keyword arguments.

add_edges_from : add a collection of edges

Adding an edge that already exists updates the edge data.

Many NetworkX algorithms designed for weighted graphs use an edge attribute (by default *weight*) to hold a numerical value.

The following all add the edge e=(1, 2) to graph G:

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1, 2)
>>> G.add_edge(1, 2)            # explicit two-node form
>>> G.add_edge(*e)              # single edge as tuple of two nodes
>>> G.add_edges_from( [(1, 2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

```
>>> G.add_edge(1, 2, weight=3)
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

For non-string attribute keys, use subscript notation.

```
>>> G.add_edge(1, 2)
>>> G[1][2].update({0: 5})
>>> G.edges[1, 2].update({0: 5})
```

add_node(*name*, ***kwargs*)

Add a single node *node_for_adding* and update node attributes.

node_for_adding [node] A node can be any hashable Python object except None.

attr [keyword arguments, optional] Set or change node attributes using key=value.

add_nodes_from

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn't change on mutables.

engine

attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class LiSE.character.**FacadeEntity**(*mapping*, ***kwargs*)

class LiSE.character.**FacadeEntityMapping**(*facade*)

Mapping that contains entities in a Facade.

All the entities are of the same type, `facadecls`, possibly being distorted views of entities of the type `innercls`.

engine

attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

class LiSE.character.**FacadePlace**(*mapping*, *real_or_name*, ***kwargs*)

Lightweight analogue of Place for Facade use.

class LiSE.character.**FacadePortal**(*mapping*, *other*, ***kwargs*)

Lightweight analogue of Portal for Facade use.

class LiSE.character.**FacadePortalMapping**(*facade*)

class LiSE.character.**FacadePortalPredecessors**(*facade*, *destname*)

facadecls
alias of *FacadePortal*

innercls
alias of *LiSE.portal.Portal*

class LiSE.character.**FacadePortalSuccessors**(*facade*, *origname*)

facadecls
alias of *FacadePortal*

innercls
alias of *LiSE.portal.Portal*

class LiSE.character.**FacadeThing**(*mapping*, *real_or_name*, ***kwargs*)

class LiSE.character.**RuleFollower**
Mixin class. Has a rulebook, which you can get a RuleMapping into.

class LiSE.character.**SenseFuncWrap**(*character*, *fun*)
Wrapper for a sense function that looks it up in the code store if provided with its name, and prefills the first two arguments.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

2.3 node

A base class for nodes that can be in a character.

Every actual node that you're meant to use will be a place or thing. This module is for what they have in common.

class LiSE.node.**Node**(*character*, *name*)
The fundamental graph component, which edges (in LiSE, "portals") go between.

Every LiSE node is either a thing or a place. They share in common the abilities to follow rules; to be connected by portals; and to contain things.

adj
Return a mapping of portals connecting this node to its neighbors.

character
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

clear()
Delete everything

delete()
Get rid of this, starting now.

Apart from deleting the node, this also informs all its users that it doesn't exist and therefore can't be their avatar anymore.

edge
Return a mapping of portals connecting this node to its neighbors.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

historical(*stat*)
Return a reference to the values that a stat has had in the past.

You can use the reference in comparisons to make a history query, and execute the query by calling it, or passing it to `self.engine.ticks_when`.

name
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

new_thing(*name*, ***stats*)
Create a new thing, located here, and return it.

one_way(*other*, ***stats*)
Connect a portal from here to another node, and return it.

one_way_portal(*other*, ***stats*)
Connect a portal from here to another node, and return it.

path_exists(*dest*, *weight=None*)
Return whether there is a path leading from me to `dest`.

With `weight`, only consider edges that have a stat by the given name.

Raise `ValueError` if `dest` is not a node in my character or the name of one.

portal
Return a mapping of portals connecting this node to its neighbors.

portals()
Iterate over `Portal` objects that lead away from me

predecessors()
Iterate over nodes with edges leading here from there.

preportals()
Iterate over `Portal` objects that lead to me

shortest_path(*dest*, *weight=None*)
Return a list of node names leading from me to `dest`.

Raise `ValueError` if `dest` is not a node in my character or the name of one.

shortest_path_length(*dest*, *weight=None*)
Return the length of the path from me to `dest`.

Raise `ValueError` if `dest` is not a node in my character or the name of one.

successor
Return a mapping of portals connecting this node to its neighbors.

successors()
Iterate over nodes with edges leading from here to there.

two_way(*other*, ***stats*)
Connect these nodes with a two-way portal and return it.

two_way_portal(*other*, ***stats*)
Connect these nodes with a two-way portal and return it.

class LiSE.node.**RuleMapping**(*node*)
Version of *LiSE.rule.RuleMapping* that works more easily with a node.

class LiSE.node.**UserDescriptor**
Give a node's user if there's only one

If there are many users, but one of them has the same name as this node, give that one.

Otherwise, raise AmbiguousUserError.

usermapping
alias of *UserMapping*

class LiSE.node.**UserMapping**(*node*)
A mapping of the characters that have a particular node as an avatar.

