Cortix Documentation

Release 0.1.0

Valmor F. de Almeida, Taha Azzaoui, Gilberto E. Alas

Oct 11, 2019

CONTENTS

CHAPTER

ONE

SRC

1.1 cortix_main

class cortix_main.**Cortix**(*use_mpi=False*, *splash=False*)

Bases: object

Cortix main class definition.

The typical Cortix run file workflow:

1. Create the *Cortix* object
2. Create tne (nested) network of modules
3. Run and close *Cortix*

__del__()

Destructs a Cortix simulation object.

> **Warning:** By the time the body of this function is executed, the machinery of variables may have been deleted already. For example, *logging* is no longer there; do the least amount of work here.

__init__(*use_mpi=False*, *splash=False*)

Construct a Cortix simulation object.

Parameters

- **use_mpi** (*bool*) – True for MPI, False for multiprocessing.
- **splash** (*bool*) – Show the Cortix splash image.

network

Network – A network of modules and their connectivity.

use_mpi

bool – *True* for MPI, *False* for Multiprocessing.

use_multiprocessing

bool – *False* for MPI, *True* for Multiprocessing.

splash

bool – Show the Cortix splash image.

comm

mpi4py.MPI.Intracomm – MPI.COMM_WORLD (if using MPI else None).

rank
int – The current MPI rank (if using MPI else None).

size
int – size of the group associated with MPI.COMM_WORLD.

close()
Closes the cortix object properly before destruction.

User is strongly advised to call this method at the end of the run file otherwise timings will not be recorded.

network

run(*save=False*)
Run the Cortix network simulation.

1.2 module module

class module.**Module**
Bases: `object`

Cortix module super class.

This class provides facilities for creating modules within the Cortix network. Cortix will map one object of this class to either a Multiprocessing or MPI process depending on the user's configuration.

Note: This class is to be inherited by every Cortix module. In order to execute, modules *must* override the *run* method, which will be executed during the simulation.

__init__()
Module super class constructor.

Note: This constructor must be called explicitly in the constructor of every Cortix module like so:

super().__init__()

name
str – A name given to the instance. Default is the derived class name.

port_names_expected
list(str), None – A list of names of ports expected in the module. This will be compared to port names during runtime to check against the intended use of the module.

state
any – Any *pickle-able* data structure to be passed in a *multiprocessing.Queue* to the parent process or to be gathered in the root MPI process. Default is *None*.

use_mpi
bool – *True* for MPI, *False* for Multiprocessing

use_multiprocessing
bool – *False* for MPI, *True* for Multiprocessing

ports
list(Port) – A list of ports contained by the module

id
int – An integer set by the external network once a module is added to it. The *id* is the position of the module in the network list. Default: None.

__network
Network – An internal network inherited by the derived module for nested networks.

get_port (*name*)
Get port by name; if it does not exist, create one.

Parameters **name** (*str*) – The name of the port to get

Returns **port** – The port object with the corresponding name

Return type *Port*

network

recv (*port*)
Receive data from a given port

Warning: This function will block until data is available

Parameters **port** (Port, *str*) – A Port object to send the data through, or its string name

Returns **data** – The data received through the port

Return type any

run (**args*)
Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **arg[0]** (*int*) – Index of the state in the communication queue.

- **arg[1]** (*multiprocessing.Queue*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

run_and_save()

send(*data*, *port*)
Send data through a given port.

Parameters

- **data** (*any*) – The data being sent out - must be pickleable
- **port** (Port, *str*) – A Port object to send the data through, or its string name

1.3 network module

class network.**Network**
Bases: object

Cortix network.

num_networks
int – Number of instances of this class.

__init__()
Module super class constructor.

max_n_modules_for_data_copy_on_root
int – When using MPI the *network* will copy the data from all modules on the root process. This can generate an *out of memory* condition. This variable sets the maximum number of processes for which the data will be copied. Default is 1000.

add_module(*m*)
Alternative name to *module()*.

connect(*module_port_a*, *module_port_b*, *info=None*)
Connect two modules using either their ports directly or inferred ports.

A connection always opens a channel for data communication in both ways. That is, both sends and receives are allowed.

Note: The simplest form of usage is with arguments: (*module_a*, *module_b*). In this case, a *port* with the name of *module_a* **must** exist in *module_b*, and vice-versa (port names as *str* in lower case). In addition, the connect must not be called again with these same two modules, else the underlying connection will be overriden.

For more rigorous connection, the user is advised to fully specify the module and the port in each list argument.

Parameters

- **module_port_a** (*list([Module,Port]) or list([Module,str]) or* Module) – First *module-port* to connect.
- **module_port_b** (*list([Module,Port]) or list([Module,str]) or* Module) – Second *module-port* to connect.

- **info** (*str*) – Information on the directionality of the information flow. This is for graph visualization purposes only. The default value will use the order in the argument list to define the direction. Default: None. If set to *bidiretional*, will create a double headed arrow in the graph figure.

draw(*graph_attr=None, node_attr=None, engine='twopi', lr=False, ports=False, node_shape='hexagon'*)

module(*m*)
Add a module.

num_networks = 0

run()
Run the network simulation.

This function concurrently executes the *cortix.src.module.run* function for each module in the network. Modules are run using either MPI or Multiprocessing, depending on the user configuration.

Note: When using multiprocessing, data from the modules state are copied to the master process after the *run()* method of the modules is finished.

1.4 node module

class node.**Graph**
Bases: `object`

add(*mod*)

connect(*m1, m2*)

1.5 port module

class port.**Port**(*name=None, use_mpi=False*)
Bases: `object`

Provides a method of communication between modules.

The Port class provides an interface for creating ports and connecting them to other ports for the purpose of data transfer. Data exchange takes place by send and/or receive calls on a given port. The concept of a port is that of a data transfer "interaction." This can be one- or two-way with sends and receives. A port is connected to only one other port; as two ends of a pipe are connected.

__eq__(*other*)
Check for port equality

__init__(*name=None, use_mpi=False*)
Constructs a Port object

Parameters

- **name** (*str*) – The name of the Port object
- **use_mpi** (*bool*) – True for MPI, False for Multiprocessing

id
int

name
string

use_mpi
bool

`__repr__`()
Port name representation

connect(*port*)
Connect this port to another port

Ports must be connected for data to flow between them.

Parameters **port** (`Port`) – A Port object to connect to

recv()
Receive data from the connected port.

Warning: This function will block if no data has been sent yet.

Returns **data**

Return type any

send(*data*, *tag=None*)
Send data to the connected port.

If the sending port is not connected do nothing.

Parameters

- **data** (*any*) – This data must be pickleable
- **tag** (*int, optional*) – MPI tag used in sending data

CHAPTER

TWO

EXAMPLES

2.1 city_justice

2.1.1 adjudication module

class adjudication.**Adjudication**(*n_groups=1*, *pool_size=0.0*)

Bases: `cortix.src.module.Module`

Adjudication Cortix module used to model criminal group population in an adjudication system.

Notes

These are the *port* names available in this module to connect to respective modules: *probation*, *jail*, *arrested*, *prison*, and *community*. See instance attribute *port_names_expected*.

__init__(*n_groups=1*, *pool_size=0.0*)

Parameters

- **n_groups** (*int*) – Number of groups in the population.
- **pool_size** (*float*) – Upperbound on the range of the existing population groups. A random value from 0 to the upperbound value will be assigned to each group.

run(**args*)

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **arg[0]** (*int*) – Index of the state in the communication queue.
- **arg[1]** (*multiprocessing.Queue*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.1.2 arrested module

class arrested.**Arrested**(*n_groups=1*, *pool_size=0.0*)

Bases: `cortix.src.module.Module`

Arrested Cortix module used to model criminal group population in an arrested system.

Notes

These are the *port* names available in this module to connect to respective modules: *probation*, *adjudication*, *jail*, and *community*. See instance attribute *port_names_expected*.

__init__(*n_groups=1*, *pool_size=0.0*)

Parameters

- **n_groups** (*int*) – Number of groups in the population.
- **pool_size** (*float*) – Upperbound on the range of the existing population groups. A random value from 0 to the upperbound value will be assigned to each group.

run(**args*)

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **arg[0]** (*int*) – Index of the state in the communication queue.
- **arg[1]** (*multiprocessing.Queue*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.1.3 community module

class community.**Community**(*n_groups=1*, *non_offender_adult_population=100*, *offender_pool_size=0.0*, *free_offender_pool_size=0.0*)

Bases: `cortix.src.module.Module`

Community Cortix module used to model criminal group population in a community system. Community here is the system at large with all possible adult individuals included in a society.

Notes

These are the *port* names available in this module to connect to respective modules: *probation*, *adjudication*, *jail*, *prison*, *arrested*, and *parole*. See instance attribute *port_names_expected*.

