Networking.py Documentation

author: Samir Farooq
company: University of Rochester Medical Center
team: Rochester Center for Health Informatics
email: samir_farooq@urmc.rochester.edu or martin_zand@urmc.rochester.edu
last revised: 18 January 2018

The use of this program requires the following packages: datetime, csv, copy, math, matplotlib, numpy, networkx.

class Networking.**Node**(*date='9999-12-31 23:59:59', primary_field='NPI'*)

A datetime node designed to be an instance of the Networking.**PatientPath** class (e.g. in terms of a patient- a node refers to a single encounter with the health-care system). Note that this is a backend class (i.e. not designed for users) and hence we will not elaborate on the parameters, attributes, and methods beyond their table summaries. Attributes or methods in bold refer to that which the end user may sometimes find useful. Otherwise, they are largely meant for backend purposes only.

Important Definition: A singlenode refers to when a Node only contains one piece of information. A multinode is when multiple tuples refer to the same datetime stamp (i.e. multiple things occur at the same time). For a better understanding of the difference, please see the example under Networking.**Networks** in the `field_properties` parameter (located in the Parameter section).

Parameters:

date : Time stamp in format of: 'YYYY-MM-DD HH:MM:SS'
primary_field : The primary data field for networking tasks (i.e. the key)

Attributes:

`data` : dictionary : Contains the Network fields as its keys, and the corresponding data as its values. If the value is of the dictionary type then the keys of the value dictionary correspond to the data, and the values of the value dictionary correspond to the number of occurrences.
`date` : float : The date (in the form of `matplotlib.dates.date2num`) of occurrence if the information in the node.
`tag` : string : Corresponds to the Network `tag` to which this node belongs. This attribute is particular useful if the `merge_Networks` function is applied.
parent : Reference to a parent node.
child : Reference to a child node.

primary_field : They primary field for performing network tasks (i.e. the key)
sibling : If the node is a multinode, then it keeps track of fields that are paired with the primary key.
up_fields : List of fields which should only be updated if the primary key has never been paired with it before.
sibling_checked : Whether the node was prepared as a multinode or not.
is_class : Whether or not the node should be treated as a classification object.
Class : The classification of the node- default is None.
color : Useful for tracking whether or not the node has been referenced to.

Methods:

`get_field (fieldname=None, string=False)` : If `string` is False, then returns `data[fieldname]`, otherwise: if the field is not a dictionary (i.e. not a multinode) then it returns str(`data[fieldname]`). If the field is a dictionary, then we append together every key in the dictionary, into a list, and we adjust for multiple value entries by including them the amount of times that they are specified. Then we sort the list (to always yield consistent results) and then string together the list (separated by commas) and return the result (with parenthesis on either ends of the string). Note that: if the field has an empty value (i.e. ‘’), then we replace it with ‘UNKNOWN’ and if the field doesn't exist then we replace it with ‘NOT AVAILABLE’ (this is if `string` = True). If the field doesn't exist when `string` = False, we return an empty dictionary.
`is_multinode ()` : The actual truthfulness depends on the definition of a multinode. The truthfulness of this method answers: Two or more primary keys are found in this node? If however your definition of a multinode is that there exists a field with multiple values, then try using `sibling_checked` attribute. Returns bool.
set_color : Set the color attribute.
get_color : Retrieves the color attribute.
get_date : Retrieves the date.
add_field : Given a new fieldname, adds the data corresponding to it into the `data` attribute.
get_data : Returns `data`.
has_parent : Returns bool.
has_child : Returns bool.
has_siblings : Returns bool. Warning: This method has not been updated after some changes to the Node class, and hence may not be in working order. Alternately use `is_multinode`.
get_all_siblings : Returns a list of Node classes which are the siblings of the Node in questions.

add_parent : Updates the parent of the current Node and the child of the new parent to be the current Node.
add_child : Updates the child of the current Node and the parent of the new child to be the current Node.
check_sibling : If the Node has already been prepared for allowing multinode entries, then it does nothing. Otherwise prepares the Node for allowing multinode entries.
create_blank_sibling : Preparation for entering multinode entries.
sibling_decision : Using `field_properties` it makes a decision on how to update the Node data with multinode entries.
add_sibling : Updates the current Node's data, of all fields, with multinode entries.
get_child : Returns the child node.
get_parent : Returns the parent node.
display : prints the datafield.

class Networking.**PatientPath**(*node, primary_field='NPI'*)

A patient path class designed to be an instance of the Networking.**Networks** class. Note that this is a semi-backend class, meaning that while many of the attributes and methods are not designed to be manipulated by backend users, there are some which the backend user may find very useful in their analysis. Attributes or methods in bold refer to that which the end user may sometimes find useful.