Getting characters from here isn't any better than getting them from the engine direct, but with this you can do things like use the .get() method to get a character if it's a user and otherwise get something else; or test whether the character's name is in the keys; and so on.

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

2.4 place

The type of node that is a location.

Though both things and places are nodes, things are obliged to be located in another node. Places are not.

class LiSE.place.**Place**(*character*, *name*)
The kind of node where a thing might ultimately be located.

db
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

delete()
Remove myself from the world model immediately.

2.5 thing

The sort of node that is ultimately located in a Place.

Things may be located in other Things as well, but eventually must be recursively located in a Place.

There's a subtle distinction between "location" and "containment": a Thing may be contained by a Portal, but cannot be located there – only in one of the Portal's endpoints. Things are both located in and contained by Places, or possibly other Things.

class LiSE.thing.**Thing**(*character*, *name*)

The sort of item that has a particular location at any given time.

If a Thing is in a Place, it is standing still. If it is in a Portal, it is moving through that Portal however fast it must in order to arrive at the other end when it is scheduled to. If it is in another Thing, then it is wherever that is, and moving the same.

clear()

Unset everything.

db

attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

delete()

Get rid of this, starting now.

Apart from deleting the node, this also informs all its users that it doesn't exist and therefore can't be their avatar anymore.

follow_path(*path*, *weight=None*)

Go to several `Place`s in succession, deciding how long to spend in each by consulting the ``weight` stat of the `Portal` connecting the one `Place` to the next.

Return the total number of turns the travel will take. Raise `TravelException` if I can't follow the whole path, either because some of its nodes don't exist, or because I'm scheduled to be somewhere else.

go_to_place(*place*, *weight=''*)

Assuming I'm in a `Place` that has a `Portal` direct to the given `Place`, schedule myself to travel to the given `Place`, taking an amount of time indicated by the `weight` stat on the `Portal`, if given; else 1 turn.

Return the number of turns the travel will take.

location

The `Thing` or `Place` I'm in.

travel_to(*dest*, *weight=None*, *graph=None*)

Find the shortest path to the given `Place` from where I am now, and follow it.

If supplied, the `weight` stat of the :class:'Portal's along the path will be used in pathfinding, and for deciding how long to stay in each Place along the way.

The `graph` argument may be any NetworkX-style graph. It will be used for pathfinding if supplied, otherwise I'll use my `Character`. In either case, however, I will attempt to actually follow the path using my `Character`, which might not be possible if the supplied `graph` and my `Character` are too different. If it's not possible, I'll raise a `TravelException`, whose `subpath` attribute holds the part of the path that I *can* follow. To make me follow it, pass it to my `follow_path` method.

Return value is the number of turns the travel will take.

2.6 portal

Directed edges, as used by LiSE.

class LiSE.portal.**Portal**(*graph*, *orig*, *dest*, *idx=0*)
Connection between two Places that Things may travel along.

Portals are one-way, but you can make one appear two-way by setting the `symmetrical` key to `True`, eg. `character.add_portal(orig, dest, symmetrical=True)`. The portal going the other way will appear to have all the stats of this one, and attempting to set a stat on it will set it here instead.

character
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

delete()
Remove myself from my `Character`.

For symmetry with `Thing` and :class'Place'.

destination
Return the Place object at which I end

engine
attrgetter(attr, ...) –> attrgetter object

Return a callable object that fetches the given attribute(s) from its operand. After f = attrgetter('name'), the call f(r) returns r.name. After g = attrgetter('name', 'date'), the call g(r) returns (r.name, r.date). After h = attrgetter('name.first', 'name.last'), the call h(r) returns (r.name.first, r.name.last).

origin
Return the Place object that is where I begin

reciprocal
If there's another Portal connecting the same origin and destination that I do, but going the opposite way, return it. Else raise KeyError.

unwrap()
Return a deep copy of myself as a dict, and unwrap any wrapper objects in me.

update(*d*)
Works like regular update, but only actually updates when the new value and the old value differ. This is necessary to prevent certain infinite loops.

class LiSE.portal.**RuleMapping**(*portal*)
Mapping to get rules followed by a portal.

2.7 rule

The fundamental unit of game logic, the Rule, and structures to store and organize them in.

A Rule is three lists of functions: triggers, prereqs, and actions. The actions do something, anything that you need your game to do, but probably making a specific change to the world model. The triggers and prereqs between them

specify when the action should occur: any of its triggers can tell it to happen, but then any of its prereqs may stop it from happening.

Rules are assembled into RuleBooks, essentially just lists of Rules that can then be assigned to be followed by any game entity – but each game entity has its own RuleBook by default, and you never really need to change that.

class LiSE.rule.**ActionList**(*rule*)
A list of action functions for rules

class LiSE.rule.**AllRuleBooks**(*engine*)

class LiSE.rule.**AllRules**(*engine*)
A mapping of every rule in the game.

You can use this as a decorator to make a rule and not assign it to anything.

new_empty(*name*)
Make a new rule with no actions or anything, and return it.

