wnutils Documentation

Release 2.2.3

Clemson University

Aug 10, 2021

Contents

CHAPTER 1

Contents

1.1 Overview

wnutils is a python package for reading and plotting webnucleo files.

1.1.1 Installation

Install from PyPI with pip by typing in your favorite terminal:

```
$ pip install wnutils
```

1.1.2 Authors

- Bradley S. Meyer <mbradle@clemson.edu>
- Norberto Davila <ndavila@clemson.edu>

1.1.3 Contribute

- Issue Tracker: https://github.com/mbradle/wnutils/issues/
- Source Code: https://github.com/mbradle/wnutils/

1.1.4 License

1.1.5 Usage

The best way to get started using wnutils is to follow the tutorials. You may also want to visit our galleries or check out our Jupyter Notebooks or our Code Samples.

1.2 Changelog

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

1.2.1 Version 2.2.3

Fix:

- Unneeded print method introduced in 2.2.0 removed since it caused documentation errors.

1.2.2 Version 2.2.2

Fix:

- The tutorial data are now downloaded from OSF.

1.2.3 Version 2.2.1

Fix:

- An error in assigning the atomic number of species starting with 'n' that was introduced in 2.2.0 has been fixed.

1.2.4 Version 2.2.0

New:

- It is now possible to print out newly created XML files to the standard output.
- The link to webnucleo has been updated.

Fix:

- An ambiguity in retrieving atomic number, mass number, and state data from a nuclide with name 'n' (that is, neutron or nitrogen) has been fixed.

1.2.5 Version 2.1.0

New:

- It is now possible to parse XML files with XInclude with wnutils.

1.2.6 Version 2.0.1

Fix:

- A typo in a warning in the get_zone_data() routine has been fixed.

1.2.7 Version 2.0.0

New:

- It is now possible to add fixed or time-dependent curves to the XML nucleon number and abundance chain movies. The data are added via an array of tuples, which is a backwards incompatible change from the capability added in version 1.10.0.
- The method to return chain abundances has been promoted to the API.

Fix:

- The XML method to return all abundances in zones now returns the abundances for all species.

1.2.8 Version 1.10.2

Fix:

- An error introduced in 1.10.1 in reading zone data has been fixed.

1.2.9 Version 1.10.1

Fix:

- Parser now treats the nuclide name attribute in zone data as optional, as expected from the schema.

1.2.10 Version 1.10.0

New:

- It is now possible to add extra curves to the XML nucleon number and abundance chain movies.
- The animation tutorial has been updated to include information on the abundance chain movie and on adding extra curves.

Fix:

- Parser now treats the reaction source as optional in the input XML file, as expected from the schema.
- The assignment of mass number for abundance chain movies has been fixed.

1.2.11 Version 1.9.0

New:

- A method to create an abundance chain movie has been added.
- Movie routines now return the animation, and the movie file name is now an optional keyword.

Fix:

- Mis-assigments of spin and mass excess in the H5 class have been fixed.

1.2.12 Version 1.8.0

New:

- A method to retrieve the root type of an Xml object has been added.
- A method to retrieve zone data has been added.
- A method to retrieve Z, A, and state label from a nuclide name has been added.
- A link to code samples has been added.

1.2.13 Version 1.7.1

New:

- A link to the tutorials in Jupyter notebook form has been added.

Fix:

- Some tutorial typos have been fixed.

1.2.14 Version 1.7.0

New:

- A new class allows the user to create webnucleo XML and write that XML to a file.

Fix:

- The reaction rate calculator now computes the reaction rate from rate table data by not extrapolating from lowest and highest temperature values. This means that, for temperatures below the lowest temperature in the table, the rate is computed at the lowest table temperature. Similarly, for temperatures above the highest temperature in the table, the rate is computed at the highest table temperature. This treatment is in agreement with how libnucnet computes rates from rate tables.

1.2.15 Version 1.6.0

New:

- A method to validate the XML against libnucnet schemas has been added.

Fix:

- State data is now parsed from XML correctly.
- An error in creating IUPAC element names has been fixed.

1.2.16 Version 1.5.2

Fix:

- The license attribute string has been shortened.
- A typo in the tutorials has been fixed.

1.2.17 Version 1.5.1

Fix:

- An error in constructing species names has been fixed.

1.2.18 Version 1.5.0

New:

- State labels are now rendered as subscripts in species latex names.

1.2.19 Version 1.4.4

Fix:

- The markdown indicator in setup.py has been fixed.

1.2.20 Version 1.4.3

Fix:

- The XPath expressions in some routines have been fixed.

1.2.21 Version 1.4.2

Fix:

- Nuclide naming for neutron and di-neutron has been fixed.

1.2.22 Version 1.4.1

Fix:

- Storage for a single fit for a Non-Smoker rate entry has been fixed.

1.2.23 Version 1.4.0

New:

- It is now possible to retrieve reaction data from webnucleo xml files and compute rates for standard rate functions.

1.2.24 Version 1.3.0

New:

- It is now possible to set plot method arguments as a tuple giving an argument and a dictionary of optional keyword arguments.

1.2.25 Version 1.2.2

Fix:

- An XPath error in an xml routine has been fixed.
- A number of typos in the tutorials have been fixed.
- The name of an h5 movie routine has been changed to better reflect its purpose.

1.2.26 Version 1.2.1

Fix:

- A logical error in an h5 routine has been fixed.

1.2.27 Version 1.2.0

New:

- Routines to create certain movies have been added.

Fix:

- Some tutorial typos have been fixed and some missing text has been added.

1.2.28 Version 1.1.1

Internal:

- An integer type error has been fixed.

1.2.29 Version 1.1.0

New:

- The nuclear partition function data for each nuclide have been added to the nuclear data output.
- It is now possible to retrieve the abundances of all nuclides in zones or a subset of zones in the xml namespace.
- It is now possible to retrieve the network limits in the xml namespace.

Internal:

- XPath selection of zones has been improved.

1.2.30 Version 1.0.0

New:

- Initial release

1.3 Documentation

1.3.1 wnutils

wnutils package

A package of python routines to read and plot webnucleo xml and hdf5 files.

Submodules

wnutils.base module

class wnutils.base.**Base**
Class for setting wnutils parameters and utilities.

apply_class_methods(*plt*, *keyword_params*)
Method to apply plot functions.

Args: plt (matplotlib.pyplot): A pyplot plot instance.

keyword_params (dict): A dictionary of functions that will be applied to the plot. The key is the function and the value is the argument of the function.

Returns: On successful return, the functions have been applied to the plot.

create_nuclide_name(*z*, *a*, *state*)
Method to create the name of a nuclide.

Args: z (int): An integer giving the nuclide's atomic number.

a (int): An integer giving the nuclide's mass number.

state (str): A string giving the nuclide's state suffix.

Returns: str: The nuclide's name.

get_latex_names(*nuclides*)
Method to get latex strings of nuclides' names.

Args: nuclides (list): A list of strings giving the nuclides.

Returns: dict: A dictionary of latex strings.

get_z_a_state_from_nuclide_name(*name*)
Method to get the Z, A, and state from the name of a nuclide.

Args: name (str): The nuclide's name.

Returns: A tuple containing the Z, A, and state label corresponding to the name.

list_rcParams()
Method to list default rcParams.

Returns: Prints the default matplotlib.rcParams.

make_time_t9_rho_title_str(*props*, *i*)
Method to create a default title string.

Args: props (dict): A dictionary of float. The dictionary must contain entries that are numpy.array objects containing *time*, the time in seconds, *t9*, the temperature in billions of Kelvins, and *rho*, the mass density in grams per cubic centimeter.

`i` (`int`): An integer giving the location in the arrays of the properties to use to construct the string.

Returns: `str`: The default title string.

make_time_title_str(*time*)
Method to create a default title string.

Args: `props` (`dict`): A dictionary of `float`. The dictionary must contain at least one `numpy.array` object containing *time*, the time in seconds.

`time` (`int`): A float giving the time to use to construct the string.

Returns: `str`: The default title string.

set_plot_params(*my_mpl*, *my_params*)
Method to set plot parameters.

Args: `my_mpl` (matplotlib): A matplotlib instance.

`my_params` (`dict`): A dictionary with rcParams to be applied.

Returns: On successful return, the `matplotlib.rcParams` have first been set to their defaults and then updated with the values in my_params.

show_or_close(*plt*, *kwargs*)
Method to show or close plot.

Args: `plt` (`matplotlib.pyplot`): A pyplot plot instance.

`keyword_params` (`dict`): A dictionary of functions that will be applied to the plot. The key is the function and the value is the argument of the function.

Returns: On successful return, the plot has been shown or closed.

wnutils.h5 module

class wnutils.h5.**H5**(*file*)
Bases: *wnutils.base.Base*

A class for reading and plotting webnucleo HDF5 files.

Each instance corresponds to an hdf5 file. Methods extract data and plot data from the file.

Args: `file` (`str`): The name of the hdf5 file.

get_group_mass_fractions(*group*)
Method to return mass fractions from a group in an hdf5 file.

Args: `group` (`str`): The name of the group.

Returns: `Dataset`: A 2d hdf5 dataset. The first index indicates the zone and the second the species.

get_group_properties_in_zones(*group*, *properties*)
Method to return properties in all zones for a group.

Args:

`group` (`str`): A string giving the group name.

`properties` (`list`): A list of strings or tuples of up to three strings giving the properties to be retrieved.

Returns: `dict`: A dictionary of `list` giving the properties in the zones as strings.

get_group_properties_in_zones_as_floats(*group*, *properties*)
Method to return properties in all zones for a group as floats.

Args:

`group` (`str`): A string giving the group name.

`properties` (`list`): A list of strings or tuples of up to three strings giving the properties to be retrieved.

Returns: `dict`: A dictionary of `numpy.array` giving the properties in the zones as floats.

get_group_zone_properties(*group*, *zone*)
Method to return all properties in a zone in a group.

Args:

`group` (`str`): A string giving the group name.

`zone` (`tuple`): A three element tuple giving the three labels for the zone.

Returns: `dict`: A dictionary of strings giving all the properties in the zone in the group.

get_iterable_groups()
Method to return the non-nuclide data groups in an hdf5 file.

Returns: `list`: A list of strings giving the names of the groups.

get_nuclide_data()
Method to return nuclide data from an hdf5 file.

Returns:

`dict`: A dictionary of the nuclide data. Each entry is itself a dictionary containing the nuclide's index, name, z, a, source (data source), state, spin, and mass excess.

get_zone_labels_for_group(*group*)
Method to return zone labels for a group in a webnucleo hdf5 file.

Args: `group` (`str`): The name of the group.

Returns: `list`: A list of `tuple` giving the labels for the zones in a group.

get_zone_mass_fractions_in_groups(*zone*, *species*)
Method to return zone mass fractions in all groups.

Args:

`zone` (`tuple`): A three element tuple giving the three labels for the zone.

`species` (`list`): A list of strings giving the species whose mass fractions are to be retrieved.

Returns: `dict`: A dictionary of `numpy.array` giving the mass fractions in the groups.

get_zone_properties_in_groups(*zone*, *properties*)
Method to return zone properties in all groups.

Args:

`zone` (`tuple`): A three element tuple giving the three labels for the zone.

`properties` (`list`): A list of strings or tuples of up to three strings giving the properties to be retrieved.

Returns: dict: A dictionary of list giving the properties in the groups as strings.

get_zone_properties_in_groups_as_floats(*zone*, *properties*)
Method to return zone properties in all groups as floats.

Args:

zone (tuple): A three element tuple giving the three labels for the zone.

properties (list): A list of strings or tuples of up to three strings giving the properties to be retrieved.

Returns: dict: A dictionary of numpy.array giving the properties in the groups as floats.

make_mass_fractions_movie(*species*, *movie_name=''*, *property=None*, *fps=15*, *xfactor=1*, *use_latex_names=False*, *title_func=None*, *rcParams=None*, *plotParams=None*, ***kwargs*)
Method to make a movie of mass fractions in the zones.

Args:

species (list): A list of the species to include in the movie.

movie_name (str): A string giving the name of the resultin movie file.

property (str, optional): A string giving property to be the x axis. Defaults to zone index.

fps (float, optional): A float giving the frames per second in the resulting movie.

xfactor (float, optional): A float giving the scaling of the x axis.

use_latex_names (bool, optional): If set to True, species names converted to latex format.

title_func (optional): A function that applies the title to each frame of the movie. The function must take a single argument, an int giving the index of the frame to which the title will be applied. Other data can be bound to the function. The function must return either a str giving the title or a two-element tuple in which the first element is a string giving the title and the second element is a dict with optional matplotlib.pyplot.title keyword arguments. The default is a title giving the time in seconds.

rcParams (dict, optional): A dictionary of matplotlib.rcParams to be applied to the movie. Defaults to the default rcParams.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the lines in the movie.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns: The animation.

plot_group_mass_fractions(*group*, *species*, *use_latex_names=False*, *rcParams=None*, *plotParams=None*, ***kwargs*)
Method to plot group mass fractions vs. zone.

Args:

group (str): A string giving the group.

species (list): A list of strings giving the species to plot.

use_latex_names (bool, optional): If set to True, species names converted to latex format.

rcParams` (dict, optional): A dictionary of matplotlib.rcParams to be applied to the plot. Defaults to the default rcParams.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the plot. The list must have the same number of elements as species.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns: A matplotlib plot.

plot_group_mass_fractions_vs_property(*group*, *prop*, *species*, *xfactor=1*, *use_latex_names=False*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot group mass fractions vs. zone property.

Args:

group (str): A string giving the group.

prop (str or tuple): A string or tuple of up to three strings giving the property (which will serve as the plot abscissa).

species (list): A list of strings giving the species to plot.

xfactor (float, optional): A float giving the scaling for the abscissa values. Defaults to 1.

use_latex_names (bool, optional): If set to True, species names converted to latex format.

rcParams` (dict, optional): A dictionary of matplotlib.rcParams to be applied to the plot. Defaults to the default rcParams.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the plot. The list must have the same number of elements as species.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns: A matplotlib plot.

plot_group_property_in_zones(*group*, *property*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot a group property vs. zone.