Parameters:

node : A class of Networking.**Node** which initiates the patient's path.
primary_field : The primary data field for networking tasks (i.e. the key)

Attributes:

`patient_id` : string : The patient's identification code.
`Class` : hashable object : The class (group) the patient belongs to.
`length` : integer : The number of nodes in the path of the patient.
path_head : Node which initiates the patient's path. All subsequent nodes are children of this node.
primary_field : They primary field for performing network tasks (i.e. the key)
has_multinode : Returns True if a node in the path is a multinode.
stage : This is a counter used during iteration.
cur_node : This is a node tracker used during iteration.
parent_Network : A reference to the Networking.**Networks** class that this patient path belongs to.

Important Methods:

`as_list` (*field=None, string=True, with_dates=False, clean=False*)

Returns the data field of every node as a list object.

Parameters:

`field` : string or None, (default=None)
The field to pull out from the Node's data. If `field` = None, then the `primary_field` is used.

`string` : boolean, (default=True)
If True, then all the data is converted into strings as explained in Networking.**Node**.`get_field`.

`with_dates` : boolean, (default=False)
If True, then all the dates are also returned as a list object, whose indexes correspond to the index of the returned data.

`clean` : boolean, (default=False)
If True, then if a string value of 'NOT AVAILABLE' is encountered (if string=True) then it is not appended to the list. If string=False, then if an empty dictionary is encountered, then it is also not appended to the list.

Returns:

`path` : list
The list of all the data fields, where each index corresponds to a Node's data field.

`dates` : list
This is only returned if `with_dates`=True. Contains a list of all the date times, where each index correspond to that of the `path`.

update_class (*new_class*)

Performs a thorough class update on the patient- meaning it also updates all attributes linked to it's previous and new classes.

Parameters:

`new_class` : string, integer, float, or any hashable object.
The new class (group) that the patient belongs to.

Other Methods:

get_dates () : Returns the dates of each node in the path.
update_len : Updates `length` by one. The call to this also updates the parent Network's `max_len` if this becomes greater than the maximum.
set_patient_id : Set the patient identification code.
get_patient_id : Returns the patient identification code.
get_head : Returns `path_head`.
display : Prints the data field.
set_parent_Network : Set the parent Network that the patient path belongs to.

Properties:

- Can be used in a for loop, doing so iterates through every node.
- Is compatible with the len() function, which if applied, returns the number of nodes in the path.

class Networking.**Networks**(*filepath=None, primary_field='NPI', date='visit_date', p_id='id', tag='NPIpath', lastnames=True, conversion_rules={}, field_properties={'Classification':'up', 'proc':'np', 'provider_name':'up', 'specific_proc':'u'}*)

This class was designed to act as a data multi-functional data structure for holding patient paths, stored in individual Networks.**PatientPath** classes.

Parameters:

`filepath` : string, (default=None)

The filepath to build the patient networks from. The file must be a .csv file with a header. One of the columns must be a date field (explained more in the `date` parameter). If filepath=None then the Network(s) will not be built. This can be done later through the `buildNetworks` method.

`primary_field` : string, (default='NPI')

The primary data field for networking tasks (i.e. the key). This must correspond to the header name found in the file.

`date` : string, (default='visit_date')

This is the column of the csv file which corresponds to the date field. The input must be the same name as the header of the date field. The entries within this column must have a string format of 'YYYY-MM-DD HH:MM:SS'. Note that the '-' and ':' placements are ignored, and hence they can be of any character (e.g. 2016/03/05 12:45.00 would also be accepted). These days will then be transformed into a numerical value (using the package `matplotlib.dates.date2num`, which gives the number of days since '0001-01-00 00:00:00' (e.g. inputting '0001-01-01 00:00:00' would yield a value of 1.0, which is the minimum input date).

`p_id` : string, (default='id')

This is the column of the csv file which corresponds to the patient's identification (id) code. The input must be the same as the header of the field containing the patient ids.

`tag` : string, (default='NPIpath')

Used as an identifier of the Network. It becomes useful for when patient paths are merged with other networks (containing different tags), in order to keep track of which network the node came from.