class LiSE.rule.**PrereqList**(*rule*)
A list of prereq functions for rules

class LiSE.rule.**Rule**(*engine*, *name*, *triggers=None*, *prereqs=None*, *actions=None*, *create=True*)
A collection of actions, being functions that enact some change on the world, which will be called each tick if and only if all of the prereqs return True, they being boolean functions that do not change the world.

action(*fun*)
Decorator to append the function to my actions list.

always()
Arrange to be triggered every tick, regardless of circumstance.

duplicate(*newname*)
Return a new rule that's just like this one, but under a new name.

prereq(*fun*)
Decorator to append the function to my prereqs list.

trigger(*fun*)
Decorator to append the function to my triggers list.

class LiSE.rule.**RuleBook**(*engine*, *name*)
A list of rules to be followed for some Character, or a part of it anyway.

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert(*i*, *v*)
S.insert(index, value) – insert value before index

class LiSE.rule.**RuleFollower**
Interface for that which has a rulebook associated, which you can get a *RuleMapping* into

class LiSE.rule.**RuleFuncList**(*rule*)
Abstract class for lists of functions like trigger, prereq, action

append(*v*)
S.append(value) – append value to the end of the sequence

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert(*i*, *v*)
S.insert(index, value) – insert value before index

class LiSE.rule.**RuleFuncListDescriptor**(*cls*)
Descriptor that lets you get and set a whole RuleFuncList at once

class LiSE.rule.**RuleMapping**(*engine*, *rulebook*)
Wraps a *RuleBook* so you can get its rules by name.

You can access the rules in this either dictionary-style or as attributes. This is for convenience if you want to get at a rule's decorators, eg. to add an Action to the rule.

Using this as a decorator will create a new rule, named for the decorated function, and using the decorated function as the initial Action.

Using this like a dictionary will let you create new rules, appending them onto the underlying *RuleBook*; replace one rule with another, where the new one will have the same index in the *RuleBook* as the old one; and activate or deactivate rules. The name of a rule may be used in place of the actual rule, so long as the rule already exists.

class LiSE.rule.**TriggerList**(*rule*)
A list of trigger functions for rules

LiSE.rule.**roundtrip_dedent**(*source*)
Reformat some lines of code into what unparse makes.

2.8 query

The query engine provides Pythonic methods to access the database.

This module also contains a notably unfinished implementation of a query language specific to LiSE. Access some stats using entities' method `historical`, and do comparisons on those, and instead of a boolean result you'll get a callable object that will return an iterator over turn numbers in which the comparison evaluated to `True`.

class LiSE.query.**QueryEngine**(*dbstring*, *connect_args*, *alchemy*, *pack=None*, *unpack=None*)

exception IntegrityError

exception OperationalError

exist_edge(*character*, *orig*, *dest*, *idx*, *branch*, *turn*, *tick*, *extant=None*)
Declare whether or not this edge exists.

exist_node(*character*, *node*, *branch*, *turn*, *tick*, *extant*)
Declare that the node exists or doesn't.

Inserts a new record or updates an old one, as needed.

initdb()
Set up the database schema, both for allegedb and the special extensions for LiSE

LiSE.query.**slow_iter_turns_eval_cmp**(*qry*, *oper*, *start_branch=None*, *engine=None*)
Iterate over all turns on which a comparison holds.

This is expensive. It evaluates the query for every turn in history.

LiSE.query.**windows_intersection**(*windows*)
Given a list of (beginning, ending), return another describing where they overlap.

Return type list

LiSE.query.**windows_union**(*windows*)
Given a list of (beginning, ending), return a minimal version that contains the same ranges.

Return type list

CHAPTER 3

ELiDE

3.1 game

3.2 board

3.2.1 pawn

3.2.2 spot

3.2.3 arrow

3.3 screen

3.4 card

3.5 charmenu

3.6 charsview

3.7 dialog

3.8 dummy

3.9 menu

3.10 pallet

3.11 rulesview

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

allegedb, 2
allegedb.cache, 5
allegedb.graph, 8
allegedb.query, 11
allegedb.wrap, 13

l

LiSE.character, 24
LiSE.engine, 15
LiSE.node, 32
LiSE.place, 34
LiSE.portal, 36
LiSE.query, 38
LiSE.rule, 36
LiSE.thing, 35

Index

A

AbstractCharacter (class in LiSE.character), 24