Args:

group (str): A string giving the group.

property (str or tuple): A string or tuple of up to three strings giving the group.

rcParams` (dict, optional): A dictionary of matplotlib.rcParams to be applied to the plot. Defaults to the default rcParams.

plotParams (dict, optional): A dictionary of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the plot.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns: A matplotlib plot.

plot_zone_mass_fractions_vs_property(*zone*, *prop*, *species*, *xfactor=1*, *yfactor=None*, *use_latex_names=False*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot zone mass fractions vs. zone property.

Args:

`zone` (`tuple`): A three element tuple giving the zone.

`prop` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will serve as the plot abscissa).

`species` (`list`): A list of strings giving the species to plot.

`xfactor` (`float`, optional): A float giving the scaling for the abscissa values. Defaults to 1.

`yfactor` (`list`, optional): A list of floats giving factor by which to scale the mass fractions. Defaults to not scaling. If supplied, must by the same length as `species`.

`use_latex_names` (`bool`, optional): If set to True, species names converted to latex format.

`rcParams` (`dict`, optional): A dictionary of `matplotlib.rcParams` to be applied to the plot. Defaults to the default rcParams.

`plotParams` (`dict`, optional): A dictionary of valid `matplotlib.pyplot.plot` optional keyword arguments to be applied to the plot.

`**kwargs`: Acceptable `matplotlib.pyplot` functions. Include directly, as a `dict`, or both.

Returns: A matplotlib plot.

plot_zone_property_vs_property(*zone*, *prop1*, *prop2*, *xfactor=1*, *yfactor=1*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot a property vs. a property in a zone.

Args:

`zone` (`tuple`): A three element tuple giving the zone labels.

`prop1` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the abscissa of the plot).

`prop2` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the ordinate of the plot).

`xfactor` (`float`, optional): A float giving the scaling for the abscissa values. Defaults to 1.

`yfactor` (`float`, optional): A float giving the scaling for the ordinate values. Defaults to 1.

`rcParams` (`dict`, optional): A dictionary of `matplotlib.rcParams` to be applied to the plot. Defaults to the default rcParams.

`plotParams` (`dict`, optional): A dictionary of valid `matplotlib.pyplot.plot` optional keyword arguments to be applied to the plot.

`**kwargs`: Acceptable `matplotlib.pyplot` functions. Include directly, as a `dict`, or both.

Returns: A matplotlib plot.

wnutils.multi_h5 module

class wnutils.multi_h5.**Multi_H5**(*files*)

Bases: *wnutils.base.Base*

A class for reading and plotting webnucleo multiple HDF5 files.

Each instance corresponds to a set of HDF5 files. Methods plot data from the files.

Args: `files` (`list`): The names of the HDF5 files.

get_files()

Method to return the names of the input files.

Returns: `list`: A list of `str` giving the files

get_h5()

Method to return individual H5 instances.

Returns: `list`: A list of individual *wnutils.h5.H5* instances.

plot_zone_mass_fraction_vs_property(*zone*, *prop*, *species*, *xfactor=1*, *use_latex_names=False*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot a mass fraction versus a property.

Args:

`zone` (`tuple`): A three-element `tuple` giving the three `str` labels of the zone.

`prop` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the abscissa of the plot).

`species` (`str`): A string giving the species.

`xfactor` (`float`, optional): A float giving the scaling for the abscissa values. Defaults to 1.

`use_latex_names` (`bool`, optional): If set to True, converts species labels to latex format.

`rcParams` (`dict`, optional): A dictionary of `matplotlib.rcParams` to be applied to the plot. Defaults to the default rcParams.

`plotParams` (`list`, optional): A list of dictionaries of valid `matplotlib.pyplot.plot` optional keyword arguments to be applied to the plot. The list must have the same number of elements number of files in the class instance.

`**kwargs`: Acceptable `matplotlib.pyplot` functions. Include directly, as a `dict`, or both.

Returns:

A matplotlib plot.

plot_zone_property_vs_property(*zone*, *prop1*, *prop2*, *xfactor=1*, *yfactor=1*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot a property vs. a property in the files.

Args:

`zone` (`tuple`): A three-element `tuple` giving the three `str` labels of the zone.

`prop1` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the abscissa of the plot).

`prop2` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the ordinate of the plot).

xfactor (float, optional): A float giving the scaling for the abscissa values. Defaults to 1.

yfactor (float, optional): A float giving the scaling for the ordinate values. Defaults to 1.

rcParams (dict, optional): A dictionary of : obj: *matplotlib.rcParams* to be applied to the plot. Defaults to the default rcParams.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the plot. The list must have the same number of elements number of files in the class instance.

**kwargs: Acceptable: obj: *matplotlib.pyplot* functions. Include directly, as a dict, or both.

Returns: A matplotlib plot.

wnutils.multi_xml module

class wnutils.multi_xml.**Multi_Xml**(*files*)

Bases: *wnutils.base.Base*

A class for reading and plotting webnucleo multiple xml files.

Each instance corresponds to a set of xml files. Methods plot data from the files.

Args: files (list): The names of the xml files.

get_files()

Method to return the names of the input files.

Returns: list: A list of str giving the files

get_xml()

Method to return individual Xml instances.

Returns: list: A list of individual *wnutils.xml.Xml* instances.

plot_mass_fraction_vs_property(*prop*, *species*, *xfactor=1*, *use_latex_names=False*, *labels=None*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot a mass fraction versus a property.

Args:

prop (str or tuple): A string or tuple of up to three strings giving the property (which will be the abscissa of the plot).

species (str): A string orgiving the species.

xfactor (float, optional): A float giving the scaling for the abscissa values. Defaults to 1.

use_latex_names (bool, optional): If set to True, converts species labels to latex format.

rcParams (dict, optional): A dictionary of matplotlib.rcParams to be applied to the plot. Defaults to the default rcParams.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the plot. The list must have the same number of elements as the number of files in the class instance.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns:

A matplotlib plot.

plot_property_vs_property(*prop1*, *prop2*, *xfactor=1*, *yfactor=1*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot a property vs. a property in the files.

Args:

`prop1` (`str` or `tuple`): A string or tuple of up to three strings giving the property(which will be the abscissa of the plot).

`prop2` (`str` or `tuple`): A string or tuple of up to three strings giving the property(which will be the ordinate of the plot).

`xfactor` (`float`, optional): A float giving the scaling for the abscissa values. Defaults to 1.

`yfactor` (`float`, optional): A float giving the scaling for the ordinate values. Defaults to 1.

`rcParams` (`dict`, optional): A dictionary of : obj: *matplotlib.rcParams* to be applied to the plot. Defaults to the default rcParams.

`plotParams` (`list`, optional): A list of dictionaries of valid `matplotlib.pyplot.plot` optional keyword arguments to be applied to the plot. The list must have the same number of elements as the number of files in the class instance.

`**kwargs`: Acceptable: obj: *matplotlib.pyplot* functions. Include directly, as a `dict`, or both.

Returns: A matplotlib plot.

wnutils.xml module

class wnutils.xml.**New_Xml**(*xml_type='nuclear_network'*)

Bases: *wnutils.base.Base*

A class for creating webnucleo xml files.

Each instance corresponds to new xml. Methods set the nuclide, reaction, or zone data or write the xml to a file.

Args: `xml_type` (`str`, optional): The type of xml file to be created ("nuclear_data", "reaction_data", "nuclear_network", "zone_data", or "libnucnet_input"). Defaults to "nuclear_network".

set_nuclide_data(*nuclides*)

Method to set the nuclide data.

Args:

`nuclides` (`dict`): A dictionary containing the nuclides to be created and their data.

Returns: On successful return, the underlying xml has been created with the data in `nuclides`.

set_reaction_data(*reactions*)

Method to set the reaction data.

Args:

`reactions` (`dict`): A dictionary containing the reactions to be set and their data.

Returns: On successful return, the underlying xml has been created with the data in `reactions`.

set_zone_data(*zones*)
Method to set the zone data.

Args:

`zones` (`dict`): A dictionary containing the zones to be set and their data.

Returns: On successful return, the underlying xml has been created with the data in `reactions`.

write(*file*, *pretty_print=True*)
Method to write the xml to a file.

Args:

`file` (`str`): A string giving the name of output xml file.

`pretty_print` (`bool`, optional): If set to True, routine outputs the xml in nice indented format.

Returns: On successful return, the underlying xml has been written to `file`.

class wnutils.xml.**Reaction**
Bases: *wnutils.base.Base*

A class for storing and retrieving data about reactions.

compute_rate(*t9*, *user_funcs=' '*)
Method to compute rate for a reaction at input t9.

Args: `t9` (`float`): The temperature in billions of K giving the rate for the reaction.

`user_funcs` (`dict`, optional): A dictionary of user-defined functions associated with a user_rate key.

Returns: `float`: The computed rate.

get_data()
Method to return the data for a reaction.

Returns: `dict`: A dictionary containing the rate data for the reaction.

get_latex_string()
Method to return the latex string for a reaction.

Returns: `str`: The reaction string.

get_string()
Method to return the string for a reaction.

Returns: `str`: The reaction string.

class wnutils.xml.**Xml**(*file*)
Bases: *wnutils.base.Base*

A class for reading and plotting webnucleo xml files.

Each instance corresponds to an xml file. Methods extract data and plot data from the file.

Args: `file` (`str`): The name of the xml file.

get_abundances_vs_nucleon_number(*nucleon='a'*, *zone_xpath=' '*)
Method to retrieve abundances summed over nucleon number in zones.

Args: `nucleon` (`str`): String giving the nucleon number to sum over. Must be 'z', 'n', or 'a'. Defaults to 'a'.

`zone_xpath` (`str`, optional): XPath expression to select zones. Defaults to all zones.

Returns: `numpy.array`: A two-dimensional array in which the first index gives the zone and the second gives the nucleon number value.

get_all_abundances_in_zones(*zone_xpath=' '*)
Method to retrieve all abundances in zones.

Args: `zone_xpath` (`str`, optional): XPath expression to select zones. Defaults to all zones.

Returns: `numpy.array`: A three-dimensional array in which the first index gives the zone, the second gives the atomic number, and the third gives the neutron number. The array value is the abundance in the zone given by the first index of the species with atomic number and neutron number given by the second and third indices, respectively. The abundance of the species is the sum of the abundances of all states of that species.

get_all_properties_for_zone(*zone_xpath*)
Method to retrieve all properties in a zone in an xml file

Args: `zone_xpath` (`str`): XPath expression to select zone. Must be evaluate to a single zone.

Returns: `dict`: A dictionary containing all the properties in the zone as strings.

get_chain_abundances(*nucleon*, *zone_xpath=''*, *vs_A=False*)
Method to retrieve the abundances in a chain (fixed Z or N).

Args:

`nucleon` (`tuple`, optional): A tuple giving the nucleon. The first entry must be the nucleon type (must be 'z' or 'n') while the second entry must be the value.

`zone_xpath` (`str`, optional): A string giving the XPath expression to select the zones. Defaults to all zones.

`vs_A` (`bool`, optional): A boolean to select whether abscissa data should be mass number.

Returns: A `tuple` containing an array of the nucleon values as the first element and a two-d `numpy.array` as the second element. The first index of the two-d array indicates the step and the second the abundance of the species with the corresponding nucleon number in the first element with the same index.

get_mass_fractions(*species*, *zone_xpath=' '*)
Method to retrieve mass fractions of nuclides in specified zones.

Args: `species` (`list`): List of strings giving the species to retrieve.

`zone_xpath` (`str`, optional): XPath expression to select zones. Defaults to all zones.

Returns: `dict`: A dictionary of `numpy.array` containing the mass fractions of the requested species in the zones as floats.

get_network_limits(*nuc_xpath=' '*)
Method to retrieve the network limits from the nuclide data.

Args: `nuc_xpath` (`str`, optional): XPath expression to select the nuclides. Defaults to all nuclides.

Returns: `dict`: A dictionary of `numpy.array` containing the network limits. The array with key *z* gives the atomic values. The array with key *n_min* gives the lowest neutron number present for the corresponding atomic number. The array with key *n_max* gives the highest neutron number present for the corresponding atomic number.

get_nuclide_data(*nuc_xpath=' '*)
Method to retrieve nuclear data from webnucleo XML.

Args: nuc_xpath (str, optional): XPath expression to select nuclides. Defaults to all nuclides.

Returns: dict: A dictionary of nuclide data. The data for each nuclide are themselves contained in a dict.

get_properties(*properties*, *zone_xpath=' '*)
Method to retrieve properties in specified zones in an xml file

Args: properties (list): List of strings or tuples (each of up to three strings) giving requested properites.

zone_xpath (str, optional): XPath expression to select zones. Defaults to all zones.

Returns: dict: A dictionary of lists containing the properties in the zones as strings.

get_properties_as_floats(*properties*, *zone_xpath=' '*)
Method to retrieve properties in zones in an xml file as floats.

Args: properties (list): List of strings or tuples (each of up to three strings) giving requested properites.

zone_xpath (str, optional): XPath expression to select zones. Defaults to all zones.

Returns: dict: A dictionary of numpy.array containing the properties in the zones as floats.

get_reaction_data(*reac_xpath=' '*)
Method to retrieve reaction data from webnucleo XML.

Args: reac_xpath (str, optional): XPath expression to select reactions. Defaults to all reactions.

Returns: dict: A dictionary of reaction data. The data for each reaction are themselves contained in a *Reaction*.

get_type()
Method to retrieve the root type of the webnucleo XML.

Returns: str: One of *nuclear_data*, *reaction_data*, *nuclear_network*, *zone_data*, *libnucnet_input* indicating the root type of the XML.

get_zone_data(*zone_xpath=''*)
Method to retrieve zone data from webnucleo XML.