`lastnames` : boolean or string, (default=True)

This is to identify whether or not to shorten provider names down to only their last names (name simplification). This is useful for visualizations for when the provider names become too long. For instances where there are multiple providers with the same last name, then we add the first letter of the first name (this is determined by the primary key; assuming every name is unique to the `primary_field` that was specified). If there are still non-unique names, further appending of letters is not performed (this is an area that can be updated in future versions). If `lastnames`=False, then this task is ignored. Otherwise `lastnames` must match the header for where the provider names are found. This function only works if the names are in the format: 'LASTNAME, FIRSTNAME (MI)', where middle initial (MI) is optional. If `lastnames`=True, then by default we change the input into `lastnames`='name' (as we make an assumption to the header name). If string input into `lastnames` is not found, then name simplification is ignored. Upon completion of name simplification, a new attribute is created called `idtoname` which is a dictionary; the keys are the unique primary field codes and the values are the simplified names (currently there is a bug with this attribute).

`conversion_rules` : dictionary, (default={})

If some fields need to be treated as integers or floats, then they must be specified using the conversion rules. Otherwise they will be treated as string objects. The key of the `conversion_rules` dictionary is the field name in the header, and its corresponding value is either 'int' or 'float' (to specify the type of conversion).

`field_properties` : dictionary, (default={'Classification':'up', 'proc':'np', 'provider_name':'up', 'specific_proc':'u'})

This dictionary specifies how to handle tuples which occur on the same date: the keys of the dictionary are the headers (aside from the headers specifying the date and patient_id- these are ignored), and their values are one of the following instructions:

- 'u' - Always keep count set to 1 (default for the `primary_field` if not included in `field_properties`).
- 'n' - Always update the count for every tuple (default for every field not specified in `field_properties` with the exception of the `primary_field`).
- 'up' - Only update when the primary_field has changed its value.
- 'np' - Only update while the primary_field is equivalent to the first value it encountered.

Consider the following arbitrary example:

patient, provider, taxonomy, general_procedure, procedure, date
x, P1, 'Pediatrician', 'Lab', 'Blood Draw', '2017-05-04 01:00:00'

x, P1, 'Pediatrician', 'Lab', 'Eye Exam', '2017-05-04 01:00:00'
x, P2, 'Surgeon', 'Lab', 'Blood Draw', '2017-05-04 01:00:00'
x, P2, 'Surgeon', 'Lab', 'Eye Exam', '2017-05-04 01:00:00'
x, P2, 'Surgeon', 'Surgery', 'Ocular Surgery', '2017-08-01 09:00:00'

Assume in this example the primary_field = 'provider'. In this example, patient x has two procedures performed on him on May 04, 2017 at 1am, which are 'Blood Draw' and 'Eye Exam'. Notice first that both are generally considered to be 'Lab' procedures. Second, notice that there are two providers involved in these procedures, provider 1 and 2; a Pediatrician and Surgeon respectively. Since these occur at the same time, there will only be 1 node built for this time-stamp. Now if we naively count the providers, taxonomy, general_procedure, and procedure we would arrive at the counts: {P1:2, P2:2, 'Lab':4, 'Blood Draw':2, 'Eye Exam':2}. However, this count is incorrect, because the patient only saw provider P1 and P2 once in that time stamp, and he only had one 'Blood Draw' and 'Eye Exam', which are two 'Lab's. So to correct for this we would specify that: field_properties = {'provider':'u', 'taxonomy':'up', 'general_procedure':'np', 'procedure':'u'}. This will yield the desired counts: {P1:1, P2:1, 'Lab':2, 'Blood Draw':1, 'Eye Exam':1}. Note that the patient and date fields are ignored from the count because they are assumed to have been specified in the `p_id` and `date` parameters as the patient's id code and the time-stamp, respectively. Note that this multiple instances (tuples) occur for a single datetime stamp, the node which is created from it would be called a multinode.