AbstractEngine (class in LiSE.engine), 15
AbstractEntityMapping (class in allegedb.graph), 8
AbstractSuccessors (class in allegedb.graph), 8
action() (LiSE.rule.Rule method), 37
ActionList (class in LiSE.rule), 37
add() (allegedb.wrap.MutableWrapperSet method), 14
add_avatar() (LiSE.character.Character method), 28
add_avatar() (LiSE.engine.Engine.Character method), 19
add_character() (LiSE.engine.Engine method), 22
add_edge() (allegedb.graph.DiGraph method), 9
add_edge() (allegedb.graph.MultiDiGraph method), 10
add_edge() (LiSE.character.Facade method), 30
add_edges_from() (allegedb.graph.DiGraph method), 9
add_node() (LiSE.character.Facade method), 31
add_places_from() (LiSE.character.Character method), 28
add_places_from() (LiSE.engine.Engine.Character method), 19
add_portal() (LiSE.character.Character method), 28
add_portal() (LiSE.engine.Engine.Character method), 19
add_portals_from() (LiSE.character.Character method), 28
add_portals_from() (LiSE.engine.Engine.Character method), 19
add_thing() (LiSE.character.Character method), 28
add_thing() (LiSE.engine.Engine.Character method), 20
adj (LiSE.node.Node attribute), 32
adj_cls (allegedb.graph.DiGraph attribute), 9
adj_cls (allegedb.graph.Graph attribute), 9
adj_cls (allegedb.graph.MultiDiGraph attribute), 10
adj_cls (allegedb.graph.MultiGraph attribute), 11
adj_cls (LiSE.character.Character attribute), 28
adj_cls (LiSE.engine.Engine.Character attribute), 20
advance() (LiSE.engine.Engine method), 22
advancing() (allegedb.ORM method), 2
all_branches() (allegedb.query.QueryEngine method), 11
allegedb (module), 2
allegedb.cache (module), 5
allegedb.graph (module), 8
allegedb.query (module), 11
allegedb.wrap (module), 13
AllegedGraph (class in allegedb.graph), 8
AllegedMapping (class in allegedb.graph), 8
AllRuleBooks (class in LiSE.rule), 37
AllRules (class in LiSE.rule), 37
always() (LiSE.rule.Rule method), 37
append() (allegedb.wrap.ListWrapper method), 13
append() (allegedb.wrap.SubListWrapper method), 14
append() (LiSE.rule.RuleFuncList method), 37
avatars() (LiSE.character.Character method), 28
avatars() (LiSE.engine.Engine.Character method), 20

B

batch() (allegedb.ORM method), 2
become() (LiSE.character.AbstractCharacter method), 24
branches (allegedb.cache.Cache attribute), 5
btt() (allegedb.ORM method), 2

C

Cache (class in allegedb.cache), 5
char_cls (LiSE.engine.Engine attribute), 22
Character (class in LiSE.character), 26
character (LiSE.character.Character.PortalSuccessorsMapping attribute), 27
character (LiSE.character.Character.ThingPlaceMapping attribute), 28
character (LiSE.engine.Engine.Character.PortalSuccessorsMapping attribute), 18
character (LiSE.engine.Engine.Character.ThingPlaceMapping attribute), 19
character (LiSE.engine.Engine.Portal attribute), 20
character (LiSE.node.Node attribute), 32
character (LiSE.portal.Portal attribute), 36
Character.AvatarGraphMapping (class in LiSE.character), 26
Character.AvatarGraphMapping.CharacterAvatarMapping (class in LiSE.character), 26

Character.PlaceMapping (class in LiSE.character), 26
Character.PortalPredecessorsMapping (class in LiSE.character), 27
Character.PortalPredecessorsMapping.Predecessors (class in LiSE.character), 27
Character.PortalSuccessorsMapping (class in LiSE.character), 27
Character.PortalSuccessorsMapping.Successors (class in LiSE.character), 27
Character.ThingMapping (class in LiSE.character), 27
Character.ThingPlaceMapping (class in LiSE.character), 28
CharacterSense (class in LiSE.character), 29
CharacterSenseMapping (class in LiSE.character), 29
CharRuleMapping (class in LiSE.character), 25
clear() (allegedb.graph.AbstractSuccessors method), 8
clear() (allegedb.graph.AllegedGraph method), 8
clear() (allegedb.graph.AllegedMapping method), 8
clear() (allegedb.graph.GraphMapping method), 10
clear() (allegedb.graph.MultiEdges method), 11
clear() (LiSE.engine.Engine.Thing method), 21
clear() (LiSE.node.Node method), 32
clear() (LiSE.thing.Thing method), 35
close() (allegedb.ORM method), 2
close() (allegedb.query.QueryEngine method), 12
close() (LiSE.engine.Engine method), 22
cls (allegedb.cache.SettingsTurnDict attribute), 7
cls (allegedb.cache.TurnDict attribute), 8
cls (LiSE.character.Facade.PortalPredecessorsMapping attribute), 30
cls (LiSE.character.Facade.PortalSuccessorsMapping attribute), 30
coinflip() (LiSE.engine.AbstractEngine method), 15
commit() (allegedb.ORM method), 2
commit() (allegedb.query.QueryEngine method), 12
connect() (allegedb.graph.AllegedMapping method), 8
contains_entity() (allegedb.cache.Cache method), 5