Args: zone_xpath (str, optional): XPath expression to select zones. Defaults to all zones.

Returns: dict: A dictionary of zone data. The data for each zone are themselves two dict, one containing properties and one containing mass fractions.

make_abundance_chain_movie(*movie_name=None*, *nucleon=('z', 26)*, *zone_xpath=''*, *plot_vs_A=False*, *fps=15*, *title_func=None*, *rcParams=None*, *plotParams=None*, *extraFixedCurves=None*, *extraCurves=None*, ***kwargs*)
Method to make of movie of abundances in a chain (fixed Z or N).

Args:

movie_name (str, optional): A string giving the name of resulting movie file.

nucleon (tuple, optional): A tuple giving the nucleon. The first entry must be the nucleon type (must be 'z' or 'n') while the second entry must be the value.

zone_xpath (str, optional): A string giving the XPath expression to select the zones. Defaults to all zones.

plot_vs_A (bool, optional): A boolean to select whether abscissa should be mass number.

fps (float, optional): A float giving the frames per second in the resulting movie.

title_func (optional): A function that applies the title to each frame of the movie. The function must take a single argument, an int giving the index of the frame to which the title will be applied. Other data can be bound to the function. The function must return either a str giving the title or a two-element tuple in which the first element is a string giving the title and the second element is a dict with optional matplotlib.pyplot.title keyword arguments. The default is a title giving the time in seconds, the temperature in billions of Kelvins, and the mass density in grams / cc.

rcParams (dict, optional): A dictionary of matplotlib.rcParams to be applied to the movie. Defaults to the default rcParams.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the lines in the movie.

extraFixedCurves (list, optional): A list of tuple objects giving fixed curves to appear on each frame of the animation. The first element of the tuple is a list giving the abscissa values for the curve, the second element is the ordinate values for the curve, and the third element, if present, is a dict of matplotlib.pyplot.plot optional keyword arguments to be applied to the extra fixed curves in the movie.

extraCurves (list, optional): A list of tuple objects giving curves to appear on each frame of the animation. The first element of the tuple is a list giving the abscissa values for the curve, the second element is a two-d numpy.array giving the ordinate values for the curve corresponding to each timestep in the animation, and the third element, if present, is a dict of matplotlib.pyplot.plot optional keyword arguments to be applied to the extra fixed curves in the movie.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns: The animation.

make_abundances_vs_nucleon_number_movie(*movie_name=''*, *nucleon='a'*, *zone_xpath=''*, *fps=15*, *title_func=None*, *rcParams=None*, *plotParams=None*, *extraFixedCurves=None*, *extraCurves=None*, ***kwargs*)

Method to make of movie of abundances summed by nucleon number.

Args:

movie_name (str, optional): A string giving the name of resulting movie file.

nucleon (str, optional): A string giving the nucleon (must be 'z', 'n', or 'a'). Defaults to 'a'.

zone_xpath (str, optional): A string giving the XPath expression to select the zones. Defaults to all zones.

fps (float, optional): A float giving the frames per second in the resulting movie file.

title_func (optional): A function that applies the title to each frame of the movie. The function must take a single argument, an int giving the index of the frame to which the title will be applied. Other data can be bound to the function. The function must return either a str giving the title or a two-element tuple in which the first element is a string giving the title and the second element is a dict with optional matplotlib.pyplot.title keyword arguments. The default is a title giving the time in seconds, the temperature in billions of Kelvins, and the mass density in grams / cc.

rcParams (dict, optional): A dictionary of matplotlib.rcParams to be applied to the movie. Defaults to the default rcParams.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the lines in the movie.

extraFixedCurves (list, optional): A list of tuple objects giving fixed curves to appear on each frame of the animation. The first element of the tuple is a list giving the abscissa values for the curve, the second element is the ordinate values for the curve, and the third element, if present, is a dict of matplotlib.pyplot.plot optional keyword arguments to be applied to the extra fixed curves in the movie.

extraCurves (list, optional): A list of tuple objects giving curves to appear on each frame of the animation. The first element of the tuple is a list giving the abscissa values for the curve, the second element is a two-d numpy.array giving the ordinate values for the curve corresponding to each timestep in the animation, and the third element, if present, is a dict of matplotlib.pyplot.plot optional keyword arguments to be applied to the extra fixed curves in the movie.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns: The animation.

make_network_abundances_movie(*movie_name=''*, *zone_xpath=''*, *fps=15*, *title_func=None*, *rcParams=None*, *imParams={}*, *show_limits=True*, *plotParams={'color': 'black'}*, ***kwargs*)

Method to make of movie of network abundances.

Args:

movie_name (str, optional): A string giving the name of resulting movie file.

zone_xpath (str, optional): A string giving the XPath expression to select the zones. Defaults to all zones.

fps (float, optional): A float giving the frames per second in the resulting movie file.

title_func (optional): A function that applies the title to each frame of the movie. The function must take a single argument, an int giving the index of the frame to which the title will be applied. Other data can be bound to the function. The function must return either a str giving the title or a two-element tuple in which the first element is a string giving the title and the second element is a dict with optional matplotlib.pyplot.title keyword arguments. The default is a title giving the time in seconds, the temperature in billions of Kelvins, and the mass density in grams / cc.

rcParams (dict, optional): A dictionary of matplotlib.rcParams to be applied to the movie. Defaults to the default rcParams.

imParams (dict, optional): A dictionary of matplotlib.pyplot.imshow options to be applied to the movie. The default is equivalent to calling with imParams={'origin':'lower', 'cmap': cm.BuPu, 'norm': LogNorm(), 'vmin': 1.e-10, 'vmax': 1}. *cm* in this call is the matplotlib.cm namespace. Any or all of these options can be overridden or others added by setting any of them in the input dict.

plotParams (list, optional): A list of dictionaries of valid matplotlib.pyplot.plot optional keyword arguments to be applied to the network limits. Defaults are shown in the usage statement.

**kwargs: Acceptable matplotlib.pyplot functions. Include directly, as a dict, or both.

Returns: The animation.

plot_abundances_vs_nucleon_number(*nucleon='a'*, *zone_xpath='[last()]'*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot abundances summed by nucleon number.

Args:

`nucleon` (`str`, optional): A string giving the nucleon (must be 'z', 'n', or 'a'). Defaults to 'a'.

`zone_xpath` (`str`, optional): A string giving the XPath expression to select the zones. Defaults to the last zone.

`rcParams` (`dict`, optional): A dictionary of `matplotlib.rcParams` to be applied to the plot. Defaults to the default rcParams.

`plotParams` (`list`, optional): A list of dictionaries of valid `matplotlib.pyplot.plot` optional keyword arguments to be applied to the plot. The list must have the same number of elements as the number as zones selected by the zone XPath.

`**kwargs`: Acceptable `matplotlib.pyplot` functions. Include directly, as a `dict`, or both.

Returns: A matplotlib plot.

plot_mass_fractions_vs_property(*prop*, *species*, *xfactor=1*, *use_latex_names=False*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot the mass fractions versus a property.

Args:

`prop` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the abscissa of the plot).

`specieslist`): A list of strings giving the species.

`xfactor` (`float`, optional): A float giving the scaling for the abscissa values. Defaults to 1.

`use_latex_names` (`bool`, optional): If set to True, converts species labels to latex format.

`rcParams` (`dict`, optional): A dictionary of `matplotlib.rcParams` to be applied to the plot. Defaults to the default rcParams.

`plotParams` (`list`, optional): A list of dictionaries of valid `matplotlib.pyplot.plot` optional keyword arguments to be applied to the plot. The list must have the same number of elements as `species`.

`**kwargs`: Acceptable `matplotlib.pyplot` functions. Include directly, as a `dict`, or both.

Returns:

A matplotlib plot.

plot_property_vs_property(*prop1*, *prop2*, *xfactor=1*, *yfactor=1*, *rcParams=None*, *plotParams=None*, ***kwargs*)

Method to plot a property vs. a property.

Args:

`prop1` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the abscissa of the plot).

`prop2` (`str` or `tuple`): A string or tuple of up to three strings giving the property (which will be the ordinate of the plot).

`xfactor` (`float`, optional): A float giving the scaling for the abscissa values. Defaults to 1.

`yfactor` (`float`, optional): A float giving the scaling for the ordinate values. Defaults to 1.

`rcParams` (`dict`, optional): A dictionary of `matplotlib.rcParams` to be applied to the plot. Defaults to the default rcParams.

`plotParams` (`dict`, optional): A dictionary of valid `matplotlib.pyplot.plot` optional keyword arguments to be applied to the plot.

`**kwargs`: Acceptable `matplotlib.pyplot` functions. Include directly, as a `dict`, or both.

Returns: A matplotlib plot.

validate()
Method to validate the xml

Returns: An error message if invalid and nothing if valid.

1.4 Tutorials

1.4.1 Installing wnutils

First ensure that you have pip installed on your system by typing (at the command line prompt $):

```
$ pip --help
```

If this command does not return a proper usage statement, install pip according to the instructions at the pip website. Note that with python version 3, you may have *pip3* instead of *pip*.

With pip installed, you may now use it to install wnutils by typing:

```
$ pip install wnutils
```

If the installation fails, you may need to install with root privileges using *sudo*:

```
$ sudo pip install wnutils
```

Alternatively, you can just install for yourself with the `--user` option:

```
$ pip install wnutils --user
```

If you have previously installed wnutils and want to upgrade, type:

```
$ pip install --upgrade wnutils
```

To test that wnutils has installed correctly, type:

```
$ pip show wnutils
```

which should return information about the package. To check that all necessary packages are in place, open python by typing:

```
$ python
```

or, perhaps for version 3:

```
$ python3
```

and try importing wnutils by typing at the python prompt:

```
>>> import wnutils
```

This command should simply return. If not, there may be a message telling you what packages your python installation is missing. For example, on some linux installations (particularly for python version 3), we have had to install *python3-tk* via:

```
$ sudo apt install python3-tk
```

In python version 2, that might be *python-tk*. Of course, to exit python, type:

```
>>> exit()
```

1.4.2 Getting the Data

In general, you will be using *wnutils* with data you have generated with webnucleo codes. However, we have already generated some data you can use with these tutorials. You must download those data from the web.

First, begin by creating a directory in which to work on the wnutils tutorials. You might create this off your home directory, so you could type:

```
$ cd ~
$ mkdir wnutils_tutorials
$ cd wnutils_tutorials
```

Next, download the data tarball and extract the data by typing:

```
$ curl -O -J -L https://osf.io/2a4kh/download
$ gunzip wnutils_tutorials_data.tar.gz
$ tar xvf wnutils_tutorials_data.tar
```

You can now check that you have the expected files by typing:

```
$ ls
```

You will see the files *my_output1.h5*, *my_output2.h5*, *my_output1.xml*, *my_output2.xml*, and *wnutils_tutorials_data.tar*. Since you have the data, you can remove the tar file if you would like by typing:

```
$ rm wnutils_tutorials_data.tar
```

You can always download that file again.

1.4.3 Reading in the Data

webnucleo data files are in either in XML or HDF5 format. *wnutils* routines can read either format.

In the following tutorials, you will enter Python commands. In your terminal, type python (or python3, or, perhaps, python2.x or python3.x, depending on your version). You will see something like:

```
Python 3.6.5 (default, Mar 29 2018, 15:38:28)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is the interactive python prompt (it will typically show up in the tutorials in red). You type your commands at this prompt. To exit python, type:

```
>>> exit()
```

and hit enter.

XML

The format of *webnucleo* XML files is described in the libnucnet technical report *XML Input to libnucnet*, available at the libnucnet Home page. The class *wnutils.xml.Xml* has methods that read these XML files. To begin, import the namespace by typing:

```
>>> import wnutils.xml as wx
```

To illustrate the use of *wnutils.xml* routines, use the files *my_output1.xml* and *my_output2.xml*, which you should have downloaded according to the *data* tutorial. For each file, create an Xml object. For example, type:

```
>>> my_xml = wx.Xml('my_output1.xml')
```

Read the nuclide data.

You can retrieve the nuclide data in the webnucleo XML file by typing:

```
>>> nuclides = my_xml.get_nuclide_data()
```

This returns a dictionary of data with the key being the nuclide name. You may print out all the data for a specific nuclide, say, o16, by typing:

```
>>> print(nuclides['o16'])
```

Or, to get specific data, try typing:

```
>>> print('The mass excess in MeV of o16 is', nuclides['o16']['mass excess'])
```

It is possible to use an XPath expression to select out only certain nuclides. For example, to get the data for nitrogen isotopes only, type:

```
>>> n = my_xml.get_nuclide_data(nuc_xpath='[z = 7]')
```

To confirm that you only retrieved nitrogen data, type:

```
>>> for isotope in n:
...     print(isotope, ':', 'Z =', n[isotope]['z'], 'A =', n[isotope]['a'])
...
```

Partition function data for the nuclei are stored in two `numpy.array` objects. The first array, with key *t9*, gives the temperature points (in billions of k) at which the partition function is evaluated. The second array, with key *partf*,

gives the partition function *G* evaluated at each of the temperature points. To see how this works, try printing out the partition function for one of the iron isotopes, say, fe56. Begin by extracting the data for the iron isotopes by typing:

```
>>> fe = my_xml.get_nuclide_data(nuc_xpath='[z = 26]')
```

Then print out the partition function *G* as a function of *t9* by typing:

```
>>> sp = 'fe56'
>>> for i in range(len(fe[sp]['t9'])):
...     print('t9 = ', fe[sp]['t9'][i], 'G(t9) = ', fe[sp]['partf'][i])
...
```

Read the network limits.

It is often useful to know the limits of the network that comprises the nuclei in the nuclear data collection. To get this information, type:

```
>>> lim = my_xml.get_network_limits()
```

This returns a `dict` of `numpy.array` objects. The array retrieved with key *z* gives the atomic numbers. The array retrieved with key *n_min* gives the smallest neutron number present for the corresponding atomic number, while the array retrieved with key *n_max* gives the largest neutron number present for the corresponding atomic number. You can print out the retrieved data by typing:

```
>>> for z in range(len(lim['z'])):
...     print('Z =', z, ': N_min =', lim['n_min'][z], ', N_max =', lim['n_max'][z])
...
```

You can retrieve a subnetwork with an XPath expression. For example, you can type:

```
>>> lim = my_xml.get_network_limits(nuc_xpath = '[z <= 5 or z >= 25]')
```

Now print out the data:

```
>>> for i in range(len(lim['z'])):
...     print('Z =', lim['z'][i], ': N_min =', lim['n_min'][i], ', N_max =', lim['n_
↪max'][i])
...
```

Read the reaction data.

You can retrieve the reaction data in the webnucleo XML file by typing:

```
>>> reactions = my_xml.get_reaction_data()
```

This returns a dictionary with the key being the reaction string and each value being a *Reaction*. To see a list of the reactions, type:

```
>>> for r in reactions:
...     print(r)
...
```

You can use an XPath expression to select the reactions. For example, you can type:

```
>>> reactions = my_xml.get_reaction_data('[count(non_smoker_fit) = 1]')
```

Since the reaction data include the reaction type, you can confirm your request by typing:

```
>>> for r in reactions:
...     data = reactions[r].get_data()
...     print(r, ': type is', data['type'])
...
```

You may choose a particular reaction from the dictionary by typing, for example:

```
>>> reac = reactions['n + he4 + he4 -> be9 + gamma']
```

It is then possible to retrieve the *reactants*, *products*, the reaction string, and code giving the source by typing:

```
>>> print(reac.reactants)
>>> print(reac.products)
>>> print(reac.get_string())
>>> print(reac.source)
```

You can also compute the rate for the reaction (among interacting multiplets and assuming one of the standard rate forms *single_rate*, *rate_table*, or *non_smoker_fit*) at a variety of temperatures by typing:

```
>>> import numpy as np
>>> t9s = np.power(10., np.linspace(-2,1))
>>> for t9 in t9s:
...     print(t9, reac.compute_rate(t9))
...
```

To compute the rate for user-defined rate functions, each defined with a *user_rate key*, first write a python routine for each rate function, then bind any data to that function (which must still take *t9* as an argument), and then create a dictionary of the functions associated with each *key*. Pass that dictionary into the *compute_rate* method with the keyword *user_funcs*.

Read all properties in a zone.

In a *webnucleo* XML file, a *zone* is a collection of the *mutable* quantities during a network calculation. For a single-zone network calculation, a zone is often a time step in the calculation. The zone will contain mass fractions of the network species at the time step to which the zone corresponds and properties, which can be any quantity, such as time, temperature, or density. The properties themselves have a *name* and up to two *tags*, called *tag1* and *tag2*. If the property only has a name, it can be retrieved by a `str`. If the property has tags, the identifier for the property is a `tuple` of up to three strings, namely, the *name*, *tag1*, and *tag2*.

To retrieve all the properties of a given zone, say, the 10th zone, type:

```
>>> props = my_xml.get_all_properties_for_zone('[position() = 10]')
```

Now you can print out the properties and their values in this zone by typing:

```
>>> for prop in props:
...     print(str(prop).rjust(25), ':', props[prop])
...
```

Notice the conversion to `str` to print out the *('exposure', 'n')* tuple correctly.

Read properties in all zones.

You can retrieve selected properties in all zones. For the present example, you retrieve the *time*, *t9* (temperature in billions of Kelvins), and *rho* (mass density in g/cc) by typing:

```
>>> props = my_xml.get_properties( ['time','t9','rho'] )
```

The properties are returned in the dictionary *props*. Each dictionary element is a list of strings giving the properties in the zones. To see this, type:

```
>>> print(props['time'])
```

This prints all the times. Print the first time entry by typing:

```
>>> print(props['time'][0])
```

To see the types, print:

```
>>> type(props)
```

which shows that it is a hash (`dict`). Next, type:

```
>>> type(props['time'])
```

which shows that each dictionary entry is a `list`. Next, type:

```
>>> type(props['time'][0])
```

which shows each list entry is a `str`.

To retrieve properties with tags, you need to enter the appropriate tuple. For example, type:

```
>>> props = my_xml.get_properties(['time', ('exposure', 'n')])
```

To print out the exposures, type:

```
>>> for i in range(len(props[('exposure', 'n')])):
...     print('time:', props['time'][i], 'exposure:', props[('exposure', 'n')][i])
...
```

Read properties of selected zones.

You can select out the zones whose properties you wish to read by using an XPath expression. For example, you can retrieve the *time*, *t9*, and *rho* properties, as in the above example, but only for the last 10 zones. Type:

```
>>> props = my_xml.get_properties(
...     ['time','t9','rho'], zone_xpath='[position() > last() - 10]'
... )
```

You can print the zone properties, for example, by typing:

```
>>> print(props['t9'])
```

Confirm that there are only the properties for 10 zones by typing:

```
>>> print(len(props['t9']))
```

Read zone properties as floats.

Properties are by default strings. When you wish to manipulate them (for example, to plot them), you want them to be `float` objects. You can retrieve them as floats by typing:

```
>>> props = my_xml.get_properties_as_floats( ['time','t9','rho'] )
```

The returned hash has entries that are `numpy.array`, which you confirm with:

```
>>> type(props['rho'])
```

You can confirm that the array entries are floats by typing:

```
>>> type(props['rho'][0])
```

You can print out the entries by typing:

```
>>> for i in range(len(props['time'])):
...     print(
...         'Zone = {0:d} time(s) = {1:.2e} t9 = {2:.2f} rho(g/cc) = {3:.2e}'.format(
...             i, props['time'][i], props['t9'][i], props['rho'][i]
...         )
...     )
...
```

This will output the time, temperature (in billions of K), and mass density (in g/cc) in all zones (time steps).

Read mass fractions in zones.

You can retrieve the mass fractions in zones. For example, to get the mass fractions of o16, si28, and s36, type:

```
>>> x = my_xml.get_mass_fractions(['o16','si28','s36'])
```

The method returns a `dict` of `numpy.array`. Each array element is a `float`. You can print the mass fraction of silicon-28 in all zones by typing:

```
>>> print(x['si28'])
```

The method also accepts the *zone_xpath* keyword to select specific zones. For example, to retrieve the mass fraction in the first 10 zones, type:

```
>>> x = my_xml.get_mass_fractions(
...      ['o16','si28','s36'], zone_xpath='[position() <= 10]'
... )
```

Read all abundances in zones.

You can retrieve abundances in the zones as a function of atomic and neutron number. The retrieved data are stored in a three-dimensional `numpy.array`. The first index gives the zone, the second gives the atomic number, and the third gives the neutron number. The array value is the abundance (per nucleon). Zones can be selected by XPath.

To see how this works, retrieve the abundances in all zones by typing:

```
>>> abunds = my_xml.get_all_abundances_in_zones()
```

Now print out the abundances in the 50th zone (remember the zero-indexing) by typing:

```
>>> for z in range(abunds.shape[1]):
...     for n in range(abunds.shape[2]):
...         print('Z =', z, ', N =', n, ', Y(Z,N) =', abunds[49,z,n])
...
```

You could do the same by typing:

```
>>> abunds = my_xml.get_all_abundances_in_zones(zone_xpath='[position() = 50]')
>>> for z in range(abunds.shape[1]):
...     for n in range(abunds.shape[2]):
...         print('Z =', z, ', N =', n, ', Y(Z,N) =', abunds[0,z,n])
...
```

This is because the XPath selects only one zone, which will have index 0 in the retrieved data.

Retrieve abundances summed over nucleon number in zones.

It is often convenient to retrieve the abundances of the nuclei in a network file summed over proton number (z), neutron number (n), or mass number (a). To do so, type:

```
>>> y = my_xml.get_abundances_vs_nucleon_number()
```

This returns a two-dimensional `numpy.array` in which the first index gives the zone and the second the mass number a. To print out the abundances versus mass number in the eighth zone, type:

```
>>> for i in range(y.shape[1]):
...     print('A:', i, 'Y(A):', y[7,i])
...
```

To retrieve the abundances summed over atomic (proton) number (z), use the keyword *nucleon*:

```
>>> y = my_xml.get_abundances_vs_nucleon_number(nucleon='z')
```

To retrieve the abundances in particular zones, for example, in the last 10 zones, use an XPath expression:

```
>>> y = my_xml.get_abundances_vs_nucleon_number(nucleon='n', zone_xpath='[position() >
↪ last() - 10]')
```

Retrieve abundances for a chain of species.

To retrieve the abundances for a set of isotopes or isotones, use the method to get chain abundances. For example, to retrieve the isotopic abundances for $Z = 30$ for all timesteps, type:

```
>>> n, y = my_xml.get_chain_abundances(('z', 30))
```

The method returns a `tuple` with the first element being an array of neutron numbers for the isotopes and the second element being a two dimensional `numpy.array` with the abundances for each step. To print the isotopic abundances in the final step, type:

```
>>> step = y.shape[0] - 1
>>> for i in range(y.shape[1]):
...     print('N =', n[i], ', Y[N] =', y[step, i])
...
```

To return the isotonic abundances for $N = 25$ in the first thirty timesteps, type:

```
>>> z, y = my_xml.get_chain_abundances(('n', 25), zone_xpath="[position() <= 30]")
```

To return the same isotonic abundances, but as a function of the mass number, set the keyword variable *vs_A* to True:

```
>>> a, y = my_xml.get_chain_abundances(('n', 25), zone_xpath="[position() <= 30]", vs_
↪A=True)
```

To print these abundances in the thirtieth step, type:

```
>>> step = y.shape[0] - 1
>>> for i in range(y.shape[1]):
...     print('A =', a[i], ', Y[A] =', y[step, i])
...
```

Multi_XML

The *wnutils.multi_xml.Multi_Xml* class allows you to access and plot data from multiple webnucleo XML files. First import the namespace by typing:

```
>>> import wnutils.multi_xml as mx
```

Then create a class instance from a `list` of XML files. For this tutorial, type

```
>>> my_multi_xml = mx.Multi_Xml(['my_output1.xml','my_output2.xml'])
```

Methods allow you to access or plot data from the files.

Read data from the individual XML instances.

To retrieve the individual XML instances from a Multi_Xml instance, type:

```
>>> xmls = my_multi_xml.get_xml()
```

To retrieve the original file names, type:

```
>>> files = my_multi_xml.get_files()
```

Of course the number of XML instances must equal the number of files. To confirm, type:

```
>>> print(len(xmls) == len(files))
```

Use the methods on the individual instances. For example, type:

```
>>> for i in range(len(xmls)):
...     props = xmls[i].get_properties(['time'])
...     print(files[i],'has',len(props['time']),'zones.')
...
```

H5

Methods that read webnucleo HDF5 files are in the namespace *wnutils.h5*. The class that contains these methods is *wnutils.h5.H5*. Begin by importing the namespace by typing:

```
>>> import wnutils.h5 as w5
```

Then create an object for your file *my_output1.h5* (which you already downloaded according to the instructions in the *data* tutorial) by typing:

```
>>> my_h5 = w5.H5('my_output1.h5')
```

Read the nuclide data.

The nuclide data are in a group of their own in the file. To retrieve the data (as a `dict` of `dict` with the nuclide names as the top-level dictionary keys), type:

```
>>> nuclides = my_h5.get_nuclide_data()
```

Print out the data for, say, o16, by typing:

```
>>> print(nuclides['o16'])
```

Print out the mass excess and spin for all species by typing:

```
>>> for nuclide in nuclides:
...     print(nuclide, nuclides[nuclide]['mass excess'], nuclides[nuclide]['spin'])
...
```

Read the names of the iterable groups.

Iterable groups are the groups in the HDF5 file that typically represent timesteps (that is, the groups that are not the nuclide data group). To retrieve their names (as a `list` of `str`), type:

```
>>> groups = my_h5.get_iterable_groups()
```

Print them out by typing:

```
>>> for group in groups:
...     print(group)
...
```

Read the zone labels for a group.

In a webnucleo HDF5 file, a zone is contained in a group and typically represents a spatial region. Zones are specified by three labels, which we denote by a `tuple`. To retrieve and print out the labels for a given group, say, *Step 00010*, type:

```
>>> labels = my_h5.get_zone_labels_for_group('Step 00010')
>>> for i in range(len(labels)):
...     print('Zone',i,'has label',labels[i])
...
```

Read all properties in a zone for a group.

To retrieve all the properties from a zone in a group, type, for example:

```
>>> zone = ('2','0','0')
>>> props = my_h5.get_group_zone_properties('Step 00010', zone)
```

You can print those properties out by typing:

```
>>> for prop in props:
...     print(str(prop).rjust(25), ':', props[prop])
...
```

Read properties in all zones for a group.

It is possible to retrieve the properties in all zones for a group as as `dict` of `list`. Each list entry is a `str`. For example, to retrieve and print the properties *time*, *t9*, and *rho* in all zones for a given group, say, *Step 00024*, type:

```
>>> p = ['time','t9','rho']
>>> props = my_h5.get_group_properties_in_zones('Step 00024',p)
>>> labels = my_h5.get_zone_labels_for_group('Step 00024')
>>> for i in range(len(labels)):
...     print('In',labels[i],'time=',props['time'][i],'t9=',props['t9'][i],'rho=',
↪props['rho'][i])
...
```

Read properties in all zones for a group as floats.

It is often desirable to retrieve the properties in zones for a group as floats. For example, one may again retrieve *time*, *t9*, and *rho* from *Step 00024* but, this time, as floats by typing:

```
>>> p = ['time','t9','rho']
>>> props = my_h5.get_group_properties_in_zones_as_floats('Step 00024',p)
>>> type(props['time'])
>>> type(props['time'][0])
```

Read mass fractions in all zones for a group.

You can read all the mass fractions in all the zones for a given group. For a group *Step 00021*, type:

```
>>> x = my_h5.get_group_mass_fractions('Step 00021')
```

The array x is a 2d HDF5 `Dataset`. The first index gives the zone and the second the species. To print out the mass fraction of ne20 in all the zones, type:

```
>>> i_ne20 = (my_h5.get_nuclide_data())['ne20']['index']
>>> labels = my_h5.get_zone_labels_for_group('Step 00021')
>>> for i in range(x.shape[0]):
...     print('Zone',labels[i],'has X(ne20) =',x[i,i_ne20])
...
```

Read properties of a zone in the groups.

It is possible to retrieve properties from a given zone in all groups. To retrieve the properties *time*, *t9*, and *rho* in all groups for the zone with labels *1*, *0*, *0*, type:

```
>>> zone = ('1','0','0')
>>> props = my_h5.get_zone_properties_in_groups(zone, ['time','t9','rho'])
```

This returns a `dict` of `list` of `str`. To print the properties out in the groups, type:

```
>>> groups = my_h5.get_iterable_groups()
>>> for i in range(len(groups)):
...     print(
...         groups[i], ': ', props['time'][i], props['t9'][i], props['rho'][i]
...     )
...
```

Read properties of a zone in the groups as floats.

One often wants the properties of a zone in the groups as floats. To retrieve the properties *time*, *t9*, and *rho* in all group for the zone with labels *1*, *0*, *0*, type:

```
>>> zone = ('1','0','0')
>>> props = my_h5.get_zone_properties_in_groups_as_floats(zone, ['time','t9','rho'])
```

This returns a `dict` of `numpy.array`. Each array entry is a `float`. To print the properties out in the groups, type:

```
>>> groups = my_h5.get_iterable_groups()
>>> for i in range(len(groups)):
...     print(
...         '{0:s}: time(s) = {1:.2e} t9 = {2:.2f} rho(g/cc) = {3:.2e}'.format(
...             groups[i], props['time'][i], props['t9'][i], props['rho'][i]
...         )
...     )
...
```

Read mass fractions in a zone in the groups.

You can retrieve the mass fractions of specific species for a given zone in all the iterable groups. For example, to retrieve *o16*, *o17*, and *o18* in the zone with labels *1*, *0*, *0*, type:

```
>>> species = ['o16','o17','o18']
>>> zone = ('1','0','0')
>>> x = my_h5.get_zone_mass_fractions_in_groups( zone, species )
```

To print them out, you can now type:

```
>>> groups = my_h5.get_iterable_groups()
>>> for i in range(len(groups)):
...     print(groups[i],':','X(o16)=',x['o16'][i],'X(o17)=',x['o17'][i],'X(o18)=',x[
↪'o18'][i])
...
```

Multi_H5

The *wnutils.multi_h5.Multi_H5* class allows you to access and plot data from multiple webnucleo HDF5 files. First import the namespace by typing:

```
>>> import wnutils.multi_h5 as m5
```

Then create a class instance from a `list` of HDF5 files. For this tutorial, type

```
>>> my_multi_h5 = m5.Multi_H5(['my_output1.h5','my_output2.h5'])
```

Methods allow you to access or plot data from the files.

Read data from the individual HDF5 instances.

To retrieve the individual HDF5 instances from a Multi_H5 instance, type:

```
>>> h5s = my_multi_h5.get_h5()
```

To retrieve the original file names, type:

```
>>> files = my_multi_h5.get_files()
```

Of course the number of HDF5 instances must equal the number of files. To confirm, type:

```
>>> print(len(h5s) == len(files))
```

Use the methods on the individual instances. For example, type:

```
>>> for i in range(len(h5s)):
...     props = h5s[i].get_zone_properties_in_groups(('0','0','0'), ['time'])
...     print(files[i],'has',len(props['time']),'groups.')
...
```

1.4.4 Creating and Writing XML Data

The preferred format for webnucleo data input is XML. *wnutils* routines allow users to create or update such XML.

The format of *webnucleo* XML files is described in the libnucnet technical report *XML Input to libnucnet*, available at the libnucnet Home page. The class *wnutils.xml.New_Xml* has methods that create new XML, set the nuclide, reaction, or zone data in the XML, and write the XML to a file. To begin, import the namespace by typing:

```
>>> import wnutils.xml as wx
```

Nuclide XML Data

Extract a subset of nuclide data.

Begin by retrieving the data that you wish to update. For this tutorial, use the file *my_output1.xml* which you should have downloaded according to the :ref:data tutorial. Read in the data by typing

```
>>> old_xml = wx.Xml('my_output1.xml')
```

Now get a subset of the nuclide data using an XPath expression. For this tutorial, get a subset that excludes calcium isotopes or any species with mass number 30 by typing

```
>>> nuclide_subset = old_xml.get_nuclide_data("[not(z = 20) and not(a = 30)]")
```

Confirm that the subset does not have the excluded species by examining the result of typing

```
>>> for nuc in nuclide_subset:
...     print(nuclide_subset[nuc]['z'], nuclide_subset[nuc]['a'])
...
```

Now create new nuclear data XML by typing

```
>>> subset_xml = wx.New_Xml(xml_type='nuclear_data')
```

Set the data in the new XML by typing

```
>>> subset_xml.set_nuclide_data(nuclide_subset)
```

and write the data to an XML file by typing

```
>>> subset_xml.write('subset_nuclear_data.xml')
```

You can now read those data into an Xml object by typing

```
>>> xml = wx.Xml('subset_nuclear_data.xml')
```

Now compare the two data files. Get the calcium and A=30 isotopes from both files and print out by typing

```
>>> check_old = old_xml.get_nuclide_data("[(z = 20) or (a = 30)]")
>>> print(len(check_old))
>>> check_new = xml.get_nuclide_data("[(z = 20) or (a = 30)]")
>>> print(len(check_new))
```

The old XML file contains calcium and A=30 isotopes but the new XML file does not.

Update existing nuclide data.

To update existing data, retrieve the nuclide data by typing

```
>>> nuclides = old_xml.get_nuclide_data()
```

nuclides is a dictionary with an entry for each nuclide chosen by the XPath expression input to the *get_nuclide_data()* method. The above routine call retrieves all the nuclide data. Each dictionary entry is itself a dictionary. To see the contents of an entry, type

```
>>> print(nuclides['o16'])
```

This shows that the dictionary entries for o16. Update the data for this species by typing

```
>>> nuclides['o16']['source'] = 'made-up data'
>>> nuclides['o16']['mass excess'] = 100
>>> nuclides['o16']['t9'] = [1,2,3,4]
>>> nuclides['o16']['partf'] = [1, 4, 9, 16]
```

Confirm the changes by typing

```
>>> print(nuclides['o16'])
```

Now create a new nuclear data XML file by typing

```
>>> updated_xml = wx.New_Xml(xml_type='nuclear_data')
```

Set the data in the new XML by typing

```
>>> updated_xml.set_nuclide_data(nuclides)
```

and write the data to an XML file by typing

```
>>> updated_xml.write('updated_nuclear_data.xml')
```

You can now read those data into an Xml object by typing

```
>>> xml = wx.Xml('updated_nuclear_data.xml')
```

Validate those data against the libnucnet XML nuclear data schema by typing

```
>>> xml.validate()
```

This will simply return, which shows that the data are valid. Next, retrieve the nuclide data and print out the o16 data:

```
>>> updated_nuclides = xml.get_nuclide_data()
>>> print(updated_nuclides['o16'])
```

The data in the new file are those that you have updated.

Add to existing nuclide data.

To add to existing data, retrieve the nuclide data by typing

```
>>> nuclides = old_xml.get_nuclide_data()
```

Create a new species in the nuclide data by typing

```
>>> nuclides['new'] = {}
```

Notice that the key can be any string different from the existing keys. Now add the data:

```
>>> nuclides['new']['z'] = 122
>>> nuclides['new']['a'] = 330
>>> nuclides['new']['source'] = 'made-up'
>>> nuclides['new']['state'] = ''
>>> nuclides['new']['mass excess'] = 500
>>> nuclides['new']['spin'] = 0.
>>> nuclides['new']['t9'] = [1,2,3,4]
>>> nuclides['new']['partf'] = [1,4,9,16]
```

Create the new XML, set the data, and write out the XML:

```
>>> extended_xml = wx.New_Xml(xml_type='nuclear_data')
>>> extended_xml.set_nuclide_data(nuclides)
>>> extended_xml.write('extended_nuclear_data.xml')
```

Read in the extended XML, validate, and print out the nuclide data to confirm the new species has been added:

```
>>> xml = wx.Xml('extended_nuclear_data.xml')
>>> xml.validate()
>>> extended_nuclides = xml.get_nuclide_data()
>>> for nuc in extended_nuclides:
...     print(nuc, extended_nuclides[nuc]['z'], extended_nuclides[nuc]['a'])
...
```

Create new nuclide data.

To create new nuclide XML data, first create a nuclide data dictionary:

```
>>> nuclides = {}
```

Now add species:

```
>>> t9 = [1,2,3,4]
>>> partf = [1,4,9,16]
>>> nuclides['new1'] = {'z': 13, 'a': 26, 'state': 'g', 'source': 'wn_tutorial',
↪'mass excess': -12.2101, 'spin': 5, 't9': t9, 'partf': partf}
>>> t9 = [1,2,3,4]
>>> partf = [1,8,27,64]
>>> nuclides['new2'] = {'z': 13, 'a': 26, 'state': 'm', 'source': 'wn_tutorial',
↪'mass excess': -11.9818, 'spin': 0, 't9': t9, 'partf': partf}
```

Create the new XML, set the data, write out the XML, read in the XML, and print out the nuclide data:

```
>>> new_xml = wx.New_Xml(xml_type='nuclear_data')
>>> new_xml.set_nuclide_data(nuclides)
>>> new_xml.write('new_nuclear_data.xml')
>>> xml = wx.Xml('new_nuclear_data.xml')
>>> new_nuclides = xml.get_nuclide_data()
>>> for nuc in new_nuclides:
...     print(nuc, new_nuclides[nuc]['z'], new_nuclides[nuc]['a'])
...
```

This shows the two species in the new XML file.

Reaction XML Data

Create new reaction XML analogously to creating new nuclide XML. Update an existing reaction data dictionary or create a new one, create a new reaction XML object, set the data in the object, and write to XML.

Extract a subset of reaction data.

To extract a subset of reaction data, first retrieve the data and get the data subset with XPath by typing

```
>>> old_xml = wx.Xml('my_output1.xml')
>>> reactions = old_xml.get_reaction_data("[not(reactant = 'kr85') and not(product =
↪'kr85')]")
```

The reactions data includes all reactions in the old data set except those involving *kr85*. Now create and write to XML:

```
>>> subset_xml = wx.New_Xml(xml_type='reaction_data')
>>> subset_xml.set_reaction_data(reactions)
>>> subset_xml.write('subset_reaction_data.xml')
```

One can now read in the data and validate:

```
>>> xml = wx.Xml('subset_reaction_data.xml')
>>> xml.validate()
```

Now check that the *kr85* reactions have been excluded:

```
>>> old_kr85 = old_xml.get_reaction_data("[reactant = 'kr85' or product = 'kr85']")
>>> new_kr85 = xml.get_reaction_data("[reactant = 'kr85' or product = 'kr85']")
>>> for reaction in old_kr85:
...     print(reaction)
...
>>> for reaction in new_kr85:
...     print(reaction)
...
```

The old XML data file includes reactions involving *kr85* but the new one does not.

Update existing reaction data.

To update existing data, retrieve the reaction data by typing

```
>>> reactions = old_xml.get_reaction_data()
```

reactions is a dictionary with an entry for each reaction chosen by the input XPath expression. The above call selects all reactions. Each entry in the dictionary is itself an instance of the *wnutils.xml.Reaction* class containing data for the reaction. To see an example of the data, type

```
>>> print(reactions['n + f19 -> f20 + gamma'].reactants)
>>> print(reactions['n + f19 -> f20 + gamma'].products)
>>> print(reactions['n + f19 -> f20 + gamma'].source)
>>> print(reactions['n + f19 -> f20 + gamma'].get_data())
```

The last command shows that the rate data for the reaction are of the *non_smoker_fit* type and are contained in a dictionary. Now update the data. Type

```
>>> reactions['n + f19 -> f20 + gamma'].source = 'ka02--updated'
>>> reactions['n + f19 -> f20 + gamma'].get_data()['fits'][0]['spint'] = 99.
```

It is also possible to change the data type. Change the *n + f20 -> f21 + gamma* from *non_smoker_fit* type to a *rate_table* type:

```
>>> print(reactions['n + f20 -> f21 + gamma'].get_data())
>>> t9 = [0.1,1,2,10]
>>> rate = [200, 150, 125, 100]
>>> sef = [1,1,1,1]
>>> reactions['n + f20 -> f21 + gamma'].data = {'type': 'rate_table', 't9': t9, 'rate
↪': rate, 'sef': sef}
```

The *t9* array gives the temperatures (in billions of K) at which the rates (*rate* array) are given. The *sef* is the *stellar enhancement factor*, which is the factor by which ground-state rate is increased in a stellar environment. When no *sef* is given, set it to unity.

Now confirm that the data have been updated by typing

```
>>> print(reactions['n + f19 -> f20 + gamma'].source)
>>> print(reactions['n + f19 -> f20 + gamma'].get_data())
>>> print(reactions['n + f20 -> f21 + gamma'].data)
```

Notice that the last command simply directly accessed the Reaction class member *data* instead of using the *get_data()* method. Either is valid–the *get_data()* method is simply a legacy convenience method that returns the class member *data*. Confirm the actions are the same by typing

```
>>> print(reactions['n + f20 -> f21 + gamma'].get_data())
```

Now create new XML and write the updated data:

```
>>> updated_xml = wx.New_Xml(xml_type='reaction_data')
>>> updated_xml.set_reaction_data(reactions)
>>> updated_xml.write('updated_reaction_data.xml')
```

Now confirm that the updated XML has the changes:

```
>>> xml = wx.Xml('updated_reaction_data.xml')
>>> updated_reactions = xml.get_reaction_data()
>>> print(updated_reactions['n + f19 -> f20 + gamma'].source)
>>> print(updated_reactions['n + f19 -> f20 + gamma'].get_data())
>>> print(updated_reactions['n + f20 -> f21 + gamma'].get_data())
```

Add to existing reaction data.

It is possible to add to existing reaction data. To try this, create the reaction *ni70 -> cu65 + n + n + n + n + n + electron + anti-neutrino_e* with a single rate of 1.5 per second:

```
>>> r = wx.Reaction()
>>> r.reactants = ['ni70']
>>> r.products = ['cu65', 'n', 'n', 'n', 'n', 'n', 'electron', 'anti-neutrino_e']
>>> r.source = 'wn_tutorials'
>>> r.data = {'type': 'single_rate', 'rate': 1.5}
```

Now add this to the existing data:

```
>>> old_xml = wx.Xml('my_output1.xml')
>>> reactions = old_xml.get_reaction_data()
>>> reactions['new'] = r
```

Create and write new XML with the extended data:

```
>>> extended_xml = wx.New_Xml(xml_type='reaction_data')
>>> extended_xml.set_reaction_data(reactions)
>>> extended_xml.write('extended_reaction_data.xml')
```

Confirm that the new XML has the added data:

```
>>> xml = wx.Xml('extended_reaction_data.xml')
>>> extended_reactions = xml.get_reaction_data("[reactant = 'ni70']")
>>> for reaction in extended_reactions:
...     print(reaction)
```

(continues on next page)

(continued from previous page)

```
...
>>> print(extended_reactions['ni70 -> cu65 + n + n + n + n + n + electron + anti-
↪neutrino_e'].get_data())
```

Create new reaction data.

It is also possible to create new reaction XML data. One creates a new reaction data dictionary and then sets those data in new XML and writes the XML out. To experiment with this, create a new reaction XML file with a *non_smoker_fit* data set and two *user_rate* data sets. In *user_rate* data, each rate datum is a *property* that is denoted by a `str` giving the property *name* or a `tuple` giving the property *name* and up to two tags (*tag1* and *tag2*). First, create the reactions data and add the *non_smoker_fit* reaction:

```
>>> reactions = {}
>>> reactions['new1'] = wx.Reaction()
>>> reactions['new1'].reactants = ['ge111', 'h1']
>>> reactions['new1'].products = ['as112', 'gamma']
>>> reactions['new1'].source = 'ADNDT (2001) 75, 1 (non-smoker)'
>>> reactions['new1'].data = {'type': 'non_smoker_fit', 'fits': [{'spint': 0.5, 'spinf
↪': 1.0, 'TlowHf': -1.0, 'Tlowfit': 0.01, 'Thighfit': 10.0, 'acc': 0.035, 'a1': 204.
↪211, 'a2': -10.533, 'a3': 414.2, 'a4': -658.043, 'a5': 37.4352, 'a6': -2.17474, 'a7
↪': 326.601, 'a8': 227.497}]}
```

Now add the first *user_rate* data reaction:

```
>>> reactions['new2'] = wx.Reaction()
>>> reactions['new2'].reactants = ['c12', 'c12']
>>> reactions['new2'].products = ['mg23', 'n']
>>> reactions['new2'].source = 'CF88'
>>> reactions['new2'].data = {'type': 'user_rate', 'key': 'cf88 carbon fusion fit',
...                           'f_0.11_le_t9_lt_1.75': '0.0', 'f_1.75_le_t9_lt_3.3':
↪'0.05',
...                           'f_3.3_le_t9_lt_6': '0.07', 'f_t9_ge_6': '0.07', 'f_t9_
↪lt_0.11': '0.0'}
```

Notice that all properties in the data dictionary are of `str` type. Also note that the *user_rate* needs a *key* entry denoting the particular user-rate function that will be used to compute the rate from the data. Now add the second *user_rate* data reaction:

```
>>> reactions['new3'] = wx.Reaction()
>>> reactions['new3'].reactants = ['c12', 'he4']
>>> reactions['new3'].products = ['o16', 'gamma']
>>> reactions['new3'].source = 'Kunz et al. (2002)'
>>> reactions['new3'].data = {'type': 'user_rate', 'key': 'kunz fit', ('a', '0'): ' 1.
↪21e8',
...                           ('a', '1'): ' 6.06e-2', ('a', '10'): ' 2.e6', ('a', '11
↪'): ' 38.534',
...                           ('a', '2'): ' 32.12', ('a', '3'): ' 1.7', ('a', '4'): '␣
↪7.4e8',
...                           ('a', '5'): ' 0.47', ('a', '6'): ' 32.12', ('a', '7'):
↪' 0.',
...                           ('a', '8'): ' 0.', ('a', '9'): ' 1.53e4'}
```

Notice here that the property keys are tuples where the entries are (*name*, *tag1*). Now create and write the XML:

```
>>> new_xml = wx.New_Xml(xml_type='reaction_data')
>>> new_xml.set_reaction_data(reactions)
>>> new_xml.write('new_reaction_data.xml')
```

Confirm the new XML:

```
>>> xml = wx.Xml('new_reaction_data.xml')
>>> new_reactions = xml.get_reaction_data()
>>> for r in new_reactions:
...     print(r, new_reactions[r].get_data())
...
```

Network XML Data

For webnucleo codes, a nuclear network is a collection of nuclides and the reactions among them. If you have already created or updated nuclide data *nuclides* and reaction data *reactions* according to the steps described above, you can create a network XML file. To do so, type

```
>>> network_xml = wx.New_Xml(xml_type='nuclear_network')
```

or, simply,

```
>>> network_xml = wx.New_Xml()
```

since the default new XML is of the *nuclear_network* type. Now set the data:

```
>>> network_xml.set_nuclide_data(nuclides)
>>> network_xml.set_reaction_data(reactions)
```

and write the file:

```
>>> network_xml.write('new_nuclear_network.xml')
```

Confirm the new file has the nuclide and reaction data:

```
>>> xml = wx.Xml('new_nuclear_network.xml')
>>> new_nuclides = xml.get_nuclide_data()
>>> new_reactions = xml.get_reaction_data()
>>> for nuc in new_nuclides:
...     print(nuc)
...
>>> for reaction in new_reactions:
...     print(reaction)
...
```

Zone XML Data

Zone data in webnucleo codes represent mutable data in a calculation. As with nuclide and reaction data, wnutils routines allow you to update and create new zone data XML.

Update existing zone data.

To update zone data, first retrieve the existing data:

```
>>> zone_data = old_xml.get_zone_data()
```

Zones are denoted by up to three labels (*label1*, *label2*, *label3*) given as either a string or a tuple of strings., Each zone can contain *optional_properties* and mass fractions of nuclear species. To see the available zones, type:

```
>>> for zone in zone_data:
...     print(zone)
,..
```

Create a new zone that is a copy of the last zone:

```
>>> new_zone = zone_data["164"].copy()
```

Modify a property and a mass fraction in the new zone:

```
>>> new_zone['properties']['rho'] = -10
>>> new_zone['mass fractions'][('he4', 2, 4)] = 0.1
```

Update the zone data with the new zone:

```
>>> zone_data[('165', 'added')] = new_zone
```

Now write the data to an XML file:

```
>>> updated_zone_xml = wx.New_Xml(xml_type='zone_data')
>>> updated_zone_xml.set_zone_data(zone_data)
>>> updated_zone_xml.write('updated_zone_data.xml')
```

Confirm that the new file has the new zone and the updated data:

```
>>> xml = wx.Xml('updated_zone_data.xml')
>>> updated_zone_data = xml.get_zone_data()
>>> for zone in updated_zone_data:
...     print(zone)
...
>>> print(updated_zone_data[('165', 'added')]['properties']['rho'])
>>> print(updated_zone_data[('165', 'added')]['mass fractions'][('he4', 2, 4)])
```

Create new zone data.

To create zone XML data, first create a dictionary of zones:

```
>>> zones = {}
```

Now create property dictionaries for the zones:

```
>>> props1 = {'width': 5}
>>> props2 = {'note': 'This is a note.', ('breadth', 'length', 'width'): 7}
```

Each dictionary key is either a `str` or a `tuple` of strings. The property value can be any type–it will be converted to a string. Now create dictionaries of mass fractions:

```
>>> mass_frac1 = {('he4', 2, 4): 1}
>>> mass_frac2 = {('mn53', 25, 53): 0.7, ('fe56', 26, 56): 0.3}
```

The key for each mass fraction entry is a tuple giving the species *name*, *Z*, and *A*. Now create the zones:

```
>>> zones["0"] = {'properties': props1, 'mass fractions': mass_frac1}
>>> zones[("Ringo", "Starr")] = {'properties': {}, 'mass fractions': mass_frac2}
>>> zones[("John", "Winston", "Lennon")] = {'properties': props2, 'mass fractions':␣
↪mass_frac2}
```

Now create the zone data XML, set the data, and write the file:

```
>>> zone_xml = wx.New_Xml('zone_data')
>>> zone_xml.set_zone_data(zones)
>>> zone_xml.write('new_zone_data.xml')
```

The file *new_zone_data.xml* contains the data you created. You can validate it to ensure the data are the right XML format:

```
>>> xml = wx.Xml('new_zone_data.xml')
>>> xml.validate()
```

Libnucnet XML Data

Full libnucnet data comprises nuclear network and zone data. If you have created nuclide data (*nuclides*), reaction data (*reactions*), and zone data (*zones*), you can create full libnucnet data by typing:

```
>>> libnucnet_xml = wx.New_Xml('libnucnet_input')
>>> libnucnet_xml.set_nuclide_data(nuclides)
>>> libnucnet_xml.set_reaction_data(reactions)
>>> libnucnet_xml.set_zone_data(zones)
```

Write out the data by typing:

```
>>> libnucnet_xml.write('new_libnucnet.xml')
```

1.4.5 Plotting the Data

If you have read in the various data from a webnucleo file, you can plot them using matplotlib. For example, to plot the abundance of Z=28 nuclei in *my_output1.xml* as a function of time, you can type in Python:

```
>>> import matplotlib.pyplot as plt
>>> import wnutils.xml as wx
>>> my_xml = wx.Xml('my_output1.xml')
>>> props = my_xml.get_properties_as_floats(['time'])
>>> yz = my_xml.get_abundances_vs_nucleon_number(nucleon='z')
>>> plt.plot(props['time'],yz[:,28])
>>> plt.xscale('log')
>>> plt.xlim([1.e-14,1.])
>>> plt.ylim([0.,0.0014])
>>> plt.xlabel('time (s)')
>>> plt.ylabel('Y(28)')
>>> plt.show()
```

Of course you can also write a Python file (called, say, my_plot.py) with the above lines and execute it by typing *python my_plot.py*.

While it is always possible to make such plots with data read in with *wnutils* routines, we have written several plotting methods for commonly made plots. The rest of this tutorial demonstrates how to use these methods.

Setting RcParams

All the plotting methods accept RcParams as keywords. These can be entered as a key and value pair or as a dictionary of `matplotlib.RcParams`. You can print the list of parameters (and their default values) that can be set by typing:

```
>>> import wnutils.base as wnb
>>> wb = wnb.Base()
>>> wb.list_rcParams()
```

Since the Base class is inherited by the other wnutils classes, the *list_rcParams()* method is available from any class instance.

For the purposes of this tutorial, define a dictionary of parameters by typing:

```
>>> my_params = {'lines.linewidth': 2, 'font.size': 14}
```

Setting plot parameters

The plotting methods accept *plotParams* as a keyword. The object passed in through the keyword is a `dict` of `matplotlib.pyplot.plot` optional keyword arguments. The dictionary values govern the lines drawn on the plot. For example, calling a *wnutils* plotting routine with

```
>>> params = {'color':'black'}
```

and then *plotParams* = *params* in the plotting routine can be thought of as plotting with the command:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, color='black')
```

When the plotting routine creates multiple curves on the same plot, the object passed in through *plotParams* is a `list` of dictionaries of `matplotlib.pyplot.plot` optional keyword arguments. Each dictionary in the list corresponds to a curve on the plot.

Setting plot methods

The plotting routines also accept keywords giving `matplotlib.pyplot` methods and their arguments. In such a case, the keyword is the method, and the value is the argument to the method. For example, calling a *wnutils* plotting routine with the keyword *xlabel* = *'time (s)'* is equivalent to typing:

```
>>> import matplotlib.pyplot as plt
>>> plt.xlabel('time (s)')
```

These can be entered directly or as a dictionary. If the method takes an argument and optional keywords, enter these as a tuple. For example, calling a *wnutils* plotting routine with the keyword *savefig = ('my_fig.png', {'bbox_inches': 'tight'})* is equivalent to typing:

```
>>> plt.savefig('my_fig.png', bbox_inches = 'tight')
```

The tuple must have two elements–the argument and the dictionary of optional keyword arguments.

XML

To make graphs from XML files, first import the namespace:

```
>>> import wnutils.xml as wx
```

Then create an object for each file. For example, type:

```
>>> my_xml = wx.Xml('my_output1.xml')
```

Plot properties against each other for the zones.

You can plot properties in the zones in an XML file against each other. For example, to plot *t9* vs. *time*, type:

```
>>> my_xml.plot_property_vs_property( 'time', 't9' )
```

Now apply class methods to the plot. For example, type:

```
>>> my_xml.plot_property_vs_property( 'time', 't9', xlabel = 'time (s)', ylabel = '$T_
↪9$' )
```

You can equivalently do this by defining the method keywords in a dictionary and calling that. To do so, type:

```
>>> kw = {'xlabel':'time (s)', 'ylabel':'$T_9$'}
>>> my_xml.plot_property_vs_property('time', 't9', **kw)
```

You can also do this with both procedures. For example, type:

```
>>> kw2 = {'xlabel':'time (s)'}
>>> my_xml.plot_property_vs_property('time', 't9', ylabel = '$T_9$', **kw2)
```

You can call with the RcParams previously defined by typing:

```
>>> my_xml.plot_property_vs_property('time', 't9', rcParams=my_params, **kw)
```

You can also call the the plotParams keyword by typing:

```
>>> my_xml.plot_property_vs_property('time', 't9', rcParams=my_params, plotParams={
↪'color':'black'}, **kw)
```

Plot mass fractions against a property.