Attributes:

`Nets` : dictionary : Contains all of the patient information. The keys are the patient id codes, and the values are dictionaries: their keys depends on the user, but one of them is 'Path' which contains the patient's path (an instance of the class Networking.**PatientPath**.
`Plot` : Object of Networking.**Plot_Network** : This attribute does not exist until the `buildNetworks` method is called. Please see Networking.**Plot_Network** for more details.
`classes` : dictionary : The keys are all of the unique classes, the values are the number of patients assigned to that class
`class_colors` : dictionary : The keys are all of the unique classes, the value are the string or rgb colors that have been assigned to that class.
`pat_by_class` : dictionary : This is an alternate version of `Nets` which organizes the patient paths by the class. In other words, the keys to this dictionary are the classes, and the values are dictionaries: The keys to this dictionary are all the patients grouped under the class and their value is an instance of the Networking.**PatientPath** class.
`Overview` : dictionary : Initially this attribute is an empty dictionary,

but contains the counts of the unique values in a specified field (see the method `count_Overview` for more details).
`tag` : string : The tag of the Networking class, this is to differentiate it from other Networking classes which may exist. Also it is useful for when the `merge_Networks` function is applied.
`T_0` : float : This is the minimum date (number of days since '0001-01-00 00:00:00') in the form of days since found after the Networks were built.
`T_final` : float : This is the maximum date (number of days since '0001-01-00 00:00:00') found after the Networks were built.
`max_len` : integer : The maximum patient path that was built upon the building stage.
`max` : dictionary : For every field which was specified for either integer or float conversion, the max (the value) is stored for that field (the key).
`min` : dictionary : For every field which was specified for either integer or float conversion, the min (the value) is stored for that field (the key).
`date` : string : The field (header) name corresponding to the column with the date fields.
`p_id` : string : The field (header) name corresponding to the column with the patient identification code.
`field_propertes` : dictionary : The inputted `field_properties` from the parameters.
`last_update` : string : The datetime stamp in the form of a string when this class was last updated.
`cur_key` : integer : This quantity is used for iterating the patient paths.
`idtoname` : dictionary : Currently has a bug.
`dupnames` : set : Currently has a bug.

Important Methods:

`buildNetworks` (*`filepath, lastnames`*)

Builds the network from the filepath.

Parameters:

`filepath` : string
The filepath from where to build the network. It must be a .csv file.

`lastnames` : boolean or string
See `lastnames` under the parameters section.

`add_classes` (*`determine_class, Net='self', optional_threshold='default', class_field=None, string=False, print_result=True`*)

For every patient, the class of that patient is updated based on the `determine_class` function.

Parameters:

`determine_class` : function
: This is a function which takes as input (L, optional_threshold). L is the list of the field objects, and optional_threshold is any optional parameter which is needed aid the classification. This function is created by the user, and it must be able to handle the case where optional_treshold='default'. This function must then return a class label (any hashable object).

`Net` : Object of Networking.**Networks** or the string 'self', (default='self')
: Decides which Networking.**Networks** object to update the classes from. The parameter L is extracted from each patient in the `Net` (if `Net`='self' then `Net` will extract from itself), where L is the parameter passed into `determine_class`, using the `as_list` method from the patient's Networking.**PatientPath** object. Note that if `Net` is not 'self', then the classes of the Networks object passed into `Net` are not updated, rather only Network object running the `add_classes` method is updated.

`optional_threshold` : Any type, (default='default')
: This is any optional parameter that `determine_class` needs for classification.

`class_field` : string or None, (default=None)
: This is the field from which L should be extracted for the `determine_class` function when calling `as_list`. If `class_field`=None, then it will be set to the `primary_field` of `Net`.

`string` : boolean, (default=False)
: Whether to extract L in the form of a string during the call to `as_list` or not (see `determine_class`).

`print_result` : boolean, (default=True)
: Whether or not to print the amount of patients which were unaccounted for during the updating procedure, and also the amount of patient which did not exist.

`add_classes_from_dict` (*`class_dict`*)

If all the classes of the patients are known, then this method allows to you update the classes for each patient.

Parameters:

`class_dict` : dictionary
The keys are the classes, and the value of the keys are a list of patient identification numbers which correspond to that class.

`count_Overview` ()

For every patient, after the build, it counts all the values of that field and stores the results in the dictionary `Overview`. The key is the patient, the value is a dictionary: the key of this dictionary are the fields, and the values is another dictionary: the key are the unique field values for the patient, and the key is the number of times it occurred.

`apply_conversion_rules` (*`conversion_rules`*)

Apply the conversion rule as explained in the `conversion_rules` parameter.

Parameters:

`conversion_rules` : dictionary
Please see `conversion_rules` under the parameter section.