contains_entity_key() (allegedb.cache.Cache method), 5
contains_entity_or_key() (allegedb.cache.Cache method), 5
contains_key() (allegedb.cache.Cache method), 5
contextmanager() (allegedb.ORM method), 2
contextmanager() (LiSE.engine.AbstractEngine method), 15
convert_to_networkx_graph() (in module allegedb.graph), 11
copy_from() (LiSE.character.AbstractCharacter method), 24
count_entities() (allegedb.cache.Cache method), 5
count_entities_or_keys() (allegedb.cache.Cache method), 5
count_entity_keys() (allegedb.cache.Cache method), 5
count_keys() (allegedb.cache.Cache method), 6
count_predecessors() (allegedb.cache.EdgesCache method), 7
count_successors() (allegedb.cache.EdgesCache method), 7
critical() (LiSE.engine.Engine method), 22
cull_edges() (LiSE.character.AbstractCharacter method), 24
cull_nodes() (LiSE.character.AbstractCharacter method), 25
cull_portals() (LiSE.character.AbstractCharacter method), 25

D

db (allegedb.graph.AbstractSuccessors attribute), 8
db (allegedb.graph.Edge attribute), 9
db (allegedb.graph.GraphEdgeMapping attribute), 9
db (allegedb.graph.GraphMapping attribute), 10
db (allegedb.graph.GraphNodeMapping attribute), 10
db (allegedb.graph.MultiEdges attribute), 11
db (allegedb.graph.Node attribute), 11
db (LiSE.engine.Engine.Place attribute), 20
db (LiSE.engine.Engine.Thing attribute), 21
db (LiSE.place.Place attribute), 34
db (LiSE.thing.Thing attribute), 35
debug() (LiSE.engine.Engine method), 22
del_avatar() (LiSE.character.Character method), 29
del_avatar() (LiSE.engine.Engine.Character method), 20
del_character() (LiSE.engine.Engine method), 23
del_graph() (allegedb.ORM method), 3
del_graph() (allegedb.query.QueryEngine method), 12
delete() (LiSE.engine.Engine.Place method), 20
delete() (LiSE.engine.Engine.Portal method), 20
delete() (LiSE.engine.Engine.Thing method), 22
delete() (LiSE.node.Node method), 32
delete() (LiSE.place.Place method), 34
delete() (LiSE.portal.Portal method), 36
delete() (LiSE.thing.Thing method), 35
destination (LiSE.engine.Engine.Portal attribute), 21
destination (LiSE.portal.Portal attribute), 36
dice() (LiSE.engine.AbstractEngine method), 16
dice_check() (LiSE.engine.AbstractEngine method), 16
DictWrapper (class in allegedb.wrap), 13
DiGraph (class in allegedb.graph), 9
DiGraphPredecessorsMapping (class in allegedb.graph), 9
DiGraphPredecessorsMapping.Predecessors (class in allegedb.graph), 9
DiGraphSuccessorsMapping (class in allegedb.graph), 9
DiGraphSuccessorsMapping.Successors (class in allegedb.graph), 9
discard() (allegedb.wrap.MutableWrapperSet method), 14
disconnect() (allegedb.graph.AllegedMapping method), 8
do() (LiSE.character.AbstractCharacter method), 25

DummyEntity (class in LiSE.engine), 16
duplicate() (LiSE.rule.Rule method), 37

E

Edge (class in allegedb.graph), 9
edge (LiSE.node.Node attribute), 33
edge_cls (allegedb.ORM attribute), 3
edge_cls (LiSE.engine.Engine attribute), 23
edge_val_del() (allegedb.query.QueryEngine method), 12
edge_val_dump() (allegedb.query.QueryEngine method), 12
edge_val_set() (allegedb.query.QueryEngine method), 12
edges_dump() (allegedb.query.QueryEngine method), 12
EdgesCache (class in allegedb.cache), 7
Engine (class in LiSE.engine), 16
engine (LiSE.character.AbstractCharacter attribute), 25
engine (LiSE.character.Character.AvatarGraphMapping attribute), 26
engine (LiSE.character.Character.PlaceMapping attribute), 26
engine (LiSE.character.Character.PortalSuccessorsMapping attribute), 27
engine (LiSE.character.Character.PortalSuccessorsMapping.Successors attribute), 27
engine (LiSE.character.Character.ThingMapping attribute), 27
engine (LiSE.character.Character.ThingPlaceMapping attribute), 28
engine (LiSE.character.CharacterSense attribute), 29
engine (LiSE.character.CharacterSenseMapping attribute), 29
engine (LiSE.character.Facade attribute), 31
engine (LiSE.character.FacadeEntityMapping attribute), 31
engine (LiSE.character.SenseFuncWrap attribute), 32
engine (LiSE.engine.Engine.Character.AvatarGraphMapping attribute), 17
engine (LiSE.engine.Engine.Character.PlaceMapping attribute), 18
engine (LiSE.engine.Engine.Character.PortalSuccessorsMapping attribute), 18
engine (LiSE.engine.Engine.Character.PortalSuccessorsMapping.Successors attribute), 18
engine (LiSE.engine.Engine.Character.ThingMapping attribute), 19
engine (LiSE.engine.Engine.Character.ThingPlaceMapping attribute), 19
engine (LiSE.engine.Engine.Portal attribute), 21
engine (LiSE.node.Node attribute), 33
engine (LiSE.node.UserMapping attribute), 34