__init__(*n_groups=1*, *non_offender_adult_population=100*, *offender_pool_size=0.0*, *free_offender_pool_size=0.0*)

Parameters

- **n_groups** (*int*) – Number of groups in the population.
- **non_offender_adult_population** (*float*) – Pool of individuals reaching the adult age (SI) unit. Default: 100.
- **offender_pool_size** (*float*) – Upperbound on the range of the existing population groups. A random value from 0 to the upperbound value will be assigned to each group. This is typically a small number, say a fraction of a percent.

run(**args*)

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use 'run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **`arg[0]`** (*`int`*) – Index of the state in the communication queue.
- **`arg[1]`** (*`multiprocessing.Queue`*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.1.4 jail module

class `jail.`**`Jail`**(*n_groups=1*, *pool_size=0.0*)

Bases: `cortix.src.module.Module`

Jail Cortix module used to model criminal group population in a jail.

Notes

These are the *port* names available in this module to connect to respective modules: *probation*, *adjudication*, *arrested*, *prison*, and *community*. See instance attribute *port_names_expected*.

`__init__`(*n_groups=1*, *pool_size=0.0*)

Parameters

- **`n_groups`** (*`int`*) – Number of groups in the population.
- **`pool_size`** (*`float`*) – Upperbound on the range of the existing population groups. A random value from 0 to the upperbound value will be assigned to each group.

`run`(**args*)

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **`arg[0]`** (*`int`*) – Index of the state in the communication queue.
- **`arg[1]`** (*`multiprocessing.Queue`*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.1.5 parole module

class `parole.`**`Parole`**(*n_groups=1*, *pool_size=0.0*)

Bases: `cortix.src.module.Module`

Parole Cortix module used to model criminal group population in a parole system.

Notes

These are the *port* names available in this module to connect to respective modules: *prison* and *community*. See instance attribute *port_names_expected*.

`run`(**args*)

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **arg[0]** (*int*) – Index of the state in the communication queue.
- **arg[1]** (*multiprocessing.Queue*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.1.6 prison module

class prison.**Prison**(*n_groups=1, pool_size=0.0*)

Bases: `cortix.src.module.Module`

Prison Cortix module used to model criminal group population in a prison.

Notes

These are the *port* names available in this module to connect to respective modules: *parole*, *adjudication*, *jail*, and *community*. See instance attribute *port_names_expected*.

__init__(*n_groups=1, pool_size=0.0*)

Parameters

- **n_groups** (*int*) – Number of groups in the population.
- **pool_size** (*float*) – Upperbound on the range of the existing population groups. A random value from 0 to the upperbound value will be assigned to each group.

run(**args*)

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **arg[0]** (*int*) – Index of the state in the communication queue.
- **arg[1]** (*multiprocessing.Queue*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.1.7 probation module

class probation.**Probation**(*n_groups=1*, *pool_size=0.0*)

Bases: `cortix.src.module.Module`

Probation Cortix module used to model criminal group population in a probation.

Notes

These are the *port* names available in this module to connect to respective modules: *adjudication*, *jail*, *arrested*, and *community*. See instance attribute *port_names_expected*.

__init__(*n_groups=1*, *pool_size=0.0*)

Parameters

- **n_groups** (*int*) – Number of groups in the population.
- **pool_size** (*float*) – Upperbound on the range of the existing population groups. A random value from 0 to the upperbound value will be assigned to each group.

run(**args*)

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use 'run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **arg[0]** (*int*) – Index of the state in the communication queue.
- **arg[1]** (*multiprocessing.Queue*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.1.8 run_city_justice module

Crimninal justice network dynamics modeling.

This example uses 7 modules:

- Community
- Arrested
- Adjudication
- Jail
- Prison
- Probation
- Parole

and a population balance model is used to follow the offenders population groups between modules.

To run this case using MPI you should compute the number of processes as follows:

nprocs = 7 + 1 cortix

then issue the MPI run command as follows (replace *nprocs* with a number):

mpiexec -n nprocs run_justice.py

To run this case with the Python multiprocessing library, just run this file at the command line as

run_city_justice.py

`run_city_justice.`**`main`**`()`
Cortix run file for a criminal justice network.

`run_city_justice.`**`n_groups`**
int – Number of population groups being followed. This must be the same for all modules.

`run_city_justice.`**`end_time`**
float – End of the flow time in SI unit.

`run_city_justice.`**`time_step`**
float – Size of the time step between port communications in SI unit.

`run_city_justice.`**`use_mpi`**
bool – If set to *True* use MPI otherwise use Python multiprocessing.

2.2 droplet_swirl

2.2.1 droplet module

class `droplet.`**`Droplet`**

Bases: `cortix.src.module.Module`

Droplet Cortix module used to model very simple fluid-particle interactions.

Notes

Port names used in this module: *external-flow* exchanges data with any other module that provides information about the flow outside the droplet, *visualization* sends data to a visualization module.

`__init__``()`

`initial_time`
float

`end_time`
float

`time_step`
float

`show_time`
tuple – Two-element tuple, *(bool,float)*, *True* will print to standard output.

`run``(*args)`

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **`arg[0]`** (*`int`*) – Index of the state in the communication queue.
- **`arg[1]`** (*`multiprocessing.Queue`*) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.2.2 run_droplet_swirl module

This example uses two modules instantiated many times. It be executed with MPI (if *mpi4py* is available) or with the Python multiprocessing library. These choices are made by variables listed below in the executable portion of this run file.

To run this case using MPI you should compute the number of processes as follows:

nprocs = n_droplets + 1 vortex + 1 cortix

then issue the MPI run command as follows (replace *nprocs* with a number):

mpiexec -n nprocs run_droplet.py

To run this case with the Python multiprocessing library, just run this file at the command line as

run_droplet.py

`run_droplet_swirl.`**`main`**`()`
Cortix run file for a *Droplet-Vortex* network.

`run_droplet_swirl.`**`n_droplets`**
int – Number of droplets to use (one per process).

`run_droplet_swirl.`**`end_time`**
float – End of the flow time in SI unit.

`run_droplet_swirl.`**`time_step`**
float – Size of the time step between port communications in SI unit.

`run_droplet_swirl.`**`create_plots`**
bool – Create various plots and save to files. (all data collected in the parent process; it may run out of memory).

`run_droplet_swirl.`**`plot_vortex_profile`**
bool – Whether to plot (to a file) the vortex function used.

`run_droplet_swirl.`**`use_mpi`**
bool – If set to *True* use MPI otherwise use Python multiprocessing.

2.2.3 vortex module

`class` `vortex.`**`Vortex`**
Bases: `cortix.src.module.Module`

Vortex module used to model fluid flow using Cortix.

Notes

Any *port* name and any number of ports are allowed.

`__init__`()

`initial_time`
float

`end_time`
float

`time_step`
float

`show_time`
tuple – Two-element tuple, *(bool,float)*, *True* will print to standard output.

`compute_velocity`(*time*, *position*)
Compute the vortex velocity at the given external position using a vortex flow model

Parameters

- **`time`** (`float`) – Time in SI unit.
- **`position`** (`numpy.ndarray (3)`) – Spatial position in SI unit.

Returns **vortex_velocity**

Return type numpy.ndarray(3)

`plot_velocity`(*time=None*)
Plot the vortex velocity as a function of height.

`run`(**args*)
Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **`arg[0]`** (`int`) – Index of the state in the communication queue.

- **arg[1]** (`multiprocessing.Queue`) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

2.3 nbody

2.3.1 body module

class `body.`**Body**(*mass=1*, *rad=(0, 0, 0)*, *vel=(0, 0, 0)*, *time=100*, *dt=0.01*)

Bases: `cortix.src.module.Module`

broadcast_data()

dump(*file_name=None*)

force_from(*other_mass*, *other_rad*)

gather_data()

run()

Module run function

Run method with an option to pass data back to the parent process when running in Python multiprocessing mode. If the user does not want to share data with the parent process, this function can be overriden with *run(self)* or *run(self, *args)* as long as *self.state = None*. If *self.state* points to anything but *None*, the user must use ‘run(self, *args).

Notes

When in multiprocessing, **args* has two elements: *comm_idx* and *comm_state*. To pass back the state of the module, the user should insert the provided index *comm_idx* and the *state* into the queue as follows:

if self.use_multiprocessing:

try: pickle.dumps(self.state)

except pickle.PicklingError: args[1].put((arg[0],None))

else: args[1].put((arg[0],self.state))

at the bottom of the user defined *run()* function.

Warning: This function must be overridden by all Cortix modules

Parameters

- **arg[0]** (`int`) – Index of the state in the communication queue.
- **arg[1]** (`multiprocessing.Queue`) – When using the Python *multiprocessing* library *state_comm* must have the module's *self.state* in it. That is, *state_comm.put((idx_comm,self.state))* must be the last command in the method before *return*. In addition, self.state must be *pickle-able*.

step()

2.3.2 run_planets module

`run_planets.`**`main`**`()`

`run_planets.`**`plot_trajectories`**`(`*traj=None*`)`

CHAPTER THREE

SUPPORT

3.1 nuclear

3.1.1 actor module

This is a simple way to hide the name of species of interest in a simulation. The user would modify and copy this class into the Cortix module of interest and keep it private. Author: Valmor de Almeida dealmeidav@ornl.gov; vfda Sat Aug 15 13:41:12 EDT 2015

class `actor.`**`Actor`**(*name*)

Bases: `object`

See atoms list in Specie.

`atoms`

Returns the specific nuclides found in the specified chemical.

Returns atoms

Return type list(str)

`formula`

Returns the formula of the chemical in question.