`class_lists` (*`field=None, Tau=float('inf'), as_dict=True, string=True`*)

Generates a sequence list of values in the specified `field`. This is formatted to be particularly useful in sequence mining tasks.

Parameters:

`field` : string or None, (default=None)
The particular field of interest for generating the sequence list. The field name must correspond to the header name from the original .csv file which the networks were built from. If `field`=None, then the field automatically retrieves results from the `primary_field`.

`Tau` : integer or float, (default=float('inf'))
Sometimes the datetime stamp between two nodes are years apart. In this case one may desire that these two nodes are treated as belonging to a different sequence. The value of Tau allows you to specify how big of a gap (in terms of number of days) should be constituted as a different sequence of events.

`as_dict` : boolean, (default=True)
If `as_dict`=True, then the result is returned as a dictionary where the keys are the classes and the values are a list of sequences. If `as_dict`=False, then the result is returned as a list of sequences (list objects): where the last element of each sequence is the class that it belongs to. Note that the list of sequences are not necessarily ordered by class.

`string` : boolean, (default=True)
When retrieving class field entries, the class field is often of the dictionary type. Specifying `string`=True converts the dictionary into a string by the method corresponding to `get_field` found in the Networking.**Node** class.

Returns:

`cL` : list or dictionary
Either a list of sequences (if as_dict=False) or a dictionary of sequences, where the keys are the classes which the sequences belong to.

`set_class_colors` (*`dict_or_input='input'`*)

Specifies how to color the classes in during plotting tasks.

Parameters:

`dict_or_input` : dictionary or the string 'input', (default='input')
If `dict_or_input`='input' then for every class it will ask you to input a class color in the console window. This way does not support RGB codes, rather only color names in string format will be accepted. Otherwise `dict_or_input` must be a dictionary where the keys are the classes, and the values are the colors corresponding to that class. In this way one can input RGB codes.

`count_occurrence` (*`item, field=None, count_style='both'`*)

Counts the occurrence of an item in all the nodes of every patient path. Note this is very similar to `count_Overview`, but easier to manage if only a few items are desired to be counted. Also this method has the added benefit of counting only from singlenodes or multinodes.

Parameters:

`item` : string, int, or float
The whose number of occurrences are desired to be counted.

`field` : string or None, (default=None)
Specifies to what field this item belongs. If `field`=None, then `field` automatically looks into the `primary_field`.

`count_style` : string, (default='both')
Specifies whether to only count from singlenodes, multinodes, or both (see Networking.**Node** for how we define these Nodes). The possible options are:

- 'singlenode only'
- 'multinode only'
- 'both'

Returns:

`count` : integer or tuple
If the `count_style` is 'singlenode only' or 'multinode only' then returns the count in the form of an integer. Otherwise if 'both' is specified then returns a tuple where the first value is the count of the item in singlenodes and the second value is the count of the item in multinodes. Hence the addition of the two numbers is the total combined occurrence.

`export_as_csv` (*`field=None, round_decimals=4`*)

Writes a csv file based on the specified `field` path of each patient. The format of the csv file is as follows:

patient id 1, days between item1 and `T_0`, item1, days between item2 and item1, item2, days between item3 and item2, item3, ...

patient id 2, days between item1 and `T_0`, item1, days between item2 and item1, item2, days between item3 and item2, item3, ...

patient id 3, days between item1 and `T_0`, item1, days between item2 and item1, item2, days between item3 and item2, item3, ...

...

Parameters:

`field` : string or None, (default=None)
Specifies which field (corresponding to the header name in the original `filepath`) to find the patient path of. If `field`=None then `primary_field` is used.

`round_decimals` : integer or False, (default=4)
If the field is of the float type, then using this parameter will allow you to control up to how many decimal places to round the values to. If `round_decimals`=False then no rounding will take place.