engine (LiSE.portal.Portal attribute), 36
Engine.Character (class in LiSE.engine), 17
Engine.Character.AvatarGraphMapping (class in LiSE.engine), 17
Engine.Character.AvatarGraphMapping.CharacterAvatarMapping (class in LiSE.engine), 17
Engine.Character.PlaceMapping (class in LiSE.engine), 17
Engine.Character.PortalPredecessorsMapping (class in LiSE.engine), 18
Engine.Character.PortalPredecessorsMapping.Predecessors (class in LiSE.engine), 18
Engine.Character.PortalSuccessorsMapping (class in LiSE.engine), 18
Engine.Character.PortalSuccessorsMapping.Successors (class in LiSE.engine), 18
Engine.Character.ThingMapping (class in LiSE.engine), 18
Engine.Character.ThingPlaceMapping (class in LiSE.engine), 19
Engine.Place (class in LiSE.engine), 20
Engine.Portal (class in LiSE.engine), 20
Engine.QueryEngine (class in LiSE.engine), 21
Engine.QueryEngine.IntegrityError, 21
Engine.QueryEngine.OperationalError, 21
Engine.Thing (class in LiSE.engine), 21
EntityCollisionError, 9
error() (LiSE.engine.Engine method), 23
exist_edge() (allegedb.query.QueryEngine method), 12
exist_edge() (LiSE.engine.Engine.QueryEngine method), 21
exist_edge() (LiSE.query.QueryEngine method), 38
exist_node() (allegedb.query.QueryEngine method), 12
exist_node() (LiSE.engine.Engine.QueryEngine method), 21
exist_node() (LiSE.query.QueryEngine method), 38

F

Facade (class in LiSE.character), 29
Facade.PlaceMapping (class in LiSE.character), 29
Facade.PortalPredecessorsMapping (class in LiSE.character), 30
Facade.PortalSuccessorsMapping (class in LiSE.character), 30
Facade.StatMapping (class in LiSE.character), 30
Facade.ThingMapping (class in LiSE.character), 30
facadecls (LiSE.character.Facade.PlaceMapping attribute), 29
facadecls (LiSE.character.Facade.ThingMapping attribute), 30
facadecls (LiSE.character.FacadePortalPredecessors attribute), 32
facadecls (LiSE.character.FacadePortalSuccessors attribute), 32
FacadeEntity (class in LiSE.character), 31
FacadeEntityMapping (class in LiSE.character), 31
FacadePlace (class in LiSE.character), 31
FacadePortal (class in LiSE.character), 31

FacadePortalMapping (class in LiSE.character), 31
FacadePortalPredecessors (class in LiSE.character), 31
FacadePortalSuccessors (class in LiSE.character), 32
FacadeThing (class in LiSE.character), 32
FinalRule (class in LiSE.engine), 24
flush() (allegedb.query.QueryEngine method), 12
follow_path() (LiSE.engine.Engine.Thing method), 22
follow_path() (LiSE.thing.Thing method), 35
func (LiSE.character.CharacterSense attribute), 29
FuturistWindowDict (class in allegedb.cache), 7

G

get_delta() (allegedb.ORM method), 3
get_delta() (LiSE.engine.Engine method), 23
get_graph() (allegedb.ORM method), 3
get_turn_delta() (allegedb.ORM method), 3
get_turn_delta() (LiSE.engine.Engine method), 23
getatt() (in module allegedb.graph), 11
global_del() (allegedb.query.QueryEngine method), 12
global_get() (allegedb.query.QueryEngine method), 12
global_items() (allegedb.query.QueryEngine method), 12
global_set() (allegedb.query.QueryEngine method), 12
GlobalKeyValueStore (class in allegedb.query), 11
go_to_place() (LiSE.engine.Engine.Thing method), 22
go_to_place() (LiSE.thing.Thing method), 35
Graph (class in allegedb.graph), 9
graph_map_cls (allegedb.graph.AllegedGraph attribute), 8
graph_type() (allegedb.query.QueryEngine method), 12
graph_val_del() (allegedb.query.QueryEngine method), 12
graph_val_dump() (allegedb.query.QueryEngine method), 12
GraphEdgeMapping (class in allegedb.graph), 9
GraphMapping (class in allegedb.graph), 10
GraphNameError, 2
GraphNodeMapping (class in allegedb.graph), 10
GraphSuccessorsMapping (class in allegedb.graph), 10
GraphSuccessorsMapping.Successors (class in allegedb.graph), 10
grid_2d_8graph() (LiSE.character.AbstractCharacter method), 25

H

has_predecessor() (allegedb.cache.EdgesCache method), 7
has_successor() (allegedb.cache.EdgesCache method), 7
have_branch() (allegedb.query.QueryEngine method), 12
have_graph() (allegedb.query.QueryEngine method), 12
historical() (LiSE.node.Node method), 33

I

index() (LiSE.rule.RuleBook method), 37
index() (LiSE.rule.RuleFuncList method), 37
info() (LiSE.engine.Engine method), 23
initdb() (allegedb.ORM method), 3
initdb() (allegedb.query.QueryEngine method), 12
initdb() (LiSE.engine.Engine.QueryEngine method), 21
initdb() (LiSE.query.QueryEngine method), 38
innercls (LiSE.character.Facade.PlaceMapping attribute), 29
innercls (LiSE.character.Facade.ThingMapping attribute), 30