You can plot mass fractions of species against a property (typically the time or temperature). For example, to plot the mass fractions of he4 and fe58 versus time, type:

```
>>> my_xml.plot_mass_fractions_vs_property( 'time', ['he4','fe58'] )
```

You can add appropriate keywords. For example, you can type:

```
>>> my_xml.plot_mass_fractions_vs_property( 'time', ['he4','fe58'], use_latex_
↪names=True, xlabel = 'time (s)', xlim=[1.e-6,1], xscale = 'log', ylim=[0,1])
```

By setting the *use_latex_names* keyword to true, species names appear as a superscript mass number in front of the element name. You can of course also use the RcParams:

```
>>> my_xml.plot_mass_fractions_vs_property( 'time', ['he4','fe58'], use_latex_
↪names=True, xlabel = 'time (s)', xlim=[1.e-6,1], xscale = 'log', ylim=[0,1],␣
↪rcParams=my_params)
```

If you want to plot the mass fraction for a single species, be sure to enter that species as a list of one element:

```
>>> kw3 = {'use_latex_names': True, 'xlabel': '$T_9$', 'xlim': [10,0]}
>>> my_xml.plot_mass_fractions_vs_property( 't9', ['si28'], **kw3, ylim=[1.e-12,1.e-
↪4], yscale = 'log')
```

Finally, note that you can define the species to plot as a list that you then enter into the plot command. For example, type:

```
>>> nuclides_list = ['fe56','fe57','fe58']
>>> my_xml.plot_mass_fractions_vs_property( 'time', nuclides_list, use_latex_
↪names=True, xlabel = 'time (s)', xlim=[1.e-6,1], xscale = 'log', ylim=[0,0.5],␣
↪rcParams=my_params)
```

You can generate the list from an XPath expression. For example, try typing:

```
>>> nuclides = my_xml.get_nuclide_data(nuc_xpath='[z = 26 and (a - z >= 30 and a - z
↪<= 32)]')
>>> nuclides_list = []
>>> for nuclide in nuclides:
...     nuclides_list.append(nuclide)
...
>>> print(nuclides_list)
```

Now you can use that list in the plotting routine.

Plot abundances versus nucleon number.

To plot the summed abundances over mass number A in the last zone, type:

```
>>> my_xml.plot_abundances_vs_nucleon_number()
```

To dress that up, try typing:

```
>>> my_xml.plot_abundances_vs_nucleon_number(xlim = [0,100], ylim = [1.e-10,1],␣
↪yscale='log', xlabel = 'Mass Number, A', ylabel = 'Y(A)')
```

Use keywords to plot against atomic number (Z) or neutron number (N) or to plot against a different time step (zone), using an XPath expression. For example, to plot elemental abundances in the 20th step, type:

```
>>> my_xml.plot_abundances_vs_nucleon_number(nucleon='z', zone_xpath='[position() =␣
↪20]', xlim = [0,50], ylim = [1.e-10,1], yscale='log', xlabel = 'Atomic Number, Z',␣
↪ylabel = 'Y(Z)')
```

To add a title giving the conditions at that step, type:

```
>>> props = my_xml.get_properties_as_floats( ['time','t9','rho'] )
>>> title_str = 'time(s) = {0:.2e}, t9 = {1:.2f}, rho(g/cc) = {2:.2e}'.format(
...                 props['time'][19], props['t9'][19], props['rho'][19]
...             )
>>> my_xml.plot_abundances_vs_nucleon_number(nucleon='z', zone_xpath='[position() =␣
↪20]', xlim = [0,50], ylim = [1.e-10,1], yscale='log', xlabel = 'Atomic Number, Z',␣
↪ylabel = 'Y(Z)', title=title_str)
```

Recall that the property arrays are zero-indexed.

You can plot more than one time step (zone) by using an XPath expression. For example, to plot the first and last time steps, type:

```
>>> my_xml.plot_abundances_vs_nucleon_number(zone_xpath='[(position() = 1) or␣
↪(position() = last())]', yscale = 'log', ylim = [1.e-10,1])
```

Use a list of plot parameters to label the steps and other keywords to give the plot the desired look:

```
>>> p_params = [{'label': 'first'}, {'label': 'last'}]
>>> my_xml.plot_abundances_vs_nucleon_number(zone_xpath='[(position() = 1) or␣
↪(position() = last())]', plotParams = p_params, yscale = 'log', ylim = [1.e-10,1],␣
↪xlabel = 'A, Mass Number', ylabel = 'Y(A)', xlim = [0,100], legend = {'title':
↪'time step', 'shadow': True})
```

It is also possible to label the steps with the legend keyword. To do this, type:

```
>>> my_xml.plot_abundances_vs_nucleon_number(zone_xpath='[(position() = 1) or␣
↪(position() = last())]', yscale = 'log', ylim = [1.e-10,1], xlabel = 'A, Mass Number
↪', ylabel = 'Y(A)', xlim = [0,100], legend = (['first','last'], {'title': 'time step
↪', 'shadow': True}))
```

You can save the figure as a file, for example, by typing:

```
>>> my_xml.plot_abundances_vs_nucleon_number(zone_xpath='[(position() = 1) or␣
↪(position() = last())]', yscale = 'log', ylim = [1.e-10,1], xlabel = 'A, Mass Number
↪', ylabel = 'Y(A)', xlim = [0,100], legend = (['first','last'], {'title': 'time step
↪', 'shadow': True}), savefig = ('my_fig.png', {'bbox_inches': 'tight'}))
```

Multi_XML

To make plots from multiple webnucleo XML files, first import the namespace:

```
>>> import wnutils.multi_xml as mx
```

Next, create an object for the files:

```
>>> my_multi_xml = mx.Multi_Xml(['my_output1.xml', 'my_output2.xml'])
```

Plot a property against a property in multiple files.

You can plot a property versus another property in multiple files. For example, to plot the *t9* versus *time* in our two files, type:

```
>>> my_multi_xml.plot_property_vs_property('time','t9')
```

Since the calculations are for different exponential expansion timescales, you can label them with a legend. First, find the timescale by noting that $\rho(t) = \rho(0)\exp(-t/\tau)$. This means that $\tau = -t\,/\ln\left(\rho(t)/\rho(0)\right)$. Choose, say, step 150 to compute the *tau* for the two calcluations. You can type:

```
>>> import math
>>> xmls = my_multi_xml.get_xml()
>>> p_params = []
>>> for xml in xmls:
...     props = xml.get_properties_as_floats(['time','rho'])
...     tau = -props['time'][150] / math.log(props['rho'][150]/props['rho'][0])
...     p_params.append({'label':('{:8.2f}'.format(tau)).strip() + 's'})
...
```

Now call the plot method with the plotParams keyword by typing:

```
>>> my_multi_xml.plot_property_vs_property('time','t9', plotParams = p_params, legend=
↪{'title':'tau'})
```

Notice the call to the legend keyword. The keyword values can be any valid keyword argument to `matplotlib.pyplot.legend`. Thus, for example, you could type:

```
>>> my_multi_xml.plot_property_vs_property('time','t9', plotParams = p_params, legend=
↪{'title':'tau', 'shadow':True})
```

Plot a mass fraction against a property in multiple files.

You can also plot a mass fraction versus a property in multiple files. For example, to plot the mass fraction of fe58 as a function of time (and using the labels you defined above), type:

```
>>> my_multi_xml.plot_mass_fraction_vs_property('time', 'fe58', plotParams = p_params,
↪ legend={'title':'tau'})
```

wnutils.multi_xml.Multi_Xml plotting methods accept valid *rcParams* and other keywords, as in the *wnutils.xml.Xml* methods.

H5

To make plots from webnucleo HDF5 file, first import the namespace:

```
>>> import wnutils.h5 as w5
```

Next, create an object for each file by typing:

```
>>> my_h5 = w5.H5('my_output1.h5')
```

Plot a property versus a property for a given zone.

You can plot the values of two properties in all groups against each other for a given zone. For example, to plot *t9* versus *time* in the zone with labels *2*, *0*, *0*, type:

```
>>> zone = ('2','0','0')
>>> kws = {'xlabel': 'time (yr)', 'ylabel': '$T_9$'}
>>> my_h5.plot_zone_property_vs_property(zone, 'time', 't9', xfactor=3.15e7, **kws)
```

In the calculation that gave the output in *my_output1.h5*, the temperature and density in zones were constant in time.

Plot mass fractions versus a property for a given zone.

You can plot mass fractions against a property for a given zone. For example, type:

```
>>> my_h5.plot_zone_mass_fractions_vs_property(
...      ('1','0','0'), 'time', ['he4', 'c12','o16'], yscale = 'log',
...       ylim = [1.e-5,1], xscale = 'log', xlim = [1,1.e5], xfactor = 3.15e7,
...       xlabel = 'time (yr)', use_latex_names=True
... )
```

Note, this is equivalent to typing:

```
>>> zone = ('1','0','0')
>>> species = ['he4','c12','o16']
>>> kwa = {'xlim': [1,1.e5], 'ylim': [1.e-5,1]}
>>> kwb = {'xscale': 'log', 'yscale': 'log', 'xfactor': 3.15e7}
>>> kwc = {'xlabel': 'time (yr)', 'use_latex_names': True}
>>> my_h5.plot_zone_mass_fractions_vs_property( zone, 'time', species, **kwa, **kwb,␣
↪**kwc)
```

Or, in Python 3.5 or greater, you can type:

```
>>> kws = {**kwa,**kwb,**kwc}
>>> my_h5.plot_zone_mass_fractions_vs_property( zone, 'time', species, **kws)
```

Plot a property in the zones of a given group.

To plot a property in all the zones of a given group, say, Step number 125, you can, for example, type:

```
>>> my_h5.plot_group_property_in_zones('Step 00125', 't9')
```

This shows the temperature (in billions of Kelvins) in the zones. The innermost (first) zone is the hottest.

Plot mass fractions for a given group.

You can plot the mass fractions for a given group. The abscissa of the plot in this case will be a zone index. For example, type:

```
>>> my_h5.plot_group_mass_fractions(
...     'Step 00125', ['he4', 'c12','o16'], use_latex_names=True
... )
```

Plot group mass fractions versus a property.

In the previous example, you simply plotted the mass fractions against their zone. You can also plot against a zone property. For example, type:

```
>>> my_h5.plot_group_mass_fractions_vs_property(
...     'Step 00125', 't9', ['he4', 'c12','o16'], use_latex_names=True
... )
```

Notice that the plot shows the lowest temperature zone to the right part of the plot. To show the graph with the innermost (hottest) zones plotted to the right, use the *xlim* keyword:

```
>>> my_h5.plot_group_mass_fractions_vs_property(
...     'Step 00125', 't9', ['he4', 'c12','o16'], use_latex_names=True, xlim = [0.3,0]
... )
```

Multi_H5

To make plots from multiple webnucleo HDF5 files, first import the namespace:

```
>>> import wnutils.multi_h5 as m5
```

Next, create an object for the files:

```
>>> my_multi_h5 = m5.Multi_H5(['my_output1.h5', 'my_output2.h5'])
```

Plot a zone property against a property in multiple files.

You can plot a property versus another property in multiple files. For example, to plot the *neutron exposure* versus *time* in our two files, type:

```
>>> zone = ('0','0','0')
>>> my_multi_h5.plot_zone_property_vs_property(zone, 'time',('exposure', 'n'))
```

Notice that the *neutron exposure* property is input as a tuple because, in this case, the property identifier has two parts: a *name* string ('exposure') and a *tag1* string ('n'). As discussed in the *Reading in the Data* tutorial, a property can have a name and up to two tags; thus, the tuple identifying the property could have up to three elements. The neutron exposure is usually labeled τ_n and has units of mb^{-1}, that is, inverse millibarns. The difference in the two calculations is that the first was for a mixing timescale of 10^7 seconds while the second was for a mixing timescale of 10^9 seconds. We can thus add a legend by typing:

```
>>> p_params = [{'label':'$10^7\ s$', 'color':'black', 'linestyle':'-'}, {'label':'
↪$10^9\ s$', 'color':'black', 'linestyle':':'}]
```

Now call the plot method with the plotParams keyword by typing:

```
>>> my_multi_h5.plot_zone_property_vs_property(
...     zone, 'time',('exposure', 'n'), plotParams = p_params, legend={'title':'
↪$\\tau_{mix}$'},
...     xlabel='time (yr)', xfactor=3.15e7, ylabel='$\\tau_n(mb^{-1})$'
... )
```

As with *wnutils.multi_xml*, the legend keyword values can be any valid keyword argument to matplotlib.pyplot.legend. Thus, for example, you could type:

```
>>> my_multi_h5.plot_zone_property_vs_property(
...     zone, 'time',('exposure', 'n'), plotParams = p_params,
...     legend={'title':'$\\tau_{mix}$', 'shadow':True},
...     xlabel='time (yr)', xfactor=3.15e7, ylabel='$\\tau_n(mb^{-1})$'
... )
```

Plot a zone mass fraction against a property in multiple files.

You can also plot a mass fraction versus a property in multiple files. For example, to plot the mass fraction of fe56 as a function of time, type:

```
>>> my_multi_h5.plot_zone_mass_fraction_vs_property(zone, 'time', 'fe56', plotParams␣
↪= p_params, legend={'title':'$\\tau_{mix}$'})
```

wnutils.multi_h5.Multi_H5 plotting methods accept valid *rcParams* and other keywords, as in the *wnutils.h5.H5* methods.