Returns formula

Return type str

3.1.2 fuel_bucket module

This FuelBucket class is a container for usage with other plant-level process modules. It is meant to represent a fuel bucket of a metal fuel reactor. ———- ATTENTION: ———- This container uses Phase() for phases (cladding and fuel). Therefore user is responsible to make the "history" of the phases consistent. See Phase() info.

Author: Valmor de Almeida dealmeidav@ornl.gov; vfda

class `fuel_bucket.`**`FuelBucket`**(*specs=Empty DataFrame Columns: [] Index: []*)

Bases: `object`

`__repr__`()

Converts to string.

`__str__`()

Converts to string.

`cladding_end_thickness`
Gets the thickness of the hemispherical cladding end caps that are placed on the top and bottom of the fuel slug, in cm.

Returns **cladding_end_thickness**

Return type float

`cladding_mass`
Returns the total mass of cladding material in the bucket, in grams.

Returns **cladding_mass**

Return type float

`cladding_phase`
Returns the phase history of the cladding.

Returns **cladding_phase**

Return type dataFrame

`cladding_volume`
Returns the total volume of cladding in the bucket, in cm^3.

Returns **cladding_volume**

Return type float

`cladding_wall_thickness`
Returns the thickness of the cladding wall which is on the outside of every fuel slug, and in between both sections of fuel, in cm.

Returns **cladding_wall_thickness**

Return type float

`fresh_u235_mass`
Returns the total amount of uranium-235 in the bucket, in grams.

Returns **fresh_u235_mass**

Return type float

`fresh_u238_mass`
Returns the total amount of uranium-238 in the bucket, in grams.

Returns **fresh_u238_mass**

Return type float

`fresh_u_mass`
Returns the total amount of uranium in the bucket, in grams.

Returns **fresh_u_mass**

Return type float

`fuel_enrichment`
Returns the enrichment of the fuel slugs in the bucket, in %.

Returns **fuel_enrichment**

Return type float

`fuel_mass`
Returns the total mass of fuel in the solid phase in the bucket.

Returns **fuel_mass**

Return type float

`fuel_mass_unit`
Returns the unit that is used to measure the mass of fuel in the bucket.

Returns **fuel_mass_unit**

Return type str

`fuel_phase`
Returns the phase history of the fuel.

Returns **fuel_phase**

Return type pandas.core.frame.DataFrame

`fuel_radioactivity`
Returns the total radioactivity of the solid phase fuel, in units of curies.

Returns **fuel_radioactivity**

Return type float

`fuel_volume`
Returns the total volume of fuel in the entire bucket, in cm^3.

Returns **fuel_volume**

Return type float

`gamma_pwr`
Returns the amount of gamma radiation given off by the fuel bucket, in units of watts.

Returns **gamma_pwr**

Return type float

`get_cladding_end_thickness()`
Gets the thickness of the hemispherical cladding end caps that are placed on the top and bottom of the fuel slug, in cm.

Returns **cladding_end_thickness**

Return type float

`get_cladding_mass()`
Returns the total mass of cladding material in the bucket, in grams.

Returns **cladding_mass**

Return type float

`get_cladding_phase()`
Returns the phase history of the cladding.

Returns **cladding_phase**

Return type dataFrame

`get_cladding_volume()`
Returns the total volume of cladding in the bucket, in cm^3.

Returns **cladding_volume**

Return type float

`get_cladding_wall_thickness()`
Returns the thickness of the cladding wall which is on the outside of every fuel slug, and in between both sections of fuel, in cm.

Returns **cladding_wall_thickness**

Return type float

`get_fresh_u235_mass()`
Returns the total amount of uranium-235 in the bucket, in grams.

Returns **fresh_u235_mass**

Return type float

`get_fresh_u238_mass()`
Returns the total amount of uranium-238 in the bucket, in grams.

Returns **fresh_u238_mass**

Return type float

`get_fresh_u_mass()`
Returns the total amount of uranium in the bucket, in grams.

Returns **fresh_u_mass**

Return type float

`get_fuel_enrichment()`
Returns the enrichment of the fuel slugs in the bucket, in %.

Returns **fuel_enrichment**

Return type float

`get_fuel_mass()`
Returns the total mass of fuel in the solid phase in the bucket.

Returns **fuel_mass**

Return type float

`get_fuel_mass_unit()`
Returns the unit that is used to measure the mass of fuel in the bucket.

Returns **fuel_mass_unit**

Return type str

`get_fuel_phase()`
Returns the phase history of the fuel.

Returns **fuel_phase**

Return type pandas.core.frame.DataFrame

`get_fuel_radioactivity()`
Returns the total radioactivity of the solid phase fuel, in units of curies.

Returns **fuel_radioactivity**

Return type float

`get_fuel_volume()`
Returns the total volume of fuel in the entire bucket, in cm^3.

Returns **fuel_volume**

Return type float

get_gamma_pwr()

Returns the amount of gamma radiation given off by the fuel bucket, in units of watts.

Returns **gamma_pwr**

Return type float

get_heat_pwr()

Returns the total amount of heat generated by the bucket, in units of watts.

Returns **heat_pwr**

Return type float

get_inner_slug_id()

Returns the inner diameter of the inner section of fuel, in cm.

Returns **inner_slug_id**

Return type float

get_inner_slug_od()

Returns the outer diameter of the inner section of fuel, in cm.

Returns **inner_slug_od**

Return type float

get_n_slugs()

Returns the number of fuel slugs in the bucket.

Returns **n_slugs**

Return type int

get_name()

Returns the name of the fuel bucket.

Returns **name**

Return type str

get_outer_slug_id()

Returns the inner diameter of the outer section of fuel, in cm.

Returns **outer_slug_id**

Return type float

get_outer_slug_od()

Returns the outer diameter of the outer section of fuel, in cm. A fuel slug consists of an outer section of fuel and an inner section of fuel, with cladding on the outside of the slug and between the inner and outer sections of fuel.

Returns **outer_slug_od**

Return type float

get_radioactivity()

Returns the radioactivity of the fuel bucket, in units of curies.

Returns **radioactivity**

Return type float

get_slug_cladding_volume()
Returns the volume of cladding present in a single fuel slug, in cm^3.

Returns **slug_cladding_volume**

Return type float

get_slug_fuel_volume()
Returns the volume of fuel present in a single fuel slug, in cm^3.

Returns **slug_fuel_volume**

Return type float

get_slug_length()
Returns the length of each slug in the fuel bucket.

Returns **slug_length**

Return type float

get_slug_type()
Returns the type of slugs being stored in the bucket (inner slug or outer slug).

Returns **slug_type**

Return type str

heat_pwr
Returns the total amount of heat generated by the bucket, in units of watts.

Returns **heat_pwr**

Return type float

inner_slug_id
Returns the inner diameter of the inner section of fuel, in cm.

Returns **inner_slug_id**

Return type float

inner_slug_od
Returns the outer diameter of the inner section of fuel, in cm.

Returns **inner_slug_od**

Return type float

n_slugs
Returns the number of fuel slugs in the bucket.

Returns **n_slugs**

Return type int

name
Returns the name of the fuel bucket.

Returns **name**

Return type str

outer_slug_id
Returns the inner diameter of the outer section of fuel, in cm.

Returns **outer_slug_id**

Return type float

`outer_slug_od`
Returns the outer diameter of the outer section of fuel, in cm. A fuel slug consists of an outer section of fuel and an inner section of fuel, with cladding on the outside of the slug and between the inner and outer sections of fuel.

Returns **outer_slug_od**

Return type float

`radioactivity`
Returns the radioactivity of the fuel bucket, in units of curies.

Returns **radioactivity**

Return type float

`set_cladding_phase`(*phase*)
Set's the phase history to specific values.

Parameters **`phase`** (`dataFrame`) –

`set_fuel_phase`(*phase*)
Sets the current fuel phase to a specified phase value.

Parameters **`phase`** (`dataFrame`) –

`set_slug_length`(*x*)
Sets the length of all slugs in the bucket to x. Used for chopping.

Parameters **`x`** (`float`) –

`slug_cladding_volume`
Returns the volume of cladding present in a single fuel slug, in cm^3.

Returns **slug_cladding_volume**

Return type float

`slug_fuel_volume`
Returns the volume of fuel present in a single fuel slug, in cm^3.

Returns **slug_fuel_volume**

Return type float

`slug_length`
Returns the length of each slug in the fuel bucket.

Returns **slug_length**

Return type float

`slug_type`
Returns the type of slugs being stored in the bucket (inner slug or outer slug).

Returns **slug_type**

Return type str

3.1.3 fuel_bundle module

This FuelBundle class is a container for usage with other plant-level process modules. It is meant to represent a fuel bundle of an oxide fuel LWR reactor. There are three main data structures:

1. fuel bundle specs
2. solid phase
3. gas phase

The container user will have to provide all the data and from then on, this class will help acess the data. The printing methods reveal the contained data.

Author: Valmor de Almeida dealmeidav@ornl.gov; vfda Sun Dec 27 15:06:55 EST 2015

class `fuel_bundle.`**`FuelBundle`**(*specs=Empty DataFrame Columns: [] Index: []*)

Bases: `object`

`fresh_u235_mass`

Returns the amount of uranium-235 in the bucket, in grams.