innercls (LiSE.character.FacadePortalPredecessors attribute), 32
innercls (LiSE.character.FacadePortalSuccessors attribute), 32
InnerStopIteration, 24
insert() (allegedb.wrap.ListWrapper method), 13
insert() (allegedb.wrap.SubListWrapper method), 14
insert() (LiSE.rule.RuleBook method), 37
insert() (LiSE.rule.RuleFuncList method), 37
is_parent_of() (allegedb.ORM method), 3
iter_entities() (allegedb.cache.Cache method), 6
iter_entities_or_keys() (allegedb.cache.Cache method), 6
iter_entity_keys() (allegedb.cache.Cache method), 6
iter_keys() (allegedb.cache.Cache method), 6
iter_predecessors() (allegedb.cache.EdgesCache method), 7
iter_successors() (allegedb.cache.EdgesCache method), 7

K

keycache (allegedb.cache.Cache attribute), 6
keys (allegedb.cache.Cache attribute), 6

L

LiSE.character (module), 24
LiSE.engine (module), 15
LiSE.node (module), 32
LiSE.place (module), 34
LiSE.portal (module), 36
LiSE.query (module), 38
LiSE.rule (module), 36
LiSE.thing (module), 35
ListWrapper (class in allegedb.wrap), 13
load() (allegedb.cache.Cache method), 6
loading() (LiSE.engine.AbstractEngine method), 16
location (LiSE.engine.Engine.Thing attribute), 22
location (LiSE.thing.Thing attribute), 35

M

MultiDiGraph (class in allegedb.graph), 10
MultiDiGraphPredecessorsMapping (class in allegedb.graph), 10
MultiDiGraphPredecessorsMapping.Predecessors (class in allegedb.graph), 10
MultiDiGraphSuccessorsMapping (class in allegedb.graph), 10

MultiDiGraphSuccessorsMapping.Successors (class in allegedb.graph), 11
MultiEdges (class in allegedb.graph), 11
MultiGraph (class in allegedb.graph), 11
MultiGraphSuccessorsMapping (class in allegedb.graph), 11
MultiGraphSuccessorsMapping.Successors (class in allegedb.graph), 11
MutableMappingWrapper (class in allegedb.wrap), 13
MutableSequenceWrapper (class in allegedb.wrap), 13
MutableWrapper (class in allegedb.wrap), 14
MutableWrapperDictList (class in allegedb.wrap), 14
MutableWrapperSet (class in allegedb.wrap), 14

N

name (LiSE.character.Character.AvatarGraphMapping attribute), 26
name (LiSE.character.Character.PlaceMapping attribute), 26
name (LiSE.character.Character.ThingMapping attribute), 27
name (LiSE.character.Character.ThingPlaceMapping attribute), 28
name (LiSE.engine.Engine.Character.AvatarGraphMapping attribute), 17
name (LiSE.engine.Engine.Character.PlaceMapping attribute), 18
name (LiSE.engine.Engine.Character.ThingMapping attribute), 19
name (LiSE.engine.Engine.Character.ThingPlaceMapping attribute), 19
name (LiSE.node.Node attribute), 33
nbtt() (allegedb.ORM method), 3
new_branch() (allegedb.query.QueryEngine method), 12
new_character() (LiSE.engine.Engine method), 23
new_digraph() (allegedb.ORM method), 3
new_empty() (LiSE.rule.AllRules method), 37
new_graph() (allegedb.ORM method), 3
new_graph() (allegedb.query.QueryEngine method), 13
new_multidigraph() (allegedb.ORM method), 3
new_multigraph() (allegedb.ORM method), 4
new_thing() (LiSE.node.Node method), 33
NextTurn (class in LiSE.engine), 24
Node (class in allegedb.graph), 11
Node (class in LiSE.node), 32
node (LiSE.character.Character.AvatarGraphMapping attribute), 26
node (LiSE.engine.Engine.Character.AvatarGraphMapping attribute), 17
node_cls (allegedb.ORM attribute), 4
node_cls (LiSE.engine.Engine attribute), 23
node_map_cls (allegedb.graph.AllegedGraph attribute), 8
node_map_cls (LiSE.character.Character attribute), 29
node_map_cls (LiSE.engine.Engine.Character attribute), 20
node_val_del() (allegedb.query.QueryEngine method), 13
node_val_dump() (allegedb.query.QueryEngine method), 13
node_val_set() (allegedb.query.QueryEngine method), 13
nodes_dump() (allegedb.query.QueryEngine method), 13
NodesCache (class in allegedb.cache), 7

O

observer (LiSE.character.CharacterSense attribute), 29
one_way() (LiSE.node.Node method), 33
one_way_portal() (LiSE.node.Node method), 33
only (LiSE.character.Character.AvatarGraphMapping attribute), 26
only (LiSE.engine.Engine.Character.AvatarGraphMapping attribute), 17
origin (LiSE.engine.Engine.Portal attribute), 21
origin (LiSE.portal.Portal attribute), 36
ORM (class in allegedb), 2

P

parents (allegedb.cache.Cache attribute), 6
path_exists() (LiSE.node.Node method), 33
percent_chance() (LiSE.engine.AbstractEngine method), 16
perlin() (LiSE.character.AbstractCharacter method), 25
PickyDefaultDict (class in allegedb.cache), 7
Place (class in LiSE.place), 34
place2thing() (LiSE.character.Character method), 29
place2thing() (LiSE.engine.Engine.Character method), 20
place_cls (LiSE.engine.Engine attribute), 23