1.4.6 Animating the Data

As with plotting, if you have read in the various data from a webnucleo file, you can animate them using matplotlib. We have found, however, that it is convenient to have a handful of movie methods in the *wnutils* API. This tutorial demonstrates how to use these methods. Interested users can, if desired, build their own movie routines based on the source code of the *wnutils* routines.

Animation writers

The default animation writer is ffmpeg. If you do not already have it on your system, you should install it. In linux, use apt-get. To install, type:

```
$ sudo apt install ffmpeg
```

On a mac with MacPorts, type:

```
$ sudo port install ffmpeg
```

On Cygwin, you will probably have to build it. For example, see this web site.

Setting parameters and methods

Just like the plotting methods, the animation methods use *rcParams*, *plot parameters*, and *plot methods*. You can set these as described in the *Plotting the Data* tutorial.

XML

To make movies from XML files, import the namespace:

```
>>> import wnutils.xml as wx
```

Then create an object for each file. For example, type:

```
>>> my_xml = wx.Xml('my_output1.xml')
```

Animating the abundances versus nucleon number

To make a movie of the abundances versus mass number, you can type:

```
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4')
```

The argument *abunds.mp4* is the name of the movie file that will be created. You can add appropriate keyword arguments to adjust the movie to your taste. For example, you can type:

```
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', xlim = [0,100], ylim␣
↪= [1.e-10,1], yscale = 'log', xlabel = 'A, Mass Number', ylabel = 'Abundance')
```

You can add *rcParams* and *plotParams*. For example, type:

```
>>> rc_params = {'lines.linewidth': 2}
>>> p_params = {'color': 'black'}
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', rcParams = rc_params,
↪ plotParams = p_params, xlim = [0,100], ylim = [1.e-10,1], yscale = 'log', xlabel =
↪'A, Mass Number', ylabel = 'Abundance')
```

(continues on next page)

(continued from previous page)

By default, a title is displayed giving the time in seconds, the temperature in billions of Kelvins, and the mass density in grams per cc. You can create your own title by supplying a title function. The function must take in an integer giving the index of a given frame in the movie and must return a string giving the title or a two-element tuple for which the first element is the title string and the second is a `dict` of `matplotlib.pyplot.title` keyword options. The function can also return *None*, in which case no title is created. For example, to prevent a title from appearing in the movie, type:

```
>>> def null_title(i):
...     return None
...
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', rcParams = rc_params,
↪ plotParams = p_params, title_func=null_title, xlim = [0,100], ylim = [1.e-10,1],␣
↪yscale = 'log', xlabel = 'A, Mass Number', ylabel = 'Abundance')
```

To make a title that only displays the time and temperature, define the appropriate title function by typing:

```
>>> def my_title(props, i):
...     title_str = \
...     "time (s) = %8.2e, $T_9$ = %4.2f" % \
...     (props['time'][i], props['t9'][i])
...     return title_str
...
```

Now bind properties to the function by typing:

```
>>> props = my_xml.get_properties_as_floats(['time', 't9'])
>>> bind = lambda i: my_title(props, i)
```

Now call the animation routine with the bound function by typing:

```
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', rcParams = rc_params,
↪ plotParams = p_params, title_func=bind, xlim = [0,100], ylim = [1.e-10,1], yscale␣
↪= 'log', xlabel = 'A, Mass Number', ylabel = 'Abundance')
```

If you want to modify the title properties, have the title function return a tuple. For example, type:

```
>>> def my_title2(props, i):
...     title_str = \
...     "time (s) = %8.2e, $T_9$ = %4.2f" % \
...     (props['time'][i], props['t9'][i])
...     return (title_str, {'fontsize': 20, 'color': 'green'})
...
>>> props = my_xml.get_properties_as_floats(['time', 't9'])
>>> bind = lambda i: my_title2(props, i)
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', rcParams = rc_params,
↪ plotParams = p_params, title_func=bind, xlim = [0,100], ylim = [1.e-10,1], yscale␣
↪= 'log', xlabel = 'A, Mass Number', ylabel = 'Abundance')
```

You can plot abundances versus atomic number (z) or neutron number (n) by supplying the appropriate keyword. For example, type:

```
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', nucleon = 'z')
```

You can also select zones (steps) to plot with an XPath expression. For example, type:

```
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', nucleon = 'n', zone_
↪xpath = '[position() >= last() - 30]')
```

That creates a movie of the abundances versus neutron number for the last 30 time steps. It should be clear that, if you use an XPath expression to select zones, and if you create your own title from properties, you will need to use the same XPath expression for the properties to be fed into the title function. For example, you could type:

```
>>> props = my_xml.get_properties_as_floats(['time','t9'], zone_xpath = '[position() >
↪= last() - 30]')
>>> bind = lambda i: my_title2(props, i)
>>> my_xml.make_abundances_vs_nucleon_number_movie('abunds.mp4', nucleon = 'n', zone_
↪xpath = '[position() >= last() - 30]', title_func = bind)
```

If you do not do this, you will have a mismatch between the frames and their titles.

You can also add extra curves to the movie that either stay fixed in each frame or vary. To do so for a fixed curve, create an array of tuples. The first element of the tuple gives the array of abscissa values, the second element gives the array of ordinate values, and the third element, if provided, gives a dictionary of valid matplotlib plot options. Pass the array of tuples into the methods as the keyword parameter *extraFixedCurves*. For example, you could type:

```
>>> import numpy as np
>>> ya = my_xml.get_abundances_vs_nucleon_number(zone_xpath = "[position() = 1]")
>>> my_extra = [(np.arange(len(ya[0])), ya[0], {'lw': 0.3, 'color': 'blue', 'label':
↪'Initial'})]
>>> anim = my_xml.make_abundances_vs_nucleon_number_movie(extraFixedCurves=my_extra,␣
↪yscale = 'log', ylim = [1.e-10,1], plotParams={'label': 'Current'}, legend={'loc':
↪'upper right'})
```

This returns the animation. You can the write a movie by typing:

```
>>> anim.save('abunds.mp4', fps = 15)
```

Of course, you can also pass the movie name in as the first parameter or as a keyword to make the movie directly:

```
>>> my_xml.make_abundances_vs_nucleon_number_movie(movie_name = 'abunds.mp4',␣
↪extraFixedCurves=my_extra, yscale = 'log', ylim = [1.e-10,1], plotParams={'label':
↪'Current'}, legend={'loc': 'upper right'})
```

Animating an abundance chain

An abundance chain is the collection of abundances along a fixed Z or N. To make a movie of an abundance chain, type:

```
>>> my_xml.make_abundance_chain_movie('abund_chain.mp4')
```

The argument *abund_chain.mp4* is the name of the movie file that will be created. The default is to plot along the fixed $Z = 26$ chain. To plot against a different Z, use the *nucleon* keyword to enter a tuple. For example, to plot for $Z = 30$, type:

```
>>> my_xml.make_abundance_chain_movie(movie_name = 'abund_chain.mp4', nucleon=('z',␣
↪30), plot_vs_A=True)
```

The *plot_vs_A* keyword causes the abscissa to be mass number instead of neutron number. To plot for $N = 30$, type:

```
>>> my_xml.make_abundance_chain_movie('abund_chain.mp4', nucleon=('n', 30), plot_vs_
↪A=True)
```

As with the abundances versus nucleon number movie, you can add appropriate keyword arguments and extra curves to adjust the movie to your taste. For example, you can type:

```
>>> my_nucleon = ('z', 28)
>>> x, y = my_xml.get_chain_abundances(my_nucleon, zone_xpath="[last()]")
>>> extra_curve = [(x, y[0], {'lw': 0.5, 'label': 'Final', 'color': 'red'})]
>>> my_xml.make_abundance_chain_movie('abund_chain.mp4', nucleon = my_nucleon, xlim =␣
↪[20, 50], ylim = [1.e-10,1], yscale = 'log', xlabel = 'N, Neutron Number', ylabel =
↪'Abundance', extraFixedCurves = extra_curve, plotParams = {'label': 'Current'},␣
↪legend={'loc': 'upper right'})
```

You can also adjust the title by defining a title function and binding, as with the nucleon number movie.

Animating the network abundances

You can animate the network abundances in the neutron number-proton number plane. For example, type:

```
>>> my_xml.make_network_abundances_movie('network_abunds.mp4')
```

The black curves in the movie show the network limits. The properties of those lines are set with *plotParams*. To see how this works, type:

```
>>> my_xml.make_network_abundances_movie('network_abunds.mp4', plotParams={'color':
↪'green', 'linestyle': 'dotted'})
```

The routine takes keyword arguments, as usual. For example, type:

```
>>> my_xml.make_network_abundances_movie('network_abunds.mp4', xlim=[0,60], ylim = [0,
↪50])
```

The abundances are shown by the blue-purple color intensity. The details are set by the keyword argument *imParams*, which is a `dict` of valid `matplotlib.pyplot.imshow` options. The default is as if you had called the routine with imParams={'origin':'lower', 'cmap': cm.BuPu, 'norm': LogNorm(), 'vmin': 1.e-10, 'vmax': 1}, which shows that the abundances are displayed on a logarithmic scale with maximum value 1 and minimum value 1.e-10. We can override any or all of these. For example, to change the minimum abundance to 1.e-15 and the color map to reds, type:

```
>>> import matplotlib.cm as cm
>>> my_xml.make_network_abundances_movie('network_abunds.mp4', xlim=[0,60], ylim = [0,
↪50], imParams = {'cmap': cm.Reds, 'vmin': 1.e-15})
```

It is often desirable to add a colorbar. For example, you can create colorbar properties by typing:

```
>>> cb = {'shrink': 0.85, 'label': 'Abundance', 'aspect': 10, 'ticks': [1.e-10, 1.e-8,
↪ 1.e-6, 1.e-4, 1.e-2, 1.]}
```

The arguments to the colorbar properties are any valid `matplotlib.pyplot.colorbar` optional keyword argument. You can now type:

```
>>> my_xml.make_network_abundances_movie('network_abunds.mp4', xlim=[0,60], ylim = [0,
↪50], colorbar = cb)
```

Of course, you will want to make sure that your ticks in the colorbar are consistent with your limits. For example, you can type:

```
>>> cb = {'shrink': 0.85, 'label': 'Abundance', 'aspect': 10, 'ticks': [1.e-15, 1.e-
↪10, 1.e-5, 1.]}
>>> my_xml.make_network_abundances_movie('network_abunds.mp4', xlim=[0,60], ylim = [0,
↪50], imParams = {'cmap': cm.Reds, 'vmin': 1.e-15}, colorbar = cb)
```

As with the routine to animate abundances versus nucleon number, you can use *zone_xpath* to select steps and *title_func* to define your own title string. For example, if you defined *my_title2()* as above, you can type:

```
>>> props = my_xml.get_properties_as_floats(['time','t9'])
>>> bind = lambda i: my_title(props, i)
>>> my_xml.make_network_abundances_movie('network_abunds.mp4', xlim=[0,60], ylim = [0,
↪50], imParams = {'cmap': cm.Reds, 'vmin': 1.e-15}, colorbar = cb, title_func = bind)
```

H5

To make movies from HDF5 files, import the namespace:

```
>>> import wnutils.h5 as w5
```

Then create an object for each file. For example, type:

```
>>> my_h5 = w5.H5('my_output1.h5')
```

Animating the mass fractions in zones

Most commonly one writes out HDF5 files for multi-zone network calculations. The output in *my_output1.h5* and *my_output2.h5* is for one-dimensional multi-zone network calculations in which matter burns in the individual zones and mixes between the zones. In such calculations, one generally wants to see the evolution of the mass fractions in the zones as a function of time. To see an example of how you can do this, type:

```
>>> my_h5.make_mass_fractions_movie(['o16','ne20'], 'mass_fracs.mp4')
```

This creates a movie *mass_fracs.mp4* of o16 and ne20 in the zones as a function of time. The x axis shows zone indices. The y axis gives mass fractions. The scale of the y axis changes. Since you probably want that fixed, call with keyword arguments. For example, you can type:

```
>>> my_h5.make_mass_fractions_movie(['o16','ne20'], 'mass_fracs.mp4', ylim = [1.e-10,
↪1], yscale = 'log', ylabel = 'Mass Fraction')
```

To keep the legend from moving, call with the *legend* keyword. For example, type:

```
>>> my_h5.make_mass_fractions_movie(['o16','ne20'], 'mass_fracs.mp4', ylim = [1.e-10,
↪1], yscale = 'log', ylabel = 'Mass Fraction', legend={'loc': 'lower right'})
```

To use your own title, define a title function as before. For example, to change from seconds to years, type:

```
>>> def my_time_title(props, i):
...     title_str = "time (yr) = %8.2e" % (props['time'][i] / 3.15e7)
...     return title_str
...
```

Next, bind data to the function by typing:

```
>>> zone = ('0','0','0')
>>> props = my_h5.get_zone_properties_in_groups_as_floats( zone, ['time'] )
>>> bind = lambda i: my_time_title(props, i)
```

Now you can call the routine with the title function by typing:

```
>>> my_h5.make_mass_fractions_movie(['o16','ne20'], 'mass_fracs.mp4', ylim = [1.e-10,
↪1], yscale = 'log', ylabel = 'Mass Fraction', legend={'loc': 'lower right'}, title_
↪func=bind, use_latex_names=True)
```

This example also labels the species with superscripts for the species mass number.

You can also plot the zone abundances against a zone property. Since each zone in *my_output1.h5* has a temperature, you can plot against that by typing:

```
>>> my_h5.make_mass_fractions_movie(['o16','ne20'], 'mass_fracs.mp4', property='t9',␣
↪ylim = [1.e-10,1], yscale = 'log', ylabel = 'Mass Fraction', legend={'loc': 'lower␣
↪right'}, title_func=bind, use_latex_names=True, xlim=[0.3,0], xlabel='$T_9$')
```

Notice the *xlim* to get the temperatures oriented correctly with zone index.

As with other movies, you can call the routine with *rcParams* and *plotParams*, as desired.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

w

Index

N

P

R

S

V

W

X