Returns **fresh_u235_mass**

Return type float

`fresh_u238_mass`

Returns the amount of uranium-238 in the bucket, in grams.

Returns **fresh_u238_mass**

Return type float

`fresh_u_mass`

Returns the amount of uranium in the bundle, in grams.

Returns **fresh_u_mass**

Return type float

`fuel_enrichment`

Returns the enrichment of the fuel pins in the bundle, in %.

Returns **fuel_enrichment**

Return type float

`fuel_mass`

Returns the total numerical value for mass of fuel in the solid phase in the bundle.

Returns **fuel_mass**

Return type float

`fuel_mass_unit`

Returns the unit used to measure the mass of fuel in the bundle.

Returns **fuel_mass_unit**

Return type str

`fuel_pin_length`

Returns the length of each fuel pin in the fuel bundle. A fuel pin is a cylindircal section of uranium fuel that is surrounded by cladding.

Returns **fuel_pin_length**

Return type float

`fuel_pin_radius`

Returns the radius of the fuel pin, in cm.

`fuel_pin_volume`
Returns the volume of fuel in each fuel pin, in cm^3.

Returns **fuel_pin_volume**

Return type float

`fuel_radioactivity`
Returns the total radioactivity of the fuel in the solid phase in the fuel bundle.

Returns **fuel_radioactivity**

Return type float

`fuel_rod_od`
Returns the outer diameter of the fuel rod, in cm. A fuel rod consists of a fuel pin surrounded by cladding.

Returns **fuel_rod_od**

Return type float

`fuel_volume`
Returns the total volume of fuel in the bundle, in cm^3.

Returns **fuel_volume**

Return type float

`gamma_pwr`
Returns the total amount of gamma radiation given by the fuel bundle, in watts.

Returns **gamma_pwr**

Return type float

`gas_mass`
Returns the total numerical value for mass of the fuel in the gas phase.

`gas_phase`
Returns the gas phase history of the fuel.

Returns **gas_phase**

Return type dataFrame

`gas_radioactivity`
Returns the total radioactivity of the fuel in the gas phase in the fuel bundle, in curies.

Returns **gas_radioactivity**

Return type float

`get_fresh_U235_mass``()`
Returns the amount of uranium-235 in the bucket, in grams.

Returns **fresh_u235_mass**

Return type float

`get_fresh_u238_mass``()`
Returns the amount of uranium-238 in the bucket, in grams.

Returns **fresh_u238_mass**

Return type float

`get_fresh_u_mass``()`
Returns the amount of uranium in the bundle, in grams.

Returns **fresh_u_mass**

Return type float

get_fuel_enrichment()
Returns the enrichment of the fuel pins in the bundle, in %.

Returns **fuel_enrichment**

Return type float

get_fuel_mass()
Returns the total numerical value for mass of fuel in the solid phase in the bundle.

Returns **fuel_mass**

Return type float

get_fuel_mass_unit()
Returns the unit used to measure the mass of fuel in the bundle.

Returns **fuel_mass_unit**

Return type str

get_fuel_pin_length()
Returns the length of each fuel pin in the fuel bundle. A fuel pin is a cylindircal section of uranium fuel that is surrounded by cladding.

Returns **fuel_pin_length**

Return type float

get_fuel_pin_radius()
Returns the radius of the fuel pin, in cm.

get_fuel_pin_volume()
Returns the volume of fuel in each fuel pin, in cm^3.

Returns **fuel_pin_volume**

Return type float

get_fuel_radioactivity()
Returns the total radioactivity of the fuel in the solid phase in the fuel bundle.

Returns **fuel_radioactivity**

Return type float

get_fuel_rod_od()
Returns the outer diameter of the fuel rod, in cm. A fuel rod consists of a fuel pin surrounded by cladding.

Returns **fuel_rod_od**

Return type float

get_fuel_volume()
Returns the total volume of fuel in the bundle, in cm^3.

Returns **fuel_volume**

Return type float

get_gamma_pwr()
Returns the total amount of gamma radiation given by the fuel bundle, in watts.

Returns **gamma_pwr**

Return type float

get_gas_mass()
Returns the total numerical value for mass of the fuel in the gas phase.

get_gas_phase()
Returns the gas phase history of the fuel.

Returns **gas_phase**

Return type dataFrame

get_gas_radioactivity()
Returns the total radioactivity of the fuel in the gas phase in the fuel bundle, in curies.

Returns **gas_radioactivity**

Return type float

get_heat_pwr()
Returns the total amount of heat produced by the fuel bundle, in watts.

Returns **heat_pwr**

Return type float

get_n_fuel_rods()
Returns the number of fuel rods in the bundle.

Returns **n_fuel_rods**

Return type int

get_name()
Returns the name of the fuel bundle.

Returns **name**

Return type str

get_radioactivity()
Returns the total radioactivity of the fuel bundle, in curies.

Returns **raduioactivity**

Return type float

get_solid_phase()
Returns the solid phase history associated with this fuel bundle.

Returns **solidPhase**

Return type dataFrame

heat_pwr
Returns the total amount of heat produced by the fuel bundle, in watts.

Returns **heat_pwr**

Return type float

n_fuel_rods
Returns the number of fuel rods in the bundle.

Returns **n_fuel_rods**

Return type int

`name`
Returns the name of the fuel bundle.

Returns **name**

Return type str

`radioactivity`
Returns the total radioactivity of the fuel bundle, in curies.

Returns **raduioactivity**

Return type float

`set_fuel_pin_length`(*x*)
Sets the length of all fuel pins in the bundle to x.

Returns **x**

Return type float

`set_gas_phase`(*phase*)
Sets the gas phase history of the fuel equal to phase.

Parameters **`phase`** (*`dataFrame`*) –

`set_solid_phase`(*phase*)
Sets the solid phase history of the fuel equal to phase.

Parameters **`phase`** (*`dataFrame`*) –

`solid_phase`
Returns the solid phase history associated with this fuel bundle.

Returns **solidPhase**

Return type dataFrame

3.1.4 fuel_segment module

Fuel segment Author: Valmor de Almeida dealmeidav@ornl.gov; vfda Sat Jun 27 14:46:49 EDT 2015

`class` `fuel_segment.`**`FuelSegment`**(*geometry=Series([], dtype: float64), species=[]*)
Bases: `object`

`__repr__`()
Used to pront the geometry of the fuel segment and the species that it consists of.

Returns **s**

Return type str

`__str__`()
Used to print the geometry of the fuel segment and the species that it consists of.

Returns **s**

Return type str

`geometry`
Returns the geometry of the fuel bundle (cylindrical, hexoganol, rectangular, etc).

Returns **geometry**

Return type str

get_attribute(*name*, *nuclide=None*, *series=None*)

Used to get stored fuel segment properties, either overall (as an average), or on a nuclide basis. "name" in this case refers to the attribute in question. At this point in time, series is not implemented and passing it to this function will result in an error. Possible attributes that may be retrieved with this function, as well as the name to pass to this function to retrieve them are: number of segments in the bundle (n-segments, always equal to 1), the id of the segment that makes up the bundle (segment-id), the volume of the fuel in the bundle (fuel-volume), the total volume of the segment (segment-volume), the diameter (fuel-diameter) and length (fuel-length) of the segment, the mass or mass density of the segment (mass or mass-cc, respectively), or the total or per-volume radioactivity, gamma radiation density or heat density of the fuel segment (radioactivity and radioactivityDens, gamma and gamma-dens, and heat and heat-dens, respectively).

Finally, density or total mass of a specific nuclide can be determined by passing a specific nuclide to the function, with a name value of mass or mass-cc.

Parameters

- **name** (*str*) –
- **nuclide** (*str*) –

Returns

Return type many types

get_geometry()

Returns the geometry of the fuel bundle (cylindrical, hexoganol, rectangular, etc).

Returns **geometry**

Return type str

get_specie(*name*)

Returns a specie named [name] from the list of species making up the fuel bundle. If no name is specified, this function will return None.

Parameters **name** (*str*) –

Returns **specie**

Return type obj

get_species()

Returns the species object which describes the composition of the fuel bundle. The species encapsulates all chemical species present in the fuel bundle.

Returns **species**

Return type object

specie

Returns a specie named [name] from the list of species making up the fuel bundle. If no name is specified, this function will return None.

Parameters **name** (*str*) –

Returns **specie**

Return type obj

species

Returns the species object which describes the composition of the fuel bundle. The species encapsulates all chemical species present in the fuel bundle.

Returns **species**

Return type object

3.1.5 fuelsegmentsgroups module

Author: Valmor de Almeida dealmeidav@ornl.gov; vfda

Fuel segment

VFdALib support classes

Sat Jun 27 14:46:49 EDT 2015

class `fuelsegmentsgroups.`**`FuelSegmentsGroups`**(*key=None*, *fuelSegments=None*)

Bases: `object`

Creates a dictionary of lists of fuel segment objects, with the keys typically being timestamps. Each fuel segment object has two data members, a *Pandas* Series for geometry spec and a panda DataFrame for property density.