plan() (allegedb.ORM method), 4
PlanningContext (class in allegedb), 4
pop() (allegedb.wrap.MutableWrapperSet method), 14
Portal (class in LiSE.portal), 36
portal (LiSE.node.Node attribute), 33
portal_cls (LiSE.engine.Engine attribute), 23
portals() (LiSE.character.Character method), 29
portals() (LiSE.engine.Engine.Character method), 20
portals() (LiSE.node.Node method), 33
pred_cls (allegedb.graph.DiGraph attribute), 9
pred_cls (allegedb.graph.MultiDiGraph attribute), 10
pred_cls (LiSE.character.Character attribute), 29
pred_cls (LiSE.engine.Engine.Character attribute), 20
predecessors() (LiSE.node.Node method), 33
preportals() (LiSE.node.Node method), 33
prereq() (LiSE.rule.Rule method), 37
PrereqList (class in LiSE.rule), 37
presettings (allegedb.cache.Cache attribute), 6

Q

query_engine_cls (allegedb.ORM attribute), 4

query_engine_cls (LiSE.engine.Engine attribute), 23
QueryEngine (class in allegedb.query), 11
QueryEngine (class in LiSE.query), 38
QueryEngine.IntegrityError, 38
QueryEngine.OperationalError, 38

R

reciprocal (LiSE.engine.Engine.Portal attribute), 21
reciprocal (LiSE.portal.Portal attribute), 36
remove() (allegedb.wrap.MutableWrapperSet method), 14
remove_edge() (allegedb.graph.DiGraph method), 9
remove_edge() (allegedb.graph.MultiDiGraph method), 10
remove_edges_from() (allegedb.graph.DiGraph method), 9
remove_edges_from() (allegedb.graph.MultiDiGraph method), 10
retrieve() (allegedb.cache.Cache method), 6
roll_die() (LiSE.engine.AbstractEngine method), 16
roundtrip_dedent() (in module LiSE.rule), 38
Rule (class in LiSE.rule), 37
RuleBook (class in LiSE.rule), 37
RuleFollower (class in LiSE.character), 32
RuleFollower (class in LiSE.rule), 37
RuleFuncList (class in LiSE.rule), 37
RuleFuncListDescriptor (class in LiSE.rule), 37
RuleMapping (class in LiSE.node), 34
RuleMapping (class in LiSE.portal), 36
RuleMapping (class in LiSE.rule), 38

S

send() (allegedb.graph.AlleggedMapping method), 9
send() (LiSE.character.Character.PortalSuccessorsMapping.Successors static method), 27
send() (LiSE.engine.Engine.Character.PortalSuccessorsMapping.Successors static method), 18
SenseFuncWrap (class in LiSE.character), 32
setedge() (in module allegedb), 5
setedgeval() (in module allegedb), 5
setgraphval() (in module allegedb), 5
setnode() (in module allegedb), 5
setnodeval() (in module allegedb), 5
settings (allegedb.cache.Cache attribute), 6
SettingsTurnDict (class in allegedb.cache), 7
SetWrapper (class in allegedb.wrap), 14
shallowest (allegedb.cache.Cache attribute), 7
shortest_path() (LiSE.node.Node method), 33
shortest_path_length() (LiSE.node.Node method), 33
slow_iter_turns_eval_cmp() (in module LiSE.query), 38
sql() (allegedb.query.QueryEngine method), 13
sqlmany() (allegedb.query.QueryEngine method), 13
stat (LiSE.character.AbstractCharacter attribute), 25
store() (allegedb.cache.Cache method), 7
StructuredDefaultDict (class in allegedb.cache), 7
SubDictWrapper (class in allegedb.wrap), 14
SubListWrapper (class in allegedb.wrap), 14
SubSetWrapper (class in allegedb.wrap), 14
successor (LiSE.node.Node attribute), 33
successors() (LiSE.node.Node method), 34

T

Thing (class in LiSE.thing), 35
thing2place() (LiSE.character.Character method), 29
thing2place() (LiSE.engine.Engine.Character method), 20
thing_cls (LiSE.engine.Engine attribute), 24
TimeError, 13
TimeSignal (class in allegedb), 4
TimeSignalDescriptor (class in allegedb), 4
travel_to() (LiSE.engine.Engine.Thing method), 22
travel_to() (LiSE.thing.Thing method), 35
trigger() (LiSE.rule.Rule method), 37
TriggerList (class in LiSE.rule), 38
TurnDict (class in allegedb.cache), 8
two_way() (LiSE.node.Node method), 34
two_way_portal() (LiSE.node.Node method), 34

U

unwrap() (allegedb.graph.GraphMapping method), 10
unwrap() (allegedb.wrap.ListWrapper method), 13
unwrap() (allegedb.wrap.MutableMappingWrapper method), 13
unwrap() (allegedb.wrap.MutableSequenceWrapper method), 14
unwrap() (allegedb.wrap.MutableWrapperSet method), 14
unwrap() (LiSE.engine.Engine.Portal method), 21
unwrap() (LiSE.portal.Portal method), 36
UnwrappingDict (class in allegedb.wrap), 14
update() (allegedb.graph.AlleggedMapping method), 9
update() (LiSE.engine.Engine.Portal method), 21
update() (LiSE.portal.Portal method), 36
UserDescriptor (class in LiSE.node), 34
UserMapping (class in LiSE.node), 34
usermapping (LiSE.node.UserDescriptor attribute), 34

W

warning() (LiSE.engine.Engine method), 24
windows_intersection() (in module LiSE.query), 38
windows_union() (in module LiSE.query), 38