Other Methods

`write_class (filename)` : Writes a csv file where every row is the patient id followed by the class they have been assigned to. The header of the csv file is 'id', 'class'. `filename` determines the name of the csv file (string object which should end in '.csv').
`generate_class_artists (style='rectangle', which='all', fig=None)` : If a legend is needed for the classes (during plotting), then returns a list of Artists and a list of the corresponding classes (the indexes match). `style` can be 'rectangle', 'scatter', or 'line'. Note that if `style` is something other than these three options, then it will be treated as a scatter Artist but with markers which correspond to the input of `style`. Note that if `style` is not 'rectangle' or 'line' then the figure for plotting must be specified into `fig`, although no changes to the figure will actually take place. `which` must be either a list of the classes which the artists are desired from, or 'all' which will use `classes`.keys().
`get_max (field=None)` : Returns the max value of the specified field. This field must have been converted into float or integer using `apply_conversion_rules` (this is run automatically if `filepath` was specified upon creation of the data structure, otherwise it must be run manually).
`get_min (field=None)` : Returns the min value of the specified field. This field must have been converted into float or integer using `apply_conversion_rules` (this is run automatically if `filepath` was specified upon creation of the data structure, otherwise it must be run manually).
`add_class_nodes (filepath)` : From a csv file, interweave new Nodes into the patient Paths. These new nodes will be labeled as class nodes (see `Class` and `is_class` under the attribute section of Networking.**Node**). The header information in the csv file must have matching `p_id` and `date` for the patient's id and datetime stamp respectively. **NOTE:** This method has not been thoroughly tested.
display_paths : Prints the paths of each patient. **NOTE:** This method has not been thoroughly tested.
get_max_len : Returns the path length (in terms of number of Nodes; not in terms of number of days) which is a maximum of all the patient paths.
set_tag : Input the tag to set the `tag` to. See `tag` attribute for details.
makeNode : This is a backend method which creates a new node from the csv row (tuple).

Properties

- Can be used in a for loop, doing so iterates through every patient's path (Networking.**PatientPath** objects).
- Can be indexed, where the index is the patient's identification code and the output is the PatientPath object for that patient.

class Networking.**Plot_Network**(*Network*)

This class was designed to work in conjuction with the Networking.**Networks** class (this class is referenced to by the Networks class, after running `buildNetworks`, under the attribute name `Plot`). The objective of this class is the allow networking plotting of the path of each Patient. Currently an in-depth documentation is not available for this class (please wait for future updates), but for those wishing to make use of it should first run one of the attributes: `get_isomorphic_edges` or `get_patient_edges` (depending on the desired network graph), following by running one of: `myplot` or `netplot`. `myplot` uses self-developed plotting techniques, while `netplot` uses the **networkx** package. Note that the use of `netplot` has not been thoroughly tested.

Useful Functions:

While this is not an exhaustive list of all of the functions within the Networking.py file, it is a comprehensive list of all the functions believed to be most beneficial to the end-user.

Networking.**merge_Networks**(*parent_Network, other_Networks=[], tag_colors=[]*)

> Merges the Networks objects in the list of `other_Networks` into the `parent_Network` (which must also be an object of the class Networking.**Networks**). The `tag_colors` correspond to how the nodes of the Networks objects should be colored, where the `tag_colors` is a list: the first index corresponds to the color for the `parent_Network` and all subsequent indexes are in the order of how to color the Networks in `other_Networks`. The tag colors are then added in the form of a dictionary as an attribute of `parent_Network` named `tag_colors`. This function then proceeds to interweave all of the nodes from each Networks in `other_Networks` into their corresponding places (in regards to the datetime stamp) in the `parent_Network`. Patients which exist in a Networks from `other_Networks` but do not exist in the `parent_Network` are ignored, and vice versa. If some Nodes happen to fall on the same datetime stamp, then the tag of that Node is updated to be a set containing each tag corresponding to the types of Networks which have been added into the `data` attribute of the Node. Furthermore, the path lengths are also updated after the merge. This function does not return anything, but the `parent_Network` is updated (God willing).

Networking.**class_transfer**(*Info_Network, Updating_Network*)

> For every patient in the `Info_Network`, we update the corresponding patient in the `Updating_Network` based on the class assigned to the patient in the `Info_Network`. Both of these parameters must be an object of the type Networking.**Networks**. This function does not return anything, but the classes in `Info_Network` are all updated (God willing).

Networking.**class_combine**(*Network, parent_class, child_classes*)

> All classes mentioned in `child_classes` are renamed to be called the `parent_class` in the `Network` which must be an object of the type Networking.**Networks**. Note `parent_class` must be a hashable object and `child_classes` can either be a single existing class, or a list of existing classes which need to be changed. This function returns the updated Network (God willing).

Networking.**rename_classes**(*Net, old_to_new_dict*)

> Changes the class labels in the Networks object (`Net`) based on the `old_to_new_dict` which is a dictionary whose keys are the names of the old classes and whose values are the names of the new class. It then outputs the Network with the new class names (God willing).