`AddGroup`(*key*, *fuelSegments=None*)

Appends the dictionary with a new key and associated list of fuelSegments. If the specified key is already present in the dictionary, then the specified list of fuel segments will be appended to the list of fuel segments already associated with the specified key.

Parameters

- **`key`** (*`str`*) –
- **`fuelSegments`** (*`list`*) –

`GetAttribute`(*groupKey=None*, *attributeName=None*, *nuclideSymbol=None*, *nuclideSeries=None*)

Returns the average value of an attribute amongst all elements in a group (WARNING: keys with no values associated with them will lower this average!). If groupKey is not specified, the function will return the average attribute value of every fuel segment element in the entire dictionary. If attribute is not specified, the function call will fail. If the key value specified does not match any keys in the dictionary, the function will return a value of 0.

Parameters

- **`groupKey`** (*`str`*) –
- **`attributeName`** (*`str`*) –
- **`nuclideSymbol`** (*`str`*) –
- **`nuclideSeries`** (*`str`*) –

Returns **groupAttribute**

Return type float

`GetFuelSegments`(*groupKey=None*)

Returns a list of fuel segments associated with a specified groupkey. If no group key is specified, then all elements in the dictionary will be returned. If the specified group key does not exist, then the function will return an empty list.

Parameters **`groupKey`** (*`str`*) –

Returns **fuelSegments**

Return type list

`HasGroup`(*key*)

Checks if the specified key has a group of fuel segments associated with it.

Parameters **`key`** (*`str`*) –

Returns key

Return type str

RemoveFuelSegment (*groupKey*, *fuelSegment*)

Removes a fuel segment from a list associated with a specified group key. If the specified group key or fuel segment do not exist, the function will fail.

Parameters

- **groupKey** (*str*) –
- **fuelSegment** (*str*) –

Returns

Return type empty

3.1.6 fuelslug module

Author: Valmor de Almeida dealmeidav@ornl.gov; vfda

Fuel slug

ATTENTION:

This container requires two Phase() containers which are by definition histories. The history is not checked. Therefore any inconsistency will be propagated forward. A fuel slug has two solid phases: cladding and fuel. The user will decide how to best use the underlying history data in the Phase() container of each phase.

VFdALib support classes

Thu Dec 15 16:18:39 EST 2016

class fuelslug.**FuelSlug** (*specs=Series([], dtype: float64), fuelPhase= **Phase()**: time unit: s *quantities*: None *species*: None *history* #time_stamp=1 *history end* @0.0 Series([], Name: 0.0, dtype: float64), claddingPhase= **Phase()**: time unit: s *quantities*: None *species*: None *history* #time_stamp=1 *history end* @0.0 Series([], Name: 0.0, dtype: float64)*)

Bases: object

GetAttribute (*name*, *phase=None*, *symbol=None*, *series=None*)

Returns the value of the specified attribute. Any attribute that is specified in class construction can be retrieved using this function. The attribute may also be retrived from a speciefic phase, a specific nuclide OR a specific series.

Parameters

- **name** (*str*) –
- **phase** (*str*) –
- **symbol** (*str*) –
- **series** (*str*) –

Returns attribute

Return type int or float

GetCladdingPhase ()

Returns the phase history of the cladding.

Returns **claddingPhase**

Return type dataFrame

GetFuelPhase()
Returns the phase history of the solid fuel.

Returns **fuelPhase**

Return type dataFrame

GetSpecs()
Returns the species associated with this fuel slug.

Returns **specs**

Return type str

ReduceCladdingVolume(*dissolvedVolume*)
Reduces the amount of cladding in the slug by dissolvedvolume. This will also update the dimensions of the cladding walls and end caps; volume will be taken from all sections equally such that the relative dimensions stay the same.

Parameters **dissolvedVolume** (*float*) –

ReduceFuelVolume(*dissolvedVolume*)
Reduces the amount of fuel in the slug by dissolvedVolume. This will also update the dimensions of the fuel slug, mainly the thickness of each fuel layers.

Parameters **dissolvedVolume** (*float*) –

claddingPhase
Returns the phase history of the cladding.

Returns **claddingPhase**

Return type dataFrame

fuelPhase
Returns the phase history of the solid fuel.

Returns **fuelPhase**

Return type dataFrame

specs
Returns the species associated with this fuel slug.

Returns **specs**

Return type str

3.1.7 nuclides module

Author: Valmor de Almeida dealmeidav@ornl.gov; vfda

Nuclides container. The purpose of the this container is to store and query a table of nuclides. Typically the table is filled in with data from an ORIGEN calculation or some other fission/transmutation code.

VFdALib support classes

Sat Jun 27 14:46:49 EDT 2015

class nuclides.**Nuclides**(*propertyDensities=Empty DataFrame Columns: [] Index: []*)
Bases: object

GetAttribute(*name*, *symbol=None*, *series=None*)

3.2 periodictable module

Properties of the chemical elements.

Each chemical element is represented as an object instance. Physicochemical and descriptive properties of the elements are stored as instance attributes.

Author Christoph Gohlke

Version 2015.01.29

Radiochemical data (isotopes) has been added to this table (2015-2016) Origin: http://www.radiochemistry.org/ Valmor F. de Almeida: dealmeidavf@gmail.com; dealmeidav@ornl.gov

3.2.1 Requirements

- CPython 2.7 or 3.4

References

1. http://physics.nist.gov/PhysRefData/Compositions/
2. http://physics.nist.gov/PhysRefData/IonEnergy/tblNew.html
3. http://en.wikipedia.org/wiki/%(element.name)s
4. http://www.miranda.org/~jkominek/elements/elements.db

Examples

```
>>> from elements import ELEMENTS
>>> len(ELEMENTS)
109
>>> str(ELEMENTS[109])
'Meitnerium'
>>> ele = ELEMENTS['C']
>>> ele.number, ele.symbol, ele.name, ele.eleconfig
(6, 'C', 'Carbon', '[He] 2s2 2p2')
>>> ele.eleconfig_dict
{(1, 's'): 2, (2, 'p'): 2, (2, 's'): 2}
>>> sum(ele.mass for ele in ELEMENTS)
14659.1115599
>>> for ele in ELEMENTS:
...     ele.validate()
...     ele = eval(repr(ele))
```

3.3 phase module

Phase *history* container. When you think of a phase value, think of that value at a specific point in time. This container holds the historic data of a phase; its species and quantities. This implementation treats access of time stamps within a tolerance. All searches for time stamped values are subjected to an approximation of the time stamp to avoid storing values too close to each other in time, and/or to return the closest value in time searched or no value if none can be found according to the tolerance.

3.3.1 Background

TODO: ATTENTION: The species (list of Specie) AND quantities (list of Quantity) data members have ARBITRARY density values either at an arbitrary point in the history or at no point in the history. This needs to be removed in the future to avoid confusion.

To obtain history values, associated to the phase, at a particular point in time, use the GetValue() method to access the history data frame (pandas) via columns and rows. ALERT: The corresponding values in species and quantities are OVERRIDEN and NOT to be used through the phase interface.

Author: Valmor F. de Almeida dealmeidav@ornl.gov; vfda Sat Sep 5 01:26:53 EDT 2015

Cortix: a program for system-level modules coupling, execution, and analysis.

class phase.**Phase**(*time_stamp=None*, *time_unit=None*, *species=None*, *quantities=None*)

Bases: `object`

Phase *history* container. A *Phase* consists of *Species* and *Quantities* varying with time. This container is meant to reproduce the basic idea of a material phase.

AddQuantity(*newQuant*)

Adds a new quantity object to the dataframe. See quantity.py for more details on the quantity class.

Parameters **newQuant** (*object*) –

AddRow(*try_time_stamp*, *row_values*)

Adds a row to the dataframe, with a timestamp of try_time_stamp and row values equal to row_values. Take care that the dimensions and order of the data matches up!

Parameters

- **try_time_stamp** (*float*) –
- **row_values** (*list*) –

AddSpecie(*new_specie*)

Adds a new specie object to the phase history. See species.py for more details on the specie class.

Parameters **new_specie** (*obj*) –

ClearHistory(*value=0.0*)

Set species and quantities of history to a given value (default to zero value), all time stamps are preserved.

Parameters **value** (*float*) –

GetActors()

Returns a list of all the actors in the phase history.

Returns **list(self.__phase.colums)**

Return type list

GetColumn(*actor*)

Returns an entire column of data. A column is the entire history of data associated with a specific actor.

Parameters **actor** (*str*) –

Returns **list(self.__phase.loc[**

Return type , actor]): list

GetQuantities()
Returns the list of *Quantities*. The values in each *Quantity* are synchronized with the *Phase* data frame.

Returns quantities

Return type list

GetQuantity(*name*)
Returns the quantity evaluated at the last time step of the phase history. This also updates the value of the quantity object. If the quantity name does not exist the return is None.

Parameters **name** (*str*) –

GetRow(*try_time_stamp=None*)
Returns an entire row of the phase dataframe. A row is a series of values that are all at the same time stamp.

Parameters **try_time_stamp** (*float*) –

Returns list(self.__phase.loc[time_stamp,

Return type]): list

GetSpecie(*name*)
Returns the species specified by name if it exists, or none if it doesn't.

Parameters **name** (*str*) –

Returns specie

Return type str

GetSpecies()
Returns every single species in the phase history.

Returns species

Return type list

GetTimeStamps()
Returns a list of all the time stamps in the phase history.

Returns timeStamps

Return type list

GetValue(*actor*, *try_time_stamp=None*)
Deprecated: use get_value()

ResetHistory(*try_time_stamp=None*, *value=None*)
Set species and quantities of history to a given value (default to zero value) only one time stamp is preserved (default to last time stamp).

Parameters

- **try_time_stamp** (*float*) –
- **value** (*float*) –

ScaleRow(*try_time_stamp*, *value*)
Multiplies all of the data in a row (except time stamp) by a scalar value.

Parameters

- **try_time_stamp** (*float*) –
- **value** (*float*) –

SetSpecieId(*name*, *val*)
Sets the flag of a specie "name" equal to val.

Parameters

- **name** (*str*) –
- **val** (*int*) –

SetValue(*actor*, *value*, *try_time_stamp=None*)
For the record: old def SetValue(self, time_stamp, actor, value):

Parameters

- **actor** (*str*) –
- **value** (*float*) –
- **try_time_stamp** (*float*) –

WriteHTML(*fileName*)
Convert the *Phase* container into an HTML file.

Parameters **fileName** (*str*) –

__init__(*time_stamp=None*, *time_unit=None*, *species=None*, *quantities=None*)
Sometimes an empty Phase object is created by user code. This case needs adequate logic for None types. Note on usage: when passing quantities, do set the value argument explicitly to help define the type and avoid SetValue() errors with Pandas. This is to be investigated later. Also, the usage of a DataFrame needs to be re-evaluated. Maybe better to use a Quantity object and a Specie object with a Pandas Series history as a value to avoid the existance of a value in Quantity and a value in Phase that are not in sync.

get_quantity(*name*, *try_time_stamp=None*)
New version. Get the quantity *name* at a point in time closest to *try_time_stamp* up to a tolerance. If no time stamp is passed, the whole history is returned.

Parameters

- **name** (*str*) –
- **try_time_stamp** (*float, int or None*) – Time stamp of desired quantity value. Default: None returns the whole quantity history.

Returns **quant.value**

Return type float or int or other

get_quantity_history(*name*)
Create a Quantity *name* history. This will create a fully qualified Quantity object and return to the caller. The function is typically needed for data output to a file through *pickle*. Since the value attribute of a quantity can be any data structure, a time-series is built on the fly and stored in the value attribute. In addition the time unit is added to the final return value as a tuple.

Parameters **name** (*str*) –

Returns **quant_history**

Return type tuple(*Quantity*,str)

get_value(*actor*, *try_time_stamp=None*)
Returns the value associated with a specified actor at a specified time stamp.

Parameters

- **actor** (*str*) –

- **try_time_stamp** (*float*) –

Returns self.__phase.loc[time_stamp, actor]

Return type float

has_time_stamp (*try_time_stamp*)
Checks to see if try_time_stamp exists in the phase history.

Parameters **try_time_stamp** –

quantities
Returns the list of *Quantities*. The values in each *Quantity* are synchronized with the *Phase* data frame.

Returns quantities

Return type list

set_value (*actor*, *value*, *try_time_stamp=None*)
New version. Discontinue using SetValue()

species
Returns every single species in the phase history.

Returns species

Return type list

timeStamps
Returns a list of all the time stamps in the phase history.

Returns timeStamps

Return type list

time_stamps
Get all time stamps in the index of the data frame.

Returns time_stamps

Return type list

time_unit
Returns the time unit of the *Phase*.

Returns time_unit

Return type str

3.4 phase_new module

Phase *history* container. When you think of a phase value, think of that value at a specific point in time. This container holds the historic data of a phase; its species and quantities. This implementation treats access of time stamps within a tolerance. All searches for time stamped values are subjected to an approximation of the time stamp to avoid storing values too close to each other in time, and/or to return the closest value in time searched or no value if none can be found according to the tolerance.

3.4.1 Background

TODO: ATTENTION: The species (list of Species) AND quantities (list of Quantity) data members have ARBITRARY density values either at an arbitrary point in the history or at no point in the history. This needs to be removed in the future to avoid confusion.

To obtain history values, associated to the phase, at a particular point in time, use the GetValue() method to access the history data frame (pandas) via columns and rows. ALERT: The corresponding values in species and quantities are OVERRIDEN and NOT to be used through the phase interface.

Author: Valmor F. de Almeida dealmeidav@ornl.gov; vfda Sat Sep 5 01:26:53 EDT 2015

Cortix: a program for system-level modules coupling, execution, and analysis.

class `phase_new.`**`PhaseNew`**(*name=None*, *time_stamp=None*, *time_unit=None*, *species=None*, *quantities=None*)

Bases: `object`

Phase *history* container. A *Phase* consists of *Species* and *Quantities* varying with time. This container is meant to reproduce the basic idea of a material phase.

`ClearHistory`(*value=0.0*)

Set species and quantities of history to a given value (default to zero value), all time stamps are preserved.

Parameters **`value`** (*`float`*) –

`GetQuantities`()

Returns the list of *Quantities*. The values in each *Quantity* are synchronized with the *Phase* data frame.

Returns **quantities**

Return type list

`ResetHistory`(*try_time_stamp=None*, *value=None*)

Set species and quantities of history to a given value (default to zero value) only one time stamp is preserved (default to last time stamp).

Parameters

- **`try_time_stamp`** (*`float`*) –
- **`value`** (*`float`*) –

`__init__`(*name=None*, *time_stamp=None*, *time_unit=None*, *species=None*, *quantities=None*)

Sometimes an empty Phase object is created by user code. This case needs adequate logic for None types. Note on usage: when passing quantities, do set the value argument explicitly to help define the type and avoid set_value() errors with Pandas. This is to be investigated later. Also, the usage of a DataFrame needs to be re-evaluated. Maybe better to use a Quantity object and a Species object with a Pandas Series history as a value to avoid the existance of a value in Quantity and a value in Phase that are not in sync.

`actors`

Returns a list of names of all the actors in the phase history.

Returns **list(self.__df.colums)**

Return type list

`add_quantity`(*new_quant*)

Adds a new quantity object to the dataframe. See quantity.py for more details on the quantity class.

Parameters **`new_quant`** (*`object`*) –

`add_row`(*try_time_stamp*, *row_values*)

Adds a row to the *DataFrame*, with a *timestamp* equal to *try_time_stamp* and row values equal to *row_values*. The length of *row_values* must match the number of columns in the data frame.

Parameters

- **`try_time_stamp`** (*`float`*) –
- **`row_values`** (*`list`*) –

`add_single_species`(*new_species*)

Adds a new specie object to the phase history. See species.py for more details on the Species class.

Parameters **`new_species`** (*`obj`*) –

`df`

Die hard access.

`get_column`(*actor*)

Returns an entire column of data. A column is the entire history of data associated with a specific actor.

Parameters **`actor`** (*`str`*) –

Returns **list(self.__df.loc[**

Return type , actor]): list

`get_quantity`(*name*, *try_time_stamp=None*)

Get the quantity *name* at a point in time closest to *try_time_stamp* up to a tolerance. If no time stamp is passed, the value at the last time stamp is returned.

Parameters

- **`name`** (*`str`*) –
- **`try_time_stamp`** (*`float, int or None`*) – Time stamp of desired quantity value. Default: None, returns the value at the last time stamp.

Returns **quant.value**

Return type float or int or other

`get_quantity_history`(*name*)

Create a Quantity *name* history. This will create a fully qualified Quantity object and return to the caller. The function is typically needed for data output to a file through *pickle*. Since the value attribute of a quantity can be any data structure, a time-series is built on the fly and stored in the value attribute. In addition the time unit is added to the final return value as a tuple.

Parameters **`name`** (*`str`*) –

Returns **quant_history**

Return type tuple(*Quantity*,str)

`get_row`(*try_time_stamp=None*)

Returns an entire row of the phase dataframe. A row is a series of values that are all at the same time stamp.

Parameters **`try_time_stamp`** (*`float`*) –

Returns **list(self.__df.loc[time_stamp,**

Return type]): list

`get_species`(*name*)

Returns the species specified by name if it exists, or None if it doesn't.

Parameters **`name`** (*`str`*) –

Returns **specie**

Return type str

get_species_concentration(*name*, *try_time_stamp=None*)
Returns the species concentration at *try_time_stamp*.

Parameters

- **name** (*str*) –
- **try_time_time_stamp** (*float*) –

Returns concentration

Return type float

get_value(*actor*, *try_time_stamp=None*)
Returns the value associated with a specified actor at a specified time stamp.

Parameters

- **actor** (*str*) –
- **try_time_stamp** (*float*) – Default is None which returns the last time stamp.

Returns self.__df.loc[time_stamp, actor]

Return type any

has_time_stamp(*try_time_stamp*)
Checks to see if try_time_stamp exists in the phase history.

Parameters **try_time_stamp** –

plot(*name='phase-plot-name'*, *time_unit='s'*, *legend=None*, *nrows=2*, *ncols=2*, *dpi=200*)

plot_species(*name, scaling=[1.0, 1.0], title=None, xlabel='Time [s]', ylabel='y', legend='no-legend', filename_tag=None, figsize=[6, 5], dpi=100*)

quantities
Returns the list of *Quantities*. The values in each *Quantity* are synchronized with the *Phase* data frame.

Returns quantities

Return type list

scale_row(*try_time_stamp*, *value*)
Multiplies all of the data in a row (except time stamp) by a scalar value.

Parameters

- **try_time_stamp** (*float*) –
- **value** (*float*) –

set_species_id(*name*, *val*)
Sets the flag of a species "name" equal to val.

Parameters

- **name** (*str*) –
- **val** (*int*) –

set_value(*actor*, *value*, *try_time_stamp=None*)

species
Returns every single species in the phase history.

Returns species

Return type list

time_stamps
Get all time stamps in the index of the data frame.

Returns time_stamps

Return type list

time_unit
Returns the time unit of the *Phase*.

Returns time_unit

Return type str

write_html (*fileName*)
Convert the *Phase* container into an HTML file.

Parameters **fileName** (*str*) –

3.5 quantity module

class quantity.**Quantity** (*name='null-quantity-name'*, *formalName='null-quantity-formal-name'*, *formal_name='null-quantity-formal-name'*, *value=0.0*, *unit='null-quantity-unit'*)
Bases: object

todo: this probably should not have a "value" for the same reason as Species. this needs some thinking.

well not so fast. This can be used to build a quantity with anything as a value. For instance a history of the quantity as a time series.

GetFormalName ()
Returns the formal name of the quantity.

Returns formalName

Return type str

GetUnit ()
Returns the units of the quantity.

Returns unit

Return type str

GetValue ()
Gets the numerical value of the quantity.

Returns value

Return type any type

SetFormalName (*fn*)
Sets the formal name of the property to fn.

Parameters **fn** (*str*) –

SetName (*n*)
Sets the name of the quantity in question to n.

Parameters **n** (*str*) –

SetUnit (*f*)
Sets the units of the quantity to f (for example, density would be in units of g/cc.

Parameters **f** (*str*) –

SetValue (*v*)
Sets the numerical value of the quantity to v.

Parameters **v** (*float*) –

__repr__ ()
Used to print the data stored by the quantity class. Will print out name, formal name, the value of the quantity and its unit.

Returns **s**

Return type str

__str__ ()
Used to print the data stored by the quantity class. Will print out name, formal name, the value of the quantity and its unit.

Returns **s**

Return type str

formalName
Returns the formal name of the quantity.

Returns **formalName**

Return type str

formal_name
Returns the formal name of the quantity.

Returns **formalName**

Return type str

get_name ()
Returns the name of the quantity.

Returns **name**

Return type str

name
Returns the name of the quantity.

Returns **name**

Return type str

plot (*x_scaling=1*, *y_scaling=1*, *title=None*, *x_label='x'*, *y_label=None*, *file_name=None*, *same_axis=True*, *dpi=300*)
This will support a few possibities for data storage in the self.__value member.

Pandas Series. If self.__value is a Pandas Series, plot against the index. However the type stored in the Series matter. Suppose it is a series of a *numpy* array. This must be of the same rank for every entry. This plot method assumes it is an iterable type of the same length for every entry in the series. A plot of all elements in the type against the index of the series will be made. The plot may have all elements in one axis or each element in its own axis.

unit
Returns the units of the quantity.

Returns unit

Return type str

value

Gets the numerical value of the quantity.

Returns value

Return type any type

3.6 specie module

Author: Valmor de Almeida dealmeidav@ornl.gov; vfda

This Specie class is to be used with other classes in plant-level process modules.

NB: Species is always used either in singular or plural cases, the class named here reflects one species. If many species are used in an external context, the species object name can be used without conflict.

For unit testing do at the linux command prompt: python specie.py

NB: The Specie() class encapsulates either the molecular or empirical chemical formula of a compound. This is done as follows. Say MAO2 is either a molecular or empirical chemical formula of a ficticious compound denoting minor actinides dioxide. The list of atoms is given as follows:

[‘0.49*Np-237’, ‘0.42*Am-241’, ‘0.08*Am-243’, ‘0.01*Cm-244’, ‘2.0*O-16’]

note the MA forming nuclides add to 1 = 0.49 + 0.42 + 0.08 + 0.01. Therefore the number of atoms in this compound is 3. 1 MA “atom” and 2 O. Note that the total number of “atoms” is obtained by summing all multipliers: 0.49 + 0.42 + 0.08 + 0.01 + 2.0. The nuclide is indicated by the element symbol followed by a dash and the atomic mass number. Here the number of nuclide types is 5 (self._nNuclideTypes).

The numbers preceeding the nuclide symbol before the * will be referred to as multipliers. The sum of the multipliers will add to the number of “atoms” in the formula. WARNING: a multiplier could be in the format 0.00e-00. In this case a hiphen may appear twice, e.g.: 1.549e-09*U-233

Other forms can be used for common true species

[‘Np-237’, ‘2.0*O-16’] or [‘Np-237’, ‘O-16’, ‘O-16’] or [‘2*H’, ‘O’] or [‘H’, ‘O’, ‘H’] etc…

This code will calculate the molar mass of any species with a given valid atom list using a provided periodic table of chemical elements. The user can also reset the value of the molar mass with a setter method.

Sat May 9 21:40:48 EDT 2015 created; vfda

`class specie.Specie(name='null', formula_name='null', phase='null', atoms=[], molarCC=0.0, massCC=0.0, flag=None)`

Bases: object

todo: phase should not be here; concentrations should not be here only molar quantities should be here see the Phase container

GetAtoms()

GetFlag()

Returns the flag associated with the species.

Returns flag

Return type str

GetFormula()

Returns the molecular or empirical formula of the species. It is usually a list, for example, of the form ['2*H', 'O'].

Returns **formula**

Return type list

GetFormulaName()

Returns the formulaic name of the compound. For example, "Dihydrogen monoxide".

Returns **self.__formula_name**

Return type str

GetMassCC()

Returns the numerical value of the mass density of the species (mass/volume).

Returns **massCC**

Return type float

GetMassCCUnit()

Returns the unit used to measure the mass density of the species.

Returns **massCCUnit**

Return type str

GetMolarCC()

Returns the numerical value for the number (molar) density of the species (moles/volume).

Returns **molarCC**

Return type float

GetMolarCCUnit()

Returns the unit used to measure molar density of the species.

Returns **molarCCUnit**

Return type str

GetMolarGammaPwr()

Returns the amount of gamma radiation produced per mole of this species (measured in units of power).

Returns **molarGammaPwr**

Return type float

GetMolarGammaPwrUnit()

Returns the unit used to measure the amount of gamma radiation produced per mole of this species.

Returns **molarGammaPwrUnit**

Return type str

GetMolarHeatPwr()

Returns the amount of heat generated per mole of this species.

Returns **molarHeatPwr**

Return type float

GetMolarHeatPwrUnit()

Returns the unit used to measure the amount of heat generated per mole of this species.

Returns **molarHeatPwrUnit**

Return type str

GetMolarMass()

Returns the numerical value for the molar mass of the species. Units are given by molarMassUnit.

Returns **molarMass**

Return type float

GetMolarMassUnit()

Returns the unit used to measure the molar mass of the species.

Returns **molarMassUnit**

Return type str

GetMolarRadioactivity()

Returns the numerical value for molar radioactivity of the species.

Returns **molarRadioactivity**

Return type float

GetMolarRadioactivityFractions()

Returns a list of numbers that speciefies the % of molar reactivity that comes from each type of atom in the species. For example, a molarRadioactivityFraction of [0.65, 0.35] for water means that 65% of the molar radioactivity comes from the hydrogen atoms and 35% comes from the oxygen atom.

Returns **molarRadioactivityFractions**

Return type list

GetMolarRadioactivityUnit()

Returns the unit used to measure molar radioactivity.

Returns **molarRadioactivityUnit**

Return type str

GetNAtoms()

Returns the total number of atoms comprising the species. For example, water is comprised of three atoms.

Returns **nAtoms**

Return type int

GetNNuclideTypes()

Returns the number of different types of atoms comprising the species. For example, water is composed of two different types of atoms, hydrogen and oxygen.

Returns **nNuclideTypes**

Return type int

GetName()

Returns the empirical name of the species. For example, “water”.

Returns **name**

Return type str

GetPhase()

Returns the phase history of the species.

Returns **phase**

Return type dataFrame

SetAtoms(*atoms*)

SetFlag(*f*)
Sets the flag associated with the species to f.

Parameters **f** (`str`) –

SetFormula(*atoms*)
Sets the species' formula equal to atoms. Will automatically update the molar mass of the species, and will also fail if atoms is not a list of strings.

Parameters **atoms** (`list`) –

SetFormulaName(*f*)
Sets the formulaic name to f.

Returns **self.__formula_name**

Return type str

SetMassCC(*v*)
Sets the numerical value of the mass density equal to v.

Parameters **v** (`float`) –

SetMassCCUnit(*v*)
Sets the units used to measure mass density to v.

Parameters **v** (`str`) –

SetMolarCC(*v*)
Sets the numerical value for the molar density of the species to v.

Parameters **v** (`float`) –

SetMolarCCUnit(*v*)
Sets the unit used to measure the molar density of the species to v.

Parameters **v** (`str`) –

SetMolarGammaPwr(*v*)
Sets the amount of gamma radiation produced per mole of this species to v.

Parameters **v** (`float`) –

SetMolarGammaPwrUnit(*v*)
Sets the unit used to measure the amount of gamma radiation produced per mole of this species to v.

Parameters **v** (`str`) –

SetMolarHeatPwr(*v*)
Sets the amount of heat generated per mole of this species to v.

Parameters **v** (`float`) –

SetMolarHeatPwrUnit(*v*)
Sets the unit used to measure the amount of heat generated per mole of this species to v.

Parameters **v** (`str`) –

SetMolarMass(*v*)
Sets the molar mass of the species equal to v.

Parameters **v** (`float`) –

SetMolarMassUnit(*v*)
Sets the unit used to measure the molar mass of the species to v.

Parameters **v** (*str*) –

SetMolarRadioactivity(*v*)

Sets the molar radioactivity of the species equal to v.

Parameters **v** (*float*) –

SetMolarRadioactivityFractions(*fracs*)

Sets molarRadioactivityFractions equal to fracs. Fracs must be a list of floatswith the same length as there are different atoms in the species, or the function call will fail. (e.g. self._atoms and fracs must be of the same length). Take care to ensure that the elements of fracs match with the elements of self._atoms! (65% is in the same position in fracs as hydrogen is in self._atoms, following the above example).

Parameters **fracs** (*list*) –

SetMolarRadioactivityUnit(*v*)

Sets the unit used to measure molar radioactivity to v.

Parameters **v** (*str*) –

SetName(*n*)

Sets the empirical name of the species to n.

Parameters **n** (*str*) –

SetPhase(*p*)

Sets the phase history to p.

Parameters **p** (*dataFrame*) –

atoms

flag

Returns the flag associated with the species.

Returns **flag**

Return type str

formula

Returns the molecular or empirical formula of the species. It is usually a list, for example, of the form ['2*H', 'O'].

Returns **formula**

Return type list

formula_name

Returns the formulaic name of the compound. For example, "Dihydrogen monoxide".

Returns **self.__formula_name**

Return type str

massCC

Returns the numerical value of the mass density of the species (mass/volume).

Returns **massCC**

Return type float

massCCUnit

Returns the unit used to measure the mass density of the species.

Returns **massCCUnit**

Return type str

`molarCC`
Returns the numerical value for the number (molar) density of the species (moles/volume).

Returns molarCC

Return type float

`molarCCUnit`
Returns the unit used to measure molar density of the species.

Returns molarCCUnit

Return type str

`molarGammaPwr`
Returns the amount of gamma radiation produced per mole of this species (measured in units of power).

Returns molarGammaPwr

Return type float

`molarGammaPwrUnit`
Returns the unit used to measure the amount of gamma radiation produced per mole of this species.

Returns molarGammaPwrUnit

Return type str

`molarHeatPwr`
Returns the amount of heat generated per mole of this species.

Returns molarHeatPwr

Return type float

`molarHeatPwrUnit`
Returns the unit used to measure the amount of heat generated per mole of this species.

Returns molarHeatPwrUnit

Return type str

`molarMass`
Returns the numerical value for the molar mass of the species. Units are given by molarMassUnit.

Returns molarMass

Return type float

`molarMassUnit`
Returns the unit used to measure the molar mass of the species.

Returns molarMassUnit

Return type str

`molarRadioactivity`
Returns the numerical value for molar radioactivity of the species.

Returns molarRadioactivity

Return type float

`molarRadioactivityFractions`
Returns a list of numbers that speciefies the % of molar reactivity that comes from each type of atom in the species. For example, a molarRadioactivityFraction of [0.65, 0.35] for water means that 65% of the molar radioactivity comes from the hydrogen atoms and 35% comes from the oxygen atom.

Returns molarRadioactivityFractions

Return type list

molarRadioactivityUnit
Returns the unit used to measure molar radioactivity.

Returns molarRadioactivityUnit

Return type str

nAtoms
Returns the total number of atoms comprising the species. For example, water is comprised of three atoms.

Returns nAtoms

Return type int

nNuclideTypes
Returns the number of different types of atoms comprising the species. For example, water is composed of two different types of atoms, hydrogen and oxygen.

Returns nNuclideTypes

Return type int

name
Returns the empirical name of the species. For example, "water".

Returns name

Return type str

phase
Returns the phase history of the species.

Returns phase

Return type dataFrame

3.7 species module

class species.**Species**(*name='null-species-name'*, *formula_name='null-species-formula-name'*, *atoms=[ ]*, *flag='null-species-flag'*, *info=None*)

Bases: object

All SI units (kg,s,K,Pa,J,W).

The Specie() class encapsulates either the molecular or empirical chemical formula of a compound. This is done as follows. Say MAO2 is either a molecular or empirical chemical formula of a ficticious compound denoting minor actinides dioxide. The list of atoms is given as follows:

['0.49*Np-237', '0.42*Am-241', '0.08*Am-243', '0.01*Cm-244', '2.0*O-16']

note the MA forming nuclides add to 1 = 0.49 + 0.42 + 0.08 + 0.01. Therefore the number of atoms in this compound is 3. 1 MA "ficticious" atom and 2 O. Note that the total number of "atoms" is obtained by summing all multipliers: 0.49 + 0.42 + 0.08 + 0.01 + 2.0. The nuclide is indicated by the element symbol followed by a dash and the atomic mass number. Here the number of nuclide types is 5 (self.num_nuclide_types).

The numbers preceeding the nuclide symbol before the * will be referred to as multipliers. The sum of the multipliers will add to the number of "atoms" in the formula. WARNING: a multiplier could be in the format 0.00e-00. In this case a hiphen may appear twice, e.g.: 1.549e-09*U-233

Other forms can be used for common true species

[‘Np-237’, ‘2.0*O-16’] or [‘Np-237’, ‘O-16’, ‘O-16’] or [‘2*H’, ‘O’] or [‘H’, ‘O’, ‘H’] etc…

This code will calculate the molar mass of any species with a given valid atom list using a provided periodic table of chemical elements. The user can also reset the value of the molar mass with a setter method.

`reorder_formula``()`

Takes a list of atoms for a molecular or empirical formula and places it in order of decreasing magnitude of stoichiometric coefficient. For example, [O, 2*H] will be returned as [2*H, O]. This is used for printing purposes. The internal order will not change.

Returns **atoms2**

Return type list

`update_molar_mass``()`

Updates the molar mass of the species after the molecular formula has been changed.

3.8 stream module

Author: Valmor F. de Almeida dealmeidav@ornl.gov; vfda

Stream container

VFdALib support classes

Sat Aug 15 17:24:02 EDT 2015

`class` `stream.`**`Stream`**`(`*timeStamp*, *species=None*, *quantities=None*, *values=0.0*`)`

Bases: `object`

`GetActors``()`

Returns the actors present in the stream of data.

Returns **list(self.stream.columns)**

Return type list

`GetQuantities``()`

Returns all the quantities given by the stream.

Returns **self.quantities**

Return type list

`GetQuantity``(`*name*`)`

Returns the specified quantity called “name” from the stream, or none if the specified name does not exist.

Parameters **`name`** (*`str`*) –

Returns **quant**

Return type float

`GetRow``(`*timeStamp=None*`)`

Returns an entire row of data from the stream. A row of data is all the data in a dataframe at a specified time stamp, given by timeStamp. If timeStamp is not specified, this function will return the entire stream dataframe.

Parameters **`timeStamp`** (*`float`*) –

Returns

- **self.stream.loc[self.timestamp,** (*]) or self.stream.loc[timeStamp, :]):*)
- *list*

GetSpecie(*name*)
Returns a specie named "name" from the stream.

Parameters **name** (*str*) –

Returns **specie**

Return type obj

GetSpecies()
Returns a list of all species in the stream.

Returns **self.species**

Return type list

GetTimeStamp()
Returns the time stamp of the stream.

Returns **self.timeStamp**

Return type float

GetValue(*actor*, *timeStamp=None*)
Returns the value associated with a specified "actor" at a specified "timeStamp". If no timeStamp is specified, then the function will return all values associated with the specified actor at all time stamps.

Parameters

- **actor** (*str*) –
- **timeStamp** (*float*) –

Returns

- *self.stream.loc[self.timeStamp, actor] or self.stream.loc[timeStamp,*
- **actor]** (*list or float, respectively.*)

SetSpecieId(*name*, *val*)
Sets the numerical id of the specie of name "name" to val.

Parameters

- **name** (*str*) –
- **val** (*int*) –

SetValue(*actor*, *value=None*, *timeStamp=None*)
Sets the value associated with a specified actor at a specified timeStamp to "value". If no value is specified, the value will default to 0.0. If no timeStamp is specified, it will set all values associated with actor to the specified value (or 0.0 if value = None).

Parameters

- **actor** (*str*) –
- **value** (*float*) –
- **timeStamp** (*float*) –

PYTHON MODULE INDEX

INDEX